

Principe des Systèmes d'Exploitation TP2

Gestion des processus

Table des matières

Ex. 1 : Compilation d'un programme C mono-fichier :.....	3
Ex. 2 : Compilation de programme C multi-fichier.....	8
Ex. 3 Lancement d'un exécutable à partir d'un programme C et traitement des erreurs.....	9
Ex. 4 Exemple de création de processus fils dont le code est le même que celui de leur père.....	12

Ex. 1 : Cycle de vie des processus :

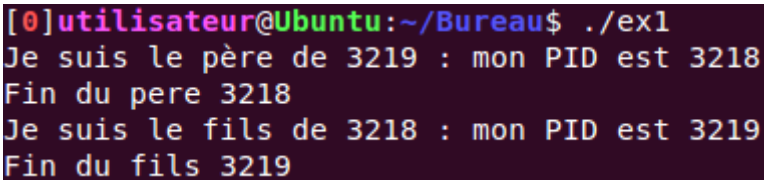
a)

Code en C :

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

int main ()
{
    pid_t rc = fork();
    if(rc>0){
        printf("Je suis le père de %d : mon PID est %d\n", rc,getpid());
        printf("Fin du pere %d\n", getpid());
    }else{
        printf("Je suis le fils de %d : mon PID est %d\n", getppid(),getpid());
        printf("Fin du fils %d\n", getpid());
        fflush(stdout);
    }
    return 0;
}
```

Exemple d'exécution du code :



```
[0]utilisateur@Ubuntu:~/Bureau$ ./ex1
Je suis le père de 3219 : mon PID est 3218
Fin du pere 3218
Je suis le fils de 3218 : mon PID est 3219
Fin du fils 3219
```

b)

Changement du code source :

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

int main ()
{
    pid_t rc = fork();
    if(rc>0){
        sleep(30);
        printf("Je suis le père de %d : mon PID est %d\n", rc,getpid());
        printf("Fin du pere %d\n", getpid());
    }else{
        printf("Je suis le fils de %d : mon PID est %d\n", getppid(),getpid());
        printf("Fin du fils %d\n", getpid());
        fflush(stdout);
    }
    return 0;
}
```

Après l'exécution du programme, l'affichage du fils s'exécute mais pas celui du père (qui doit attendre 30s) , l'affichage de **ps au** donne :

```
utilisa+  3325  0.0  0.0  4196   620 pts/2    S+   08:19   0:00  ./ex1
utilisa+  3326  0.0  0.0    0     0 pts/2    Z+   08:19   0:00  [ex1] <defunct>
```

Le processus fils est en *état « terminé » (=zombie)*.

Pour qu'il puisse disparaître, il faut que le processus père récupère le code retour (à la fin de l'exécution du programme).

c)

Modification du code source :

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

int main ()
{
    pid_t rc = fork();
    if(rc>0){
        printf("Je suis le père de %d : mon PID est %d\n", rc,getpid());
        printf("Fin du pere %d\n", getpid());
    }else{
        sleep(30);
        printf("Je suis le fils de %d : mon PID est %d\n", getppid(),getpid());
        printf("Fin du fils %d\n", getpid());
        fflush(stdout);
    }
    return 0;
}
```

Après la fin du processus père, le fils va se faire adopter par le processus *init* (PID 1981)

```
[...]
-init-----Thunar-----3*[{Thunar}]
             applet.py-----{applet.py}
             at-spi-bus-laun-----dbus-daemon
                                 3*[{at-spi-bus-laun}]
             at-spi2-registr-----{at-spi2-registr}
             bamfdaemon-----3*[{bamfdaemon}]
             blueman-applet-----2*[{blueman-applet}]
             dbus-daemon
             dconf-service-----2*[{dconf-service}]
             deja-dup-monito-----2*[{deja-dup-monito}]
             evolution-calen-----4*[{evolution-calen}]
             evolution-sourc-----2*[{evolution-sourc}]
             ex1
             gconfd-2
```

