CM1 - Processus légers (threads)

LIFPCA - Programmation concurrente et administration système

Florence ZARA - Université Lyon 1

Printemps 2026-2027

Source

Ce cours a été réalisé au fil des années, par les différents responsables de cette UE : Fabien Ricco, Yves Caniou, Matthieu Moy, Grégoire Pichon, Florence Zara.

Equipe pédagogique 2026-2027 : Sylvain Brandel, Yves Caniou, Carina Deaconu, Gaspard Thévenon, Florence Zara.

Plan du cours

CM1 : Processus légers (threads)

1. Introduction sur l'UE
2. Introduction
   - Théorie
   - Création et destruction des threads
3. Synchronisation
   - Section critique
   - Variables de condition
   - Sémaphore
   - Moniteur de Hoare
4. Questions

Introduction sur l'UE

- **Page Web de l'UE :**
  `https://perso.liris.cnrs.fr/florence.zara/Web/pages_html/LIFPCA.html`
- **Fonctionnement :**
  - CM / TP / TD : classique
  - QCM, DM(s) et jeu de piste !
- **Modalité de Contrôle de Connaissances :**
  - Contrôle Intermédiaire (CI)
  - Contrôle Terminal (CT)
  - Dates des épreuves sur l'emploi du temps : voir page Web
  - Note de LIPCA de session 1 = 40% CI + 60% CT
  - *Uniquement pour les étudiants n'ayant pas validé l'UE lors de la session 1, il y aura un Contrôle Terminal de Rattrapage (CTR) qui remplacera la note de CT*
  - *Note de LIPCA de session 2 = 40% CI + 60% CTR*

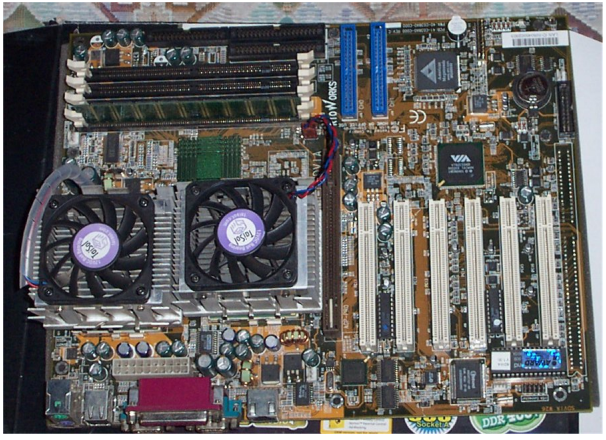
- **Page Web de l'UE :**
  `https://perso.liris.cnrs.fr/florence.zara/Web/pages_html/LIFPCA.html`
- **Fonctionnement :**
  - QCM d'auto-évaluation à faire chaque semaine :
    `http://demo710.univ-lyon1.fr/ASR7/` à terminer au max
    1 semaine après le CM correspondant.
    Nombre de tentatives illimitées.
    Vous devez avoir toutes les réponses correctes à chaque étape.
  - Beaucoup de TP en C++. Vous devez être à l'aise en C++
  - Les épreuves comprendront des problèmes avec des threads et problèmes de concurrence, de l'ordonnancement et de l'administration système.

Introduction

Parallélisme

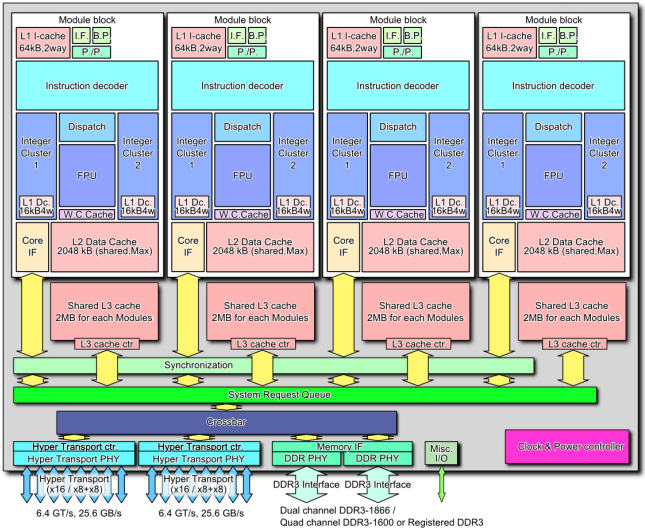
- Parallélisme = « Plusieurs choses en même temps »
- Analogie avec le travail d'équipe : on va plus vite à plusieurs (ou pas)

C'est quoi un multi-processeur ?



Source : https://commons.wikimedia.org/wiki/File:Dual_processor.jpg

Et un multi-cœur ?



=

Multi-processeur

- Plusieurs puces « processeur » dans un ordinateur
- En général, utilisé sur les machines haut-de-gamme
- Synonyme : multi-socket

Processeur multi-cœur

- **2001** : premier multi-cœur (power 4), puis $\approx$ **2010** : généralisation du multi-cœur
- Depuis **2020** : présent partout
  - smartphone bas de gamme = 2 cœurs / haut de gamme = 4 à 8 cœurs,
  - ordinateur personnel = 2 à 8 cœurs / serveur = souvent $\geq$ 16 cœurs
- Également dans les processeurs spécialisés :
  - GPU haut de gamme = >1000 unités de traitement.
  - Processeurs many-core = centaines de cœurs.
- Un ordinateur peut contenir plusieurs processeurs multi-cœur.

