Introduction au Temps Réel (Embarqué)

Exemples et Exercices pour cours 1

Alexander Saidi
ECL - LIRIS

Janvier 2017

Un premier exemple

Exemple 1 : Connaitre le nombre de *threads* disponibles sur votre machine.

```
// Ex1.cpp
// Pour compiler au clavier :  g++ -Wall -std=c++11 Ex1.cpp -lpthread
#include <thread>
#include <iostream>

void doSomeWork( void ){
    std::cout << "hello from thread ..." << std::endl;
    return;
}

int main(){
    std::thread t( doSomeWork );
    t.join();

    std::cout << "Std : nb  threads disponibles dans ce système : " ;
    std::cout<< std::thread::hardware_concurrency()<<std::endl;

    return 0;
}
// Résultats :  2 .. 8 threads disponibles
```

Deuxième exemple

- Threads sur une Fonction et une Fonction-objet.

```
// Ex2.cpp
#include <iostream>
#include <thread>
using namespace std;

void fonction_simple() {                       // une simple fonction
  cout << "fonction_simple est lancée \n";
  for (int i : {1, 2, 3, 4, 5})               // Remarquer "for"
    cout << '(' << i << ") ";
  cout << endl;
}

struct Fonction_objet {                        // Une Fonction_objet
  void operator()(){                           // opérateur d'appel de Fonction_objet
      cout << "Fonction_objet est lancé \n";
      for (int i : {10, 20, 30, 40, 50})  cout << '(' << i << ") ";
      cout << endl;
  }
};

int main(){
  thread t1 {fonction_simple};                 // ou thread t1(fonction_simple)
  thread t2 {Fonction_objet()};                // remarquer les ()

  t1.join();     // attendre  t1
  t2.join();     // attendre t2
  return 0;
}
```

Deuxième exemple (suite)

```
/* Plusieurs traces
f est lancé
(F est lancé                <<-- remarquer la '('
(10) (20) (30) (40) (50)
1) (2) (3) (4) (5)          <<-- fermée ici

OU

f est lancé
(1) (2) (3) (4) (5)
F est lancé
(10) (20) (30) (40) (50)

OU

f est lancé
(F est lancé
(10) (1) 20() 2() 30) ((40) 3() (504) ) (
5)

*/
```

• On remarque bien que les 2 threads font appel à "cout" et peuvent très bien s'entremêler dans leurs écritures à l'écran.

Paramètres d'un thread

- Passage de paramètres à un thread.

```
// Ex3.cpp
#include <iostream>
#include <thread>

void fonction_sans_parametre() {
  std::cout << "fonction_sans_parametre \n ";
  for (int j=0; j< 10; j++) std::cout << "fonction_sans_parametre : j = " << j << '\n';
}
void fonction_avec_parametre(int x){
  std::cout << "fonction_avec_parametre : i received x = " << x << std::endl;
  for (int i=0; i< x; i++) std::cout << "fonction_avec_parametre : i = " << i << std::endl;
}
int main() {
  std::thread first (fonction_sans_parametre);
  std::thread second (fonction_avec_parametre,10);
  std::thread third (fonction_avec_parametre,20);

  std::cout << "main, fonction_sans_parametre and fonction_avec_parametre now execute concurrently
    ...\n";

  // synchroniser les threads:
  first.join();                    // pauses until first finishes
  second.join();                   // pauses until second finishes
  third.join();
  std::cout << "fonction_sans_parametre et fonction_avec_parametre ont fini.\n";
  return 0;
}
```

Valeur calculée par un thread

- Solutions pour récupérer une valeur calculée par une tâche T :
  1- On utilise les paramètres (par référence) modifiés par la tâche T,
  2- La tâche T calcule et transmet une valeur (par *return*),
  3- Passage par une mémoire partagée (ou par une variable globale).

Méthode 1- un paramètre est passé (par référence) obligatoirement par le mot clef **"ref"** à la fonction *modifier_v* qui modifie le vecteur **v**.
La fonction *afficher* (ci-dessous) se contente d'afficher un vecteur.

```
// Ex4.cpp
#include <thread>
#include <iostream>
#include <future>
#include <vector>
using namespace std;

void afficher(vector<double> v){   // ne modifie pas 'v'
    for (double x : v)             // Cette boucle 'for'
      std::cout << x << ", ";      // est utile si v est un vector<T>
   cout << endl;
}
```

Valeur calculée par un thread (suite)

```
void modifier_v(vector<double> &v){ // modifie 'v'
   for (double& x : v) // Remarquer l'itération sur un container
     x+=100;
}

int main(){
  vector<double> vec1 {1,2,3,4,5,6,7,8,9}; // initialisation

  thread t1 {modifier_v, ref(vec1)}; // ref obligatoire, on attend "&v"
  t1.join();

  afficher(vec1);   // pour voir les modifications
  return 0;
}

//101, 102, 103, 104, 105, 106, 107, 108, 109,
```

Récupération de la valeur de retour

Méthode 2- récupérer le résultat d'une fonction exécutée par un *thread*.

- ◦ La fonction retourne une valeur (par *return*)
- ➛ Cette valeur est capturée par la fonction **get()**.

Dans l'exemple ci-dessous, on fait la somme des éléments d'un vecteur à l'aide de deux tâches : chacune fait la somme d'une moitié du vecteur.

```cpp
// Ex5.cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <future>

int somme_d_une_tranche(std::vector<int>::iterator deb, std::vector<int>::iterator fin){
  return std::accumulate(deb, fin, 0);  // accumulate est prédéfinie
}

int main(){
    std::vector<int> v(10000, 1);    // un vecteur de 10000 "1"s

    int taille_du_vecteur=v.end()-v.begin(); // taille de v

    // iterateur sur le milieu du vecteur
    std::vector<int>::iterator milieu =  v.begin()+taille_du_vecteur/2;
```

Récupération de la valeur de retour (suite)

```
    // on fait la somme de v par deux taches
    // chaque tâche fait la somme d'une moitié
    auto thread_une_moitie = std::async(std::launch::async,
                             somme_d_une_tranche, v.begin(), milieu);
    auto thread_autre_moitie = std::async(std::launch::async,
                             somme_d_une_tranche, milieu,v.end());

    std::cout << "La somme sera ";
    cout << thread_une_moitie.get()+thread_autre_moitie.get();
    cout << '\n';
    return 0;
}
// La somme sera 10000
```

- Noter la création des threads (voir ci-après).
- Le mot clé **auto** : on demande à C++ de décider du type !
  - *auto x=10;* C++ choisit le type (int) de x.

