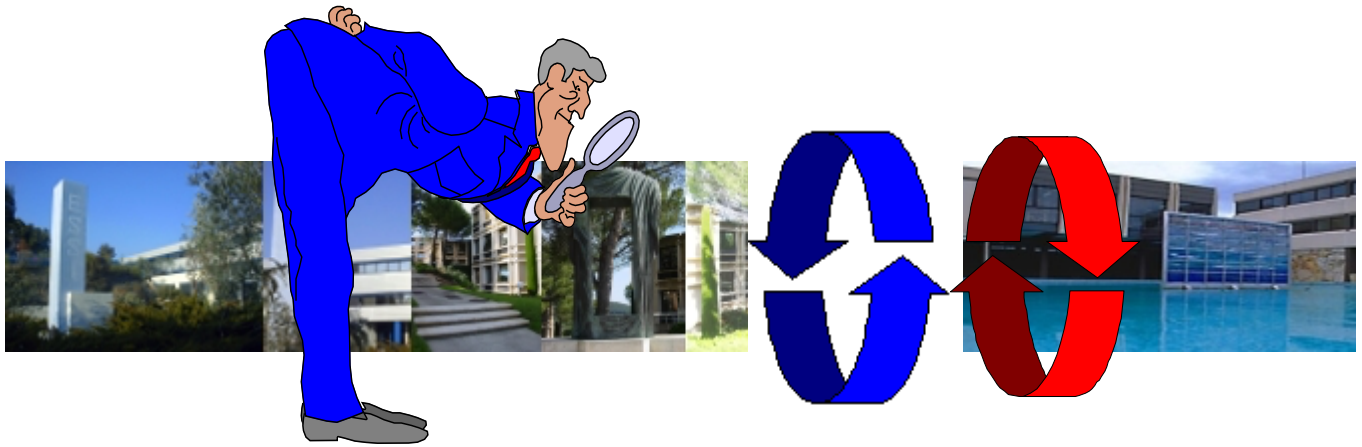
Wait free synchronisation

riveill@unice.fr

http://www.i3s.unice.fr/~riveill



Wait free synchronisation

- Objectif : améliorer de l'efficacité des programmes concurrents en ne bloquant plus les processus
  - PROs : facilité de programmation, modèles bien compris
  - CONs : bloquer un processus « coûte cher » en temps d'exécution

- 2 pistes : améliorer les algorithmes de verrouillage
  1. Remplacer les verrous bloquants par des verrous plus réactif (spinlock)
  2. Supprimer les verrous bloquants par une algorithme non bloquante

Remplacer les verrous bloquants par des spinlock (attente active)

Mise œuvre d'un spinlock
Principe

```
int test_and_set(int *s) {
      // début partie atomique
      int tmp = *s;
      *s = 1;
      // fin partie atomique
      return tmp;
}
```

- **initialisation**

```
int lock = 0;
```

- **Prise du verrou**

```
int lock(int *lock) {
      // boucle si le lock est déjà à 1
      while (test_and_set(lock) == 1);
}
```

- **Libération du verrou**

```
void unlock(int *lock) {
      *lock = 0
}
```

ATTENTION : Ceci est du pseudo-code, montré ici juste pour illustrer le principe.

S'il était utilisé normalement dans un compilateur, les garanties nécessaires d'atomicité, non-usage de cache, non-réorganisation par l'optimiseur etc. ne serait surement pas assurée

Partage de variable entre thread / `volatile`

1. Variables non volatiles
   - Atomicité des lectures/écritures : oui si la variable est codée sur 1 mot
     - Non pour long, double qui sont codés sur 2 mots mais aussi accès aux éléments d'un tableau
   - Visibilité des écritures : la spécification Java impose que la modification de la valeur d'une variable par une thread soit connue des autres threads uniquement lors de l'acquisition ou du relâchement du moniteur d'un objet (synchronized)
     - Entry/exit consistency model
2. Variable `volatile` : la spécification Java impose
   - Atomicité des lectures/écritures + visibilité des écritures 'immédiates'
     - Y compris pour long / double
3. Java introduit aussi une autre classe d'objet
   - Les objets 'Atomique'

