Threads (2) Synchronisation des processus concurrents : mutex

Introduction

Les threads partagent leur mémoire.

Le partage de mémoire est généralement voulu et avantageux :

- cela évite de gaspiller de la mémoire
- c'est un mécanisme de communication inter-thread (et inter-processus) très rapide

L'important est de bien savoir gérer l'accès concurrent à la mémoire

(Rappel : il faut faire attention aux fonctions/librairies non réentrantes, ou non "MT-safe")

Un exemple pour commencer...

Un thread peut être arrêté n'importe quand pour laisser sa place à un autre thread :

- y compris au milieu d'une ligne
- y compris au milieu d'une instruction basique en C (ex : `i++`)

Problème : si un thread A travaille sur une zone mémoire M, et est arrêté alors que M est inconsistante, le thread B trouvera M en état inconsistant.

- Comportement imprévisible ! (bug, plantage, exploitation, mauvaise note)

Un exemple pour commencer...

```
long long z=0;

void* th(void *r) {
  for(long long a=0;a<1000000;a++)
    z++;
}

int main() {
  pthread_t id1,id2;
  pthread_create(&id1,NULL,th,NULL);
  pthread_create(&id2,NULL,th,NULL);
  pthread_join(id1,NULL);
  pthread_join(id2,NULL);
  printf("%Ld\n",z);
}
```

sortie : 1020102 ou 1271305 ou 948249...

Atomicité

Code assembleur de z++ :

```
movq    z(%rip), %rax
addq    $1, %rax
movq    %rax, z(%rip)
```

On aimerait que ces 3 instructions ne puissent être interrompues.

Instruction(s) atomique : instruction(s) ne pouvant être interrompues.

Pour que l'exemple précédent soit correct, il faudrait rentre l'instruction z++ atomique.

Atomicité d'une instruction

L'atomicité peut se faire au niveau du processeur.

Il faut distinguer 2 choses :

- Atomicité vis à vis d'une interruption
  Une instruction processeur est atomique vis à vis d'une interruption.

- Atomicité vis à vis des autres coeurs
  Dans les ordinateurs multiprocesseurs et/ou multicoeurs, du fait de la pipeline, une variable peut changer entre sa lecture et son écriture. Elle n'est donc pas (par défaut) atomique vis à vis des autres coeurs .
  Sur x86 : on peut rendre atomique une instruction machine avec le préfixe `lock`.

Atomicité d'une instruction : exemple

Pour que le programme précédent fonctionne comme souhaité

- On peut incrémenter `z` avec l'instruction assembleur `incq`
  $\Rightarrow$ l'opération sera atomique vis a vis des interruptions.
  $\Rightarrow$ le programme fonctionnera correctement si la machine a un unique coeur.

- Pour que le programme soit "correct" dans le cas général :
  Il faut rendre atomique l'instruction : `lock incq`
  $\Rightarrow$ le programme fonctionne comme souhaité.

  Problème : Du fait que la pipeline ne sert presque plus, c'est beaucoup plus lent...

Atomicité d'un ensemble d'instructions

En pratique, les opérations sur une zone mémoire prennent généralement plusieurs instructions

Problèmes :

- Il n'est pas possible (ni raisonnable) de bloquer les interruptions / les autres coeurs dans le mode utilisateur.
- Pourquoi ? Un processus malveillant/planté/bogué pourrait bloquer tous les autres processus...

- Il n'est pas possible d'utiliser un mécanisme du type `lock` sur un ensemble d'instructions

Atomicité d'un ensemble d'instructions

De toutes façons : on ne veut pas l'atomicité d'un ensemble d'instructions...
Cela bloquerait tous les coeurs pour accéder à une zone mémoire, alors que les autres ne travaillent pas forcement sur cette zone...

La bonne solution n'est pas d'avoir des sections atomiques, mais des sections où on a l'exclusivité sur une partie de la mémoire.

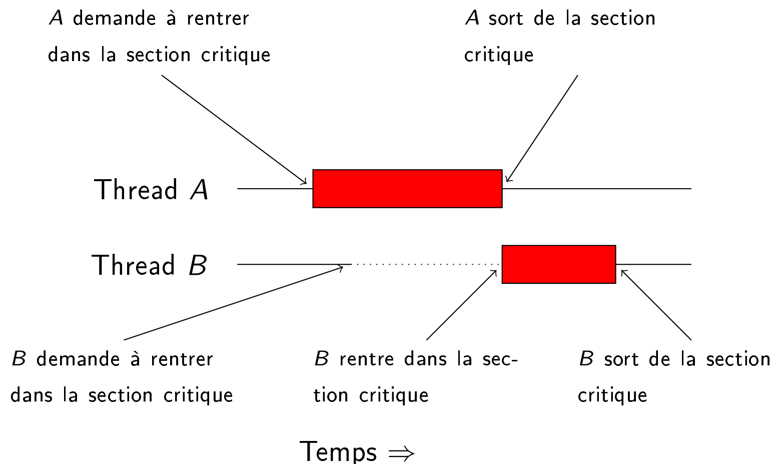
Cela s'appelle une section critique .