Threads : Lacement immédiat ou différé

- Dans le lancement d'un thread **th** par

  std::thread th(fonc,params);

th est lancé immédiatement (dès que possible, décidé par le (RT)OS).

☞ Un appel à **join()** permet d'attendre la fin de **th**.

- Par contre, dans (les 3 syntaxes ci-dessous sont équivalentes) :

  std::future<T> res = std::async(std::launch::async, fonc,params);

  Ou *auto res = std::async(std::launch::async, fonc,params);*

  Ou *std::future<T> res (std::async(fonc,params));*

le thread en charge de fonc() est lancé de manière asynchrone :

➞ C-à-d. : lancé maintenant ou plus tard ou jamais !

☞ Demande express de lancement + récupération d'une valeur de retour se font par la fonction prédéfinie **get()**. . . . ⇝

Threads : Lacement immédiat ou différé (suite)

- Enfin, dans

std::future<T> res (std::launch::deffered(fonc, params));

on procède à un lancement différé du thread exécutant *fonc*.

☞ Demande express de lancement + récupération d'une valeur de retour se font par **get()**.

Avec async / deffered : un appel à **join()** peut nous faire attendre.

- Avec deffered : un appel à **get()** permet de demander explicitement que le thread associé à *fonc* soit lancé et qu'il nous délivre ses résultats

☞ Même si *fonc()* renvoie rien (= *void*), on attendra !

- Rappel : un appel habituel simple (sans thread) à *fonc* est synchrone et l'exécution a lieu immédiatement (et on attend la fin de *fonc*).

- Exemple de lancements : $\cdots \rightsquigarrow$

Threads : Lacement immédiat ou différé (suite)

```cpp
// Ex6.cpp
#include <iostream>     // pour std::cout
#include <future>       // pour std::async, std::future, std::launch
#include <chrono>       // pour std::chrono::milliseconds
#include <thread>       // pour std::this_thread::sleep_for

void affiche_10_valeurs_avec_attente (char c, int ms) {
  for (int i=0; i<10; ++i) {
    std::this_thread::sleep_for (std::chrono::milliseconds(ms));
    std::cout << c;
  }
}

int main (){
  std::cout << "with launch::async:\n";
  std::future<void> foo = std::async (std::launch::async,
                                affiche_10_valeurs_avec_attente,'*',100);
  std::future<void> bar = std::async (std::launch::async,
                                affiche_10_valeurs_avec_attente,'@',200);

  // async "get" (wait for foo and bar to be ready and terminate):
  foo.get();
  bar.get();
  std::cout << "\n\n";

  std::cout << "with launch::deferred:\n";
  foo = std::async (std::launch::deferred,
                                affiche_10_valeurs_avec_attente,'*',100);
  bar = std::async (std::launch::deferred,
                                affiche_10_valeurs_avec_attente,'@',200);
```

Threads : Lacement immédiat ou différé (suite)

```
  // deffered "get" (perform the actual calls):

  foo.get();
  bar.get();
  std::cout << '\n';

  return 0;
}

// with launch::async:        *@**@**@**@**@*@@@@@

// with launch::deferred:     **********@@@@@@@@@@
```

☞ Remarquer l'entrelacement des caractères pour en déduire l'ordre de lancement des threads.

Méthode 3- utilisation d'une variable globale / partagée.

- ➢ Soulève d'autres questions !
- ➢ Voir l'exemple (variable globale) suivant.

Besoin de synchronisation : Un exemple trivial

- Incrémentation d'une variable globale par 2 tâches : la valeur finale de la variable globale est non déterministe car les *var_globale++* s'entrelacent :

```
// Ex7.cpp
#include <thread>
#include <iostream>

int var_globale=0;

void incrementer( int val ) {
    std::cout << "thread incrementation lance : val = " << val << std::endl;
    for (int i=0; i< val; i++) var_globale++;
}

int main( int argc, char *argv[] ) {
    std::thread t1( incrementer, 30000 );
    std::thread t2( incrementer, 30000 );
    t1.join(); t2.join();

    std::cout << "Fin des threads : la variable globale =  " << var_globale  << std::endl;
    return 0;
}

/* Ex : plusieurs exécutions avec le parammètre d'appel = 30000

thread incrementation lance : val = 30000
thread incrementation lance : val = 30000
Fin des threads : la variable globale vaut  51198
```

Besoin de synchronisation : Un exemple trivial (suite)

```
thread incrementation lance : val = 30000
thread incrementation lance : val = 30000
Fin des threads : la variable globale vaut  60000

thread incrementation lance : val = 30000
thread incrementation lance : val = 30000
Fin des threads : la variable globale vaut  47120
*/
```

➙ Comportement non déterministe.

Les mécanismes de Synchronisation

- Explication (détails d'une exécution) via l'exemple $N \leftarrow N+1$ (ou $N++$)
- Problème d'accès concurrent aux ressources.
  - ➙ Synchronisation et accès en concurrence.
- Il existe plusieurs solutions pour la synchronisation (voir aussi cours).
- Problème d'une **section critique**.
  - ➙ **Sémaphores** (peuvent être logiciels !) et mécanismes similaires
  - ➙ **Messages**
  - ➙ **Moniteurs**
  - ➙ **Solution matérielle** spécifique (TAS, verrouillage Bus, Masquage It.)
  - ➙ ....
- Sémaphores et opérateurs **P** et **V**...

Gestion d'accès en exclusion mutuelle

- **Mutex** (sémaphore d'exclusion mutuelle) avec une valeur init = 1.
  ➥ Avant d'accéder à la ressource (dans une *section critique*), une tâche prend un *ticket* (comme dans un parking) qu'elle rendra à la fin.
  ➥ Les attentes doivent être courtes pour permettre de satisfaire les contraintes temporelles : **on sort vite de la section critique.**

- Un *mutex* fonctionne comme l'instruction TAS.

- **TAS** : instruction élémentaire (insécable, atomique, exécutée en 1 cycle) pour lire et écrire un mot mémoire.
  ➛ Voir plus loin (TAS).

Sémaphores pour la synchronisation

- Un sémaphore **binaire** (0 ticket) peut servir à synchroniser deux tâches.
  - ➙ La valeur initial du sémaphore = 0
  - ➙ Tant que l'un n'aura pas produit un appel à V, l'autre ne passera pas.

