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);
}
```

> On note :
> Attention à rand pseudo aléatoire, utiliser srand(time) permet d'assurer que les exécutions varient un peu.
> La partie "maximiser la concurrence" ça veut dire créer tout le monde avant le premier join. On peut discuter la boucle qui ferait : `for (int i<N) { thread t (work,i); t.join() }`
> Elle crée bien N thread mais sans concurrence possible.
> J'ai fait `emplace_back` ici mais `push_back` irait bien aussi (on donne les deux syntaxes).

createjoin.cpp

```cpp
int createAndWait (int N) {
      vector<thread> threads;
      threads.reserve(N);
      for (int i=0; i < N ; i++) {
            // push_back, plus explicite sur l'invocation du ctor de thread
            // threads.push_back(thread(work,i));
            // NB : emplace_back forward les arguments au constructeur de thread
            threads.emplace_back(work,i);
            std::cout << "created "<< i << endl;
      }
      for (int i=0; i < N ; i++) {
            threads[i].join();
            std::cout << "joined "<< i << endl;
      }
      return 0;
}
```

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

Donc ce qu'on sait :

- chaque thread exécute ses instructions dans l'ordre (sauf si le matériel/compilo s'en mêle avec out of order execution. Mais n'en parlons pas à ce stade.)
- les creations par le père précèdent le début des exécutions des fils
- les terminaisons par les fils précèdent le join du père
- les autres instructions ne sont pas comparables/ordonnées

Le dessin suivant exhibe ça :

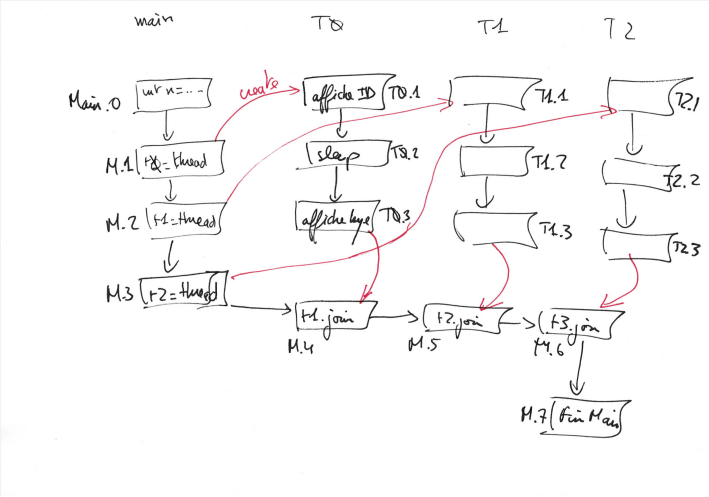
- en noir les actions liées au sein du code d'un thread
- en rouge les synchros inter-thread

On dessine ici pour trois thread crées, chacun ayant trois instructions : "mon id est ...", "sleep(rand)", "id XXX se finit"

Le main crée puis join les thread.

On remarque un *ordre partiel* : il est impossible de déterminer si $T0.1$ précède $T1.1$ par exemple. Les deux actions seront respectivement après M1 et après M2 (deux actions qui sont ordonnées au sein du main) mais leur ordre n'est pas fixé par le programme en l'état.

Ce dessin permet de capturer tous les entrelacements possibles si on le lit correctement.



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 ?

Réponse :

- started précède finished (ordonné dans fils).
- created précède joined (ordonné dans pere).
- finished précède joined (sémantique join = synchro)

MAIS created est incomparable à started et finished, on peut observer c,s,f ou s,c,f ou s,f,c.

En particulier on peut voir passer finished avant created !

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

```
1  #include <vector>
2  #include <thread>
3  using namespace std;
4
5  class Compte {
6        int solde;
7  public :
8        Compte(int solde=0):solde(solde) {}
9        void crediter (size_t val) { solde+=val ;}
10       int getSolde() const {return solde;}
11 };
12 const int JP = 10000;
```

```
13 void jackpot(Compte & c) {
14       for (int i=0; i < JP; i++)
15             c.crediter(1);
16 }
17
18 const int NB_THREAD = 10;
19 int main () {
20       vector<thread> threads;
21       // TODO : creer des threads qui font jackpot sur un compte
22       for (auto & t : threads) {
23             t.join();
24       }
25       // TODO : tester solde = NB_THREAD * JP
26       return 0;
27 }
```

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

On instancie la variable partagée AVANT de créer les thread pour permettre de leur passer.

On note le std::ref pour garantir quand on passe la réf du compte (à la création des thread) qu'elle persistera pendant la vie du thread. C'est le programmeur qui fait cette garantie (attention à ne pas mentir !).

Si par exemple, la boucle de join était faite après un bloc qui déclare l'instance de Compte, le comportement serait incorrect.

