Process creation with call Fork

Operative Systems

Alejandro Aguilar, Leopoldo Pla,
Álvaro Sánchez

Call FORK

int fork(*void*)

The creation of a process in Unix based Operative Systems is done by the call **fork()**.

The call fork is used when we want to create a child process from a parent process. It seems very obvious that the process that **makes the call** is the **parent**, and the process **resultant of the call** is the **child**.

¿How do we use the fork() call?

fork();

fork() returns two values (one for each branch of execution):
- In the parent's branch, it returns the **PID** value of **the child**.
- In the child's branch, it returns **ZERO** (this is very useful to determine what code is executed by the children and by the parent).

The call divides the execution of the program in two branches. In the first, the father is executing the lines of code normally, and in the second, this is done by the child.

The child is a clone of the father, with the same variables and values (except the shared memory, and a unique unique **PID** (Process IDentification, some like the process's DNI) .

Call FORK

int fork(*void*)

This is a example of code using **fork()**:

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

int main()
{
    pid_t pid; // Declaration of the variable that will stores the PID value of the fork

    pid = fork();

    switch(pid)
    {
    case -1: // Error case
        printf("I'm sorry but you've had an abortion sir\n");
        break;
    case 0: // Child case
        printf("I am Luke Skywalker, my PID is %d and my PPID is %d\n",getpid(),getppid());
        break;
    default: // Parent case
        printf("Luke, I am your father. My PID is %d and your PID is %d\n",getpid(),pid);
    }

    exit (0);
}
```

int getpid(*void*)
Returns the value of the process's PID

int getppid(*void*)
Returns the value of the parent's PID

Execution check

If we want to make sure that the program is running correctly we can execute it in **background mode** and verify its execution with the order **ps**.

To do this, we have to put a '&' after the program's execution command. An example of what will be shown on the terminal would be like this:

```
$ partyhard &
I am the hard, my PID is 944 and my PPID is 943
I am 944 and I die
I am the party, my PID is 943 and the PID of my son is 944
I am 943 and I die
$ ps
PID   TTY          TIME   CMD
893   pts/1    00:00:00   bash
943   pts/1    00:00:00   creaproc
944   pts/1    00:00:00   creaproc
946   pts/1    00:00:00   ps
$
```

Process Termination

Call WAIT

int **wait(**int state**)** or int **wait(**NULL**)**

The **wait()** call makes a process stop until it recieves a signal (it can be sent by the **exit() call** or not) or one of its children dies.

The call returns the PID of the dead child in case of success, and -1 in case of error.

Call EXIT

void **exit(**int state**)**

The **exit(**int state**)** call makes a process end with the state "state". It sends a signal to the parent process, and makes it stop waiting.

Process Termination

This is a simple example of how the calls **exit()** and **wait()** work:

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

int main()
{
   int state, number;

   switch(fork())
   {
   case -1: // Error case
      perror("Fork error");    exit(1);
   case 0: // Child case
      number = 13;
      printf("I am the child and my killer is %d...\n",numero);    sleep(15);    exit(number);
   default: // Parent case
      wait(&state);
      printf("I am the parent, and ");
      if((state & 0x7F) != 0) { printf("My child died with a signal\n"); }
      else { printf("The killer of my child has been exit(%d).\n",(state>>8) & 0xFF); }
      exit(0);
   }
}
```

Signals

```
#include <signal.h>
```

Signals are methods of communication between fathers and childrens. Thanks to that, the father can order the soon to do some especific process, or even kill him.

To use a signal we have to declare a handler first. A handler will contain the steps that the signal is transfering to the process.

There are a lot of different signals that we can use but we will talk about the most used ones.

SIGKILL: This is a signal with a clear objective, it will kill the process.

SIGALRM: This is a signal which simulates, more or less, the command "sleep". We can program a alarm with this signal and it will stop the program there until the alarms activated.

KILL: This signal allows a process to send something to another process or even himself. If is empty, this signal will kill the process. If not, the process will do which it says.