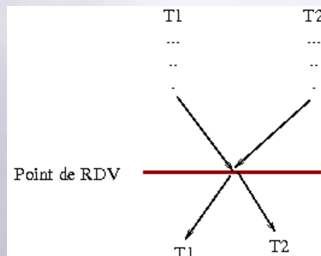
- **Exemple** pour 2 tâches :

  Bool First=True;

  Semaphore S init 0;

  Code RDV de T_i :

```
if (First)
    First=False; P(S);
else
    First=True; V(S);
```



☞ Protection du booléen *First* ? (car accès concurrent à First possible).

Sémaphores pour la synchronisation (suite)

- Exemple : le schéma précédent est traduit dans le code suivant :
  ➙ Les tâches T1 et T2 sont deux agents qui se donnent RDV pour échanger chacune une partie d'un code secret.

☞ Avec C++11, il n'y a pas de sémaphore initialisable à 0 (mais seulement 1). Par contre, on peut les réaliser soi-même avec différents outils disponibles (voir plus loin pour leur réalisation).

- **Pseudo-code :**

Sémaphores pour la synchronisation (suite)

```
// Les entetes ....

Semaphore synchro_RDV(0);   // Voir "Sémaphores binaires" plus loin

std::mutex protege_code;  // protège l'accès aux code secret

std::string secret="";
bool First=true;


void demander_RDV();   // définie ci-dessous

void agent1(int num){
  for (int i=0; i<num; i++) {
    std::cout.put('*').flush();
    std::this_thread::sleep_for(std::chrono::milliseconds(num));
  }
  demander_RDV();
  std::lock_guard<std::mutex>lock(protege_code);    // sera libéré ...
  secret+="Berce mon coeur d'";                      // tout seul
  std::cout << "agent 1 a eu le RDV et a passé le code \n";
}

void agent2(int num){
  for (int j=0; j<num; j++) {
    std::cout.put('+').flush();
    std::this_thread::sleep_for(std::chrono::milliseconds(num));
  }
  demander_RDV();
  std::lock_guard<std::mutex>lock(protege_code);
  secret+="une longueur monotone !";
```

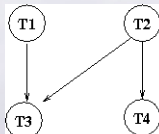
Sémaphores pour la synchronisation (suite)

```
   std::cout << "agent 2 a eu le RDV et a appris le code = " << secret << '\n';
}

void demander_RDV() {
   if (First) {First=false; synchro_RDV.P(); }
   else       {First=true;synchro_RDV.V(); }
}

int main(){
     std::thread t1( agent1, 20 );
     std::thread t2( agent2, 10 );
     t1.join(); t2.join();

     std::cout << "Fin des threads : les agents se planquent le code secret était : " << secret  <<
       std::endl;
     return 0;
}
// *++*++*++*++*++**************agent 1 a eu le RDV et a passé le code
// agent 2 a eu le RDV et a appris le code = Berce mon coeur d'une longueur monotone !
// Fin des threads : les agents se planquent le code secret était : Berce mon coeur d'une longueur
   monotone !
```

- Nous reviendrons sur cet exemple un peu plus loin.
- Dans la suite, on montre un cas de synchronisation simple à l'aide des sémaphores avant de voir un exemple concret.

Sémaphores pour la synchronisation (suite)

- Soit le graphe de précédences suivant à réaliser avec les sémaphores :



- Une solution qui minimise le nombre de sémaphores :

```
S : Sémaphore init 0;
parbegin   // début lancement parallèle des tâches (par des threads)
        begin T1; P(S); T3; end;
        begin T2; V(S); T4; end;
parend
```

Détails Sémaphores

- A considérer comme un distributeur de tickets dans un parking avec un nombre limité de places.
  - ➥ Cas d'une place (une cabine téléphonique)
  - ➥ Cas de N places (un parking)

```
struct Semaphore {
    valeur : entier;
    attente : file d attente;
  }

Fonction V(Sem : Semaphore) {}
     Si Sem.val < 0 Alors Débloquer_suivant(Sem.attente);
     Finsi
     Sem.val ++;
    }

Fonction P(Sem : Semaphore) {}
     Sem.val --;
     Si Sem.val < 0       Alors Attendre(Sem.attente);
     Finsi
    }
```

Détails Sémaphores (suite)

- Les environnements / compilateurs proposent les sémaphores sous différentes formes.
  - ➙ Via différentes fonctions d'utilisation (verrouillage, libération, attente, deadlock, ...).
- L'exemple suivant montre un cas de synchronisation simple.

Partage d'écran via un mutex

- Un exemple de partage de l'écran :
  - ➥ Si chacun écrit, les messages sont mélangés.
- Ci-dessous, deux threads utilisent un mutex pour l'accès à l'écran, et deux autres n'en utilisent pas

```
// Ex7—bis.cpp
#include <future>
#include <mutex>
#include <iostream>
#include <string>

using std::mutex;

mutex print_Mutex;  // enable synchronized output with print_char_by_char()

void print_char_by_char (const std::string& s) {
  for (int i=0; i< 5; i++) {
    for (char c : s) {
      print_Mutex.lock();
      std::cout.put(c);
      print_Mutex.unlock();
    }
    print_Mutex.lock();
    std::cout << std::endl;
    print_Mutex.unlock();
```

Partage d'écran via un mutex (suite)

```
    }
}

void print_with_no_mutex (const std::string& s) {
    for (int i=0; i< 5; i++) {
        for (char c : s) {
            std::cout.put(c);
        }
        std::cout << std::endl;
    }
}

int main() {
    // Ces 2 threads se partagent l'écran avec un mutex
    auto f1 = std::thread(print_char_by_char, "Hello1 from a first thread");
    auto f2 = std::thread(print_char_by_char, "Hello2 from a second thread");

    // un appel sans thread
    print_char_by_char("Hello 1 from the main thread");

    // Ces 2 threads se partagent l'écran sans  mutex
    std::thread f3(print_with_no_mutex, "Hello3 from a 3rd thread");
    std::thread f4(print_with_no_mutex, "Hello4 from a 4th thread");

    f1.join();f2.join();f3.join();f4.join();
}
```

- Traces :

