Programmes non-bloquants et logique de séparation

Étienne Lozes

LSV, ENS Cachan

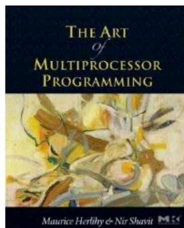
6 février 2009

Plan

1. Exemples
2. Propriétés fonctionnelles
3. Propriétés de progrès

Références

Sur la concurrence non-bloquante :



- *The art of multiprocessor programming*, Maurice Herlihy, Nir Shavit

Sur la logique de séparation concurrente :

- *Resource, Concurrency and local reasonning* O'Hearn, CONCUR'04
- *Modular Safety Checking for Fine-Grained Concurrency*, Cristiano Calcagno, Matthew Parkinson, Viktor Vafeiadis.

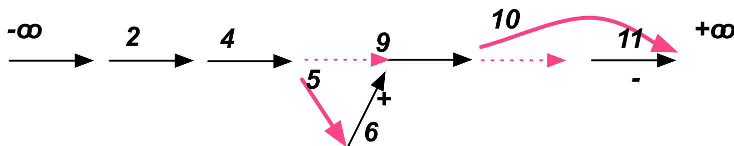
Exemples

Premier exemple : un module concurrent de gestion de liste

```
private function
search(v)

  t=head ;
  while
(t.next.val< v)
do t :=t.next ;
end
```

```
function
insert(v)

list.lock()
t :=search(v)
t' := cons(v,[t])
t.next :=t' ;
list.unlock()
```

```
function
remove(v)

list.lock()
t :=search(v)
t' := t.next
t.next :=t'.next ;
free(t') ;
list.unlock()
```



Exclusion mutuelle et efficacité

Avantages :

- Les moniteurs de Hoare sont simples conceptuellement
- On sait bien prouver de tels programmes

Inconvénients :

- on utilise souvent des verrous, risque de deadlock
- beaucoup de perte de temps CPU

Exclusion mutuelle et efficacité

Avantages :

- Les moniteurs de Hoare sont simples conceptuellement
- On sait bien prouver de tels programmes

Inconvénients :

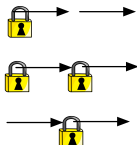
- on utilise souvent des verrous, risque de deadlock
- beaucoup de perte de temps CPU

Dans ce qui suit, on programme avec :

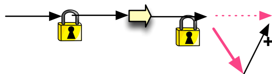
- parcimonie d'exclusion mutuelle
- optimisme sur la fréquence des interférences
- on affine le grain : on découpe la mémoire partagée en zones de conflit les plus petites possibles
- on simule l'atomicité en faisant une boucle `while(echec_commit)tentative_commit`.

Affiner le grain : lock-coupling list

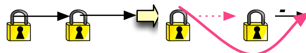
```
function
search(v)

t' :=head;
t :=t'.next;
acquire t'.lock
while (t.val< v)
do
 acquire t.lock
 release t'.lock
t' :=t
t :=t'.next;
end
```



```
function
insert(v)

(t,t') :=search(v)
t'' :=
cons(v,0,t)
t'.next :=t'';
release t'.lock
```



```
function
remove(v)

(t,t') :=search(v)
acquire t.lock
t'.next := t.next
free(t);
release t'.lock
end
```



Parcimonie et optimisme : lazy lock-coupling

On peut faire mieux :

- On avance sans prendre de locks
- Un élément est dans la liste si il y est physiquement et si il n'a pas été marqué comme étant sur le point d'être enlevé.

Retrait en deux temps : logique puis physique.

```
function search(v)
t :=head;
while t.val<v do
  t' :=t;
  t :=t'.next;
end;
```

```
function contains(v)
(t,t') :=search(v);
return t.val==v &! t.marked
```

Lazy lock-coupling : retrait

```
#def validate(t,t') !t'.marked & !t.marked &
t==t'.next
function remove(v)

while !done do
  (t,t') :=search(v)
  acquire t'.lock
  acquire t.lock
  if validate(t,t') then
   t.marked :=true ; (1)
   t'.next :=t.next (2)
   done :=true ;
  endif
  release t.lock ;
  release t'.lock ;
wend
```

