Algorithmes concurrents à grain fin

Synchronisation et algorithmes

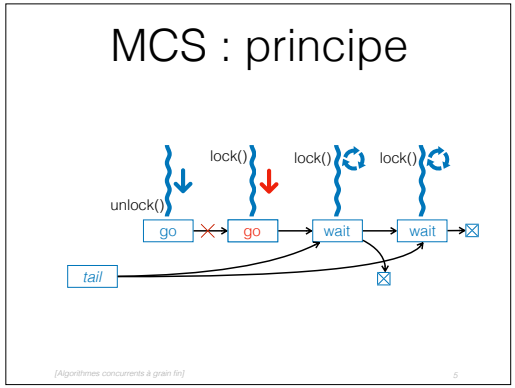
Algorithmes d'attente active

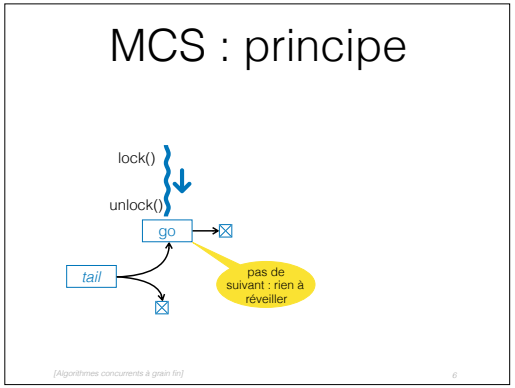
cf Herlihy verrous à attente active

MCS: Mellor-Crummey & Scott

Equitable : ordre FIFO
Efficace :

- attente active uniquement en mémoire locale
- surcoût mémoire constant

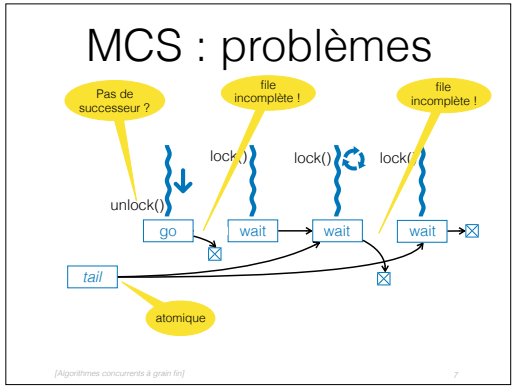


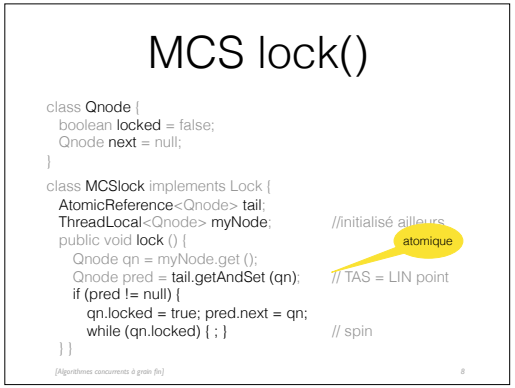


Principe : une file de *qnode* ; chacun comprenant un booléen (*wait/go*) et le lien vers le suivant.
Chaque processus a son propre *qnode* et attend en boucle active tant qu'il est dans l'état *wait*.
Seul le premier est dans l'état *go* et exécute la section critique.
Pour sortir de la section critique, le premier fait *unlock()* ce qui le retire de la file et met le suivant à *go*.
Pour ajouter un nouveau processus à la fin de la file, 2 étapes : *tail* doit pointer sur le nouveau *qnode*, et le précédent
(l'ancien tail) aussi.
Avantages :

- chaque processus boucle sur son *qnode* privé, pas d'interférence ni pollution du cache ; pas de pb de performance
- les processus sont servis dans l'ordre PEPS=FIFO (équité)

Cas initial : ajouter le *qnode* à la liste vide. État initialisé à "*go*" (passe à "*wait*" seulement si liste non vide).
Cas *unlock* où il n'y a pas de suivant : je suis l'unique processus, personne à réveiller.



