Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)

UniversitatPolitècnica de València

fSO

Lab session 4
Process creation in UNIX

Content

1. Objectives

Process creation is one of the most important services that multiprogrammed operating systems provide. In this lab session, we study the system calls provided by UNIX to create processes. The lab session objectives are:

- Acquiring skills using POSIX calls: `fork()`, `exec()`, `exit()` and `wait()`.
- Using program examples to analyze the former system calls in terms of **returning values** and **attributes inherited.**
- Viewing the different **process states: running (R), suspended (S)** and **zombie (Z)**

In this practice, we work with concepts taught in Seminar 3.

2. Introduction

UNIX uses a cloning mechanism for the creation of new processes. It creates a new process control block (PCB) structure that gives supports to clone the process that call to **fork()**. Both processes, created (child) and creator (parent) are equal at the creation moment, but they have **different PID and PPID identifiers**. The **child process inherits** a copy of file **descriptors, code and variables from the parent process**. Each process (parent and child) has its own memory space, so child process variables are allocated on different memory positions to those of parent's, so when a process modify a variable, this change is not reflected on the other process. In summary, we can say that a call to **fork()** creates a new "child" process that:

- It is an exact copy of its creator (parent) except in the PID and PPID values.
- It inherits the parent process variables and evolves independently from the parent.
- It inherits the file descriptor table of the parent process and it can share files with the parent.

exec() call changes the memory map of a process, i.e., "what it will run" and its variables. Calling to **exec()** changes the contents of the process memory map by loading new data, code, etc., and putting the process program counter pointing to the first code address. The code is loaded from a file on disk that contains executable code, specified as an **exec()** parameter. Decoding such an executable file (complying with ELF format in Linux), the operating system creates the corresponding memory map regions with code and initialized data.

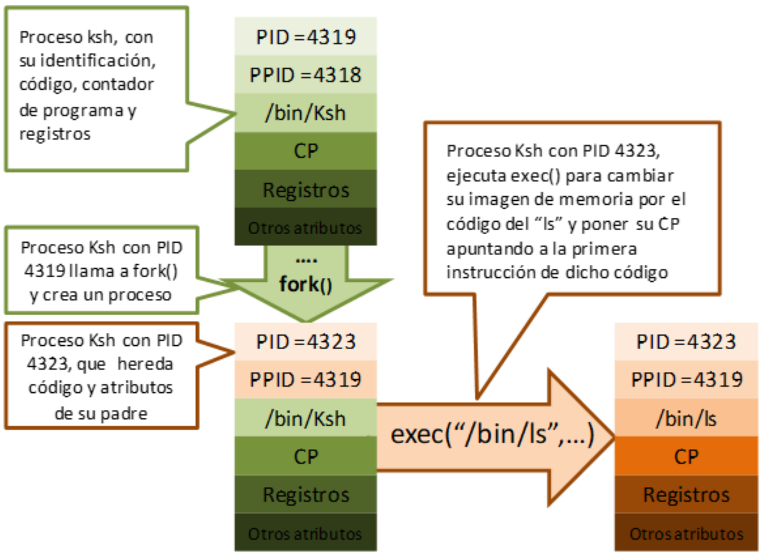

Proceso ksh, con su identificación, código, contador de programa y registros
PID =4319
PPID =4318
/bin/Ksh
CP
Registros
Otros atributos
Proceso Ksh con PID 4319 llama a fork() y crea un proceso
....
fork()
Proceso Ksh con PID 4323, que hereda código y atributos de su padre
PID =4323
PPID =4319
/bin/Ksh
CP
Registros
Otros atributos
Proceso Ksh con PID 4323, ejecuta exec() para cambiar su imagen de memoria por el código del "ls" y poner su CP apuntando a la primera instrucción de dicho código
exec("/bin/ls",...)
PID =4323
PPID =4319
/bin/ls
CP
Registros
Otros atributos

Figure-1: fork() and exec() calls

The creation of processes within the system generates a tree of parentship among them that links the processes in terms of synchronization. A typical application case of synchronization between parent and children is the Shell.

The Unix Shell is a process that creates a child process for every command invoked from the terminal. The Shell process waits for the process executing the command to complete and prints the *prompt* on the screen (except when running the command in background) and then it waits again for a new command. This synchronization of the Shell with its children is made via the calls **wait()** or **waitpid()**, and **exit()**.

