UNIVERSITE A.MIRA-BEJAIA
FACULTE DES SCIENCES EXACTES
DEPARTEMENT INFORMATIQUE

TRAVAUX PRATIQUES
PROGRAMMATION SYSTEME
Série 1 : Gestion de Processus

Exercice1 : La primitive fork() : Un processus peut se dupliquer – et donc créer un nouveau processus – par la fonction : `int fork(void)`.
Cette fonction permet la création dynamique d'un nouveau processus qui s'exécute de façon concurrente avec le processus qui l'a créé et qui est une copie conforme.
Cette fonction rend (par le pid_t contenu dans le header file <sys/types.h>) :
• -1 en cas d'échec,
• 0 dans le processus fils,
• Le PID du processus fils dans le père.

Exemple:

```
#include <stdio.h>
int main( void ) {
      int pid = fork();
      if ( pid == 0 ) {
            printf( "C'est le processus fils qui affiches\n" );
      } else {
            printf( "C'est le processus père qui affiche:\n"
                    " Le pid du fils est: %d\n", pid );
      }
      return 0;
}
```

Q1) Dérouler le programme et expliquer chacune des instructions.
Q2) Dérouler le programme plusieurs fois et vérifier si le processus père s'exécute avant le processus fils.
Q3) Ecrire un programme qui crée un nouveau processus et qui affiche pour les deux processus (père et fils) les caractéristiques générales d'un processus :
• Identifiant du processus.
• Identifiant du père du processus.
• Répertoire de travail du processus.
Il faudrait utiliser ce qui suit (consulter le manuel man pour plus d'informations) :

```
#include <unistd.h>
#include <sys/types.h>
int getpid(void); /* pid courant*/
int getppid(void); /* pid du père */
int chdir(const char *chemin); /* changer le répertoire de travail*/
char * getcwd(char * buf, unsigned long taille); /* récupérer le
chemin absolu du répertoire de travail courant taille=256.*/
```

Exercice2 : Les primitives Exit() et Wait() : La fonction **exit** met fin au processus qui l'a émis, avec un code de retour **status** :
#include <stdlib.h>

```
void exit (int status)
```

Si le processus à des fils lorsque **exit** est appelé, ils deviennent des « zombies », le pid de leur processus père est changé en 1, qui est l'identifiant du processus init. Le processus 1 et le

processus 0 sont chargés de l'ordonnancement des processus. Le père du processus qui effectue un **exit** reçoit son code retour à travers un appel à **wait :** `int wait (int * terminaison)`.
On ajoutera les deux fichiers en-tête suivants : `<sys/types.h>` et `<sys/wait.h>`.
Un processus exécutant l'appel système **wait** est endormi jusqu'à la terminaison d'un de ses fils. Lorsque cela se produit, le père est réveillé et **wait** renvoie le PID du fils qui vient de mourir.
Q1) Ecrire un programme qui positionnes une attente de quelques secondes (`sleep(10)`) dans le processus fils pour que le processus père se termine avant le fils.
Q2) Puis ajouter les instructions qui permettront au processus père d'attendre la terminaison de son fils et qui afficheront le code retour de celui-ci.

Exercice3 : Les primitives de la famille exec() :
Lorsqu'un processus exécute un appel exec, il charge un autre programme exécutable en conservant le même environnement système. La partie de code qui suit l'appel d'une primitive de la famille exec ne s'exécute pas, car le processus où elle se trouve est remplacé par un autre.
Les deux fonctions de base sont **execl et execv.**

```
int execl (char *path, char *arg0, char *arg1 ... );
int execv (char *path, char * argv[ ] );
```

Dans le cas de **execl**, les arguments de la commande à lancer sont fournis sous la forme d'une liste terminée par un pointeur nul :
• **path** est une chaîne de caractères donnant le chemin absolu du nouveau programme à substituer et à exécuter.
• **arg0, arg1, …, argn** sont les arguments du programme.
• Le premier argument, **arg0**, reprend en fait le nom du programme.

