TP: Processus

R305

Département Informatique

2022

# Exercice 1

## Introduction

#include <sys/types.h>  
#include <unistd.h>  
#include <stdint.h>  
#include <time.h>  
#include <stdlib.h>  
#include <stdio.h>  
  
#define NB\_VALUES 10  
  
pid\_t getpid(void);  
  
int sleep\_times[NB\_VALUES] = {1, 1, 1, 2, 2, 3, 3, 4, 5, 6};  
  
int main(int argc, char \*\*argv)  
{  
 printf("Processus >>%jd<< est lance\n", (intmax\_t)getpid());  
 int sleeptime = sleep\_times[rand() % NB\_VALUES];  
 printf("Processus >>%jd<< dort pendant %d sec\n", (intmax\_t)getpid(),sleeptime);  
 sleep(sleeptime);  
 printf("Processus >>%jd<< termine\n", (intmax\_t)getpid());  
 exit(0);  
}

Le programme C ci-dessus simule un traitement informatique en suspendant son activité pour un laps de temps aléatoire.

Dans un premier temps, veuillez compiler ce programme en nommant le programme exécutable résultant sleeper

## La commande ps

Ouvrez un terminal shell et lancez la commande suivante:

./sleeper & ps -l

Questions:

* que fait la commande ps -l
* expliquez l’usage du caractère &
* sachant que la commande ps affiche le PID et le PPID des processus en cours d’exécution, retrouvez l’arbre hiérarchique complet des processus affichés par ps.
* quel est le rôle joué par le processus shell dans l’arbre hiérarchique ?
* que pouvez-vous déduire sur le fonctionnement du shell ?

# Exercice 2

## introduction à fork

pid\_t pid;  
 pid = fork();  
 if (pid > 0)  
 {  
 printf("Je suis le parent[pid=%jd] => fils[pid=%jd] \n", (intmax\_t)getpid(), (intmax\_t)pid);  
 }  
 else  
 {  
 if (!pid)  
 {  
 printf("Je suis le fils[pid=%jd] <= mon parent[pid=%jd] \n", (intmax\_t)getpid(), (intmax\_t)getppid());  
 }  
 else  
 {  
 if (pid == -1)  
 {  
 perror("fork");  
 }  
 }  
 }

Voici un code C qui permet de tester le mécanisme fork.

Dans un premier, complétez ce code et testez le.

Vérifiez que les PIDs affichés sont cohérents.

## serveur de calcul

Le programme présenté en [Annexe A](Annexe%20A) simule un serveur de calcul.

Les requêtes de calcul se trouvent dans un tableau requetes\_tab: il s’agit de réaliser des opérations arithmétiques à deux opérandes de type float.

Les opérations considérées sont l’addition, la soustraction, la multiplication et la division.

Ces opérations sont codées en caractères ascii indiqués par le tableau char operations[].

Avant de passer à l’étape suivante, veuillez compiler et tester le code du serveur de calcul. Vous allez remarquer qu’après un certain temps, qui simule le traitement des opérations, les résultats des différentes opérations seront affichés.

### A faire:

Nous souhaitons maintenant réaliser une version parallèle de ce serveur. Les calculs arithmétiques demandés s’effectueront en parallèle dans cette version.

Proposez une version parallèle du ce serveur en utilisant le mécanisme fork:

* le processus père devra attendre la fin de tous les processus fils avant de terminer.
* comparez les temps d’exécution entre la version séquentielle et la version parallèle. Que peut-t-on déduire ?

# Exercice 3

Le code en [Annexe B](#Annexe%20B) présente un simulateur de différents cas de terminaison d’un processus.

Ce simulateur traite en particulier de 4 cas de terminaison:

* une fin du processus avec un exit
* une fin du processus avec un return
* une fin du processus provoquée par la réception d’un signal non géré
* une fin du processus provoquée par une erreur

## A faire:

Nous vous demandons de créer un programme C qui utilisera la technique fork/exec pour lancer ce simulateur.

Vous ne devez pas modifier le code du simulateur présenté mais le lancer avec un fork/exec depuis votre programme.

Votre programme devra récupérer correctement le statu de terminaison du simulateur (son fils) et en déduire l’événement qui en était la cause.

# Exercice 4

Attention cet exercice fera l’objet d’un rendu individuel qui comptera pour le contrôle continu.

## étape 1

Nous vous demandons de réaliser votre propre programme shell simplifié.

Ce programme devra répondre aux spécifications suivantes:

* il devra présenter un invite en attendant que l’utilisateur saisisse le nom d’un programme à lancer avec éventuellement des paramètres
* vous devez lancer le programme demandé par l’utilisateur en utilisant fork/exec. Attention il faudra prendre en compte les paramètres donnés par l’utilisateur.
* vous devez attendre la fin du processus lancé avant d’en lancer un nouveau (comme un process group foreground)

## étape 2

Modifiez votre programme pour prendre en compte le lancement de processsus en background. Ainsi, si l’utilisateur termine sa commande avec le caractère & alors le processus sera lancé en background et votre shell pourra prendre en compte de nouvelles commandes immédiatement.

## Aide/Astuces

#### Aide 1

L’algorithme général d’un programme shell est la boucle suivante:

TantQue (vrai) :  
 Lire une commande utilisateur  
 Separer le nom du programme des eventuels parametres  
 Lancer la commande utilisateur avec le mécanisme fork/exec  
 Si(Mode Foreground) Alors   
 Attendre la fin du processus fils et lire son statu

#### Aide 2

Les fonctions scanf ou fgets permettent de lire une chaine de caractères du clavier.

#### Aide 3

La fonction strtok permet de séparer une chaine de caractères en plusieurs tokens en utilisant un séparateur.

