Synchronisation

Sections critiques

Mutex et Verrous

Sémaphores

Problèmes classiques

Moniteurs

Atomicité

Mise en contexte

Les processus et threads peuvent exécuter concurremment

Les accès concurrents aux données partagées peuvent être incohérents

Pour garantir la cohérence, il faut des mécanismes de synchronisation.

Producteur-consommateur, le retour

```
while (counter == BUFFER_SIZE) /*Wait*/;
buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
counter++;
```

```
while (counter == 0) /*Wait*/;
next_consumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
```

Comment faire `counter++` et `counter--` en même temps?

Condition de course

Exécution simultanée de `counter++` et `counter--`:

T1	T2
`r1 = read counter`	`r2 = read counter`
`r1 = r1 + 1`	`r2 = r2 - 1`
`store r1 -> counter`	`store r2 -> counter`

Résultat dépend de l'ordre d'exécution: *condition de course*

Problème **très courant** mais **rare**!

Difficile à reproduire, tester, ou même détecter

Section critique

Une *section critique* est une séquence d'instructions qui accède à des données partagées et a donc besoin de synchronisation.

Pendant qu'un thread est dans une section critique, aucun autre thread ne devrait accéder au même données.

Le *problème de la section critique* est de trouver une protocole:

- Comment *entrer* dans la section critique
- Comment *sortir* de la section critique

On veut une propriété d'*atomicité*!

Structure d'une section critique

On veut donc d'un côté écrire quelque chose de la forme:

```
...do something local...;

BEGIN-critical-section
  ...do something global...;
END-critical-section

...do more local things...;
```

Et de l'autre définir ces `BEGIN/END`

Propriétés d'une section critique

Une section critique devrait avoir les propriétés suivantes:

- *Exclusion mutuelle*: si un thread est dans une section critique, aucun autre thread ne peut interférer.
- *Progrès*: Si aucun thread n'est dans une section critique, et qu'il y a des threads qui veulent entrer dans une section critique, Tôt ou tard (*eventually*) l'un d'entre eux devrait pouvoir entrer.
- *Attente limitée*: si un thread veut entrer dans une section critique, il devrait y a une limite au nombre de fois que d'autres threads peuvent entrer dans leurs sections critiques avant qu'il ne puisse entrer dans la sienne.

Solution de Peterson

```
flag[myself] = true;
turn = other;
while (flag[other] && turn == other)
    /*Wait*/;

...do something global...;

flag[myself] = false;

...do something local...;
```

Propriétes de la solution de Peterson

The good: *exclusion mutuelle*, *progrès*, et *attente limitée*

The bad:

- Gère seulement le cas de 2 threads.
- Ne distingue pas différentes données partagées.

The ugly:

- Présume que les instructions `load` et `store` sont *atomiques*.
- Présume des propriétés du *modèle de mémoire* pas garanties par certaines machines modernes.

Support matériel

Systèmes uniprocesseurs peuvent bloquer les interruptions

Solutions généralement basées sur la notion de verrouillage (*locking*)

Les architectures fournissent donc du soutient plus spécifique

Instructions *atomiques*:

- `test&set`
- `compare&swap`

Systèmes *transactionnels* optimistes:

- `load-locked` ... `store-conditional`
- `start-transation` ... `commit`

Solutions basées sur les locks

Un *verrou* est un objet avec opérations `acquire` et `release`

```
...do something local...;
acquire(lock);
...do something global...;
release(lock);
...do more local things...;
```

Chaque donnée partagée est associée à un verrou

Chaque verrou “protège” une partie des données partagées

`acquire(lock)` se prononce aussi “vérouille [ces données]”

Contrat de verrouillage

Chaque donnée partagée est associée à **un** verrou

Pas d'accès aux données partagées sans verrouillage

Le bon verrou est utilisé à chaque fois

Pas de condition de course!

(enfin, presque)

Verrouillage avec `test&set`

`test&set (target)` fonctionne comme

```
bool v = *target; *target = true; return v;
```

mais atomique!

```
void acquire (*bool lock) {
    while (test&set (lock))
        /*Wait*/;
}

void release (*bool lock) {
    *lock = false;
}
```

Verrouillage avec `compare&swap`

`compare&swap(target, expected, new)` est comme

```
v = *target;
if (v == expected) *target = new;
return v;
```

mais atomique!

```
void acquire (*bool lock) {
    while (compare&swap (lock, false, true))
        /*Wait*/;
}

void release (*bool lock) { *lock = false; }
```

Attente limitée avec `test&set`

```
waiting[myself] = true;
while (waiting[i] && test&set (lock)) /*Wait*/;
waiting[myself] = false;
```

```
i = myself;
do { i = (i + 1) % n; }
while (i != myself && !waiting[i]);
if (i == j) *lock = false;
else waiting[i] = false;
```

Sémaphores

spinlock trop primitif: utilisé pour construire d'autres opérations

Sémaphore est un outil de synchronisation de plus haut niveau

Deux opérations: `V` et `P` (ou `wait` et `signal`)

```
void wait (semaphore S) {
    while (S <= 0) /*WAIT*/;
    S--;
}

void signal (semaphore S) {
    S++;
}
```

Usage de sémaphores

Pas d'attente active!