```
1 vector<thread> threads;
2 { // bloc
3 Compte c;
4 for (int i...) threads.emplace_back(work,std::ref(c));
5 } // fin bloc (BUG)
6 for (auto & t : threads) t.join();
```

A ce stade le solde du compte est un int.

jackpot.cpp

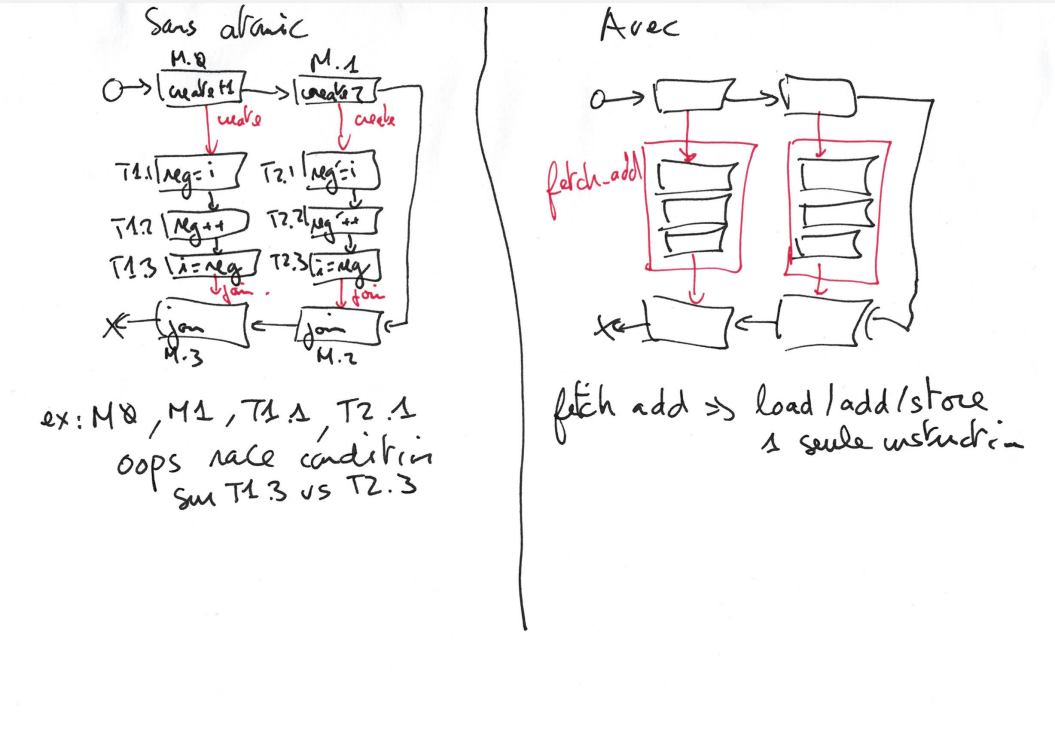
```
1  int main () {
2        vector<thread> threads;
3        threads.reserve(NB_THREAD);
4        Compte c;
5        for (int i=0; i < NB_THREAD ; i++) {
6              // std::ref pour garantir que "c" existera plus longtemps que les threads.
7              threads.emplace_back(jackpot,std::ref(c));
8        }
9        for (auto & t : threads) {
10             t.join();
11       }
12       if (c.getSolde() == N*JP) {
13             cout << "Dragon dort sur trésor " << c.getSolde() << endl;
14             return 0;
15       } else {
16             cout << "Ma cassette, mon sang !" << c.getSolde() << endl;
17             return 1;
18       }
19 }
```

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

Ben ça marche pas bien. On manque d'ordre/contraintes sur les actions des threads. Du coup le résultat est mal déterminé.

Ce dessin explique le problème. On doit décomposer l'incrément du solde en 3 instructions : load, add, store en pratique. On peut donc exhiber des traces problématiques, dès que les deux thread commencent à travailler en même temps (i.e. se passent la main alors qu'ils sont dans le traitement).

On a donc assez probablement un solde inférieur à ce qu'il devrait être.



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.

C'est lié aux causes possibles de commutation : E/S, sleep ou yield explicite, synchro, épuisement de quantum.

Ici c'est l'épuisement de quantum la source de commutation intempestive principale, donc si compter 100 pièces prend moins d'un quantum, vu qu'on ne fait que ça, peu de chance de commuter.

Le programme reste faux (!)

On refuse dans l'UE e.g. les synchro avec des sleep pour "attendre" les autres. Ou de compter sur des effets dûs au quantum/hypothèses en général sur le scheduler.

Ici en multi core, on peut encore observer des effets mauvais, malgré le temps court d'exécution.

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

Donc on utilise `atomic<int>` pour déclarer le solde du compte

jackpot.cpp

