Processus légers (threads)

LIFPCA-Programmation concurrente et administration système

Grégoire Pichon

Univ. Claude Bernard Lyon 1

Printemps 2024

Sylvain Brandel, Yves Caniou, Guillaume Damiand, Meriem Ghali, Laurent Lefèvre, Thibaut Modrzyk, Grégoire Pichon, Alec Sadler, Florence Zara, Jerry Lacmou Zeutouo

LIFPCA : programmation concurrente et administration système 1/2

- Page du cours : `https://asr-lyon1.gitlabpages.inria.fr/prog-concurrente/`
- Fonctionnement :
  - CM / TP / TD : classique, et QCM, DM(s) et jeu de piste !
  - Évaluation :
    - Examen de TP
    - Contrôle milieu parcours
    - Contrôle final
    - Peut-être des Qroc surprises (en CM, TD ou TP)
    - Dates sur la page du cours, dans Tomuss

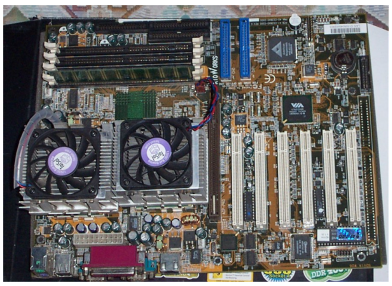
LIFPCA : programmation concurrente et administration système 2/2

- Page du cours : `https://asr-lyon1.gitlabpages.inria.fr/prog-concurrente/`
- Fonctionnement :
  - QCM d'auto-évaluation à faire chaque semaine : `http://demo710.univ-lyon1.fr/ASR7/` à terminer au max **1 semaine après le cours** correspondant. Nombre de tentatives illimitées, mais vous devez avoir toutes les réponses correctes à chaque étape.
  - Beaucoup de TP (C++). Vous devez être à l'aise en C++ (⇒ préparez-vous ...) !
  - Des examens avec : des threads et problèmes de concurrence, mais aussi avec de l'ordo (Mid) et en plus de l'administration système (CT).
  - Une évolution de la maîtrise : vocabulaire, recul sur les problèmes dont les solutions sont moins guidées.

Parallélisme

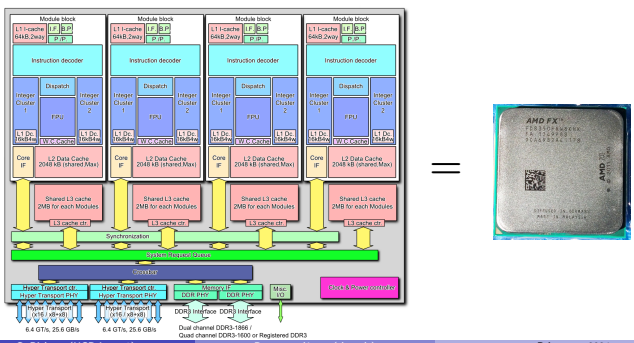
- Parallélisme = « Plusieurs choses en même temps »
- Analogie avec le travail d'équipes : on va plus vite à plusieurs (ou pas)

C'est quoi un multi-processeur ?



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

Et un multi-cœur ?



=

Source : https://upload.wikimedia.org/wikipedia/commons/e/ec/AMD_Bulldozer_block_diagram_(8_core_CPU).PNG