(1) : retrait logique
(2) : retrait physique

Lazy lock-coupling : retrait

```
  #def validate(t,t') !t'.marked & !t.marked &
  t==t'.next
function remove(v)

while !done do
  (t,t') :=search(v)
  acquire t'.lock
  acquire t.lock
  if validate(t,t') then
   t.marked :=true ; (1)
   t'.next :=t.next (2)
   done :=true ;
  endif
  release t.lock ;
  release t'.lock ;
wend
```

(1) : retrait logique
(2) : retrait physique

Boucle tentative-echec-reprise

Lazy lock-coupling : retrait

```
#def validate(t,t') !t'.marked & !t.marked &
t==t'.next
function remove(v)

while !done do
  (t,t') :=search(v)
  acquire t'.lock
  acquire t.lock
  if validate(t,t') then
   t.marked :=true ; (1)
   t'.next :=t.next (2)
   done :=true ;
  endif
  release t.lock ;
  release t'.lock ;
wend
```

(1) : retrait logique
(2) : retrait physique

Boucle tentative-echec-reprise

Remarque : on utilise le GC pour effacer.
Pas possible de s'en passer !

Lazy lock-coupling : insertion

```
function insert(v)

while !done do  (t,t') :=search(v)
   acquire t'.lock
   acquire t.lock
  if validate(t,t') then
   t'' := cons(v,0,false,t)
   t'.next :=t'';
  endif
  release t.lock;
  release t'.lock;
wend
```

Lazy lock-coupling : bilan

Bilan ;

- Le parcours est beaucoup plus rapide qu'avant.
- insertions et effacements peuvent causer plusieurs parcours, mais *en étant optimiste* on le refait peu de fois et on y gagne.

Lazy lock-coupling : bilan

Bilan ;

- Le parcours est beaucoup plus rapide qu'avant.
- insertions et effacements peuvent causer plusieurs parcours, mais *en étant optimiste* on le refait peu de fois et on y gagne.
- c'est tout de même assez compliqué pour être sûr que ça marche sans faire de faute mémoire (cf. preuve de programme)...
- ... et en dehors des fautes mémoires, ce n'est pas bien clair ce que veut dire "fonctionne bien"...

Lazy lock-coupling : bilan

Bilan ;

- Le parcours est beaucoup plus rapide qu'avant.
- insertions et effacements peuvent causer plusieurs parcours, mais *en étant optimiste* on le refait peu de fois et on y gagne.
- c'est tout de même assez compliqué pour être sûr que ça marche sans faire de faute mémoire (cf. preuve de programme)...
- ... et en dehors des fautes mémoires, ce n'est pas bien clair ce que veut dire "fonctionne bien"...
- ... encore moins comment le vérifier !

Lazy lock-coupling : bilan

Bilan ;

- Le parcours est beaucoup plus rapide qu'avant.
- insertions et effacements peuvent causer plusieurs parcours, mais *en étant optimiste* on le refait peu de fois et on y gagne.
- c'est tout de même assez compliqué pour être sûr que ça marche sans faire de faute mémoire (cf. preuve de programme)...
- ... et en dehors des fautes mémoires, ce n'est pas bien clair ce que veut dire "fonctionne bien"...
- ... encore moins comment le vérifier !
- une remarque : le retrait logique est essentiel pour implémenter un basculement de l'état global en une seule instruction (cf linearisabilité). Ca ne marche pas avec un retrait physique.

Une pile concurrente [Treiber]

```
atomic bool CAS(a,b,c){
if (*a==b) {*a=c ;return true ;} return false ;}
```

Une pile concurrente [Treiber]

```
atomic bool CAS(a,b,c){
if (*a==b) {*a=c;return true;} return false;}
```

```
void push(value  v) {
  cell *t, *x;
  x = alloc();
  x->data = v;
  do {
    t = top_ptr;
    x->next = t;
  } while (!CAS(&top_ptr,t,x));
}
```

```
value pop() {
  cell *t, *x;
  do {
    t = top_ptr;
    if (t == NULL) return EMPT
    x = t->next;
  }
  while (!CAS(&top_ptr,t,x));
  return t->data;
}
```

Treiber

- "ça marche"
- pas de verrou, donc pas de deadlock possible
- pas de "livelock" (voir conditions de progrès)
- il existe plus compliqué...