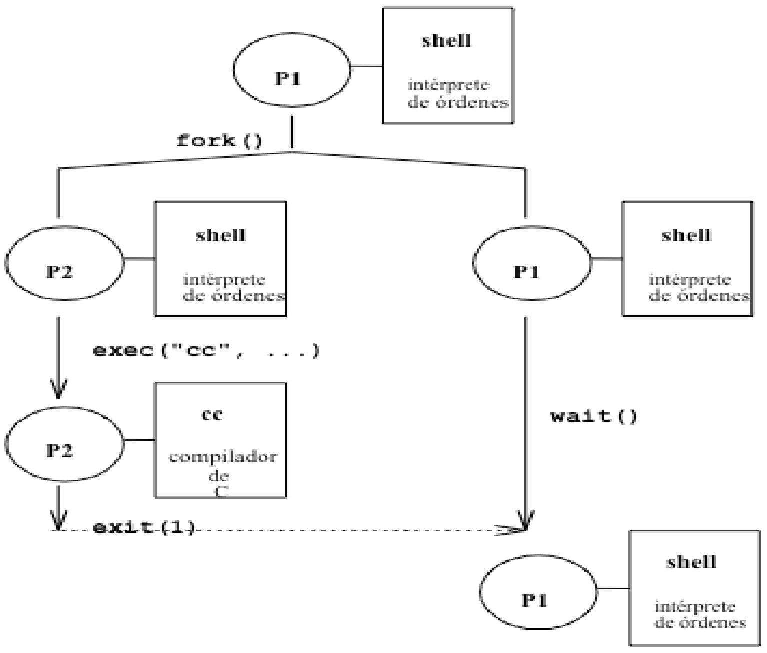


Figure-2: UNIX Shell

In the Annex, **exit()** and **wait()** calls are fully described. **exit()** terminates the execution of a process and **wait()** suspends the execution of the process until one of its children processes ends. You can also use the system manual for getting details on these calls, i.e.:

```
$ man fork
```

3. Process creation in UNIX

UNIX creates new processes with system call **fork().**

Using command mkdir create a new directory named fso_pract4 to put all your lab files:

```
$ mkdir fso_pract4
$ cd fso_pract4
```

Then download the source code files available on folder “src_students”.

Exercise 1: Creating a child process with fork() “my_child.c”

```
/* my_child.c */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    printf("Parent process %ld \n", (long)getpid());
    fork();
    printf("I am %ld process, my parent is %ld\n",
           (long)getpid(),(long)getppid());
    sleep(15);
    return 0;
}
```

Figure-3: my_child.c code

Compile and run *my_child.c* in the background. Fill in table 1 with the PID, PPID and command of the running created processes.

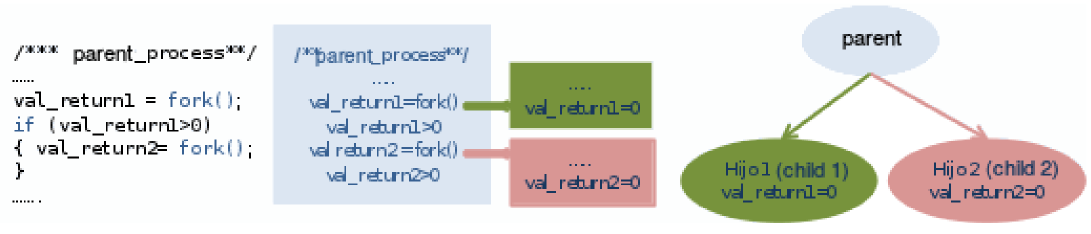
```
$ gcc my_child.c -o my_child
$ ./my_child&
$ ps -la
```

	PID	PPID	COMMAND
Parent process			
Child process			

Table-1: Exercise-1, creating processes with **fork()**

4. Returning values in `fork()`

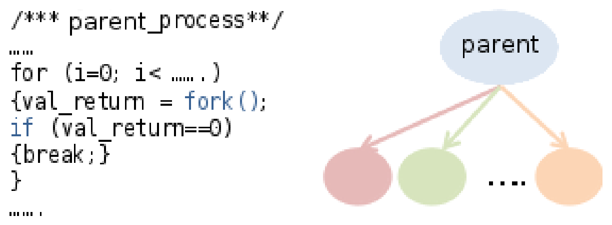
The returning value of **fork()** allows to identify in the code the parent and child processes and decide what code each one executes after the **fork()** call. In case of success, **fork()** returns 0 to the child and a positive value (the child's PID) to the parent. Figure-4 describes the situation:



```
/*** parent_process**/
......
val_return1 = fork();
if (val_return1>0)
{ val_return2= fork();
}
...... .
```

Figure-4: From left to right: C code that allows to differentiate parent and child relying on the value returned by fork(), and the generated processes.

