Threads (3) : Sémaphores et variables de condition

Introduction

- Les threads partagent leur mémoire
- Il faut protéger les accès mémoires concurrents
- Les mutex sont des verrous qui permettent de délimiter des sections critiques
- Mais des fois, les mutex ne suffisent pas...

On va voir :

- Les sémaphores
- Les variables de conditions

Problème classique : 5 philosophes

- 5 philosophes sont autour d'une table ronde
- Il y a une baguette entre chaque philosophe (i.e. : une baguette pour deux philosophes)
- Un plat de sushis pour tout le monde est au centre
- La vie d'un philosophe se résume à deux actions : penser et manger
- Pour manger, il doit prendre les 2 baguettes (à sa gauche et à sa droite), puis il peut commencer à manger
- Quand il a fini, il repose les baguettes et peut commencer à penser

- Chaque baguette : une ressource (un mutex)

But :

- Tout le monde doit pouvoir manger (pas de famine)
- Limiter les attentes

Problème classique : 5 philosophes

```
mutex baguette[5];

void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);
    lock(&baguette[i]);
    lock(&baguette[(i+1)%5]);
    printf("%d mange\n",i);
    unlock(&baguette[i]);
    unlock(&baguette[(i+1)%5]);
  }
}
```

Correct ?

Non : interblocage. Si tout le monde a la baguette à gauche, tout le monde est bloqué

5 philosophes : solution ?

```
mutex baguette[5],general;

void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d␣pense\n",i);
    lock(&general);
    lock(&baguette[i]);
    lock(&baguette[(i+1)%5]);
    printf("%d␣mange\n",i);
    unlock(&baguette[i]);
    unlock(&baguette[(i+1)%5]);
    unlock(&general);
  }
}
```

Correct... mais qu'un seul philosophe mange à la fois. Les mutex baguette[i] ne servent à rien $\rightarrow$ une seule section critique

5 philosophes : solution ?

```
mutex baguette[5],general;

void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d␣pense\n",i);
    lock(&general);
    lock(&baguette[i]);
    lock(&baguette[(i+1)%5]);
    unlock(&general);
    printf("%d␣mange\n",i);
    unlock(&baguette[i]);
    unlock(&baguette[(i+1)%5]);
  }
}
```

Mieux... mais un philosophe doit des fois attendre inutilement pour manger.

5 philosophes : solution ?

```
void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);
    while(1) {
      lock(&general);
      if(baguette[i]==1 && baguette[(i+1)%5]==1) {
        baguette[i]=baguette[(i+1)%5]=0;
        unlock(&general);
        break;
      }
      unlock(&general);
    }
    printf("%d mange\n",i);
    baguette[i]=baguette[(i+1)%5]=1;
  }
}
```

Attente active et famine...

Parenthèse : Famine ?

On peut distinguer deux types de famines :

- la famine "avérée" : un thread est indéfiniment bloqué (quelle que soit le déroulement de la suite)
- la famine "probabiliste" : il y a une probabilité non nulle qu'un thread reste bloqué indéfiniment longtemps.

Si on s'autorise la famine "avérée", il existe une solution simple aux 5 philosophes sans interblocage, ni situation de compétition :

- On désigne un philosophe qui n'a pas le droit de prendre de baguettes (donc ni de manger et penser)

Parenthèse : Famine ?

Si on s'autorise la famine "probabiliste" (mais pas "avérée"), la solution précédente est bonne (modulo le fait qu'elle soit en attente active)

Note :

- La famine est, des fois, pas très grave (e.g. si les threads font le même travail).
- Beaucoup considèrent que la famine probabiliste n'est pas un vrai problème.

Ordre d'importance :
famine "probabiliste" / famine "avérée" / interblocage / situation de compétition

5 philosophes : solution ?

Plusieurs (idées) de solutions

- Contre l'attente active : rajouter des temporisations aléatoires
  - bricolage : pas optimal (et toujours possible famine)
- Prendre une baguette (lock), essayer de prendre l'autre (trylock). Si la deuxième n'est pas disponible, reposer la première (unlock) et recommencer.
  - attente active (et possible famine)
- Prendre une baguette (lock), essayer de prendre l'autre (trylock). Si la deuxième n'est pas disponible, reposer la première (unlock) et recommencer dans le sens contraire .
  - (possible famine)

5 philosophes : solution ?