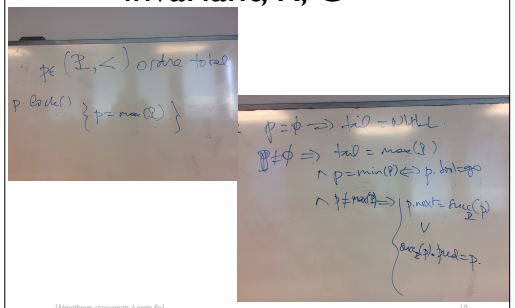


1. L'ajout nécessite de modifier 2 pointeurs, donc non atomique ⟹ tail atomique, chaînage plus tard
2. Le point de linéarisation étant le lien *tail,* la liste peut être incomplète
3. Elle peut même avoir plusieurs trous
4. Que faire si celui qui fait *unlock* ne sait pas qu'il a un suivant ??? ⟹ vérifier que tail == moi, sinon c'est qu'il y a quelqu'un derrière ⟹ attendre qu'il finisse de réparer le chaînage

MCS unlock()

```
public void unlock () {
  Qnode qn = myNode.get ();
  if (qn.next == null) {
    // normalement je suis le dernier ; vérifier
    if (tail.compareAndSet (qnode, null))    // CAS = LIN 1
      return; // ouf, je suis bien le dernier
    // non, il y a un suivant : attendre qu'il finisse de chaîner
    while (qn.next == null) { ; }
  }
  qn.next.locked = false;                    // LIN 2
  qn.next = null;
  }
}
```


invariant, R, G***



MCS Queue Lock

```
class Qnode {
 boolean locked = false;
 qnode   next   = null;
}
```


MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
   while (qnode.locked) {}
  }}}
```


getAndSet = TAS (atomically set the value, return old value)

MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
   while (qnode.locked) {}
  }}}
```

Make a QNode

getAndSet = TAS (atomically set the value, return old value)
LIN point

MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
   while (qnode.locked) {}
  }}}
```

add my Node to the tail of queue

getAndSet = TAS (atomically set the value, return old value)
LIN point

MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
    while (qnode.locked) {}
  }}}
```

Fix if queue was non-empty

MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
   while (qnode.locked) {}
  }}}
```

Wait until unlocked

MCS Queue Unlock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void unlock() {
  if (qnode.next == null) {
   if (tail.CAS(qnode, null)
    return;
   while (qnode.next == null) {}
  }
 qnode.next.locked = false;
}}
```


MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void unlock() {
  if (qnode.next == null) {
   if (tail.CAS(qnode, null)
    return;
   while (qnode.next == null) {}
  }
 qnode.next.locked = false;
}}
```

Missing successor?

MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void unlock() {
  if (qnode.next == null) {
   if (tail.CAS(qnode, null)
    return;
    while (qnode.next == null) {}
  }
 qnode.next.locked = false;
}}
```

If really no successor, return

MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void unlock() {
  if (qnode.next == null) {
   if (tail.CAS(qnode, null)
    return;
  while (qnode.next == null) {}
  }
 qnode.next.locked = false;
}}
```

Otherwise wait for successor to catch up

CAS // LIN1
locked := false // LIN2

Successor is slow to fix chain; wait until chain completed
CAS // LIN1
locked := false // LIN2

MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference queue;
 public void unlock() {
  if (qnode.next == null) {
   if (tail.CAS(qnode, null)
    return;
   while (qnode.next == null) {}
  }
 qnode.next.locked = false;
}}
```

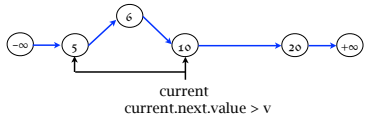
Pass lock to successor

CAS // LIN1
locked := false // LIN2

Listes chaînées concurrentes

Cf. *Proving Correctness of Highly-Concurrent Linearisable Objects*
Vafeiadis, Herlihy, Hoare, Shapiro. PPoPP 2006

Liste chaînée d'entiers



L.add (6)
L.remove (12)
L.contains (-1)

Invariant de structure:

∧ *Head.value = HEADVAL* ∧ *Tail.value = TAILVAL*

∧ ∀*n* ≠ *Tail: n.next* ≠ null
∧ *n.value* < *n.next.value*

Comment se convaincre que c'est correct ?

```
class SeqList implements IntSet {
  boolean add (int v) { … }
  boolean remove (int v) { … }
  boolean contains (int v) { … }
}
```

En séquentiel, on peut utiliser le test :

- Pré-condition, post-condition
  - assert
- Invariant de structure
  - assert
- Relation d'abstraction
  - comparer à une autre mise en œuvre

Stress tests aléatoires, conditions aux limites, etc.

rappel

Dans le cas séquentiel les assert et les tests suffisent. Vérifier tous les chemins.
Ici :

- stress test : beaucoup d'ajouts/retraits avec valeurs communes ou différentes
- condition aux limites : liste vide

Objet concurrent correct

Sûreté :
- Conforme spécification séquentielle
  - Pré-condition, post-condition
  - Invariant de structure
  - Relation d'abstraction
- Point de linéarisation

Vivacité :

	Bloquant	*Non bloquant*
∃ *vivace*	sans étreinte mortelle	sans verrou
∀ *vivaces*	sans famine	sans attente

Vérifier du code concurrent ?