A common way of generating children processes is as a fan. Figure-5 shows the C code to get a process tree in fan.



```
/*** parent_process**/
......
for (i=0; i< ......)
{val_return = fork();
if (val_return==0)
{break;}
}
...... .
```

Figure-5: Processes created in fan, scheme code and parentship structure tree.

Exercise 2: Returning value from `fork()` "range_process.c"

Create a file named *range_process.c* and type in the code needed to create five children processes in fan. Add system calls and sentences in order to:

- Create five processes in fan by iterating in a **for** loop.
- Each child does `"printf("child created in iteration = % d", i)"` and finishes its execution with `exit(i)`.
- The parent process performs a `sleep(10)` after creating the children and then calls to `exit()`.

Compile the program and run it in background and then do **`ps -l`**. Fill table – 2 with the information.

	PID	PPID	COMMAND	STATE
Parent process				
Child process 1				
Child process 2				
Child process 3				
Child process 4				
Child process 5				

Table-2: Exercise-2, creating processes in fan

Question-1: Why are child processes generated by *range_process.c* in the observed state?

5. The Init process

In UNIX systems, *init* (*initialization*) is the first process that starts running after loading the kernel and so it is the process tree root. All processes in the system, except process *swapper*, descend from process *init*. Init runs as a daemon and typically has PID 1.

At startup, UNIX performs a sequence of steps known as *bootstrap* to load a copy of the operating system in main memory and to start its processes. Once loaded, the control is transferred to the kernel starting address, and it begins to run. The kernel initializes internal data structures, mounts the main file system and creates the process with PID 0, named *swapper* process.

Process 0 creates process *init* calling to **fork()**. Process *init* reads /etc/ttytab file and creates a login process for every terminal connected to the system. Once a user enters a correct login and password (/etc/passwd file contains login info for local users), the login process launches a command interpreter or Shell.

Thereafter the responsible for addressing the user in this terminal is the Shell interpreter, while the login process falls asleep until a logout happens.

Note. Daemon processes or daemons do system functions running in user mode.

Exercise 3: Processes adopted by INIT "adopted_process.c"

We will check that *init* adopts those processes which parent process has ended without waiting for them. To do this we have to ensure that children remain active after their parent finishes.

Copy the file *range_process.c* in *adopted_process.c* using "cp" command, as follows:

```
$ cp range_process.c adopted_process.c
```

Edit *adopted_process.c* and add a **sleep(20)** sentence before **exit(i)** for every child. Make sure that children are suspended in **sleep()** for more time than the parent. Compile *adopted_process.c* and run it in the background, then run **ps -l** several times until all processes end. Look in the PPID of the children at the beginning and at the end of all processes and record them in Table-3.

	PID	Start PPID	EndPPID	COMMAND	STATE
Parentprocess					
Childprocess 1					
Childprocess 2					
Childprocess 3					
Childprocess 4					
Childprocess 5					

Table-3: Exercise-3, *adopted_process*

Question 2: In processes created with *adopted_process,* which process ends first and how it affects to the other processes?

6. Process waiting

It is necessary to avoid zombie processes since they can overload the system. A process passes to zombie state when it calls to **exit()** and its parent has not yet called to **wait()**. To avoid zombie processes, a parent process should always wait to its children by calling to **wait()** or **waitpid()** before calling to **exit()**.

wait() call is blocking and suspends the process until one of its children ends. You can find **wait()** detailed description in the annex.

Exercise 4: The parent should wait "parent_wait.c"

In this exercise, we will modify program *range_process.c* so it no longer generates zombie processes when running it. To do so, copy file *range_process.c* into another file named *parent_wait.c*:

```
$ cp range_process.c parent_wait.c
```

Edit *parent_wait.c* adding instructions and the required system calls so the parent process waits for their children. Parent process must wait for all its children. Remember that you have to compile and run with:

```
$ gcc parent_wait.c -o parent_wait
$ ./parent_wait &
$ ps -la
```

Question 3: Are there zombie processes created when running *parent_wait*? Justify your answer.