Note : si les philosophes sont sur une table linéaire (5 philosophes, 6 baguettes), la solution triviale marche.

Ce qui pose problème : les cycles

Solution : casser les cycles

5 philosophes : solution ?

```
void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);
    if(i==0) {
     lock(&baguette[(i+1)%5]);
     lock(&baguette[i]);
    } else {
     lock(&baguette[i]);
     lock(&baguette[(i+1)%5]);
    }
    printf("%d mange\n",i);
    unlock(&baguette[i]);
    unlock(&baguette[(i+1)%5]);
  }
}
```

Correct. Mais pas très joli, et l'asymétrie introduit des biais (des philosophes attendent plus que d'autres en moyenne)

5 philosophes : Solution de Dijkstra

On limite à 4 le nombre de philosophes qui peuvent rentrer dans la phase "prendre des baguettes"

- au plus 4 arcs dans le graphe $\Rightarrow$ pas de cycle.

Principe des sémaphores .

(Exercice : est il possible que le 5ème philosophe attende inutilement ?)

(Exercice : combien, au minimum, faut-il autoriser de philosophes dans la phase "prendre des baguettes" pour qu'il n'y ait pas d'attente inutile ?)

Les sémaphores

Un autre type de verrou est le sémaphore

- Introduit par Dijkstra ($\sim$ 1963)
- Premier verrou à apparaître dans un vrai système
- Plus général qu'un mutex
- (Mais il a plutôt un intérêt historique et, ici, didactique)

- 3 opérations :
  - init(N) : initialise le sémaphore à N "ressources"
  - P() (ou wait(), ou down()) : demande une ressource. Si il n'y a plus de ressource libre, attend jusqu'à ce qu'une ressource se libère
  - V() (ou signal(), ou post(), ou up()) : libère une ressource

- garantie : au plus N threads possèdent une "ressource"

Sémaphore : différence avec un mutex

- Un sémaphore avec N=1 simule (un peu près) un mutex
- (Un sémaphore avec N=1 est un sémaphore binaire)
- Mais : le mutex est un booléen, le sémaphore est un entier
  - unlock();unlock(); n'aura pas le même comportement que V();V();
- Mais : un mutex doit être débloqué par le thread qui l'a bloqué.
  - On peut se servir d'un sémaphore comme d'un "signal" pour débloquer un autre thread

```
init(sem,0);
```

Thread A :

```
sleep(2);
// pas syncro
V(sem);
//synchro
```

Thread B :

```
sleep(1);
// pas syncro
P(sem);
//synchro
```

Sémaphores POSIX

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned
    int value);
int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct
    timespec *abs_timeout);

int sem_post(sem_t *sem);
```

- `pshared` = 0 : le sémaphore n'est pas partagé avec un autre processus (uniquement entre les threads de ce processus)
- `pshared` = 1 : le sémaphore est partagé. `sem` doit être dans une zone mémoire partagé entre les différents processus

5 philosophes avec sémaphore

```
mutex baguette[5];
semaphore sem;
init(sem,4);

void *philosophe(void *a) {
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);
    wait(sem);
    lock(baguette[i]);
    lock(baguette[(i+1)%5]);
    post(sem);
    printf("%d mange\n",i);
    unlock(baguette[i]);
    unlock(baguette[(i+1)%5]);
  }
}
```

Parfait ! pas de famine (même "probabiliste"), pas d'attente inutile.

Sémaphore : Implémentation ?

La première solution (Dijkstra) utilisait des blocages d'interruptions. Cela n'est plus valable sur des machines multiprocesseurs, mais cela peut être simulé avec un mutex

