Erreurs à éviter et conseils pour l'examen

Suite à la correction du CC intermédiaire, voici une explication des erreurs classiques (et trouvées dans beaucoup de copies). Lisez ce document en entier que vous ayez bien ou mal réussi le CC : on en profite pour rappeler les points les plus importants du cours.

Question 4 : Moniteurs de Hoare

Certains d'entre vous n'ont pas compris ou pas retenu le principe des moniteurs de Hoare. C'est un concept essentiel de ce cours, que vous avez déjà appliqué dans plusieurs TD et TP (par exemple le producteur-consommateur, qui est le b-a-ba de la programmation concurrente et que vous devez donc tous maîtriser sur le bout des doigts). C'est inacceptable qu'autant d'étudiants n'aient pas su dire de quoi il s'agissait sur leur copie.

Un moniteur de Hoare, c'est :

- Un ensemble de variables
- Un ensemble de fonctions qui agissent sur ces variables
- Un mutex
- Une (ou plus) variable de condition.

Chaque fonction va verrouiller le mutex en début de fonction et le déverrouiller à la fin de la fonction (sauf cas particulier où on est sûr que la concurrence ne pose pas problème et où on peut se passer de mutex). En C++11, ce verrouillage/déverrouillage peut être fait simplement en déclarant une variable de type `std::unique_lock<std::mutex>` en lui passant le mutex en paramètre (le destructeur de cette fonction s'occupera du déverrouillage), ou bien en appelant explicitement `mutex.lock()` et `mutex.unlock()` en début/fin de fonction.

Quand une fonction doit attendre qu'une condition devienne vraie, elle fait :

```
while (!condition()) {
    cond_var.wait(lock);
}
```

(`cond_var` est une variable de condition, `lock` un verrou)

La fonction `wait` permet d'attendre qu'un `notify_all` ou un `notify_one` soit appelé sur la variable de condition, et elle relâche `lock` pendant l'attente.

Une erreur fréquente est d'utiliser un `if` au lieu d'un `while` est incorrect. La fonction `wait` des variables de conditions ne donne pas de garantie que la condition est vraie en sortie.

Question 9 : inversion de priorités

Très peu d'entre vous avaient la définition correcte d'une inversion de priorité. Relisez les transparents (ht tps://asr-lyon1 slide 28) : il faut 3 tâches pour faire une inversion de priorité. Nous avons vu un exemple en TD.

Question 10 : temps de réponse

Les réponses étaient globalement bonnes pour cet exercice. Une erreur classique est de considérer la différence entre le démarrage de la tâche et la fin de la tâche, au lieu de la différence entre l'**arrivée** de la tâche et la fin. C'est différent dans le cas où la tâche ne peut pas démarrer immédiatement.

Gestion des accès au parking

La première partie est strictement plus simple que le producteur-consommateur vu en TD et revu en TP : on bloque quand le parking est plein, mais par définition les voitures ne peuvent sortir que quand elles sont dans le parking, donc la sortie d'une voiture ne peut pas être bloquante.

Instantiation des threads

L'énoncé vous demandait : « Cette fonction doit exécuter NB_ESSAIS fois de suite les opérations void demande_entree(cote c) puis sortie() ». C'est bien chaque thread qui doit appeler les deux fonctions, on ne peut donc pas écrire :

```
std::thread t(p.demande_entree);
std::thread t(p.sortie);
```

Chaque thread doit exécuter les 3 actions. Un thread exécute une fonction qui donne son comportement. Ici la fonction peut être :

```
void comportement_thread() {
    cout << "avant\n";
    b.barrier();
    cout << "après\n";
}
```

et on l'instanciera avec :

```
std::thread t(comportement_thread);
```

Bien sûr, vu qu'il y a plusieurs threads, il faut aussi stocker les threads dans un tableau et faire un `.join()` sur chaque thread, comme vous l'avez fait plusieurs fois en TP.

Il faut bien distinguer :

- La fonction qui crée les threads, par exemple la fonction `main`, qui appelle le constructeur de `std::thread` et la fonction `join()`.
- La fonction exécutée par les threads, c'est à dire celle passée en paramètre au constructeur de `std::thread`.
- Lorsqu'on en utilise, les méthodes des moniteurs de Hoare, qui sont appelés par la fonction exécutée par les threads. Bien sûr, comme ce sont des méthodes, vous ne pouvez pas les appeler sans un objet (vous pouvez écrire `moniteur.methode()`, mais pas juste `methode()` si vous n'êtes pas déjà dans une méthode de la même classe).

En particulier, le moniteur de Hoare est appelé par les threads, mais il ne contient ni ne crée aucun thread. Inversement, la fonction qui crée les threads est séquentielle, elle n'a pas besoin d'utiliser d'outil dédié à la synchronisation.

Mutex et variables partagées

- Attention à ne pas oublier les mutexes sur les accès à variables partagées.
- L'objet mutex (la variable de type `std::mutex`) doit être partagée par toutes les fonctions qui accèdent à l'objet (d'où l'idée des moniteurs de Hoare où on range les mutexes dans la même classe ou structure que les variables). Ça n'a pas de sens d'écrire :

```
int x;

std::mutex m_f;
void f() {
    std::unique_lock<std::mutex> lock(m_f);
    x++;
}

std::mutex m_g;
void g() {
    std::unique_lock<std::mutex> lock(m_g);
    x++;
}
```

Car `f` et `g` accèderaient à la même variable `x`, mais on n'aurait pas d'exclusion mutuelle entre `f` et `g` car `f` verrouille `m_f` et `g` verrouille `m_g`.

On ne peut pas non plus écrire :

```
void f() {
    std::mutex m;
    m.lock();
    x++;
    m.unlock();
}
```

car alors `m` serait locale à la fonction `f`, donc on aurait une instance de mutex par appel de `f` (et donc aucune exclusion mutuelle) !

Sur l'exercice parking en particulier

J'ai vu très souvent des appels à la méthode `demande_entree()` à l'intérieur de la méthode `sortie()`. Ces deux fonctions sont appelées par les voitures en entrée et en sortie du parking. Le fait qu'une voiture sorte peut débloquer une voiture bloquée dans `demande_entree()`, mais certainement pas entraîner que la voiture courante refasse une demande d'entrée !