Corrigé – Synchronisation Concours blanc

1. On peut avoir $a_1 < a_2 < b_1 < b_2$ ou $a_1 < b_1 < a_2 < b_2$ ou $a_1 < b_1 < b_2 < a_2$ ou $b_1 < b_2 < a_1 < a_2$ ou $b_1 < a_1 < a_2 < b_2$ ou $b_1 < a_1 < b_2 < a_2$.
2. On utilise deux sémaphores initialisés à 0 :

```
sem_t aDone, bDone; // init a 0
```

   On propose :

```
void* T1(void* arg) {
    a1();
    sem_post(&aDone); // signale que a1 est fini
    sem_wait(&bDone); // attend que b1 soit fini
    a2();
    return NULL;
}

void* T2(void* arg) {
    b1();
    sem_post(&bDone); // signale que b1 est fini
    sem_wait(&aDone); // attend que a1 soit fini
    b2();
    return NULL;
}
```

3. Dans le code non-correct :

```
sem_wait(&barrier);
```

   le sémaphore `barrier` est initialisé à 0 et n'est incrémenté qu'une seule fois (quand `count == n`). Donc tant que le $n^{\text{ième}}$ thread n'a pas exécuté `sem_post(&barrier)`, tout thread arrivant à `sem_wait(&barrier)` est bloqué et un seul sera débloqué.

   Considérons par exemple qu'on a trois threads T_1, T_2 et T_3 et l'ordonnancement qui consiste à exécuter chacun leur tour la fonction jusqu'à la ligne `sem_wait(&barrier);` où il seront bloqués tant que le sémaphore ne sera pas incrémenté. Le troisième thread va alors valider la conditionnelle et incrémenter le sémaphore. Dès lors, l'un des trois threads va passer à `step2` et les deux autres seront bloqués et rien ne permettra de les débloquer (il n'y a aucune incrémentation du sémaphore prévue).
4. Dans la solution correcte, chaque thread exécute :

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

   Le $n^{\text{ième}}$ thread exécute une première fois `sem_post(&barrier)` ce qui libère un thread. Ensuite, **chaque thread libéré** exécute immédiatement `sem_post(&barrier)`, libérant le suivant. Ainsi, les n threads franchissent la barrière : le signal circule comme un tourniquet.
5. Après passage des n threads, le dernier thread a exécuté `sem_post(&barrier)` alors qu'aucun thread n'attend. Donc la valeur de `barrier` est 1 à la fin (tourniquet ouvert), et il ne reste aucun thread bloqué.

6. Si un nouveau thread cherchait à emprunter la barrière il le ferait donc sans problème.

   On remarque que la solution suivante résout aussi le problème en remettant la variable `barrier` à 0, si on a proposé une telle solution alors les réponses aux deux dernières questions sont différentes :

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

    pthread_mutex_lock(&mutex);
    count++;


    if (count == n) {
         for(int i=0;i<n;i++)
             sem_post(&barrier);
         count =0;
    }
    pthread_mutex_unlock(&mutex);
    sem_wait(&barrier);