```
#include <thread>
#include <iostream>
#include <atomic>

using namespace std;

class Compte {
      atomic<int> solde;
public :
      Compte(int solde=0):solde(solde) {}
      void crediter (size_t val) { solde+=val ;}
      int getSolde() const {return solde;}
};
```

Si créditer utilise bien += pour ajouter au solde, le reste du code n'est pas modifié. Utiliser $solde = solde + val$; est incorrect par contre.

Le dessin du corrigé de la question précédente (à droite) explique le résultat. Les trois actions load/add/store sont groupées en une seule atomiquement (au niveau hardware).

Sans avoir ajouté d'ordre (synchro) entre les instructions des deux thread, on parvient quand même à écarter les exécutions problèmatiques du cas sans protection.

1.3 Section Critique.

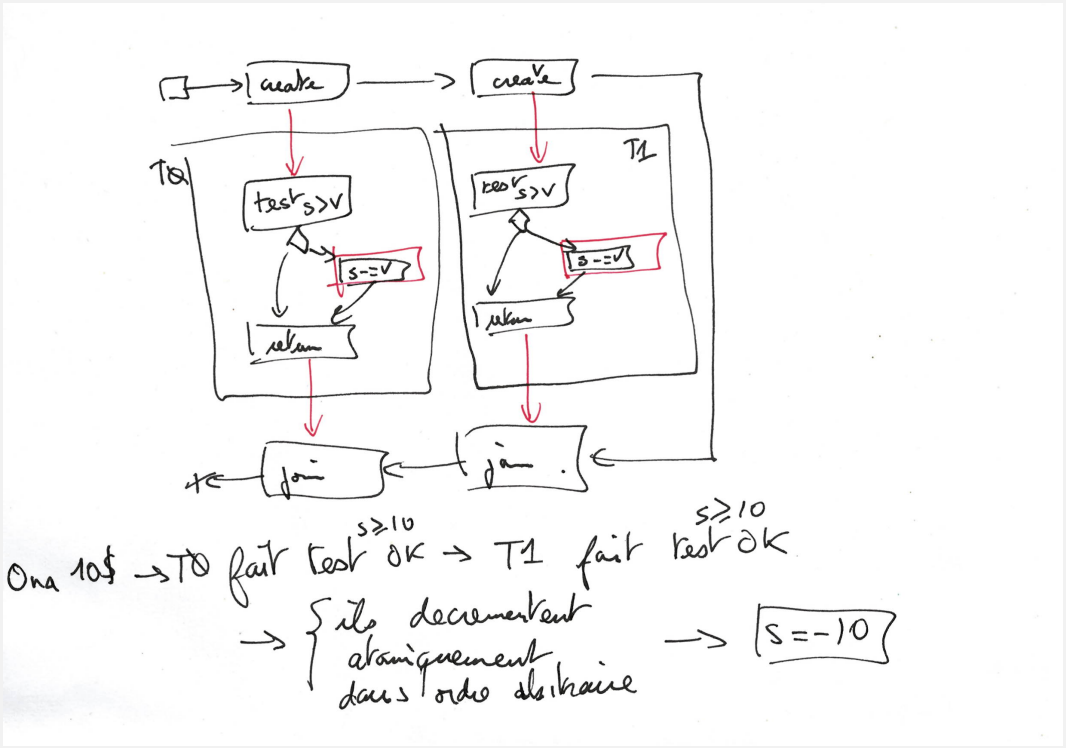
On ajoute au Compte de la question précédente une opération : `bool debiter (int val)` qui doit controler que le solde du compte est suffisant (la banque ne prête pas), et si c'est le cas réduire le solde du montant indiqué. La fonction rend vrai si le débit a eu lieu.