Valeur entière interprétable comme nombre de ressources disponibles

Sémaphore binaire équivalent à un verrou (aussi *mutex*)

Peut résoudre divers problèmes de synchronisation

E.g., garantir que S_1 ait lieu avant S_2:

```
P1:                          P2:
     S1;                          wait (sema);
     signal (sema);               S2;
```

Implémentation de sémaphores

Éviter l'attente active, avec l'aide du système d'exploitation

Les opérations `wait` et `signal` doivent être atomiques

On peut utiliser un *spinlock* de manière interne

L'attente active restante est rare

Exemple de code de sémaphore

```
void wait (sem *S) {
  acquire (S->lock);
  S->value--;
  if (S->value < 0) {
     push (myself,
           S->queue);
     block_and_release
        (myself,
         S->lock);
  } else
     release (S->lock);
}
```

```
void signal (sem *S) {
  acquire (S->lock);
  S->value++;
  if (S->value <= 0)
     wakeup
        (pop (S->queue));
  release (S->lock);
}
```

Interblocage (deadlock)

étreinte fatale: quand des threads attendent indéfiniment un événement que seul l'un d'eux peut causer

Cela correspond à un cycle dans le graphe de dépendances

```
acquire (dst->L);        acquire (src->L);
acquire (src->L);        acquire (dst->L);
src->val = dst->val;     dst->val = src->val;
release (src->L);        release (dst->L);
release (dst->L);        release (src->L);
```

Plus de verrous $\Rightarrow$ plus haut risque d'interblocage

Famine (starvation)

Se produit lorsqu'un thread n'a jamais l'opportunité de progresser

Peut être causé par un *deadlock* ou un *live deadlock*, ou toute sorte d'autres circonstances indésirables

Difficile à éliminer complètement

Inversion de priorité

Dans un système d'ordonnancement avec priorité

Situation où un thread de basse priorité bloque un thread de plus haute priorité

Cas simple:

```
P-low                    P-high
acquire (L);             ..dosomething..
..dosomething..          acquire (L);
..do more..              ...
..guess what..           ...
..OK, here it is..       ...
release (L);             ...
```

Inversion de priorité long

Les sections critiques devraient être de courte durée

Mais le blocage peut durer plus longtemps

```
P-low
acquire (L);
..work..
...
...
...
```

```
P-med
...sleep...
...wakeup!
..work..
...
..work..
..no hurry..
```

```
P-high
..sleep...
..sleep...
..wakeup!
acquire (L);
...
...
```

Solutions à l'inversion de priorité

Protocole d'*héritage de priorité*: le thread qui a le verrou L obtient la priorité la plus haute parmi les threads en attente

"Casser" le verrou: le thread de plus basse priorité perd le verrou et doit recommencer sa section critique

Problèmes de synchronisation classiques

Problèmes classiques

- Problème des *Producteurs/consommateurs* (*Bounded-buffer*)
- Problème des *Readers and Writers*

  Contrôler des accès en lecture et en écriture
- Problème des *Dining philosophers*

  ...qui n'affecte pas que les philosophes qui ne savent manger des spaghetti qu'avec deux fourchettes

Producteurs/consommateurs naïf

Sémaphore `full` (initialisé à 0), compte le nombre d'éléments utilisés:

```
..Make next_produced..      wait (full);
..Add next_produced..       ..Get next_consumed..
signal(full);               ..Use next_consumed..
```

Comme l'indique le titre, c'est un peu naïf

Producteurs/consommateurs moins naïf

Sémaphore `full` initialisé à 0, `empty` initialisé à N

```
..Make next_produced..      wait (full);
wait (empty);               ..Get next_consumed..
..Add next_produced..       signal (empty);
signal(full);               ..Use next_consumed..
```

Mais c'est pas encore ça!