Dans les deux cas

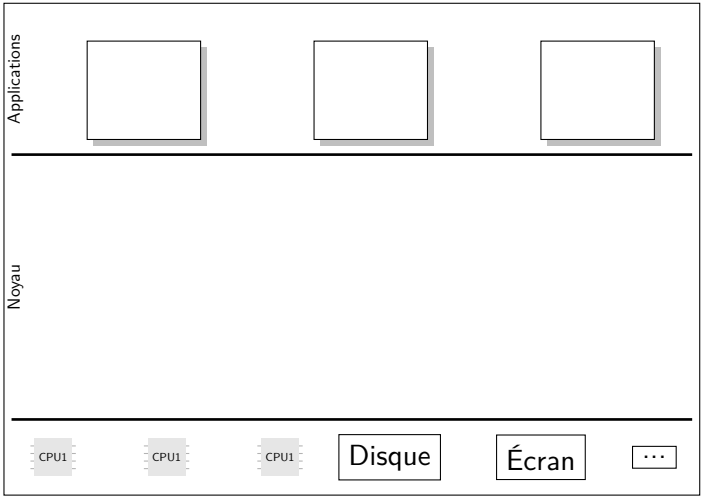
- Plusieurs cœurs/processeurs n'accélèrent pas les calculs *séquentiels*
- L'ordinateur fait « plusieurs choses à la fois » (parallélisme) pour aller plus vite
- Besoin de gérer le parallélisme (répartition du travail, ressources partagées)

Parallélisme ...

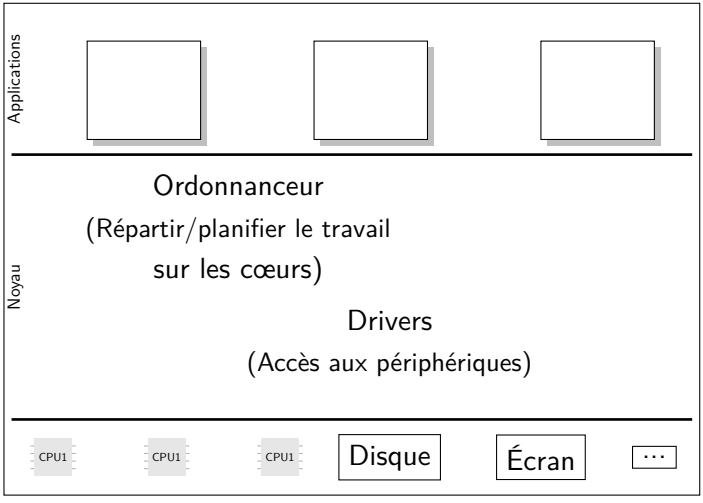
```
int main() {
    int T[1000];
    for (int i = 0; i < 1000; ++i)
        T[i] = i;
}
```

$\rightsquigarrow$ Ce programme va-t-il plus vite sur une machine multi-cœur (ou multi-processeur) que sur une machine simple cœur ?

Rôles du système d'exploitation



Rôles du système d'exploitation



Objectifs de ce cours

- Comprendre les mécanismes de gestion du multi-tâches dans un OS
- Savoir les exploiter pour écrire des programmes parallèles
- Connaître les problèmes classiques liés au parallélisme et les solutions pour les éviter

Première notion fondamentale : les processus

Processus

- Permet de partager un même matériel entre plusieurs utilisateurs
- Isole les programmes
  - Communications simplifiées mais encadrées
  - Sécurité
  - Stabilité (exemple : crash de Firefox ne doit pas gêner LibreOffice, voire un crash d'un onglet dans Firefox ne doit pas impacter les autres onglets)
- Notion des années 70, toujours d'actualité
- Pas toujours suffisant (1 utilisateur voulant faire du calcul)

`http://www.numerama.com/tech/196257-le-multi-processus-se-deploie-dans-firefox-comment-en-profiter.html`

Exemple

Le mini chat, code sur la page du cours

Exemple

Le mini chat, code sur la page du cours

- La même application doit pouvoir enregistrer les évènements de l'utilisateur et attendre un message du réseau.

Exemple

Le mini chat, code sur la page du cours

- La même application doit pouvoir enregistrer les évènements de l'utilisateur et attendre un message du réseau.
- **1re idée de solution**, il faut :
  - Un *processus* qui lit sur le réseau (en attente)
  - Un *processus* qui fait le reste (affichage, saisie du clavier ...)
- Mais comment les faire communiquer sans se retrouver avec le même problème ?

Exemple

Le mini chat, code sur la page du cours

- La même application doit pouvoir enregistrer les évènements de l'utilisateur et attendre un message du réseau.
- **1re idée de solution**, il faut :
  - Un *processus* qui lit sur le réseau (en attente)
  - Un *processus* qui fait le reste (affichage, saisie du clavier ...)
- Mais comment les faire communiquer sans se retrouver avec le même problème ?
- **2e idée :**
  - Un *fil de traitement* ou *thread* qui lit et écrit le résultat dans une variable.
  - Un *fil de traitement* qui fait le reste (notamment l'affichage de la variable).
  - Une structure de données accessible par les deux pour communiquer (la fameuse variable).
- C'est un problème de **producteur/consommateur** simplifié

Exemple

Le mini chat, code sur la page du cours

- La même application doit pouvoir enregistrer les évènements de l'utilisateur et attendre un message du réseau.
- **1re idée de solution**, il faut :
  - Un *processus* qui lit sur le réseau (en attente)
  - Un *processus* qui fait le reste (affichage, saisie du clavier ...)
- Mais comment les faire communiquer sans se retrouver avec le même problème ?
- **2e idée :**
  - Un *fil de traitement* ou *thread* qui lit et écrit le résultat dans une variable.
  - Un *fil de traitement* qui fait le reste (notamment l'affichage de la variable).
  - Une structure de données accessible par les deux pour communiquer (la fameuse variable).
- C'est un problème de **producteur/consommateur** simplifié

Attention

La solution présentée a seulement pour but de présenter le cours.
Ce n'est pas la meilleure solution à ce problème.

