

## *Programmation parallèle* (PARTIE VII – C++11/14)

Bruno Bachelet

Loïc Yon

- Développement des processeurs multicoeurs
  - Limitation technique à l'augmentation des fréquences d'horloge
  - Nouvelle voie d'amélioration des performances d'une application
- Applications avec interface graphique nécessairement parallèles
  - 1 *thread* → interface graphique (gestion des événements)
  - 1 ou plusieurs *threads* → application métier
  - Evite le gel de l'interface graphique
- Beaucoup d'applications peuvent bénéficier du parallélisme
  - Possibilité d'exécution de tâches en parallèle
  - Réduction des temps de latence d'un programme
    - Accès à une ressource système (mémoire, fichier...) ⇒ latence
    - Mécanisme de mise en attente d'un *thread* ⇒ gain même sur un seul coeur
    - Règle empirique: «optimal» pour nombre de *threads* = 2 × nombre de coeurs

- La conception d'une application parallèle est difficile
  - ❑ Choisir la granularité des tâches à paralléliser
  - ❑ Synchroniser les tâches
  - ❑ Contrôler l'accès aux ressources partagées
- Avant C++11: bibliothèques/extensions non standards
  - ❑ POSIX Threads, OpenMP...
- Depuis C++11: API objet standard
  - ❑ Couche bas niveau (équivalent POSIX): `thread`, `mutex`...
  - ❑ Couche intermédiaire (abstraction): `async`, `future`...
- Avec C++17
  - ❑ Couche haut niveau (algorithmes parallèles): `for_each`, `reduce`...

- Représente un nouveau fil d'exécution dans le programme
- Nouveau contexte d'exécution
  - Possède sa propre pile
- Mais partage des données
  - Accès (lecture/écriture) au tas du programme
  - Attention aux accès concurrents
- A ne pas confondre avec un processus
  - Granularité plus fine
    - Intra-programme vs. inter-programme
  - Plus léger
    - Moins de consommation de ressources
    - Processeurs adaptés aux *threads*

- Entête: `#include <thread>`
- *Threads* représentés par la classe «`thread`»
  - Deux états sont possibles
    - Actif: représente une exécution parallèle effectivement en cours
    - Inactif: symbolise un *thread*, mais aucune exécution parallèle effective
  - Pour savoir si le *thread* est actif: méthode «`joinable`»
  - Possède un identifiant unique: méthode «`get_id`»
    - Peut être utilisé comme clé dans un conteneur associatif
- Un *thread* démarre lors de sa construction
  - Si on lui fournit un «*callable*» (i.e. fonction, foncteur ou lambda)
    - Ce *callable* est automatiquement exécuté au lancement du *thread*
    - Des arguments peuvent être fournis
    - Exemple: `t = std::thread(ma_fonction,param1,param2);`  
⇒ exécution en parallèle de `ma_fonction(param1,param2)`
  - Sinon le *thread* est inactif

- Possibilité de transférer le contrôle d'un *thread* actif
  - Exemple: `t1 = t2;`
  - Opération de mouvement (dépouillement de «`t2`»)
    - Si «`t2`» est actif, alors «`t1`» devient actif et «`t2`» inactif
  - Intérêt: permet de séparer les phases de déclaration et de lancement

```
std::thread t;  
...  
t = std::thread(ma_fonction,param1,param2);
```
- L'exécution d'un *thread* se termine à la sortie de la fonction associée
- Le *thread* principal n'attend pas automatiquement la fin des *threads* qu'il a lancé
  - Attendre chaque *thread* à l'aide de sa méthode «`join`»
  - Sinon, erreur de conception  $\Rightarrow$  erreurs très probables à l'exécution

## ■ Exemple

```
void zzz(void) {  
    std::cout << "zzz..." << std::endl;  
    std::this_thread::sleep_for(std::chrono::seconds(1));  
}
```

```
int main(void) {  
    std::thread t; // Déclaration d'un thread inactif  
  
    t = std::thread(zzz); // Exécution parallèle de la fonction  
                          // L'objet temporaire est un thread actif  
  
    t.join(); // Attente de la fin du thread  
}
```

# Lancement de plusieurs *threads*

---

## ■ Exemple

```
void zzz(unsigned n) {  
    std::cout << "[" << n << "]" zzz..." << std::endl;  
    std::this_thread::sleep_for(std::chrono::seconds(1));  
}  
  
int main(void) {  
    std::thread t[4];  
  
    for (unsigned i = 0; i<4; ++i) t[i] = std::thread(zzz,i);  
    for (unsigned i = 0; i<4; ++i) t[i].join();  
}
```



# Exécution parallèle d'une lambda

---

- Lambda sans capture

```
t[i] = std::thread(
    [](unsigned n) {
        std::cout << "[" << n << "]" zzz..." << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
    ,i);
```

- Lambda avec capture

```
t[i] = std::thread(
    [=]() {
        std::cout << "[" << i << "]" zzz..." << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
    );
```

# Accès à une ressource partagée

---

- Exemple précédent, sortie prévue (ordre non garanti)

[0] zzz...