```
[0]utilisateur@Ubuntu:~/Bureau$ ./ex1
Je suis le père de 3675 : mon PID est 3674
Fin du pere 3674
[0]utilisateur@Ubuntu:~/Bureau$ Je suis le fils de 1981 : mon PID est 3675
Fin du fils 3675
```

d)

Une des bonnes propriétés du « père adoptif » est le fait de pouvoir adopter le « fils orphelin » pour lui permettre de s'exécuter et se terminer correctement pour ne pas le laisser en processus zombie et qu'il disparaisse de la table des processus.

e)

Modification du code source :

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

int main ()
{
    pid_t rc = fork();
    if(rc>0){
        printf("Je suis le père de %d : mon PID est %d\n", rc,getpid());
        printf("Fin du pere %d\n", getpid());
        wait();
    }else{
        sleep(30);
        printf("Je suis le fils de %d : mon PID est %d\n", getppid(),getpid());
        printf("Fin du fils %d\n", getpid());
        fflush(stdout);
    }
    return 0;
}
```

Exécution du programme :

```
[0]utilisateur@Ubuntu:~/Bureau$ ./ex1
Je suis le père de 3777 : mon PID est 3776
Fin du pere 3776
Je suis le fils de 3776 : mon PID est 3777
Fin du fils 3777
```

ps au lors de l'exécution du programme.

```
utilisa+  3776  0.0  0.0  4200  620 pts/2  S+   08:41   0:00  ./ex1
utilisa+  3777  0.0  0.0  4196   88 pts/2  S+   08:41   0:00  ./ex1
```

On peut observer que malgré le fait que le père est fini son exécution, il ne s'arrête pas et attend la fin du processus fils.

f)

Modification du code source :

Le père va ici afficher successivement les processus fils qui se terminent.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    pid_t rc = fork();
    if(rc>0){
        pid_t rc2 = fork();
        if(rc2>0){
            pid_t rc3 = fork();
            if(rc3>0){
                printf("Je suis le père de %d et %d et %d : mon PID est %d\n",
                    rc,rc2,rc3,getpid());
                int id;
                int status;
                id=wait(&status);
                while(id!=-1){
                    printf("\nFin du processus %d \n",id);
                    id=wait(&status);
                }
                printf("Fin du pere %d\n", getpid());
                fflush(stdout);
            }else{
                printf("Je suis le fils de %d : mon PID est %d\n", getppid(),getpid());
                printf("Fin du fils %d\n", getpid());
                sleep(3);
                fflush(stdout);
                exit(0);
            }
        }else{
            printf("Je suis le fils de %d : mon PID est %d\n", getppid(),getpid());
            printf("Fin du fils %d\n", getpid());
            sleep(2);
            fflush(stdout);
            exit(0);
        }
    }else{
        printf("Je suis le fils de %d : mon PID est %d\n", getppid(),getpid());
        printf("Fin du fils %d\n", getpid());
        sleep(1);
        fflush(stdout);
        exit(0);
    }
    return 0;
}
```

Ex. 2 : Combiner fork/exec/wait pour créer des processus exécutant des programmes différents :

a)

Code source :

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

int main (){
    pid_t rc = fork();
    if(rc==0){
        execl("/usr/bin/xterm","xterm",NULL);
    }
}
```

b)

```
graph LR
    xfce4-power-man[xfce4-power-man] --- {xfce4-power-man}
    xfce4-terminal[xfce4-terminal] --- bash1[bash]
    bash1 --- ex2[ex2]
    ex2 --- xterm[xterm]
    xterm --- bash2[bash]
    bash1 --- bash3[bash]
    bash3 --- pstree[pstree]
    bash1 --- gnome-ptv-help[gnome-ptv-help]
```

On peut observer que le processus *ex2* est le père du processus qui utilise *xterm*.

c)

```
graph LR
    xfce4-power-man[xfce4-power-man] --- {xfce4-power-man}
    xfce4-terminal[xfce4-terminal] --- bash1[bash]
    bash1 --- ex2[ex2]
    ex2 --- sh[sh]
    sh --- xterm[xterm]
    xterm --- bash2[bash]
    bash1 --- bash3[bash]
    bash3 --- pstree[pstree]
    bash1 --- gnome-ptv-help[gnome-ptv-help]
```