```
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 ?

Attention à la version de debit fournie, elle a une forme un peu bizarre (un seul return...), pour faciliter les dessins et l'introduction du mutex ensuite.

Oui, on tombe joyeusement en négatif. On peut atteindre au plus $-10 * NB\_THREAD$.



Le test et sa mise à jour ne sont pas atomiques.

atomic fournit toujours une garantie sur les exécutions de $solde- = val$, donc on aura une valeur cohérente dans solde à la fin avec le nombre de fois où débiter à rendu *true*.

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

losepotmutex.cpp

```
class Compte {
      mutable mutex m;
      int solde;
public :
      Compte(unsigned int solde=0):solde(solde) {}
      void crediter (unsigned int val) {
            m.lock();
            solde+=val ;
            m.unlock();
      }
      bool debiter (unsigned int val) {
            m.lock();
            bool doit = solde >= val;
            if (doit) {
                  solde-=val ;
            }
            m.unlock();
```

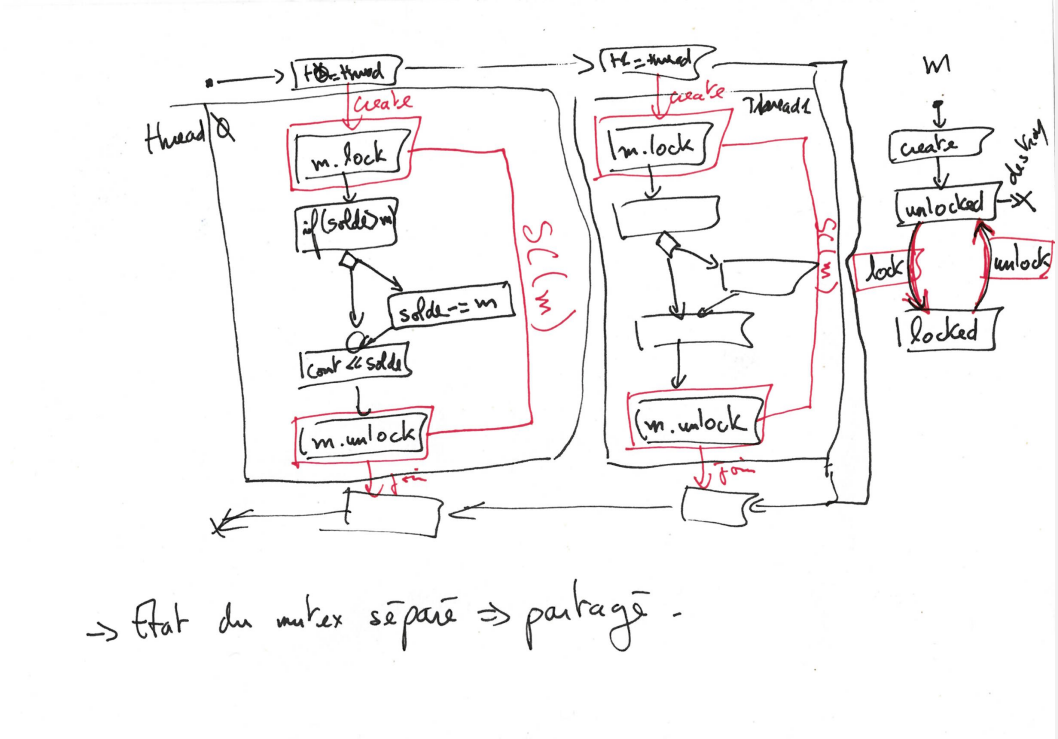
```
18              return doit;
19        }
20        int getSolde() const {
21              m.lock();
22              auto ret = solde;
23              m.unlock();
24              return ret;
25        }
26  };
```

Sur le dessin, on a les deux thread de d'habitude, mais aussi le lock, qui a son propre état interne.

Quand on franchit l'action lock/unlock dans le thread, simultanément il faut franchir l'arc rouge du mutex.

