

Structures non bloquantes

Gaël Thomas

gael.thomas@lip6.fr

Basé sur le cours de Herlihy & Shavit

Université Pierre et Marie Curie
Master Informatique
M2 – Spécialité SAR

Solution naturelle

Construire des algorithmes qui ne bloquent pas (du plus fort au plus faible) :

- ✓ **Wait-free** : toute opération se termine en un nombre fini de pas (i.e., pas de blocage d'un thread, pas de famine)
- ✓ **Lock-free** : si les opérations sont appelées infiniment souvent, elles se terminent infiniment souvent (pas de blocage du programme, même si certains threads ne sont jamais élus)
- ✓ **Obstruction-free** : tout thread s'exécutant seul termine son opération en un nombre fini de pas (impliqué par lock-free, choisir de ne laisser qu'un thread tourner et appeler infiniment souvent une opération)

Algorithme à verrou : pas obstruction free (si verrou pris, bloque les autres threads)

Limite à l'accélération

Limite : la loi d'Amdahl

p : pourcentage du code exécutable en parallèle
 $\Rightarrow (1 - p)$: pourcentage du code exécuté en séquentiel
 $\Rightarrow p/n$: exécution du code parallèle sur n cœurs
 $\Rightarrow a = 1/(1 - p + p/n)$: accélération maximale théorique
 \Rightarrow limite pour $n \rightarrow \infty$: $a \rightarrow 1/(1 - p)$

Application numérique :

$p = 0,25 \Rightarrow a \rightarrow 1/0,75 = 4$ quand $n \rightarrow \infty$ (3,7 à 32 cœurs)

$p = 0,95 \Rightarrow a \rightarrow 1/0,95 = 20$ quand $n \rightarrow \infty$ (12,55 à 32 cœurs, 17,42 à 128 cœurs)

\Rightarrow ça vaut le coup de se battre pour paralléliser les quelques pourcents restants!

Structures non bloquantes

1. Les outils
2. La pile
3. La queue
4. La liste chaînée
5. Algorithmes de verrouillage

Les outils : les instructions atomique

Instructions atomiques

- ✓ Atomic add : addition atomique en mémoire
old-value \leftarrow **atomic-add**(variable, value)
- ✓ Atomic swap : écrit une valeur en mémoire et renvoie son ancienne valeur
old-value \leftarrow **atomic-swap**(variable, value)
- ✓ Atomic compare and swap (CAS) : compare avant de faire un swap
old-value \leftarrow **atomic-CAS**(variable, old, new)

```
if(variable == old) {
    variable = new;
    return old;
} else
    return value;
```

18/11/13

Mémoires Transactionnelles

5

Les outils : les modèles mémoire

Modèle mémoire : définit les ordonnancements possible des accès mémoire

Principe commun à tous les modèles mémoire (utile!)

- ✓ En locale, toute lecture succédant à une écriture lit la dernière valeur écrite

```
write x, value
read x -> lit la valeur value
```

18/11/13

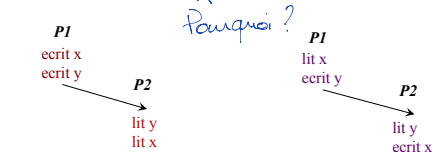
Mémoires Transactionnelles

6

Les outils : le modèle mémoire du pentium

Le Total Store Order (TSO)

- ✓ Les lectures ne sont jamais réordonnées entre elles
- ✓ Les écritures ne sont jamais réordonnées entre elles
- ✓ Une écriture après une lecture n'est pas réordonnée
- ✓ Une lecture après une écriture peut être réordonnée si il s'agit de deux cases mémoire distinctes



Lectures non réordonnées
et écritures non réordonnées
Lit forcément la valeur x écrite

Écriture après lecture
Impossible de lire la valeur x écrite
à la fin

18/11/13

Mémoires Transactionnelles

7

Les outils : le modèle mémoire du pentium

Le cache assure une cohérence séquentielle
LTSO
assure que le CPU a les dernières intr. au cache dans le bon ordre.

Attention à la lecture après écriture sur deux cases distinctes en TSO

Initialement : $t1 = t2 = 0$

P2
écrit y, 1
écrit x, 1

P3
 $t2 = \text{lit } x$

P1
écrit x, 2
 $t1 = \text{lit } y$

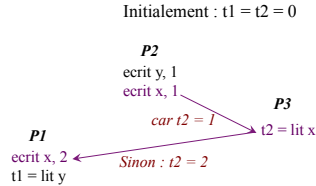
18/11/13

Mémoires Transactionnelles

8

Les outils : le modèle mémoire du pentium

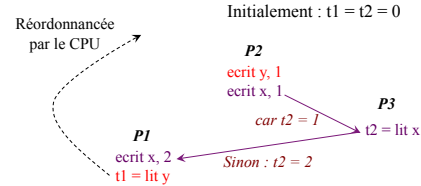
Attention à la lecture après écriture sur deux cases distinctes en TSO



Sortie autorisée : $t2 = 1$

Les outils : le modèle mémoire du pentium

Attention à la lecture après écriture sur deux cases distinctes en TSO



Sortie autorisée : $t1 = 0, t2 = 1$

Instructions atomiques du pentium

Instructions atomiques

- ✓ Atomic add : addition atomique en mémoire
old-value \leftarrow **atomic-add**(variable, value)
assembleur : **lock xadd**
- ✓ Atomic swap : écrit une valeur en mémoire et renvoie son ancienne valeur
old-value \leftarrow **atomic-swap**(variable, value)
assembleur : **xchg** (génère un **lock** assembleur)
- ✓ Atomic compare and swap (CAS) : compare avant de faire un swap
old-value \leftarrow **atomic-CAS**(variable, old, new)
assembleur : **lock cmpxchg**

Les outils : le modèle mémoire du pentium

Deux instructions particulières

- ✓ La barrière mémoire (mfence) : toutes les lectures et écritures précédentes sont visibles
Pour éviter un ré-ordonnement d'une lecture après écriture, mettre un mfence
 - ☞ Exécute toutes les écritures en attente
 - ☞ Exécute toutes les lectures attente \Rightarrow vide le buffer des écritures et des lectures en attente
- ✓ Le lock (préfixe d'instruction) : un mfence + visibilité immédiate de l'écriture de l'instruction préfixée

