Processus légers

ASR7-Programmation concurrente

Yves Caniou

Univ. Claude Bernard Lyon 1

séance 1

Yves Caniou	yves.caniou@univ-lyon1.fr	CM + TD + TPs
Matthieu Moy	matthieu.moy@univ-lyon1.fr	TD + TP
Frédéric Suter	frederic.suter@cc.in2p3.fr	TPs

1. Introduction
   - Théorie
   - Création et destruction des threads

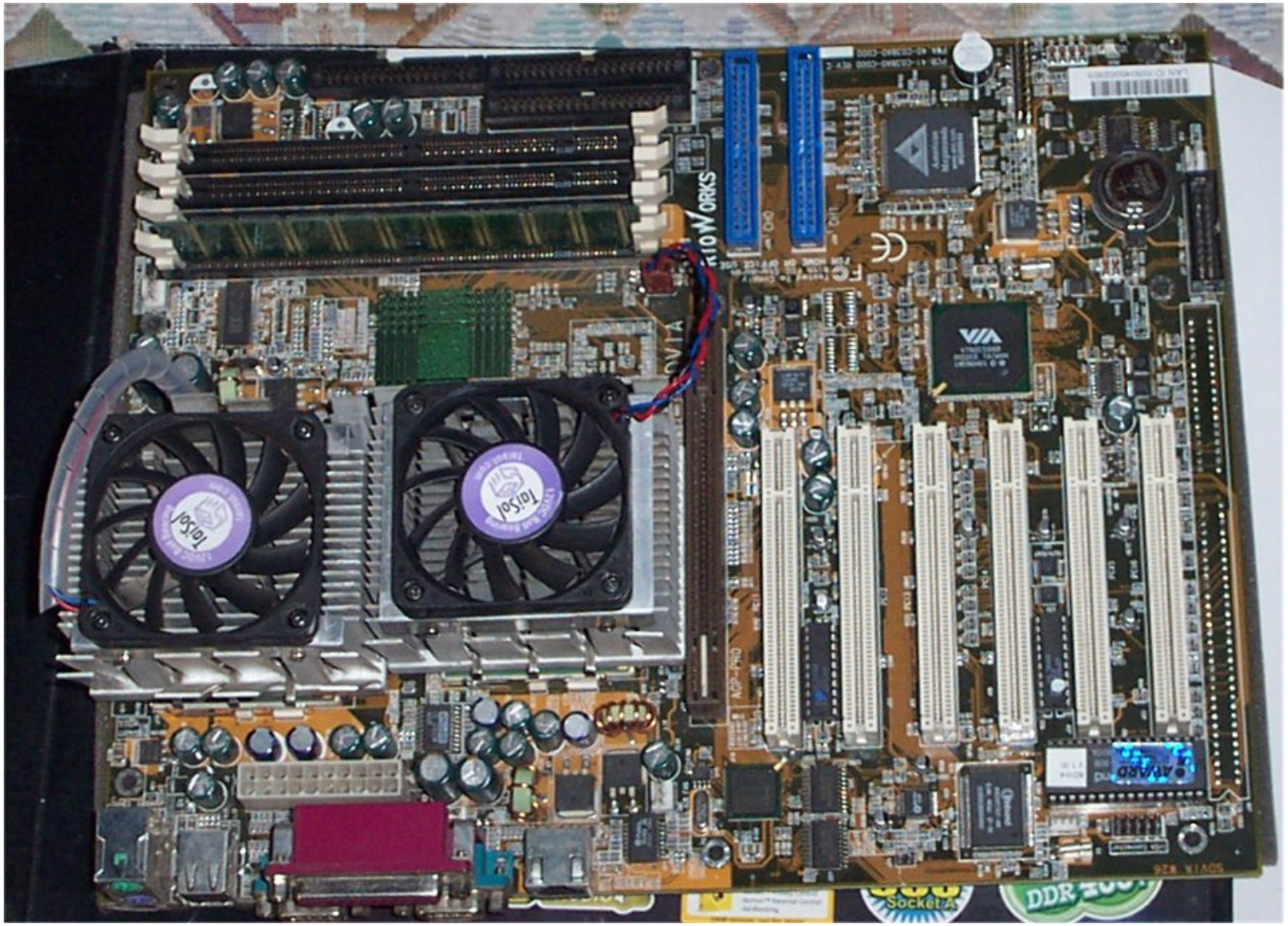
2. Synchronisation
   - Section critique
   - Variables de condition
   - Sémaphore
   - Moniteur

Parallélisme ...

