Chapitre 12

Les threads

1. Processus

- Chaque application (emacs, Word, etc.) exécutée sur un ordinateur lui est associée un processus représentant l'activité de celle-ci.

- À ce processus est associé un ensemble de ressources personnalisées comme l'espace mémoire, le temps CPU etc.

- Ces ressources seront dédiées à l'exécution des instructions du programme associé à l'application.

- Les processus coûtent cher au lancement (l'espace mémoire à calculer, les variables locales au processus à définir, etc.).

2. Multitâche

- C'est la possibilité que le système d'exploitation a pour faire fonctionner plusieurs programmes en même temps et cela, sans conflit.

- Par exemple: écouter de la musique et lire son courrier en même temps.

- Un système monoprocesseur (et non pas monoprocessus, ne pas confondre processus et processeur), simule le multitâche en attribuant un temps (un quantum ou temps déterminé à l'avance) de traitement pour chaque processus (description très simpliste du rôle du monoprocesseur).

- Un système multiprocesseur peut exécuter les différents processus sur les différents processeurs du système.

- Généralement, un programme est livré avec un tel système pour permettre de distribuer de manière efficace les processus sur les processeurs de la machine.

- Un programme est dit multitâche s'il est capable de lancer plusieurs parties de son code en même temps.

- À chaque partie du code sera associé un processus (sous-processus) pour permettre l'exécution en parallèle.

- La possibilité de lire l'aide en ligne et d'imprimer à la fois sous Word. Ce sont deux sous-processus relevant d'un processus qui est associé à l'application Word.

- Un serveur réseau (processus) et le traitement de requêtes (sous-processus associé à chaque requête).

- Nous aimerions que le traitement de ces requêtes soit fait de manière optimale de telle sorte à pouvoir servir toutes les requêtes dans un espace de temps raisonnable.

- Comme chaque processus coûte cher au lancement, sans oublier qu'il va avoir son propre espace mémoire, la commutation (passage d'un processus à un autre) ainsi que la communication interprocessus (échange d'informations) seront lourdes à gérer.

Question

- Peut-on avoir un sous-processus qui permettrait de lancer une partie du code d'une application sans qu'il soit onéreux?

- La réponse est venue avec les threads qui sont appelés aussi « processus légers ».

3. Threads

- Un thread est un sous-ensemble d'un processus, partageant son espace mémoire et ses variables.

- De ce fait, les coûts associés suite à son lancement sont réduits, donc il est plus rapide.

- En plus de cela, à chaque thread est associé des unités propres à lui: comme sa pile (pour gérer les instructions à exécuter par le thread), le masque de signaux (les signaux que le thread doit répondre), sa priorité d'exécution (dans la file d'attente), des données privées, etc.

- Il faut ajouter à cela, les threads peuvent être exécutés en parallèle par un système multitâche.

- Cependant, le partage de la mémoire et les variables du processus entraînent un certain nombre de problèmes lorsqu'il y a un accès partagé à une ressource, par exemple.

- Deux agents (threads) voulant effectuer des réservations de places d'avion (ressource: nombre de places disponibles dans un avion).

- On protège l'accès à cette ressource dès qu'un thread est en train de réaliser une écriture (réaliser une réservation) sur cette ressource.

4. C++ et les threads

- Les anciens standards, « C++98 » et « C++03 », du langage C++ ne disposait pas d'une librairie native pour manipuler les threads.

- Plusieurs paquetages ont été rendus disponibles pour combler cette carence.

- Parmi ces paquetages, nous pouvons citer « POSIX threads », « Mach C-threads », « Solaris 2 UI-threads ». Et tout récemment, « wxThread », « Boost thread », etc.

- À noter que les systèmes d'exploitation récents implémentent dans leur noyau la gestion des threads.

- Le comité responsable de standardiser le langage C++ a introduit dans le standard « C++11 », la notion de threads.

- Cette notion est fortement liée aux threads « POSIX ».

- Les anciennes notes de cours « C12-1169-v1.pdf », disponibles sur le site web du cours, utilisaient le paquetage « Posix threads », connu aussi sous le nom de « Pthread ». Pour la version « 1.02 » du cours, nous allons utiliser les threads du standard « C++11 ».

5. Threads

Le standard « C++11 » permet une gestion simplifiée des threads à travers un ensemble d'éléments définis dans « std::thread ».

Quand un programme C++ est exécuté, il lui est associé au moins un thread. Ce thread exécute la fonction « main » du programme. Ce même programme peut lancer des threads supplémentaires qui vont s'exécuter de manière concurrentielle.