Partage d'écran via un mutex (suite)

```
Hello 1 from the main thread
Hello 1 from the main thread
Hello 1 from the main thread
Hello 1 from the main thread
Hello 1 from the main thread
Hello2 from a secondHello1 from  ath freadir                    <<<——— !!!
Hello2 from a second tst thread
Hello1 from a first thread
Hello1 from a first thread
Hello1 from a first thread
Hello1 from a first thread
hread                                                           <<<——— !!!
Hello2 from a second thread
Hello2 from a second thread
Hello2 from a second thread
Hello3 from a 3rd thread
Hello3 from a 3rd thread
Hello3 from a H3ello 2 frormd  tthhreaed main thread            <<<——— !!!
Hello 2 from the main thread
Hello 2 from the main thread
Hello 2 from the main thread
Hello 2 from the main thread

Hello3 from a 3rd thread Hello4 from
Hello3 from a 3rd t har e4athd th
read                                                            <<<——— !!!
Hello4 from a 4th thread
Hello4 from a 4th thread
Hello4 from a 4th thread
Hello4 from a 4th thread
```

Partage d'écran via un mutex (suite)

Le même exemple avec plus de fluidité :

```
// Ex7-ter.cpp

#include <future>
#include <mutex>
#include <iostream>
#include <string>
#include <thread>

using std::mutex;

mutex print_Mutex;  // enable synchronized output with print_char_by_char_with_mutex()

void print_char_by_char_with_mutex (const std::string& s) {
  for (int i=0; i< 5; i++) {
    print_Mutex.lock();

    for (char c : s)  std::cout.put(c);
    std::cout << std::endl;

    print_Mutex.unlock();    // pour laisser passer les autres

    // Eventuellement, temporiser ou mieux : yield()
    std::this_thread::yield();  // passer la main !
  }
}

void print_char_by_char_with_no_mutex (const std::string& s) {
  for (int i=0; i< 5; i++) {
```

Partage d'écran via un mutex (suite)

```
      for (char c : s) {
        std::cout.put(c);
      }
      std::cout << std::endl;
    }
}

int main() {
  auto f1 = std::thread(print_char_by_char_with_mutex, "Hello1 from a first thread");
  auto f2 = std::thread(print_char_by_char_with_mutex, "Hello2 from a second thread");
  print_char_by_char_with_mutex("Hello 0 from the main thread");

  std::thread f3(print_char_by_char_with_no_mutex, "Hello3 from a 3rd thread");
  std::thread f4(print_char_by_char_with_no_mutex, "Hello4 from a 4th thread");

  f1.join(); f2.join(); f3.join(); f4.join();

}

/* ———————————— TRACES ————————————
Hello 0 from the main thread
Hello1 from a first thread
Hello 0 from the main thread
Hello 0 from the main thread
Hello 0 from the main thread
Hello 0 from the main thread
Hello1 from a first thread
Hello2 from a second thread
Hello1 from a first thread
Hello2 from a second thread
Hello2 from a second thread
Hello1 from a first thread
```

Partage d'écran via un mutex (suite)

```
Hello2 from a second thread
Hello1 from a first thread
HHello2 eflroml Ho4 frelolo3m  far 4tomh a  3trhrad  tseead
Hello4 from a 4th thread
Hello4 from a 4th thread
Hello4 from a 4th thread
Hello4 from a 4th thread
hread
Hello3 from a 3rd threacd
Hello3 from a 3rd threado
Hello3 from a 3rd threandd
Hello3 from a 3rd thread
 thread
*/
```

Un autre Exemple de mutex

- Dans cet exemple, on montre :
  - L'utilisation d'un mutex dans l'incrémentation concurrente d'une variable globale,
  - Passage de paramètres à un thread

```
// Ex8.cpp
#include <thread>
#include <iostream>
#include <mutex>

int var_globale1=0, var_globale2=0;

std::mutex mtex;    // pour l'exclusion mutuelle

void incrementer_sans_exclusion_mutuelle( int val ){
    std::cout << "thread incrementation sans mutex lance : val = " << val << std::endl;
    for (int i=0; i< val; i++) var_globale1++;
    return;
}

void incrementer_avec_eclusion_mutuelle( int val ){
    std::cout << "thread incrementation avec mutex lance : val = " << val << std::endl;
    for (int i=0; i< val; i++) {
      mtex.lock();
      var_globale2++;
```

Un autre Exemple de mutex (suite)

```
        mtex.unlock();
      }
      return;
}

int main( int argc, char *argv[]){
        // création des threads
      std::thread t1( incrementer_sans_exclusion_mutuelle, 30000 );
      std::thread t2( incrementer_sans_exclusion_mutuelle, 30000 );
      std::thread t3( incrementer_avec_eclusion_mutuelle, 30000 );
      std::thread t4( incrementer_avec_eclusion_mutuelle, 30000 );
      t1.join();       // attendre la fin de t1
      t2.join();t3.join(); t4.join();

      std::cout << "Fin des threads : la variable globale 1 vaut  " << var_globale1  << std::endl;
      std::cout << "\t \t et la variable globale 2 vaut  " << var_globale2  << std::endl;

      return 0;
}
//---------------------------------------------------------------
//g++ Ex8.cpp -Wall -std=c++0x  -lpthread

// thread incrementation sans mutex lance : val = 30000
// thread incrementation avec mutex lance : val = 30000
// thread incrementation sans mutex lance : val = 30000
// thread incrementation avec mutex lance : val = 30000
// Fin des threads : la variable globale 1 vaut  44483
//                   et la variable globale 2 vaut  60000

// .... et d'autres traces d'exécutions différentes
```

Abandon du temps d'exécution

- Un thread se met en attente via la fonction **yield** ou par un **sleep**
- La fonction *yield()* permet à un thread d'abandonner son exécution au profit d'un autre thread (choisi par l'ordonnanceur / arbitre).

  Elle est très utile lorsqu'un thread boucle sur une activité et bloque constamment un verrou pour réaliser une action (cf. Synchronisation et Section Critique ci dessous).

- N.B. : On peut aussi "passer la main" pendant un délai écoulé (même un délai null) de différentes manières :
  - ➙ **sleep(n)** attendre n secondes (1,2,...)
  - ➙ **usleep(1..1000000)** micro secondes (usleep(1000000)=sleep(1))
- Ces deux cas font appel à la bibliothèque standard de C/C++. ../..
  - ➙ Il est préférable d'utiliser les (nouveaux) mécanismes de C++.

Abandon du temps d'exécution (suite)

En C++11 et C++14 :

- On peut travailler au niveau des secondes / milliseconds / nanoseconds par la bibliothèque "std::" :

  ➙ **this_thread::sleep_for(chrono::milliseconds(durée));**

  Ou **chrono::seconds(0)**

  ➙ Ou encore en nanosecondes comme dans cet exemple :

```
using namespace std::chrono;        /

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20});
auto t1 = high_resolution_clock::now();

cout << duration_cast<nanoseconds>(t1−t0).count() << " nanoseconds passed\n";
```

☞ **En général, on préfère une attente** (même de 0 secondes) à un *yield*. Néanmoins, "yield" ne dépend pas d'un délai (voir exemple ci-dessous).