Processus lourd ou processus léger (thread) ?

Processus

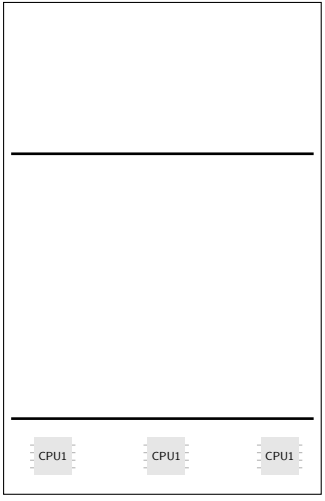
- Contient tout ce qui est nécessaire à l'exécution (registres du processeur, ressources, mémoire...)
- Commutation de contexte lente
- Communication par des appels système
- Programmation simple

Thread - Processus léger

- Un processus contient un ou plusieurs threads
- Chaque thread a son propre compteur programme, sa propre pile
- Ces threads partagent les mêmes ressources, la même mémoire
- Commutation de contexte plus rapide
- Communications = partage de variables. Simple ?
- Attention aux variables partagées

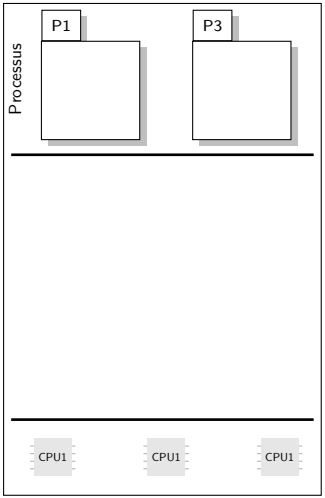
1. Introduction sur l'UE
2. Introduction
    - Théorie
    - Création et destruction des threads
3. Synchronisation
    - Section critique
    - Variables de condition
    - Sémaphore
    - Moniteur de Hoare
4. Questions

Thread/processus



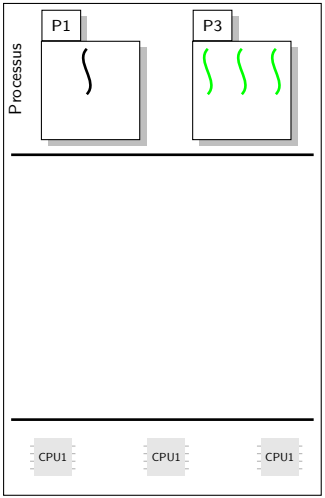
Thread/processus

- Les processus sont des rassemblements de ressources pour un programme



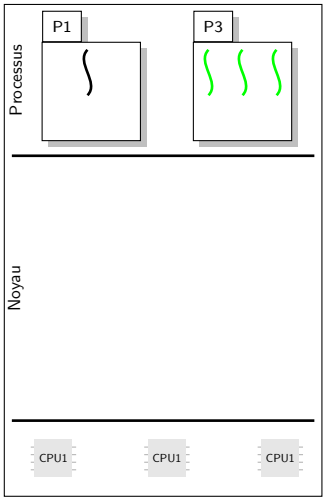
Thread/processus

- Les processus sont des rassemblements de ressources pour un programme
- Les threads sont des chemins d'exécution dans les processus



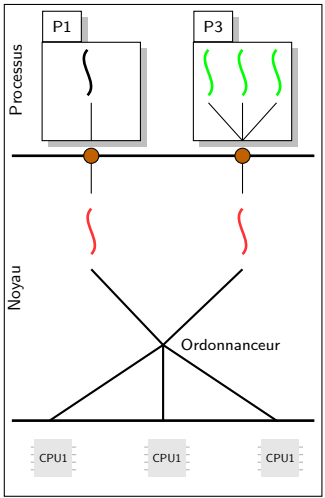
Thread/processus

- Les processus sont des rassemblements de ressources pour un programme
- Les threads sont des chemins d'exécution dans les processus
- Mais, comment sont gérés les threads par le noyau ?



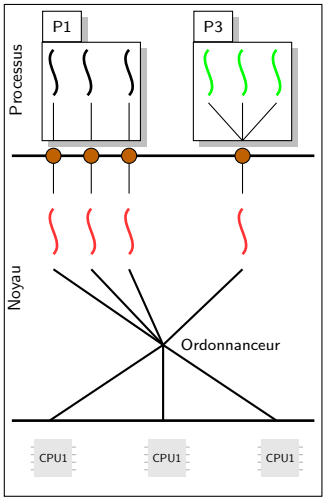
Thread/processus

- Les processus sont des rassemblements de ressources pour un programme
- Les threads sont des chemins d'exécution dans les processus
- Mais, comment sont gérés les threads par le noyau ?
  - Le noyau ne voit que les processus ?



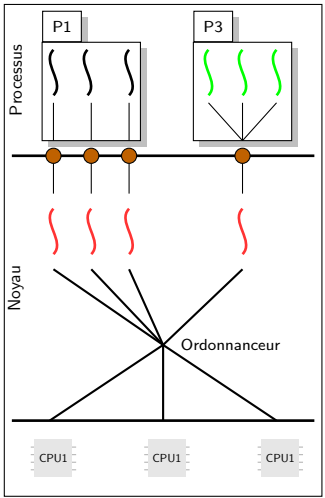
Thread/processus

- Les processus sont des rassemblements de ressources pour un programme
- Les threads sont des chemins d'exécution dans les processus
- Mais, comment sont gérés les threads par le noyau ?
  - Le noyau ne voit que les processus ?
  - Ou le noyau a connaissance des threads ?