Problèmes :
- Peut-on tester le point de linéarisation ?
- Rely, guarantee non testables
- On ne peut pas tester `assert` en-dehors du point de linéarisation
- Les sondes détruisent la concurrence

Test : prendre un instantané, vérifier raisonnable
- ↘ taille de l'espace ↗ proba conflit

Seule approche fiable :
- Afficher le point de linéarisation
- *Prouver* les invariants (sortie, structure, abstraction) en ce point

rappel

Point de linéarisation : il faudrait comparer avec une exécution séquentielle dans le même ordre...
↘ taille de l'espace ↗ proba conflit = dans liste, *head.val = 0; tail.val = 10*

Liste chaînée concurrente à verrouillage gros grain

```
class CoarseGrainList implements IntSet {
  boolean synchronized add (int v) { … }
  boolean synchronized remove (int v) {  …  }
  boolean synchronized contains (int v) {  … }
}
```

Correct ?

- Identique séquentiel
- On fait confiance au compilateur
- Point de linéarisation : prise du verrou

'synchronized' prend un verrou au début, le relâche à la fin.
Chaque méthode prend effet (c-à-d ses effets de bord sont visibles et la valeur de retour est calculée) au moment où le verrou est relâché.

Liste chaînée concurrente à verrouillage gros grain

```
class CoarseGrainList implements IntSet {
  boolean add (int v) {
    lock.lock();
    try { … }
    finally {lock.unlock();}
  }
  …
}
```

Correct?

- Idem "synchronized"
- Point de linéarisation : prise du verrou

'synchronized' prend un verrou au début, le relâche à la fin.
Chaque méthode prend effet (c-à-d ses effets de bord sont visibles et la valeur de retour est calculée) au moment où le verrou est relâché.

Granularité

Gros grain : toute la liste est verrouillée
- Exécution séquentielle
- Trivialement correct
- Goulot d'étranglement ?

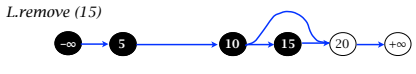
Grain fin :
- Nœud par nœud
- Nettement plus complexe
- Facile de se tromper !

Verrouillage à grain fin

Principe : à chaque nœud son verrou distinct

Quel algorithme ?
- Il faut verrouiller *deux* nœuds
  - *add* : de part et d'autre du point d'insertion
  - *remove* : le nœud à enlever, et son prédécesseur
  - *contains* : le nœud qui contient (peut-être) la valeur, et son prédécesseur
- Pendant le passage de relais, un seul verrou

Verrous couplés : Hand-over-Hand list



Ensemble d'entiers réalisé par liste chaînée triée
Concurrence à grain fin : verrous couplés
Prouver :
- invariant de structure
- invariant de l'abstraction
- point de linéarisation

Rely : *un nœud verrouillé reste accessible, quelles que soient les opérations concurrentes*

Rely

Rely: « à tout moment, *pred* est accessible depuis Head selon un chemin trié »

Grâce à *Rely* je peux prouver que mon code est correct
- post-condition
- terminaison
- *guarantee*

Pourquoi il faut deux verrous : add || remove dangereux (Herlihy fig. 9.8+9.9)
Pas d'étreinte mortelle car tous les processus prennent verrous dans le même ordre

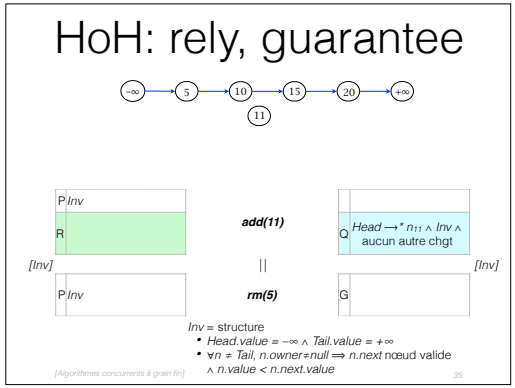
Établir post, *guarantee*

Supposons que :
- pré-condition vraie initialement
- *rely* vrai sur chaque pas atomique

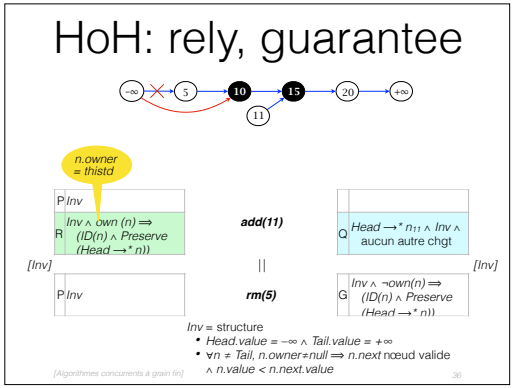
Montrer que mon processus promet : « tout nœud accessible depuis *pred* reste accessible (sauf éventuellement *curr* dans le cas de *remove*), et l'ordre de tri est respecté »