SIGALRM

void alarma()

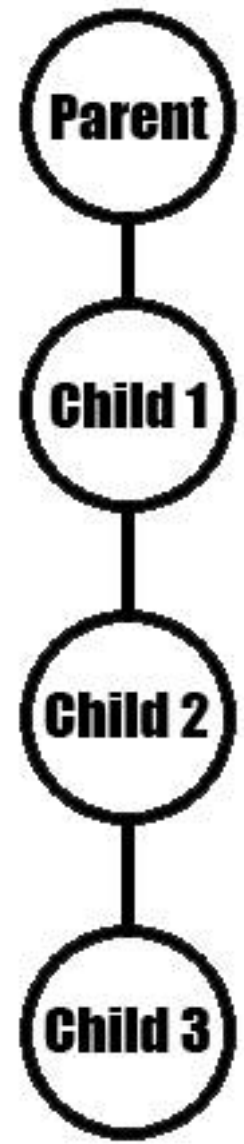
```c
#include <stdlib.h>
#include <signal.h>

void alarma(); // Captures the signal SIGALRM

int main()
{
   signal(SIGALRM,alarma); // we declare the alarm

   printf("Ejemplo de signal\n");

   alarm(15); // we wait 15 secs until the alarm activated

   printf("Fin\n");

   exit(0);
}

void alarma()
{
   signal(SIGALRM,SIG_IGN);

   printf("¡Alarma!\n");
}
```

KILL

```
#include <signal.h>

main()
{
    printf("I am going to kill myself\n");

    kill(getpid(),SIGKILL); //now Kill will try to end himself

    perror("I'm still alive"); // error
}
```

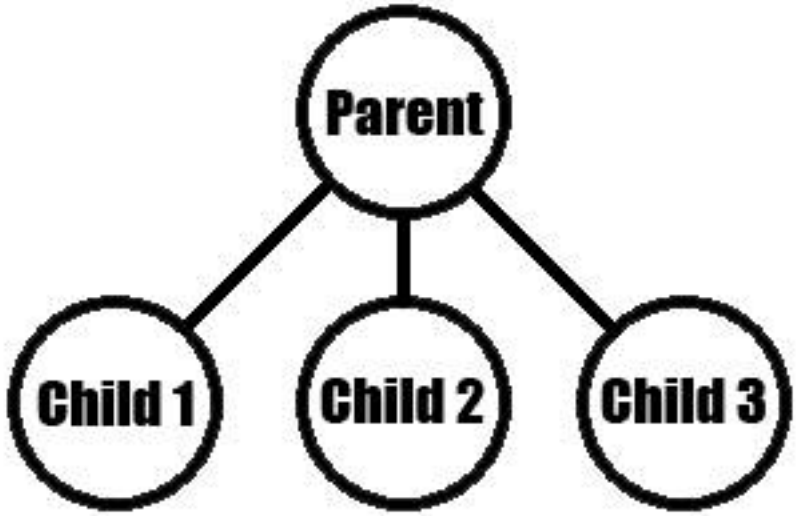
Vertical Process Creation

```
int main()
{
    int i;
    int e;

    for(i = 0;i < 3;i++)
    {
        if(fork()==0)
        { printf("My pid: %d --- My ppid: %d\n",getpid(),getppid()); }
        else
        {
            wait(&e);
            exit(0);
        }
    }
}
```



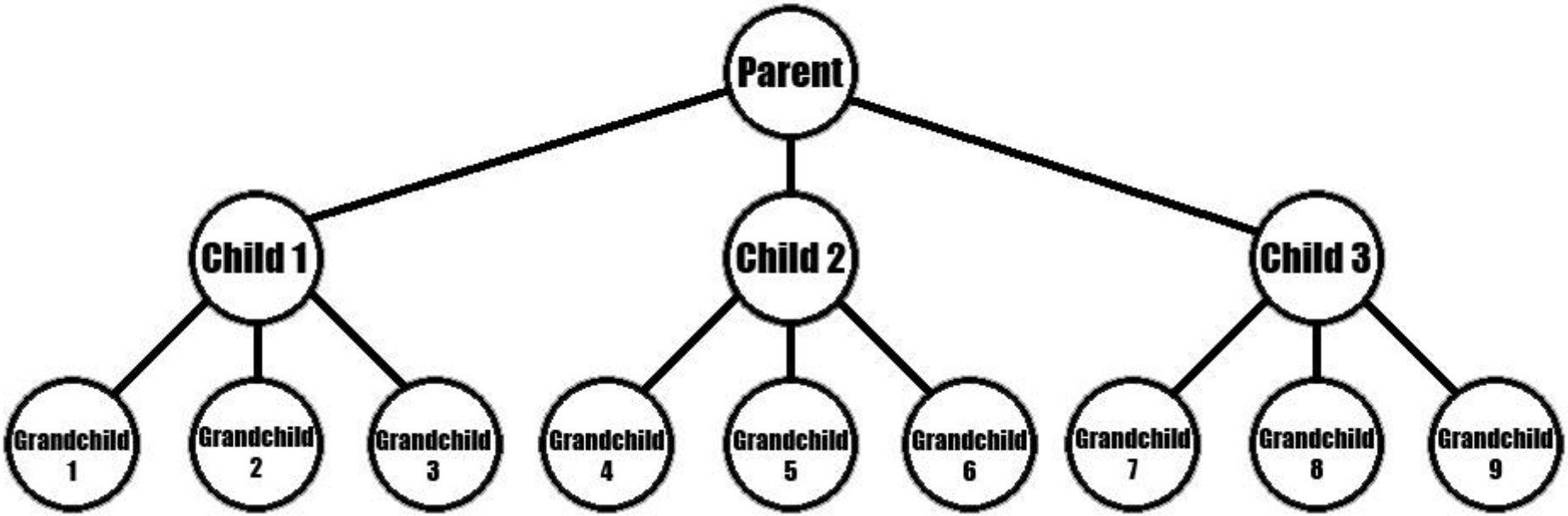
Horizontal Process Creation

```
int main()
{
    int i;
    int e;

    for(i = 0;i < 3;i++)
    {
        if(fork()==0)
        {
            printf("My pid: %d --- My ppid: %d\n",getpid(),getppid());
            exit(0);
        }
        else
        { wait(&e); }
    }
}
```