Objets atomiques

- Atomicité : ses effets ne sont pas visibles avant la fin de l'opération mais ils sont ensuite visibles par tous
- Le paquetage java.util.concurrent.atomic introduit les variables atomiques
  - Permet d'exploiter au mieux les instructions de type TAS (test-and-set) ou CAS (compare-and-swap)
  - Objectif : programmer l'entrée en section critique par attente active
    - Même principe que Peterson, mais plus simple à mettre en oeuvre
- Concerne principalement les types simples :
  - *AtomicBoolean*, *AtomicInteger*, *AtomicReference* and *AtomicLong*.
- Par exemple, voici quelques méthodes du type *AtomicInteger* :
  - int addAndGet(int delta)
    - Atomically add given value to current value
  - boolean compareAndSet(int expect, int update)
    - Atomically set the value to update if old value equals to expect
  - int incrementAndGet() or int decrementAndGet()
    - Atomically increment/ decrement current value by one

Instruction **Test&Set**
Présent sur les processeurs des années 1970

- Test and Set : test et affecte une variable de manière atomique

```
bool Test&Set (bool *lock) { // exécuté de manière
atomique
    if (lock == false) {*lock = true; return false;}
    else return true;
}
```

// Si lock est faux → mis à vrai et faux est retourné

// Si lock est vrai → vrai est retourné

- Utilisation : variable partagée **lock**, initialisée à **false**

```
int lock(int *lock) { while (test_and_set(lock)); }

void unlock(int *lock) {*lock = false}
```

  - Il s'agit d'un algorithme par attente active mais implémenté à l'aide d'une instruction spécifique du processeur
  - A utiliser si la probabilité d'attente est faible, bien plus simple que les algorithmes vues en TD
  - Mais il faut utiliser l'instruction machine 'T&S'

Instruction **Compare&Swap**
Supporté par les architectures multi-cœurs depuis 2003

- Compare and Swap : test et échange la valeur d'une variable de manière atomique

```
int compare_and_swap(int* reg, int oldval, int newval) {
   int old_reg_val = *reg;
   if (old_reg_val == oldval) *reg = newval;
   return old_reg_val;
}
```

  - **Si reg = oldval → l'ancienne valeur de reg est retournés et reg est modifié**
  - **Si reg != oldval → la valeur de reg est retournée**

- Utilisation
  - Pour implémenter les primitives de synchronisation tel que sémaphore ou verrou
  - Mais aussi pour implémenter des algorithmes de synchronisation sans attente (lock-free algorithms, wait-free algorithms) mis en évidence par Maurice Herlihy (1991)

Exemple d'utilisation classe Atomic Java

```
// une classe compteur
public class SimpleCompteur {
    private static int compte ;
    public void compte() { compte++ ; }
    public static int getCompte() { return compte ; }
}

// utilisation de la classe compteur
// 5 threads lancés en parallèle exécutent
public void run() { // chaque thread possède son propre
compteur
    SimpleCompteur compteur = new SimpleCompteur() ;
    for (int i = 0 ; i < 100 ; i++) { compteur.compte() ; }
}
```

→ Résultat : **compteur = 352** // change à chaque exécution

Exemple d'utilisation classe Atomic Java

- Pour corriger cela → construire une section critique
  - Avec ce que l'on a vu en cours

1. Utilisation d'un verrou

```
public class SimpleCompteur {
    private static int compte ; // objet de synchronization
    private static Object key = new Object() ;
    public void compte() {
        synchronized(key) { compte++ ; }
    }
    public static int getCompte() { return compte ; }
}
// il faut que toutes les threads partagent le même objet verrou
```

- Autres solutions
  1. Peterson + variables volatiles
  2. Section critique avec CAS
  3. AtomicInteger

