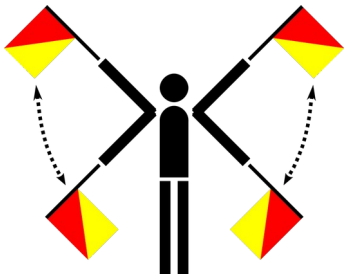
L2 - *« Systèmes d'exploitation - avancé »* *L. Mascarilla,* *E. Zahzah*	**TD n°3 -**  **Sémaphores**	La Rochelle Université

Relisez les cours 5 (mutex et sémaphores) et 6 (pthreads).

Exercice 1. Rendez-vous

a) Ecrivez un programme *rendezvous.c* qui lance deux threads qui affichent chacun un message « Thread i – partie 1 », où i est le numéro du thread (0 ou 1). Ensuite, chacun des deux threads attend que l'autre thread ait affiché son message avant d'afficher « Thread i – partie 2 ». Par exemple :

```
Thread 1 - Partie 1
Thread 0 - Partie 1
Thread 0 - Partie 2
Thread 1 - Partie 2
```

Ce mécanisme dans lequel chaque thread attend l'autre est appelé un « rendezvous » : les secondes parties des deux threads sont exécutées uniquement quand toutes les premières parties l'ont été. Ici le rendez-vous, pour chaque thread, est réalisé juste avant le second affichage (« Thread i – partie 2 »)

Vous utiliserez pour cela deux sémaphores (*sem_t sem1, sem2;* ***// placées en variables globales***), deux fonctions threads distinctes et deux appels à `pthread_create`.
Réfléchissez à l'utilisation de *sem_post* (opération V) et *sem_wait* (opération P) pour synchroniser les deux threads :

- les valeurs des deux sémaphores sont initialement à 0 : aucun thread n'a réalisé d'affichage
- Quand un thread (par exemple le premier) a affiché son message (« `Thread 0 - Partie 1 »`), il réalise une opération V sur son sémaphore : `V(sem1)`. Puis il attend sur une opération P sur l'autre sémaphore que l'autre thread ait signalé son affichage : `P(sem2)`

b) Récrivez ce code en utilisant une seule fonction thread, mais toujours deux appels à *pthread_create* avec en argument le numéro (0 ou 1) du thread. Les sémaphores seront placées dans un tableau (*sem_t sem[2];*).

Exercice 2. Barrier

La barrière est la généralisation du code précédent au cas où N threads doivent être synchronisés : les N threads doivent avoir exécuté leurs premières parties avant de passer à la seconde partie du code. La portion de code où tous les threads sont synchronisés est appelée « point critique ».

Par exemple :

```
Thread 1 - Partie 1
Thread 0 - Partie 1
Thread 2 - Partie 1
Thread 1 - Point critique
Thread 0 - Point critique
Thread 2 - Point critique
```

Vous utiliserez pour cela :

- une variable globale n : le nombre de threads qui a effectué la partie 1
- un mutex
- un sémaphore de nom « barriere »

L'algorithme de chaque thread peut se décrire sous la forme :

```
// protection de la variable n en écriture
P(mutex)
    n = n + 1
V(mutex)

// quand les n threads sont passés...
si N=n alors
    V(barriere) //..effectuer une opération V qui déclenche le passage d'un thread
sinon // on attend ...
    // Tourniquet
    P(barriere)  // attente d'une opération V
    V(barriere)  // qui est propagée à un autre thread
finsi
```

Le tourniquet est un motif (pattern) de programmation classique qui permet de faire placer un thread à la fois :
- initialement le tourniquet est bloqué (aucun thread ne passe)
- le n^{ieme} thread débloque le tourniquet et le premier thread qui passe débloque le suivant : les threads passent les uns après les autres.

Remarque : vous pouvez tenter de proposer d'autres algorithmes mais attention aux interblocages.