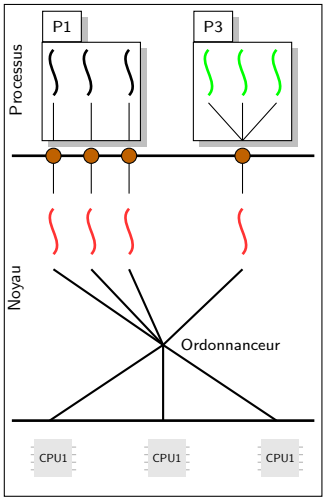
Thread/processus

- Les processus sont des rassemblements de ressources pour un programme
- Les threads sont des chemins d'exécution dans les processus
- Mais, comment sont gérés les threads par le noyau ?
  - Le noyau ne voit que les processus ?
  - Ou le noyau a connaissance des threads ?
- Les états *prêt*, *en exécution*, *bloqué* sont-il des états de threads ?



Thread/processus

- Les processus sont des rassemblements de ressources pour un programme
- Les threads sont des chemins d'exécution dans les processus
- Mais, comment sont gérés les threads par le noyau ?
  - Le noyau ne voit que les processus ?
  - Ou le noyau a connaissance des threads ?
- Les états *prêt*, *en exécution*, *bloqué* sont-il des états de threads ?
- Que se passe-t-il quand un thread fait un **fork** ?



Thread utilisateur - green thread

- Le noyau n'a pas connaissance des threads
- Portable
- Gestion sans appel système
- Ce n'est qu'une simulation de parallélisme

Thread lié - thread noyau

- Un thread système pour chaque thread
- Modèle simple
- Gestion par appel système
- Modèle depuis Linux 2.5 - 2.6

Threads multiplexés

- Entre les deux précédents
- Plusieurs threads systèmes affectés à un processus qui a lui-même plusieurs threads
- Plus souple
- Plus complexe
- Modèle Windows XP et +

Différences Thread/Processus

Dans les 2 cas suivants, pour programmer le serveur, utiliseriez-vous des threads ou des processus et pourquoi ?

- Serveur de connexion à distance (type terminal serveur ou ssh)

- Serveur de fichiers (type NFS ou SMB)

1. Introduction sur l'UE
2. Introduction
   - Théorie
   - Création et destruction des threads
3. Synchronisation
   - Section critique
   - Variables de condition
   - Sémaphore
   - Moniteur de Hoare
4. Questions

Les threads en C++

En utilisant la classe thread de la bibliothèque standard C++ 2011

- Construction d'un thread

```
template <class Fn, class... Args>
explicit thread::thread (Fn&& fn, // fonction à appeler
                         Args&&... args);
                         // arguments de la fonction
```

  La liste des arguments est dynamique grâce au template.

- Attente de la fin d'un thread (pas de résultat)

```
void join();
```

Attention

Sous linux, la bibliothèque utilisée est aussi la bibliothèque pthread, il faut donc ajouter les options de compilation : −std=c++11 −pthread

Exemple (dispo sur la page du cours)

```
// g++ thread-uneFonction.cpp -std=c++11 -pthread
#include <thread>
#include <iostream>
using namespace std;

int num = 3;
void uneFonction(int i, string mess) {
        cout << "Je suis le thread " << i
             << " mon message est : " << mess << endl;
        num = num+10;
}
int main() {
        string message = "coucou";
        thread t(uneFonction, 1, message);
        // ...
        t.join();
        cout << "Le nombre est :" << num << endl;
}
```

Attention

Cela semble simple mais :

- l'utilisation d'un template pose des problèmes, par exemple
  - Si certains paramètres sont des réferences (&) cela donne une référence sur un objet temporaire ce qui est impossible. Dans ce cas, il faut utiliser std :: ref () ou remplacer le passage par référence par un passage par adresse (∗, comme en C)
  - Le compilateur maîtrise mal les template et cela rend les messages d'erreur incompréhensibles
- On ne peut pas copier un thread ⇒ le créer directement au bon endroit et ne pas utiliser l'affectation [1].
- Il faut faire attention à la destruction automatique de la variable contenant le thread.

1. Ou alors maîtriser les subtilités du constructeur par déplacement et de std :: move() en C++11, cf. cours de prog avancée en M1

Synchronisation

Retour sur l'exemple du chat

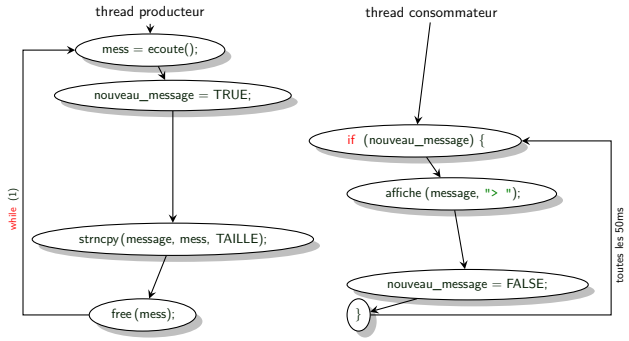
Dans l'exemple il y a un thread qui produit une information et un thread qui la lit :

- C'est un problème de **producteur/consommateur**
- Très simplifié
- Un seul producteur
- Un seul consommateur
- Une file d'attente avec une seule case
- On peut cependant voir les problèmes posés