Les outils : le modèle mémoire du C

Pas de modèle mémoire avant C11

Instructions atomiques de gcc

- ✓ Atomic add : addition atomique en mémoire
`old-value ← atomic-add(variable, value)`
`gcc : __sync_fetch_and_add`
- ✓ Atomic swap : écrit une valeur en mémoire et renvoie son ancienne valeur
`old-value ← atomic-swap(variable, value)`
`gcc : __sync_lock_test_and_set (vraiment bizarre!)`
- ✓ Atomic compare and swap (CAS) : compare avant de faire un swap
`old-value ← atomic-CAS(variable, old, new)`
`gcc : __sync_val_compare_and_swap`
- ✓ Barrière mémoire
`__sync_synchronize()`

18/11/13

Mémoires Transactionnelles

13

Le modèle mémoire Java

Modèle mémoire défini depuis Java 5

Tout réordonnancement possible sauf

- ✓ Variable volatile : barrière mémoire à chaque accès (équivalent lock)
- ✓ Entrée et sortie de section critique : barrière mémoire (équivalent lock)

18/11/13

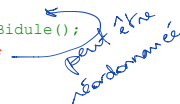
Mémoires Transactionnelles

14

Le modèle mémoire Java

Piège classique : éviter la prise de verrou lors d'une initialisation

```
if(!inited) {  
    synchronize(obj) {  
        if(!inited) {  
            bidule = new Bidule();  
            inited = true;  
        }  
    }  
}
```



⇒ peut voir inited = true alors que bidule n'a pas été initialisé

Solution : marquer inited comme variable volatile
Remarque : impossible en TSO, sauf si le compilateur optimise le code...

18/11/13

Mémoires Transactionnelles

15

Structures non bloquantes

1. Les outils
2. La pile
3. La queue
4. La liste chaînée
5. Algorithmes de verrouillage

18/11/13

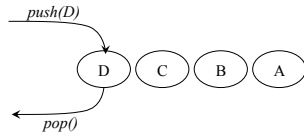
Mémoires Transactionnelles

16

Le b.a.-ba : la pile

Deux opérations en mode LIFO

- ✓ push(Element e) : empile un élément
- ✓ Element pop() : dépile un élément



18/11/13

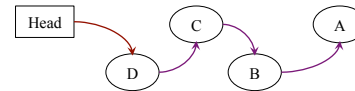
Mémoires Transactionnelles

17

La pile avec verrou

```
Class Stack {
    Node head;
}

Class Node {
    Node next;
    Element element;
}
```



18/11/13

Mémoires Transactionnelles

18

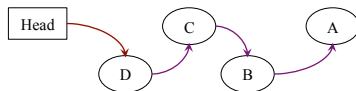
La pile avec verrou

```
Class Stack {
    Node head;
}

Class Node {
    Node next;
    Element element;
}

synchronized void Stack.push(Element element) {
    Node n = new Node(head, element);
    head = n;
}

synchronized Element pop() {
    Node n = head;
    head = n.next;
    return n.element;
}
```



18/11/13

Mémoires Transactionnelles

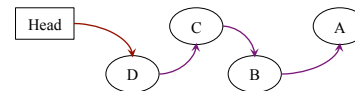
19

La pile sans verrou (Scott 91)

```
Class Stack {
    Node head;
}

Class Node {
    Node next;
    Element element;
}
```

Principe : atomic-cas sur la tête



18/11/13

Mémoires Transactionnelles

20

La pile sans verrou (Scott 91)

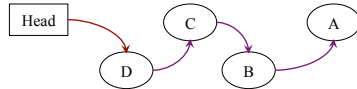
```

Class Stack {
    Node head;
}

Class Node {
    Node next;
    Element element;
}

void Stack.push(Element element) {
    do {
        Node n = new Node(head, element);
    } while (atomic-cas(&head, n.next, n) != n.next);
}

```



La pile sans verrou (Scott 91)

Obstruction-free : quelque soit l'instant, si on interrompt les threads sauf un, dans le pire cas, deux tours de boucle pour le thread qui tourne

Lock-free: si les threads appellent infiniment souvent push ou pop, elles seront exécutées infiniment souvent (preuve : il faut réussir pop ou push pour bloquer un pop ou push)

Pas wait-free : un push peut être retardé indéfiniment par d'autres push.

```
void Stack.push(Element element) {
    do {
        Node n = new Node(head, element);
    } while (atomic-cas(&head, n.next, n) != n);
}

Element Stack.pop() {
    do {
        Node n = head;
        if (n == null) error("No such element");
    } while (atomic-cas(&head, n, n.next) != n);
    return n.element;
}
```

La pile sans verrou (Scott 91)

```

Class Stack {
    Node head;
}
—

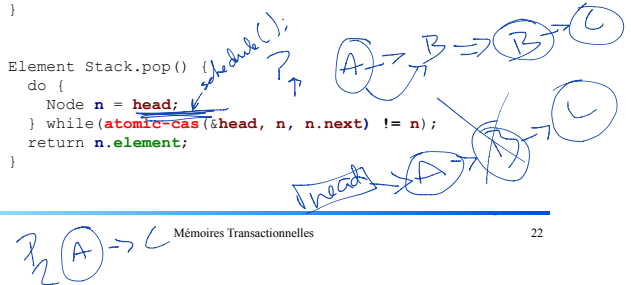
void Stack.push(Element element) {
    do {
        Node n = new Node(head, element);
    } while (atomic-cas(&head, n.next, n) != n.next);
}
>

```

```

Element Stack.pop() {
    do {
        Node n = head;
    } while(atomic-cas(head, n, n.next) != n);
    return n.element;
}

```



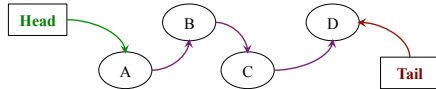
Structures non bloquantes

1. Les outils
2. La pile
3. **La queue**
4. La liste chaînée
5. Algorithmes de verrouillage

La queue

Deux opérations :

- ✓ void enqueue(Element e) : ajoute l'élément en queue
- ✓ Element dequeue() : dépile l'élément en tête



18/11/13

Mémoires Transactionnelles

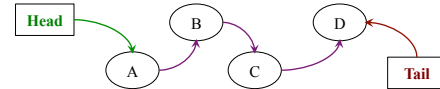
25

La queue avec verrou

```

Class Queue {
    Node head = null;
    Node tail = null;
}

Class Node {
    Node next;
    Element element;
}
  