Sections critiques

Une section critique est une section du code où il n'y a au maximum qu'un thread à la fois

Si un thread B veut rentrer dans une section critique, et qu'un autre thread A est déjà dans la section critique, on doit faire attendre le thread B jusqu'à ce que le thread A sorte de la section critique.

Sections critiques



Sections critiques

Avantage :

- on ne bloque pas les autres threads qui ne travaillent pas sur la section critique
- il peut y avoir plusieurs sections critiques différentes, qui n'interfèrent pas entre elles.

Sections critiques

Il faut un mécanisme pour entrer et sortir des sections critiques.

Ce qu'on attend d'un mécanisme de gestion des sections critiques :

- l'exclusion mutuelle : deux threads ne sont pas en même temps dans la section critique
- la progression : le processus continue de progresser dans tous les cas possibles d'exécution
- l'attente bornée : un thread qui demande à rentrer dans une section critique ne va pas attendre indéfiniment

Sections critiques

Ce n'est pas un problème trivial. Plusieurs solutions fausses ont été publiées

Les problèmes en cas de mauvaise gestion des sections critiques :

- Si l'exclusion mutuelle est pas respectée :
  Situation de compétition (race condition) : deux threads sont dans une section critique en même temps : non déterminisme (bug, plantage, exploitation...)
- Si la progression n'est pas respectée :
  Interblocage (deadlock) : le processus est bloqué.
- Si l'attente bornée n'est pas respectée :
  Famine (starvation) : un thread ne voit jamais sa demande aboutir

Exemple simple : 2 threads

Solution 1 (?)

```
int intA=0,intB=0;
```

Thread A :

```
while(1) {
 while(intB) /*wait*/;
 intA=1;
 //sect. critique
 intA=0;
 //sect. normale
}
```

Thread B :

```
while(1) {
 while(intA) /*wait*/;
 intB=1;
 //sect. critique
 intB=0;
 //sect. normale
}
```

Correct ?

Non ! Les 2 threads peuvent être dans la section critique en même temps (situation de compétition)

Exemple simple : 2 threads

Solution 2 (?)

```
int intA=0,intB=0;
```

Thread A :

```
while(1) {
 intA=1;
 while(intB) /*wait*/;
 //sect. critique
 intA=0;
 //sect. normale
}
```

Thread B :

```
while(1) {
 intB=1;
 while(intA) /*wait*/;
 //sect. critique
 intB=0;
 //sect. normale
}
```

Correct ?

Non ! Les 2 threads peuvent se bloquer mutuellement (interblocage)

Exemple simple : 2 threads

Solution 3 (?)

```
int rnd=0;
```

Thread A :

```
while(1) {
 while(rnd!=0) /*wait*/;
 //sect. critique
 rnd=1;
 //sect. normale
}
```

Thread B :

```
while(1) {
 while(rnd!=1) /*wait*/;
 //sect. critique
 rnd=0;
 //sect. normale
}
```

Correct ?

Non ! Quand le thread A termine, B est bloqué indéfiniment (famine)

Exemple simple : 2 threads

Solution 4 (?)

```
int rnd=0;
int intA=0,intB=0;
```

Thread A :

```
while(1) {
 intA=1;
 rnd=1;
 while(intB && rnd==1)
     /*wait*/;
 //sect. critique
 intA=0;
 //sect. normale
}
```

Thread B :

```
while(1) {
 intB=1;
 rnd=0;
 while(intA && rnd==0)
     /*wait*/;
 //sect. critique
 intB=0;
 //sect. normale
}
```

Correct ?

Oui ! (Solution de Peterson)

Attente active et passive

L'attente avant d'entrer dans une section critique peut être :

- active (*spinlock*) : le thread continue de tourner jusqu'à ce qu'il a le droit d'entrer dans la section critique
  Dans l'exemple précédent (Peterson), l'attente est active
- passive : le thread est mis en pause jusqu'à ce qu'il a la possibilité de rentrer dans la section critique
  Dans ce cas, il faut interférer avec l'ordonnanceur
  Avantage : on laisse le temps CPU aux autres threads qui peuvent faire des choses plus constructives

En général, on préfère l'attente passive. (Mais dans certains cas très particuliers, l'attente active peut être plus avantageuse.)

Les primitives

En général, on ne reprogramme pas soi même les tests d'entrée dans une section critique.

- C'est fastidieux
- le risque d'erreur est très important
- on ne tire pas parti des possibilités de l'OS.

On passe par des primitives (du langage, de librairies et/ou du système) : des verrous .

