Multithreading et synchronisation

Guillaume Salagnac

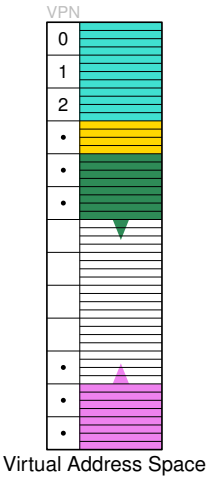
Insa de Lyon – Informatique

Plan

Rappel : la notion de processus

Définition : processus
«un programme en cours d'exécution»

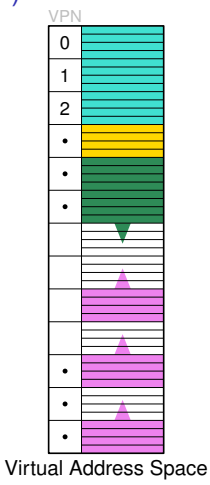
- isolés les uns des autres
  - en temps : CPU virtuel
  - en espace : mémoire virtuelle
- Process Control Block
  - numéro = PID
  - environnement, répertoire courant, fichiers ouverts...
  - copie des registres CPU
  - vue mémoire = Page Table
- Page Table
  - instructions = .text
  - variables globales = .data
  - tas d'allocation = .heap
  - pile d'exécution = .stack



Notion de thread (VF fil d'exécution)

Définition : thread
«une tâche indépendante à l'intérieur d'un processus»

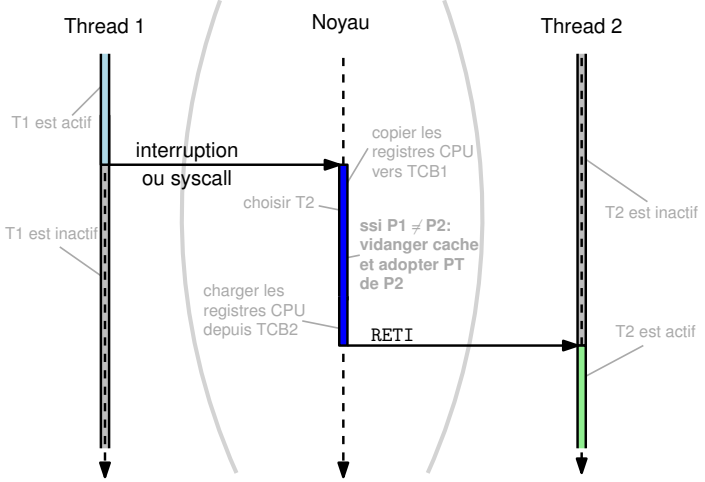
- pourquoi les threads ?
  - profiter de plusieurs CPU
  - faciliter la programmation
- vue mémoire commune
  - pas d'isolation matérielle
  - variables globales partagées
  - tas d'allocation commun
- ordonnancement indépendant
  - un VCPU privé
    TCB = *Thread Control Block*
  - une pile d'exécution privée
  - variables locales privées



Notion de thread : remarques

- parfois appelé «processus léger» mais vision archaïque
  - en vrai : un PCB = une PT et un/plusieurs TCB
  - par ex : `task_struct` et `mm_struct` dans Linux
- un thread ne peut pas vivre en dehors d'un processus
  - besoin d'une vue mémoire
- un processus vivant a toujours au moins un thread
  - «*main thread*» = thread qui exécute `main()`
  - lorsque zéro thread ▶ processus terminé

Rappel : changement de contexte



API POSIX : Threads

```
#include <pthread.h>

/* opaque typedefs */ pthread_t, pthread_attr_t;

// create and start a new thread
int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void * (*function) (void *),
                   void *arg);

// terminate the current thread
void pthread_exit(void *retval);
// terminate another thread
int pthread_cancel(pthread_t thread);

// wait for another thread to terminate
int pthread_join(pthread_t thread, void **retvalp);
```


Plan

Accès concurrents à une variable partagée

Variable partagée

```
int var = 5;
```

Thread A

```
{
  ...
  ...
  var = var+1;
  ...
  ...
}
```

Thread B

```
{
  ...
  ...
  var = var-1;
  ...
  ...
}
```

Question : que vaut `var` à la fin de l'exécution ?