Exercise 5: Communicating completion status to the parent "final_state.c"

exit() call allows a child to pass information to its parent about its completion state. The value passed to **exit()** is received on **wait(int *stat_loc)** throug "stat_loc". To do the exercise copy *parent_wait.c* file to *final_state.c*

```
$ cp parent_wait.c final_state.c
```

Modify *final_state.c* with the **main()** code shown in Figure-6:

- Using a **for** loop and variable "i", create four processes that display a message stating the iteration "i" in which they are created.
- All processes must wait 10 seconds before calling to `exit()`.
- All parents must wait for the completion of their children with `wait(&stat_loc)`, and print their PID and the child exit status value got on parameter "stat_loc".

  Note: The maximum value that can be passed to exit is 255 (8 bits), WEXITSTATUS macro applied to the wait parameter gives the value passed to exit.

```
int i;
pid_t val_return;
int final_state;
  for (i=0; i<MAX_PROC; i++) {
    val_return = fork();
    if (val_return == 0) {
      printf("Child %ld created in iteration %d \n",
      (long)getpid(),i);
    } else {
      printf("Parent %ld, iteration %d \n", (long)getpid(), i);
      printf("I have created a child %ld \n", (long)val_return);
      break;
    }
  }
  while (wait(&final_state) > 0) {
    printf("Parent %ld iteration %d \n", (long)getpid(), i);
    printf("My child said %d \n", WEXITSTATUS(final_state));
    printf("My child said %d \n", final_state/256);
  }
  exit(i);
```

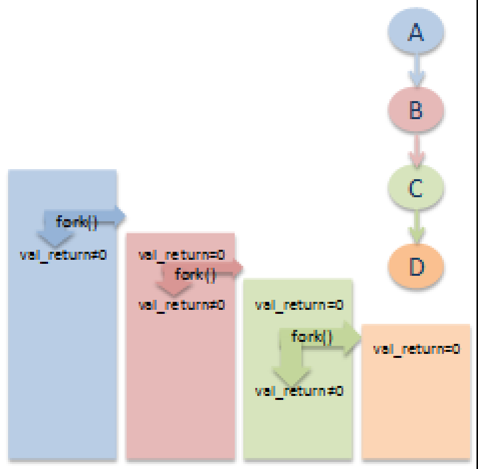


Figure-6: Processes in chain

Compile *final_state.c* and run it, use the ps command and fill in the following table:

	PID	Exitvalue
Parent process		
Process created with i = 0		
Process created with i = 1		
Process created with i = 2		
Process created with i = 3		

Table-4: Exercise-5

Question 4: What is the first statement that run the children processes created? How many loop iterations executes each process?

7. System call exec()

exec() changes the memory map of the process that calls it. Six variants for this call are detailed in the annex. Figure-7 illustrates the combined use of **fork()** and **exec()** to create a new process running **ls -l** command.

```
/* change_memory 1 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    pid_t childpid;
    int status, x;
    char* arguments [] = { "ls", "-R", "/", 0 };

    chilpid = fork();
    if ( childpid == -1) {
        printf("fork failed \n");
        exit(1);
    } else if (childpid == 0) {
        if(execl("/bin/ls", "ls", "-l", NULL) <0 ) {
             printf("Could not execute: ls \n");
            exit(1);
        }
    }
    x = wait(&status);
    if ( x!= childpid)
         printf("Child has been interrupted by a signal \n");
    exit(0);
}
```

Figure-7: Code to create a new process running **ls** command.

Exercise 6: Creating a process to run `ls` command

Without running code in Figure-7, indicate which process/es (parent, child or both) call the **printf()** instances in the code and if they are executed only when a certain condition is met.

	Parent: what condition?	Child: what condition?
"fork failed\n"		
"Could not execute: ls\n"		
"Child has been interrupted by a signal\n"		

Table-5: Exercise-6

Exercise 7: Changing the memory map of a process "change_memory1.c"

File *change_memory1.c* has the code shown in Figure-7, where the child executes the **ls –l** command. Use the **execvp()** variant to make the child run the following command:

```
$ ls -R /
```

While *change_memory1.c* is running, open another console, run **ps -la** command, and write in Table-6 the results:

	PID	PPID	COMMAND
Parent process			
Child process			

Table-6: Exercise-7

Question 5: What is the main difference between the results in this table and those obtained in exercise 1?