Exemple d'utilisation classe Atomic Java

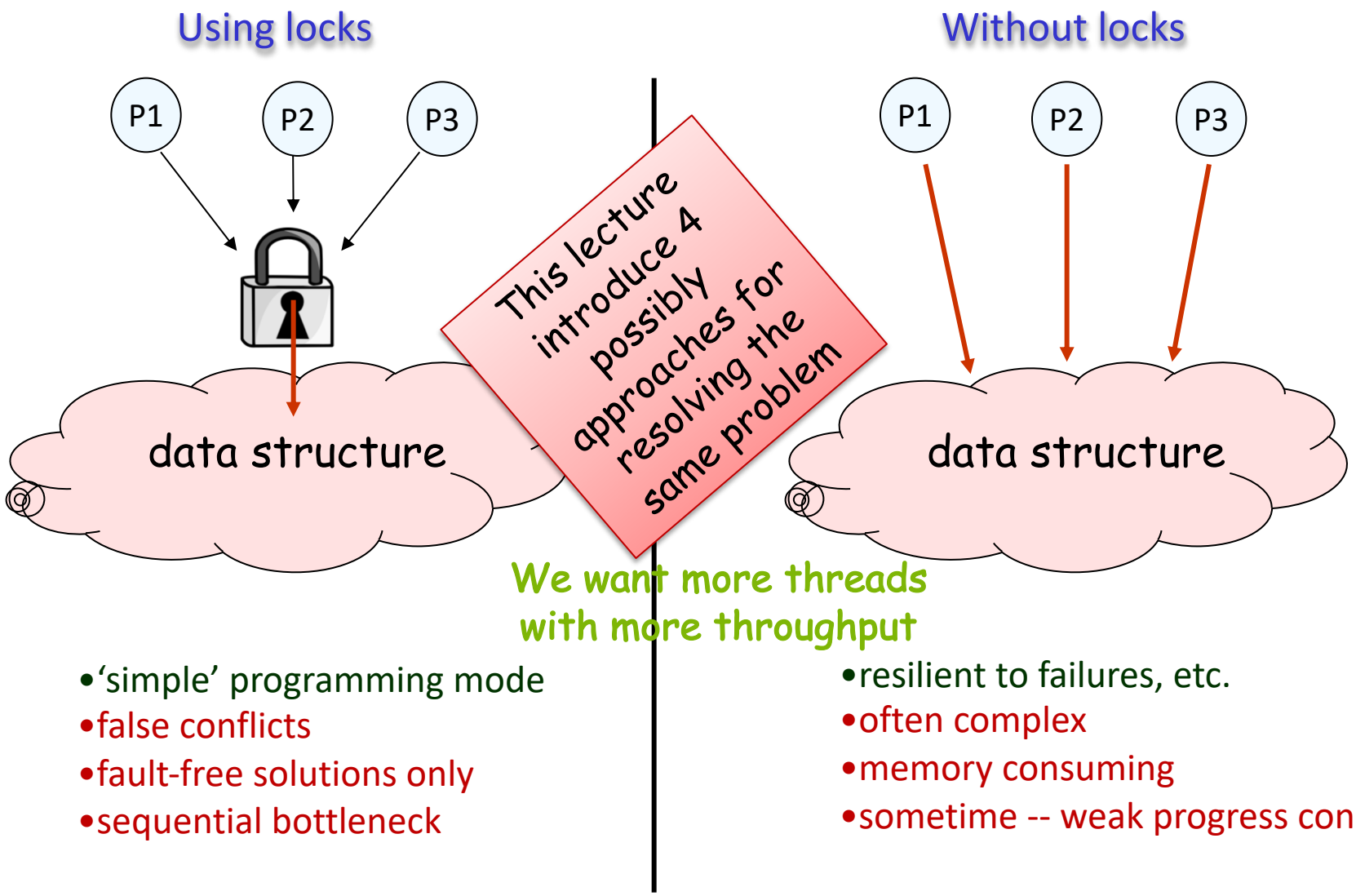
- Utilisation de la classe AtomicInteger

```
public class SimpleCompteur {
   private static AtomicInteger compte = new
AtomicInteger(0) ;
   public void compte() {
     compte.incrementAndGet() ;
   }
   public static int getCompte() {
      return compte.intValue() ;
   }
```