[1] zzz...

[2] zzz...

[3] zzz...

- Mais sortie possible

[[01] zzz...] zzz...

[2] zzz...

[3] zzz...

- Car problème de partage de ressource
  - La sortie standard est partagée par tous les *threads*
  - Synchronisation nécessaire: chacun son tour  $\Rightarrow$  *mutex*

- Mutual exclusion
- Objectif: empêcher l'exécution simultanée d'une portion de code par plusieurs *threads*
- Mécanisme de jeton
  - *Mutex* partagé par plusieurs *threads*
- Acquisition d'un *mutex* (méthode «**lock**»)
  - Si le *mutex* n'est pas verrouillé, le *thread* obtient l'accès
    - La méthode «**lock**» se termine
  - Si le *mutex* est verrouillé, le *thread* se met en pause
    - La méthode «**lock**» continue
  - Possibilité d'utiliser la méthode «**try\_lock**» qui n'attend pas

- Libération d'un *mutex* (méthode «**unlock**»)
  - Doit être déclenchée par le *thread* qui a verrouillé
  - Signal aux *threads* en attente  $\Rightarrow$  l'un d'eux acquiert alors le *mutex*

- Exemple

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

```
void zzz(unsigned n) {  
    mutex.lock();  
    std::cout << "[" << n << "]" zzz..." << std::endl;  
    mutex.unlock();  
    std::this_thread::sleep_for(std::chrono::seconds(1));  
}
```

# Eviter l'interblocage de *threads* (1/2)

---

- Attention: oubli de libération  $\Rightarrow$  blocage potentiel
  - *Thread* 1: verrouillage du *mutex* *m*
  - *Thread* 2: demande d'acquisition de *m*  $\Rightarrow$  attente libération *m*
  - *Thread* 1: fin (sans libérer *m*)
  - *Thread* principal: attente fin de *thread* 2...
  
- Pour éviter les blocages  $\Rightarrow$  toujours utiliser un «verrou»
  - Il s'agit d'un *wrapper* implémentant l'idiome RAI
  - *Wrapper* = objet (verrou) qui encapsule un objet (*mutex*)
    - Il se fait passer pour l'objet  $\Rightarrow$  interface similaire
    - Il contrôle l'appel à ses méthodes
  - RAI: *Resource Acquisition Is Initialization*
    - Le constructeur acquiert le *mutex*
    - Le destructeur libère le *mutex*
  - Réduction du risque de blocage des *threads*

# Eviter l'interblocage de *threads* (2/2)

---

## ■ Plusieurs types de verrous

### □ `lock_guard`

- Interface restreinte: appel explicite à «`lock`» et «`unlock`» impossible
- Copie impossible
- Remplacé par «`scoped_lock`» en C++17

### □ `unique_lock`

- Même interface que le *mutex*
- Copie impossible, mais mouvement autorisé ( $\Rightarrow$  transfert de propriété)

## ■ Exemple

```
void zzz(unsigned n) {  
    { std::lock_guard<std::mutex> verrou(mutex);  
      std::cout << "[" << n << "]" zzz..." << std::endl; }  
  
    std::this_thread::sleep_for(std::chrono::seconds(1));  
}
```

- Mécanisme de synchronisation des *threads*
- Attendre qu'une condition sur des données partagées se réalise
  - ❑ Attente passive du *thread* (comme l'acquisition d'un *mutex*)
  - ❑ Plusieurs *threads* peuvent attendre la même condition
  - ❑ Méthode «**wait**»
- Un *thread* qui change l'état des données surveillées émet un signal vers le ou les *threads* en attente
  - ❑ Méthodes «**notify\_one**» et «**notify\_all**»

# Variable de condition (2/4)

---

- Implémentation: un *mutex* est nécessaire
- Un *mutex* est associé à la condition
  - Ce qui permet de synchroniser l'accès aux données partagées
- Signal émis  $\Rightarrow$  un ou plusieurs *threads* tentent d'acquérir le *mutex*
  - Le premier qui verrouille le *mutex* peut vérifier la condition
  - Quel que soit l'état de la condition, le *thread* devra libérer le *mutex*
    - Condition fausse  $\Rightarrow$  «wait» libère automatiquement le *mutex*
    - Condition vraie  $\Rightarrow$  «wait» termine avec le *mutex* verrouillé
  - Car les autres *threads* ayant reçu le signal sont en attente du *mutex*
- Exemple (1/4): variables globales

```
std::condition_variable condition;  
std::mutex                  mutex;  
unsigned                    compteur = 0;
```



- Exemple (2/4): 1 maître et 3 esclaves

```
int main(void) {  
    std::thread t[4];  
  
    t[0] = std::thread(maitre);  
    for (unsigned i = 1; i<4; ++i) t[i] = std::thread(esclave,i);  
    for (unsigned i = 0; i<4; ++i) t[i].join();  
}
```