Exercice 3. Producteurs/consommateurs

Deux types de threads appelés producteurs et consommateurs ont accès à un tableau d'entiers : `int buf[BUF_SIZE]` . Les producteurs écrivent dans le tableau et les consommateurs lisent dans ce même tableau. Chaque écriture remplie une case du tableau qui sera libérée lors de la lecture.
Pour simplifier, une variable globale `i` indiquera l'indice de la première case libre, et un lecteur lira la case immédiatement précédente. Cette variable `i` sera mise à jour par les threads. Il ne doit bien sûr pas y avoir de pertes (données non lues ou écrasées).

Dans notre cas, les producteurs écriront un nombre aléatoire compris entre 0 et 99 (fonction `rand() %100`).
En théorie, le nombre de producteurs et de consommateurs n'est pas nécessairement connu ni fixe

(certains peuvent se terminer avant d'autres). Dans un premier temps, chaque producteur comprendra une boucle infinie d'écriture, de même, chaque consommateur comprendra une boucle infinie de lecture.

La difficulté est que les producteurs ne doivent pas bloquer les consommateurs et inversement.

Vous utiliserez :
- un mutex : pour protèger l'accès au tableau
- un sémaphore « `nb_places` » : qui indique le nombre de places disponibles dans le tableau.
- un sémaphore « `nb_donnees` » : qui indique le nombre de données produites non consommées.

Quand un consommateur a lu une valeur, il doit faire une opération `V()` sur `nb_places` et un producteur fera une opération `P()` avant d'écrire dans une case.

Quand un consommateur veut lire une valeur, il doit faire une opération `P()` sur `nb_donnees` et un producteur fera une opération `V()` après la lecture.

Exercice 4. Le diner des philosophes

Problème :
- Cinq philosophes (ou plus) se trouvent autour d'une table ronde
- Chacun des philosophes a devant lui un plat de spaghetti
- A gauche de chaque plat de spaghetti se trouve une fourchette.

Un philosophe ne peut manger que s'il a deux fourchettes, il n'y a donc pas suffisamment de fourchettes pour que les philosophes mangent tous simultanément.

Un philosophe n'a que trois états possibles :

- penser pendant un temps indéterminé ;
- être affamé : dans ce cas il prend les **deux fourchettes** (gauche puis droite).
- manger pendant un temps indéterminé.

Quand il a mangé, il repose ses fourchettes puis recommence à penser…

Le but est d'écrire un programme qui représente les philosophes par des threads (un thread par philosophe) qui utilise des sémaphores pour gérer les situations suivantes :

- une fourchette ne peut être utilisée que par un seul philosophe à la fois

- il ne doit pas y avoir interblocage

- un philosophe ne doit pas mourir de faim

- plusieurs philosophes peuvent manger en même temps.

Remarque : Les temps indéterminés (durée de la pensée d'un philosophe) seront simulés par :

sleep(rand()%3). Le temps d'un repas sera fixé à une seconde.

a) Plusieurs solutions classiques existent pour ce problème (dont certaines fausses !), toutes plus ou moins complexes. Nous allons utiliser ici l'idée suivante : s'il n'y avait que 4 philosophes (et toujours autant de fourchettes), il ne pourrait pas y avoir d'interblocages. En effet, chaque philosophe peut prendre une fourchette et il en reste une sur la table, c'est-à-dire qu'un des philosophes peut manger

à tout instant.

Nous allons donc imposer que seuls 4 philosophes puissent essayer de prendre leurs fourchettes à un instant donné. Pour cela, nous utiliserons un sémaphore initialisé à la valeur 4 : quand un philosophe veut prendre ses fourchettes il doit, au préalable, faire une opération P sur ce sémaphore. Quand il a reposé ses fourchettes il réalise une opération V. Ceci garanti qu'à un instant donné seuls 4 philosophes (au plus) peuvent manger.

Vous représenterez les fourchettes par :

```
#define NB_PHILOSOPHES 5

#define gauche(i) i

#define droite(i) (i+1)%NB_PHILOSOPHES

pthread_mutex_t fourchettes[NB_PHILOSOPHES];
```

où gauche et droite représentent les indices des fourchettes gauche et droite pour le philosophe numéro i.

Bonus :

il existe une autre solution (simple) à ce problème : s'il y a au moins un philosophe gauche et au moins un droitier il ne peut y avoir interblocage. Testez cette solution.