Abandon du temps d'exécution (suite)

Un schéma d'utilisation de *yield*

➙ Le mécanisme *mutex* a été présenté plus haut.

```
int tab[10];
mutex verrou;      // pour véroullier l'accès à tab

void fonction(....){
    for(;;){
        verrouiller(verrou);
          ICI TRAVAILLER SUR TAB
        deverrouiller(verrou);

        /* sans le yield, le thread risque de reprendre tout de suite
        le  verrou sans laisser le temps aux autres threads de travailler sur le tableau */

        yield();
    }
     ...
}
```

- Exemple d'attente et de *yield()* :

  on reprend l'exemple précédent et on y ajoute un abandon provisoire au profit des autres threads par **yield()** :

Abandon du temps d'exécution (suite)

```cpp
// Ex9.cpp
#include <thread>
#include <iostream>
#include <mutex>

 int var_globale1=0;
 int var_globale2=0;

 std::mutex _mutex;

 void incrementer_sans_mutex( int val ){
     std::cout << "thread incrementation sans mutex lance : val = " << val << std::endl;
     for (int i=0; i< val; i++) {
       var_globale1++;
       std::this_thread::yield(); // hint to reschedule to the next thread
     }
     return;
 }

 void incrementer_avec_mutex( int val ){
     std::cout << "thread incrementation avec mutex lance : val = " << val << std::endl;
     for (int i=0; i< val; i++) {
       _mutex.lock();
       var_globale2++;
       _mutex.unlock();
       std::this_thread::yield(); // hint to reschedule to the next thread
     }
     return;
 }
```

Abandon du temps d'exécution (suite)

```
int main( int argc, char *argv[] ){
    std::thread t1( incrementer_sans_mutex, 30000 );
    std::thread t2( incrementer_sans_mutex, 30000 );
    std::thread t3( incrementer_avec_mutex, 30000 );
    std::thread t4( incrementer_avec_mutex, 30000 );
    t1.join();
    t2.join();
    t3.join();
    t4.join();

    std::cout << "Fin des threads : la variable globale 1 vaut  " << var_globale1  << std::endl;
    std::cout << "\t \t et la variable globale 2 vaut  " << var_globale2  << std::endl;
    return 0;
}

/*
 thread incrementation sans mutex lance : val = thread incrementation avec mutex lance : val =
   3000030000
thread incrementation sans mutex lance : val = 30000

thread incrementation avec mutex lance : val = 30000
Fin des threads : la variable globale 1 vaut  34277
                 et la variable globale 2 vaut  60000

D'autres traces avec d'autres exécutions...
*/
```

Abandon du temps d'exécution (suite)

Attente aléatoire :

- Comme indiqué ci-dessus, on peut laisser la main aux autres threads en demandant un délai d'attente par *sleep_for*.
- L'exemple ci-dessus montre comment utiliser un délai tiré au hasard (entre 10 et 100):

```
int print_a_char (char c) {
// random—number generator (use c as seed to get different sequences)
  std::default_random_engine dre (c);
  std::uniform_int_distribution < int >id (10, 1000);

// loop to print character after a random period of time
  for (int i = 0; i < 10; ++i) {
      std::this_thread::sleep_for(std::chrono::milliseconds (id(dre)));
      std::cout.put (c).flush ();
    }
  return c;
}
```

Variable atomic en C++

- En C++11, il est possible de grandement accélérer la manipulation des variables protégées par un mutex en utilisant (lorsque cela est possible) une variable **atomic** à la place de la variable protégée.
  - ➙ L'accès à une variable atomic (en lecture-écriture) est réalisé par des mécanismes de bas niveau, ce qui accélère les manipulations.
- Une variable atomic peut être d'un type de base : int, char, long et pointeur.
- Quelques fonctions d'accès à une variable atomic :
  - ➙ création et initialisation : *atomic<int> x(val);*
  - ➙ incrément/décrément par ++/--
  - ➙ accès direct ou par load: *x.load();*
  - ➙ échange de valeur (via exchange), etc.
- Consulter la documentation pour les autres fonctions d'accès à une variable atomic.

Variable atomic en C++ (suite)

- Dans cet exemple, la variable globale incrémentée sera déclarée atomic.

```
// Ex9.cpp

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

std::atomic<int> var_globale(0);     // initialisation

void incrementer_atomic( int val ){
    std::cout << "thread incrementation atomic lance : val = " << val << std::endl;
    for (int i=0; i< val; i++) var_globale++;
    return;
}


int main( int argc, char *argv[] ){
    std::thread t1( incrementer_atomic, 30000 );
    std::thread t2( incrementer_atomic, 30000 );
    t1.join();
    t2.join();

    std::cout << "\t \t et la variable globale 3 vaut  " << var_globale  << std::endl;  //ou
      var_globale.load() aussi.
    return 0;
}

// Les résultats sont les même qu'avec un mutex.
```

TAS et Section Critique

- Une section critique (SC) est une partie de code qui doit être exécutée par un processus / threaed à l'exclusion de tous les autres.

- Dans un système d'exploitation, un **TAS** permet de résoudre le problème de **Section Critique** (SC) de la manière suivante pour n processus P_i :

  - ➙ Chaque processus P_i possède une variable locale $Test_i$.
  - ➙ Les processus partagent une variable globale *Verrou*.
    - ➥ Deux opérandes : un registre R et un mot mémoire B .
    - ➥ On copie B dans R et on place 1 dans B.

```
Procédure TAS (var a,b : entier)          TAS=Test And Set
  Début
            a := b;  b:=1;
  Fin TAS;
```

TAS et Section Critique (suite)

- Dans le pseudo-code précédent :
  - Le 1er processus qui exécute *TAS* trouve $Verrou = 0$ et entre en SC.
  - Les autres trouveront $Verrou = 1$ et attendront.
- Schéma du code de chaque processus P_i :

```
Testᵢ : entier;
Répéter    <SR>
          Testᵢ = 1;
          Tantque Testᵢ=1 faire
                    TAS(testᵢ, Verrou);
          finTq
          < SC >
          Verrou := 0;
Jusqu'à Faux;
```

TAS et Section Critique (suite)

- Ce schéma garantie *l'exclusion mutuelle* mais souffre d'une *attente bornée*.
  - ➙ On peut rendre cette attente plus productive (par d'autres moyens).
- Ce qui caractérise un TAS : il est géré par des instruction insécables (souvent au niveau hardware).
- Dans les problèmes d'exclusion mutuelle, la solution avec TAS est plus rapide (si un seul ticket suffit).
- Alternativement, on peut réaliser ces exclusions avec d'autres mécanismes (moins rapides).