La même pile sans GC

```
void push(value  v) {
  cell *t, *x;
  x = alloc();
  x->data = v;
  do {
    t = top_ptr;
    x->next = t;
  } while (!CAS(&top_ptr,t,x));
}
```

```
value pop() {
  cell *t, *x;
  do {
    t = top_ptr;
    if (t == NULL) return EMPT
    x = t->next;
  } while (!CAS(&top_ptr,t,x));
  data_t data = t->data;
  free(t);
  return data;
}
```

La même pile sans GC

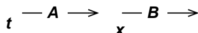
```
void push(value  v) {
  cell *t, *x;
  x = alloc();
  x->data = v;
  do {
    t = top_ptr;
    x->next = t;
  } while (!CAS(&top_ptr,t,x));
}
```

```
value pop() {
  cell *t, *x;
  do {
    t = top_ptr;
    if (t == NULL) return EMPT
    x = t->next;
  }
  while (!CAS(&top_ptr,t,x));
  data_t data = t->data;
  free(t);
  return data;
}
```

CA NE MARCHE PAS !!

Le problème ABA, importance du GC

1. on part de la pile
   `A : :B : :nil.`
2. Thread 1 commence pop et est préempté



5.
```
value pop() {
  cell *t, *x;
  do {
    t = top_ptr;
    if (t == NULL) return EMPT
    x = t->next; (1 STOP)
  }
  while (!CAS(&top_ptr,t,x));
  data_t data = t->data;
  free(t);
  return data;
}
```

Le problème ABA, importance du GC

1. on part de la pile `A : :B : :nil`.
2. Thread 1 commence pop et est préempté
3. Thread 2 fait pop `:A` (t de 1 est maintenant libre)

t

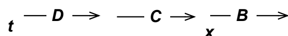
— B ⟶
x

5.
```
value pop() {
   cell *t, *x;
   do {
      t = top_ptr;
      if (t == NULL) return EMPT
      x = t->next; (1 STOP)
   }
   while (!CAS(&top_ptr,t,x));
   data_t data = t->data;
   free(t);
   return data;
}
```

Le problème ABA, importance du GC

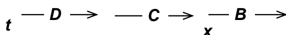
1. on part de la pile
   `A : :B : :nil.`
2. Thread 1 commence pop et est préempté
3. Thread 2 fait pop `:A`
4. Thread 2 fait `push(C) ;push(D)` et alloue D en t de 1
5.

```
value pop() {
  cell *t, *x ;
  do {
    t = top_ptr ;
    if (t == NULL) return EMPT
    x = t->next ; (1 STOP)
  }
  while (!CAS(&top_ptr,t,x)) ;
  data_t data = t->data ;
  free(t) ;
  return data ;
}
```



Le problème ABA, importance du GC

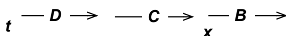
1. on part de la pile `A : :B : :nil`.
2. Thread 1 commence pop et est préempté
3. Thread 2 fait pop `:A`
4. Thread 2 fait `push(C) ;push(D)` et alloue D en t de 1
5. Thread 1 reprend et passe le test




```
value pop() {
  cell *t, *x ;
  do {
    t = top_ptr ;
    if (t == NULL) return EMPT
    x = t->next ; (1 STOP)
  }
  while (!CAS(&top_ptr,t,x)) ;
  data_t data = t->data ;
  free(t) ;
  return data ;
}
```

Le problème ABA, importance du GC

1. on part de la pile `A : :B : :nil`.
2. Thread 1 commence pop et est préempté
3. Thread 2 fait `pop :A`
4. Thread 2 fait `push(C) ;push(D)` et alloue D en t de 1
5. Thread 1 supprime C,D d'un coup `pop :D`, puis pop encore `pop :B`




```
value pop() {
  cell *t, *x ;
  do {
    t = top_ptr ;
    if (t == NULL) return EMPT
    x = t->next ; (1 STOP)
  }
  while (!CAS(&top_ptr,t,x)) ;
  data_t data = t->data ;
  free(t) ;
  return data ;
}
```