NB: c'est le même thread qui doit faire lock et unlock avec des mutex. cf. Sémaphores pour des synchros où l'on débloque l'autre thread.

On voit aussi ici qu'on doit déclarer le mutex comme étant "mutable", un nouveau mot clé pour contourner la nature "const" des accesseurs en lecture.



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

Donc la création du unique_lock va faire directement un lock sur le mutex (sauf si on demande std::defer). Sa destruction (i.e. son destructeur) fait le unlock automatiquement.

Il protège donc le bloc dans lequel il est déclaré : comme c'est une variable ordinaire, il est stack alloc, et sera détruit en sortant du bloc.

Il évite donc d'oublier les unlock, particulièrement quand la structure de controle est complexe (e.g.

plusieurs “return”, exceptions, . . .).

Sur la version de travail de debiter, ça ne change pas grand chose, mais si on compare debiter2 et debiter3 du corrigé, le mécanisme prend tout son intérêt.

On peut faire le rapprochement avec le bloc “synchronized” de Java.

losepotguard.cpp

```
class Compte {
      // mutable contre le getSolde() qui est const
      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;
      }
      bool debiter2 (unsigned int val) {
            unique_lock<mutex> g(m);
            if (solde >= val) {
                  solde-=val ;
                  return true;
            } else {
                  return false;
            }
      }
      bool debiter3 (unsigned int val) {
            m.lock();
            if (solde >= val) {
                  solde-=val ;
                  m.unlock();
                  return true;
            } else {
                  m.unlock(); // ne pas oublier !!
                  return false;
            }
      }
      int getSolde() const {
            unique_lock<mutex> g(m);
            return solde;
      }
};
```

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 ?

Si le mutex protège déjà les accès au solde, atomic semble ne pas apporter grand chose. La seule chose qu'il garantit c'est les load/store atomic à ce stade : le mutex protège déjà.

Si getSolde n'est pas protégé, a priori, ça ne devrait rien changer : On fait return solde, on lira la valeur avant ou après une écriture (dans tous les cas ce sera OK a priori).

En réalité on est déjà dans du Undefined Behavior ! Il est interdit de lire et écrire à la même adresse simultanément !

C'est ok si : "IF you're reading/writing 4-byte value AND it is DWORD-aligned in memory AND you're running on the I32 architecture, THEN reads and writes are atomic."

En général c'est du UB en C++, atomic fera ce qu'il faut faire sur votre archi matérielle.

Si le solde était un "long double" e.g. qui fasse plus qu'un mot mémoire, on voit mieux les problèmes ! Donc c'est une faute de lire pendant que quelqu'un y écrit (on peut lire la moitié de l'écriture de l'autre).

Il faudrait donc remettre de l'atomic ! Mais vu qu'on a construit un mutex par Compte, autant aller au bout de l'approche, la version mutex n'a pas besoin d'utiliser atomic, mais doit bien protéger tous ses accès avec le mutex, lecture et écriture.

Cela nous donne une forme générale pour une classe MT-safe sans trop réfléchir, similaire à ce qu'on a en annotant une classe Java par "synchronized" sur chaque méthode.

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`.

Donc mutex n'est pas copiable, ce qui rend la version générée par le compilo du ctor par copie non disponible. Il faut locker l'objet source pour faire la copie.

transferts.cpp

```
1  // NB : vector exige Copyable, mais mutex ne l'est pas...
2  Compte(const Compte & other) {
3        other.lock();
4        solde = other.solde;
5        other.unlock();
6  }
```

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.

Fonction exécutée par les threads :

transferts.cpp

```
void transfertJob (int index, Banque & banque) {
      std::cout << "started "<< index << endl;
      for (int i= 0 ; i < 100000; i++) {
            int debite = rand() % banque.size();
            int credite = rand() % banque.size();
            int val = rand() % 70 + 30;
            banque.transfert(debite,credite,val);
            //std::this_thread::sleep_for (std::chrono::milliseconds(rand()%10));
      }
      std::cout << "finished "<< index << endl;
}
```

Et le main :

transferts.cpp

```
int main () {
      using namespace pr;
      const int N = 10;
      ::srand(::time(nullptr));
      vector<thread> threads;
      threads.reserve(N);

      Banque b(200,100);

      for (int i=0; i < N ; i++) {
            // std::ref pour passer au thread par référence
            threads.emplace_back(transfertJob,i,std::ref(b));
            std::cout << "created "<< i << endl;
      }
      for (int i=0; i < N ; i++) {
            threads[i].join();
            std::cout << "joined "<< i << endl;
      }

      return 0;
}
```

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

> A priori oui, notre classe Compte est MT-safe, on peut s'en servir dans n'importe quel contexte MT sans risquer de fautes. D'où l'intérêt des classes de librairie MT safe.

