TD 3 : 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

Question 1. Ecrire 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.

Question 2. 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 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.

createjoin.cpp

```cpp
#include <thread>
#include <iostream>
#include <cstdlib>
#include <vector>

using namespace std;


// creer, join des threads, sans échanges
namespace ex0 {

void child (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;
}

int createAndWait (int N) {
      vector<thread> threads;
      threads.reserve(N);
      for (int i=0; i < N ; i++) {
```

```
            threads.emplace_back(thread(child,i));
            std::cout << "created "<< i << endl;
      }
      for (int i=0; i < N ; i++) {
            threads[i].join();
            std::cout << "joined "<< i << endl;
      }

      return 0;
}

}

int main_1 () {
      ::srand(::time(nullptr));
      return ex0::createAndWait(3);
}
```

Question 3. 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 :

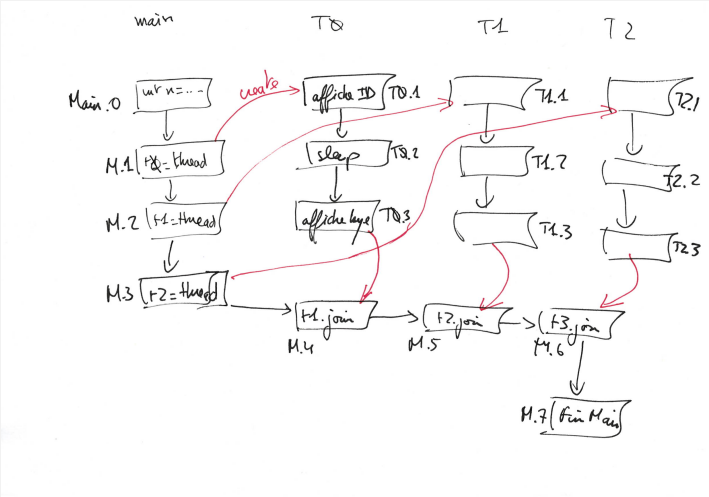
- en noir les actions liées au sein du code d'un thread
- en rouge les synchros inter-thread

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.



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`.

Question 4. Ecrivez cette classe Compte.

Question 5. Ecrivez une fonction `void jackpot (Compte * c)` qui crédite 10000 pièces d'or sur le compte fourni, mais une par une (ding, ding, ding...). Ensuite dans un main, instancier un compte, et créer N threads qui exécutent le code de la fonction Jackpot.

bon le mien est très configurable, mais on peut faire plus simple.

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.

A ce stade le solde du compte est un int.

