



## TD N°2 : Process management

### Course reminder

The primitive ***fork()*** creates a child process of the calling process (the parent), with the same program as the latter. The value returned by ***fork()*** is :

- -1: In the event of failure, for example if the process table is full.
- 0 : the value returned to the child, and its *pid* and the *pid* of its parent can be found by calling the functions *getpid()* and *getppid()* respectively.
- A positive integer: the value returned to the parent, which represents the *pid* of its created child.

A process can end its execution by calling the primitive ***exit()***:

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

The process passes its return code as an argument to ***exit()***: an integer between 0 and 255. In the C language, executing “***return n***” from the function *main()* is equivalent to calling ***exit(n)***. A process can also be terminated by a signal, sent by another process or by the operating system itself in the event of abnormal termination.

A parent process can obtain information about the termination of a child using the call ***wait()***:

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

When ***wait()*** is called, the parent process goes to sleep waiting for one of its children to die. When a child dies, ***wait()*** returns the PID of the dead child.

### Exercise n°1

1) What does the following program display?

```
int main() {
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) {
        printf("Dans fils : x=%d\n", ++x);
        exit(0);
    }
    printf("Dans pere : x=%d\n", --x);
    exit(0);
}
```

```
./exo11
Dans pere : x=0
Dans fils : x=2
mohamed@mohamed-VirtualBox:~/TD2$
```

Après l'appel de ***fork()*** on assiste à la création du fils qui va hériter la valeur *x* dont le contenu est 1.

Après le ***if*** le fils incrémente la variable *x* et l'affiche. D'où le fils affiche 2 pour la variable *x* puis se termine en exécutant ***exit(0)***.

Le père décrémente le contenu de la variable x et l'affiche puis se termine. D'où il affiche 0.

Pour les deux processus la valeur de x est 1 au début.

- 2) Illustrate the execution of the following programs through a tree structure, explaining the display of each process and its filiation.

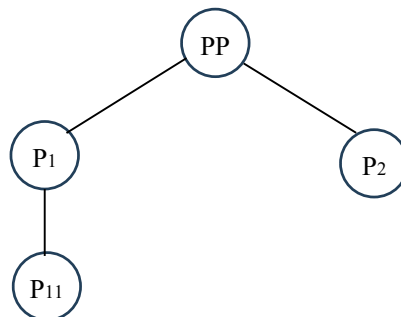
*Program 1*

```
int main() {  
    fork(); // (1)  
    fork(); // (2)  
    printf("hello!\n");  
    exit(0);  
}
```

Le premier *fork()* est exécuté par le processus père (P) d'où il aura la création d'un processus fils (P1) et le deuxième *fork()* est exécuté par le processus père (P) et son fils (P1) d'où on aura la création de deux processus (P2 qui est créé par le père et P11 qui est créée par le processus P1).

Chacun de ces processus affiche le message "hello !" et se termine.

Donc on aura 4 messages "hello !" qui sont affichés.



```
mohamed@mohamed-VirtualBox:~/TD2$ ./exo121  
hello!  
hello!  
mohamed@mohamed-VirtualBox:~/TD2$ hello!  
hello!
```

The terminal screenshot shows the execution of the program. The first two lines are "hello!" from the parent and P1. The next two lines are "hello!" from P11 and P2.

*Program 2*

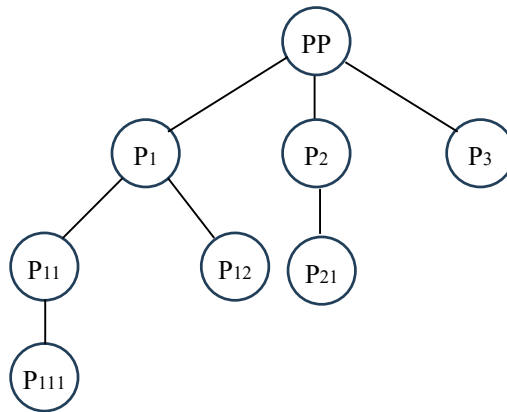
```
int main() {  
    fork(); // (1)  
    fork(); // (2)  
    fork(); // (3)  
    printf("hello!\n");  
    exit(0);  
}
```

Les processus créés seront représentés selon cette arborescence :

Le père (P) crée trois processus par chaque appel de *fork()* : *P1, P2 et P3*

où P1 est créé par le premier *fork()*, P2 est créé par le deuxième *fork()*, et P3 est créé par le troisième *fork()*.