Le problème ABA, importance du GC

1. on part de la pile `A : :B : :nil`.
2. Thread 1 commence pop et est préempté
3. Thread 2 fait `pop :A`
4. Thread 2 fait `push(C) ;push(D)` et alloue D en t de 1
5. Thread 1 supprime C,D d'un coup `pop :D`, puis pop encore `pop :B`

```
value pop() {
  cell *t, *x ;
  do {
    t = top_ptr ;
    if (t == NULL) return EMPT
    x = t->next ; (1 STOP)
  }
  while (!CAS(&top_ptr,t,x)) ;
  data_t data = t->data ;
  free(t) ;
  return data ;
}
```

`(pop :A ;push(C) ;push(D))` || `(pop :D ;pop :B)`

CECI N'EST PAS CORRECT !

Propriétés fonctionnelles

Traces avec appel et retour : les "histoires"

Evènements

- appel de fonction : push(A), insert(5), deq()...
- retour de fonction : enq :void, pop :B, ...

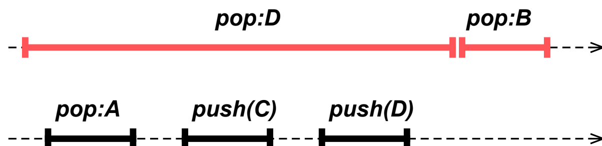
Evènements localisés : (ThreadId,evt)

Histoires : suite d'événements localisés

Histoire séquentielle : tout appel est immédiatement suivi d'un retour, (sauf éventuellement le dernier appel)

La trace du problème ABA

(1,pop())(2,pop())(2,pop :A)(2,push(C))(2,push :)...
...(2,push(D))(2,push :)(1,pop :D)(1,pop())(1,pop :B)



Projections, complétions

Projection sur le thread t : $H \downharpoonright t$ = sous histoire localisée en t

Histoire bien formée : H est bien formée si $H \downharpoonright t$ est séquentielle pour tout t.

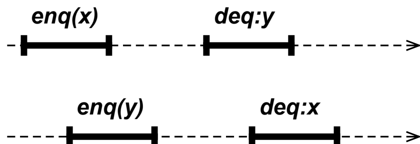
Complétion d'une histoire : On rajoute des retours de fonction aux appels pendants.

Projection sur la ressource r : $H \downharpoonright r$ = trace (sans localités) des évènements concernant la ressource r.

Linéarisabilité, intuitivement

Intuitivement : dans toute fonction, il y a une instruction qui joue le rôle de commutateur. La fonction a un effet global basé sur l'instantané de la mémoire à ce moment-là.

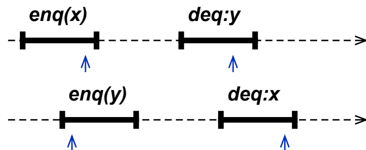
Exemple linéarisable



Linéarisabilité, intuitivement

Intuitivement : dans toute fonction, il y a une instruction qui joue le rôle de commutateur. La fonction a un effet global basé sur l'instantané de la mémoire à ce moment-là.

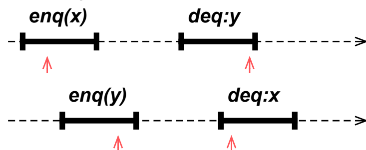
Exemple linéarisable



Linéarisabilité, intuitivement

Intuitivement : dans toute fonction, il y a une instruction qui joue le rôle de commutateur. La fonction a un effet global basé sur l'instantané de la mémoire à ce moment-là.

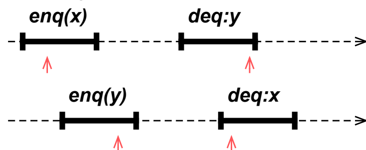
Exemple linéarisable



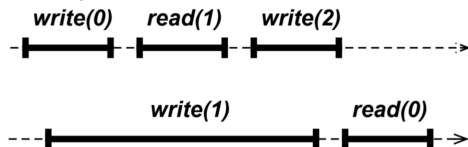
Linéarisabilité, intuitivement

Intuitivement : dans toute fonction, il y a une instruction qui joue le rôle de commutateur. La fonction a un effet global basé sur l'instantané de la mémoire à ce moment-là.