Les primitives

Il existe différents types de verrous :

- sémaphores (POSIX 1.b)
- mutex (pthreads)
- rwlocks (pthreads)
- barrières (pthreads)
- variables de condition / moniteurs (pthreads)

Attention ! Les verrous proposés par les langages/systèmes ne sont que des primitives (pour simplifier la vie).

Une mauvaise utilisation peut toujours mener à des problèmes : situation de competition, interblocage ou famine...

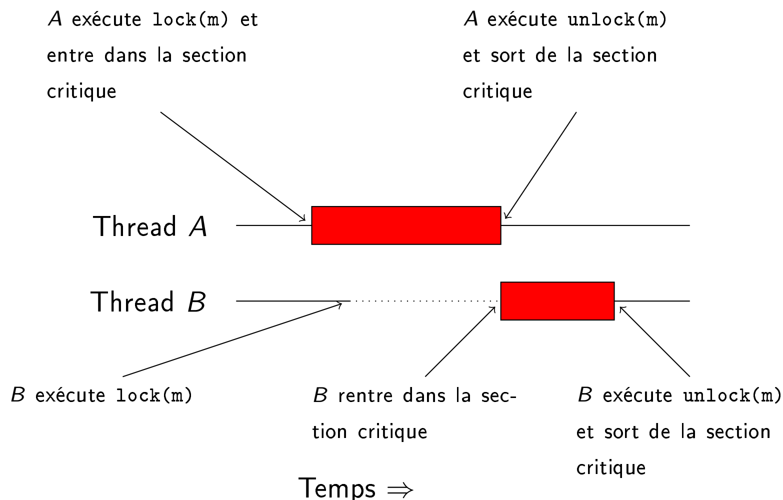
Verrou le plus simple : le mutex

Mutex : assure qu'au plus un thread est dans la section critique à un moment donné.

Pseudo-code :

```
mutex m;

//section non critique
lock(m);
//section critique;
unlock(m);
//section non critique
..
```

Verrou le plus simple : le mutex



Les mutex de pthreads

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_init(pthread_mutex_t *mutex, const
    pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

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

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Exemple :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mutex);
//section critique
pthread_mutex_unlock(&mutex);
```

Les mutex de pthreads

Note :

- Deux façons d'initialiser un mutex :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

ou

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex,NULL);
```

- `mutexattr` permet d'initialiser un mutex avec d'autres attributs que ceux par défaut (récursif, partagé...). NULL = défaut
- `trylock` essaye de bloquer le mutex. Si il échoue, il renvoie directement la main avec une erreur (retour $\neq 0$)

Implémentation d'un mutex (?)

Ici, un mutex est un entier.

```
int a=1;

lock(int &a) {
 while(a==0) /*wait*/;
 a=0;
}

unlock(int &a) {
 a=1;
}
```

Problème : pour que ce soit correct, le test et l'affectation doivent se faire atomiquement

Implémentation possible d'un mutex

```
int a=0;

lock(int &a) {
 int tmp=0;
 while(1) {
  xchg(a,tmp);
  if(tmp==1) break;
 }
}

unlock(int &a) {
 a=1;
}
```

`xchg(a,b)` échange (atomiquement) a et b. (instruction x86)

- Problème : attente active et possible famine

Pseudo-implémentation idéale d'un mutex

```
int libre=1;
queue liste_attente;

lock() { //atomiquement
 while(1) {
  if(libre) {
   libre=0;
   return;
  }
  liste_attente.push(thread_courant());
  wait();
 }
}

unlock() { //atomiquement
 if(liste_attente.non_vide())
   signal(liste_attente.pop());
 else
   libre=1;
}
```

Où sont implémentés les verrous ?

Problème des verrous en mode utilisateur :

- pour éviter l'attente active, il faut jouer avec l'ordonnanceur
- pour éviter les famines, il faut une file d'attente

Ces taches sont souvent laissées aux OS

Problème des verrous en mode noyau : les appels systèmes sont très lents !

Bon compromis : combiner les deux

Implémentation dans NPTL

- `lock()` sur un verrou libre : opération atomique
- `lock()` sur un verrou non libre : opération atomique + appel système (`futex()`) qui met le thread en pause, et rajoute à une liste d'attente
- `unlock()` : opération atomique + (si la liste d'attente est non vide) appel système à `futex` pour libérer le thread suivant.

- coté utilisateur : un boléen (libre ou non) et le nombre de threads en attente
- coté système : une liste d'attente

Mutex récursif

NPTL (et d'autres implémetations) introduisent d'autres possibilités (non POSIX, "non portables"). Par exemple :

- mutex récursif : l'action de bloquer un mutex déjà bloqué par le thread courant ne bloque pas. Exemple :

```
foo(int i)
{
  lock(mutex);
  // section critique
  if(i>0) foo(i-1);
  // section critique
  unlock(mutex);
}
```

Read-write locks

On pourrait permettre à plusieurs threads qui ne modifient pas la mémoire, de travailler (en lecture seule) sur une zone mémoire.