Exemple :

```
void main(){
execl("/bin/ls", "ls", "-l", NULL);
printf("Erreur lors de l'appel à ls \n");}
```

Q) A l'aide des primitives fork et execl, écrire le programme qui permet de faire exécuter la commande ps avec l'option –l depuis un processus existant. Ce dernier devra se mettre en attente du code retour de la commande ps.

Exercice4 : Gestion de threads : Les threads sont définis comme des processus dans la bibliothèque `<pthread.h>` mais sont attachés au processus qui les a créés, leur ID est donné par la variable `pthread_t` et peut être récupéré par la primitive `pthread_self()`. La compilation doit inclure la bibliothèque `-l pthread`.
Primitive de creation: `int pthread_create(pthread_t *thread, const pthread_attr_t, *attr, void *(*start_routine)(void *), void *arg)`.
Primitive de terminaison: `pthread_exit(void *ret)`: **ret** est la valeur retourné, qui peut être récupéré par un autre thread en exécutant `pthread_join(pthread_t thread, void **retour)`. La primitive join est bloquante.
Les attributs d'un thread sont définis dans `pthread_attr_t` et contient : l'adresse de départ et la taille de sa pile, la politique d'ordonnancement associée, sa priorité, son attachement ou son détachement. Ces attributs (xxx) peuvent être modifiés par les primitives `pthread_attr_init()`, `pthread_attr_getxxx()` , `pthread_attr_setxxx()`.

Q1) Ecrire un programme qui permet de créer 3 threads et affichent leur ID.
Q2) Ecrire un programme à trois threads qui font appel à une fonction qui modifie une variable i partagée par tous.

-FIN-

UNIVERSITE A.MIRA-BEJAIA
FACULTE DES SCIENCES EXACTES
DEPARTEMENT INFORMATIQUE

TRAVAUX PRATIQUES
PROGRAMMATION SYSTEME
Série 2 : Gestion de Signaux

Un signal est un message envoyé par le noyau à un processus ou à un groupe de processus pour signaler un événement survenu au niveau du système (touche clavier CTRL-C par exemple). Le noyau Linux dispose de 64 signaux différents. Chaque signal est identifié par un numéro et est décrit par un nom préfixé de SIG. Seul le signal 0 ne comporte pas de nom, les signaux de 1 à 31 correspondent aux signaux classiques et les signaux de 32 à 63 correspondent aux signaux temps réel. Pour avoir une liste des signaux et leur description sur le shell : `man 7 signal` .

Exerice1 : Primitive `signal()` : La prise en compte du signal par le processus s'effectue à travers l'exécution d'une fonction de gestion du signal appelé handler de signal grâce à la primitive **signal**.

```
#include <signal.h>
void (*signal(int num-sig, void (*fonc) (int))) (int) ;
```

Le handler *fonc est attaché au signal num-sig. La valeur de *fonc peut etre :
- SIG_DFL : l'action par défaut est utilisé
- SIG_IGN : le signal est ignoré (sauf SIGKILL)
- un pointeur vers une fonction définie dans le code. Cette fonction a comme argument un entier num-sig et ne peut renvoyer aucune valeur.

Question : Dérouler le programme suivant et en expliquer chaque ligne.

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

void sighandler(int);

int main () {
   signal(SIGINT, sighandler);

   while(1) {
      printf("Je vais dormir une seconde...\n");
      sleep(1);
   }
   return(0);
}

void sighandler(int signum) {
   printf("J'ai capture le signal %d, je me reveille...\n", signum);
   exit(1);
}
```

Remarque : Une autre primitive plus complexe permet aussi la prise en compte d'un signal : `sigaction()`.

Exercice2 : la primitive `kill()` : Un processus envoie un signal à un autre processus en appelant la primitive **kill(**) :

```
#include <signal.h>
int kill (pid_t pid, int num-sig) ;
```

- Si pid>0 le signal num-sig est délivré au processus pid.
- Si pid=0 le signal est délivré à tous les processus du groupe du processus pid
- Si pid=-1 le signal est envoyé à tous les processus du système sauf le processus 1 et le processus appelant.
- Si pid<0 le signal est envoyé à tous les processus dont le PGID=|pid|.

Question : Ecrire un programme à deux processus tels que le premier réveille le deuxième qui dormait.

Exercice3 : La primitive `pause( )`

La primitive pause permet à un processus de bloquer en attendant un signal (n'importe lequel !).