1. Introduction sur l'UE
2. Introduction
   - Théorie
   - Création et destruction des threads
3. Synchronisation
   - Section critique
   - Variables de condition
   - Sémaphore
   - Moniteur de Hoare
4. Questions

mini-chat

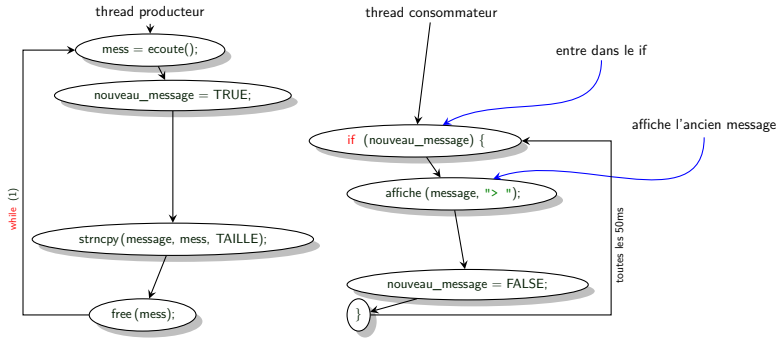
Dans l'exemple du mini-chat, l'échange d'informations entre les deux threads nécessite 2 variables partagées nouveau_message et message. Elles doivent être modifiées en une seule fois. **Que se passe-t-il si le thread perd la main au milieu de la modification ?**



Section critique - Exemple

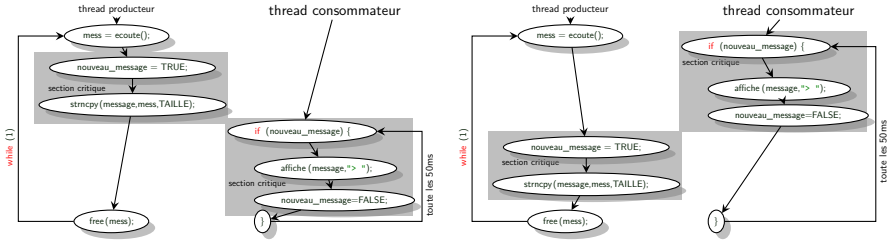
mini-chat

Dans l'exemple du mini-chat, l'échange d'informations entre les deux threads nécessite 2 variables partagées nouveau_message et message. Elles doivent être modifiées en une seule fois. **Que se passe-t-il si le thread perd la main au milieu de la modification ?**



Section critique - Exemple

Certaines parties du code ne peuvent pas être « mélangées » lors de l'exécution, elles forment une *section critique*



Définition

Section critique

On appelle *section critique* une ou plusieurs sections de code où **il ne doit y avoir qu'un thread à la fois**

- Pour ne pas accéder en même temps en écriture à des données partagées.
- Pour ne pas modifier une donnée qu'on est en train de lire.

Attention

- À cause du multithreading **il faut « protéger » l'accès à ces sections critiques**
- En utilisant une variable partagée ?

```
while (!passage_autorise) attend();
passage_autorise = FALSE;
```

- **NON : Le problème est que le test de la variable et sa mise à jour forment une section critique.**
  Il faut *tester et modifier* la variable en même temps !

Définition

Pour mettre en place les sections critiques, on peut utiliser le système.

Il propose des variables partagées qui peuvent être **testées et modifiées de manière unitaire** : les *verrous* ou *mutex* (**mut**ual **ex**clusion).

mutex

Un *mutex* est une primitive de synchronisation, elle permet :

- de vérifier qu'on est autorisé à passer
- d'interdire le passage aux autres

Ce qu'il faut bien comprendre

Les deux actions sont faites en *une seule instruction atomique*

Cela veut dire qu'une instruction qui ne peut pas être interrompue

Mutex en C++

En utilisant la classe std :: mutex

- Le constructeur :

```
mutex()
```

  crée et initialise le mutex, cette action est atomique.
- Le verrouillage :

```
void lock()
```

  verrouille ou bloque si le mutex est déjà utilisé.
- La tentative de verrouillage :

```
bool try_lock()
```

  si le mutex est libre, la méthode le verrouille et retourne vrai, sinon, elle retourne faux (sans verrouiller).
- La libération du mutex :

```
bool unlock()
```

lock ou *trylock* ?

Ce qu'il faut comprendre

- lock **bloque le thread** pendant que la section critique est occupée. Pendant que le thread est bloqué, il n'utilise pas inutilement le processeur. Il sera réveillé lors de la libération du mutex. Cela permet de faire une *attente passive*.
- try_lock **ne bloque pas le thread**. Il est donc possible de faire autre chose puis de réessayer la section critique. Cela permet de faire une *attente active*.

Attention

Un thread en section critique bloque les autres. Il faut minimiser la durée du séjour.

Si vous n'avez pas une bonne raison d'utiliser trylock, ne l'utilisez pas.

Exemple : compteur (sans protection)

```
// g++ -std=c++11 -pthread thread-compteur.cpp
#include <thread>
#include <iostream>
using namespace std;
#define NB 100000000

class Compteur {
public:
  void incr() {
    valeur++;
  }
  Compteur() {
    valeur = 0;
  }
  int valeur;
};

void compte (Compteur &c) {
  for (int i = 0; i < NB; ++i)
    c.incr();
}

int main() {
  Compteur c;
  thread t1(compte, ref(c));
  thread t2(compte, ref(c));
  t1.join();
  t2.join();
  cout << c.valeur << endl;
}
```

Affiche :

Exemple : compteur (sans protection)

```
// g++ -std=c++11 -pthread thread-compteur.cpp
#include <thread>
#include <iostream>
using namespace std;
#define NB 100000000

class Compteur {
public:
  void incr() {
    valeur++;
  }
  Compteur() {
    valeur = 0;
  }
  int valeur;
};

void compte (Compteur &c) {
  for (int i = 0; i < NB; ++i)
    c.incr();
}

int main() {
  Compteur c;
  thread t1(compte, ref(c));
  thread t2(compte, ref(c));
  t1.join();
  t2.join();
  cout << c.valeur << endl;
}
```

Affiche : 100791270 (ou 105302176, ou 107127837, ...)

