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 n° du processus fils (PID) 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.
• Le propriétaire réel.
• Le propriétaire effectif.
• Le groupe propriétaire réel.
• Le groupe propriétaire effectif.
Avant d'afficher ces informations, on prendra bien soin de préciser si l'on est dans le processus père ou fils.
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.*/
int getuid(void); /* Id du propriétaire réel */
```

```
int getgid(void); / ID du groupe du propriétaire réel */
int geteuid(void); /* propriétaire effectif */
int getegid(void); /* groupe du propriétaire effectif */
```

Exercice2 : La primitive 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 chargé 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-tete 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 : La primitive system( ) :
La fonction **system** lance l'exécution d'un processus SHELL interprétant la commande passée en argument dans la chaîne de caractères **ch** : `int system (char *ch)`
Cette fonction crée pour cela un nouveau processus, qui se termine avec la fin de la commande. Le processus à l'origine de l'appel de **system** est suspendu jusqu'à la fin de la commande.
Q1) Ecrire un programme qui lance la commande shell « sleep 60 » à l'aide de la primitive system.
Q2) Lancer l'exécutable correspondant en arrière plan (en utilisant &) et taper la commande 'ps – l'. Expliquer le résultat obtenu.

Exercice4 : 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[ ] );
```

Il existe d'autres fonctions analogues dont les arguments sont légèrement différents :
execle (), execlp (), execv (), execcve (), execvp ().
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");
}
```

Dans le cas de **execv**, les arguments de la commande sont sous la forme d'un vecteur de pointeurs (de type argv [ ]) dont chaque élément pointe sur un argument, le vecteur étant terminé par le pointeur NULL :

Exemple :

```
#include <stdio.h>
#define NMAX 5
void main () {
char *argv [NMAX];
argv [0] = "ls";
argv [1] = "-l";
argv [2] = NULL;
execv ("/bin/ls", argv);
printf("Erreur lors de l'appel à ls \n");
}
```

Q1) 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.
Q2) Reprendre le programme précédent mais en utilisant la commande execv.

Exercice5 : Gestion de threads

Les threads sont définis comme des processus dans la bibliothèque `<pthread.h>`mais sont attachés au processus qui les a crées, 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-