1.000.000 itérations 5 threads	Valeur finale	Temps réponse
Unsafe	25636349	50 ms
Lock	50000000	1.731 ms
Atomic	50000000	1.069 ms

Remplacer les verrous bloquants par une algorithmique non bloquante

Concurrent Data Structures



- 'simple' programming mode
- false conflicts
- fault-free solutions only
- sequential bottleneck

- resilient to failures, etc.
- often complex
- memory consuming
- sometime -- weak progress condition

4 approches

1. Fine-Grained Synchronization
   - Instead of using a single lock …
   - Split object into
     - Independently-synchronized components
   - Methods conflict when they access
     - The same component …
     - At the same time
2. Optimistic Synchronization
   - Search without locking …
   - If you find it, lock and check …
     - OK: we are done
     - Oops: start over
   - Evaluation
     - Usually cheaper than locking, but
     - Mistakes are expensive

4 approches

3. Lazy Synchronization
   - Postpone hard work
   - Removing components is tricky
     - Logical removal
       - Mark component to be deleted
     - Physical removal
       - Do what needs to be done
4. Lock-Free Synchronization
   - Don't use locks at all
     - Use compareAndSet() & relatives …
   - Advantages
     - No Scheduler Assumptions/Support
   - Disadvantages
     - Complex
     - Sometimes high overhead

On ne regardera surement pas ces deux dernières étapes par manque de temps…

4 approaches illustrated on a single example

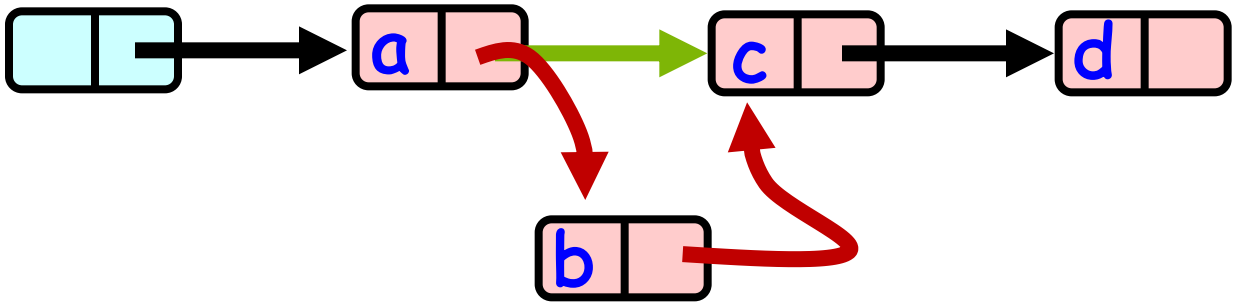
- Linked List
  - Common application
  - Building block for other apps
- Linked List specification
  - Unordered collection of items
  - Tail reachable from head
  - No duplicates

```
public interface Set<T> {
    // add(x) put x in set
    public boolean add(T x);
    // remove(x) take x out of set
    public boolean remove(T x);
    // contains(x) tests if x in set
    public boolean contains(T x);
}
```

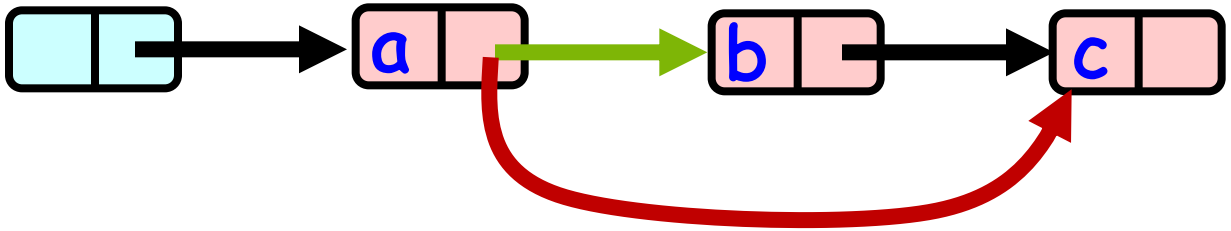
```
public class Node {
    public T item;
    // hash code
    public int key;
    public Node next;
}
```