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

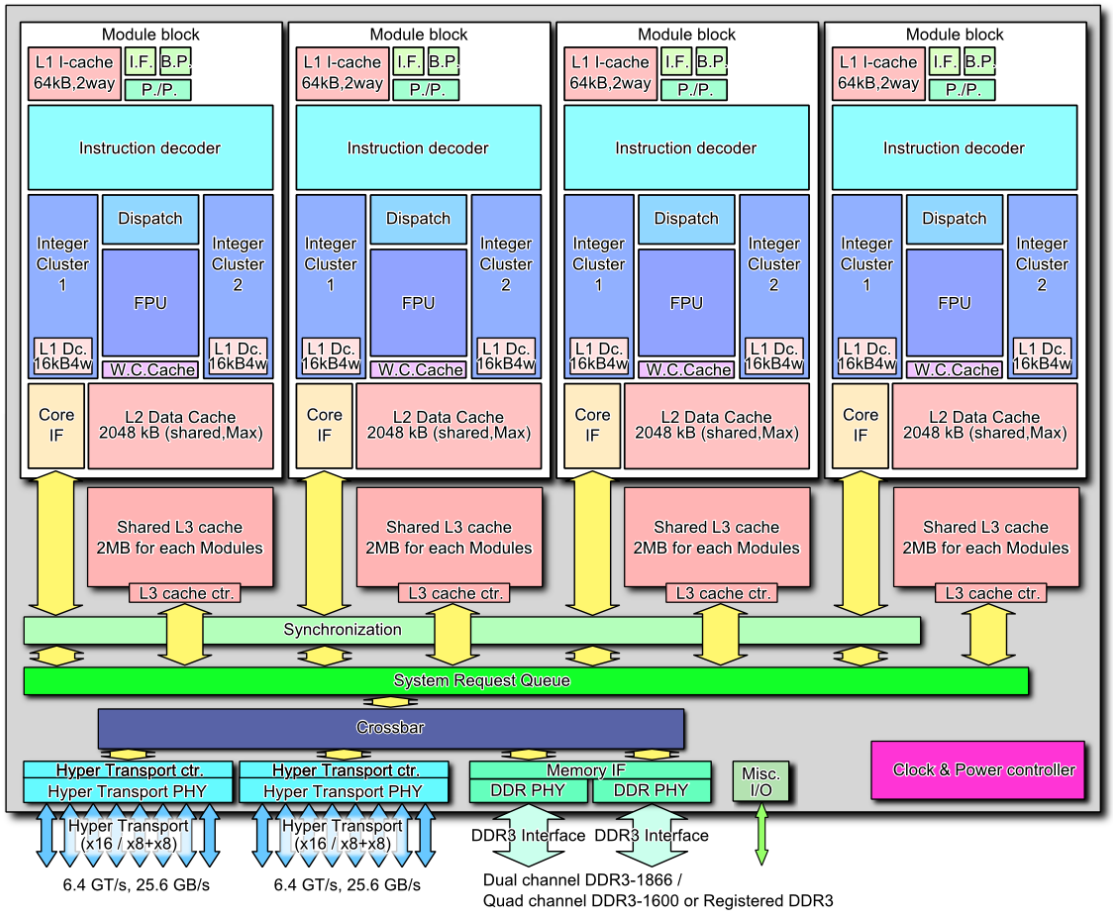
$\rightsquigarrow$ Ce programme va-t-il plus vite sur une machine multi-cœur (ou multi-processeur) que sur une machine simple cœur ?

C'est quoi un multi-processeur ?



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

Et un multi-cœurs ?