- Exemple (3/4): les esclaves attendent une condition

```
void esclave(unsigned n) {  
    std::unique_lock<std::mutex> verrou(mutex);  
    std::cout << "[" << n << "]" attend..." << std::endl;  
    condition.wait(verrou,[](){ return compteur==5; });  
    std::cout << "[" << n << "]" termine" << std::endl;  
}
```

- Exemple (4/4): le maître modifie les données associées à la condition

```
void maitre(void) {  
    for (unsigned i = 0; i<5; ++i) {  
        { std::lock_guard<std::mutex> verrou(mutex);  
            std::cout << "[0] compteur = " << compteur << std::endl; }  
  
        std::this_thread::sleep_for(std::chrono::seconds(1));  
  
        { std::lock_guard<std::mutex> verrou(mutex);  
            ++compteur; }  
  
        condition.notify_all();  
    }  
}
```

- Couche pour masquer les mécanismes *multithread*
  - ❑ Simplifier le code
  - ❑ Eviter les interblocages
  - ❑ Garantir l'attente de la fin des *threads*
  
- Fonction «**async**»
  - ❑ Exécution asynchrone d'un *callable*
  - ❑ Création et démarrage automatique d'un *thread*
  - ❑ Garantie de l'attente de la fin du *thread*
  
- Objets «**future**» et «**promise**»
  - ❑ Mécanisme de synchronisation
  - ❑ Attente des résultats d'un *thread*

- Syntaxe similaire au constructeur d'un *thread*
  - ❑ Arguments: politique d'asynchronisme + *callable*
  - ❑ Politique d'asynchronisme
    - `std::launch::async`: lancement sur un nouveau *thread*
    - `std::launch::deferred`: lancement en mode «*lazy*»
- Exemple...

```
for (unsigned i = 0; i<4; ++i)
    async(std::launch::async, zzz, i);
```
- ...qui ne fait pas ce qu'on pense
  - ❑ «`async`» retourne un objet «`future`»
  - ❑ Son destructeur attend la fin du *thread* (équivalent de «`join`»)
  - ❑ Dans l'exemple, l'exécution est donc synchrone !!!

- Eviter de laisser un objet «**future**» dans une *rvalue*
  - ❑ Destruction immédiate  $\Rightarrow$  synchronisation
- Stocker l'objet «**future**» dans une variable locale
  - ❑ Variable détruite à la fin du bloc d'instructions
  - ❑ Donc bien choisir l'endroit de la déclaration

- Exemple

```
{  
    std::future<void> f[4];  
  
    for (unsigned i = 0; i<4; ++i)  
        f[i] = async(std::launch::async,zzz,i);  
  
    // Attente des threads à la fin du bloc  
}
```

- Représente la valeur retournée par l'exécution d'un *thread*
  - ❑ Encapsule le mécanisme de synchronisation du *thread*
  - ❑ L'attente de la fin du *thread* est prise en charge
  
- Méthode «**wait**»
  - ❑ Attend que le résultat soit disponible
  - ❑ Autrement dit, que le *thread* se termine
  
- Méthode «**get**»
  - ❑ Retourne le résultat une fois qu'il est disponible
  - ❑ Attend aussi que le *thread* se termine
  
- Si aucune des deux méthodes n'est appelée, le destructeur attend le résultat
  - ❑ Afin d'être sûr que la fin du *thread* sera toujours attendue

## ■ Exemple

```
double calcul(unsigned n) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    return (n+1)*(n+1);
}

int main(void) {
    std::future<double> f[4];
    double somme = 0;

    for (unsigned i = 0; i<4; ++i)
        f[i] = std::async(std::launch::async, calcul, i);

    for (unsigned i = 0; i<4; ++i) somme += f[i].get();

    std::cout << "somme = " << somme << std::endl;
}
```

- Permet à un *thread* de fournir un résultat avant la fin de son exécution
- Représente une valeur associée à un objet «**future**»
  - Lorsque le *thread* attribue une valeur à un objet «**promise**»
    - Avec la méthode «**set\_value**»
  - L'objet «**future**» associé est informé
    - Sa méthode «**get**» ou «**wait**» en attente est débloquée
- Exemple (1/2)

```
void calcul(std::promise<double> p1,  
           std::promise<double> p2) {  
    std::this_thread::sleep_for(std::chrono::seconds(1));  
    p1.set_value(3);  
    std::this_thread::sleep_for(std::chrono::seconds(1));  
    p2.set_value(7);  
}
```



- Pas de constructeur de copie
  - Utiliser «`std::move`» pour invoquer le constructeur de mouvement

- Exemple (2/2)

```
int main(void) {
    std::promise<double> p1, p2;

    std::future<double> f1 = p1.get_future();
    std::future<double> f2 = p2.get_future();

    std::future<void> f = std::async(std::launch::async, calcul,
                                   std::move(p1),
                                   std::move(p2));

    f1.wait(); std::cout << "valeur1 = " << f1.get() << std::endl;
    f2.wait(); std::cout << "valeur2 = " << f2.get() << std::endl;
}
```