Exemple linéarisable



Exemple non linéarisable



Spécification

Spécification :
La spécification d'une ressource r est un sous-ensemble de traces d'évènements *non localisés* et *séquentiels*
ex : pour la pile concurrente,
$\{w \in (\texttt{push(A)} + \texttt{push} : +\text{pop}() + \texttt{pop} : \texttt{A})^*\} \cap$
$\{\text{mots séquentiels}\} \cap \{\text{mots de pile}\}$

$\prec$ **-satisfaire une spécification** :
toute exécution a une histoire H telle qu'il existe $H' \prec H$, qui admet une complétion H'' dont la trace non localisée $H'' \downharpoonleft r$ appartient à la spécification de r.

Linéarisabilité

Formellement : $H' \prec_{lin}^{1} H$ si mêmes histoires à une permutation d'appels ou de retours de fonctions voisins *et le nombre de recouvrements d'intervalles n'est pas plus grand dans H'*.

$\prec_{lin}$ est la clôture réflexive transitive de $\prec_{lin}^{1}$.

Exemples :
(1,A()),(2,B()),(1,A :),(2,B :) $\succ_{lin}$ (1,A()),(1,A :),(2,B()),(2,B :)
(1,A()),(2,B()),(1,A :),(2,B :) $\succ_{lin}$ (2,B()),(2,B :),(1,A()),(1,A :)
(1,A()),(1,A :),(2,B()),(2,B :) $\not\succ_{lin}$ (2,B()),(2,B :),(1,A()),(1,A :)

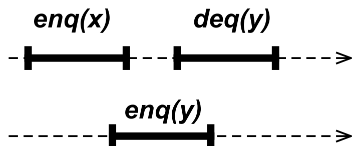
Cohérence séquentielle

Intuitivement : *tout se passe comme si chaque appel de fonction commence une section critique, termine sans être préempté, et laisse la place à un autre appel d'un autre thread.*

Formellement : $H \sim_{seq} H'$ si $H \downharpoonleft t = H' \downharpoonleft t$ pour tout thread t.

Critiques

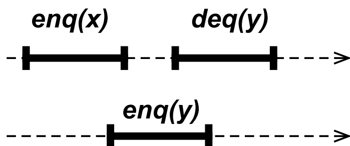
Trop restrictif :
cette file est séquentiellement cohérente.



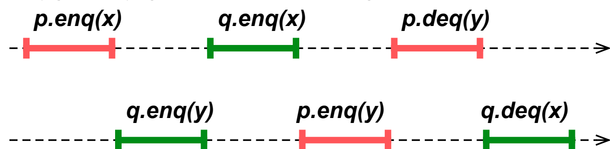
:

Critiques

Trop restrictif :
cette file est séquentiellement cohérente.



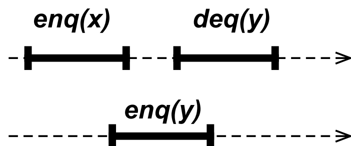
Non compositionnel :
$H \downharpoonright p$, $H \downharpoonright q$ cohérents, mais pas H



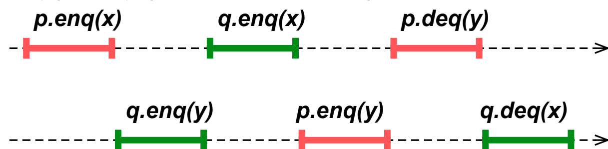
⋮

Critiques

Trop restrictif :
cette file est séquentiellement cohérente.



Non compositionnel :
$H \downharpoonright p$, $H \downharpoonright q$ cohérents, mais pas H



Contraire aux optimisations :
des fois, c'est bien de s'autoriser à ne pas être séquentiel.

