TME1 : Programmation Concurrente, généralités

Exercice 1 – Implantation de processus et threads

Question 1
En utilisant l'appel système `pid_t fork(void);`, créer deux processus affichant la valeur de retour de l'appel à `fork`.

Question 2
Ecrire maintenant un programme possédant une variable entière `i`, créant un processus fils et tel que le père affiche un message indiquant qu'il est le père et demande à l'utlisateur de saisir une valeur pour `i` au clavier et tel que le fils commence par dormir 4 secondes, puis affiche le contenu de `i`.

Question 3
En utilisant la fonction `pid_t getpid(void);` (`pid_t` est compatible avec `int`), écrire le même programme avec des `threads` POSIX sauf que le père attend cette fois ci la fin du fils avant de terminer. Que constatez vous quant au contenu de la variable `i` dans le fils ? Qu'en deduisez vous sur le fonctionnment des threads Posix ? Quelle précautions sont à prendre pour éviter au programme précédant de produire un résultat indéfini ?

Exercice 2 – Compteur partagé

Question 1 – Architecture de base
Ecrire un programme utilisant l'API des `pthread` effectuant la tâche suivante :

Le programme principal initialise deux variables entières `temp` et `SHARED_compteur` à 0 puis lance un nombre `NB_THREAD` de pthread éxecutant la routine suivante :

- lire la valeur de la variable partagée dans `temp`
- rendre la main à l'ordonnanceur (`usleep, sched_yield`)
- incrémenter la valeur de `temp`
- rendre la main à l'ordonnanceur
- incrémenter la variable `SHARED_compteur`

Une fois les `pthreads` lancés, le programme principal attend tant que la valeur de `SHARED_compteur` soit `NB_THREAD` puis il affiche `"TERMINE"`

Dans un premier temps, ne pas synchroniser le programme.

Question 2 – Terminaison et jonction
Le programme principal utilise une boucle qui vérifie la valeur du compteur. Cela s'appelle une attente active qui occupe inutilement le processeur. Pour détecter la terminaison des threads, on peut utiliser le mécanismes de jonction. Ecrire une variante du programme avec jonction.

Remarque:
Les ressources allouées à un thread ne sont libérées que dans quelques cas :

- Le thread principale prend fin.

- On fait un `pthread_join`
- Le thread fini est dans un état `detached`, auquel cas son code de retour est inaccessible.

Le programme affiche-t-il tout le temps "TERMINE" ? Si non, donnez une exécution (avec `NB_THREAD=2`) qui n'affiche rien

Exercice 3 – Producteurs et Consommateurs

Le patron Producteur/Consommateur est un des patrons les plus courants dès lors qu'on utilise des threads. On le retrouve dans de très nombreuses applications :

- Jeux : les routines d'IA modifient l'aire de jeu et la routine d'affichage affiche le nouvel univers
- Sommation : Des routines auxiliaires calculent de éléments de la somme totale, et une routine regroupe les différents éléments
- ...

Question 1 – Architecture de base

La patron s'articule autour de deux ensembles :

- une équipe de producteurs qui produit des données
- une équipe de consommateurs qui les consomme.

Les deux équipes communiquent entre elles par une zone de transfert. Dans cet exercice on utilisera un `FIFO` de taille fixe comme Buffer de communication entre les équipes. Une implantation simple de cette structure de donnée se fait en utilisant un tableau de taille fixe et deux pointeurs dans ce tableau : l'un pour désigner le prochain endroit où on pourra consommer une donnée et l'autre pour désigner le prochain endroit où mettre une donnée.

Dans notre exercice, on enregistre un nombre `BUF_SIZE` maximum de données de type `int`. On écrira une fonction `Buffer* CreerBuffer()` pour créer un buffer.

On écrira en outre les fonctions auxiliaires :

- `void EcrireBuffer(Buffer *buf, int val)` pour ajouter un élément (afficher "P" sur la sortie d'erreur si le buffer est plein).
- `int LireBuffer(Buffer *buf)` pour lire le plus ancien élément (si disponible, afficher "V" sur la sortie d'erreur si le buffer est vide)

Ecrire les fonctions mentionées ci-dessus

Question 2

Ecrire deux fonctions qui seront exécutées respectivement par les threads producteurs et les threads consommateurs :

- Une fonction de prototype : `void * THREAD_Producteur(void * arg)`, implantantant la tache dévolue au producteur, i.e. produisant `NB_TURN` entiers à intervalle régulier (CF commande `usleep` pour la temporisation)
- Une fonctions de prototype : `void * THREAD_Consommateur(void * arg)`, implantant la tâche dévolue au consommateur, i.e. consommant des entiers à intervalle régulier (CF commande `usleep` pour la temporisation) et affichant les entiers consommés.

Ecrire cette premiere version sans synchronisation

En redirigeant la sortie d'erreur vers `/dev/null`, donnez un exemple d'affichage du programme.

Quels sont les problèmes posés par ce programme ?

Rappel : API POSIX Thread (Extrait)

Quelles sont les fonctionalités de base de l'API POSIX PThread
cf. Rappels Posix Threads

Creation de thread

```
int pthread_create(pthread_t  *  thread,
                   pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void * arg);
```

Exemple :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* THREAD_Routine(void *arg)
{    /* routine de thread*/
  return NULL;
}

int main(void)
{
  pthread_t thread_id;
  int ok;

  ok = pthread_create(&thread_id, NULL, THREAD_Routine, /* arg */ NULL);
  if(ok!=0)
    {
      fprintf(stderr,"Impossible de creer de le thread\n");
      exit(EXIT_FAILURE);
    }
}
```

Destruction de thread

```
void pthread_exit(void *retval);
// note: retval est disponible dans pthread_join
```

Identification

```
  pthread_t pthread_self(void);
  int pthread_equal(pthread_t thread1, pthread_t thread2);
```

Jonctions

```
int pthread_join(pthread_t th, void **thread_return);
```