TP 1

Systèmes temps réel

Le but du TP est de se familiariser avec les concepts de synchronisation et les problèmes courants lors d'une exécution dans un environnement multitâche. N'oubliez pas de bien lire toutes les notes d'une question avant d'y répondre.

1 Exercice préliminaire

Question 1. Ecrivez un programme composé de deux threads (lancés par votre fonction main). Le premier thread devra faire deux affichages : "systeme "et "-". Le deuxième fera également deux affichages : "temps" et "reel". Utilisez des sémaphores pour que l'exécution du programme produise l'affichage de "systeme temps-reel" à l'écran quelque soit l'ordre de lancement des threads.

Notes

- Utilisez la bibliothèque `<pthread.h>` pour créer les threads et la bibliothèque `<semaphore.h>` pour la gestion des sémaphores.
- Pour plus de lisibilité vous redéfinirez les primitives `sem_wait()` et `sem_post()` par des macro-fonctions `P()` et `V()`;
- Utilisez le flag -lpthread pour pouvoir compiler le programme.

2 Réveil d'un thread sur un évènement

Le problème consiste à écrire un programme permettant à N threads de réveiller un thread en particulier. Votre programme sera constitué de deux types de thread : les réveilleurs et celui à réveiller. Le thread à réveiller doit se réveiller une fois que tous les threads réveilleurs lui ont signalé le même évènement (par l'intermédiaire d'un sémaphore).

Question 2. Implémentez le programme en question.

Notes

- Le nombre de thread total et le nombre de threads réveilleurs doit être facilement modifiable (macro-constantes);
- Le programme doit vérifier à la compilation que le nombre de thread réveilleurs est bien inférieur au nombre de thread total (provoque une erreur de compilation dans le cas contraire).
- L'ordre d'affichage doit être indépendant de l'ordre de lancement / exécution des threads.

3 Tampon de Lecture/Ecriture

Le problème est composé de deux tâches et d'un tampon partagé de taille N. Une première tâche "ecrivain" appelle une fonction `ecrire_tampon(data)` chargée d'écrire dans le tampon partagé. La deuxième tâche "lecteur" appelle une fonction `lecture_tampon()` chargée de lire une donnée précédemment écrite par la tâche "ecrivain". Le tampon sera un simple tableau de N cases.

Question 3. Décrire de façon simple chaque thread à l'aide des primitives P et V et des sémaphores de la question précédente.

Notes
- Il n'est pas nécessaire de faire apparaître la gestion du buffer circulaire pour cette question.

Question 4. Combien de sémaphores sont nécessaires pour l'implémentation de ce type de problème? Préciser également le ou les valeurs initiales.

Question 5. Implémentez le problème en question. Les tâches seront implémentées sous forme de thread.

Notes
- Le tampon est circulaire pour traiter les données dans l'ordre de production;
- Une lecture et une écriture doivent pouvoir se faire en parallèle si c'est pour des cases différentes;
- Pour plus de lisibilité vous redéfinirez les primitives `sem_wait()` et `sem_post()` par des macro-fonctions `P()` et `V()`;
- Utilisez le flag -lpthread pour pouvoir compiler le programme.

4 Aérodrome

L'exercice consiste à modéliser (très succinctement) un aérodrome possédant une *unique piste* d'atterrissage et de décollage. L'aérodrome est composé d'une *zone d'attente* (limitée) proche de la piste permettant aux avions d'attendre leur tour pour décoller et d'une *zone d'approche* (limitée) pour les avions souhaitant atterrir. Ce système peut être représenté par deux files d'attente et quatre tâches (threads). Quand un avion se trouve dans une ces files d'attente, il n'aura pas encore décollé ou atterri. Il aura effectivement décollé ou atterri lorsqu'il disparaitra d'une des files d'attente.

Files d'attente:

- `Air[]` de taille `N` contenant les avions souhaitant atterrir représentant la zone d'approche;
- `Sol[]` de taille `M` contenant les avions souhaitant décoller représentant la zone d'attente de décollage.

Threads:

- `SortirAvion` qui sort les avions du hangar et les place dans le buffer d'attente de décollage;
- `Decollage` qui prend un avion du buffer d'attente et le fait décoller en utilisant la piste;
- `AmenerAvion` qui place des avions dans le buffer d'attente d'atterrissage;
- `Atterrissage` qui prend un avion dans le buffer d'attente d'atterrissage et le fait atterrir en utilisant la piste.

Question 6. Quels sont les problèmes de type "producteur/consommateur" et de type "exclusion mutuelle" ? Préciser ensuite le nombre de sémaphore/mutex à utiliser ainsi que leurs valeurs initiales.

Question 7. Décrire de façon simple chaque processus à l'aide des primitives P et V et des sémaphores de la question précédente.

Notes
- Il n'est pas nécessaire de faire apparaître la gestion des files d'attente pour cette question.

Question 8. L'ordre de prise des sémaphores est-il important? Si oui, précisez le(s) problème(s) que cela peut engendrer.

Question 9. Supposons que le nombre d'avions souhaitant atterrir et décoller est infini. Est-ce que le système peut fonctionner correctement avec n'importe quelle politique d'ordonnancement ? Précisez le(s) problème(s) qui pourraient apparaître dans certains cas.

Question 10. Implémentez le problème en utilisant la bibliothèque `<pthread.h>` pour créer les deux processus et la bibliothèque `<semaphore.h>` pour la gestion des sémaphores.

Notes
- Deux variables représenteront le nombre d'avions à faire atterrir et le nombre d'avions à faire décoller;
- En ce qui concerne la gestion des files d'attentes vous pourrez utiliser le travail effectué dans la première partie;
- Définition et gestion d'un mutex avec la bibliothèque `<pthread.h>` : `pthread_mutex_t` *mutex*, `pthread_mutex_lock()`, `pthread_mutex_unlock()`
- Utilisez le flag -lpthread pour pouvoir compiler le programme