jackpot.cpp

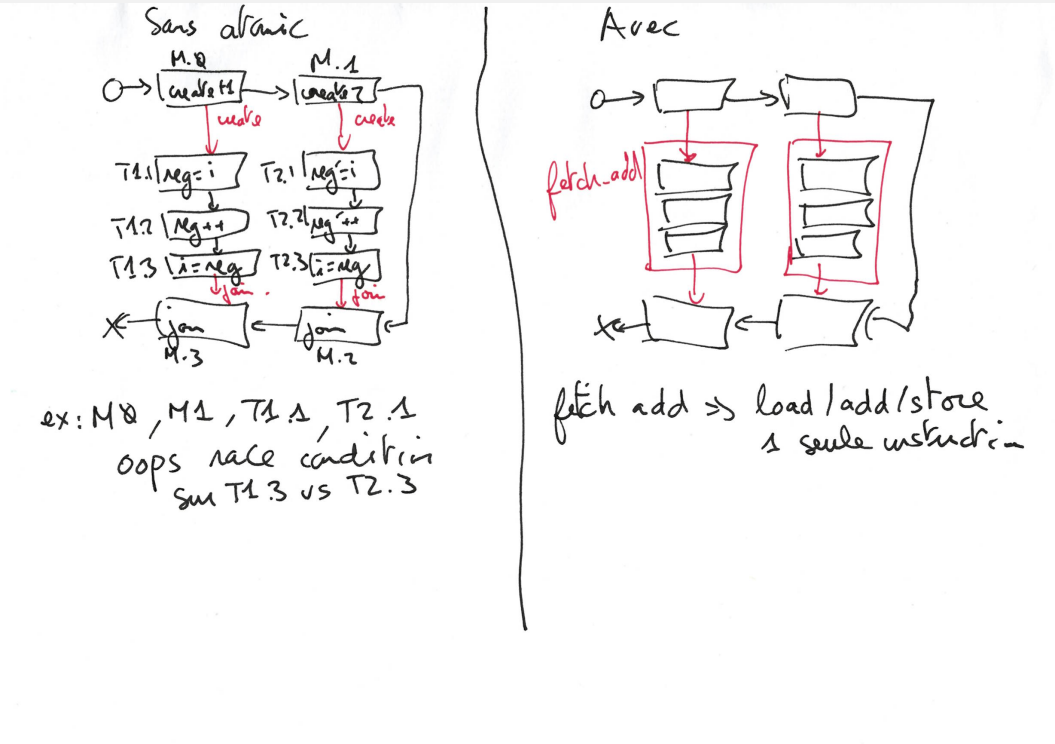
```cpp
#include <thread>
#include <atomic>
#include <iostream>
#include <random>

using namespace std;

namespace jp {

const int JP = 10000;
const int NB_THREAD = 10;

class Compte {
      atomic<int> solde;
// int solde;
public :
      Compte(int solde=0):solde(solde) {}
```

```
      void crediter (size_t val) { solde+=val ;}
      int getSolde() const {return solde;}
};

void child (int index, Compte &compte) {
      std::cout << "started "<< index << endl;
      for (int i= 0 ; i < JP; i++) {
            compte.crediter(1);
      }
      std::cout << "finished "<< index << endl;
}

int createAndWait (int N) {
      vector<thread> threads;
      threads.reserve(N);
      Compte c;
      for (int i=0; i < N ; i++) {
            threads.emplace_back(thread(child,i,std::ref(c)));
            std::cout << "created "<< i << endl;
      }
      for (int i=0; i < N ; i++) {
            threads[i].join();
            std::cout << "joined "<< i << endl;
      }
      if (c.getSolde() == N*JP) {
            cout << "Dragon dort sur trésor" << endl;
      } else {
            cout << "Ma cassette, mon sang !" << c.getSolde() << endl;
      }
      return 0;
}

}

int main6 () {
      // donc on pase time a srand pour poser une graine qui varie d'une execution à l'
          autre
      ::srand(::time(nullptr));
      return jp::createAndWait(jp::NB_THREAD);
}
```

Question 6. Quel sera le solde du compte à la fin du programme pour N=10 ?

Ben ça marche pas bien. On manque d'ordre/contrainte sur les actions des threads.

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).



Question 7. 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 8. 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 (l.14 /15 du corrigé).

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.

Question 9. On reprend l'exemple du Jackpot, mais cette fois-ci en débit, le LosePot retire 1000 par paquets de 10. On lance N thread qui exécutent cette fonction ; le compte peut-il tomber en négatif ? Quelle garantie fournit `atomic` ici ?

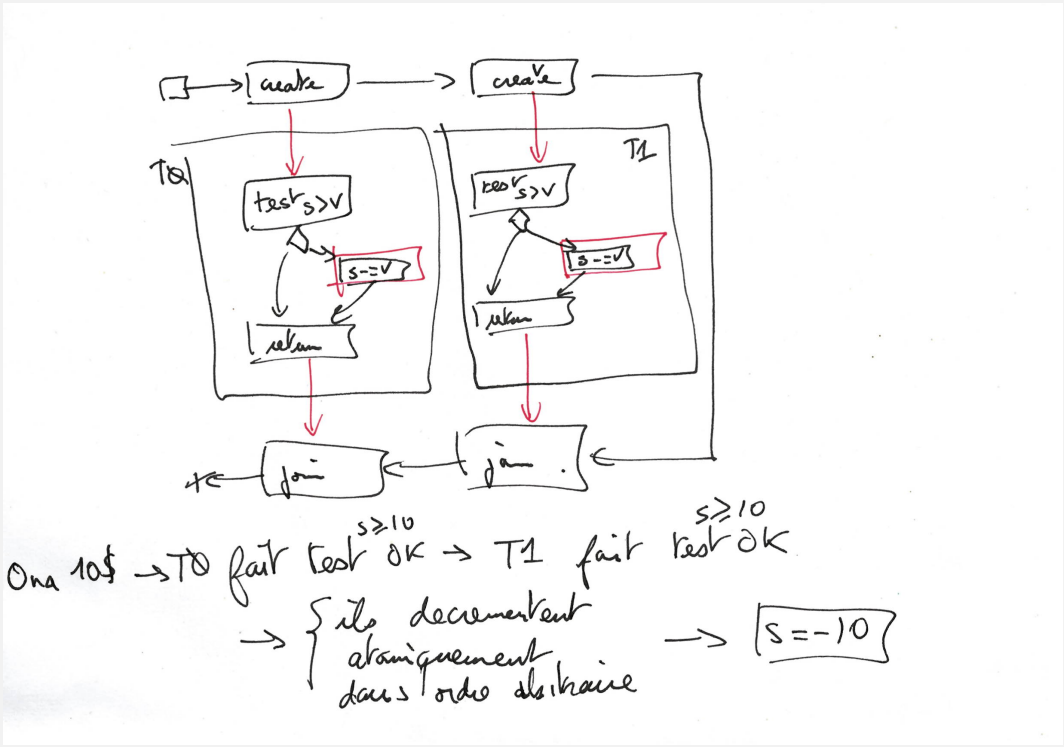
Attention à la version de debit fournie, elle a une forme un peu bizarre (un seul return...), pour faciliter les dessins et l'introduction du mutex ensuite, donc y coller d'assez près.