P1 exécute le deuxième et le troisième *fork()*. D'où il crée deux fils P11 et P12.  
P2 exécute le troisième *fork()*. D'où il crée un seul fils P21.  
Le processus P11 exécute à son tour le troisième *fork()*. D'où la création d'un fils P111



```

mohamed@mohamed-VirtualBox:~/TD2$ ./exo122
hello!
hello!
mohamed@mohamed-VirtualBox:~/TD2$ hello!
hello!
hello!
hello!
hello!
hello!
hello!

```

3) How many lines does "TLEMCEN !" print for each of the following programs?

*Program 1*

```

int main() {
    int i;

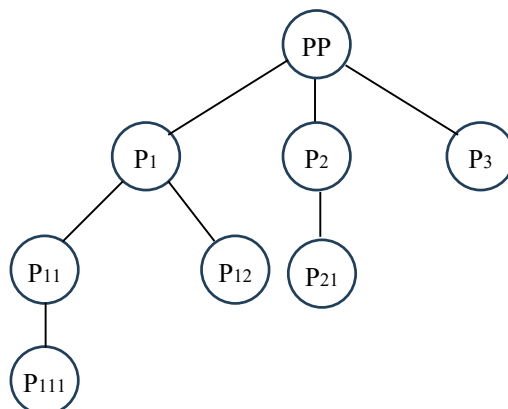
    for(i=0; i<3; i++)
        fork();
    printf("TLEMCEN!\n");
    exit(0);
}

```

```

TLEMCEN!TLEMCEN!mohamed@mohamed-VirtualBox:~/TD2$ TLEMCEN!TLEMCEN!TLEMCEN!TLEMCEN!TLEMCEN!TLEMCEN!
mohamed@mohamed-VirtualBox:~/TD2$ ./exo131
TLEMCEN!TLEMCEN!mohamed@mohamed-VirtualBox:~/TD2$ TLEMCEN!TLEMCEN!TLEMCEN!TLEMCEN!TLEMCEN!TLEMCEN!

```



Pour chacune des valeurs de  $i=0, 1$  et  $2$  le processus père  $P$  crée les processus fils :  $P1, P2$  et  $P3$ .

Pour chacune des valeurs de  $i=1$  et  $2$  le processus  $P1$  (le premier processus crée par  $P$ ) crée deux processus fils :  $P11$  et  $P12$

Pour la valeur de  $i=2$  le processus  $P2$  crée un processus :  $P21$

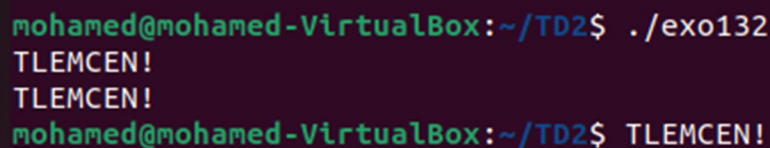
Pour la valeur de  $i=2$  le processus  $P11$  crée un processus :  $P111$

Chaque processus affiche le message "TLEMCEN !"

Donc : le nombre de fois est **8**.

*Program 2*

```
int main() {  
    if (fork())  
        fork();  
    printf("TLEMCEN!\n");  
    exit(0);  
}
```



```
mohamed@mohamed-VirtualBox:~/TD2$ ./exo132  
TLEMCEN!  
TLEMCEN!  
mohamed@mohamed-VirtualBox:~/TD2$ TLEMCEN!
```

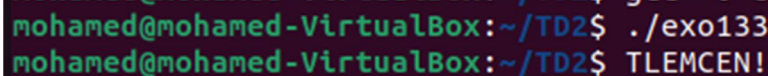
Dans ce cas le deuxième `fork()` est exécuté seulement par le père (il est exécuté que si le premier `fork()` est différent de 0).

Donc : le premier `fork()` crée un premier fils par le père et le deuxième `fork()` crée un deuxième fils par le père. Donc on a trois processus et chacun d'eux affiche le message "TLEMCEN !".