```



18/11/13

Mémoires Transactionnelles

26

La queue avec verrou

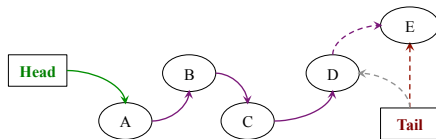
```

Class Queue {
    Node head = null;
    Node tail = null;
}

Class Node {
    Node next;
    Element element;
}
  
```

```

synchronized void Queue.enqueue(Element e) {
    Node n = new Node(null, e);
    if(tail != null) { tail.next = n; }
    else { head = n; }
    tail = n;
}
  
```



18/11/13

Mémoires Transactionnelles

27

La queue avec verrou

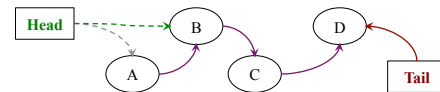
```

Class Queue {
    Node head = null;
    Node tail = null;
}

Class Node {
    Node next;
    Element element;
}
  
```

```

synchronized Element Queue.dequeue() {
    Node n = head;
    head = n.next;
    if(head == null) tail = null;
    return n.element;
}
  
```



18/11/13

Mémoires Transactionnelles

28

La queue sans verrou

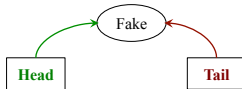
```

Class Queue {
    Node head = new Node(null, null);
    Node tail = head;
}

Class Node {
    Node next;
    Element element;
}
    
```

Principes :

- ✓ Un faux nœud pour gérer le cas où la liste est vide
- ✓ Tail mis à jour de façon paresseuse
- ✓ A chaque instant :
 - ☞ Le premier nœud (si il existe) est en seconde position
 - ☞ Les listes tail et head se rejoignent
 - ☞ Le dernier nœud de ces listes est le nœud en queue



18/11/13

Mémoires Transactionnelles

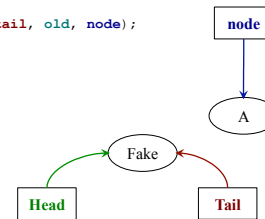
29

La queue sans verrou

Insertion de A en l'absence de concurrence

```

void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while (old.next != NULL) {
            CAS(&tail, old, old.next);
        } while (CAS(&old.next, null, node) != null);
        CAS(&tail, old, node);
    }
}
    
```



18/11/13

Mémoires Transactionnelles

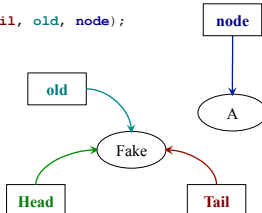
30

La queue sans verrou

Insertion de A en l'absence de concurrence

```

void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while (old.next != NULL) {
            CAS(&tail, old, old.next);
        } while (CAS(&old.next, null, node) != null);
        CAS(&tail, old, node);
    }
}
    
```



18/11/13

Mémoires Transactionnelles

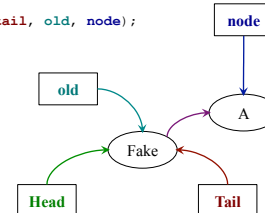
31

La queue sans verrou

Insertion de A en l'absence de concurrence

```

void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while (old.next != NULL) {
            CAS(&tail, old, old.next);
        } while (CAS(&old.next, null, node) != null);
        CAS(&tail, old, node);
    }
}
    
```



18/11/13

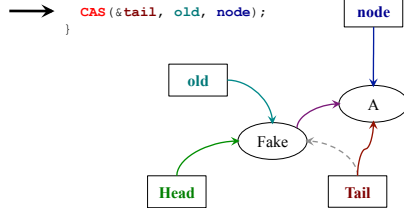
Mémoires Transactionnelles

32

La queue sans verrou

Insertion de A en l'absence de concurrence

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while (old.next != NULL) { old = tail; }
        CAS(&tail, old, old.next);
    } while (CAS(&old.next, null, node) != null);
}
```



18/11/13

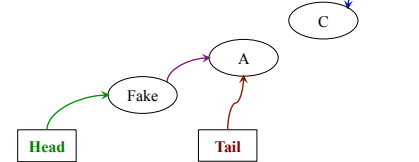
Mémoires Transactionnelles

33

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while (old.next != NULL) { old = tail; }
        CAS(&tail, old, old.next);
    } while (CAS(&old.next, null, node) != null);
    CAS(&tail, old, node);
}
```



18/11/13

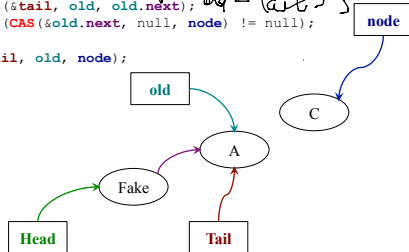
Mémoires Transactionnelles

34

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while (old.next != NULL) { old = tail; }
        CAS(&tail, old, old.next);
    } while (CAS(&old.next, null, node) != null);
    CAS(&tail, old, node);
}
```



18/11/13

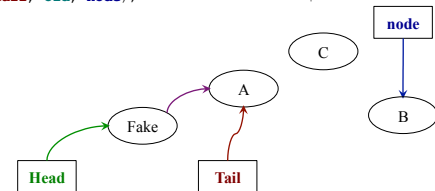
Mémoires Transactionnelles

35

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while (old.next != NULL) { old = tail; }
        CAS(&tail, old, old.next);
    } while (CAS(&old.next, null, node) != null);
    CAS(&tail, old, node);
}
```



18/11/13

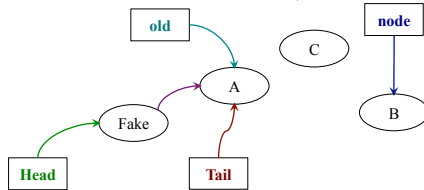
Mémoires Transactionnelles

36

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while (old.next != NULL) {
            CAS(&tail, old, old.next);
        } while (CAS(&old.next, null, node) != null);
    } while (CAS(&tail, old, node));
}
```



18/11/13

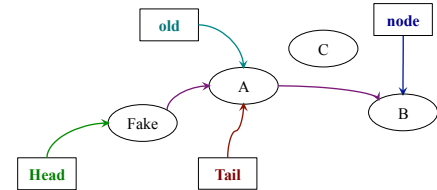
Mémoires Transactionnelles

37

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while (old.next != NULL) {
            CAS(&tail, old, old.next);
        } while (CAS(&old.next, null, node) != null);
    } while (CAS(&tail, old, node));
}
```



