Semestre 3. Licence 2 Ingénierie Informatique Cours Principes des Systèmes d'Exploitation

TP2 Systèmes d'Exploitation : Processus sous Unix

I. Environnement de développement

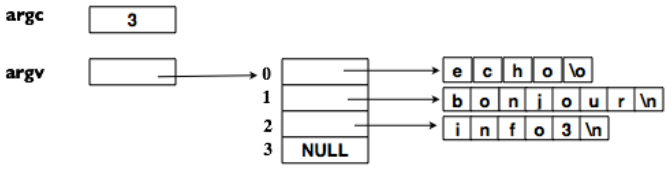
L'environnement de travail pour Les TPs de programmation des systèmes d'exploitation.
- Utilisation obligatoire du système Linux ou MAC OS
- Édition conseillée avec emacs, gedit, etc.
- Compilation avec gcc -o programme_executable programme.c ...

II. Bases en langage C

Les Arguments de la fonction main()
Il est possible de transmettre des arguments (en général par le shell) à *main* () d'un exécutable C lors du lancement de son exécution grâce aux deux arguments : **`main (int argc, char **argv)`** ou `int main (int argc, char *argv[])` où :
- **argc** (arguments count) est le compteur d'arguments ou nombre d'arguments présents (taille du tableau argv), nom du programme exécutable inclus
- **argv** (arguments vector) est un vecteur de pointeurs sur chaînes de caractères (un tableau de argc arguments, sous forme de chaîne de caractères) qui contiennent les arguments de la ligne de commande, à raison de un par chaîne ; argv [0] pointe sur le nom de l'exécutable et argv [argc] vaut NULL.

Exemple:`$echo bonjour info3`

argc vaut 3, argv[0] pointe sur "echo" ; argv[1] pointe sur "bonjour" ; argv[2] pointe sur "info3"



Ainsi la commande : `$echo bonjour info3` donnera la valeur 3 à `argc` et initialisera `argv` avec les 3 chaînes de caractères "echo", "bonjour" et "info3".
NB: Comme argv[0] contient toujours le nom de la commande, `argc` est toujours supérieur ou égal à 1.

Exercice 0: Test des arguments

Tester l'exécutable de l'exemple `exo1.c` suivant:

```c
#include <stdio.h>
int main ( int argc, char *argv[] ) {
        int i = 0;
        printf("argc = %d\n", argc);
        for (i=0; i<argc; i++) {
                printf("argv[%d] =  %s\n", i, argv[i]);
        }
        }
```

```
$gcc  -o exo1  exo1.c
```

Placer vous dans le repertoire de l'exécutable généré à la compilation

```
$./exo1
```

Exercice 1: La fonction `systeme()`

La fonction fait partie de la librairie standard C. Lors de son appel à l'intérieur d'un programme, celui-ci est interrompue, et par conséquent l'informations nécessaires à la reprise ultérieure du programme interrompue est sauvegardée.
Ainsi un processus qui exécute la commande correspondant à la chaîne de caractère passée en paramètre est créé.
Créer un processus à l'aide de l'exemple du fichier `AppSystem.c` en lançant la commande "`ls -a`" dans un shell.

`AppSystem.c`

```c
#include <stdlib.h>/*pour atoi()*/
```

```c
int main(int argc, char **argv)
{ int valRetour; /*retourne 0 si le shell ne peut pas s'éxécuter,
sinon retourne le code de la commande*/
valRetour=system("pstree");
        return valRetour;
}
```

Compiler avec :
`$gcc -o AppSystem AppSystem.c`
Placer vous dans le repertoire de l'exécutable généré à la compilation
`$./AppSystem`

- Vérifier que le processus Appsystem est dans l'arbre des processus encours d'exécution donnée par **pstree**
- Noter le processus init, processus privilégié créé au démarrage du système (/sbin/init)

« Ancêtre » de tous les processus »

- Noter également, le processus AppSystem généré par la commande pstree dans les processus encours d'exécution.
- Relancer le shell, puis faites la commande **pstree**.
- Le processus AppSystem est t-il dans l'arbre des processus?Sinon pourquoi?

Exercice 2: La fonction `fork()`
Sous Unix, quand un processus crée un autre, deux possibilités existent:
- Les deux processus s'exécutent parallèlement
- Le processus créateur attend la fin du processus créé.