=

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
- Processeur multi-cœur :
  - **2001** : Premier multi-cœur (power 4)
  - $\approx$ **2010** : Généralisation du multi-cœur
  - **2018** : présent partout (smartphone bas de gamme = 2 cœurs, smartphone haut de gamme = 4 à 8 cœurs, ordinateur personnel = 2 à 4 cœurs, serveur = souvent $\geq$ 8 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-cœurs.
- 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)

Première notion fondamentale : les processus

Processus

- Permet de partager un même matériel entre plusieurs utilisateurs
- Isole les programmes
  - Communications simplifiées mais encadrées
  - Sécurité
  - Stabilité
- Notion des années 70 (plusieurs utilisateurs sur un gros serveur)
- Pas toujours adaptés (1 utilisateur voulant faire du calcul //)
- Parfois difficile à utiliser (exécuter une action précise parrallèlement au programme principal)

Exemple (Le mini chat)

- Le même processus 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.

Léger et lourd ?

Processus

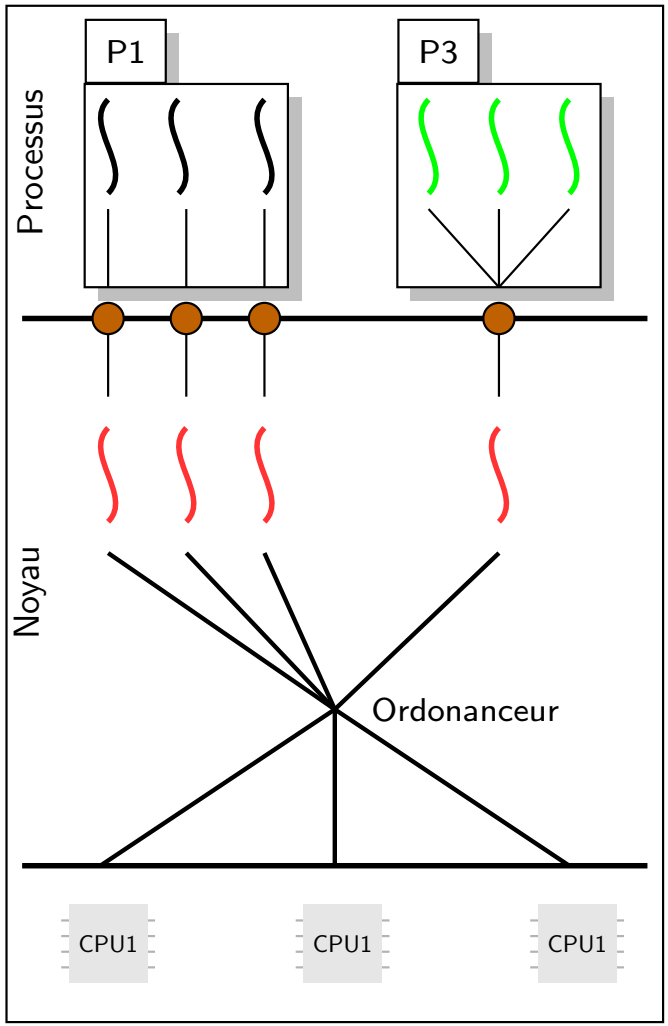
- Contient tout ce qui est nécessaire à l'exécution (registre du processeur, ressources, mémoire...)
- Commutation de contexte lente
- Communication par des appels système
- Programmation simple

Thread

- Un processus contient plusieurs threads
- 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 multi-thread

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 Thead/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 et −lpthread

Exemple

```
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";
  int num = 3;
  thread t(uneFonction, num, message);
  ...
  t.join();
  cout << "le nombre est " << num << endl;
}
```

Attention

Cela semble simple mais :

- l'utilisation d'un template pose des problèmes par exemple
  - Il provoque une copie d'objet (or certains objets ne doivent pas être copiés).
  - 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 () ;
  - Le compilateur maîtrise mal les template et cela rend les messages d'erreur incompréhensibles.
- On ne peut utiliser que des fonctions statiques (et pas les méthodes d'un objet).
- On ne peut pas copier un thread (il faut utiliser std : :move()).
- Il faut faire attention à la destruction automatique de la variable contenant le thread.