Producteurs/consommateurs complet

Sémaphore `full` initialisé à 0, `empty` initialisé à N

Verrou `L` pour protéger les champs du buffer

```
..Make next_produced..
wait (empty);
acquire (L);
..Add next_produced..
release (L);
signal (full);
```

```
wait (full);
acquire (L);
..Get next_consumed..
release (L);
signal (empty);
..Use next_consumed..
```

Problème des Readers-Writers

Une donnée est partagée entre plusieurs threads

Certains threads n'y accèdent qu'en lecture

D'autre y accèdent en lecture et écriture

Problème est de contrôler les threads de manière à:

- permettre plusieurs accès simultanés en lecture
- exclusion mutuelle en cas d'accès en écriture

Propriété désirable: aucun thread ne souffre de famine

Une solution pour Readers-Writers

```
acquire (RW);
..work..
release (RW);
```

```
acquire (L)
counter++;
if (counter == 1)
  acquire (RW);
release (L);
..work..
acquire (L)
counter--;
if (counter == 0)
  release (RW);
release (L);
```

Problème des Dining philosophers

Les philosophes passent leur vie à penser et manger

Ils ne parlent pas à leurs voisins

Ils sont 5, assis autour d'une table ronde

Il y a un grand bol de riz au centre

Il y a 5 assiettes et 5 baguettes répartis autour de la table

Un philosophe a besoin de 2 baguette pour manger

Difficultés liées aux mutex

Il est trop facile de les utiliser incorrectement:

- Oublis
- Usage du mauvais verrou
- `acquire` sans le `release` correspondant
- Appels répétés: `acquire(x); ...; acquire (x);`
- ...

Risque d'interblocage

Mauvaises performances si la granularité est trop grossière

Mauvaises performances si la granularité est trop fine

Moniteurs

Abstraction de plus niveau pour exclusion mutuelle et synchronisation

Protège l'accès aux données internes d'un *abstract data type*

```
(define-monitor
  ;; Shared variables declarations.
  (define-variable var1 : type1)
  (define-variable var2 : type2)
  ;; Operations on those variables.
  (define-function fun1 (args1) ...)
  (define-function fun2 (args2) ...))
```

Le moniteur garanti l'exclusion mutuelle dans les fonctions `foo1`, `foo2`, ...

Variables de conditions

Système de synchronisation utilisé dans un moniteur

```
(define-condition C)
```

Opérations sur ces *condition variables*:

- `wait`: Bloquer le thread jusqu'à ce que la condition soit *signalée*
  Le thread quitte sa section critique pendant l'attente
- `signal`: Réveille 1 des threads qui attend cette condition
- `broadcast`: Réveille les threads qui attendent cette condition

Variantes de sémantique

Si P_1 attend C et P_2 fait `signal(C)`:

- *Signal and wait*: P_1 entre dans sa section critique et P_2 est mis en attente
- *Signal and continue*: P_2 continue et P_1 doit attendre que P_2 quitte sa section critique

Certains systèmes n'offrent que `broadcast` et l'appellent `signal`

Un moniteur pour philosophes

```
(define-monitor
  (define-variable sticks (array bool 5))
  (define-condition change)
  (define-function begin_lunch (p)
    (while (not (and sticks[p] sticks[(p+1)%5]))
      (wait change))
    (set sticks[p] false)
    (set sticks[(p+1)%5] false))
  (define-function end_lunch (p)
    (set sticks[p] true)
    (set sticks[(p+1)%5] true)
    (broadcast change))
```

Autre moniteur pour philosophes

```
enum { T, H, E } state[5];        condition ready[5];
void update (int p) {
  if (state[p] == H
       && state[(p+4)%5] != E && state[(p+1)%5] != E)
     (state[p] = E; signal (ready[p]));
}
void begin_lunch (int p) {
  state[p] = H; update (p);
  if (state[p] != E) wait (ready[p]);
}
void end_lunch (int p) {
  state[p] = T;
  update ((p+4)%5); update ((p+1)%5);
```

Un moniteur avec des sémaphores

```
semaphore mutex = true;
semaphore next = 0;
int next_count = 0;
```

Le corps de chaque fonction devient:

```
{
  sem_wait (mutex);
  ...body...
  sem_signal (next_count > 0 ? next : mutex);
}
```

Cela assure l'exclusion mutuelle

Moniteurs et deadlock

Appel d'une fonction du moniteur depuis lui-même

mutex récursifs ou multiples points d'entrée

Risques classiques d'interblocage

Variables de condition avec des sémaphores

```
struct condvar {
    semaphore sem = 0;
    int count = 0;
}
```

```
cv_wait (condvar *cv) {
  cv->count++;
  sem_signal
     (next_count > 0
      ? next : mutex);
  sem_wait (cv->sem);
  cv->count--;
}
```

```
cv_signal (condvar* cv) {
  if (cv->count > 0) {
    next_count++;
    sem_signal (cv->sem);
    sem_wait (next);
    next_count--;
}
```

Réveil des threads dans un moniteur

Quel thread devrait réveiller `cv_signal`?