```
#include <unistd.h>
int pause (void) ;
```

Question : Ecrire un programme qui fait endormir un processus jusqu'au réveil par un CTRL-C.

Exercice4 : La primitive `alarm( )`

La primitive alarm( ) permet d'armer une alarme et qui se déclenchera, à la fin d'un délai, par la réception d'un signal SIGALRM. L'opération alarm(0) annule une temporisation précédemment armée. L'alarme est utilisée pour limiter les temps d'attente (pause!).

```
#include <unistd.h>
unsigned int alarm (unsigned int nbsec) ;
```

Question : Réécrire le programme de l'exercice précédent en incluant une alarme qui limite le temps d'attente.

Note : Liste des signaux de Linux :
SIGALRM (14) : Alarme
SIGKILL(4) : processus tué (instruction non valide)
SIGINT(2) : CTRL C
SIGTSTP(20) : CTRL Z
SIGCONT(18) : reprise d'un processus
SIGQUIT(3) : CTRL \ (interruption clavier avec sauvegarde de l'image mémoire)
SIGFPE(8) : erreur arithmétique (division par 0)
SIGCHLD (17) : envoyé par le noyau lors de la mort d'un fils

-FIN-

UNIVERSITE A.MIRA-BEJAIA
FACULTE DES SCIENCES EXACTES
DEPARTEMENT INFORMATIQUE

TRAVAUX PRATIQUES
PROGRAMMATION SYSTEME
Série 3 : Synchronisation entre processus

I. Semaphores Posix de Linux est une interface portable de système d'exploitation, et le **X** exprime l'héritage UNIX (*Portable Operating System Interface).*

```
#include <semaphore.h>
```

- Pour declarer un objet semaphore: `sem_t sem;`

-La primitive sem_init() sert à initialiser le sémaphore. L'argument pshared doit être 0 pour le sémaphore local à un processus (ou les threads):
`int sem_init(sem_t *sem, int pshared, unsigned int value);`

- La primitive sem_wait() est l'equivalent de l'operation P : `int sem_wait(sem_t * sem);`

- La primitive sem_post() est l'équivalent de l'operation V : `int sem_post(sem_t * sem);`

- Pour récuperer la valeur d'un semaphore: `int sem_getvalue(sem_t *sem, int *valp);`

- La primitive sem_destroy() est utilisé pour libérer les ressources occupées par le sémaphore :
`int sem_destroy(sem_t * sem);`

Question : Reprendre l'exercice sur la gestion des threads (série1) et ajouter des sémaphores pour protéger la variable i.

II. Verrou mutex : C'est une variable de type pthread_mutex_t pour la synchronisation des threads.
Création et initialisation : `int pthread_t_mutex_t verrou=PTHREAD_MUTEX_INITILIAZER ;`

Primitive lock( ) bloquante: `int pthread_mutex_lock(pthread_mutex_t *verrou);`

Primitive trylock( ) non bloquante:
`int pthread_mutex_trylock(pthread_mutex_t *verrou);`
Si le verrou est déjà verrouillé, le thread appelant reçoit `EBUSY` comme erreur.

Primitive unlock( ) : `int pthread_mutex_unlock( pthread_mutex_t *verrou);`

Primitive destroy( ): `int pthread_mutex_destroy(pthread_mutex_t, *verrou);`

Question: même que précédemment mais avec des verrous.

-FIN-