Example 1

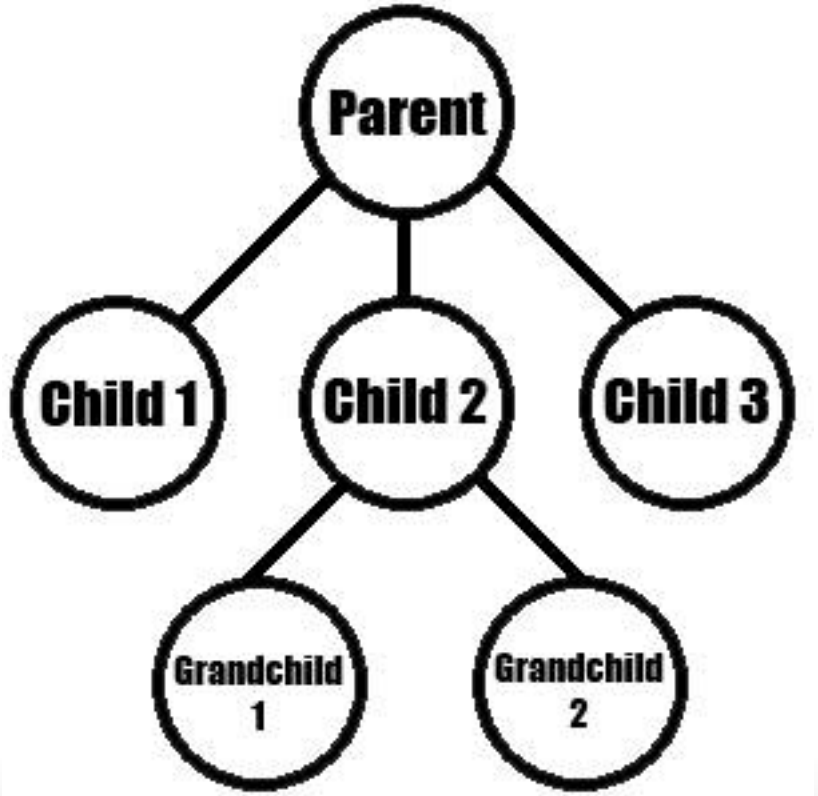
```
int main()
{
    int i, pid, j, e;
    int f1 = 0;
    int f2 = 0;

    for(j = 0;j < 3;j++)
    {
        pid = fork();
        if(pid == 0)
        {
            for(i = 0; i< 3; i++)
            {
                pid = fork();
                if(pid == 0) {
                    printf("My pid: %d --- My ppid: %d\n", getpid(), getppid());
                    exit(0);
                }
                else
                    wait(&e);
            }
             printf("My pid: %d --- My ppid: %d\n", getpid(), getppid());
             exit(0);
        }
        else
            wait(&e);
    }
}
```



Process Tree generated by the Example 1

Example 2

```
int main()
{
	int i;
	int pid;
	int e;
	int pid1, pid2;

	for(i = 0;i < 3; i++)
	{
		pid = fork();
		if(pid == 0)
		{
			printf("My pid: %d --- My ppid: %d\n", getpid(), getppid());
			if(i == 1)
			{
				pid1 = fork();
				if(pid1 == 0) { printf("My pid: %d --- My ppid: %d\n", getpid(), getppid());	exit(0); }
				else { wait(&e); }
				pid2 = fork();
				if(pid2 == 0) { printf("My pid: %d --- My ppid: %d\n", getpid(), getppid());	exit(0); }
				else { wait(&e); }
			}
			exit(0);
		}
		else { wait(&e); }
	}
}
```



Process Tree generated by the Example 2