Test and Set en C++

- A l'image de *atomic*, la bibliothèque C++11 propose le type *std::atomic_flag* et des opérations qui sont garanties d'être exécutées de manière insécable.
  - ➙ De ce fait, on a moins besoin d'organiser soi-même les accès via un mutex.

- En utilisant ces fonctions, il est possible de demander un accès en exclusion mutuel comme pour un TAS : c'est le système qui se charge de la réalisation et des protections.

- Dans l'exemple suivant, 5 threads (stockés dans un tableau) utilisent le mécanisme de TAS pour accéder et écrire à l'écran.
  - ➙ Les écritures ont lieu de manière concurrente pour chaque tâche :

Test and Set en C++ (suite)

```
// Ex10.cpp

#include <thread>
#include <vector>
#include <iostream>
#include <atomic>

std::atomic_flag lock = ATOMIC_FLAG_INIT;

void f(int n){
   for (int cnt = 0; cnt < 10; ++cnt) {
     // wait for access and acquire lock
     while (lock.test_and_set(std::memory_order_acquire));   // tourner en rond !
     std::cout << "GOT ACCESS : Output from thread " << n << " : " << cnt << '\n';
     lock.clear(std::memory_order_release);            // release lock
   }
}

int main(){
     std::vector<std::thread> v;
     for (int n = 0; n < 5; ++n) {
         v.emplace_back(f, n);   // store a thread for "f(n)"
     }
     for (auto& t : v) {
         t.join();
     }
}


// TRACE : ......
```

Test and Set en C++ (suite)

```
// Output from thread 0 : 0
// Output from thread 1 : 0
// Output from thread 3 : 0
// Output from thread 0 : 1
// Output from thread 1 : 1
// Output from thread 2 : 0
// Output from thread 1 : 2
// Output from thread 0 : 2
// Output from thread 2 : 1
// Output from thread 0 : 3
// Output from thread 4 : 0
// Output from thread 0 : 4
// Output from thread 4 : 1
// Output from thread 0 : 5
// Output from thread 4 : 2
// ...
```

- Lattente active ci-dessus peut être transformée en passive (wait).
- Ainsi, on peut soi-même réaliser une variable protégée similaire à *atomic* protégée par *test_and_set* de C++11.

Manipulation simplifiée des mutex

- Nous avons vu l'utilisation classique des mutex (et les mécanismes similaires comme *atomic* et *TAS*).

- C++11 propose d'autres fonctions qui facilitent la manipulation des mutex.

- Pour une variable d'exclusion mutuelle *mutex verrou;*

  On peut utiliser *lock_guard<std::mutex> lock(verrou);*

  à la place de *verrou.lock();*.

➙ L'intérêt est qu'on n'a pas besoin de libérer le *verrou* par unlock() :

➙ La libération aura lieu par C++, même en cas d'exception / erreur.

Manipulation simplifiée des mutex (suite)

- Un exemple
  - → *lock_guard<std::mutex> lock(maMutex)* n'a pas besoin de *unlock*.

N.B. : il en est de même pour les fichiers qu'on ouvre mais qui se ferment tous seuls (voir *file* de l'exemple!

```
#include <string>
#include <mutex>
#include <iostream>
#include <fstream>
#include <stdexcept>

void write_to_file (const std::string & message) {
    // mutex to protect file access
    static std::mutex maMutex;

    // lock mutex before accessing file
    std::lock_guard<std::mutex> lock(maMutex);

    // try to open file
    std::ofstream file("example.txt");
    if (!file.is_open())
        throw std::runtime_error("unable to open file");

    // write message to file
    file << message << std::endl;
```

Manipulation simplifiée des mutex (suite)

```
    // file will be closed 1st when leaving scope (regardless of exception)

    // maMutex will be unlocked 2nd (from lock destructor)
    // when leaving scope  (regardless of exception)
}
```

➙ On remarque l'intérêt de

std::lock_guard<std::mutex> lock()

qui nous dispense de penser à *unlock* (une des causes de dead-lock).

Mutex récursif

- Un autre mécanisme de simplification de code.
- Par défaut, un thread ne peut pas bloquer un mutex plus d'une fois sous peine de s'enfermer dans un dead-lock et tout bloquer.

- Il arrive cependant qu'un thread ait besoin (de tenter) de verrouiller un mutex plusieurs fois.

  → Pour éviter un dead-lock dans ce cas, on peut utiliser un **mutex récursif** (sinon, trop de précautions doivent être prise).

- Dans l'exemple suivant, l'opération *both* peut tenter le verrouillage plusieurs fois un mutex sans risque de dead-lock.
- On constate que les fonctions *mul* et *div* bloquent le *mutex* car ces fonctions peuvent être appelées normalement (sans passer par *both*).
- Aussi, la fonction *both* procède et bloque le même mutex.
- Avec un *mutex* ordinaire, un appel à *both()* bloquerait tout.

Mutex récursif (suite)

```
struct Complex {
    std::recursive_mutex mutex;
    int i;

    Complex() : i(0) {}

    void mul(int x){
        std::lock_guard<std::recursive_mutex> lock(mutex);
        i *= x;
    }

    void div(int x){
        std::lock_guard<std::recursive_mutex> lock(mutex);
        i /= x;
    }

    void both(int x, int y){
        std::lock_guard<std::recursive_mutex> lock(mutex);
        mul(x);
        div(y);
    }
};
```

Mutex et lock unique

- La classe **unique_lock** est une généralisation d'accès aux mutex.
- Elle permet de procéder à un verrouillage différé / limité dans le temps/ récursive, à un transfert de verrou ...
- Un **unique_lock** équivaut à une réservation unique d'un verrou.
  - ➞ Un lock sans deadlock suivra **unique_lock**.
  - ➞ Elle peut être utilisée avec les variables de condition (voir plus loin).

```
#include <mutex>
#include <thread>
#include <chrono>

struct Box {
    explicit Box(int num) : num_things{num} {}
    int num_things;
    std::mutex m;
};

void transfer(Box &from, Box &to, int num){
    // don't actually take the locks yet
    std::unique_lock<std::mutex> lock1(from.m, std::defer_lock);
    std::unique_lock<std::mutex> lock2(to.m, std::defer_lock);

    // lock both unique_locks without deadlock
```

Mutex et lock unique (suite)

```
    std::lock(lock1, lock2);

    from.num_things -= num;
    to.num_things += num;

    // 'from.m' and 'to.m' mutexes unlocked in 'unique_lock' dtors
}

int main(){
    Box acc1(100);
    Box acc2(50);

    std::thread t1(transfer, std::ref(acc1), std::ref(acc2), 10);
    std::thread t2(transfer, std::ref(acc2), std::ref(acc1), 5);

    t1.join();
    t2.join();
}
```

- Par ailleurs, la fonction *try_lock* et *try_lock_for* permettent de **tenter** un verrouillage sans être bloqué (on essaie !).