Exemple : compteur avec mutex (avec protection)

```
#include <mutex>

class Compteur {
public:
  mutex m; // Déclaration du mutex
           // (et initialisation implicite)
  int valeur = 0;
  void incr() {
    m.lock(); // Verrouillage

    valeur++; // Section
              // critique

    m.unlock(); // Déverouillage
  }
};
```

Affiche :

Exemple : compteur avec mutex (avec protection)

```
#include <mutex>

class Compteur {
public:
  mutex m; // Déclaration du mutex
           // (et initialisation implicite)
  int valeur = 0;
  void incr() {
    m.lock(); // Verrouillage

    valeur++; // Section
              // critique

    m.unlock(); // Déverouillage
  }
};
```

Affiche : 200000000 (toujours !), mais met plus de temps

Automatiser la libération du mutex

Il peut arriver « d'oublier » de libérer le mutex.

Par exemple à cause d'un return (ou d'une exception) :

```
int f() {
  m.lock();
  ...
  if (error) {
    return -1; // Oups, on a oublié le m.unlock() !
  }
  ...
  m.unlock();
}
```

Automatiser la libération du mutex

Pour cela on peut utiliser un unique_lock,
qui est un objet dont le constructeur réserve le mutex et dont le destructeur le libère :

```
int f() {
  std::unique_lock<std::mutex> lck(m);
  ...
  if (error) {
    return -1; // Appel automatique du destructeur (merci C++).
               // Le mutex est libéré.
  }
  ...
  // Pas de .unlock() à faire, c'est automatique
}
```

Exemple : compteur avec unique_lock

```
class Compteur {
public:
  mutex m;
  int valeur = 0;
  void incr() {
    unique_lock<std::mutex> l(m);
    valeur++;
  }
};
```

Fait exactement la même chose que la version précédente, mais « plus pratique ».

Ce qu'il faut retenir

Définition

- Section critique = portion de code qui doit rester en **exclusion mutuelle**
- Mutex = outil fourni par le système pour faire des sections critiques

Cas plusieurs threads

accès à variable partagée $\Rightarrow$ protection par mutex nécessaire

1. Introduction sur l'UE
2. Introduction
   - Théorie
   - Création et destruction des threads
3. Synchronisation
   - Section critique
   - Variables de condition
   - Sémaphore
   - Moniteur de Hoare
4. Questions

Variable de condition - Exemple

Mini-chat

Dans l'exemple, tous les échanges se font grâce à une seule variable.

Que se passe-t-il si le consommateur est lent et que la variable est occupée ?

Variable de condition - Exemple

Mini-chat

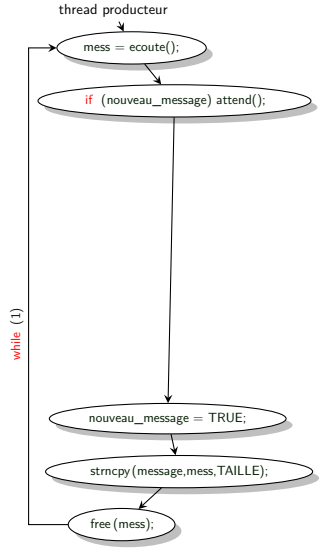
Dans l'exemple, tous les échanges se font grâce à une seule variable.

Que se passe-t-il si le consommateur est lent et que la variable est occupée ?

Dans l'idéal,

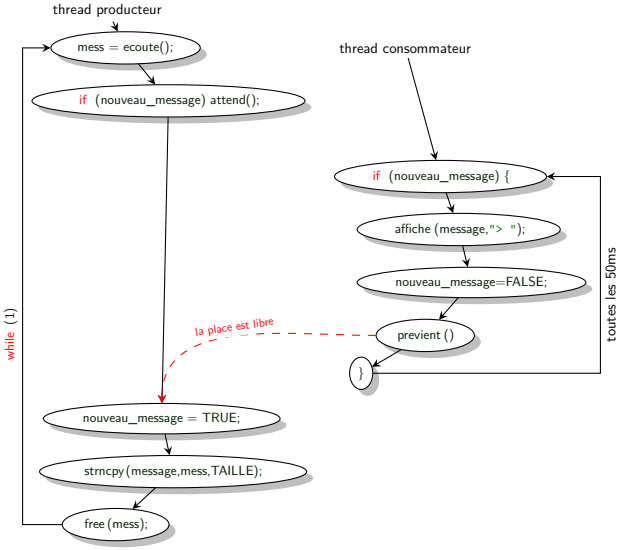
- Si la variable est libre, le producteur l'utilise
- Sinon ?
- Une attente active utilise le processeur pour rien
- Une attente passive, comment ?