Retour sur l'exemple

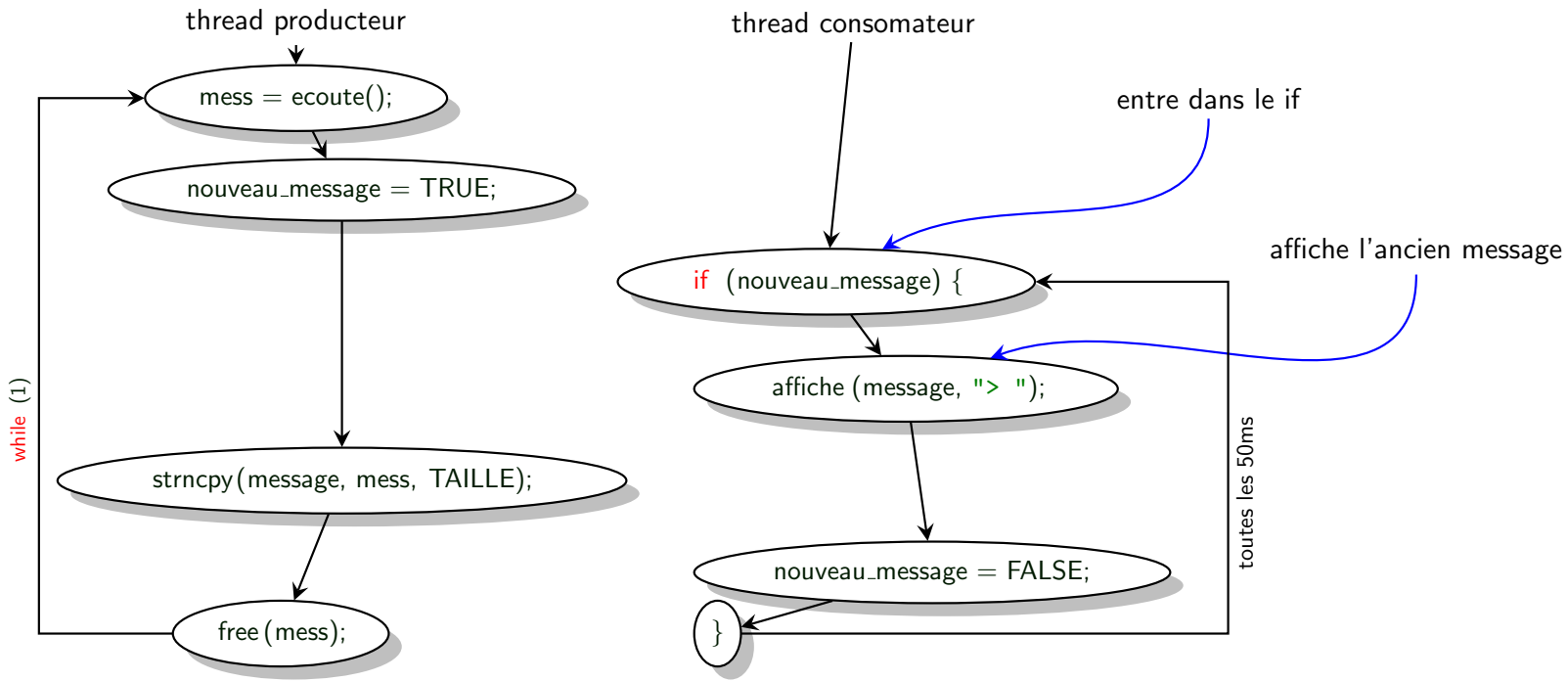
Dans l'exemple il y a un thread qui produit une information et un thread qui la lit :