18/11/13

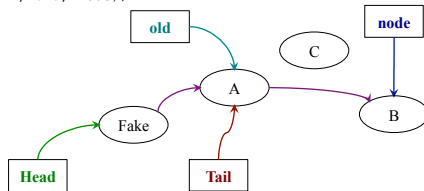
Mémoires Transactionnelles

38

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while (old.next != NULL) {
            CAS(&tail, old, old.next);
        } while (CAS(&old.next, null, node) != null);
    } while (CAS(&tail, old, node));
}
```



18/11/13

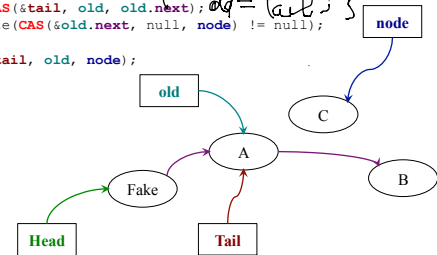
Mémoires Transactionnelles

39

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while (old.next != NULL) {
            CAS(&tail, old, old.next);
        } while (CAS(&old.next, null, node) != null);
    } while (CAS(&tail, old, node));
}
```



18/11/13

Mémoires Transactionnelles

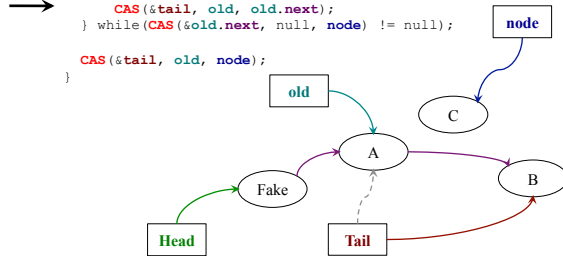
40

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while(old.next != NULL)
            CAS(&tail, old, old.next);
    } while(CAS(&old.next, null, node) != null);
    CAS(&tail, old, node);
}
```

C prend en charge l'avancement de tail pour B



18/11/13

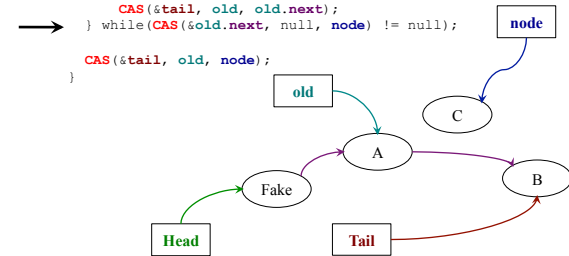
Mémoires Transactionnelles

41

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while(old.next != NULL)
            CAS(&tail, old, old.next);
    } while(CAS(&old.next, null, node) != null);
    CAS(&tail, old, node);
}
```



18/11/13

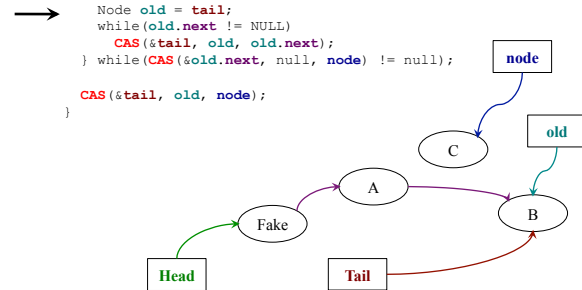
Mémoires Transactionnelles

42

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while(old.next != NULL)
            CAS(&tail, old, old.next);
    } while(CAS(&old.next, null, node) != null);
    CAS(&tail, old, node);
}
```



18/11/13

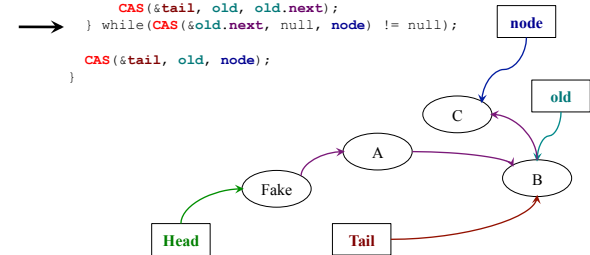
Mémoires Transactionnelles

43

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while(old.next != NULL)
            CAS(&tail, old, old.next);
    } while(CAS(&old.next, null, node) != null);
    CAS(&tail, old, node);
}
```



18/11/13

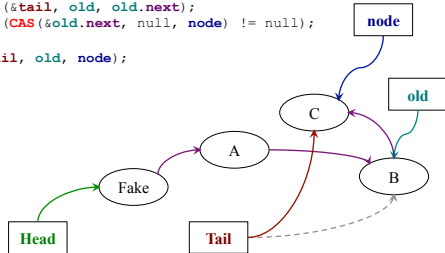
Mémoires Transactionnelles

44

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while(old.next != NULL)
            CAS(&tail, old, old.next);
    } while(CAS(&old.next, null, node) != null);
    → CAS(&tail, old, node);
}
```



18/11/13

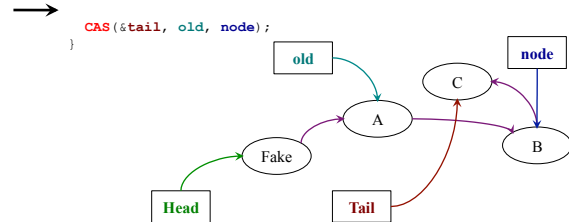
Mémoires Transactionnelles

45

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while(old.next != NULL)
            CAS(&tail, old, old.next);
    } while(CAS(&old.next, null, node) != null);
    → CAS(&tail, old, node);
}
```



18/11/13

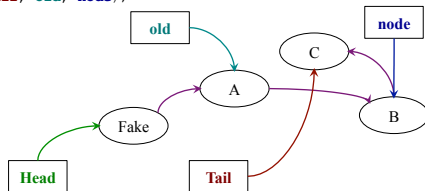
Mémoires Transactionnelles

46

La queue sans verrou

Insertion de C avec concurrence pendant l'insertion de B

```
void Queue.enqueue(Element e) {
    Node node = new Node(null, e);
    do {
        Node old = tail;
        while(old.next != NULL)
            CAS(&tail, old, old.next);
    } while(CAS(&old.next, null, node) != null);
    → CAS(&tail, old, node);
}
```



