TD 4 : Thread, Lock

Objectifs pédagogiques :

- **thread**
- **atomic, mutex**
- **section critique**

Introduction

Dans ce TD, nous allons aborder l'utilisation des thread en C++ pour faire de la programmation concurrente. L'exercice permet de voir plusieurs mécanismes que l'on peut employer pour protéger les données critiques des accès concurrent.

Pour être sûr de ne pas entrer en conflit avec d'autres applications, nous utiliserons le namespace `pr` pour notre implémentation.

1.1 Création de Thread

On considère une fonction `void work(int id)` qui affiche son identifiant id, puis dort pendant une durée aléatoire comprise entre 0 et 1000 ms, puis affiche un deuxième message.

work

```
void work (int index) {
      std::cout << "started "<< index << endl;
      auto r = ::rand() % 1000 ; // 0 to 1 sec
      std::this_thread::sleep_for (std::chrono::milliseconds(r));
      std::cout << "finished "<< index << endl;
}
```

Question 1. Ecrire une fonction `void createAndWait(int N)` qui crée N threads exécutant `work` avec pour identifiant leur ordre de création compris entre 0 et $N-1$, puis attend qu'ils soient tous terminés. On souhaite maximiser la concurrence.

On doit pouvoir l'utiliser avec le main suivant :

work

```
int main (int argc, const char ** argv) {
      int N = 3;
      if (argc > 1) {
            N=atoi(argv[1]);
      }
      // pour eviter des executions trop reproductibles, pose le seed.
      ::srand(::time(nullptr));
      return pr::createAndWait(N);
}
```

Question 2. Si l'on exécute le programme, quels sont les entrelacements possibles ?

Question 3. Si on ajoute un affichage "created id" juste après l'invocation du constructeur de thread, et un affichage "joined id" juste après l'appel à `join`. Que peut on dire, pour un identifiant donné, de l'ordre entre les messages : created/started/finished/join ?

1.2 Variables partagées.

On considère une classe Compte munie d'un attribut entier représentant le solde, d'un constructeur positionant le solde initial, d'une opération membre `void crediter (int val)` qui ajoute de l'argent au solde, et d'une opération membre `int getSolde() const`.

On donne aussi une fonction `void jackpot (Compte & c)` qui crédite 10000 pièces d'or sur le compte fourni, mais une par une (ding, ding, ding...).

jackpot.cpp

```cpp
#include <vector>
#include <thread>
using namespace std;

class Compte {
      int solde;
public :
      Compte(int solde=0):solde(solde) {}
      void crediter (size_t val) { solde+=val ;}
      int getSolde() const {return solde;}
};
const int JP = 10000;
void jackpot(Compte & c) {
      for (int i=0; i < JP; i++)
            c.crediter(1);
}

const int NB_THREAD = 10;
int main () {
      vector<thread> threads;
      // TODO : creer des threads qui font jackpot sur un compte
      for (auto & t : threads) {
            t.join();
      }
      // TODO : tester solde = NB_THREAD * JP
      return 0;
}
```

Question 4. Complétez le main, de façon à instancier un compte, et créer N threads qui exécutent le code de la fonction Jackpot.

Question 5. Quel sera le solde du compte à la fin du programme pour N=10 ?

Question 6. Si le nombre de pièces d'or du jackpot est faible (disons 100), on n'observera pas de problème en général sur cet exemple, expliquez pourquoi.

Question 7. Comment utiliser un `atomic` pour corriger ces problèmes ? Expliquez l'effet au niveau des entrelacements possibles.

1.3 Section Critique.

On ajoute au Compte de la question précédente une opération : `bool debiter (int val)` qui doit controler que le solde du compte est suffisant (la banque ne prête pas), et si c'est le cas réduire le solde du montant indiqué. La fonction rend vrai si le débit a eu lieu.

```cpp
class Compte {
...
      bool debiter (unsigned int val) {
            bool doit = (solde >= val);
            if (doit) {
                  solde-=val ;
            }
            return doit;
      }
...
};
```

```
void losepot(Compte & c) {
      for (int i=0; i < JP / 10; i++)
            c.debiter(10);
}
```

Question 8. On reprend l'exemple du Jackpot, mais cette fois-ci en débit, le LosePot retire 10000 par paquets de 10. On lance N thread qui exécutent cette fonction ; le compte peut-il tomber en négatif ? Quelle garantie fournit `atomic` ici ?

Question 9. Introduire un mutex dans la classe Compte pour sécuriser son utilisation dans un contexte Multi-Thread.

Question 10. Utiliser un `unique_lock` plutôt que l'API lock/unlock. Comparer les syntaxes.

Question 11. Dans les versions avec un mutex, qu'apporte l'utilisation d'un atomic pour l'attribut solde ? Si l'on n'utilise pas `atomic` est-ce nécessaire de protéger la méthode `getSolde` avec le mutex ?

Question 12. La classe `Compte` en l'état ne dispose pas de constructeur par copie : mutex n'est pas copiable, ce qui rend la version générée par le compilateur de ce constructeur par copie non disponible. Comment écrire ce constructeur ?

NB: Sans constructeur par copie, on ne peut par exemple pas insérer un Compte dans un `std::vector`.