*** point de linéarisation

montrer avec variable fantôme (cf slides PPoPP)

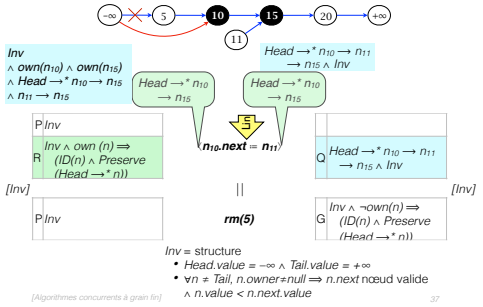


Regardons comment se passent un add et un remove concurrents.
Rely : un nœud verrouillé reste accessible, quelles que soient les opérations concurrentes.



Le processus 1 veut insérer n11 dans la liste et au même moment le processus 2 veut rendre n5 inaccessible.
Prenons comme hypothèse le Rely de P1.

HoH: rely, guarantee



HoH correct ?

Pas d'étreinte mortelle

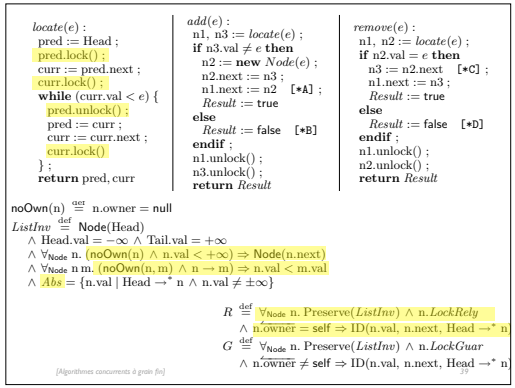
Pas de famine

Point de linéarisation :

- add réussi : au verrouillage de current
- add échoué : au verrouillage de pred
- remove réussi : au verrouillage de pred
- remove échoué : au verrouillage de current
- contains ?

Concentrons-nous sur le moment critique. Le 1er processus a déjà fait son parcours HoH, verrouillant les nœuds 10 et 15 entre lesquels doit se faire l'insertion. Le nœud 11 a déjà été alloué et initialisé.

L'instruction suivante va faire l'affectation (atomique) de n10.next ; il s'agit du point de linéarisation.
Puisque n10 et n15 sont verrouillés, et qu'on a vérifié par le parcours qu'ils étaient accessibles, le Rely dit qu'ils continuent à rester accessibles.
Après l'affectation, puisque n10 reste accessible, maintenant n11 est accessible, et n15 le reste.
Je peux donc déduire le Guarantee et la post-condition.



```
locate(e) :
  pred := Head ;
  pred.lock() ;
  curr := pred.next ;
  curr.lock() ;
  while (curr.val < e) {
    pred.unlock() ;
    pred := curr ;
    curr := curr.next ;
    curr.lock()
  } ;
  return pred, curr
```

```
add(e) :
  n1, n3 := locate(e) ;
  if n3.val ≠ e then
    n2 := new Node(e) ;
    n2.next := n3 ;
    n1.next := n2   [*A] ;
    Result := true
  else
    Result := false   [*B]
  endif ;
  n1.unlock() ;
  n3.unlock() ;
  return Result
```

```
remove(e) :
  n1, n2 := locate(e) ;
  if n2.val = e then
    n3 := n2.next   [*C] ;
    n1.next := n3 ;
    Result := true
  else
    Result := false   [*D]
  endif ;
  n1.unlock() ;
  n2.unlock() ;
  return Result
```

$$\mathsf{noOwn}(n) \stackrel{\text{def}}{=} n.\text{owner} = \text{null}$$

$$\begin{aligned} \mathit{ListInv} \stackrel{\text{def}}{=}\ & \mathsf{Node}(\text{Head}) \\ & \wedge\ \text{Head.val} = -\infty \wedge \text{Tail.val} = +\infty \\ & \wedge\ \forall_{\mathsf{Node}}\, n.\ (\mathsf{noOwn}(n) \wedge n.\text{val} < +\infty) \Rightarrow \mathsf{Node}(n.\text{next}) \\ & \wedge\ \forall_{\mathsf{Node}}\, n\, m.\ (\mathsf{noOwn}(n, m) \wedge n \rightarrow m) \Rightarrow n.\text{val} < m.\text{val} \\ & \wedge\ \mathit{Abs} = \{n.\text{val} \mid \text{Head} \rightarrow^* n \wedge n.\text{val} \neq \pm\infty\} \end{aligned}$$

