Rapport Lab 1

Réalisé par :

- Hugo Pauthier
- Vincent Hardouin

2 - Creating and Running a Process (1) - fork

Definition

Le fork créé un nouveau processus en dupliquant le processus appelé.
Comme il s'agit d'un nouveau processus alors il a un autre PID et son PPID réfère celui du processus qui l'a créé. Les deux processus évoluent dans des espaces mémoire différent.

3 - Small program to show different message for each process

Code

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

int main() {
    if (fork() == 0) {
        printf("Hey ! Je suis le processus fils. PID : %d \n", getpid());
    } else {
        printf("Hey ! Je suis le processus parent. PID : %d \n", getpid());
    }
}
```

Résultat

Ce programme une fois compilé et lancé nous donne :

```
ING4-systemes-d-exploitation/week1/lab1 on  main [!?]
❯ ./part2fork
Hey ! Je suis le processus parent. PID : 69397
```

```
Hey ! Je suis le processus fils. PID : 69398
```

Commentaire

Nous voyons bien que la fonction fork a créé un nouveau processus comme ces derniers n'ont pas le même pid.
La fonction fork retourne le PID de l'enfant au processus parent et elle retourne 0 au processus enfant.
S'il y a une erreur elle renvoie -1 au parent et aucun processus enfant n'est créé.

4 - Data share

Avec l'exemple donné nous avons obtenu :

```
❯ ./part2forkb
Hey ! Je suis le processus fils. PID : 69771
Hey ! Je suis le processus parent. PID : 69770
5
```

Commentaire

Le processus enfant souhaite incrémenter la variable `i`. Le `sleep` nous permet de bloquer l'exécution du parent afin d'être sur que l'enfant ait le temps d'exécuter son code. Or en sortie, on constate que la variable `i` a toujours la valeur 5.

On peut donc conclure que les données ne sont pas partagées entre processus parents / enfants.

5 - Create more than one process

Nous avons créé le programme suivant :

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


void create_fork() {
```

```
    if(fork() == 0) {
        printf("Hey ! Je suis le processus fils de PPID : %d et mon PID est :
%d \n", getppid(), getpid());
    }
}

int main() {
    if (fork() == 0) {
        printf("Hey ! Je suis le processus fils de PPID : %d et mon PID est :
%d \n", getppid(), getpid());
        create_fork();
    } else {
        printf("Hey ! Je suis le processus parent. PID : %d \n", getpid());
        create_fork();

    }
    sleep(2);
}
```

Celui-ci permet de créer un processus fils qui lui même créé un processus fils. D'autre part, le processus parent créé un second processus fils.

```
❯ ./part2forkc
Hey ! Je suis le processus parent. PID : 70557
Hey ! Je suis le processus fils de PPID : 70557 et mon PID est : 70560
Hey ! Je suis le processus fils de PPID : 70557 et mon PID est : 70561
Hey ! Je suis le processus fils de PPID : 70560 et mon PID est : 70562
```

Commentaire

Nous voyons bien que le processus Parent (PID : 70557) a créé 2 processus (PID : 70560, 70561).
Et que le premier processus enfant (PID : 70560) en a créé 1 (PID : 70562)

3 - Creating and Running a Process (2) - exec

3 - Run Exec

Pour ouvrir une application sous Mac à l'aide d'une commande nous pouvons faire : `open -a appName`.

Dans un programme C, nous pouvons donc faire un exec avec la même commande :

```c
#include <unistd.h>

int main() {
    execlp("open", "open", "-a", "firefox", (char *) NULL);
}
```

Le `(char * NULL)` permet de dire qu'on a plus d'arguments à ajouter à notre commande.

4 - What happens after exec

Comme le `exec` remplace le processus actuel (il remplace son code et ses données) alors la suite du programme n'est pas lu, nous n'avons donc pas l'affichage de la valeur de `i`.

5 - System

Nous voulons implémenter notre méthode `system` afin de voir comment celle-ci fonctionne.

Nous avons créé une méthode `mySystem` :

```c
void mySystem(char* cmd) {
    int pid = fork();
    int status;

    if (pid == 0) {
        sleep(2);
        execl("/bin/sh", "sh", "-c", cmd, (char *) NULL);
    }
```

```
    waitpid(pid, &status, 0);

    printf("Fin de la commande, tout c'est bien déroulé.\n");
}
```

Commentaire

Dans notre méthode, nous avons créé un `fork` afin de lancer un `exec`. Ce dernier, lancera notre commande.

Ensuite, dans le processus parent, nous avons utilisé la méthode `waitpid` qui nous permet d'attendre la fin d'exécution d'un processus. Dans notre cas, nous lui avons fourni le PID de notre processus fils afin de l'attendre dans notre processus parent.

`waitpid` retourne le pid du processus fils, ou `-1` s'il y a eu une erreur.

Résultat, nous avons notre processus parent qui continue seulement lorsque le processus fils a fini son exécution.

Fork vs Exec

La fonction `fork` créé un processus copié du processus parent, mais à son propre code de retour.
Tandis que, `exec` remplace le processus actuel par un nouveau programme.