Exercise 8: Calling to `execl()` in "change_memory1.c"

Modify *change_memory1.c* using system call **`execl()`** where you have to specify **ls** options as parameters, like:

```
execl("/bin/ls","ls","-R","/",NULL);
```

Exercise 9: Passing arguments in "change_memory2.c"

To increase program *change_memory1.c* functionality and to be able to run command **ls** with different arguments that are passed to the program in the command line, we are going to use the two arguments from **main(int argc, char * argv[])**. As we saw in the C programming lab session, parameter *argc* is an integer that is equal to the number of command line arguments. At least there is always one argument, argv[0], that is the program name. The parameter *argv[]* is an array of pointers to string (char arrays) that are the arguments passed in the command line.

From *change_memory1.c* create *change_memory2.c* to execute the commands passed as arguments, use variables *argc* and *argv*.

Check with the following executions:

```
$./change_memory2 ps
$./change_memory2 ps -l
$./change_memory2 ls -lh
$./change_memory2 ls -lht
```

Optional exercise

Figure-8 shows a code that performs the sum of all the elements of a matrix with dimensions NUMROWS and DIMROW. There are three loops: the first one initializes the array values row by row, the second one obtains for every row the sum of all its elements and the last one performs the final sum adding the results got for every row.

```
/* addrows.c */

#include <stdio.h>
#include <math.h>
#define DIMROW 100
#define NUMROWS 20

typedef struct row {
  int vector [DIMROW];
  long add;
} row;

matrix row [NUMROWS];
int main() {

  int i, j, k;
  long total_add = 0 ;
  // Initializing to 1 all array elements
  for (i = 0; i < NUMROWS; i++) {
    for (j = 0; j < DIMROW; j++) {
      matrix [i][j] .vector = 1 ;
      array[i].add = 0 ;
    }
  }
  for (i = 0; i < NUMROWS; i++) {
    for (k = 0; k < DIMROW; k++) {
      array[i].add += array[i][k].vector;
    }
  }
  for (i =0; i< NUMROWS; i ++) {
    total_add += array[i].add;
  }
  printf ("The total addition is %ld\n", total_add);
}
```

Figure-8: Code of "addrows.c" to add the rows of a matrix

Compile and run "addrows.c". To run it, use the command **time** that allows determining the time that it takes to run a program:

```
$ time ./addrows
```

Write the time returned by the former command and the result of total_add.

Create a program called *addrowspro.c* where you create as many processes (children) as rows. Each process must compute the sum of one row (`matrix[i].sum`) and end by calling to **`exit(matrix[i].sum)`**. The parent process must wait for all children, accumulate in the `total_sum` variable the values got from **wait()**, and finally print that value, which has to be the same value got with *addrows.c*.

Run the program with command **time** and compare it with the previous version.

Note. The mechanism of passing a value from **exit()** to **wait()** is intended to allow children informing their parent about their ending status and not to pass a computation result. Therefore, the use of **exit()** made in the previous exercise lies outside of the proper use of the fore mentioned mechanism. It is used here in that way due to the lack of another interprocess communication mechanism.

8. Annex

8.1 System calls syntax

A system call is invoked through a function always returns a value with the information about the service provided or the error that has occurred in their execution. This return can be ignored, but it is recommended to test it.

Calls return values:

- Return a value of - 1 or pointer to NULL when it occurs error in the execution of the call
  - There is an external variable *errno,* which indicates a code on the type of error that has occurred. In the file of *errno.h* (#include <errno.h>) there are references to this variable declarations.
- The variable *errno* does not change value if a system call returns successfully, it is therefore important to take such fair value after the call to the system and only when these return error, that is, when the call returns - 1.