$$\begin{aligned} R \stackrel{\text{def}}{=}\ & \forall_{\mathsf{Node}}\, n.\ \text{Preserve}(\mathit{ListInv}) \wedge n.\mathit{LockRely} \\ & \wedge\ \overleftarrow{n.\text{owner}} = \mathsf{self} \Rightarrow \text{ID}(n.\text{val}, n.\text{next}, \text{Head} \rightarrow^* n \\ G \stackrel{\text{def}}{=}\ & \forall_{\mathsf{Node}}\, n.\ \text{Preserve}(\mathit{ListInv}) \wedge n.\mathit{LockGuar} \\ & \wedge\ \overleftarrow{n.\text{owner}} \neq \mathsf{self} \Rightarrow \text{ID}(n.\text{val}, n.\text{next}, \text{Head} \rightarrow^* n \end{aligned}$$

extrait article Vafeiadis
locate : hand-over-hand lock
Inv : un nœud et son successeur non verrouillés sont ordonnés
R : un nœud verrouillé accessible le reste

Liste optimiste

Rely : un nœud verrouillé reste accessible
Mais verrouillage bloque autres processus
Assurer *Rely* avec moins de verrous ?

Solution :

- Traverser sans verrouiller
- Une fois position trouvée, verrouiller
- *Valider* : traverser de nouveau, vérifier accessible, non modifié
- Si échec, recommencer à zéro
  Normalement, cela arrive rarement

Pq revalidation nécessaire : cf Herlihy fig. 9.15
Trouver la solution ensemble
Animer au tableau
Regarder le code ensemble

Liste optimiste

Verrouiller moins :

- Traverser sans ver
- Une fois position t

Que pourrait-il se passer :

1. parcours sans verrou atteint n_p, n_c
2. n_p ou n_c retiré de la liste
3. verrouillage de n_p et n_c

Solution :

- *Valider* : traverser de nouveau
- Cela vérifie n_p et n_c accessibles
- *Rely*: un nœud accessible verrouillé ne sera pas modifié
- Si échec, recommencer à zéro

Normalement, cela arrive rarement

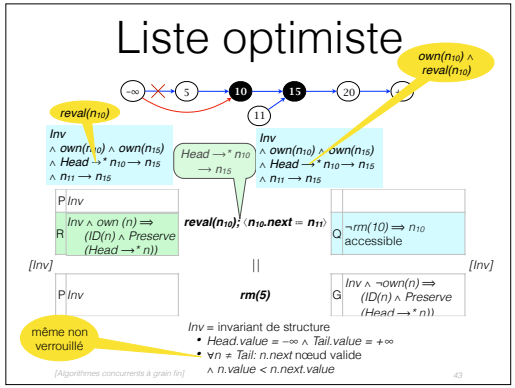
*** contains sans verrou?

Montrer (exemple) pq contains sans verrou n'est pas linéarisable

Dans ce cas la validation échouerait

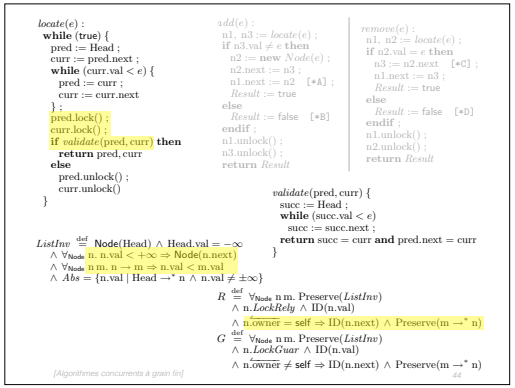
La validation montre que n_p et n_c accessibles

Rely: un nœud accessible verrouillé ne sera pas modifié



Montrons que la validation de n10 est correcte.
On sait qu'un nœud validé *est ou bien a été accessible* ; mais il est possible qu'il ne le soit plus, cf. n5 !. Comme n10 est verrouillé, il ne peut pas être retiré (grâce au double verrouillage). Donc, par *rely*, n10 est toujours accessible (le fait de valider n10 montre qu'il est resté accessible depuis le moment du verrouillage). Et donc la post-condition de la validation est que n10 est accessible, et donc la garantie reste vraie.

La preuve pour l'instruction suivante est inchangée.