    step2();
    return NULL;
}
```

7. La réponse ici va dépendre de la solution proposée dans le début du sujet :
   Dans la tentative non correcte à un seul tourniquet, on peut avoir le scénario suivant :
   — À l'itération k du `while`, le tourniquet est ouvert (valeur 1) après le passage de chaque thread.
   — Un thread rapide commence l'itération k+1, incrémente `count` (qui a maintenant dépassé n de toute manière car n'est jamais ré-initilisé), atteint `sem_wait(&barrier)` et passe immédiatement car `barrier` est encore ouvert.
   — Pendant ce temps, un thread lent n'a pas encore fini la phase de sortie de l'itération k car il n'a pas accédé au sémaphore avant le thread rapide.
   Donc mélange entre itérations : la contrainte de barrière réutilisable n'est pas satisfaite.
   Pour être plus concret : considérons T_1, T_2 et T_3 tels que chacun des threads exécute son code l'un après l'autre jusqu'au `wait`. T_3 va alors débloquer la barrière, laissant passer par exemple T_1 qui libère la barrière juste après l'avoir passée, continue son exécution dans un deuxième tour de boucle et repasse la barrière avant même que T_2 ou T_3 n'aient repris la main, T_1 peut alors passer et repasser dans plusieurs tours de boucle en passant puis libérant la barrière sans que T_2 ni T_3 n'aient commencé leur deuxième tour.
8. Il faut l'initialiser à 1. Dans cette solution, à la fin de la boucle, `newpassage` vaut 1 et `barrier` aussi alors que ce n'était pas sa valeur initiale. Pour recommencer un nouveau tour de boucle avec les bonnes valeurs, on doit décrémenter une fois au total `barrier`, on peut donner ce rôle au thread qui débloque le deuxième passage et donc ajouter dans le code suivant la condtionnelle `if (count==0)` on conserve l'incrémentation de `newpassage` et on ajoute la décrémentation (`wait`) de `barrier`.
   Justification :
   — Au début de l'itération 1, aucun thread ne doit passer `barrier` avant l'arrivée des n threads ⇒ `barrier=0`.
   — La phase 2 ne doit pas bloquer au tout début ⇒ `newpassage=1`.
9. Complétion (thread oxygène). On suppose `oxyQueue` et `hydroQueue` initialisés à 0, et une barrière `barrier3` de taille 3. Le schéma impose qu'un thread qui "forme" une molécule conserve le mutex jusqu'après la barrière et que l'oxygène le libère après la barrière.

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

    if (hydrogen >= 2) {
        // libere 2 H et 1 O (soi-meme)
        sem_post(&hydroQueue);
        sem_post(&hydroQueue);
        hydrogen -= 2;

        sem_post(&oxyQueue);
        oxygen -= 1;
        // mutex conserve: bloque les arrivants d'autres molecules
    } else {
        pthread_mutex_unlock(&mutex);
    }

    sem_wait(&oxyQueue);
    bond();

    barrier_wait(&barrier3);

    // libere le mutex exactement une fois par molecule (il n'y a qu'un O)
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

Réponses demandées :

— Condition : `hydrogen >= 2`.

— Réveils : 2 `sem_post(&hydroQueue)` et 1 `sem_post(&oxyQueue)`.

— Mises à jour : `hydrogen -= 2;` puis `oxygen -= 1;`.

— Déverrouillage : après la barrière, par le thread oxygène.

10.

```
void* hydrogen_thread(void* arg) {
    pthread_mutex_lock(&mutex);
    hydrogen++;

    if (hydrogen >= 2 && oxygen >= 1) {
        // libere 2 H et 1 O
        sem_post(&hydroQueue);
        sem_post(&hydroQueue);
        hydrogen -= 2;

        sem_post(&oxyQueue);
        oxygen -= 1;
        // mutex conserve
    } else {
        pthread_mutex_unlock(&mutex);
    }

    sem_wait(&hydroQueue);
    bond();

    barrier_wait(&barrier3);

    // ici: pas d'unlock (dans ce schema)
    return NULL;
}
```

Le schéma garantit exactement une libération de mutex par molécule. Lors du passage de la barrière l'un des trois threads concerné détient le mutex mais on ne sait pas lequel. Comme il y a exactement un oxygène par groupe, le choix "O libère" évite le risque de libérer plusieurs fois, et impose que le mutex reste détenu jusqu'à ce que les 3 threads aient exécuté `lier()`.

11. Absence de mélange entre molécules :
    - Quand une molécule est formée (condition satisfaite dans un des threads), ce thread conserve le mutex et libère exactement 3 autorisations sur les files (`oxyQueue/hydroQueue`) correspondant à 2H+1O.
    - Les 3 threads autorisés exécutent `lier()`, puis attendent tous sur `barrier3`.
    - Tant que les 3 n'ont pas atteint la barrière, elle ne s'ouvre pas ; donc un des 3 détient encore le mutex, ce qui empêche de former (et donc de libérer) les autorisations de la molécule suivante.

    Ainsi, aucun thread de la molécule suivante ne peut exécuter `bond()` avant que les 3 de la molécule courante ne l'aient fait.