18/11/13

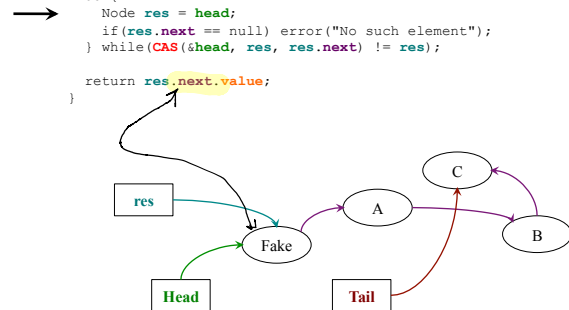
Mémoires Transactionnelles

47

La queue sans verrou

Suppression de la tête sans concurrence

```
Element Queue.dequeue() {
    do {
        Node res = head;
        if(res.next == null) error("No such element");
    } while(CAS(&head, res, res.next) != res);
    return res.next.value;
}
```



18/11/13

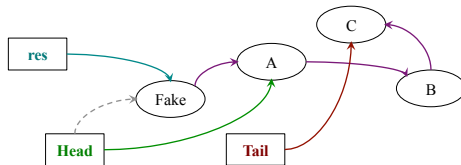
Mémoires Transactionnelles

48

La queue sans verrou

Suppression de la tête sans concurrence

```
Element Queue.dequeue() {  
  do {  
    Node res = head;  
    if(res.next == null) error("No such element");  
    } while(CAS(&head, res, res.next) != res);  
  }  
  return res.next.value;  
}
```



18/11/13

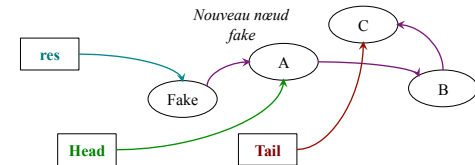
Mémoires Transactionnelles

49

La queue sans verrou

Suppression de la tête sans concurrence

```
Element Queue.dequeue() {  
  do {  
    Node res = head;  
    if(res.next == null) error("No such element");  
    } while(CAS(&head, res, res.next) != res);  
  }  
  return res.next.value;  
}
```



18/11/13

Mémoires Transactionnelles

50

La queue sans verrou

Obstruction-free : si un seul thread élu, finira par réussir à faire son enqueue ou dequeue

Lock-free : si les threads appellent infiniment souvent enqueue ou dequeue, au moins un passera de temps en temps (preuve : pour faire tourner un thread dans enqueue ou dequeue, il faut qu'il y ait des enqueue et dequeue qui terminent)

Pas wait-free : on peut faire tourner indéfiniment un thread dans une des boucles

18/11/13

Mémoires Transactionnelles

51

Structures non bloquantes

1. Les outils
2. La pile
3. La queue
4. La liste chaînée
5. Algorithmes de verrouillage

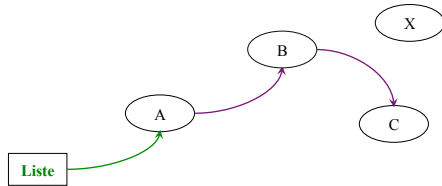
18/11/13

Mémoires Transactionnelles

52

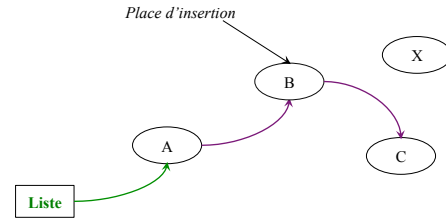
La liste chaînée

Le grand problème : insertion et suppression au même endroit



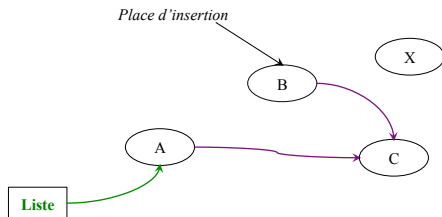
La liste chaînée

Le grand problème : insertion et suppression au même endroit



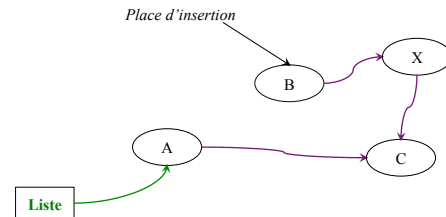
La liste chaînée

Le grand problème : insertion et suppression au même endroit



La liste chaînée

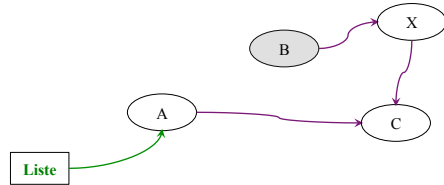
Le grand problème : insertion et suppression au même endroit



La liste chaînée

Principe de solution (Tim Harris, DISC 2001)

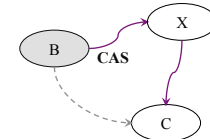
- ✓ Changer la couleur d'un nœud qui va être supprimé
- ✓ Si pendant le parcours ou pendant l'insertion la couleur du nœud d'insertion change, le supprimer et recommencer au début



La liste chaînée

Principe de solution (Tim Harris, DISC 2001)

- ✓ Changer la couleur d'un nœud qui va être supprimé
- ✓ Si pendant le parcours ou pendant l'insertion la couleur du nœud d'insertion change, le supprimer et recommencer au début



Problème :

- ✓ Lors de l'insertion, on ne peut faire qu'un CAS sur un next

Solution :

- ✓ Utiliser le bit de poids faible pour stocker la couleur dans le pointeur

La liste chaînée

```
typedef uintptr_t coloredPointer;
```

```
Node pointer(coloredPointer ptr) { return (Node)(ptr & ~2); }
```

```
int mark(coloredPointer ptr) { return ptr & 1; }
```

```
Class Node {
    coloredPointer next;
    Element element;
};
```

*retourne
une ptr
sans couleur*

Idée : avant de supprimer un nœud n, marquer n.next

- ✓ La liste est toujours connectée (i.e. tout nœud non supprimé est toujours dans la liste et elle est ordonné)
- ✓ Des nœuds marqués "à supprimer" peuvent être dans la liste

La liste chaînée triée d'entier : le parcours

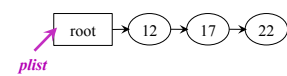
```
void del(coloredPointer* plist, int value) {
```

```
    restart:
```

```
    coloredPointer* pred = plist;
```

```
    while(!found) {
```

```
        Node cur = pointer(*pred);
```



```
        pred = &cur->next;
    } }
```