Variable de condition

- Une variable de condition est un moyen de synchronisation utilisé pour bloquer un ou plusieurs threads jusqu'à :
  - l'arrivée d'une notification (envoyée par un autre thread)
  - l'arrivée d'un time-out
  - un réveille douteux (un réveil qui ne devait pas arriver) !
- Un exemple (syntaxe Posix modifée en C++11) de ce dernier cas :

```
// In a waiting thread
while(!buf->full)
    wait(&buf->cond, &buf->lock);

// In any other thread:
if(buf->n >= buf->size){
    buf->full = 1;
    signal(&buf->cond);
}
```

Variable de condition (suite)

- Dans cet exemple, un autre thread place *buf->full=1* (la condition attendue) AVANT de signaler évènement *buf->cond*.
  - ➙ Le thread qui attend vérifie ce booléen pour éviter un réveil douteux.
- En C++11 :
  - ➙ Une variable de condition (*une std::condition_variable*) est utilisée typiquement avec un *mutex* pour réaliser les échanges entre threads.
  - ➙ Tout thread qui attend une condition (sur *une std::condition_variable*) doit d'abord récupérer un mutex par **std::unique_lock**.
  - ➙ L'opération **wait** libère automatiquement le mutex avant de suspendre le thread (et le mettre en attente).
  - ➙ Lorsque la condition est signalée (par *notify*), le thread est réveillé et récupère le mutex.

Variable de condition (suite)

- Exemple :

```
// Ex11.cpp

#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <iostream>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread(){
    // Wait until main() sends data
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{return ready;}); // lambda calcul predicat
    // the predicat returns  false  if the waiting should be continued

    //after the wait, we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // Send data back to main()
    processed = true;
    std::cout << "Worker thread signals data processing completed\n";
```

Variable de condition (suite)

```
    // Manual unlocking is done before notifying, to avoid
    // that the waiting thread gets blocked again.
    lk.unlock();
    cv.notify_one();
}

int main(){
    std::thread worker(worker_thread);

    data = "Example data";
    // send data to the worker thread
    {   std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {   std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, []{ return processed;});
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
}

// Output:
// main() signals data ready for processing
// Worker thread is processing data
// Worker thread signals data processing completed
// Back in main(), data = Example data after processing
```

Variable de condition (suite)

- Un autre exemple :

```cpp
// Ex12.cpp

#include <iostream>
#include <condition_variable>
#include <thread>
#include <chrono>

std::condition_variable cv;
std::mutex cv_m; // This mutex is used for three purposes:
                 // 1) to synchronize accesses to i
                 // 2) to synchronize accesses to std::cerr
                 // 3) for the condition variable cv
int i = 0;

void waits(){
    std::unique_lock<std::mutex> lk(cv_m);
    std::cerr << "Waiting... \n";
    cv.wait(lk, []{return i == 1;}); //  wait again if i != 1
    std::cerr << "...finished waiting. i == 1\n";
}

void signals(){
    std::this_thread::sleep_for(std::chrono::seconds(1));
    {
        std::lock_guard<std::mutex> lk(cv_m);
        std::cerr << "Notifying...\n";
    }
    cv.notify_all();
```

Variable de condition (suite)

```
    std::this_thread::sleep_for(std::chrono::seconds(1));

    {
        std::lock_guard<std::mutex> lk(cv_m);
        i = 1;
        std::cerr << "Notifying again...\n";
    }
    cv.notify_all();
}

int main(){
    std::thread t1(waits), t2(waits), t3(waits), t4(signals);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
}

Possible output:
Waiting...
Waiting...
Waiting...
Notifying...
Notifying again...
...finished waiting. i == 1
...finished waiting. i == 1
```

Variable de condition (suite)

- Exemple *wait_until* :

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

std::condition_variable cv;
std::mutex cv_m;
std::atomic<int> i = ATOMIC_VAR_INIT(0);

void waits(int idx){
    std::unique_lock<std::mutex> lk(cv_m);
    auto now = std::chrono::system_clock::now();
    if(cv.wait_until(lk, now + std::chrono::milliseconds(idx*100), [](){return i == 1;}))
        std::cerr << "Thread " << idx << " finished waiting. i == " << i << '\n';
    else
        std::cerr << "Thread " << idx << " timed out. i == " << i << '\n';
}

void signals(){
    std::this_thread::sleep_for(std::chrono::milliseconds(120));
    std::cerr << "Notifying...\n";
    cv.notify_all();
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    i = 1;
    std::cerr << "Notifying again...\n";
    cv.notify_all();
}
```

Variable de condition (suite)

```
int main(){
    std::thread t1(waits, 1), t2(waits, 2), t3(waits, 3), t4(signals);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
}


Possible output:
Thread 1 timed out. i == 0
Notifying...
Thread 2 timed out. i == 0
Notifying again...
Thread 3 finished waiting. i == 1
```

Variable de condition (suite)

- Exemple *wait_for* :

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

std::condition_variable cv;
std::mutex cv_m;
std::atomic<int> i{0};

void waits(int idx){
    std::unique_lock<std::mutex> lk(cv_m);
    if(cv.wait_for(lk, std::chrono::milliseconds(idx*100), [](){return i == 1;}))
        std::cerr << "Thread " << idx << " finished waiting. i == " << i << '\n';
    else
        std::cerr << "Thread " << idx << " timed out. i == " << i << '\n';
}

void signals(){
    std::this_thread::sleep_for(std::chrono::milliseconds(120));
    std::cerr << "Notifying...\n";
    cv.notify_all();
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    i = 1;
    std::cerr << "Notifying again...\n";
    cv.notify_all();
}
```