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].

> La solution avec les trois méthodes permet de respecter le contrat d'un Lockable (mutex), i.e. on peut passer des comptes à `std::lock` la version multiple.
>
> Le corrigé adopte cette approche, donc on fera `compte.lock()` plutôt que `compte.getLock().lock()`.
>
> Ajouts dans la classe Compte :

transferts.cpp

```
1   // accès au lock pour la banque
2   void lock () const {
3       m.lock();
4   }
5   void unlock() const {
6       m.unlock();
7   }
8   // pour satisfaire le contrat Lockable exigé par la fonction multi lock
9   bool try_lock () const {
10      return m.try_lock();
11  }
```

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

> Si l'on veut une transaction, il faut empêcher l'accès au compte débité jusqu'à ce que le compte crédité soit mis à jour.
>
> Une solution possible est de prendre les deux locks sur les deux comptes, avant de commencer à faire des opérations, et les relacher à la fin de la transaction.
>
> Donc conceptuellement :
>
> - debiteur.lock()
> - crediteur.lock()
> - if (debiteur.debiter(m)) crediteur.crediter(m);
> - debiteur.unlock() ; crediteur.unlock();

transferts.cpp

```
1   void transfert_deadlock(size_t deb, size_t cred, unsigned int val) {
```

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

```
            Compte & debiteur = comptes[deb];
            Compte & crediteur = comptes[cred];
            debiteur.lock();
            crediteur.lock();
            if (debiteur.debiter(val)) {
                  crediteur.crediter(val);
            }
            debiteur.unlock();
            crediteur.unlock();
      }
```

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

Par défaut on a utilisé un `mutex` qui ne permet pas de double lock, même si le même thread détient déjà le lock. Ce comportement est différent de celui eg. des barrières synchronized de Java.

Ici, on lock le compte, puis on essaie de faire des crédits/débits dessus, donc on reprend le même lock. C'est un deadlock assez trivial.

Pour permettre une réacquisition il nous faut un `recursive_mutex`. En dehors de la déclaration, l'utilisation est le même que le mutex "normal".

Un même thread peut le lock plusieurs fois, mais le nombre d'unlock et de lock doit rester cohérent. Souvent permet une programmation plus simple au niveau client de la classe.

C'est aussi la sémantique proposée en Java pour "synchronized".

Attention à mettre à jour le paramètre générique du unique lock

transferts.cpp

```
class Compte {
      mutable recursive_mutex m;
      int solde;
public :
      Compte(int solde=0):solde(solde) {}
      void crediter (unsigned int val) {
            unique_lock<recursive_mutex> g(m);
            solde+=val ;
      }
      bool debiter (unsigned int val) {
            unique_lock<recursive_mutex> g(m);
```

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.

Donc deux threads T0 et T1 pour l'exemple.

T0 pioche la paire (i,j) et T1 pioche la paire (j,i).

T0 lock i

T1 lock j

T0 et T1 vont maintenants'interbloquer

Tout autre thread essayant de manipuler i ou j va aussi s'échouer.

Pour corriger; la bonne approche est d'ordonner les prises de locks. Par exemple modifer l'acquisition :

- if (i > j) debiteur.lock(); crediteur.lock();
- else crediteur.lock(); debiteur.lock();

Avec un ordre total imposé sur l'ensemble des locks, plus de cycles de deadlock possibles quand on fait des multi-acquisitions.

Le système dispose d'un ordre total sur les mutex/lock ; la fonction *lock*() prend (en varargs, séparés par des virgules) un ensemble de locks, en respectant cet ordre.

