Exercice 1 : Synchronisation.

Pour les étudiants souhaitant traiter un sujet du type Mines Ponts, vous traîterez l'exercice suivant en plus du sujet fourni.

1 Rendez-vous et barrière de synchronisation

Deux threads T_1 et T_2 exécutent les fonctions suivantes :

```
void* T1(void* arg) {
    a1();
    a2();
    return NULL;
}
```

```
void* T2(void* arg) {
    b1();
    b2();
    return NULL;
}
```

1. Si on lance deux threads en parallèle qui exécutent respectivement `T_1` et `T_2`, quels sont les entrelacements possibles des exécutions de `a_1`,`a_2`,`b_1` et `b_2` (on supposera que ces quatres fonctions s'exécutent de manière atomique) ?
2. À l'aide de deux sémaphores initialisés à 0, écrire une version modifiée des fonctions `T1` et `T2` garantissant simultanément :
   — a_2 ne peut pas être exécutée avant que b_1 ne soit terminée ;
   — b_2 ne peut pas être exécutée avant que a_1 ne soit terminée.

On considère maintenant $n \geq 2$ threads exécutant tous le code suivant :

```
void* worker(void* arg) {
    step1();
    step2();
    return NULL;
}
```

La contrainte que l'on souhaite garantir est la suivante : aucun thread ne doit exécuter `step2()` tant que tous les n threads n'ont pas terminé d'exécuter `step1()`.

Les variables partagées sont :

```
int n;
int count = 0;
pthread_mutex_t mutex;
sem_t barrier; /*initialise \‘a 0*/
```

Nous allons considérer une première solution :

```
void* worker(void* arg) {
    step1();

    pthread_mutex_lock(&mutex);
    count++;
    pthread_mutex_unlock(&mutex);

    if (count == n) {
        sem_post(&barrier);
    }

    sem_wait(&barrier);

    step2();
    return NULL;
}
```

3. Décrire un ordonnancement concret (par exemple pour $n = 3$) qui met en évidence que cette solution n'est pas fonctionnelle.
4. Modifier la solution proposée afin de fournir une solution correcte au problème. Expliquer brièvement le principe.
5. Une fois que les n threads ont franchi la barrière, quelle est la valeur du sémaphore `barrier` ?
6. Que se passerait-il si un $(n + 1)$eme thread cherchait à franchir la barrière ?

Pour la suite du sujet on notera `barrier_wait` la portion de code corrigée précédemment qui se trouve entre `step1()` et `step2` dans la fonction `worker`.

On considère $n \geq 2$ threads exécutant une boucle :

```
void* worker(void* arg) {
    while(true) {
        step();
        barrier_wait();
    }
}
```

La fonction `barrier_wait()` doit permettre de garantir que tous les n threads ont terminé leur tour de boucle (l'exécution de la fonction `step()`) avant qu'un d'entre eux soit en mesure de recommencer une nouvelle exécution de `step()` et donc un nouveau tour de boucle.

7. Donner un ordonnancement montrant que cette solution permet un mélange entre deux itérations consécutives du `while`. On proposera un ordonnancement concret en se contentant par exemple de 3 threads pour simplifier la description.

8. Voici une tentative de solution fonctionnelle utilisant un nouveau sémaphore `newpassage` en plus des variables déjà utilisées jusqu'ici :

```
void barrier_wait() {

    pthread_mutex_lock(&mutex);
    count++;
    if (count == n) {
        sem_wait(&newpassage);
        sem_post(&barrier);
    }
    pthread_mutex_unlock(&mutex);

    sem_wait(&barrier);
    sem_post(&barrier);

    pthread_mutex_lock(&mutex);
    count--;
    if (count == 0) {
        sem_post(&newpassage);
    }
    pthread_mutex_unlock(&mutex);

    sem_wait(&newpassage);
    sem_post(&newpassage);
}
```

Donner l'initialisation correcte de `newpassage` pour que cela fonctionne. En sortie d'un tour de boucle, quelles sont les valeurs des deux sémaphores `barrier` et `newpassage` ? Rajouter une instruction dans le code qui permet d'avoir ces valeurs correctes pour permettre d'effectuer correctement le prochain tour de boucle.

2 Application : construire des molécules H_2O

On considère deux types de threads :

— threads oxygène (O),

— threads hydrogène (H).

Une molécule d'eau requiert exactement **2 H et 1 O**.

Nous allons considérer deux types de threads : Oxygen et Hydrogen, ils s'exécutent de manière concurrentes mais doivent tous exécuter simmultannément une opération `lier()` lorsque deux threads de type hydrogène et un de type oxygène sont réunis au même point de synchronisation. Plus précisemment, il vont chacun appeler la fonction `lier()` et on souhaite garantir à l'aide d'une barrière de synchronisation qu'il aient tous les trois terminé leur opération de liaison avant de procéder à la suite du programme et qu'aucun autre thread ne commence à invoquer la fonction `lier()` tant que la barrière n'est pas franchie.

— Si un thread de type oxygène arrive à la barrière c'est qu'il a terminé l'appel à `lier()` et si on n'a pas deux threads hydrogène présents, il doit attendre.

— Si un tread de type hydrogène arrive à la barrière il doit attendre qu'il y ait un thread de type oxygène et un autre thread de type hydrogène.

Formellement, si l'on note S la suite des threads qui exécutent `lier()`, et si l'on partitionne S en blocs consécutifs de taille 3, alors pour tout bloc B_i : il contient un thread de type oxygène et deux de type hydrogène et il n'existe pas d'exécution où un élément de B_{i+1} exécute `lier()` avant que tous les éléments de B_i aient exécuté `lier()`.

Voici les variables partagées par l'ensemble des threads :

```
int oxygen = 0;
int hydrogen = 0;

pthread_mutex_t mutex;

sem_t oxyQueue;
sem_t hydroQueue;

barrier_t barrier3; // barriere de taille 3
```

9. Compléter le code suivant pour un thread oxygène :

```
void* oxygen_thread(void* arg) {
    pthread_mutex_lock(&mutex);
    oxygen++;

    if (/* condition */) {
        /* sem_post(...)




        */
    } else {
        pthread_mutex_unlock(&mutex);
    }

    sem_wait(&oxyQueue);
    lier();

    barrier_wait(&barrier3);
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

10. Ecrire de manière analogue le code d'un thread hydrogène et indiquer pourquoi ce thread ne libère pas le mutex après le passage de la barrière.
11. Montrer que l'utilisation de la barrière de taille 3 et du mutex garantit l'absence de mélange entre deux molécules consécutives.