Digression : l'échec de Peterson

L'algorithme de Peterson :

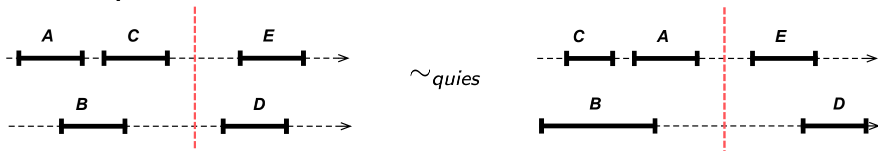
```
interested[i] := true ;
polite := i ;
while(interested[other]=true and polite=i) ;
...
interested[i] :=false ;
```

Cet algorithme est parfait théoriquement...
mais quand on l'implémente sur un multicoeur, on risque des interbloquages. Pourquoi ?

Cohérence des états de repos (quiescents)

Intuitivement : les états de repos sont des barrières de synchronisation, on réordonne comme on veut à l'intérieur des barrières.

Exemple :



Propriétés :

- incomparable avec cohérence séquentielle (exercice !)
- compositionnel
- c'est bien pour raisonner sur du code optimisable (matériel, compilateur)

Propriétés de progrès

Les conditions de progrès pour la concurrence bloquante

non-interblocage : dans tout état, il y a un ordonancement qui permet à au moins un thread de rentrer en section critique (une def possible).

non-famine : si chacun demande à rentrer un nombre fini de fois en section critique, tout le monde sera servi quel que soit l'ordonnancement.

Les conditions de progrès pour la concurrence bloquante

non-interblocage : dans tout état, il y a un ordonancement qui permet à au moins un thread de rentrer en section critique (une def possible).

non-famine : si chacun demande à rentrer un nombre fini de fois en section critique, tout le monde sera servi quel que soit l'ordonnancement.

Conditions de progrès interdépendant

- pas tauIérant aux fautes
- ni aux attentes longues imprévues (défaut de page, surcharge de proc,...)

Propriétés de progrès *indépendant* plus prisées...
...et on ne fait que de l'**attente active**

Notion de *non attente* (wait-free)

Tout appel finit (en un nombre fini d'opérations).

bounded wait-free : on peut borner ce nombre d'opérations.

population-oblivious : la borne ne dépend pas du nombre de threads.

Notion de *non verrouillage* (lock-free)

Infiniment souvent, on trouve un appel qui termine.

Affaiblissement de wait-free.

Suffisant en pratique, et autorise des algos plus efficaces que wait-free.

Notion de *non obstruction* (obstruction free)

Dans toute configuration, pour tout thread t, si on préempte tous les threads autre que t, t termine

Affaiblissement de lock-free.

Condition de progrès dépendant, mais :

- si on fait insère dans la boucle d'attente active un `i++ ;sleep(2**i) ;` (exponential backoff), ça marchera très bien.
- pas besoin de bloquer tout le monde en pratique, uniquement ceux qui sont en conflit.

Résumé et exemples

Conditions de correction fonctionnelle

- non compositionnel : consistance séquentielle.
- compositionel : consistance quiescente, linéarisabilité
- linéarisabilité vraiment important (cf lazy list).
- linéarisabilité $\Rightarrow$ c. séquentielle et quiescent
- consistance séquentielle et consistance quiescente : incomparables.

Résumé et exemples

Conditions de correction fonctionnelle

- non compositionnel : consistance séquentielle.
- compositionel : consistance quiescente, linéarisabilité
- linéarisabilité vraiment important (cf lazy list).

Conditions de progrès

- wait-free : on ne boucle jamais infiniment longtemps (ex : lazy search)
- lock-free : infiniment souvent quelqu'un termine (ex : insert, delete)
- obstruction-free : quiconque prend la main à un moment quelconque est sûr de terminer à partir de ce moment (ex : Treiber push, pop)
- wait-free $\Rightarrow$ lock-free $\Rightarrow$ obstruction-free.

La preuve formelle aujourd'hui

- outil semi-automatique pour les fautes mémoires (RGSep)
- quelques pistes pour faire des preuves de linéarisabilité (Vafeiadis, SAS'08)
- un outil (?) pour les conditions de progrès (Cook et al, POPL'09)

Il reste encore pas mal de travail, il existe de nombreuses façons de programmer, et autant de prouver