losepot.cpp

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

using namespace std;

namespace lp {

const int JP = 10000;
const int NB_THREAD = 10;

class Compte {
      atomic<int> solde;
// int solde;
public :
      Compte(int solde=0):solde(solde) {}
      void crediter (unsigned int val) { solde+=val ;}
      bool debiter (unsigned int val) {
            bool doit = solde >= val;
            if (doit) {
                  solde-=val ;
            }
            return doit;
      }
      int getSolde() const {return solde;}
};

void child (int index, Compte &compte) {
      std::cout << "started "<< index << endl;
      int debits = 0;
      for (int i= 0 ; 10 * i < JP; i++) {
            if (compte.debiter(10)) {
                  debits++;
            }
      }
      std::cout << "finished "<< index << " : " << debits << endl;
}

int createAndWait (int N) {
      vector<thread> threads;
      threads.reserve(N);
      Compte c(15000);
      for (int i=0; i < N ; i++) {
            threads.emplace_back(thread(child,i,std::ref(c)));
            std::cout << "created "<< i << endl;
```

```
	}
	for (int i=0; i < N ; i++) {
		threads[i].join();
		std::cout << "joined "<< i << endl;
	}
	if (c.getSolde() < 0) {
		cout << "Négatif !! Mes sous !" << c.getSolde()<< endl;
	} else {
		cout << "Ok, compte vide :" << c.getSolde() << endl;
	}
	return 0;
}

}

int main12 () {
	::srand(::time(nullptr));
	return lp::createAndWait(lp::NB_THREAD);
}
```

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

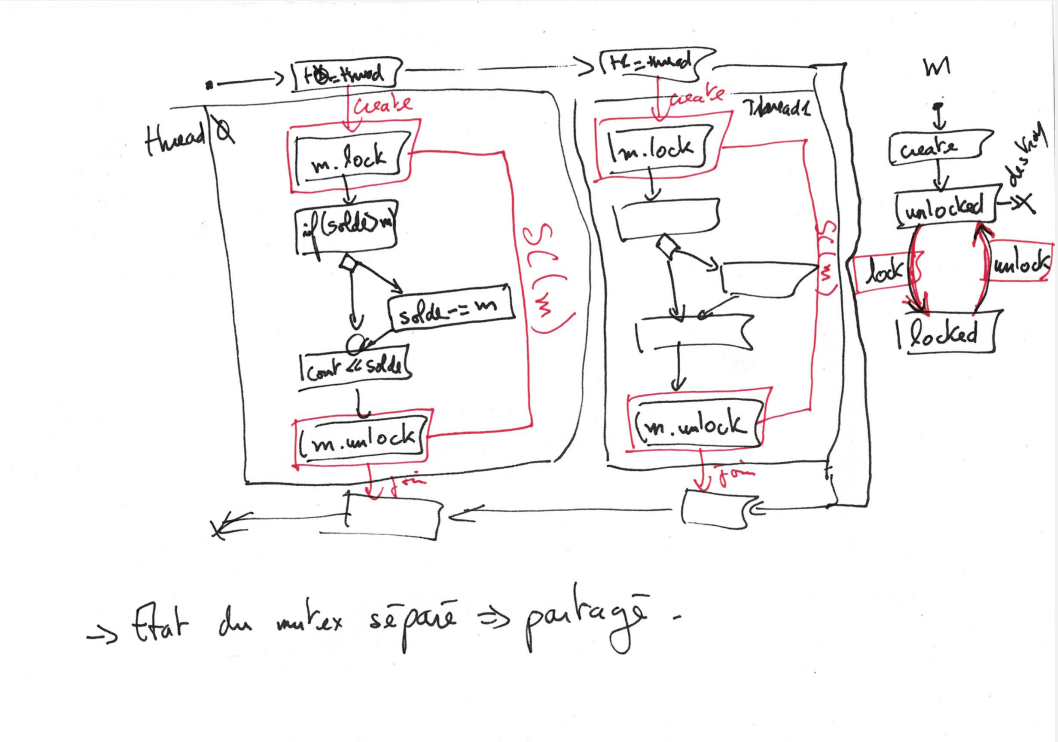
losepotmutex.cpp