La fonction `fork()` permet à un processus de se dupliquer. La représentation en mémoire du processus enfant créé est une copie exacte du parent par simple recopie de la zone mémoire associée, donc les deux processus exécutent le même code. La valeur de retour de `fork()` permet de différencier les deux processus: elle vaut 0 pour le processus enfant et elle est le numéro d'identification du processus enfant (PID ou l'Identification du Processus) dans le processus parent.
Au retour d'une fonction `fork()`,
- La fonction `getpid()`, permet d'obtenir le PID,
- La fonction `getppid()`, permet d'obtenir le numéro d'identification du parent (le PPID)

Après l'exécution réussie de fork(),
deux processus indépendants existent
– Le « père » qui reçoit la valeur du PID du fils
– Le « fils » (copie du « père ») qui reçoit la valeur 0
• Ils exécutent le même programme : leurs exécutions continuent
avec l'instruction qui suit fork()
Tester l'exemple du fichier fork.c suivant et vérifier s'il est déterministe ? Autrement dit est ce que toute exécution affichera toujours les mêmes résultats ?

```c
#include <stdio.h>/*pour printf()*/
#include <unistd.h>/*pour fork(), getpid() ...*/
//#include <sys/wait.h> pour wait
int main ( void ) {
pid_t pid;
pid = fork ();
if ( pid == 0 ) {
printf ("je suis le fils %d (père %d)\n",
getpid (), getppid () );
} else {
printf ("je suis le père %d (fils %d)\n",
getpid (), pid );
//sleep(1);/*attendre une seconde*/
}
return 0;
}
```

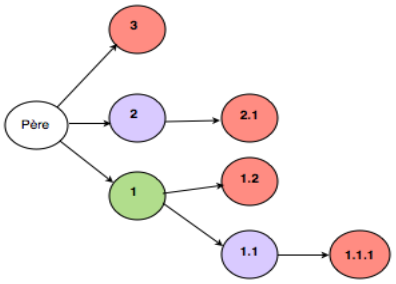
Le résultat est imprévisible si on commente l'instruction sleep(1) ; dans ce cas, deux cas sont possibles selon l'ordonnaceur :
-Le fils se termine avant le père.
-Le père se termine avant le fils, dans ce cas, le processus init de PID=1 adopte le processus.
NB :Il faut noté que l'exécution du code du processus fils commence à partir de l'instruction qui suit le *fork()* à savoir le test `if(pid==0)`, mais noté que la valeur de la variable *pid* dans le processus fils est à 0, et elle est égale au PID du fils dans le processus père.

Exercice 3: la fonction forkN()
Créer le fichier `forkn.c` suivant:

```c
/*forkN.c fait une boucle où est appelé fork()*/
/*Usage de forkN N ou N est le nombre de tour de boucle */
#include <stdio.h>/*pour prntf()*/
#include <unistd.h>/*pour fork(), getpid() ...*/
#include <stdlib.h>/*pour atoi()*/
//#include <sys/wait.h> //pour wait

int main(int argc, char **argv)
{
int i,N;
pid_t pid;
printf("le programme %s de numéro d'identification %d\n", argv[0],
getpid());
printf("est lancé depuis le shell de numéro d'identification %d\n",
getppid());
N = atoi(argv[1]);
for (i=0;i<N;i++)
{
pid=fork();
switch(pid)
{
case -1:
printf("fork a agendré une erreur\n");
perror("fork :");
break;
case 0:
printf("Ici enfant numéro= %d de parent %d\n", getpid(), getppid());
break;
default:
printf("Ici processus %d qui a engendré un enfant de numéro %d\n",
getpid(), pid);
}
}
sleep(1); /*attendre une seconde*/
//wait(NULL);
return 0;
}
```

Le fichier stdlib.h contient le prototype de la fonction `atoi()` qui permet de transformer une chaîne de caractères en un entier : `int atoi (char *chaine) ;`
La fonction retourne un entier et prend en argument une chaîne de caractère.
Compiler le programme avec:
`$gcc -o forkn forkn.c`
Exécuter le programme avec un paramètre 1 puis noter les résultats obtenus.
`$./forkn 1`
Lancer le programme avec un argument de 3 par la commande:
`$./forkn 3`
Dessiner l'arbre des processus du programme.