```
init(int &s,int N) {
  lock(mutex);
  s=N;
  unlock(mutex);
}

V(int &s) {
  lock(mutex);
  s++;
  unlock(mutex);
}
```

```
P(int &s) {
  while(1) {
    lock(mutex);
    if(s>0) {
      s--;
      unlock(mutex);
      return;
    }
    unlock(mutex);
  }
}
```

- Attente active et famine

Sémaphore : Implémentation ?

Comment implémenter un sémaphore avec une attente passive ?

C'est les mêmes problèmes qu'on avait déjà vu avec les mutex :

- il faut une file (éviter la famine)
- il faut communiquer avec l'ordonnanceur (pour éviter l'attente active)

(Les sémaphores sont disponibles dans POSIX, mais imaginons que ce ne soit pas le cas.)

Comment peut on faire pour les reprogrammer correctement ?

On ne peut pas simuler un sémaphore avec des mutex seuls

Mais on peut le faire avec un mutex + une variable de condition

Variable de condition

Plus généralement, supposons qu'on veut qu'un thread bloque jusqu'à ce qu'une condition (qui peut être compliquée) soit vérifiée

```
//...
while(1) {
   lock();
   if(condition()==1) {
      // operations d'entree de section critique
      unlock();
      break;
   }
   unlock();
}
//section critique
lock();
// operations de sortie de section critique
unlock();
//...
```

On voudrait avoir le même comportement que le code ci-dessus, mais sans les famines et dans l'attente active. Comment faire ?

Variable de condition

Variable de condition : primitive qui permet de mettre en attente (dans une queue) un thread en débloquant (atomiquement) un mutex

```
mutex m;
cond c;

//...
lock(m);
while(condition()==0)
   wait(c,m);
// operations d'entree de section critique
unlock(m);
//section critique
lock(m);
// operations de sortie de section critique
signal(c);
unlock(m);
//...
```

Variable de condition

Une variable de condition fonctionne toujours de pair avec un mutex

3 primitives :

- wait(c,m) : (atomiquement) débloquer m, mettre le thread en pause, et le rajouter dans la queue de c
  - au moment de l'appel, m doit être bloqué !
- signal(c) : débloque le premier thread de la queue de c
- broadcast(c) : débloque tous les threads de la queue de c

signal ou broadcast ?

Si tous les threads en attente attendent sur la même condition : signal()

Si les threads ont des conditions différentes : broadcast()

Sinon : un thread est débloqué, mais si sa condition n'est pas validée, il va devoir re-entrer en sommeil. Si un autre thread a sa condition validée, il ne sera pas réveillé par défaut.

Variables de condition dans pthread

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init(pthread_cond_t *cond,
    pthread_condattr_t *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex, const struct timespec *
    abstime);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

5 philosophes avec mutex+cond

```
void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);

    pthread_mutex_lock(&mutex);
    while(baguette[i]==0 || baguette[(i+1)%5]==0)
      pthread_cond_wait(&cond,&mutex);
    baguette[i]=baguette[(i+1)%5]=0;
    pthread_mutex_unlock(&mutex);

    printf("%d mange\n",i);

    pthread_mutex_lock(&mutex);
    baguette[i]=baguette[(i+1)%5]=1;
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);
  }
}
```

Sémaphores avec mutex+cond

```
typedef struct {
  int n;
  pthread_mutex_t m;
  pthread_cond_t c;
} sem_t;

void sem_init(sem_t *s,
              int n)
{
  s->n=n;
  pthread_mutex_init(
        &s->m,NULL);
  pthread_cond_init(
        &s->c,NULL);
}
```

```
void sem_post(sem_t *s)
{
 pthread_mutex_lock(&s->m);
 s->n++;
 pthread_cond_signal(&s->c);
 pthread_mutex_unlock(&s->m);
}

void sem_wait(sem_t *s)
{
 pthread_mutex_lock(&s->m);
 while(s->n<=0)
  pthread_cond_wait(&s->c,
                    &s->m);
 s->n--;
 pthread_mutex_unlock(&s->m);
}
```

Pour finir...

- Moniteurs = mutex + variable de condition associée au mutex
- Généralise les autres primitives
- Par exemple, en C++11 : les seules primitives introduites dans la STD sont `<mutex>` et `<condition_variable>`

Interlude : Concurrence C++11

Le C++11 introduit plusieurs classes pour gérer les threads et la concurrence :

- threads
- mutex
- variables de condition

<thread>

- `std::thread` représente un thread (similaire à `pthread_t`)
- le thread est lancé à la construction :

```
void fct(int a, double b) {
  //...
}

std::thread th(fct,42,3.14);
```

- grâce à la "magie" des templates, pas besoin de jouer avec un argument `void*`

<thread>

- `std::thread` est remplaçable (mais pas copiable). `operator=` déplace le thread.