Attendre une condition

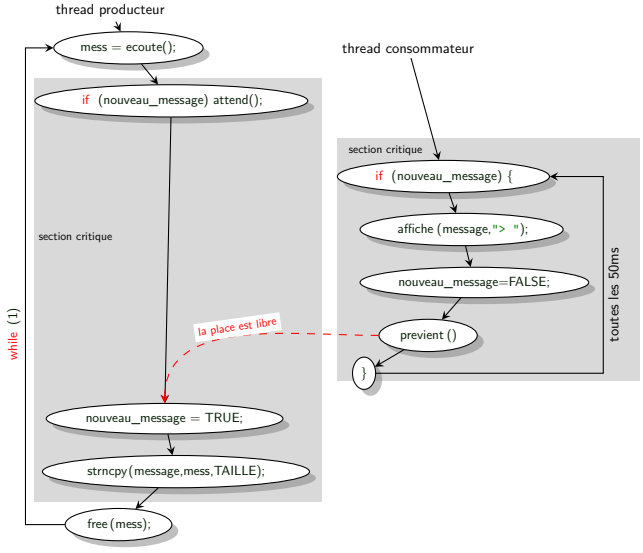


thread consommateur

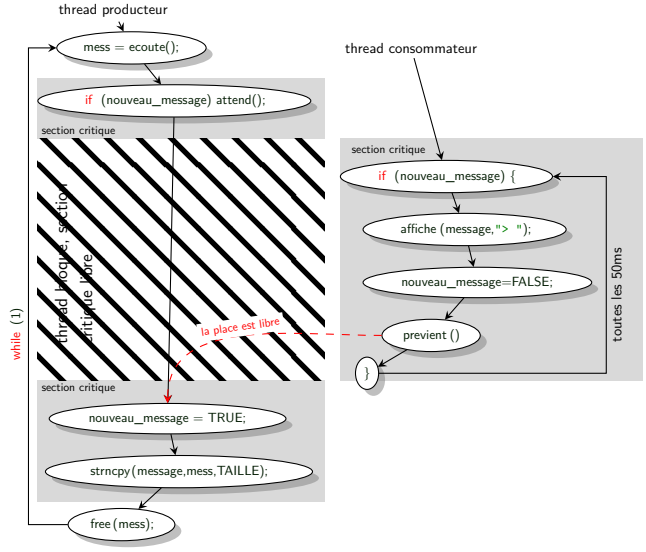
Attendre une condition



Attendre une condition



Attendre une condition



Attente passive du producteur

S'il n'y a plus de place, attendre jusqu'à ce que le consommateur dise qu'il y a de la place.

Pour une attente passive, il faut :

- Un moyen d'attendre jusqu'à un « évènement ».
- Peut-on utiliser les primitives d'attente et les signaux (`pause()` et `pthread_kill ()`) ?
  Non car si le thread n'attend pas, il ne faut rien signaler.
- Il faut que le test et l'attente soient atomiques.
- Il faut que le thread libère le mutex pendant l'attente.
- Il faut que le thread ré-obtienne le mutex dès son déblocage (de façon atomique).

Définition - Variable de condition

Variable de condition

La *variable de condition* est une variable partagée
qui permet l'attente dans une section critique.

En utilisant les classes `std :: condition_variable` (avec unique lock) ou
`std :: condition_variable_any` avec les mutex

Définition - Variable de condition

- Le constructeur

```
condition_variable();
condition_variable_any();
```

- L'attente →

http://en.cppreference.com/w/cpp/thread/condition_variable/wait

```
void wait(unique_lock<mutex>& lck);
```

Attend un notify. Dans la documentation, il est conseillé de l'utiliser dans une boucle while car « certaines implémentations ont des déblocages intempestifs ».

- Le déblocage

```
void notify_one();
void notify_all();
```

Notifie un thread en attente ou tous les threads en attente.

Exemple : compteur positif

- But : compteur avec méthode incr() et decr()
- Valeur toujours positive
- Si on fait decr() sur un compteur à 0, on attend que le compteur repasse > 0.
- Première tentative :

```
class Compteur {
public:
  mutex m;
  int valeur = 0;
  void incr() {
    unique_lock<std::mutex> l(m);
    valeur++;
  }
  void decr() {
    unique_lock<std::mutex> l(m);
    valeur--;
    //assert(valeur >= 0); // Temporaire
  }
};
```

Exemple : compteur positif

- Avec variable de condition :

```
#include <condition_variable>
struct Compteur {
  mutex m;
  condition_variable c; // Déclaration
  int valeur = 0;
  void incr() {
    unique_lock<std::mutex> l(m);
    valeur++;
    c.notify_one(); // Réveiller decr()
  }
  void decr() {
    unique_lock<std::mutex> l(m);
    while (!(valeur > 0)) {
      c.wait(l); // Attendre incr()
    }
    valeur--;
  }
```

Sémaphore

Défini par Edsger Dijkstra, c'est le premier outil de synchronisation :

- C'est un compteur partagé.
- Il est initialisé avec une valeur positive ou nulle.
- La fonction **V** (Verhogen) permet de l'incrémenter.
- La fonction **P** (Proberen) permet de le tester et le décrémenter en une seule opération atomique.
- *Le compteur n'est jamais négatif*,
  P est bloquante si la valeur du compteur n'est pas suffisante.

Un *mutex* peut être vu comme un sémaphore initialisé à 1.

Notre compteur positif est un sémaphore initialisé à 0.

Utilisation des sémaphores

- Attribution de ressources ou jeton :
  Par exemple plusieurs tâches partagent un tampon T et un sémaphore $S_{donnees}$ initialisé à 0 qui indique le nombre de données disponibles.

Lire dans le tampon

Data: m taille des données à lire

begin

  P($S_{donnees}$) // Attend qu'une donnée soit disponible

  data ← retire(T,m)

end

Result: data

Écrire dans le tampon

Data: *data* des données de taille n

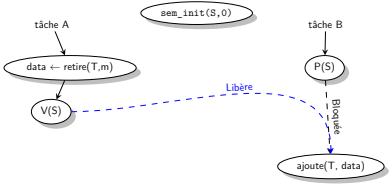
begin

  ajoute(T,data)

  $V(S_{donnees})$ `// signale qu'une donnée est disponible`

end

- Imposer un ordre entre des tâches



Utilisation des sémaphores

- Attribution de ressources ou jeton :

  Par exemple plusieurs tâches partagent un tampon T de taille L, un sémaphore $S_{donnees}$ initialisé à 0 qui indique le nombre de données disponibles, et un sémaphore S_{vide} initialisé à L qui indique le nombre de cases vides disponibles.