- intuition : var==5
- réalité : var==5 ou var==4 ou var==6

Explication : code source ≠ instructions du processeur

Variable partagée

```
var: 00000005
```

Thread A

```
...
...
LOAD   REGa←[var]
INCR   REGa
STORE  REGa→[var]
...
...
```

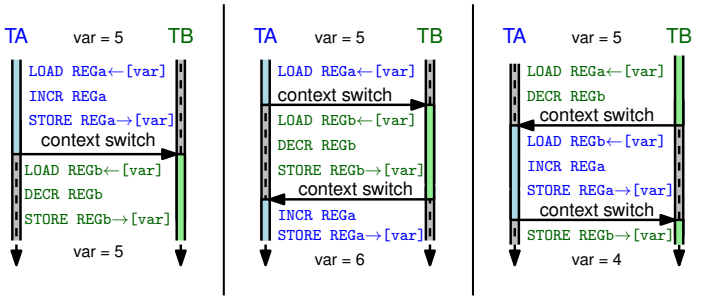
Thread B

```
...
...
LOAD   REGb←[var]
DECR   REGb
STORE  REGb→[var]
...
...
```

Remarque : A et B exécutés sur des (V)CPU distincts

- REGa et REGb (physiquement ou logiquement) distincts

Quelques exécutions possibles



Remarque : 1 CPU ou 2 CPU ▶ problème semblable

Notion de *«data race»*

VF «situation de concurrence», course critique, accès concurrents

Définition : *data race*

Situation où le résultat du programme dépend de l'ordre dans lequel sont exécutées les instructions des threads

Remarques

- plusieurs accès concurrents à une ressource partagée
  - variable globale, fichier, réseau, base de données...
  - écriture+écriture = problème
  - écriture+lecture = problème
- concurrence : parallélisme et/ou entrelacement
  - i.e. quand on ne maîtrise pas l'ordre temporel des actions
- risques : corruption de données et/ou crash
- mauvaise nouvelle : très difficile à débugger en pratique
- bonne nouvelle : des protections efficaces existent

Situation de concurrence : exemples

• deux écritures concurrentes = conflit

```
Thread A:   x=10
Thread B:   x=20
```

Question : valeur finale de x ?

• une lecture et une écriture concurrentes = conflit

```
Init:       x=5
Thread A:   x=10
Thread B:   print(x)
```

Question : valeur affichée ?

Précepte : *data race* = bug

Un programme dans lequel plusieurs tâches peuvent se retrouver en situation de concurrence est un programme incorrect.

Objectif : garantir l'exclusion mutuelle

Définitions

- Action atomique : action au cours de laquelle aucun état intermédiaire n'est visible depuis l'extérieur
- Ressource critique : objet partagé par plusieurs threads et susceptible de subir une data race
- Section critique : morceau de programme qui accède a une ressource critique

Idée : on veut que chaque section critique s'exécute de façon atomique

Définition : exclusion mutuelle

Interdiction pour plusieurs threads de se trouver simultanément à l'intérieur d'une section critique

Idée : «verrouiller» l'accès à une section critique déjà occupée

Exclusion mutuelle par verrouillage

Variables partagées

```
int var = 5;
lock_t L;
```

Thread A

```
{
  ...
  lock(L);
  var = var+1;
  unlock(L);
  ...
}
```

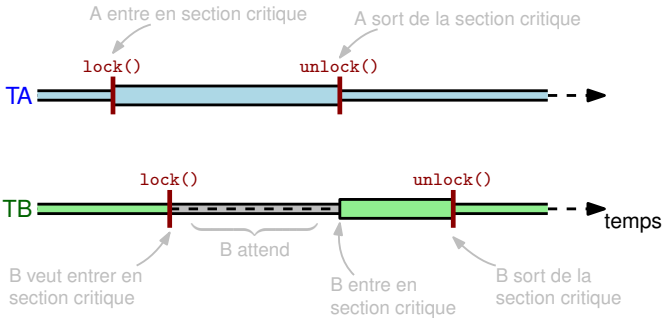
Thread B

```
{
  ...
  lock(L);
  var = var-1;
  unlock(L);
  ...
}
```

On voudrait ces deux méthodes atomiques :

- `lock(L)` pour prendre le verrou L en exclusivité
  - ► un seul thread peut entrer en section critique
- `unlock(L)` pour relâcher le verrou L
  - ► permet aux autres threads de le prendre à leur tour

Exclusion mutuelle : illustration



Solution naïve (et incorrecte)

partagé

```
int turn = 1;
```

Thread A

```
while(1)
{ ...
  while(turn==2)
  {/* attendre */ }
// section critique
  turn = 2;
  ...
}
```

Thread B

```
while(1)
{ ...
  while(turn==1)
  {/* attendre */ }
// section critique
  turn = 1;
  ...
}
```

- Exclusion mutuelle : OK
- Attente active : exécution pas très efficace
- Problème : alternance stricte ► progression non garantie

Problème : comment garantir l'exclusion mutuelle ?