```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();
            return doit;
      }
      int getSolde() const {
            m.lock();
            auto ret = solde;
            m.unlock();
            return ret;
      }
};
```

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.



Question 11. Utiliser un `lock_guard` plutôt que l'API lock/unlock. Comparer les syntaxes.

Donc la création du lock_guard va faire un lock sur le mutex. Sa destruction fait le unlock.

Il protège donc le bloc dans lequel il est déclaré. Il évite d'oublier les unlock.

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) {
            lock_guard<mutex> g(m);
            solde+=val ;
      }
      bool debiter (unsigned int val) {
            lock_guard<mutex> g(m);
            bool doit = solde >= val;
            if (doit) {
                  solde-=val ;
            }
```

```
            return doit;
      }
      bool debiter2 (unsigned int val) {
            lock_guard<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 {
            lock_guard<mutex> g(m);
            return solde;
      }
```

Question 12. 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 !

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 devrait ne pas utiliser atomic, mais bien protéger tous ses accès avec, lecture et écriture.

1.4 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. On 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 13. L'utilisation de la classe Compte dans un `vector` pose un problème : le compilateur se plaint que la classe n'est pas copiable. Expliquez le problème et corrigez le.

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

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

Question 14. Le comportement sera-t-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.

transferts.cpp

```
class Banque {
      typedef vector<Compte> Comptes;
      Comptes comptes;
public :
      Banque (size_t ncomptes, size_t solde) : comptes (ncomptes, Compte(solde)){
      }
      void transfert_basic(size_t deb, size_t cred, unsigned int val) {
            Compte & debiteur = comptes[deb];
            Compte & crediteur = comptes[cred];
            if (debiteur.debiter(val)) {
                  crediteur.crediter(val);
            }
      }
```

Et le main :

transferts.cpp

```
            debiteur.unlock();
            crediteur.unlock();
      }
      size_t size() const {
            return comptes.size();
      }

};

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_multilock(debite,credite,val);
            //std::this_thread::sleep_for (std::chrono::milliseconds(rand()%10));
      }
      std::cout << "finished "<< index << endl;
}

int createAndWait (int N) {
      vector<thread> threads;
      threads.reserve(N);

      Banque b(200,100);

      for (int i=0; i < N -2 ; i++) {
            threads.emplace_back(thread(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;
}

}

int main55 () {
      ::srand(::time(nullptr));
      return ex3::createAndWait(10);
}
```

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 observatuer ne doit pouvoir voir un des comptes débités et l'autre pas encore crédité.

Question 15. Proposez une stratégie de synchronisation pour permettre ce comportement transactionnel. On ajoute un accesseur `mutex & getMutex()` au compte pour permettre de l'écrire, ou l'on définit les trois méthode `void lock () const, void unlock() const, bool try_lock () const` par délégation sur le mutex stocké.

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.

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.getLock().lock()
- crediteur.getLock().lock()
- if (debiteur.debiter(m)) crediteur.crediter(m);

- debiteur.getLock().unlock() ; crediteur.getLock().unlock();

transferts.cpp

```
class Compte {
      // mutable contre le getSolde() qui est const
      // recursive_mutex à la question 15
      mutable recursive_mutex m;
      int solde;
public :
      Compte(int solde=0):solde(solde) {}
      void crediter (unsigned int val) {
            lock_guard<recursive_mutex> g(m);
            solde+=val ;
      }
      bool debiter (unsigned int val) {
            lock_guard<recursive_mutex> g(m);
            bool doit = solde >= val;
            if (doit) {
                  solde-=val ;
            }
            return doit;
      }
      int getSolde() const {
            lock_guard<recursive_mutex> g(m);
            return solde;
      }
      // accès au lock pour la banque
      void lock () const {
            m.lock();
      }
      void unlock() const {
            m.unlock();
      }
      // pour satisfaire le contrat Lockable exigé par la fonction multi lock
      bool try_lock () const {
            return m.try_lock();
      }
      // NB : vector exige Copyable, mais mutex ne l'est pas...
      Compte(const Compte & other) {
            other.lock();
            solde = other.solde;
            other.unlock();
      }
};
```