Pseudocode (tiré de l'article de Vafeiadis).
add, remove sans changement par rapport à HoH
locate: parcours, verrouillage, validation
validate: vérifie qu'on arrive au même endroit, sans changement
ListInv: vrai pour tout nœud, pas seulement nœud déverrouillé
R: si accessible et verrouillé (⟹ non retiré), reste accessible. Pas suffisant à mon avis !!! car ne montre pas que la revalidation est correcte, et n'explique pas non plus pourquoi contains doit prendre le verrou.

Liste paresseuse

Répartition habituelle :
- 10 % add
- 5% remove
- 85 % contains

Objectifs :
- contains sans verrou
- autres : traversée une seule fois

remove non atomique ; linéarisabilité ?

Solution :
- positionner un booléen de marquage
- enlever de la liste ultérieurement
- parcours vérifie que nœud non marqué ; sinon recommencer à zéro

*** contains sans verrou

montrer (exemple) que contains sans verrou est linéarisable (point de linéarisation = marquage)

Si on visite un nœud c'est qu'il a été ajouté par add (atomique)
Par contre, il peut avoir fait l'objet d'un remove : non atomique
Rendre remove atomique

Trouver la solution ensemble
Animer au tableau
Regarder le code ensemble
• recommencer à zéro ==> while true

Liste paresseuse : correcte ?

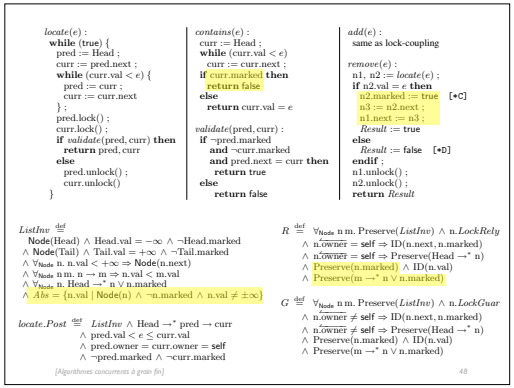
Rely : tout nœud non marqué est accessible

Relation d'abstraction : tout nœud non marqué est dans l'ensemble

Famine !

Point de linéarisation
- *remove* : positionner marquage
- *contains*: c'est compliqué…

linéarisation de "contains" : cf Herlihy fig 9.21



```
locate(e) :
  while (true) {
    pred := Head ;
    curr := pred.next ;
    while (curr.val < e) {
      pred := curr ;
      curr := curr.next
    } ;
    pred.lock() ;
    curr.lock() ;
    if validate(pred, curr) then
      return pred, curr
    else
      pred.unlock() ;
      curr.unlock()
  }
```

```
contains(e) :
  curr := Head ;
  while (curr.val < e)
    curr := curr.next ;
  if curr.marked then
    return false
  else
    return curr.val = e

validate(pred, curr) :
  if ¬pred.marked
     and ¬curr.marked
     and pred.next = curr then
       return true
  else
       return false
```

```
add(e) :
  same as lock-coupling

remove(e) :
  n1, n2 := locate(e) ;
  if n2.val = e then
    n2.marked := true    [*C]
    n3 := n2.next ;
    n1.next := n3 ;
    Result := true
  else
    Result := false    [*D]
  endif ;
  n1.unlock() ;
  n2.unlock() ;
  return Result
```

$$ListInv \stackrel{\text{def}}{=} \mathsf{Node}(\mathrm{Head}) \wedge \mathrm{Head.val} = -\infty \wedge \neg \mathrm{Head.marked} \wedge \mathsf{Node}(\mathrm{Tail}) \wedge \mathrm{Tail.val} = +\infty \wedge \neg \mathrm{Tail.marked} \wedge \forall_{\mathsf{Node}}\, \mathrm{n}.\ \mathrm{n.val} < +\infty \Rightarrow \mathsf{Node}(\mathrm{n.next}) \wedge \forall_{\mathsf{Node}}\, \mathrm{n\,m}.\ \mathrm{n} \rightarrow \mathrm{m} \Rightarrow \mathrm{n.val} < \mathrm{m.val} \wedge \forall_{\mathsf{Node}}\, \mathrm{n}.\ \mathrm{Head} \rightarrow^* \mathrm{n} \vee \mathrm{n.marked} \wedge \mathit{Abs} = \{\mathrm{n.val} \mid \mathsf{Node}(\mathrm{n}) \wedge \neg \mathrm{n.marked} \wedge \mathrm{n.val} \neq \pm\infty\}$$

$$locate.Post \stackrel{\text{def}}{=} ListInv \wedge \mathrm{Head} \rightarrow^* \mathrm{pred} \rightarrow \mathrm{curr} \wedge \mathrm{pred.val} < e \leq \mathrm{curr.val} \wedge \mathrm{pred.owner} = \mathrm{curr.owner} = \mathsf{self} \wedge \neg \mathrm{pred.marked} \wedge \neg \mathrm{curr.marked}$$