- 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

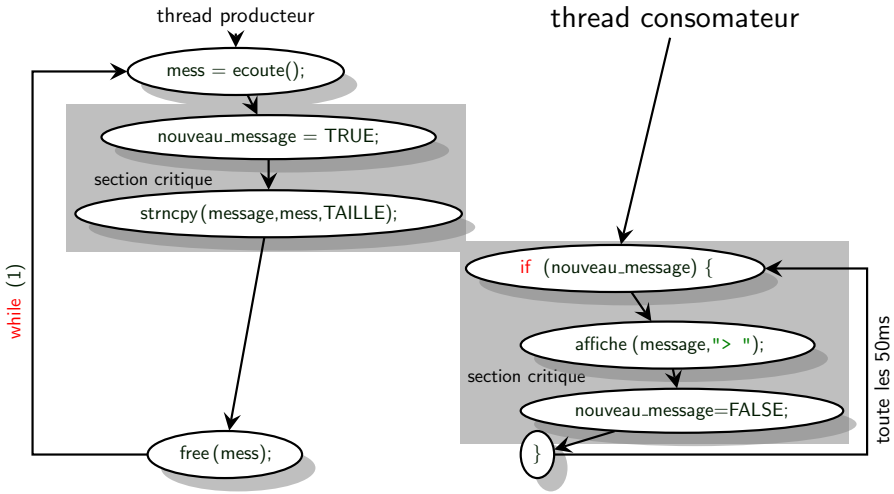
Exemple (mini-chat)

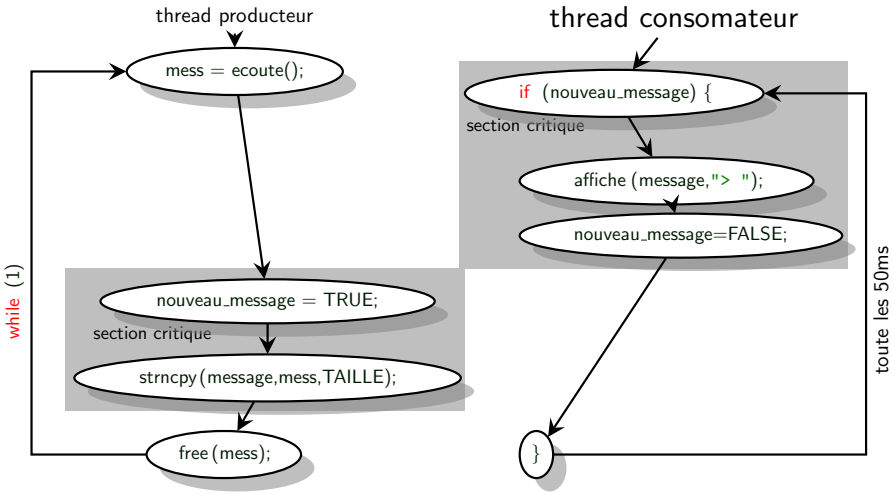
Dans l'exemple du mini-chat, l'échange d'informations entre les deux threads nécessite 2 variables partagées nouveau_message et message. Elles doivent être modifiées en une seule fois. Que se passe-t'il si le thread perd la main au millieu de la modification ?



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;
```

- 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*.

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 renvoie vrai, sinon, elle renvoie faux.
- 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.

Automatiser la libération

Il peut arriver « d'oublier » de libérer le mutex. Par exemple à cause d'une exception

```
try{
  m.lock();
  ...
  if (error) {
    throw(std::system::error("Problème de lecture sur la socket"));
  }
  ...
  m.unlock();
} catch (std::system::error &e) {
  //!!! le mutex n'est pas libéré
}
```

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