Autrement dit : comment implémenter `lock()` et `unlock()` ?

Propriétés souhaitables

- Exclusion mutuelle : à chaque instant, au maximum une seule tâche est en section critique
  - sinon risque de *data race*
- Progression : si aucune tâche n'est en section critique, alors une tâche exécutant `lock()` ne doit pas se faire bloquer
  - sinon risque de *deadlock*, en VF interblocage
- Équité : aucune tâche ne doit être obligée d'attendre indéfiniment avant de pouvoir entrer en section critique
  - sinon risque de *starvation*, en VF famine, privation
- Généralité : pas d'hypothèses sur le nombre de tâches ou sur leurs vitesses relatives
  - on veut une solution universelle
- Bonus : implem simple, algo prouvable, exécution efficace...

Solution naïve n° 2 (incorrecte aussi)

partagé

```
bool occup = 0;
```

Thread A

```
while(1)
{ ...
  while(occup != 0)
  {/* rien */ }
  occup = 1;
// section critique
  occup = 0;
  ...
}
```

Thread B

```
while(1)
{ ...
  while(occup != 0)
  {/* rien */ }
  occup = 1;
// section critique
  occup = 0;
  ...
}
```

- Progression : OK
- Exclusion mutuelle : non garantie
- Problème : consultation-modification non atomique

Solutions correctes

Masquer les interruptions

- idée : empêcher tout changement de contexte
- ▶ dangereux, et inapplicable sur machine multiprocesseur

Approche purement logicielle

- idée : programmer avec des instructions atomiques
- autrefois seulement `LOAD` et `STORE` ▶ par ex. algo de Peterson
- de nos jours : `TEST-AND-SET`, `COMPARE-AND-SWAP` ▶ spin-lock
- ▶ attente active = souvent inefficace à l'exécution

Approche noyau : intégrer synchronisation et ordonnancement

- idée : programmer avec des instructions atomiques
  - mais les cacher dans le noyau (derrière des appels système)
- ▶ permet de bloquer / réveiller les threads au bon moment

Mutex : définition

Verrou exclusif, ou en VO mutex lock

- objet abstrait = opaque au programmeur
- deux états possibles : libre=unlocked ou pris=locked
- offre deux méthodes atomiques `lock()` et `unlock()`

`lock(L)`
si le verrou `L` est libre, le prendre
sinon, attendre qu'il se libère

`unlock(L)`
libérer le verrou `L`

Remarques

- `lock()` et `unlock()` implémentés comme appels système
- threads en attente = état `BLOCKED` dans l'ordonnanceur
  - une file de threads suspendus pour chaque mutex
- invoquer `unlock()` réveille *un* thread suspendu (s'il y en a)
  - attention : ordre de réveil non spécifié

API POSIX : Mutex locks

```
#include <pthread.h>

/* opaque typedefs */ pthread_mutex_t, pthread_mutexattr_t;

// create a new mutex lock
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *mutexattr);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

// attempt to lock a mutex without blocking
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```


Exclusion mutuelle : en résumé

Notion de «*data race*»

- plusieurs accès concurrents à une même variable
- accès non atomique ▶ données incohérentes

Section critique

- morceau de code qu'on veut rendre atomique
- ▶ exécution nécessairement en exclusion mutuelle

Solution : utiliser un mutex lock

```
lock(L);
/* section critique */
unlock(L);
```

- ▶ nécessite que tous nos threads jouent le jeu

Plan

Notion de «*deadlock*», en VF interblocage

Définition : interblocage, en VO deadlock

Situation dans laquelle deux (ou plusieurs) tâches concurrentes se retrouvent suspendues car elles s'attendent mutuellement

►...pour toujours

Exemple :

```
Init: Mutex X, Y
Thread A: lock(X); lock(Y); print("A"); ...
Thread B: lock(Y); lock(X); print("B"); ...
```

Exemple de trace d'exécution menant à un interblocage

1. Thread A : lock(X)
2. Thread B : lock(Y)
3. Thread A : lock(Y) ► suspend A
4. Thread B : lock(X) ► suspend B

Interblocage : conditions de Coffman

Quatre conditions nécessaires :

- exclusion mutuelle : chaque ressource critique est non partageable entre plusieurs threads
- *hold and wait* : un thread possède déjà une ressource et attend pour en avoir une autre
- non-préemption : une ressource verrouillée ne peut pas être réquisitionnée, seulement libérée volontairement
- attente circulaire : A attend que B libère une ressource, mais B attend que C libère une autre ressource, etc.

Solutions aux problèmes d'interblocages :

- Prévention : garantir leur absence, par conception
- Détection : reconnaître le problème quand il arrive
- Résolution : régler le blocage une fois qu'il est là

Plan

Scénario producteur-consommateur : introduction

Deux threads communiquent via une file FIFO partagée

Producteur