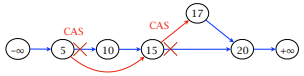
$$R \stackrel{\text{def}}{=} \forall_{\mathsf{Node}}\, \mathrm{n\,m}.\ \mathrm{Preserve}(ListInv) \wedge \mathrm{n}.LockRely \wedge \overline{\mathrm{n.owner}} = \mathsf{self} \Rightarrow \mathrm{ID(n.next, n.marked)} \wedge \overline{\mathrm{n.owner}} = \mathsf{self} \Rightarrow \mathrm{Preserve(Head} \rightarrow^* \mathrm{n)} \wedge \mathrm{Preserve(n.marked)} \wedge \mathrm{ID(n.val)} \wedge \mathrm{Preserve(n} \rightarrow^* \mathrm{n} \vee \mathrm{n.marked)}$$

$$G \stackrel{\text{def}}{=} \forall_{\mathsf{Node}}\, \mathrm{n\,m}.\ \mathrm{Preserve}(ListInv) \wedge \mathrm{n}.LockGuar \wedge \overline{\mathrm{n.owner}} \neq \mathsf{self} \Rightarrow \mathrm{ID(n.next, n.marked)} \wedge \overline{\mathrm{n.owner}} \neq \mathsf{self} \Rightarrow \mathrm{Preserve(Head} \rightarrow^* \mathrm{n)} \wedge \mathrm{Preserve(n.marked)} \wedge \mathrm{ID(n.val)} \wedge \mathrm{Preserve(m} \rightarrow^* \mathrm{n} \vee \mathrm{n.marked)}$$

Pseudocode (tiré de l'article de Vafeiadis).

Liste sans verrous

Objectif : éliminer les verrous complètement
- Affectation de pointeurs par CAS
- Si conflit : échec, recommencer à zéro

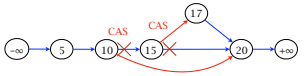


Trouver la solution ensemble
Animer au tableau
Regarder le code ensemble
• màj pointeur+booléen = while true
• aide = while true

Liste sans verrous

Objectif : éliminer les verrous complètement
- Affectation de pointeurs par CAS
- Si conflit : échec, recommencer à zéro

Problème : remove concurrent avec add



Solution :
- CAS booléen de marquage *current* = 15
- CAS pointeur *pred* = 10
- si modifié, échec : recommencer

Trouver la solution ensemble
Animer au tableau
Regarder le code ensemble
• màj pointeur+booléen = while true
• aide = while true

Liste sans verrous

Objectif : éliminer les verrous complètement
- Affectation de pointeurs par CAS
- Si conflit : échec, recommencer à zéro

Problème : remove en deux CAS

Solution :
- CAS à la fois booléen de marquage, pointeur
- Si modifié : échec, recommencer
- DCAS ?
- AtomicMarkableReference
- Enlever nœud marqué de la liste: aider les autres processus

Trouver la solution ensemble
Animer au tableau
Regarder le code ensemble
• màj pointeur+booléen = while true
• aide = while true

Marqueur et pointeur doivent être testés de façon atomique ! Cf Herlihy fig. 9.22

*** expliquer

pourquoi findAndCompress retire physiquement nœuds marqués

cf Herlihy p. 217 et fig 9.22

On ne peut pas modifier le *next* d'un nœud marqué. Il faut donc retirer ce nœud de la liste au plus tôt.

```
public boolean add (int v) {
    boolean retval = false;
    assert rep ();
    assert valueOK (v);
    boolean wasIn = concurrent || internal_contains (v);
    // head ----> -14 --> -2 -----> 3 --> 7 --> 99 --> tail
    //   |--> -22 --^       |--> 0 --^
    try {
        restart: // if CAS fails, start again from scratch
        while (true) {
            Window w = findAndCompress (head, v);
            Node pred = w.left;
            Node current = w.right;
            assert v <= current.value;
            if (v == current.value) {
                retval = false;
                return retval;
            }
            assert v < current.value;
            Node n = new Node(v, current);
            if (!pred.IgnoredPlusNext.compareAndSet (current, n,
                false, false))
                continue restart;
            retval = true;
            return retval;
        }
    } finally {
        assert rep ();
        assert concurrent || internal_contains (v);
        assert concurrent || retval  == !wasIn;
    }
}
```


Java code

"add" uses a CAS to insert the new node. CAS may fail either if the current node has been marked deleted, or if its next pointer has changed. Either way, restart the loop from the beginning.