On peut observer que l'exécution est la même sauf que il y a un processus *sh* en plus dans l'exécution.

d)

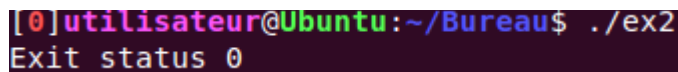
Modification du code source :

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

int main (){
    pid_t rc = fork();
    if(rc>0){
        int status =0;
        wait(&status);
        if(WIFEXITED(status))
            printf("Exit status %d \n",WEXITSTATUS(status));
        else if(WIFSIGNALED(status))
            printf("Killed by signal %d \n",WTERMSIG(status));

    }else{
        execl("/usr/bin/xterm","xterm",NULL);
    }
}
```

Lorsque l'on ferme le terminal xterm, il s'agit d'un arrêt normal.



```
[0]utilisateur@Ubuntu:~/Bureau$ ./ex2
Exit status 0
```

e) Les fonctions **execlp** , **execv**, **execvp** et **execl** sont toutes des fonctions reliées a **execve** qui permet d'effectuer un *appel-systeme*.

- **execv** va permettre de passer les arguments sous forme de tableau.
- **execvp** est une commande **execv** où le fichier à exécuter sera recherché à l'aide de la variable d'environnement PATH.
- **execl** va permettre de passer les arguments sous forme de liste
- **execlp** est une commande **execl** où le fichier à exécuter sera recherché à l'aide de la variable d'environnement PATH.

Ex. 3 : Contrôle d'un processus par envoi de signaux :

a)

SIGINT (CTRL+C):

```
[0]utilisateur@Ubuntu:~/Bureau$ ./ex3
Attention : ce programme boucle indéfiniment !
.....^C
[0]utilisateur@Ubuntu:~/Bureau$
```

SIGSTOP (CTRL+Z):

```
[0]utilisateur@Ubuntu:~/Bureau$ ./ex3
Attention : ce programme boucle indéfiniment !
...^Z
[1]+  Arrêté                  ./ex3
[0]utilisateur@Ubuntu:~/Bureau$
```

bg et fg:

```
[0]utilisateur@Ubuntu:~/Bureau$ ./ex3
Attention : ce programme boucle indéfiniment !
....^Z
[1]+  Arrêté                  ./ex3
[0]utilisateur@Ubuntu:~/Bureau$ fg %1
./ex3
.....^Z
[1]+  Arrêté                  ./ex3
[0]utilisateur@Ubuntu:~/Bureau$ bg %1
[1]+ ./ex3 &
[0]utilisateur@Ubuntu:~/Bureau$ .....g...fg %1.
./ex3
.^Z
[1]+  Arrêté                  ./ex3
[0]utilisateur@Ubuntu:~/Bureau$
```

b)

```
[0]utilisateur@Ubuntu:~$ kill 4902
[0]utilisateur@Ubuntu:~$
```

Le **kill** provoque bien l'arrêt du programme.

```
[0]utilisateur@Ubuntu:~/Bureau$ ./ex3
Attention : ce programme boucle indéfiniment !
.....Complété
[0]utilisateur@Ubuntu:~/Bureau$
```

c)

```
[0]utilisateur@Ubuntu:~$ kill -SIGSTOP 4908  
[0]utilisateur@Ubuntu:~$ kill -SIGCONT 4908  
[0]utilisateur@Ubuntu:~$ kill 4908
```

On fait successivement le **SIGSTOP** qui va arrêter le processus puis **SIGCONT** qui va le reprendre.

```
[0]utilisateur@Ubuntu:~/Bureau$ ./ex3  
Attention : ce programme boucle indéfiniment !  
.....  
[1]+  Arrêté                  ./ex3  
[0]utilisateur@Ubuntu:~/Bureau$ .....ls
```