TD 5 : Processus, fork, signaux

Objectifs pédagogiques :

- **création de processus, wait**
- **signaux**

Introduction

Dans ce TD, nous allons étudier l'API proposée par POSIX pour la création et la gestion de processus. Il faudra donc un système POSIX (en particulier pas windows) pour compiler et exécuter nos programmes.

Le standard POSIX définit une API en C à bas niveau pour l'interaction entre les processus utilisateurs et le système d'exploitation. Cette API reste accessible dans les programmes C++.

1.1 Fork en séquence et en parallèle

Question 1. Proposez un programme qui crée N processus fils en parallèle, puis attend leur terminaison. Chaque processus créé doit afficher son *pid*, le *pid* de son pere, et son numéro d'ordre de création.

Question 2. Chacun des fils doit rendre à la terminaison son numéro de création. Le processus *main* doit afficher la valeur de sortie de chacun des fils et le pid du fils dont il vient de détecter la terminaison.

Question 3. En supposant que les fils ont la même durée d'exécution, vont-ils mourir dans l'ordre de leur création ? Modifiez le programme pour que le main attende la fin des fils dans l'ordre de leur création.

Question 4. On reprend la question 1, mais cette fois proposez un programme qui crée N processus à travers une chaîne de créations, c'est à dire qu'il crée un fils qui crée un fils... sur N générations. Le père ne doit se terminer que quand tous ses fils sont terminés.

1.2 Fork arbitraires et Arbre des processus

On considère le programme suivant :

fork1.cpp

```cpp
#include <iostream>
#include <unistd.h>

int main () {
      int N = 4;
      int i = 0;
      std::cout << "pid=" << getpid() << std::endl;
      while (i < N) {
            i++;
            int j = i;
            if (fork() == 0) {
                  if (i%2==0) {
                        N = i - 1;
                        i = 0;
                  } else {
                        N = 0;
                  }
                  std::cout << "pid=" << getpid() << " ppid=" << getppid() << " j=" << j <<
                      " N=" << N << std::endl;
            }
```

```
    }
    return 0;
}
```

Question 5. En comptant le processus initial, combien de processus engendre l'execution du programme ? Donnez l'arbre des descendances de processus, et l'affichage de chacun d'entre eux.

Question 6. Modifiez le programme pour que le père ne se termine qu'après que tous les autres processus soient terminés. On n'attendra pas plus de fils qu'on en a crée et on ne limitera pas le parallélisme.

1.3 Signaux

On utilisera les fonctions pour les signaux dans les exercices suivants :

- Mise en place d'un handler pour un signal : `signal-handler* signal(int sig, signal-handler *)`; où le `signal-handler` est une fonction prenant un *int* (le numéro du signal reçu) et rendant *void*, ou une des macros *SIG_IGN* (ignorer, le signal sera perdu), ou *SIG_DFL* (comportement par défaut face au signal, mourir ou ignorer le plus souvent)
- Envoyer un signal à un processus ou groupe de processus : `int kill(int pid, int signal)`
- Demander au système de nous envoyer un *SIGALRM* au bout de *n* secondes : `int alarm(int seconds)`
- Manipulation d'un ensemble de signaux (type `sigset_t`): `sigfillset(sigset *); sigemptyset(sigset*); sigdelset(sigset *,signal); sigaddset(sigset*,signal);`
- Se bloquer en attente des signaux qui ne sont *PAS* dans le masque argument : `int sigsuspend(sigset *mask)`. NB: si un des signaux est "pending" il est traité immédiatement.
- Bloquer (mettre en attente dans "pending") les signaux qui sont dans le masque (ensemble de signaux) argument : `int sigprocmask(int how,sigset *mask,sigset * old)`. On passera *SIG_BLOCK* dans *how* pour ajouter un signal au masque actif, et *nullptr* dans old si on ne se soucie pas du masque précédent.