La liste chaînée triée d'entier : le parcours

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



```
    pred = &cur->next;
  } }
```

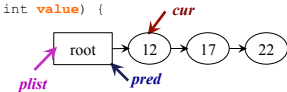
18/11/13

Mémoires Transactionnelles

61

La liste chaînée triée d'entier : le parcours

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



```
    pred = &cur->next;
  } }
```

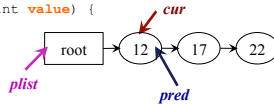
18/11/13

Mémoires Transactionnelles

62

La liste chaînée triée d'entier : le parcours

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



```
    pred = &cur->next;
  } }
```

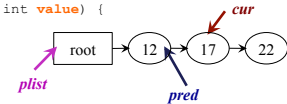
18/11/13

Mémoires Transactionnelles

63

La liste chaînée triée d'entier : le parcours

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



```
    pred = &cur->next;
  } }
```

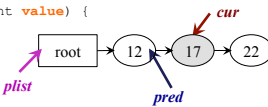
18/11/13

Mémoires Transactionnelles

64

La liste chaînée triée d'entier : compaction

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



Principe : supprime les nœuds marqués, même si pas le propriétaire

```
    if(mark(cur->next)) {                /* cur doit être supprimé */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
    } else continue;
  }
  pred = &cur->next;
}
```

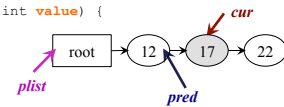
18/11/13

Mémoires Transactionnelles

65

La liste chaînée triée d'entier : compaction

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



Principe : supprime les nœuds marqués, même si pas le propriétaire

```
    if(mark(cur->next)) {                /* cur doit être supprimé */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
    } else continue;
  }
  pred = &cur->next;
}
```

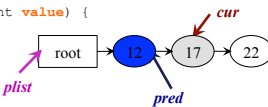
18/11/13

Mémoires Transactionnelles

66

La liste chaînée triée d'entier : compaction

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



Principe : supprime les nœuds marqués, même si pas le propriétaire

```
    if(mark(cur->next)) {                /* cur doit être supprimé */
    → if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
    } else continue;
  }
  pred = &cur->next;
}
```

Deux raisons pour rater le CAS :
 Si pred est marqué supprimé
 Si un nœud est inséré après pred
 Dans le doute, recommence le parcours

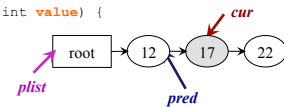
18/11/13

Mémoires Transactionnelles

67

La liste chaînée triée d'entier : compaction

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



Principe : supprime les nœuds marqués, même si pas le propriétaire

```
    if(mark(cur->next)) {                /* cur doit être supprimé */
    → if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
    } else continue;
  }
  pred = &cur->next;
}
```

Deux raisons pour rater le CAS :
 Si pred est marqué supprimé
 Si un nœud est inséré après pred
 Dans le doute, recommence le parcours

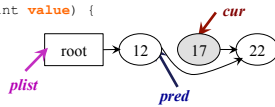
18/11/13

Mémoires Transactionnelles

68

La liste chaînée triée d'entier : compaction

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



Principe : supprime les nœuds marqués, même si pas le propriétaire

```
    if(mark(cur->next)) {                /* cur doit être supprimé */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
    } else continue;
  }
  pred = &cur->next;
}
```

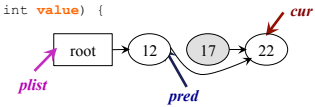
18/11/13

Mémoires Transactionnelles

69

La liste chaînée triée d'entier : compaction

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



Principe : supprime les nœuds marqués, même si pas le propriétaire

```
    if(mark(cur->next)) {                /* cur doit être supprimé */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
    } else continue;
  }
  pred = &cur->next;
}
```

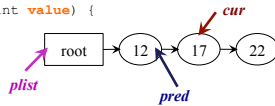
18/11/13

Mémoires Transactionnelles

70

La liste chaînée triée d'entier : la suppression

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



On suppose value = 17

```
    if(cur->value == value) {             /* trouvé! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1;
    }

    if(mark(cur->next)) {                 /* cur doit être supprimé */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
    } else continue;
  }
  pred = &cur->next;
}
```

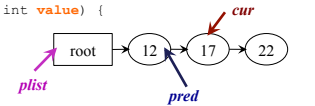
18/11/13

Mémoires Transactionnelles

71

La liste chaînée triée d'entier : la suppression

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);
```



On suppose value = 17

```
    if(cur->value == value) {             /* trouvé! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1;
    }

    if(mark(cur->next)) {                 /* cur doit être supprimé */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
    } else continue;
  }
  pred = &cur->next;
}
```

18/11/13

Mémoires Transactionnelles

72

La liste chaînée triée d'entier : la suppression

```
void del(coloredPointer* plist, int value) {
    restart:
    coloredPointer* pred = plist;
    while(!found) {
        Node cur = pointer(*pred);
        plist
        cur
        pred
        On suppose value = 17

        if(cur->value == value) {
            /* trouvé! */
            do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
            found = 1; }

        if(mark(cur->next)) {
            /* cur doit être supprimé */
            if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
            else continue;
        }
        pred = &cur->next;
    }
}
```

Le CAS ne peut rater que si next est modifié à cause
De la suppression du nœud 17 par un autre thread
Insertion avant 22, suppression de 22

18/11/13

Mémoires Transactionnelles

73

La liste chaînée triée d'entier : la suppression

```
void del(coloredPointer* plist, int value) {
    restart:
    coloredPointer* pred = plist;
    while(!found) {
        Node cur = pointer(*pred);
        plist
        cur
        pred
        On suppose value = 17

        if(cur->value == value) {
            /* trouvé! */
            do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
            found = 1; }

        if(mark(cur->next)) {
            /* cur doit être supprimé */
            if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
            else continue;
        }
        pred = &cur->next;
    }
}
```

18/11/13

Mémoires Transactionnelles

74

La liste chaînée triée d'entier : la suppression

```
void del(coloredPointer* plist, int value) {
    restart:
    coloredPointer* pred = plist;
    while(!found) {
        Node cur = pointer(*pred);
        plist
        cur
        pred
        On suppose value = 17

        if(cur->value == value) {
            /* trouvé! */
            do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
            found = 1; }

        if(mark(cur->next)) {
            /* cur doit être supprimé */
            if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
            else continue;
        }
        pred = &cur->next;
    }
}
```

18/11/13

Mémoires Transactionnelles

75

La liste chaînée triée d'entier : la suppression