5.1. Les éléments basics d'un thread

```
 1 #include <iostream>
 2 #include <thread>
 3
 4 void f(){
 5      std::cout<<"Un thread C++11\n";
 6 }
 7
 8 int main(){
 9      std::thread t(f);
10      t.join();
11      return 0;
12 }
```

- Nous avons inclus à la ligne « 2 » du programme le fichier d'en-tête « thread », afin de pouvoir utiliser les fonctionnalités associées aux threads.

- À la ligne « 9 », nous avons déclaré l'instance « t » qui représente un thread. Nous avons fait appel à un constructeur. Ce constructeur accepte comme argument une fonction. Ce thread a donc comme mission d'exécuter la fonction « f ».

- À la ligne « 10 », nous avons fait appel à la méthode « join » de la classe « thread ». Cette méthode permet de s'assurer que le thread « t » prenne fin avant que le programme ne se termine.

- Dans l'exemple qui suit, nous allons omettre de faire appel à la méthode « join » :

```
 1 #include <iostream>
 2 #include <thread>
 3
 4 void f(){
 5      std::cout<<"Un thread C++11\n";
 6 }
 7
 8 int main(){
 9      std::thread t(f);
10
11      return 0;
12 }
```

```
>.\02thread

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
Un thread C++11
terminate called without an active exception
>Exit code: 3
```

- Quand un thread est créé avec une tâche à réaliser, il est dans un état joignable. De ce fait, ce type de threads doivent être joignables ou détachés avant d'être détruits. La suppression de la ligne « 10 » a fait en sorte de mettre le thread « t » dans un état « indéterminé ». Ceci explique l'appel à « terminate ».

5.2. Les constructeurs d'un thread

Soit les deux fonctions « void f(){} » et « void g(int x){} » :

- thread t : « t » est un thread sans tâche à exécuter. Il est détaché du thread d'exécution. Le thread est dans un état « vide ». Dans ce cas, la tâche est démarrée à travers un autre thread pour être affectée par la suite à « t ».

- thread t(f) : « t » est un thread qui va exécuter la fonction « f ».

- thread t(g,5) : « t » est un thread qui va exécuter la fonction « g » dont l'argument « x » est initialisé avec la valeur « 5 ».

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

void f(){
     std::cout<<"Un thread C++11\n";
}

void g(int x){
     std::cout<<"la valeur de x est: "<< x << std::endl;
}

int main(){
     std::thread u;
     std::thread t(f);
     std::thread v(g,10);

     u = std::thread(g,20);

     t.join();
     v.join();
     u.join();

     return 0;
}
```

```
>.\03thread
Un thread C++11
la valeur de x est: 10
la valeur de x est: 20
>Exit code: 0
```

5.3. Quoi appeler dans un thread?

- Le premier argument du constructeur était un appel à une fonction.

- Dans le standard « C++11 », le mot utilisé pour décrire la nature d'un tel argument est « callable ».

- Il est possible d'appeler :

  - une fonction,
  - une méthode statique d'une classe,
  - une classe, ou une structure, qui définit l'opérateur « ( ) »,
  - une expression lambda.

- L'exemple « 04thread.cpp » montre l'utilisation des 3 premiers éléments. Nous reviendrons sur l'expression Lambda dans le prochain chapitre.

```
class w{
public:
     w() { std::cout << "constructeur w\n"; }

     void operator()() const{
          std::cout << "operateur() w\n";
          f();
     }
};
```

```
     w zz;
     std::thread h(zz);
```

5.4. Identifier un thread

- La classe thread dispose du membre « id » qui est l'identifiant du thread.
- Nous utilisons la méthode « get_id() » pour obtenir la valeur de « id ».

```
int main(){
     cout << "main, thread: " << this_thread::get_id() << endl;
     // …
     return 0;
}
```

- « this_thread » permet de savoir le thread en cours d'exécution, « get_id() » va donner l'identifiant de ce thread.

```
main, thread: 1
```

- Un appel à « get_id() » pour un thread qui n'est pas encore associé à une tâche (fonction ou autre), va retourner un message nous informant que le thread n'est pas encore dans un état exécutable.

5.5. Détacher un thread

- Un thread est créé dans un état joignable ou détaché pour éviter de lever une exception dans le programme.

- L'état joignable a déjà étudié, examinons maintenant l'état détachable.

- Pour détacher un thread, nous utilisons la méthode « detach() ».

- Un thread détaché, va fonctionner par lui-même en arrière-plan.

- Si la fonction « main » prend fin, tous les threads seront arrêtés brusquement.

- Un thread détaché ne peut plus être remis dans un état joignable.