```
while(1)
{
  item=produce();
  fifo_put(item);
}
```

Consommateur

```
while(1)
{
  item = fifo_get();
  consume(item);
}
```

Remarques :

- file = tampon circulaire de taille constante
- producteur doit attendre tant que la file est pleine
- consommateur doit attendre tant que la file est vide

Prod.-consomm. : solution naïve (et incorrecte)

partagé

```
item_t buffer[N];
int count=0;
```

Producteur

```
int in = 0;
while(1)
{
  item=produce()
  while(count == N) {}
  buffer[in] = item;
  in = (in+1) % N;
  count = count + 1;
}
```

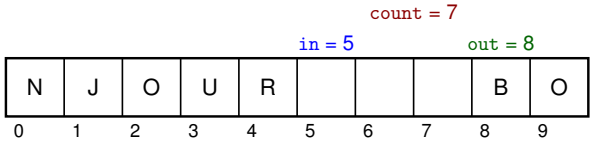
Consommateur

```
int out = 0;
while(1)
{
  while(count == 0) {}
  item = buffer[out];
  out = (out+1) % N;
  count = count - 1;
  consume(item);
}
```

Observation : ce programme a des bugs de synchronisation
► Question : comment corriger le problème ?

Producteur-consommateur : remarques

- `buffer` partagé de taille N (constante) initialement vide
  - buffer circulaire : `x%N` se lit « `x` modulo `N` »
- fonctions `produce()` et `consume()` non pertinentes
  - supposées «purement séquentielles» i.e. n'accédant à aucune ressource partagée
- variable partagée `count` pour la synchronisation
  - indique le nombre d'éléments actuellement dans le buffer
- variables `in` et `out` : non partagées
- exemple avec `N`=10 :



Tentative avec mutex 1 : deadlock

partagé

```
item_t buffer[N];
int count=0;
mutex L;
```

Producteur

```
int in = 0;
while(1)
{
  item=produce()
  lock(L);
  while(count == N) {}
  buffer[in] = item;
  in = (in+1) % N;
  count = count + 1;
  unlock(L);
}
```

Consommateur

```
int out = 0;
while(1)
{
  lock(L);
  while(count == 0) {}
  item = buffer[out];
  out = (out+1) % N;
  count = count - 1;
  unlock(L);
  consume(item);
}
```


Tentative avec mutex 2 : encore un deadlock

```
item_t buffer[N];
int count=0;
mutex L;
```

Producteur

```
int in = 0;
while(1)
{
  item=produce()
  lock(L);
  while(count == N) {}
  unlock(L);
  buffer[in] = item;
  in = (in+1) % N;
  lock(L);
  count = count + 1;
  unlock(L);
}
```

Consommateur

```
int out = 0;
while(1)
{
  lock(L);
  while(count == 0) {}
  unlock(L);
  item = buffer[out];
  out = (out+1) % N;
  lock(L);
  count = count - 1;
  unlock(L);
  consume(item);
}
```


Tentative avec mutex 3 : attente active

Producteur

```
int in = 0;
while(1)
{
  item=produce()
  lock(L);
  while(count == N) {
    unlock(L);
    lock(L);
  }
  unlock(L);
  buffer[in] = item;
  in = (in+1) % N;
  lock(L);
  count = count + 1;
  unlock(L);
}
```

Consommateur

```
int out = 0;
while(1)
{
  lock(L);
  while(count == 0) {
    unlock(L);
    lock(L);
  }
  unlock(L);
  item = buffer[out];
  out = (out+1) % N;
  lock(L);
  count = count - 1;
  unlock(L);
  consume(item);
}
```


Producteur-consommateur : à retenir

Hypothèses :

- file partagée de taille constante
- thread producteur doit attendre tant que la file est pleine
- thread consommateur doit attendre tant que la file est vide

Problèmes des solutions à base de mutex :

- mauvaise concurrence
- risques de deadlock
- attente active

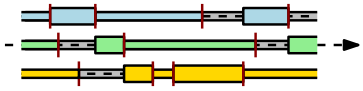
Mauvaise nouvelle

Ce scénario est insoluble avec seulement `lock()`/`unlock()`

▶ besoin d'un mécanisme spécifique pour *attendre* et *signaler* des évènements quelconques

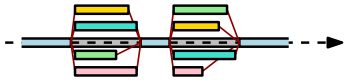
Quelques scénarios classiques de synchronisation

Exclusion mutuelle



Producteur consommateur, en VO bounded buffer problem

Boucle parallèle, en VO fork-join



Rendez-vous, en VO barrier

Notion de «variable de condition»

objet abstrait = opaque au programmeur

- contient une file d'attente de threads suspendus
- et une référence à un verrou mutex `L`
- offre trois méthodes atomiques : `wait()`, `signal()` et `broadcast()`

`wait(cond_var C, mutex L)`

1) atomiquement : déverrouiller `L`, et suspendre le thread courant
2) au réveil : re-verrouiller `L`, puis reprendre l'exécution

`signal(cond_var C)`

réveiller l'un des threads suspendus sur `C`

`broadcast(cond_var C)`

réveiller tous les threads suspendus sur `C`

Usage typique : la structure de «moniteur»