```
std::thread th[42];

for(int i=0;i<42;i++)
  th[i]=std::thread(foo,i);
```

- Fonctions membres : `join()`, `detach()`, `swap()`
- si on détruit un std : :thread alors qu'il correspond à un thread en exécution, non détaché : exception
- pas de code retour (voir <future>)

<mutex>

- `std::mutex` et `std::recursive_mutex`
- Fonctions membres : `lock()`, `try_lock`, `unlock`
- `std::lock_guard` est un conteneur pour un mutex (il le bloque à sa construction, et le débloque à sa destruction)

```
std::mutex m;
//..
{
 std::lock_guard<std::mutex> l(m);
 //section critique
}
// section normale
```

- particulièrement utile pour que le mutex soit débloqué automatiquement en cas d'exception

<mutex>

- `std::unique_lock` est un conteneur plus évolué.
- Il a notamment les mêmes fonctions membres qu'un mutex, ce qui permet de s'en servir comme un mutex (avec l'assurance qu'il sera débloqué en cas de destruction)

```
std::mutex m;
std::uniquelock<std::mutex> l(m, std::defer_lock);

l.lock();
//section critique
l.unlock();
```

`std::lock()`

- `std::lock(m1,m2,...)` permet de bloquer plusieurs mutex à la fois
- Algorithme :
  - bloque le premier mutex m_1,
  - puis essaye de bloquer les autres avec `try_lock()`.
  - Si un mutex m_i, $i > 1$ échoue, il débloque tous les autres : $m_1, \ldots m_{i-1}$,
  - puis recommence en commençant par m_i.
- Sans deadlock, et sans attente active.

5 philosophes en C++11

```
#include <thread>
#include <mutex>
std::mutex baguette[5];

void philosophe(int i) {
  while(1) {
    printf("%d pense\n",i);
    lock(baguette[i],baguette[(i+1)%5]);
    printf("%d mange\n",i);
    baguette[i].unlock();
    baguette[(i+1)%5].unlock();
  }
}

int main() {
  std::thread id[5];
  for(int i=0;i<4;i++)
    id[i]=std::thread(philosophe,i);
  id[0].join();
}
```

- Famine ? (voir l'algo de `lock()`...)

<condition_variable>

- `std::condition_variable` est une variable de condition
- constructeur sans argument

Fonctions membres :

- `wait(l)` (`l` doit être un `unique_lock<mutex>`)
- `notify_one()` = signal
- `notify_all()` = broadcast

<future>

Permet d'accéder au résultat de procédures asynchrones

```
int carre(int x) {
  for(long long i=0;i<1000000000;i++);
  return x*x;
}

int main()
{
  std::future<int> res=std::async(carre,42);
  printf("resultat=%d\n",res.get());
  return 0;
}
```

<future>

Ou bien avec des promesses :

```
void carre(int x,std::promise<int> pr) {
  for(long long i=0;i<1000000000;i++);
  pr.set_value(x*x);
}

int main()
{
  std::promise<int> pr;
  std::future<int> res=pr.get_future();
  std::thread th(carre,42,std::move(pr));
  printf("resultat=%d\n",res.get());
  th.join();
  return 0;
}
```

<atomic>

Permet de faire des opérations atomiques sur une variable

```
std::atomic<int> atint;

void th() {
  for(int i=0;i<10000000;i++)
    atint++;
}

int main() {
  atint.store(0);
  std::thread th1(th);
  std::thread th2(th);
  th1.join();
  th2.join();
  printf("%d\n",atint.load());
  return 0;
}
```

Variables `thread_local`

En C/C++ : il y a deux types de variables :

- les automatiques (ou locales) : dans la pile (défaut, mot clef `auto`)
- les statiques (ou globales) : dans le segment de données (mot clef `static`)

Le C++11 rajoute le type `thread_local`

- la variable sera locale au thread