TME 4 : Mutex et section critique

Objectifs pédagogiques :

- **atomic, mutex**
- **sections critique, interblocage**

1.1 Mise en place

On rappelle la classe `Compte` obtenue à la fin du TD 3.

```
class Compte {
      mutable mutex m;
      int solde;
public :
      Compte(int solde=0):solde(solde) {}
      void crediter (unsigned int val) {
            unique_lock<mutex> g(m);
            solde+=val ;
      }
      bool debiter (unsigned int val) {
            unique_lock<mutex> g(m);
            bool doit = solde >= val;
            if (doit) {
                  solde-=val ;
            }
            return doit;
      }
      int getSolde() const {
            unique_lock<mutex> g(m);
            return solde;
      }
      // NB : vector exige Copyable, mais mutex ne l'est pas...
      Compte(const Compte & other) {
            other.m.lock();
            solde = other.solde;
            other.m.unlock();
      }
};
```

1.2 Transaction.

On considère à présent une Banque, possédant en attribut un ensemble de K comptes (un `vector<Compte>`) initialement avec un solde de $SOLDEINITIAL$ chacun. Elle est munie d'une opération `bool transfer(int idDebit, int idCredit, size_t val)` qui essaie de débiter le compte d'indice $idDebit$ de val, et si c'est un succès, crédite le compte $idCredit$ du même montant val. La fonction rend vrai si le transfert est un succès.

transferts.cpp

```
class Banque {
      typedef vector<Compte> comptes_t;
      comptes_t comptes;
public :
      Banque (size_t ncomptes, size_t solde) : comptes (ncomptes, Compte(solde)){
      }
      void transfert(size_t deb, size_t cred, unsigned int val) {
            Compte & debiteur = comptes[deb];
```

```
            Compte & crediteur = comptes[cred];
            if (debiteur.debiter(val)) {
                  crediteur.crediter(val);
            }
      }
      size_t size() const {
            return comptes.size();
      }
};
```

Question 1. Ecrivez un programme, qui crée N threads de transaction, qui bouclent 1000 fois sur le comportement est suivant :

- Choisir i et j deux indices de comptes aléatoires, et un montant aléatoire m compris entre 1 et 100.
- Essayer de transférer le montant m de i à j.
- Dormir une durée aléatoire de 0 à 20 ms.

Question 2. Le comportement est-il correct (pas de *datarace*) avec les protections actuelles sur le Compte ?

On estime que découpler les débits des crédits est une faute, on souhaite au contraire que la mise à jour des deux comptes concernés soit atomique : soit le transfert a lieu et les deux comptes sont mis à jour (simultanément du point de vue d'un observateur), soit il n'a pas lieu. A aucun moment un observateur ne doit pouvoir voir un des comptes débités et l'autre pas encore crédité.

Question 3. Pour permettre de manipuler le mutex des comptes dans ce scenario, on propose d'ajouter un accesseur `mutex & getMutex()` au compte pour permettre de l'écrire, ou de définir les trois méthode `void lock () const, void unlock() const, bool try_lock () const` par délégation sur le mutex stocké pour implanter le contrat d'un *Lockable* C++ [1].

Question 4. Proposez une stratégie de synchronisation pour permettre ce comportement transactionnel au niveau de `transfert`.

Question 5. Le programme se bloque immédiatement, même avec un seul thread qui fait des transferts. Pourquoi ?

Question 6. Après correction du problème précédent à l'aide de `recursive_mutex`, on introduit plusieurs thread faisant des transferts, mais de nouveau on observe parfois un interblocage, le programme entier se fige. Expliquez pourquoi et corriger le problème.

1.3 Comptabilité

On souhaite ajouter au programme des thread qui vérifient la comptabilité. On se donne une nouvelle opération `comptabiliser` dans la classe `Banque` :

BFL.cpp

```
      bool comptabiliser (int attendu) const {
            int bilan = 0;
            int id = 0;
            for (const auto & compte : comptes) {
                  if (compte.getSolde() < 0) {
                        cout << "Compte " << id << " en négatif : " << compte.getSolde() <<
                            endl;
                  }
                  bilan += compte.getSolde();
                  id++;
            }
            if (bilan != attendu) {
```

[1] cf. https://en.cppreference.com/w/cpp/named_req

```
                cout << "Bilan comptable faux : attendu " << attendu << " obtenu : " <<
                    bilan << endl;
            }
            return bilan == attendu;
        }
```

Cette méthode itère sur les comptes en sommant leur solde, et vérifie que la somme vaut bien $SOLDEINITIAL * K$. Sinon elle lève une alerte sur la console et rend false.

Question 7. Un thread comptable qui tourne en concurrence avec les threads de transfert sera-t-il satisfait avec les synchronisations actuelles ? Expliquez pourquoi.

Question 8. Essayez d'observer une exécution ou le comptable détecte une erreur (on pourra augmenter le nombre de tours de boucle, éliminer les sleep etc... pour favoriser l'apparition d'une faute).

Question 9. Ajoutez un mutex dans la banque, et les synchronisations utiles pour que le thread comptable obtienne toujours les bons résultats.

Question 10. Avec un seul mutex dans la banque, la concurrence entre les thread de transfert n'est plus possible. De fait les mutex définis dans Compte ne servent plus à rien dans ce scenario. Proposez une autre approche qui réutilise les locks de Compte plutôt que de n'avoir qu'un seul lock.

Indice : il faut juste empêcher les threads de transfert d'accéder aux comptes que le comptable a déjà vus, et crédités dans son `bilan`. Attention cependant à ne pas réintroduire de deadlocks.