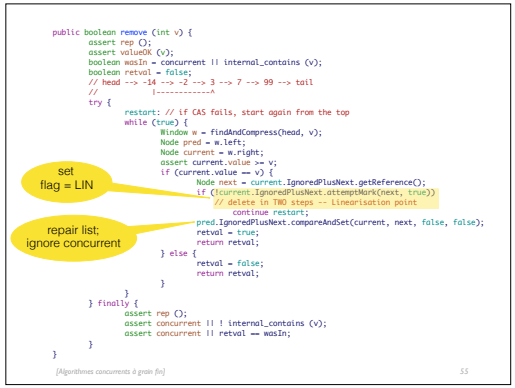
```
private Window findAndCompress (Node head, int value) {
    Node pred;
    boolean[] ignored = { false }; // is this node ignored?
    restart: // if compareAndSet fails, must start again from the
        beginning
        while (true) {
            pred = head;
            Node current = pred.IgnoredPlusNext.getReference();
            Node next;
            while (true) {
                next = current.IgnoredPlusNext.get(ignored);
                while (ignored[0]) { // 'current' is ignored: strip
                    it from the list if I can
                    if (!pred.IgnoredPlusNext.compareAndSet(current,
                        next, false, false)) {
                        // failed: start over from the top
                        continue restart;
                    }
                    current = next;
                    next = current.IgnoredPlusNext.getReference();
                }
                if (current == tail) return new Window (pred, current
                    ); // may happen if value == TAILVAL
                assert concurrent || !current.IgnoredPlusNext.
                    isMarked(); // hopefully!! could have changed by
                    the time this assert is reached
                assert pred == head || pred.value > HEADVAL;
                assert pred.value < TAILVAL;
                assert current != null;
                assert pred.value < current.value;
                if (value <= current.value)
                    return new Window(pred, current);
                pred = current;
                current = next;
            }
        }
}
```

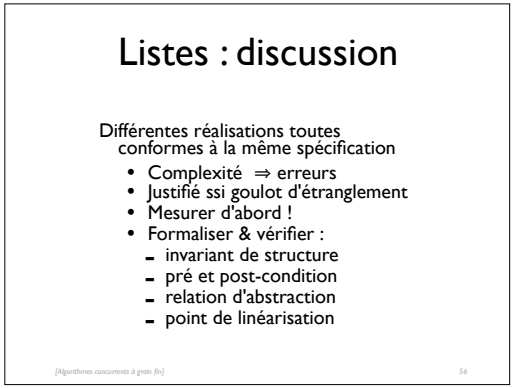
help other threads

"findAndCompress" walks the list and returns pred and current.

Normally a list-walk makes no changes: but surprisingly it contains a CAS! If this thread finds that a node is marked, this means that some concurrent thread marked it but has not finished removing it from the list; who knows how long it might remain in this state. Therefore this thread "helps" the other one by removing it ourselves.
(Why is it important to do this? Has to do with re-adding a value that has just been removed???)

As a side effect, findAndCompress checks the structural invariant (all the asserts at the end). This violates my methodology of checking the invariant with a test to *rep()*, but I couldn't think of a better way.





"attemptMark" is a first CAS to atomically update the "ignored" bit. This makes this node deleted. If CAS fails, try again.
"CompareAndSet" is a second CAS to atomically update the pointer + ignored bit. CAS may fail because a concurrent thread has helped in the meantime, but this does not matter; just ignore the return value.

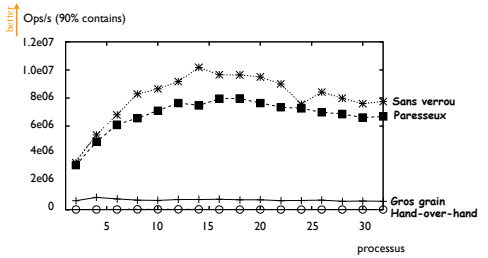
Performance

Machine 16 cœurs

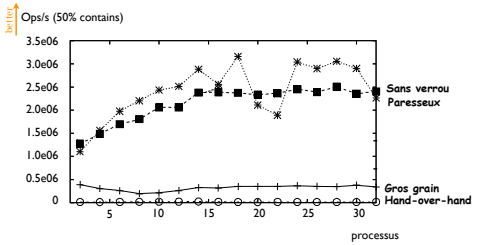
Banc d'essai synthétique

- Java
- Ensemble mis en œuvre par liste chaînée
- varier % *contains()*

Beaucoup de *contains*



Peu de *contains*



Varier % *contains*