1.4 Tour par tour

On veut construire un programme ou deux processus s'exécutent tour à tour. Pour cela on demande de n'utiliser que le mécanismes des signaux.

Question 7. Ecrire un tel programme :

- Le père commence le premier tour, mais il attend une seconde avant de commencer. Le fils commence son tour par une attente.
- Chaque processus quand il a la main affiche un message avec son pid, et le tour auquel il en est.
- A la fin de chaque tour on passe la main à l'autre processus, pour cela on lui envoie un signal, puis on se suspend en attente de sa réponse.
- Chacun des processus fait M tours d'alternance.

Question 8. Si le père n'attend pas avant de démarrer, on constate que parfois le fils reçoit le premier signal AVANT de faire le *sigsuspend*. Proposez une correction utilisant les masques (sigprocmask) pour corriger ce problème.

1.5 Signaux, Alarmes

On désire réaliser un programme qui créé N processus fils en parallèle, puis attend T secondes avant d'envoyer un SIGINT à tous ses fils. A leur création, les processus fils se bloquent en attente du signal de leur père ; à la réception de ce signal, ils affichent un message indiquant qu'ils vont se

terminer. Le processus père attend que tous ses fils soient finis avant de se terminer en affichant un message "Fin du programme".

Question 9. Ecrivez ce programme.

Question 10. Le père peut-il notifier tous ses fils sans avoir stocké leur pid ? Ecrivez une version qui a ce comportement.

Question 11. La terminaison (exit) d'un processus fils engendre un signal *SIGCHLD* à destination du père, le comportement par défaut étant de l'ignorer. Le père peut-il s'appuyer sur ce mécanisme plutôt que sur *wait* pour attendre la terminaison de ses fils ?

TME 5 : Processus, Signaux

Objectifs pédagogiques :

- **fork, wait**
- **signaux**

1.1 Github pour le TME

Les fichiers fournis sont a récupérer suivant la même procédure qu'au TME4 sous authentification github : https://classroom.github.com/a/2DXatkHd.
Vous ferez un push de vos résultats sur ce dépot pour soumettre votre travail.

1.2 Encore un Arbre des processus

forkexo2.cpp

```cpp
#include <iostream>
#include <unistd.h>

int main () {
      const int N = 3;
      std::cout << "main pid=" << getpid() << std::endl;

      for (int i=1, j=N; i<=N && j==N && fork()==0 ; i++ ) {
            std::cout << " i:j " << i << ":" << j << std::endl;
            for (int k=1; k<=i && j==N ; k++) {
                  if ( fork() == 0) {
                        j=0;
                        std::cout << " k:j " << k << ":" << j << std::endl;
                  }
            }
      }
      return 0;
}
```

Question 1. En comptant le processus initial, combien de processus engendre l'execution du programme ? Donnez l'arbre des descendances de processus, et l'affichage de chacun d'entre eux.

Question 2. Modifiez le programme pour que le père ne se termine qu'après que tous les autres processus soient terminés. On n'attendra pas plus de fils qu'on en a crée et on ne limitera pas le parallélisme.

1.3 Combat de Signaux

Nous allons construire une application où deux processus s'envoient respectivement des signaux avec kill pour simuler un combat (Vador et Luke). Le principe est que chaque combattant (processus) alterne entre une phase de défense où il se protège (en ignorant les signaux), et une phase d'attaque où il envoie un signal (coup de sabre laser) à l'adversaire (attaque) mais devient en contrepartie vulnérable aux signaux pendant un moment.

On va modéliser également des points de vie, c'est-à-dire que le fait de recevoir un signal quand on est vulnérable (dans sa phase d'attaque) va décrémenter un compteur (initialement trois vies). Quand le compteur atteint 0, le processus meurt et rend la valeur 1. Quand l'attaquant détecte la mort de son adversaire, c'est-à-dire que son pid n'existe plus (ce qui cause une exception sur kill), il meurt également et rend la valeur 0.