```
void del(coloredPointer* plist, int value) {
    restart:
    coloredPointer* pred = plist;
    while(!found) {
        Node cur = pointer(*pred);
        plist
        cur
        pred
        On suppose value = 17

        if(cur->value == value) {
            /* trouvé! */
            do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
            found = 1; }

        if(mark(cur->next)) {
            /* cur doit être supprimé */
            if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
            else continue;
        }
        pred = &cur->next;
    }
}
```

Imaginons qu'un autre thread ait inséré un nœud

18/11/13

Mémoires Transactionnelles

76

La liste chaînée triée d'entier : la suppression

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);

    On suppose value = 17

    if(cur->value == value) {          /* trouvé! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

    if(mark(cur->next)) {              /* cur doit être supprimé */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  }
}
```

Le prochain insert ou del qui passe par le nœud 17 le supprimera pour nous (quitte car found = 1)

18/11/13

Mémoires Transactionnelles

77

La liste chaînée triée d'entier : pas trouvé

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);

    if(cur == null || value < cur->value) /* pas trouvé */
      return 0;

    if(cur->value == value) {          /* trouvé! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

    if(mark(cur->next)) {              /* cur doit être supprimé */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  }
}
```

Value = 13

18/11/13

Mémoires Transactionnelles

78

La liste chaînée triée d'entier : pas trouvé

```
void del(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(!found) {
    Node cur = pointer(*pred);

    if(cur == null || value < cur->value) /* pas trouvé */
      return 0;

    if(cur->value == value) {          /* trouvé! */
      do { n = cur->next; } while(CAS(&cur->next, n, n | 1) != n);
      found = 1; }

    if(mark(cur->next)) {              /* cur doit être supprimé */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  }
}
```

Value = 13
⇒ quitte la fonction

18/11/13

Mémoires Transactionnelles

79

La liste chaînée triée d'entier

```
void add(coloredPointer* plist, int value) {
  restart:
  coloredPointer* pred = plist;
  while(true) {
    Node cur = pointer(*pred);

    if(mark(cur->next)) {              /* cur doit être supprimé */
      if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
      else continue;
    }
    pred = &cur->next;
  }
}
```

Insertion : même principe pour le parcours et la compaction

18/11/13

Mémoires Transactionnelles

80

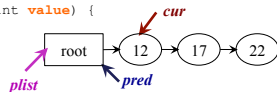
La liste chaînée triée d'entier : ajout

```
void add(coloredPointer* plist, int value) {
    restart:
    coloredPointer* pred = plist;
    while(true) {
        Node cur = pointer(*pred);
        if(cur == null || value < cur->value) /* insertion */
            if(CAS(pred, cur, new Node(cur, value)) != cur) goto restart;
        else return;

        if(mark(cur->next)) { /* cur doit être supprimé */
            if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
            else continue;
        }

        pred = &cur->next;
    }
}
```

Exemple : value = 14



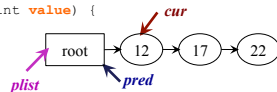
La liste chaînée triée d'entier : ajout

```
void add(coloredPointer* plist, int value) {
    restart:
    coloredPointer* pred = plist;
    while(true) {
        Node cur = pointer(*pred);
        if(cur == null || value < cur->value) /* insertion */
            if(CAS(pred, cur, new Node(cur, value)) != cur) goto restart;
        else return;

        if(mark(cur->next)) { /* cur doit être supprimé */
            if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
            else continue;
        }

        pred = &cur->next;
    }
}
```

Exemple : value = 14



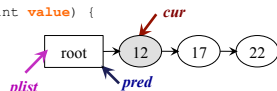
La liste chaînée triée d'entier : ajout

```
void add(coloredPointer* plist, int value) {
    restart:
    coloredPointer* pred = plist;
    while(true) {
        Node cur = pointer(*pred);
        if(cur == null || value < cur->value) /* insertion */
            if(CAS(pred, cur, new Node(cur, value)) != cur) goto restart;
        else return;

        if(mark(cur->next)) { /* cur doit être supprimé */
            if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
            else continue;
        }

        pred = &cur->next;
    }
}
```

CAS peut rater pour deux raisons :
 Cur est marqué supprimé
 Un autre nœud est inséré
 Dans le doute, recommence

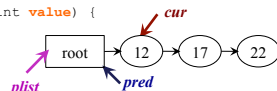


La liste chaînée triée d'entier : ajout

```
void add(coloredPointer* plist, int value) {
    restart:
    coloredPointer* pred = plist;
    while(true) {
        Node cur = pointer(*pred);
        if(cur == null || value < cur->value) /* insertion */
            if(CAS(pred, cur, new Node(cur, value)) != cur) goto restart;
        else return;

        if(mark(cur->next)) { /* cur doit être supprimé */
            if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
            else continue;
        }

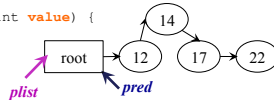
        pred = &cur->next;
    }
}
```



La liste chaînée triée d'entier : ajout

```
void add(coloredPointer* plist, int value) {
    restart:
    coloredPointer* pred = plist;
    while(true) {
        Node cur = pointer(*pred);
        if(cur == null || value < cur->value) /* insertion */
            if(CAS(pred, cur, new Node(cur, value)) != cur) goto restart;
            else return;

        if(mark(cur->next)) { /* cur doit être supprimé */
            if(CAS(pred, cur, pointer(cur->next)) != cur) goto restart;
            else continue;
        }
        pred = &cur->next;
    }
}
```



18/11/13

Mémoires Transactionnelles

85

L'algorithme CAS