```
// section non critique
lock(L);
while( predicate == 0 )
    wait(C, L);

// section critique

signal(C);
unlock(L);
// section non critique
```

- on ne rentrera en section critique que si le prédicat est satisfait

Producteur-consommateur : solution avec CV

```
item_t buffer[N];
int count=0;
mutex L;
cond_var nonempty, nonfull;
```

Producteur

```
int in = 0;
while(1) {
  item=produce()
  lock(L);
  while(count == N)
     wait(nonfull, L);
  buffer[in] = item;
  in = (in+1) % N;
  count = count + 1;
  signal(nonempty);
  unlock(L);
}
```

Consommateur

```
int out = 0;
while(1) {
  lock(L);
  while(count == 0)
     wait(nonempty, L);
  item = buffer[out];
  out = (out+1) % N;
  count = count - 1;
  signal(nonfull);
  unlock(L);
  consume(item);
}
```


Variable de condition : remarques

- `wait(C, L)` relâche le verrou L
  - au moment de l'appel, L doit être verrouillé par le thread courant sinon c'est une erreur d'exécution
- `signal()` réveille *un* thread suspendu (s'il y en a)
  - attention : ordre de réveil non spécifié
- `signal()` utile quand les threads ont tous le même prédicat
  - ▶ pas la peine de réveiller tout le monde à chaque fois
- `broadcast()` utile quand les threads ont des prédicats distincts
  - ▶ pour être sûr de réveiller aussi le bon thread
- bonne pratique : *always hold the lock while signalling*
  - simplifie l'analyse des comportements possibles

Variables de condition : remarques, suite

- bonne pratique : *always re-check predicate on wake-up*

```
while( predicate==0 )
    wait(C, L);
```

vs

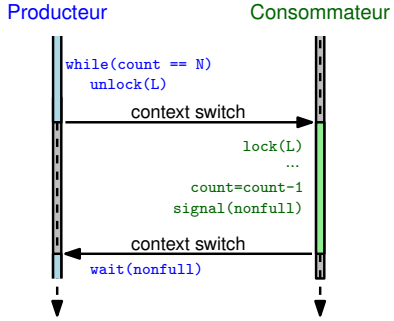
```
if( predicate==0 )
    wait(C, L);
```

  - ▶ protège contre les réveils intempestifs (*spurious wake-up*)
- Pourquoi la primitive `wait()` est-elle atomique ? vs deux notions distinctes de mutex et de CV, e.g.

```
lock(L);
while( predicate==0 ) {
   unlock(L);
   wait(C);
   lock(L);
}
```


Réponse

Découpler mutex et CV causerait des risques à l'exécution :



API POSIX : variables de condition

```
#include <pthread.h>

/* opaque typedefs */ pthread_cond_t, pthread_condattr_t;

// create a new cond variable
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr)

int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Implémenter des moniteurs dans d'autres langages

C++

- `#include <mutex>`
  - classe `std::mutex`
  - méthodes `lock()` et `unlock()`
- `#include <condition_variable>`
  - classe `std::condition_variable`
  - méthodes `wait()`, `notify_one()` et `notify_all()`

Java

- chaque `java.lang.Object` est un moniteur (mutex+CV)
- mot-clé `synchronized` pour délimiter les sections critiques
- méthodes `wait()`, `notify()`, `notifyAll()`

Plan

Threads et synchronisation : en résumé

Processus VS thread VS mémoire virtuelle

- unité d'ordonnancement = thread
- unité d'isolation mémoire = espace d'adressage virtuel
- un processus = un espace d'adressage + un/plusieurs threads

Exclusion mutuelle AKA mutex

- stratégie permettant d'éviter les *«data race conditions»*
- mécanisme : méthodes atomiques `lock()` et `unlock()`

Structure de moniteur

- mutex + variable de condition (pour la signalisation d'évènements)
- `wait(C,L)` pour attendre
- `signal(C)/broadcast(C)` pour réveiller