transferts.cpp

```
      void transfert_deadlock(size_t deb, size_t cred, unsigned int val) {
            Compte & debiteur = comptes[deb];
            Compte & crediteur = comptes[cred];
            debiteur.lock();
            crediteur.lock();
            if (debiteur.debiter(val)) {
                  crediteur.crediter(val);
            }
            debiteur.unlock();
            crediteur.unlock();
      }
```

Question 16. 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 warning.

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 lock guard

transferts.cpp

```
class Compte {
      // mutable contre le getSolde() qui est const
      // recursive_mutex à la question 15
      mutable recursive_mutex m;
      int solde;
public :
      Compte(int solde=0):solde(solde) {}
      void crediter (unsigned int val) {
            lock_guard<recursive_mutex> g(m);
            solde+=val ;
```

Question 17. 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.getLock().lock(); crediteur.getLock().lock();
- else crediteur.getLock().lock(); debiteur.getLock().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

```
      void transfert_multilock(size_t deb, size_t cred, unsigned int val) {
            Compte & debiteur = comptes[deb];
```

```
            Compte & crediteur = comptes[cred];
            lock(debiteur,crediteur);
            if (debiteur.debiter(val)) {
                  crediteur.crediter(val);
            }
            debiteur.unlock();
            crediteur.unlock();
      }
      void transfert_manuallock(size_t deb, size_t cred, unsigned int val) {
            Compte & debiteur = comptes[deb];
            Compte & crediteur = comptes[cred];
            if (deb < cred) {
                  debiteur.lock();
                  crediteur.lock();
            } else {
                  crediteur.lock();
                  debiteur.lock();
            }
            if (debiteur.debiter(val)) {
                  crediteur.crediter(val);
            }
            debiteur.unlock();
            crediteur.unlock();
      }
```

1.5 Comptabilité

La banque possède aussi un thread qu'elle detache à la création pour faire la comptabilité. Ce thread itère sur les comptes en sommant leur solde, et vérifie que la somme vaut bien $SOLDEINITIAL*K$. Sinon il lève une alerte sur la console.

Question 18. Le thread comptable sera-t-il satisfait avec les synchronisations actuelles ? Expliquez pourquoi.

BFL.cpp

```
      bool comptabiliser () const {
            lock_guard<mutex> l(m);
            int attendu = initial * comptes.size();
            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) {
                  cout << "Bilan comptable faux : attendu " << attendu << " obtenu : "
                      << bilan << endl;
            }
            return bilan == attendu;
      }
```

```
};

void comptableJob (const BanqueBFL & 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, BanqueBFL & 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;
}

int createAndWait (int N) {
      vector<thread> threads;
      threads.reserve(N);

      BanqueBFL b(200,100);

      for (int i=0; i < N -2 ; i++) {
            threads.emplace_back(thread(transfertJob,i,std::ref(b)));
            std::cout << "created "<< i << endl;
      }
      for (int i= N -2 ; i < N ; i++) {
            threads.emplace_back(thread(comptableJob,std::cref(b),10));
            std::cout << "created "<< i << endl;
      }
      for (int i=0; i < N ; i++) {
            threads[i].join();
            std::cout << "joined "<< i << endl;
      }

      return 0;
}
```

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 pendant qu'il travaille.

Question 19. Ajoutez un mutex dans la banque, et les synchronisations utiles pour que le thread comptable obtienne les bons résultats.

Donc c'est l'approche Big Fat Lock.

comptabiliser devient une méthode de la classe Banque, le mutex protège à la fois comptabiliser et l'opération de transfert.

