TP n°4 : Les conditions de synchronisation

1 Les conditions de synchronisation

1.1 Présentation

Nous avons présenté dans le cours les **régions critiques** basées sur l'attente d'une **condition de synchronisation** :

```
région p quand (condition)
    région critique
fin de région
```

La région critique est exécutée en exclusion mutuelle (sur `p`) si la condition est respectée. Les threads Linux implantent cette fonction avec les `pthread_cond_t`. Ces conditions s'utilisent toujours avec un verrou (`pthread_mutex_t`).

1.2 Un exemple

Un extrait de <pthread.h>

```c
// Initialisation d'une condition de synchronisation
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// S'endormir sur la condition en libérant le mutex
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

// Réveiller un processus en attente de la condition
int pthread_cond_signal(pthread_cond_t *cond);

// Réveiller tous les processus en attente de la condition
int pthread_cond_broadcast(pthread_cond_t *cond);

// Supprimer une condition de synchronisation
int pthread_cond_destroy(pthread_cond_t *cond);
```

Travail à faire :

- Vous pouvez retrouver toutes les informations sur la page de manuel [1] des conditions de synchronisation.
- Continuez avec la documentation des verrous [2].
- Exécutez ce programme et étudiez son fonctionnement :

1. https://www.daemon-systems.org/man/pthread_cond_init.3.html
2. https://www.daemon-systems.org/man/pthread_mutex_init.3.html

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <assert.h>

/*****************************************************************
** Outils de synchronisation.
*****************************************************************/
pthread_mutex_t mutex   = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;

/*****************************************************************
** Définition des couleurs et du nombre de peintres
*****************************************************************/
typedef enum {RED, BLUE} COLOR;
char* colors_name[] = {"RED", "BLUE"};
#define NEXT_COLOR(c) (((c) == RED) ? BLUE : RED)

COLOR color = RED;
int nb_painters = 0;

/*****************************************************************
** Chaque peintre va récupérer un numéro unique et une couleur
** et peindre trois fois en alternant les couleurs.
*****************************************************************/
void* painter (void* _unused) {

    /* Chaque peintre prend un numéro et une couleur. */
    pthread_mutex_lock(&mutex);
        int my_number = (++nb_painters);
        COLOR my_color = ((my_number % 2) ? RED : BLUE);
    pthread_mutex_unlock(&mutex);

    /* il y a trois zones à peindre */
    for(int i=0; i < 3; i++) {

        /* il faut attendre pour alterner les couleurs */
        pthread_mutex_lock(&mutex);
        while (color != my_color) {
            /* je m'endors car la condition est fausse */
            pthread_cond_wait(&condition, &mutex);
        }

        printf("numéro=%d, couleur=%s\n", my_number, colors_name[my_color]);
        sleep(1);

        color = NEXT_COLOR(color);       /* couleur suivante */
        pthread_cond_broadcast(&condition); /* réveil des peintres */
        pthread_mutex_unlock(&mutex);    /* libération */
    }

    return NULL; /* pas de résultat */
}

/*****************************************************************
** Initialiser les peintres et attendre la fin du travail.
*****************************************************************/

int main (void) {
    int n = 4;
    pthread_t peintres[ n ];

    for(int i=0; i<n; i++) {
        assert(pthread_create(&peintres[i], NULL, painter, NULL) == 0);
    }
    for(int i=0; i<n; i++) {
```

- Que se passe-t-il si le nombre de peintres est impaire ? Comment corriger la situation (**Indice** : décrémenter le nombre de peintre lors d'un départ et enrichir la condition de synchronisation avec un test sur le nombre de peintres) ?

2 Les philosophes et les spaghettis

2.1 Présentation

Nous retrouvons dans un restaurant `N` philosophes attablés pour manger un plat de spaghettis. Chaque philosophe alterne des périodes de ripaille et des périodes d'intense réflexion. Malheureusement, le restaurateur a placé seulement une fourchette entre chaque philosophe soit `N` fourchettes et, bien entendu, les philosophes ont besoin de deux fourchettes pour manger. Ils appliquent donc l'algorithme suivant :

```
Répéter
    prendre les fourchettes de droite et de gauche (si possible)
    manger
    reposer les fourchettes
    méditer sur la cuisson de spaghettis
Jusqu'à ...
```

On vous demande d'écrire un programme qui simule l'activité des philosophes pour régler le partage des fourchettes. Commencez avec deux puis 5 ou 6 philosophes.

2.2 Comment faire ?

On vous donne les indications suivantes :

- Chaque philosophe est représenté par un thread (le code est le même pour tous les threads).
- Un tableau `philosophe` code l'état des philosophes (`RIPAILLE` , `AUTRE_SITUATION`).
- La condition de synchronisation d'un philosophe est liée à l'état des philosophes placés à sa gauche et à sa droite. Elle est de la forme

```
while ((philosophe[gauche]==RIPAILLE) || (philosophe[droite]==RIPAILLE)) {
    /* je m'endors car l'un de mes voisins mange (voire les deux) */
    pthread_cond_wait(&condition, &mutex);
}
```

- Il est **fortement** conseillé d'imprimer un message pour signaler l'état d'un philosophe.

```
le philosophe 1 prend les fourchettes
le philosophe 1 commence à manger
le philosophe 1 repose les fourchettes
le philosophe 1 pense ...
etc...
```

Je vous conseille de suivre les étapes ci-dessous :

1. Le restaurateur prépare la table (en clair, commencez par initialiser le tableau, le verrou et la condition de synchronisation).
2. Dans un deuxième temps, les philosophes arrivent (en clair, créez `n` threads, un par philosophe).
3. Chaque philosophe ne s'intéresse qu'à son collègue de droite et de gauche (en clair, calculez le numéro des philosphes à gauche et à droite).
4. Les philosophes se bloquent sur la condition de synchronisation pour attendre que leurs **DEUX** fourchettes soient libres (les voisins ne mangent pas).

5. Pensez à donner de la lecture à chaque philosophe (en clair, ralentissez les processus en appelant la fonction `sleep`).