Une solution : read-write locks (rwlocks)

Deux types de sections critiques

- les sections critiques en lecture seule (celles des lecteurs)
- les sections critiques en lecture/écriture (celles des écrivains)

Read-write locks

Garantie des rwlocks :

- si un thread est dans une section critique en lecture/écriture, il n'y a aucun autre thread dans une section critique (ni en lecture seule, ni en lecture/écriture)
- (si aucun thread est dans une section critique en lecture/écriture, il n'y a pas de limite sur le nombre de threads dans une section critique en lecture seule)

Attention : on peut facilement arriver à des famines

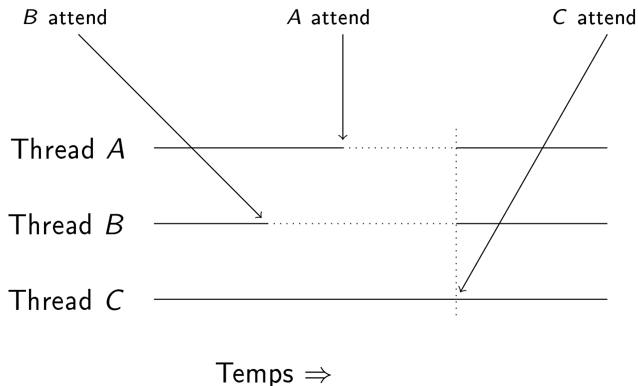
- préférer les lecteurs : il peut y avoir famine des écrivains
- préférer les écrivains : il peut y avoir famine des lecteurs

Les rwlocks de pthreads

```
#include <pthread.h>

pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;

int pthread_rwlock_init(pthread_rwlock_t * restrict
    lock, const pthread_rwlockattr_t * restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *lock);

int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
int pthread_rwlock_timedrdlock(pthread_rwlock_t *
    restrict lock, const struct timespec * restrict
    abstime);

int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *
    restrict lock, const struct timespec * restrict
    abstime);

int pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

Les barrières

Les barrières permettent de synchroniser les threads



Les barrières POSIX

```
int pthread_barrier_init(pthread_barrier_t *restrict
    barrier, const pthread_barrierattr_t *restrict attr
    , unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier)
    ;
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- count : nombre de threads qui doivent attendre à la barrière

Exemple :

```
pthread_barrier_t barrier;
pthread_barrier_init(&barrier,NULL,3);
```

Puis dans chaque thread (A, B et C) :

```
// section non synchronisee
pthread_barrier_wait(&barrier);
// section synchronisee
```

Problèmes : vitesse

Le but des programmes parallèles est de gagner en vitesse.

En utilisant les mécanismes précédents (mutex...) un processus peut perdre du temps :

- dans les attentes qu'un verrou se libère (attente active ou passive)
- dans les instructions atomiques (plus lentes à cause de problèmes de cache)
- dans les appels systèmes relatifs aux (dé)blocage de verrous

Problèmes : vitesse

Solutions :

- limiter la taille des sections critiques
- séparer les zones mémoires partagés
  (idéalement : 1 zone mémoire = 1 verrou)
- mais sans faire trop d'entrées/sorties de sections critiques

Un problème d'échelle (de granularité) peut parfois se poser

- trouver le bon compromis

Problèmes : vitesse

Exemple (bidon)

$$\sigma(n) = \sum_{1 \le i \le n : i | n} i$$

```
long long n,s=0;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;

void* th(void *r) {
  for(long long i=(long long)r; i<=n; i+=2)
    if(i%n==0) {
      pthread_mutex_lock(&m);
      s+=i;
      pthread_mutex_unlock(&m);
    }
  return NULL;
}
```

Problèmes : vitesse

Mieux :

```
long long n,s=0;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;

void* th(void *r) {
  long long tmp=0;
  for(long long i=(long long)r;i<=n;i+=2)
      if (i%n==0)
        tmp+=i;
  pthread_mutex_lock(&m);
  s+=tmp;
  pthread_mutex_unlock(&m);
  return NULL;
}
```

- plus d'opérations de calcul (ici, 2 additions en plus)
- mais beaucoup moins de sections critiques

Gros problèmes

En cas de mauvaise gestion des sections critiques :

- situation de compétition : bugs, plantages, morts (Therac-25)
- interblocage

Si on protège chaque section critique par un verrou adéquat, l'exclusion mutuelle devrait être respectée. Le problème sera généralement l'interblocage

Suite :

- Les 5 philosophes...
- Sémaphores
- Variables de condition (Moniteurs)
- Gérer les interblocages
- Concurrence en C++11