```
std::thread t(f);
t.detach();
```

5.6 La synchronisation

- Nous avons mentionné que les threads partagent les mêmes ressources (mémoire, variables du processus, etc.).

- Dans certaines situations, il est nécessaire de synchroniser les threads pour obtenir un résultat cohérent.

- Prenez l'exemple d'un avion où il reste une seule place de disponible et deux clients se présentant à deux différents guichets. Si la place est proposée au premier client et que ce dernier prend tout son temps pour réfléchir, nous n'allons pas attribuer cette place au second client qui en a fait la demande. De ce fait, nous synchronisons l'accès à la méthode de réservation de telle manière à ne pas attribuer la même place aux deux voyageurs ou bien au second voyageur avant que le premier ne prenne sa décision de la prendre ou pas.

Monitor (ou sémaphore)

- Le moniteur est utilisé pour synchroniser l'accès à une ressource partagée.

- Cette ressource peut-être un segment d'un code donné.

- Un thread accède à cette ressource par l'intermédiaire de ce moniteur.

- Ce moniteur est attribué à un seul thread à la fois (comme dans un relais 4x100m où un seul coureur tient le témoin dans sa main pour le passer au coureur suivant dès qu'il a terminé de courir sa distance).

- Pendant que le thread exécute la ressource partagée, aucun autre thread ne peut y accéder.

- Le thread libère le monitor dès qu'il a terminé l'exécution du code synchronisé.

- Nous assurons ainsi que chaque accès aux données partagées est bien « mutuellement exclusif ».

- Nous allons utiliser pour cela les « mutex » une abréviation pour « Mutual Exclusion » ou « exclusion mutuelle ».

5.7 « Mutex »

- Un mutex est un verrou possédant deux états : déverrouillé (pour signifier qu'il est disponible) et verrouillé (pour signifier qu'il n'est pas disponible). Mutex est une variable d'exclusion mutuelle.

- Il faut inclure le fichier d'en-tête « mutex » pour avoir accès aux fonctionnalités des mutex.

- On déclare un mutex comme suit :

```
std::mutex m;
```

- Cette déclaration fait appel au constructeur par défaut. À ce stade, le mutex n'appartient à aucun thread pour le moment.

Verrouillage / Déverrouillage

- Un thread peut verrouiller le mutex s'il a besoin d'un accès exclusif aux données protégées.
- Pour verrouiller un mutex, nous utilisons la méthode « lock() ».

```
std::mutex m;
//……
m.lock();
```

- Si le mutex est déjà verrouillé par un autre thread, la fonction bloque tant que le verrouillage sur le mutex n'est pas obtenu.
- Il existe une autre façon pour verrouiller un mutex.
- Nous pouvons utiliser aussi la méthode « try_lock »

```
std::mutex m;
//……
m.try_lock();
```

- Cette méthode ne bloque pas si le mutex est déjà verrouillé par un autre thread.
- Elle retourne « true » en cas de succès.

- Dans le cas contraire, le mutex est verrouillé, elle retourne « false ».

- Quand un thread a fini de travailler avec les données « protégées », il doit libérer le mutex. Pour cela, il doit le déverrouiller pour qu'un autre thread puisse l'utiliser. La méthode « unlock() » est utilisée pour réaliser une telle opération.

```
std::mutex m;
//……
m.lock();
compteur++;
m.unlock();
```

- On peut éviter d'utiliser l'appel « unlock » et gérer soi-même les accès aux mutex. On utilise pour cela « lock_guard ». Ce dernier libère par lui-même le mutex quand l'élément qui lui est associé n'est plus visible.

5.8 Variables de condition

- Les variables offrent une autre manière aux threads pour se synchroniser.

- Au lieu de se synchroniser sur « l'accès à la donnée » comme dans le cas des mutex, les variables de condition permettent aux threads de synchroniser en fonction de la valeur de la donnée.

- À l'aide de ces variables de condition, un thread peut informer d'autres threads de l'état d'une donnée partagée.

- D'autres threads peuvent être « réveillés » si la condition sur l'état de la donnée partagée, est satisfaite.

- Une variable de condition est toujours utilisée conjointement avec « lock ».

- Nous avons besoin d'inclure « mutex » et « condition_variable » afin de déclarer un mutex et une variable de condition :

```
#include <mutex>
#include <condition_variable>


std::mutex m;
std::condition_variable v;
```

- Le thread qui doit signaler que la condition est satisfaite doit notifier le thread (ou les threads) en attente de ce changement d'état, comme suit :

```
v.notify_one(); // informer un des threads en attente
v.notifiy_all(); // informer tous les threads en attente
```

- Il faudra penser à libérer le mutex.
- Le thread en attente du changement d'état de la variable de condition doit verrouiller le mutex et se mettre en attente que la condition puisse changer.