D'où le nombre de fois que le message "TLEMCEN !" est affiché, est : **3**

*Program 3*

```
int main() {  
    if (fork()==0) {  
        if (fork()) {  
            printf("TLEMCEN!\n");  
        }  
    }  
}
```



```
mohamed@mohamed-VirtualBox:~/TD2$ ./exo133  
mohamed@mohamed-VirtualBox:~/TD2$ TLEMCEN!
```

Le premier *fork()* crée un fils par le père ( $P$ ) et ce fils ( $P1$ ) crée à son tour un autre fils ( $P11$ ) (deuxième *fork()*) et affiche le message "TLEMCEN !" .

Donc seulement le processus fils  $P1$  qui affiche et les autres non ( $P$  et  $P11$ ).

D'où le nombre de fois que le message "TLEMCEN !" est affiché, est : **1**

## Exercise n°2

Let's consider the following two programs:

*Program 1*

```
int main(void) {  
    if(fork() && (fork() || fork()));  
    sleep(5);  
    exit(0);  
}
```

*Program 2*

```
int main (void) {  
    int i;  
    for (i =0; i <3; i ++)  
        if (fork ()) i ++;  
    sleep (5);  
    return i ;  
}
```

Assume that any call to the *fork()* primitive does not return an error code.

- 1) How many processes are created by each of these two programs (not counting the parent)?
- 2) Draw the tree structure of the parent and the processes created (the family tree) for each program.
- 3) Suggest a shell command line to check your answers.

### **Solution**

#### **Program 1**

- 1) How many processes are created by each of these two programs (not counting the parent)?**

Tout d'abord nous expliquons les astuces suivantes :

- **if (a && b)**

Algorithmiquement quand deux conditions sont combinées par && si la première est vraie on passera à la vérification de la deuxième condition sinon on sort avec "False".

- **if (a || b)**

Algorithmiquement quand deux conditions sont combinées par || si la première est fausse on passera à la vérification de la deuxième condition sinon on sort avec "False".

```
if(fork() && (fork() || fork()));
```

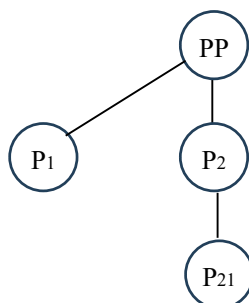
Dans ce cas, le père exécute la première *fork()* et la valeur retournée pour le père est un entier positif (le pid de son premier fils créé par la première *fork()*) alors que le fils créé ne va pas continuer car sa valeur de *fork()* est nulle (false).

Donc : avec la première *fork()* on a un fils créé : 1

Le père exécute le deuxième *fork()* et on assiste à la création d'un deuxième fils. Le père ne va pas continuer car la valeur retournée par *fork()* est positive alors que son fils continue avec la troisième *fork()* puisque sa valeur est nulle. Le deuxième fils crée alors un fils (troisième fils).

Donc : **on assiste à la création de trois processus** : deux processus créés par le père grâce la première *fork()* et la deuxième *fork()* et un fils est créé par le deuxième fils.

- 2) Draw the tree structure of the parent and the processes created (the family tree) for each program.**



### 3) Suggest a shell command line to check your answers.

```
mohamed@mohamed-VirtualBox:~/TD2$ ./exo21 & (sleep 2; ps -o "%P%p%c" | grep exo21)
[1] 4700
 2024    4700  exo21
 4700    4703  exo21
 4700    4704  exo21
 4704    4705  exo21
mohamed@mohamed-VirtualBox:~/TD2$
```

#### Program 2

##### 1) How many processes are created by each of these two programs (not counting the parent)?

i=0

Le père exécute la première fois fork(), crée un fils (p1) et incrémente le i (i=1) puis passe à la boucle for et l'incrémente une autre fois (i=2) alors que le fils (p1) hérite la valeur 0 de la variable i de son père.

- i=2 pour le père

Le père exécute la deuxième fois fork(), crée un fils (p2) et incrémente le i (i=3) puis passe à la boucle for et l'incrémente une autre fois (i=4) puis sort de la boucle alors que le fils (p2) hérite la valeur 2 de la variable i de son père.

- i=1 pour le fils (p1)

Le fils (p1) exécute la première fois fork(), crée un fils (p11) et incrémente le i (i=2) puis passe à la boucle for et l'incrémente une autre fois (i=3) et sort de la boucle alors que le fils (p11) hérite la valeur 1 de la variable i de son père (p1).