comptableBFL.cpp

```
class Compte {
      int solde;
public :
      Compte(int solde=0):solde(solde) {}
      void crediter (unsigned int val) {
            solde+=val ;
      }
      bool debiter (unsigned int val) {
            bool doit = solde >= val;
            if (doit) {
                  solde-=val ;
            }
            return doit;
      }
      int getSolde() const {
            return solde;
      }
};

class BanqueBFL {
      mutable mutex m;
      typedef vector<Compte> Comptes;
      Comptes comptes;
      const int initial;
public :
      BanqueBFL (size_t ncomptes, size_t solde) : comptes (ncomptes, Compte(solde)),
          initial(solde){
      }
      void transfert(size_t deb, size_t cred, unsigned int val) {
            lock_guard<mutex> l(m);
            Compte & debiteur = comptes[deb];
            Compte & crediteur = comptes[cred];
            if (debiteur.debiter(val)) {
                  crediteur.crediter(val);
            }
      }
      size_t size() const {
            lock_guard<mutex> l(m);
            return comptes.size();
      }
      bool comptabiliser () const {
            lock_guard<mutex> l(m);
            int attendu = initial * comptes.size();
            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) {
```

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

Question 20. 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.

Ben comptabiliser doit commencer par lock tous les comptes, simplement.
Attention, soit on le fait à la main, dans le même ordre utilisé à la main précédemment dans trasfert.
Soit il faut lock multiple en passant tous les locks à la fonction lock du système.
Surtout ne pas mélanger les deux, les ordres ne seraient pas cohérents menant à un deadlock.

comptableOpt.cpp

```cpp
#include <thread>
#include <mutex>
#include <atomic>
#include <iostream>
#include <random>

using namespace std;


// Comptabilité et transferts
namespace exBanque {

class Compte {
      // mutable contre le getSolde() qui est const
      // recursive_mutex à la question 15
      mutable recursive_mutex m;
      int solde;
public :
      Compte(int solde=0):solde(solde) {}
      void crediter (unsigned int val) {
            lock_guard<recursive_mutex> g(m);
            solde+=val ;
      }
      bool debiter (unsigned int val) {
            lock_guard<recursive_mutex> g(m);
            bool doit = solde >= val;
            if (doit) {
                  solde-=val ;
            }
            return doit;
      }
      int getSolde() const {
            lock_guard<recursive_mutex> g(m);
            return solde;
      }
      // accès au lock pour la banque
      void lock () const {
            m.lock();
```

```
      }
      void unlock() const {
            m.unlock();
      }
      // NB : vector exige Copyable, mais mutex ne l'est pas...
      Compte(const Compte & other) {
            other.lock();
            solde = other.solde;
            other.unlock();
      }
};

class Banque {
      typedef vector<Compte> Comptes;
      Comptes comptes;
      const int initial;
public :
      Banque (size_t ncomptes, size_t solde) : comptes (ncomptes, Compte(solde)),initial
          (solde){
      }
      // version à la main de l'ordre
      void transfert(size_t deb, size_t cred, unsigned int val) {
            Compte & debiteur = comptes[deb];
            Compte & crediteur = comptes[cred];
            if (deb < cred) {
                  debiteur.lock();
                  crediteur.lock();
            } else {
                  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;
}
int createAndWait (int N) {
      vector<thread> threads;
      threads.reserve(N);

      Banque b(200,100);

      for (int i=0; i < N -2 ; i++) {
            threads.emplace_back(thread(transfertJob,i,std::ref(b)));
            std::cout << "created "<< i << endl;
      }
      for (int i= N -2 ; i < N ; i++) {
            threads.emplace_back(thread(comptableJob,std::cref(b),10));
            std::cout << "created "<< i << endl;
      }
      for (int i=0; i < N ; i++) {
            threads[i].join();
            std::cout << "joined "<< i << endl;
      }

      return 0;
}

}

int main16 () {
      ::srand(::time(nullptr));
      return exBanque::createAndWait(10);
}
```

TME 3 : Thread, mutex

Objectifs pédagogiques :

- **thread**
- **atomic simples**
- **mutex**

1.1 Synchronisations de la Banque

Question 1. En suivant l'énoncé du TD3, implanter les exemples de synchronisation. On n'hésitera pas à copier coller et changer de namespace au fil des questions. Pour chaque exemple, lancer plusieurs fois l'exemple en faisant varier le nombre de threads et les constantes de manière à observer des exécutions diverses.

Activer `thread` dans Eclipse CDT.
Par défaut le link n'active pas l'option adaptée `-pthread`.
Pour le régler le compilateur, on doit faire un réglage pour chaque projet séparément. En partant d'un clic droit sur le projet, accéder à ses propriétés:

- "Project properties -> C/C++ Build -> Settings"
- Sous le "GCC C++ linker" on trouve une rubrique "General"
- Cocher : "Support for pthread (-pthread)" dans la liste

Attention, il faut faire ce réglage dans tout nouveau projet utilisant des thread.