Multi-cœur/processeur

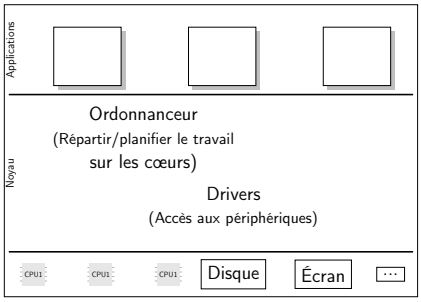
- Multi-processeur :
  - Plusieurs puces « processeur » dans un ordinateur
  - En général, utilisé sur les machines haut-de-gamme
  - Synonyme : multi-socket
- Processeur multi-cœur :
  - **2001** : Premier multi-cœur (power 4)
  - ≈ **2010** : Généralisation du multi-cœur
  - **2020** : présent partout (smartphone bas de gamme = 2 cœurs, smartphone haut de gamme = 4 à 8 cœurs, ordinateur personnel = 2 à 8 cœurs, serveur = souvent ≥ 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;
}
```

⇝ 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



Ce chapitre

- Comprendre les mécanismes de gestion du multi-tâche 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 [a])
- Notion des années 70, toujours d'actualité
- Pas toujours suffisant (1 utilisateur voulant faire du calcul //)

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

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

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

Processus Léger (thread) ou Processus Lourd ?

Processus

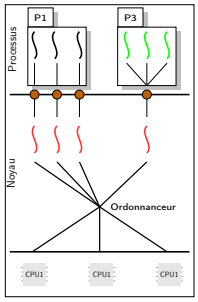
- Contient tout ce qui est nécessaire à l'exécution (registres du processeur, ressources, mémoire...)
- Commutation de contexte lente
- Communication par des appels système
- Programmation simple

Thread

- 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

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** ?



Modèles - I

Définition (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

Définition (Thread lié - thread noyau)

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

Modèles - II

Définition (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)

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

Retour sur l'exemple du chat

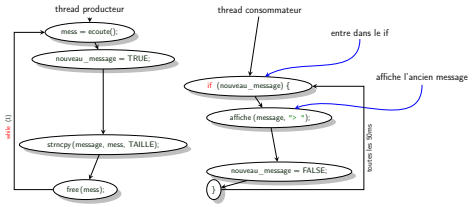
Dans l'exemple il y a un thread qui produit une information et un thread qui la lit :

- 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

Section critique

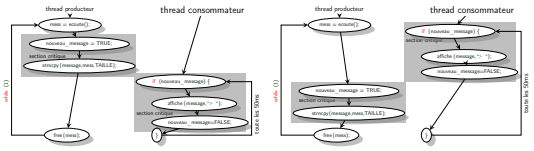
Exemple (mini-chat)

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

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



Définition

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.

- 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.

Mutex

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).

Définition (mutex)

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

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

Les deux actions sont faites en *une seule instruction atomique* (*i.e.*, 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* ?

- `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*.

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

```cpp
// 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

```cpp
#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

Il peut arriver « d'oublier » de libérer le mutex. Par exemple à cause d'un return (ou d'une exception) :

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

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

```cpp
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

```cpp
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 ».

Conclusion sur les sections critiques/mutex

- Section critique : portion de code qui doit rester en exclusion mutuelle
- Mutex : outil fourni par le système pour faire des sections critiques
- Sauf cas particulier :
  accès à variable partagée ⇒ protection par mutex nécessaire

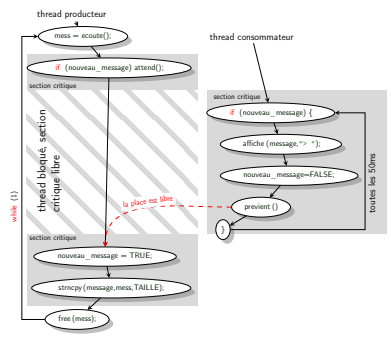
Variable de condition

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



Besoins

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).

Variables de condition

Définition (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

- 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--;
  }
};
```


Le sémaphore

Définition (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.

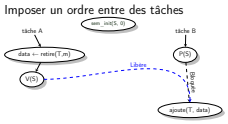
Utilisation des sémaphores

- Attribution de ressources ou jeton :
  Par exemple plusieurs tâches partagent un tampon T de taille L, et un sémaphore S_{donnes} 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
Données : m taille des données à lire
début
  P($S_{donnees}$) // Attend qu'une donnée soit disponible
  data ← retire(T,m)
  V(S_{vide}) // signale qu'il // y a de la place
fin
Résultat : data

Écrire dans le tampon
Données : *data* des données de taille n
début
  P(S_{vide}) // bloque si il n'y a pas assez de place
  ajoute(T, data)
  V($S_{donnees}$) // Signale qu'il y a une donnée
fin

- Imposer un ordre entre des tâches



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.
- ...

Moniteur de Hoare

C'est une notion proche de la programmation objet.

Définition (Moniteur)

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 (wait) qu'une condition devienne vraie.

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 (idem)
  - 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)

Moniteurs de Hoare en Java

- Certains langages facilitent la définition de moniteurs (ex : en Java les méthodes synchronized).

Exemple (En java)

```
public class compteur {
    private int valeur = 0;
    public synchronized void incr() {
        valeur++;
    }
    ...
```

Le verrou est alors associé à l'objet *synchronisé*, on peut utiliser des conditions avec wait()/ notify ()

Moniteur : conclusion

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 : utilise beaucoup l'exclusion mutuelle ;
- ⇒ 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 ?

À retenir

Thread

- Différences thread/processus
- Programmation multithread

Synchronisation

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

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 ...