Programmation Temps Réel
TP1 = synchroniser des threads avec les sémaphores

A - Consignes générales

Réalisation des exercices :

- Les programmes doivent être réalisés en langage C et tourner sous linux.
- Privilégier un style clair et lisible
- Nommer les fonctions et les variables avec des noms appropriés
- Commentez votre code
- Soignez votre indentation
- **Chaque exercice doit faire l'objet d'un programme en C séparé (fichier .c).**
- **Les réponses aux questions doivent être placées sous forme de commentaires au début du fichier.c** (contenant le main) de l'exercice correspondant.

Remise des exercices :

- Chaque fichier doit être nommé avec le numéro de l'exercice (ex: "exo1" ou "exercice1").
- **Déposez les fichiers dans le dépôt moodle prévu à cet effet sur https://moodle.univ-paris8.fr/moodle/course/view.php?id=1373**

B - Rappels

Utiliser la commande "man" ou cherchez dans le manuel sur internet pour en savoir plus sur une fonction.

```
// headers à importer
#include <pthread.h>   // pour les threads
#include <semaphore.h> // pour les semaphores
#include <fcntl.h>     // pour les flags O_CREAT, O_EXCL, ...

// sémaphore anonymes, non persistants ("deprecated" sous MacOS)
int sem_init (sem_t * sem, int pshared, unsigned int value);
int sem_destroy (sem_t * sem);

// sémaphore nommés et persistant créés dans le répertoire /dev/shm sous linux
sem_t * sem_open(const char* name, int oflag, ...);
// oflag = liste de flag séparés par des OU logiques
// O_CREAT = crée le sémaphore si il n'existe pas déjà
// O_CREAT | O_EXCL = erreur si le sémaphore existe déjà
// 2 arguments optionnels :
//    1 - les droits = S_IRUSR | S_IWUSR
//                     (droit de lecture et écriture pour l'utilisateur)
//    2 - la valeur initiale du sémaphore
int sem_close (sem_t * sem); // signaler que le sémaphore n'est plus utile
int sem_unlink (const char * name); // détruire le sémaphore

// pour le deboggage = afficher la valeur du sémaphore
int sem_getvalue(sem_t *sem, int *sval);

// pour verrouiller/deverrouiller un semaphore (anonyme ou non)
int sem_wait (sem_t * sem); // primitive P()
int sem_post (sem_t * sem); // primitive V()

// autres fonctions pour demander le sémaphore
int sem_trywait (sem_t * sem);
int sem_timedwait(sem_t * sem, const struct timespec * abs_timeout);

// Compiler votre code avec les options -Wall et -pthread :
$ gcc -Wall exercice.c -pthread
```

La compilation ne doit retourner ni erreur, ni warning.

C - Énoncés des exercices

Exercice 1 - threads non synchronisés

Ecrire un programme avec 1 thread principal, le processus lourd, affichant A sur la sortie standard et 2 threads secondaires affichant respectivement B et C sur la sortie standard. Chaque thread effectue 100 itérations. Le résultat attendu est un ensemble de 300 lettres regroupant 100 A, 100 B et 100 C sans notion d'ordre.

Remarque 1 : vous pouvez ou réutiliser le code donné en cours comme base de départ.

Remarque 2 : Pour avoir un affichage concis, utilisez `printf()` sans retour à la ligne et utilisez `fflush(stdout)` pour forcer l'affichage du contenu du tampon (sans attendre un retour à la ligne ou que le tampon soit plein).

Exercice 2 - synchronisation avec des sémaphores anonymes

Faites une copie du fichier précédent et utiliser un sémaphore anonyme pour synchroniser les threads et afficher les lettres en une séquence ordonnée (ABC)*

Exercice 3 - synchronisation avec des sémaphores nommés

Faites une copie du fichier précédent et utilisez maintenant un sémaphore nommé pour obtenir le même résultat. **Question : Que se passe-t-il si on omet sem_unlink() à la fin du programme ? Décrivez vos observations et expliquez les.**

Pour pouvoir répondre, commentez les lignes appelant sem_unlink(), re-compilez, lancez votre programme et vérifiez le contenu du répertoire /dev/shm/ dans les deux cas (présence ou non de sem_unlink()).

Exercice 4 - sémaphore persistant

Faites une copie du code écrit pour la question 3 n'utilisant pas sem_unlink() et modifiez votre programme. Faites en sorte que les sémaphores soient créés uniquement à la première exécution du programme puis réutilisés (on teste si ils existent déjà sinon on les crée et on les initialise à 0). Ajoutez un appel à sem_post() pour débloquer un sémaphore et lancer l'écriture du premier A.

Question : Que se passe-t-il quand on relance plusieurs fois le programme ? Décrivez et expliquez le comportement de votre programme.

Exercice 5 - programme plus général

Créer un programme qui reçoit un nombre N en argument et lance N threads affichant chacun un des nombres de 1 à N de manière synchronisée (avec des sémaphores nommés ou anonyme au choix) pour produire une séquence ordonnée (12...N)*. Ici, il est nécessaire d'utiliser une seule et même fonction exécutée par tous les threads et de passer le numéro à afficher en argument au lancement du thread.

Exercice 6 - synchronisation avec des sémaphores

A partir d'une copie du code écrit pour la question 2 ou 3, ajouter un thread affichant D sur la sortie standard. Utilisez des sémaphores anonymes ou nommés (selon votre préférence) pour synchroniser les threads et afficher les lettres en une séquence ordonnée (AAABCCD)*, les threads affichant respectivement 300 A, 100 B, 200 C et 100 D.

Exercice 7 - (optionnel) sémaphores et processus lourds

Refaire l'exercice 6 avec fork et des processus lourds.
Rappel sur l'utilisation de fork :

```
#include <unistd.h>

pid = fork ();    // le processus lourd est dupliqué
if (pid > 0) {
      // le processus père exécutera ce code
      int status;
      waitpid(pid, &status, 0);     // le parent attend la fin de son enfant
} else if (pid == 0) {
      // le processus fils exécutera ce code
} else {
      // ce code sera exécuté si fork retourne une erreur
}
```