```
std::unique_lock<std::mutex> k(m);
v.wait(k);
```

- La méthode « wait » libère le verrou sur le mutex, donne la chance à un autre thread de s'exécuter et se met en veille un certain temps, puis demande de récupérer le verrou sur le mutex.

6 Async

- L'appel à « async » permet d'exécuter une fonction (une méthode) en arrière-plan à l'aide d'un thread, si possible.

- Ainsi il est possible de faire autre chose pendant que cette méthode est en cours d'exécution.

- Cette exécution se fait de manière asynchrone ou différée.

```
std::async(unetache);
```

- Pour forcer le lancement immédiat de la tâche, on utilise « std::launch::async ».

```
std::async(std::launch::async,unetache);
```

- Pour différer le lancement, on utilise « std::launch::deferred ».

```
std::async(std::launch::deferred,unetache);
```

- La méthode « async » retourne une valeur du type « future ».

```
int unetache(){
    std::this_thread::sleep_for(std::chrono::seconds(5));
    return 42;
}

std::future<int> x = std::async(unetache);
```

- La méthode « async » renvoie le résultat dans la variable « x ». Pour y avoir accès, on fait appel à la méthode « get » de « std::future », comme suit :

```
std::cout << x.get(); //42
```

7 Atomic

- Un mutex permet de verrouiller l'accès à une section critique du code, partagée entre plusieurs threads. Cette opération peut-être couteuse en ressources et en latence.

- Pour éviter le surcoût nécessaire pour cette gestion, le « C++ » offre la possibilité de réaliser des opérations atomiques.

- Une opération atomique va s'exécuter jusqu'à son terme et ne sera pas interrompue par une autre tâche pendant son exécution.

- Soit « i » un entier initialisé avec la valeur « 1 » et soit « res += i*i » l'opération à réaliser. Cette dernière sera accédée par deux threads. Pour le premier thread, « i » vaut « 1 » et pour le second « i » vaut « 2 ».

- Le thread « 1 » lit la valeur de « i » (c.-à-d. « i=1 »). Il va la multiplier par elle-même (« 1*1 »). On aura donc, « temp1 = 1 ». Avant d'avoir eu l'occasion d'écrire le résultat en mémoire (« res=temp1 »), le thread « 2 » prend la main. Il va lire « i » (c.-à-d. « i=2 »), la multiplier par elle-même (« 2*2 ») et finalement, va écrire le résultat (« temp2 = 4 ») en mémoire ne sachant pas que le thread « 1 » a déjà réalisé le premier calcul. Quand le thread « 1 » va récupérer la main, il va écrire en mémoire le résultat de l'opération « 1 ». Oups! On devait avoir « 5 ».

```
#include <atomic>

atomic<int> i(1);
```

8. Compilateurs

- Pour utiliser les threads définis dans la nouvelle norme, « C++ 11 », il faut avoir un compilateur qui a été produit en tenant compte du modèle de threads « posix ».

- Pour gcc, la commande suivante « g++ -v » va retourner des informations sur le compilateur.

- Ceci va vous permettre de vérifier que le modèle de threads utilisé par votre compilateur est bien « posix » et non pas « win32 ».

```
g++ -v

……
Thread model: posix
……
```

- « Thread model : win32 » est le modèle natif du système d'exploitation Windows.

- La version de « gcc » fournie par « Mingw » utilise le modèle de threads « win32 ».

- Le lien suivant fournit diverses versions :

  http://mingw-w64.sourceforge.net/download.php

- La version qui nous intéresse est celle de « Mingw-builds Project ».

- Vous cliquez sur la version « 32 bits » ou « 64 bits ».

- Par la suite, il faudra choisir la dernière version disponible, puis « threads posix » puis « sjlj : set jump long jump ».

9. Références

« The C++ Standard Library », 2e edition, Nicolai M. Josuttis, chapitre 18.

« The C++ Programming Language », 4e edition, Bjarne Stroustrup, chapitre 41.

« C++, Concurrency in Action », Anthony Williams.