```
struct lock { int locked; } /* 0 ⇒ libre */

void lock(struct lock* lock) {
    while(CAS(&lock->locked, 0, 1));
}

void unlock(struct lock* lock) {
    lock->locked = 0;
}
```

Problème : si tous les cœurs essaient d'accéder à la ligne de cache, les performances s'écroulent

18/11/13

Mémoires Transactionnelles

87

Structures non bloquantes

1. Les outils
2. La pile
3. La queue
4. La liste chaînée
5. Algorithmes de verrouillage

18/11/13

Mémoires Transactionnelles

86

L'algorithme MCS

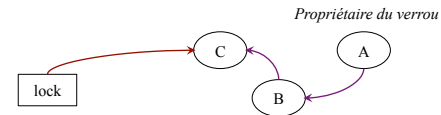
MCS = Mellor-Crummey and Scott (ASPLOS 1991)

Principe : éviter la contention sur une ligne de cache unique

- ✓ Chaque thread en attente s'ajoute dans une pile
- ✓ Spin sur son propre nœud, libéré par le précédent dans la liste

Idee : le dernier thread qui demande le verrou est stocké dans la variable partagée lock

- ✓ Si le verrou est pris, doit chaîner l'ancienne liste et attendre d'être libéré
- ✓ Sinon, le verrou est directement acquis



18/11/13

Mémoires Transactionnelles

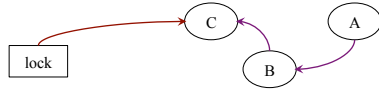
88

L'algorithme MCS

```
void CS() {
    struct node my_node(0, 1);
    struct node* pred = xchg(&lock, node);
    if(pred) {
        pred->next = my_node;
        while(!my_node->spin);
    }
    execute_cs();
    if(CAS(&lock, my_node, 0) == 0)
        return;
    while(!my_node->next);
    my_node->next->locked = 0;
}
```

```
struct node {
    struct node* next;
    int locked;
};
struct node* lock;
```

Propriétaire du verrou

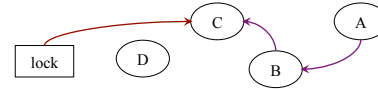


L'algorithme MCS

```
void CS() {
    struct node my_node(0, 1);
    struct node* pred = xchg(&lock, node);
    if(pred) {
        pred->next = my_node;
        while(!my_node->spin);
    }
    execute_cs();
    if(CAS(&lock, my_node, 0) == 0)
        return;
    while(!my_node->next);
    my_node->next->locked = 0;
}
```

```
struct node {
    struct node* next;
    int locked;
};
struct node* lock;
```

Propriétaire du verrou

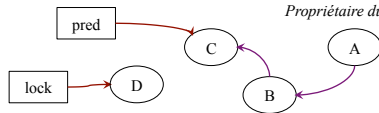


L'algorithme MCS

```
void CS() {
    struct node my_node(0, 1);
    struct node* pred = xchg(&lock, node);
    if(pred) {
        pred->next = my_node;
        while(!my_node->spin);
    }
    execute_cs();
    if(CAS(&lock, my_node, 0) == 0)
        return;
    while(!my_node->next);
    my_node->next->locked = 0;
}
```

```
struct node {
    struct node* next;
    int locked;
};
struct node* lock;
```

Propriétaire du verrou

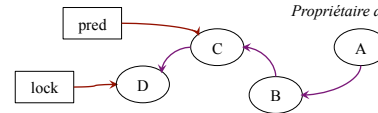


L'algorithme MCS

```
void CS() {
    struct node my_node(0, 1);
    struct node* pred = xchg(&lock, node);
    if(pred) {
        pred->next = my_node;
        while(!my_node->spin);
    }
    execute_cs();
    if(CAS(&lock, my_node, 0) == 0)
        return;
    while(!my_node->next);
    my_node->next->locked = 0;
}
```

```
struct node {
    struct node* next;
    int locked;
};
struct node* lock;
```

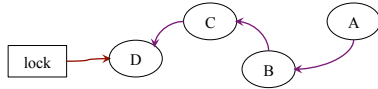
Propriétaire du verrou



L'algorithme MCS

```
void CS() {
    struct node my_node(0, 1);
    struct node* pred = xchg(&lock, node);
    if(pred) {
        pred->next = my_node;
        while(!my_node->spin);
    }
    execute_cs();
    if(CAS(&lock, my_node, 0) == 0)    Exécution de A
        return;
    while(!my_node->next);
    my_node->next->locked = 0;
}
```

Propriétaire du verrou



18/11/13

Mémoires Transactionnelles

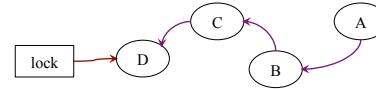
93

L'algorithme MCS

```
void CS() {
    struct node my_node(0, 1);
    struct node* pred = xchg(&lock, node);
    if(pred) {
        pred->next = my_node;
        while(!my_node->spin);
    }
    execute_cs();
    if(CAS(&lock, my_node, 0) == 0)
        return;
    while(!my_node->next);
    my_node->next->locked = 0;
}
```

Gestion de la période intermédiaire
entre le xchg et le pred->next = my_node

Propriétaire du verrou



18/11/13

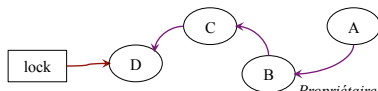
Mémoires Transactionnelles

94

L'algorithme MCS

```
void CS() {
    struct node my_node(0, 1);
    struct node* pred = xchg(&lock, node);
    if(pred) {
        pred->next = my_node;
        while(!my_node->spin);
    }
    execute_cs();
    if(CAS(&lock, my_node, 0) == 0)
        return;
    while(!my_node->next);
    my_node->next->locked = 0;
}
```

Propriétaire du verrou



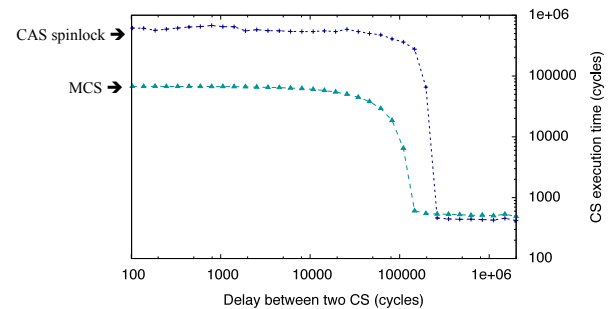
18/11/13

Mémoires Transactionnelles

95

L'algorithme MCS

Évaluation sur un 48 cœurs



18/11/13

Mémoires Transactionnelles

96

Conclusion

Verrou :

- ✓ Algorithmes compliqués (mais pas en M2 système)
- ✓ Non composable
- ✓ Performances moyennes si beaucoup de contention (bloquage)

Mémoires transactionnelles

- ✓ Abstraction simple
- ✓ Composable
- ✓ Performance mauvaise si beaucoup de contention (abort)

Structures lock-free

- ✓ Algorithmes très compliquée (même en M2 système)
- ✓ Pas composable
- ✓ Excellentes performances