Variable de condition (suite)

```
int main(){
    std::thread t1(waits, 1), t2(waits, 2), t3(waits, 3), t4(signals);
    t1.join(); t2.join(), t3.join(), t4.join();
}

Output:
Thread 1 timed out. i == 0
Notifying...
Thread 2 timed out. i == 0
Notifying again...
Thread 3 finished waiting. i == 1
```

Variable de condition (suite)

- Exemple *notify_all* :

```
#include <iostream>
#include <condition_variable>
#include <thread>
#include <chrono>

std::condition_variable cv;
std::mutex cv_m; // This mutex is used for three purposes:
                 // 1) to synchronize accesses to i
                 // 2) to synchronize accesses to std::cerr
                 // 3) for the condition variable cv
int i = 0;

void waits(){
    std::unique_lock<std::mutex> lk(cv_m);
    std::cerr << "Waiting... \n";
    cv.wait(lk, []{return i == 1;});
    std::cerr << "...finished waiting. i == 1\n";
}

void signals(){
    std::this_thread::sleep_for(std::chrono::seconds(1));
    {
        std::lock_guard<std::mutex> lk(cv_m);
        std::cerr << "Notifying...\n";
    }
    cv.notify_all();

    std::this_thread::sleep_for(std::chrono::seconds(1));
```

Variable de condition (suite)

```
    {
        std::lock_guard<std::mutex> lk(cv_m);
        i = 1;
        std::cerr << "Notifying again...\n";
    }
    cv.notify_all();
}

int main(){
    std::thread t1(waits), t2(waits), t3(waits), t4(signals);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
}



Possible output:
Waiting...
Waiting...
Waiting...
Notifying...
Notifying again...
...finished waiting. i == 1
...finished waiting. i == 1
...finished waiting. i == 1
```

Variable de condition (suite)

- Exemple *notify_all_at_thread_exit* :
  - *notify_all_at_thread_exit* permet de notifier aux autres threads qu'un thread a terminé.
  - Cela entraine la destruction des objets locaux du thread.
- Le mode d'opération est le suivant :
  - La propriété des verrous possédés par le thread courant est passée au système,
  - Le thread courant termine après la destruction de ses objets locaux,
  - Les opérations suivantes seront exécutées (pour la condition *cond* et le verrou *verrou*) :

    ```
    verrou.unlock();
    cond.notify_all();
    ```
- On peut réaliser un effet similaire par *std::promise* ou *std::packaged_task*.

Variable de condition (suite)

```
#include <mutex>
#include <thread>
#include <condtion_variable>

std::mutex m;
std::condition_variable cv;

bool ready = false;
ComplexType result;  // some arbitrary type

void thread_func(){
    std::unique_lock<std::mutex> lk(m);
    // assign a value to result using thread_local data
    result = function_that_uses_thread_locals();
    ready = true;
    std::notify_all_at_thread_exit(cv, std::move(lk));
} // 1. destroy thread_locals, 2. unlock mutex, 3. notify cv

int main(){
    std::thread t(thread_func);
    t.detach();

    // do other work ...

    // wait for the detached thread
    std::unique_lock<std::mutex> lk(m);
    while(!ready) {cv.wait(lk);}
    process(result); // result is ready and thread_local destructors have finished
}
```

Exercices

- Calcul parallèle de PI (via un cercle unitaire)
- Merge-sort multi-threads
- Bonus : proposer une version multi-threads de la méthode newton et le coder en C++11
- Une 2nde séquence d'exercices est proposée par la suite.

Exercices (suite)

A propos du calcul de PI par un cercle unitaire on peut calculer une valeur approché de PI à l'aide d'un cercle unitaire.

Principe : on échantillonne un point (couple de réels $< x, y > \in [0.0, 1.0]$) et on examine la valeur de $x^2 + y^2 \leq 1$

- ➙ Si "vrai", le point est dans le quart du cercle unitaire (on a un *hit*)
- ➙ Sinon, ...

- Après N (grand) itérations, le nombre de *hit* approxime $\frac{1}{4}$ de la surface du cercle unitaire, d'où la valeur de π (précision $= \frac{1}{\sqrt{N}}$)
- Programmer cette méthode avec des threads :
  - ➙ plusieurs threads se partagent le quart de cercle
  - ➙ plusieurs threads calculent le quart de cercle et on prend la moyenne
  - ➙ plusieurs threads : un thread par quart de cercle.

Exercices (suite)

A propos de Merge Sort

Le principe : pour trier un tableaux T de N éléments,

- Scinder T en deux sous-tableaux $T1$ et $T2$
- Trier $T1$ et $T2$
- Reconstituer T en y plaçant $T1$ puis $T2$

➙ $T1$ et $T2$ sont chacun trié et leur fusion tient compte de cela.

☞ Pour trier chacun des sous-tableaux, procéder de la même manière

☞ A priori, on peut créer autant de threads que de sous tableaux mais une gestion plus modérée des threads (pour ne pas en créer beaucoup) est recommandée.

Exercices (suite)

A propos de Newton

Purpose

Multi-threaded determination of the argument values x, in the interval $[a, b]$ (where $a < b$), such that $f(x) = 0$.

Method

For $i = 0, \ldots, n$, we define the i-th grid point g_i by

$$g_i = a\frac{n-i}{n} + b\frac{i}{n}$$

For $i = 0, \ldots, n-1$, we define the i-th sub-interval of $[a, b]$ by

$$I_i = [g_i, g_{i+1}]$$

Newton's method is applied starting at the center of each of the sub-intervals I_i for $i = 0, \ldots, n-1$ and at most one zero is found for each sub-interval.

➢ J'ai le pdf dans ce rép. Le site = https://www.coin-or.org/CppAD/Doc/multi_newton.cpp.htm

TabMat

1 Avant propos

2 Introduction aux threads C++11
- Threads : Un premier exemple
- Exemple 2
- Paramètres d'un thread
- Valeur calculée par un thread
- Récupération d'une valeur
- Divers Lacements

3 Mécanismes de Synchronisation
- Besoin de synchronisation : Un exemple trivial
- Gestion d'accès en exclusion mutuelle
- Par Test and Set (TAS)
- Sémaphores
- Détails Sémaphores
- Partage d'écran via un mutex
- Un autre Exemple de mutex
- Abandon du temps d'exécution
- Variable atomic en C++
- TAS et section critique
- Test and Set en C++
- Manipulation simplifiée des mutex
- Mutex récursif
- Mutex et lock unique
- Variable conditionnelles

4 Exercices

5 Table des matières