```
try{
  std::unique_lock<std::mutex> lck(m);
  ...
  if (error) {
    throw(std::system::error("Problème de lecture sur la socket"));
  }
  ...
} catch (std::system::error &e) {
  //!!! le mutex est libéré
}
```

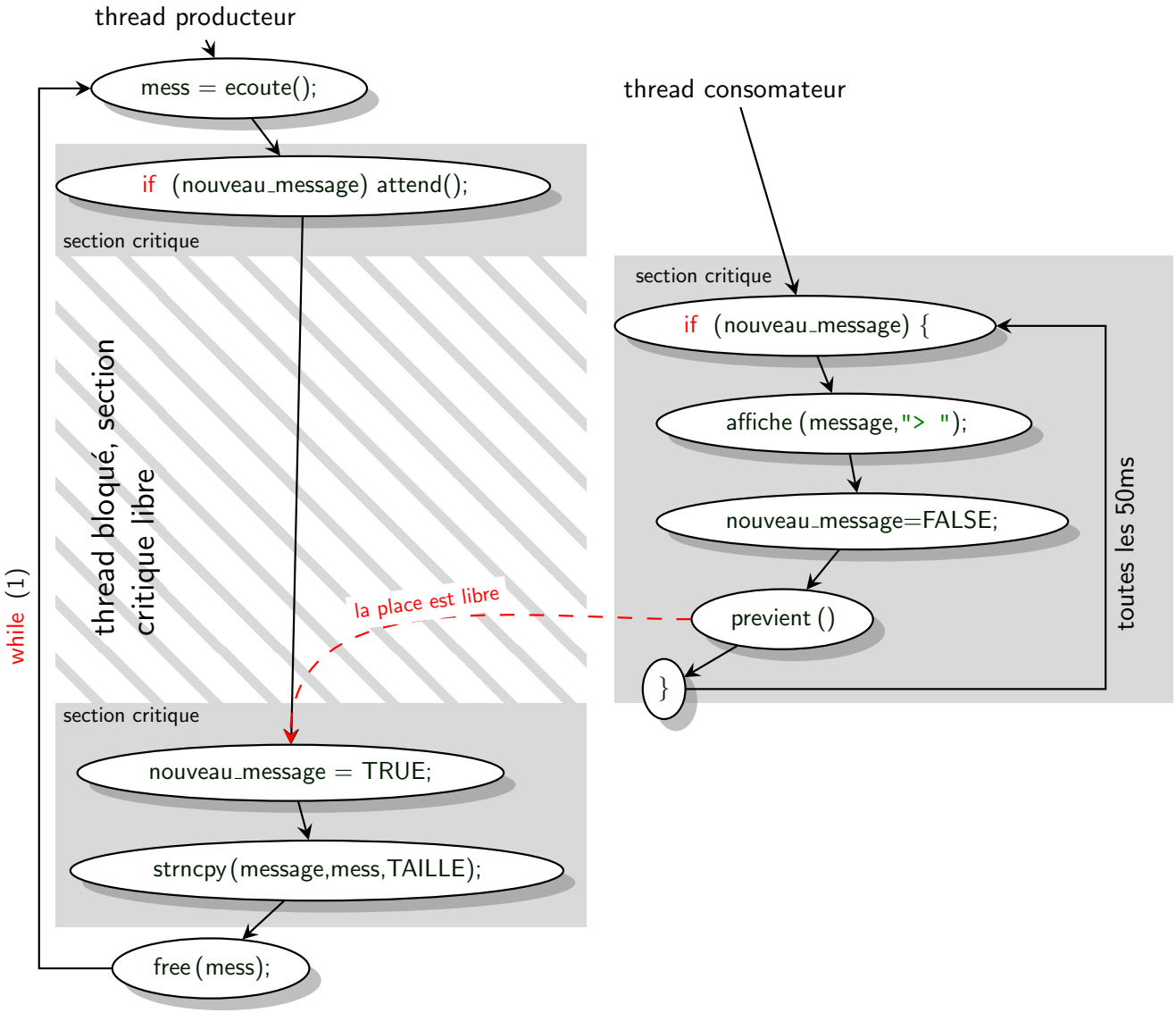
Variable de condition

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 unitaires.
- 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 unitaire).

Appels système

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 mutexes

- 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);
template <class Predicate>
void wait (unique_lock<mutex>& lck,
           Predicate pred);
```

Ou

```
while (!pred()) {
  wait(lock);
}
```

Attend un notify. Le prédicat est une fonction sans argument qui est testée au début et à la sortie du bloquage. Le wait ne bloque que si le retour est faux et ne débloque que s'il est vrai.
Dans la documentation, ils est conseillé de l'utiliser 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.

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.