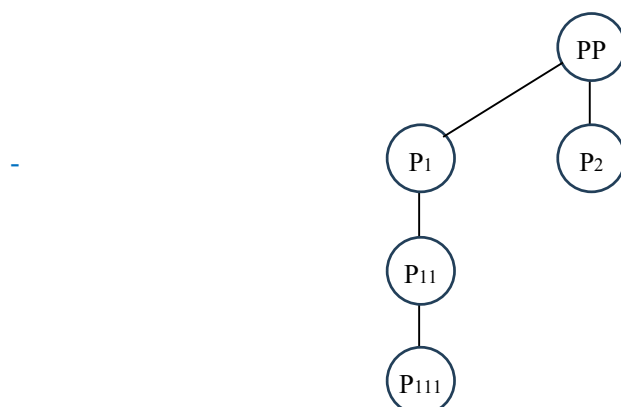
- i=2 pour le fils (p11)

Le fils (p11) exécute la première fois fork(), crée un fils (p111) et incrémente le i (i=3) puis passe à la boucle for et l'incrémente une autre fois (i=4) et sort de la boucle alors que le fils (p111) hérite la valeur 2 de la variable i de son père (p11).

Les processus P2 et P111 ont hérité la valeur (i=2) en passant à la boucle cette valeur sera incrémentée (i=3). Donc ils sortent de la boucle.

Donc : 4 processus sont créés (P1, P2, P11 et P111).

##### 2) Draw the tree structure of the parent and the processes created (the family tree) for each program.



### 3) Suggest a shell command line to check your answers.

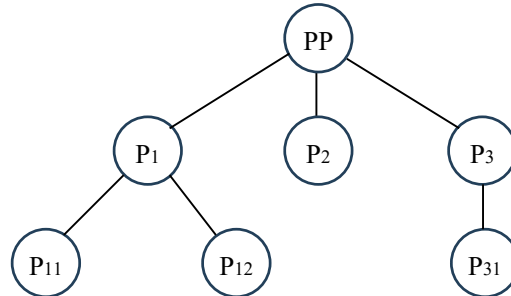
```

[5] Termine 4 ./exo22
mohamed@mohamed-VirtualBox:~/TD2$ ./exo22 & (sleep 2; ps -o "%P%p%c" | grep exo22)
[7] 4671
2024 4671 exo22
4671 4674 exo22
4671 4675 exo22
4674 4676 exo22
4676 4677 exo22
[6] Termine 4 ./exo22
mohamed@mohamed-VirtualBox:~/TD2$

```

### Exercise n°3

Write a program in C language to create the following tree structure and display the *pid* of each process and that of its parent.



```

int main() {

    if (fork()) { // création de P1
        if (fork()) { // création de P2
            if ((fork()) ; // création de P3
            else if (fork()) ; // création P31
                else exit(0);

            }
            else exit(0);
        }
        else if (fork()) { // création de P11
            if (fork()) ; // création de P12
            else exit(0) ;
        }
        else exit(0);
    }
    exit(0);
}

```

### Exercise n°4

- 1) Write a program that creates 10 child processes, each of which will display its sequence number between "0" and "9" 1000 times, i.e. the first child created has sequence number "0" and will have to display the number "0" 1000 times, and the second child created has sequence number "1" and will have to display the number "1" 1000 times, and so on for the other processes created.

```
int main() {

    for (i =0; i <10; i ++){
        if (fork()==0) {
            for (j =0; j <100; j ++){
                printf("%d",i);
                exit(0);
            }
        }
    }
}
```

- 2) Check that this program displays 10000 characters.**

Il suffit d'utiliser la commande `wc` après l'exécution du fichier.

```
mohamed@mohamed-VirtualBox:~/TD2$ ./exo41 | wc -c  
1000  
mohamed@mohamed-VirtualBox:~/TD2$
```

- 3) How will the display appear if system calls are not used to synchronize the processes created?**

[illegible]

Si on ne synchronise pas entre les exécutions des processus on obtiendra un affichage qui n'est pas dans l'ordre.

- 4) Suggest a mechanism for synchronizing the processes created so that the display shows 1000 "0" one after the other, then 1000 "1" one after the other and so on, i.e. a process created will not start the display until its predecessor has finished.

```
int main() {

    for (i =0; i <10; i ++){
        if (fork()==0) {
            for (j =0; j <100; j ++){
                printf("%d",i);
                exit(0);
            }
            Sleep (2);
        }
    }
}
```

Il suffit de synchroniser par la primitive `sleep()` entre les exécutions des processus créés. De ce fait, l'affichage sera dans l'ordre.

[illegible]