On peut donc se contenter de :

- lock (debiteur.getLock(), crediteur.getLock())

et faire confiance au système.

transferts.cpp

```
1   void transfert_multilock(size_t deb, size_t cred, unsigned int val) {
2         Compte & debiteur = comptes[deb];
3         Compte & crediteur = comptes[cred];
4         lock(debiteur,crediteur);
5         if (debiteur.debiter(val)) {
6               crediteur.crediter(val);
7         }
8         debiteur.unlock();
9         crediteur.unlock();
10  }
11  void transfert_manuallock(size_t deb, size_t cred, unsigned int val) {
12        Compte & debiteur = comptes[deb];
13        Compte & crediteur = comptes[cred];
14        if (deb < cred) {
15              debiteur.lock();
16              crediteur.lock();
17        } else {
18              crediteur.lock();
19              debiteur.lock();
20        }
21        if (debiteur.debiter(val)) {
22              crediteur.crediter(val);
23        }
24        debiteur.unlock();
25        crediteur.unlock();
26  }
```

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

```
1   bool comptabiliser (int attendu) const {
2         int bilan = 0;
3         int id = 0;
4         for (const auto & compte : comptes) {
5               if (compte.getSolde() < 0) {
6                     cout << "Compte " << id << " en négatif : " << compte.getSolde() <<
                          endl;
7               }
8               bilan += compte.getSolde();
9               id++;
10        }
```

```
11            if (bilan != attendu) {
12                  cout << "Bilan comptable faux : attendu " << attendu << " obtenu : " <<
                        bilan << endl;
13            }
14            return bilan == attendu;
15      }
```

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).

> Non, le thread comptable peut dépasser le compte i, puis un transfert se fait vers i depuis un compte que le comptable n'a pas encore atteint dans son itération. La compta sera déficitaire.
>
> Le thread comptable a vraiment besoin de bloquer les modifications sur tous les comptes qu'il a déjà traités pendant qu'il travaille.

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

> Donc c'est l'approche Big Fat Lock.
>
> *comptabiliser* est une méthode de la classe Banque, le mutex protège à la fois comptabiliser et l'opération de transfert.

comptableBFL.cpp

```
1
2   class Compte {
3         int solde;
4   public :
5         Compte(int solde=0):solde(solde) {}
6         void crediter (unsigned int val) {
7               solde+=val ;
8         }
9         bool debiter (unsigned int val) {
10              bool doit = solde >= val;
11              if (doit) {
12                    solde-=val ;
13              }
14              return doit;
15        }
16        int getSolde() const {
17              return solde;
18        }
19  };
20
21  class BanqueBFL {
22        mutable mutex m;
23        typedef vector<Compte> comptes_t;
24        comptes_t comptes;
25  public :
26        BanqueBFL (size_t ncomptes, size_t solde) : comptes (ncomptes, Compte(solde)){
27        }
```

```
28        void transfert(size_t deb, size_t cred, unsigned int val) {
29              unique_lock<mutex> l(m);
30              Compte & debiteur = comptes[deb];
31              Compte & crediteur = comptes[cred];
32              if (debiteur.debiter(val)) {
33                    crediteur.crediter(val);
34              }
35        }
36        size_t size() const {
37              unique_lock<mutex> l(m);
38              return comptes.size();
39        }
40        bool comptabiliser (int attendu) const {
41              unique_lock<mutex> l(m);
42              int bilan = 0;
43              int id = 0;
44              for (const auto & compte : comptes) {
45                    if (compte.getSolde() < 0) {
46                          cout << "Compte " << id << " en négatif : " << compte.getSolde()
                                 << endl;
47                    }
48                    bilan += compte.getSolde();
49                    id++;
50              }
51              if (bilan != attendu) {
52                    cout << "Bilan comptable faux : attendu " << attendu << " obtenu : "
                           << bilan << endl;
53              }
54              return bilan == attendu;
55        }
56  };
57
58  void comptableJob (const BanqueBFL & banque, int iterations) {
```

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.

Ben comptabiliser doit soit commencer par lock tous les comptes (violent), soit verrouiller tous les comptes qu'il a déjà comptabilisé.

Du coup, vu qu'on le fait à la main, il faut que l'ordre utilisé dans trasfert soit congruent (i.e. l'ordre à la main, basé sur les indices de comptes est bien).