Utilisation des sémaphores

- Attribution de ressources ou jeton :
  Par exemple plusieurs tâches partagent un tampon T de taille L et un sémaphore S initialisé à L

Lire dans le tampon
Données : *m* taille des données à lire

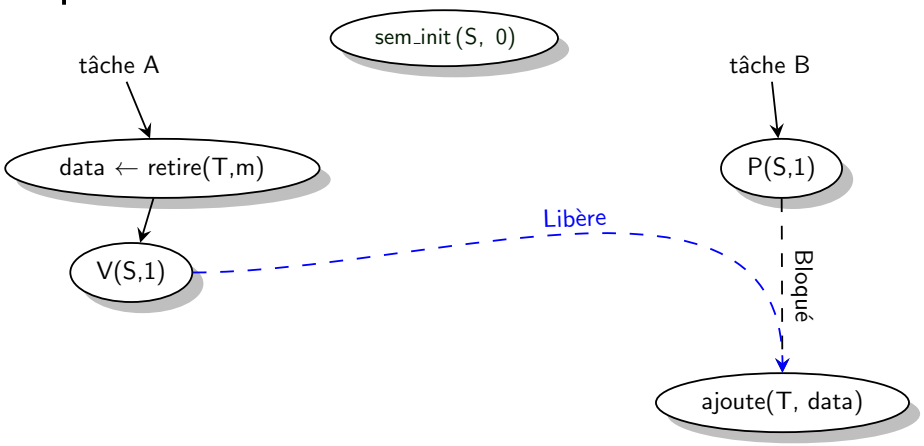
début
  data ← retire(T,m)
  V(S, m) // signale qu'il // y a de la place
fin

Résultat : data

Écrire dans le tampon
Données : *data* des données de taille *n*

début
  P(S, n) // bloque si il n'y a pas // assez de place
  ajoute(T, data)
fin

- Imposer un ordre entre tâches



Comment aider le programmeur à utiliser les primitives de synchronisation ?

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

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 partagée,
- 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.

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 efficaces, mais c'est souvent la plus simple.

Comment faire ?

- Certains langages facilite la définition de moniteurs (ex : en JAVA les méthodes synchronized)
- On peut simuler cela avec d'autres langages grâce aux mutexes et/ou au unique_lock
  - L'objet moniteur contient un mutex
  - Toutes les méthodes commencent par l'acquisition du mutex sous la forme d'un unique_lock qui est libéré automatiquement.

Le compteur

Exemple (Compteur)

Le compteur est une variable :

- partagée ;
- accessible en lecture et écriture ;
- ... mais pas en même temps.

Moniteur : exemple du compteur

```
class Compteur{
  int val;
public:
  Compteur(int v=0);
  void add(int v=1);
  void sub(int v=1);
  int get();
};

Compteur::Compteur(int v) : val(v) {}
void
Compteur::add(int v) {
  val += v;
}
void
Compteur::sub(int v){
  val -= v;
}
```

Où est le problème ?

```
Compteur::add(int v) {
   val += v;
}
```

- Que peut-il se passer si on programme les fonctions de cette manière ? Proposer un scénario avec 2 threads qui démontre le problème.
- Ce scénario est-il probable ? Est-ce bon signe ?

Moniteur : exemple du compteur

Le principe du moniteur est que le corps de chaque fonction est une section critique.

- Dans certains langages (JAVA) il suffit de spécifier cela à la déclaration de la méthode

Exemple (En java)

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

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

- En C++, il faut :
  - Soit utiliser un verrou qu'on récupère au début et libère à la fin.
  - Soit utiliser un unique_lock.

Moniteur C++ : ajout d'un verrou

- Création avec le mutex

```
class Compteur{
  int val;
  std::mutex m;
public:
  Compteur(int v=0);
  void add(int v=1);
  int get();
};
Compteur::Compteur(int v) : val(v), m() {
}
```

- Utilisation du verrou unique_lock

```
void
Compteur::add(int v) {
  unique_lock<std::mutex> lck(m);
  val += v;
}
```

Moniteur : conclusion

Centraliser les accès à des données partagées via un moniteur

- c'est plus simple ;
- une fois le moniteur crée on peut le réutiliser sans danger ;
- mais : utilise beaucoup l'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 ?

1. Introduction
   - Théorie
   - Création et destruction des threads

2. Synchronisation
   - Section critique
   - Variables de condition
   - Sémaphore
   - Moniteur

À retenir

Thread

- Différences thread/processus
- Programmation multithread

Synchronisation

- Section critique et exclusion mutuelle
- Utilisation des mutexes et Variables de condition
- Moniteur