Lire dans le tampon

Data: m taille des données à lire

begin

P($S_{donnees}$) // Attend qu'une donnée soit disponible

data ← retire(T,m)

V(S_{vide}) // signale qu'il // y a de la place

end

Result: data

Écrire dans le tampon

Data: *data* des données de taille n

begin

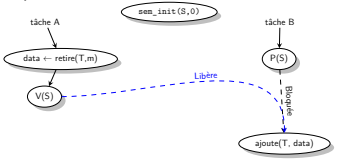
$P(S_{vide})$ `// bloque s'il n'y a pas assez de place`

ajoute(T,data)

$V(S_{donnees})$ `// signale qu'une donnée est disponible`

end

- Imposer un ordre entre des tâches



1. Introduction sur l'UE
2. Introduction
    - Théorie
    - Création et destruction des threads
3. Synchronisation
    - Section critique
    - Variables de condition
    - Sémaphore
    - Moniteur de Hoare
4. Questions

Besoins

Comment aider le programmeur à utiliser les primitives de synchronisation ?

- Encapsuler l'utilisation des mutex et des conditions ...
- Arbitrer l'accès aux variables partagées
- Automatiser l'acquisition et la libération des mutex
- ...

Définition

Moniteur de Hoare = Notion proche de la programmation objet.

Moniteur de Hoare

Objet qui gère une ressource ou un ensemble de ressources.

- contient les **données partagées**,
- définit des **accesseurs (accès en lecture)**, et des **mutateurs (accès en écriture)**,
- utilise l'**exclusion mutuelle**
  dans tous les threads, les accès ne peuvent pas être faits en parallèle.
- permet l'**attente qu'une condition devienne vraie** (wait).

On programme un moniteur pour résoudre un problème particulier et centraliser les utilisations de primitives de synchronisation. Cela permet de les tester, voire de prouver leur bon fonctionnement.

Ce n'est pas forcément la solution la plus efficace, mais c'est souvent la plus simple.
→ Pensez également à la maintenance du code !

Moniteurs de Hoare en C++

- 1 Moniteur = 1 classe contenant :
  - **Données**, de préférence privées (comme d'habitude en programmation objet)
  - **Méthodes**, de préférence privées (comme d'habitude en programmation objet)
  - **1 mutex**
    - qui est verrouillé en début de chaque méthode
    - et libéré en fin de méthode (unique_lock)
  - Éventuellement une ou plusieurs **variables de condition**

Exemple

Notre compteur positif !

Nous avons utilisé le principe du moniteur de Hoare (sans le nommer)

Moniteur de Hoare - Ce qu'il faut retenir

Centraliser les accès à des données partagées via un moniteur

- Plus simple que d'ajouter des primitives de synchro partout dans le code
- Centralisation dans une classe
- Règles de programmation simples
- Une fois le moniteur créé, on peut le réutiliser sans danger
- Mais, cela utilise beaucoup le mécanisme d'exclusion mutuelle

  $\Rightarrow$ ralentissement important du programme

  si le compteur est manipulé régulièrement
- Peut-on faire mieux ?
  - Est-ce que la lecture doit être en exclusion avec l'écriture ?
  - Peut-on utiliser plusieurs compteurs modifiables en parallèle ?

    et rassembler les données uniquement lorsque cela est nécessaire ?

Conclusion

Contenu de ce cours

1. Introduction sur l'UE
2. Introduction
   - Théorie
   - Création et destruction des threads
3. Synchronisation
   - Section critique
   - Variables de condition
   - Sémaphore
   - Moniteur de Hoare
4. Questions

CM1 - Ce qu'il faut retenir

Thread

- Différences entre thread (=processus léger) /processus (lourd)
- Programmation multithread

Synchronisation

- Section critique et exclusion mutuelle
- Utilisation des mutex et variables de condition, de sémaphores
- Moniteurs (de Hoare)

Questions

Le bug classique ...

Si deux threads font `x++` sur une variable partagée, alors `x` est incrémenté de

- A : 1
- B : 2
- C : ça dépend

La solution pour éviter ça, c'est ...

Le bug classique ...

Si deux threads font `x++` sur une variable partagée, alors `x` est incrémenté de

- A : 1
- B : 2
- C : ça dépend

La solution pour éviter ça, c'est de protéger l'accès à `x` par un **mutex** (ou, plus généralement, de rendre la section critique mutuellement exclusive)

Le vocabulaire à apprendre

- Une section de code exécutée au plus par 1 processus en même temps, c'est ...
- L'outil utilisé pour empêcher d'avoir plus de 1 processus en même temps dans une section de code, c'est ...
- L'outil utilisé pour faire de l'attente passive, c'est ...
- Une manière classique d'utiliser les deux outils ci-dessus, c'est le ... de ...
- Une généralisation du principe de mutex qui permet deompter les ressources disponibles, et de bloquer quand il n'y a plus de ressource, c'est ...

Le vocabulaire à apprendre

- Une section de code exécutée au plus par 1 processus en même temps, c'est une section critique.
- L'outil utilisé pour empêcher d'avoir plus de 1 processus en même temps dans une section de code, c'est un mutex.
- L'outil utilisé pour faire de l'attente passive, c'est une variable de conditions.
- Une manière classique d'utiliser les deux outils ci-dessus, c'est le Moniteur de Hoare.
- Une généralisation du principe de mutex qui permet deompter les ressources disponibles, et de bloquer quand il n'y a plus de ressource, c'est un sémaphore.