Surtout ne pas mélanger les deux, les ordres ne seraient pas cohérents menant à un deadlock.

comptableOpt.cpp

```
1  #include <thread>
2  #include <mutex>
3  #include <atomic>
4  #include <iostream>
5  #include <random>
6
7  using namespace std;
8
```

```
9
10 // Comptabilité et transferts
11 namespace exBanque {
12
13 class Compte {
14       // mutable contre le getSolde() qui est const
15       // recursive_mutex à la question 15
16       mutable recursive_mutex m;
17       int solde;
18 public :
19       Compte(int solde=0):solde(solde) {}
20       void crediter (unsigned int val) {
21             lock_guard<recursive_mutex> g(m);
22             solde+=val ;
23       }
24       bool debiter (unsigned int val) {
25             lock_guard<recursive_mutex> g(m);
26             bool doit = solde >= val;
27             if (doit) {
28                   solde-=val ;
29             }
30             return doit;
31       }
32       int getSolde() const {
33             lock_guard<recursive_mutex> g(m);
34             return solde;
35       }
36       // accès au lock pour la banque
37       void lock () const {
38             m.lock();
39       }
40       void unlock() const {
41             m.unlock();
42       }
43       // NB : vector exige Copyable, mais mutex ne l'est pas...
44       Compte(const Compte & other) {
45             other.lock();
46             solde = other.solde;
47             other.unlock();
48       }
49 };
50
51 class Banque {
52       typedef vector<Compte> Comptes;
53       Comptes comptes;
54       const int initial;
55 public :
56       Banque (size_t ncomptes, size_t solde) : comptes (ncomptes, Compte(solde)),initial
          (solde){
57       }
58       // version à la main de l'ordre
59       void transfert(size_t deb, size_t cred, unsigned int val) {
60             Compte & debiteur = comptes[deb];
61             Compte & crediteur = comptes[cred];
62             if (deb < cred) {
63                   debiteur.lock();
64                   crediteur.lock();
65             } else {
66                   crediteur.lock();
```

```
                debiteur.lock();
            }
            if (debiteur.debiter(val)) {
                crediteur.crediter(val);
            }
            debiteur.unlock();
            crediteur.unlock();
      }
      size_t size() const {
            return comptes.size();
      }
      bool comptabiliser () const {
            int attendu = initial * comptes.size();
            int bilan = 0;
            int id = 0;
            for (const auto & compte : comptes) {
                // NB ordre des locks congruent avec celui utilisé dans transfert
                compte.lock();
                if (compte.getSolde() < 0) {
                      cout << "Compte " << id << " en négatif : " << compte.getSolde()
                           << endl;
                }
                bilan += compte.getSolde();
                id++;
            }
            for (const auto & compte : comptes) {
                compte.unlock();
            }
            if (bilan != attendu) {
                cout << "Bilan comptable faux : attendu " << attendu << " obtenu : "
                    << bilan << endl;
            }
            return bilan == attendu;
      }
};

void comptableJob (const Banque & banque, int iterations) {
      for (int iter =0 ; iter < iterations ; iter++) {

            if (! banque.comptabiliser()) {
                cout << "Comptable faché !!" << endl;
            }
            cout << "Bilan " << iter << " fini."<<endl;
            std::this_thread::sleep_for (std::chrono::milliseconds(20));
      }
}


void transfertJob (int index, Banque & banque) {
      std::cout << "started "<< index << endl;
      for (int i= 0 ; i < 100000; i++) {
            int debite = rand() % banque.size();
            int credite = rand() % banque.size();
            int val = rand() % 70 + 30;
            banque.transfert(debite,credite,val);
            //std::this_thread::sleep_for (std::chrono::milliseconds(rand()%10));
      }
      std::cout << "finished "<< index << endl;
}
```

```
124 int createAndWait (int N) {
125       vector<thread> threads;
126       threads.reserve(N);
127 
128       Banque b(200,100);
129 
130       for (int i=0; i < N -2 ; i++) {
131             threads.emplace_back(thread(transfertJob,i,std::ref(b)));
132             std::cout << "created "<< i << endl;
133       }
134       for (int i= N -2 ; i < N ; i++) {
135             threads.emplace_back(thread(comptableJob,std::cref(b),10));
136             std::cout << "created "<< i << endl;
137       }
138       for (int i=0; i < N ; i++) {
139             threads[i].join();
140             std::cout << "joined "<< i << endl;
141       }
142 
143       return 0;
144 }
145 
146 }
147 
148 int main16 () {
149       ::srand(::time(nullptr));
150       return exBanque::createAndWait(10);
151 }
```