Le temps passé en phase de défense (signal masqué) ou d'attaque (signal reçu) est aléatoire et compris entre 0.3 et 1 seconde.

Définir les trois fonctions suivantes :

- `void attaque (pid_t adversaire)` :
  - La phase d'attaque commence par installer un gestionnaire pour le signal SIGINT, qui décrémente les « points de vie » du processus et affiche le nombre de points restants. Si 0 est atteint il affiche que le processus se termine, et celui-ci se termine en retournant 1.
  - Ensuite le processus envoie un signal SIGINT à l'adversaire ; si cette invocation échoue, on suppose que l'adversaire a déjà perdu, et le processus sort avec la valeur 0 ;
  - Ensuite le processus s'endort pour une durée aléatoire.
- `void defense()`
  - La phase de défense consiste à désarmer le signal SIGINT en positionnant son action à SIG_IGN ;
  - Ensuite le processus s'endort pour une durée aléatoire.
- `void combat(pid_t adversaire)` : boucle indéfiniment sur une défense suivie d'une attaque et invoquez-la dans le corps des deux fils.

Question 3. Assemblez ces éléments pour créer un combat entre le processus principal (vador) et son fils (luke).

Question 4. Pour attendre une durée aléatoire dans ce scenario on propose la fonction suivante `randsleep` suivante. En lisant le manuel de nanosleep https://man.cx/nanosleep, en particulier les erreurs possibles, expliquez la boucle qui est proposée dans ce code.

```c
#include <time.h>

void randsleep() {
 int r = rand();
 double ratio = (double)r / (double) RAND_MAX;
 struct timespec tosleep;
 tosleep.tv_sec =0;
 // 300 millions de ns = 0.3 secondes
 tosleep.tv_nsec = 300000000 + ratio*700000000;
 struct timespec remain;
 while ( nanosleep(&tosleep, &remain) != 0) {
  tosleep = remain;
 }
}
```

Question 5. Comment assurer que les deux processus qui s'affrontent utilisent une graine aléatoire différente ?

On propose de modifier la défense de Luke (le fils) pour qu'il affiche les coups parés. On propose procéder de la manière suivante :

- positionner un handler qui affiche "coup paré" avec sigaction
- masquer les signaux avec sigprocmask ;
- s'endormir pendant une durée aléatoire avec randsleep ;
- invoquer sigsuspend pour tester si une attaque a eu lieu, et donc afficher le message "coup paré" si le signal a été reçu.

Question 6. Le combat est-il encore équitable ? Expliquez pourquoi.

1.4 Pseudo wait avec des signaux

Nous voulons implémenter la fonction `int wait_till_pid(pid_t pid)`, qui suspend l'exécution du processus appelant jusqu'à ce qu'il prenne connaissance de la terminaison de son processus fils de pid *pid*. La fonction retourne *pid* lorsque le processus fils *pid* se termine, ou -1 en cas d'erreur (fils *pid* n'existe pas).

N.B: Si l'appelant a créé d'autres fils et que l'un de ceux-ci se termine durant l'appel, alors cette terminaison sera définitivement perdue.

Question 7. Sans s'appuyer sur *waitpid*, simplement avec *wait* et une boucle, implanter la fonction `wait_till_pid`.

Nous voulons maintenant modifier la fonction précédente en une fonction `int wait_till_pid (pid_t pid, int sec)`.

Le paramètre *sec* indique le temps maximum en secondes durant lequel le processus appelant attendra la terminaison de son fils *pid*. La fonction retourne 0 si le délai expire avant la terminaison de ce fils, sinon elle retourne *pid* comme dans la question précédente.

Question 8. Toujours sans utiliser *waitpid*, mais à plutôt l'aide de signaux, implanter ce comportement. Attention à bien désarmer une alarme temporisée si le fils meurt avant le temps imparti. Indice : On pourra se placer en attente des deux signaux : SIGCHLD (un fils est mort) et SIGALRM (temporisateur).