- LIFO: Simple mais injuste
- FIFO: Plus juste
- Priorité: `cv_wait` peut spécifier la priorité

Un moniteur avec priorité

```
(define-monitor
  (define-variable busy boolean)
  (define-condition ready)
  (define-function acquire (time)
    (if busy (wait ready time))
    (set busy true))
  (define-function release ()
    (set busy false)
    (signal ready)))
```

Variables Vides/Pleines

Autre mechanisme de synchronization inspiré du *dataflow*

```
newMVar: α → IO (MVar α)
takeMVar: MVar α → IO α
putTVar: MVar α → α → IO ()
```

`take_MVar` attend que la variable soit pleine, puis la vide

`put_MVar` attend que la variable soit vide, puis la rempli

Pourquoi Linux RCU?

Souvent, les accès en lecture sont de loin les plus fréquents

Maximiser la performance des accès en lecture

La solution au *readers-writers* n'est donc pas satisfaisante

But: accès en lecture ne bloquent **jamais**

Linux RCU

RCU = Read-Copy Update

Les accès en lecture obtiennent une “copie”

Modifications doivent préserver la validité de ces copies

Un accès en lecture peut continuer à utiliser une vieille copie

Pas forcément applicable dans tous les cas

RCU 1: Simple affectation

```
struct foo { int a; int b; int c; };
struct foo *gp = NULL;
...
  p = kmalloc (sizeof (*p), GFP_KERNEL);
  p->a = 1; p->b = 2; p->c = 3;
  gp = p;
```

Seule instruction de la section critique: `gp = p;`

Déjà atomique!

Ou presque: `rcu_assign_pointer (gp, p);`

RCU 1: Simple lecture

```
p = gp;
if (p != NULL) {
  do_something_with (p->a, p->b, p->c);
}
```

Seule instruction de la section critique: `p = gp;`

Déjà atomique!

Ou presque!

RCU 1: Simple lecture corrigée

```
rcu_read_lock ();
p = rcu_dereference (gp);
if (p != NULL) {
  do_something_with (p->a, p->b, p->c);
}
rcu_read_unlock ();
```

`rcu_dereference` protège de certaines optimisations

`rcu_read_lock` ne bloque jamais

Synchronisation optimiste

Au lieu de bloquer l'entrée pour éviter les conflits

Présumer de l'absence de conflit, et vérifier à la sortie

En cas conflit: *rollback* et on ressaie

Aussi décrit comme *lockfree*

Pas de risque d'inversion de priorité

Pas de *deadlock*, mais risque de *livelock*

Peut même arriver à être *waitfree*!

Conditions nécessaires à l'optimisme

Espérer le mieux, mais être préparé au pire

Pouvoir détecter les conflits

Pouvoir faire un *rollback*:

- Garder traces des anciennes valeurs
- Pas d'effets de bord "externes"

Pouvoir décider quand ressayer

Haskell STM

Garanti que tous les accès partagés sont protégés

Pas besoin de savoir quel verrou protège quelle donnée

Optimiste: pas de *deadlock*

Assure l'abence d'effets de bord "externes"

Détecte les conflits et garde les anciennes valeurs pour *rollback*

Sait même quand il faut ressayer

Promet un monde sans guerres

Primitives STM

Basé sur les *monades*: distinguer une commande de son exécution

```
atomically: STM α → IO α
newTVar: α → STM (TVar α)
readTVar: TVar α → STM α
writeTVar: TVar α → α → STM ()
```

`atomically` est comme une paire `acquire...release`

Pas besoin de spécifier quel(s) verrou(s) prendre

Pas d'oubli: seul `atomically` peut exécuter `STM α`

rollback: Seul `STM α` peut être exécuté dans `atomically`

Implémentation de STM

Chaque `TVar` contient un numéro de version

Chaque lecture stocke la variable et sa valeur dans un *log*

Chaque écriture stocke variable, et valeur dans un *log*

Écritures ne changent pas immédiatement les variables

Commit à la fin de `atomically`:

```
acquire (stm_lock);
if (!check_consistency (log))
  { release (stm_lock); free (log); goto start; }
perform_updates (log);
release (stm_lock);
```

Attendre avec STM

Même principe que les *condition variables*

Mais plus automatique: pas besoin de `signal`, ni de *condvar*

```
retry: STM α
orElse: STM α → STM α → STM α
```

Le *log* indique à STM quelles variables doivent changer

`orElse` permet d'attendre *plusieurs* conditions!

Philosophes fonctionnels

```
begin_lunch p =
  atomically (do
    fl <- readTVar (forks[p])
    fr <- readTVar (forks[(p+1)%5])
    if fl && fr then do
      writeTVar fl false
      writeTVar fr false
    else retry)

end_lunch p =
  atomically (do
    writeTVar (forks[p])       true
    writeTVar (forks[(p+1)%5]) true)
```