`sleep() /*Attendre la fin de chaque enfant*/`

Exercice 4 : Mémoire et fork()
Exécuter le programme memo.c suivant et décrire son déroulement. Quelles en sont les sorties possibles ?

```c
#include <stdio.h>/*pour printf()*/
#include <unistd.h>/*pour fork(), getpid() ...*/
int main(int argc, char **argv) {
int var=20;
if (fork() == 0 ) {
var=var+10;
}
else {var=var+30;
```

```
}
printf ("la variable vaut %d\n", var);
return 0;
}
```

Exercice 5 : Mémoire et fork (Suite)

Exécuter le programme memo1.c suivant et décrire son déroulement. Quelles en sont les sorties possibles ?

```
#include <stdio.h>/*pour printf()*/
#include <unistd.h>/*pour fork()*/
#include <sys/wait.h> pour wait
int main(int argc, char **argv) {
int var=20;
pid_t pid;
        if (fork() == 0 )
                {
                var=var+10;
                pid = fork();
                }
        else{
                pid = fork();
                var=var+30;
                }
        printf ("la variable vaut %d\n", var);
sleep(1); /*attendre une seconde*/
        return 0;
}
```

Exercice 6 : Une bombe de fork()

Le programme suivant bombe.c crée un très grand nombre de processus très rapidement, sature l'espace mémoire disponible et le processeur. Le SE devient alors de plus en plus lent, et probablement ne sera plus utilisable. C'est le principe de certains virus qui se répliquent tout seul. La seul possibilité sera de rebooter le système (éteindre et rallumer). Les préventions usuelles contre ce genre d'attaque se contentent de limiter le nombre de processus qu'un utilisateur peut créer.

```
#include <stdio.h>/*pour printf()*/
#include <unistd.h>/*pour fork(), getpid() ...*/
int main(int argc, char **argv) {
while(1)
{fork();
}
return 0;
}
```

Exercice 8: Sémaphore (Producteur-Consommateur)

```c
#include <stdio.h>/*pour printf()*/
#include <unistd.h>/*pour sleep()*/
#include <stdlib.h>/*pour rand()*/
#include <semaphore.h> //pour wait
#include <pthread.h> //pour wait

/* semaphore partages par tous les processus*/

sem_t mutex; /* control section critique*/
```

```c
sem_t vide; /*nb places libres*/
sem_t plein; /*nb places pleines*/

#define N 3/* nb d'emplacements*/
#define VRAI 1

pthread_t prod, cons; /*les threads*/
int tampon[N]; /*le tampon paratagé, un tableau de N places*/
int libre=0,prochain=0;/*les places dans le tampon*/
/*********************Producteur************/
void * producteur(void* arg)
{
int objet;/*un objet de type arbitraitre, ici un entier*/
while(VRAI)
      {
      objet=1+(int)(100.0*rand()/(RAND_MAX+1.0));/*produire un entier aléatoire*/
      sem_wait(&vide);/*décrément des places libres*/
      sem_wait(&mutex);/*section critique*/
      tampon[libre]=objet;/*utilisation ressource*/
      printf("Production:tampon[%d]=%d\n", libre, objet);
      libre=(libre+1)%N;
      sem_post(&mutex);/*fin section critique*/
      sem_post(&plein);/*incrément les places occupées*/
      }
return (NULL);
}
/***************Consommatuer**************/
void * consommateur(void* arg)
{
int objet;/*un objet de type arbitraitre, ici un entier*/
while(VRAI)
      {
      sem_wait(&plein);/*décrément des places occupées*/
      sem_wait(&mutex);/*section critique*/
      objet=tampon[libre];/*utilisation ressource*/
      printf("Consommation:tampon[%d]=%d\n", prochain, objet);
      prochain=(prochain+1)%N;
      sem_post(&mutex);/*fin section critique*/
      sem_post(&vide);/*incrément les places vides*/
      sleep(2);/*pour ralentir le consommateur*/
      }
return (NULL);
}
int main(int argc, char **argv)
{
/*Initialisation des sémaphore*/
sem_init(&mutex, 0, 1);/*mutex: semaphor binaire initialisé à 1*/
sem_init(&vide, 0, N);/*vide: semaphore n-aire initialisé à N (N places vides)*/
sem_init(&plein, 0, 0);/*plein: semaphore n-aire initialisé à 0 (0 place pleine)*/
/*Creation des thread*/
pthread_create(&prod, NULL, producteur, NULL);
pthread_create(&cons, NULL, consommateur, NULL);
/*attendre la fin des threads*/
pthread_join(prod,NULL);
pthread_join(cons,NULL);
return 0;
}
```

Compiler avec :

$gcc –wall –o semaph semaph.c -lpthread