Voici un exemple d’utilisation pour afficher les mots séparés par un espace:

pch = strtok (str," "); //notez le separateur  
while (pch != NULL)  
{  
 printf ("%s\n",pch);  
 pch = strtok (NULL, " "); //notez le NULL  
}

#### Aide 4

La fonction wait, vue en cours, permet d’attendre la fin d’un processus fils.

# Annexe A

#include <sys/types.h>  
#include <unistd.h>  
#include <stdint.h>  
#include <time.h>  
#include <stdlib.h>  
#include <stdio.h>  
  
// taille du tableau de requêtes  
#define TAILLE\_FILE 10  
  
// une structure pour la requete et le resultat  
typedef struct  
{  
 float operand1;  
 float operand2;  
 float resultat;  
 char operation; // '+' '\*' '/' '-'  
} requete\_t;  
  
// simulateur du calculateur  
float simulateur\_calcul(float op1, float op2, char operation)  
{  
 float result;  
 switch (operation)  
 {  
 case '+':  
 sleep(1);  
 result = op1 + op2;  
 break;  
 case '-':  
 sleep(1);  
 result = op1 - op2;  
 break;  
 case '\*':  
 sleep(2);  
 result = op1 \* op2;  
 break;  
 case '/':  
 sleep(3);  
 result = op1 / op2;  
 break;  
 default:  
 sleep(1);  
 result = 0;  
 break;  
 }  
 return result;  
}  
  
// les opérations arithmétiques  
#define NUMBER\_OPS 4  
char operations[NUMBER\_OPS] = {  
 '+',  
 '-',  
 '\*',  
 '/',  
};  
// tableau des commandes à faire  
requete\_t requetes\_tab[TAILLE\_FILE];  
  
void initialisation()  
{  
 srand((unsigned int)time(NULL));  
}  
  
// retourne un nombre float random entre 0 et a  
float random\_float(float a)  
{  
 return ((float)rand() / (float)(RAND\_MAX)) \* a;  
}  
// fonction pour remplir le tableau des opérations  
int remplir\_les\_commandes()  
{  
 for (int i = 0; i < TAILLE\_FILE; i++)  
 {  
 requetes\_tab[i].operation = operations[rand() % NUMBER\_OPS];  
 requetes\_tab[i].operand1 = random\_float(1.0);  
 requetes\_tab[i].operand2 = random\_float(1.0);  
 }  
}  
  
int main(int argc, char \*\*argv)  
{  
 initialisation();  
 remplir\_les\_commandes();  
 // traitement  
 for (int i = 0; i < TAILLE\_FILE; i++)  
 {  
 requetes\_tab[i].resultat = simulateur\_calcul(requetes\_tab[i].operand1,  
 requetes\_tab[i].operand2,  
 requetes\_tab[i].operation);  
 }  
  
 // affichage des résultats  
 for (int i = 0; i < TAILLE\_FILE; i++)  
 {  
 printf("%f %c %f = %f\n", requetes\_tab[i].operand1,  
 requetes\_tab[i].operation,  
 requetes\_tab[i].operand2,  
 requetes\_tab[i].resultat);  
 }  
 exit(0);  
}

# Annexe B

## Simulateur Terminaison

#include <sys/types.h>  
#include <unistd.h>  
#include <stdint.h>  
#include <time.h>  
#include <stdlib.h>  
#include <stdio.h>  
#include <signal.h>  
  
pid\_t getpid(void);  
  
// les differents evenements pour terminer ce processus  
#define NB\_VALUES 4  
enum naturel  
{  
 FIN\_EXIT,  
 FIN\_RETURN,  
 FIN\_SIGNAL,  
 FIN\_ERREUR  
};  
enum naturel evenements\_tab[NB\_VALUES] = {FIN\_EXIT, FIN\_RETURN, FIN\_SIGNAL, FIN\_ERREUR};  
// les differents signaux que nous allons nous envoyer.  
#define NB\_SIGNAUX 4  
int signaux\_tab[NB\_SIGNAUX] =  
 {  
 SIGTSTP,  
 SIGABRT,  
 SIGQUIT,<a name="head1234"></a>  
 SIGINT};  
  
// la fonction main du simulateur de terminaison du processus  
int main(int argc, char \*\*argv)  
{  
 srand(time(NULL));  
 printf("Processus >>%jd<< est lance\n", (intmax\_t)getpid());  
 // tirage de l evenement  
 enum naturel evenement = evenements\_tab[rand() % NB\_VALUES];  
 printf("Processus[%jd] evenement = %d\n", (intmax\_t)getpid(), evenement);  
  
 // traitement de l evenement  
 switch (evenement)  
 {  
 case FIN\_EXIT:  
 {  
 int codeexit = rand() % 10;  
 printf("Processus[%jd]: exit avec code %d\n", (intmax\_t)getpid(), codeexit);  
 exit(codeexit);  
 break;  
 }  
  
 case FIN\_SIGNAL:  
 {  
 int codesignal = signaux\_tab[rand() % NB\_SIGNAUX];  
 printf("Processus[%jd]: signal numero %d\n", (intmax\_t)getpid(), codesignal);  
 kill(getpid(), codesignal); // envoie d un signal a nous meme  
 break;  
 }  
 case FIN\_RETURN:  
 {  
 int codereturn = rand() % 10;  
 printf("Processus[%jd]: return (%d)\n", (intmax\_t)getpid(), codereturn);  
 return (codereturn);  
 break;  
 }  
 case FIN\_ERREUR:  
 {  
 printf("Processus[%jd]: fin sur erreur \n", (intmax\_t)getpid());  
 float infy = 1 / (1 - 1);//erreur  
 break;  
 }  
 default:  
 break;  
 }  
 printf("Processus >>%jd<< termine\n", (intmax\_t)getpid()); // dead code  
}