8.2 fork()

```
#include <sys/types.h>

pid_t fork (void)
```

- Description:
  - A process created child which is a "clone" of the parent: inherits much of their attributes.
  - Inheritable attributes: all except PID, PPID, pending signals, times/accounting.
- Return value:
  - 0 the child
  - PID of the child to the parent
  - (- 1) the parent if error.
- Errors
  - Error occurs when there are insufficient resources to create the process

8.3 exec()

There are 6 variants of the exec() call, that differ in the way that are passed arguments (l = list or v = array) and the environment (e), and if necessary provide the path and name of the executable file.

```
#include <unistd.h>

void execl (const char* path, char const* arg0,..., const char* argn, (char *) 0);

void execle(const char* path, const char*arg0,...,const char*argn,(char*) 0, char* const envp []);

void execlp(const char* file, const char*arg0,...,const char*argn,(char*) 0);

void execv (const char* path, char* const argv[]);

void execve(const char* path, char* const argv[], char* const envp[]);

void execvp(const char* file, char* const argv[]);
```

Calls execl (execl, execlp, execle) pass the arguments in a list and are useful if the arguments are known at compile time.

Examples of execl:

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

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

Calls execv (execv, execve, execvp) pass the arguments in an array.

Example of execv:

```
execvp(argv[1], &argv[1]);
```

- Description of `exec()`:
  - Change a process memory image by defined in an executable file.
  - Executable file can express its naming file or its path full path.
  - Some attributes of the process are preserved and, in particular:
    - Signal handling, except the captured signals whose action will be default.
    - PID and PPID, given that a new process is created not only changes its image
    - CPU (accounting) times
    - File descriptors
    - Directory of the root directory, the file creation mode mask,
  - If the executable SETUID bit is enabled, exec puts as effective UID of the process to the UID of the owner of the executable file.
    - The same happens with the SETGID bit set
- Errors:
  - Non-existent or non-executable file
  - Permissions
  - Incorrect arguments
  - Memory or insufficient resources
- Return value:
  - (-1) if exec() returns to the program that called it which indicates that an error has occurred.

- Description of parameters:
  - path: name of the executable file has to indicate his career. Example: "/ bin/cp".
  - file: name of the executable file is uses the PATH environment variable to locate the.
  - arg0: first argument corresponds to the name of the program without the path. Example: "cp"
  - arg1... argN: parameters set that receives the program for execution.
  - argv: array of pointers to strings. These strings are the list of arguments available for the new program. The last of pointers must be NULL. This array contains at least one element in argv [0], name of the program.
  - envp: array of pointers to strings, constitute the framework for the implementation of the new program.

8.4 exit()

```
#include<stdlib.h>

voidexit(int status)
```

- Description:
  - Terminates the execution of the process that invokes "normally".
  - If it is not explicitly invoked, is made implicitly at the end of entire process.
  - The State of completion *status* is transferred to the parent who runs *wait(&status)*.
  - If the parent of the process is not running *wait*, is transformed into a zombie.
  - When a process executes *exit*, all of their children are adopted by init and the PPID are renumbered 1.
- Returnvalue:
  - None
- Errors:
  - None

8.5 wait()

```
#include<sys/types.h>

#include<sys/wait.h>



pid_t wait(int*stat_loc)

pid_twaitpid(pid_tpid, int* stat_loc, int options)
```

- Description:
  - Suspends the execution of the process that invoked it, until one of the children (wait) or a child in particular (waitpid) ends.
  - If there is a zombie child, wait ends immediately, but stops.
  - Whenstat_locis not the null pointer, contains:

If **child ends with exit** :
MSB: status defined by exit, LSB: 0
If **child ends up signal** :

MSB: 0, LSB: number of signal signal (bit more high 1: core dump))

- Return value:
  - The PID of the child that is complete except:
  - (- 1): a signal or error (there are no children).
- Errors
  - The process has no children
- Waitpid(2) description
  - ArgumentPID
    - PID < - 1 wait for any child whose process group is equal to pid
    - PID = - 1: wait for any child (like wait)
    - PID = 0: wait for any child with the same group of processes
    - PID > 0: wait for the child whose pid is indicated
  - ArgumentOptions
    - WNOHANG: return immediately if the child has not completed

The completion of the with `exit(status)` allows you to store the State of completion in the address pointed to by stat_loc parameter. POSIX standard defines macros to analyze the return the process status child:

- WIFEXITED, normal termination.
- WIFSIGNALED, termination for not caught signal.
- WTERMSIG, termination by SIGTERM signal.
- WIFSTOPPED, the process is stopped.
- WSTOPSIG, termination by SIGSTOP signal (which cannot be caught or ignored).