Two basic operation : Add & Remove

Add()



Remove()



1. Coarse-Grained Synchronization
2. Fine-Grained Synchronization
3. Optimistic Synchronization
4. Lazy Synchronization
5. Lock-Free Synchronization

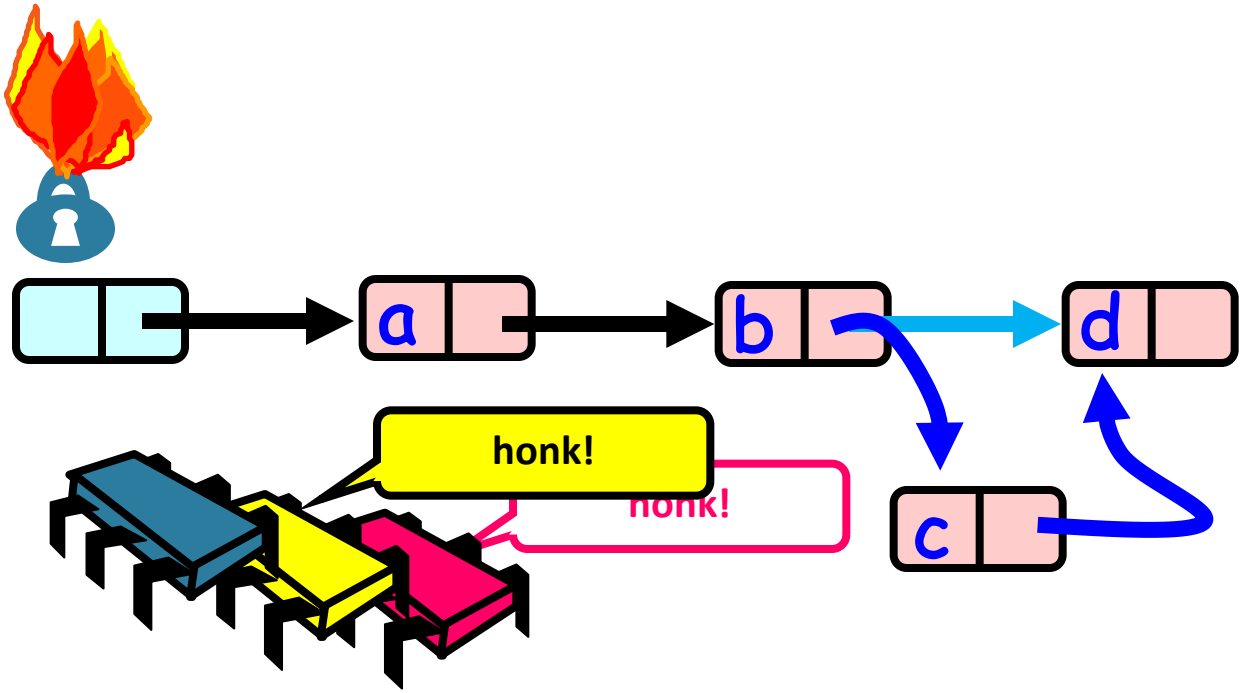
4 new approaches

Remove method (coarse grain locking)

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   lock();
   pred = this.head;
   curr = pred.next;
   while (curr.key <= key) {
      if (item == curr.item) {
         pred.next = curr.next;
         return true;
      }
      pred = curr; curr = curr.next;
    }
    return false;
 } finally {
    unlock();
 }}
```

Make sure locks released

1) Coarse Grained Locking



Simple but bottleneck

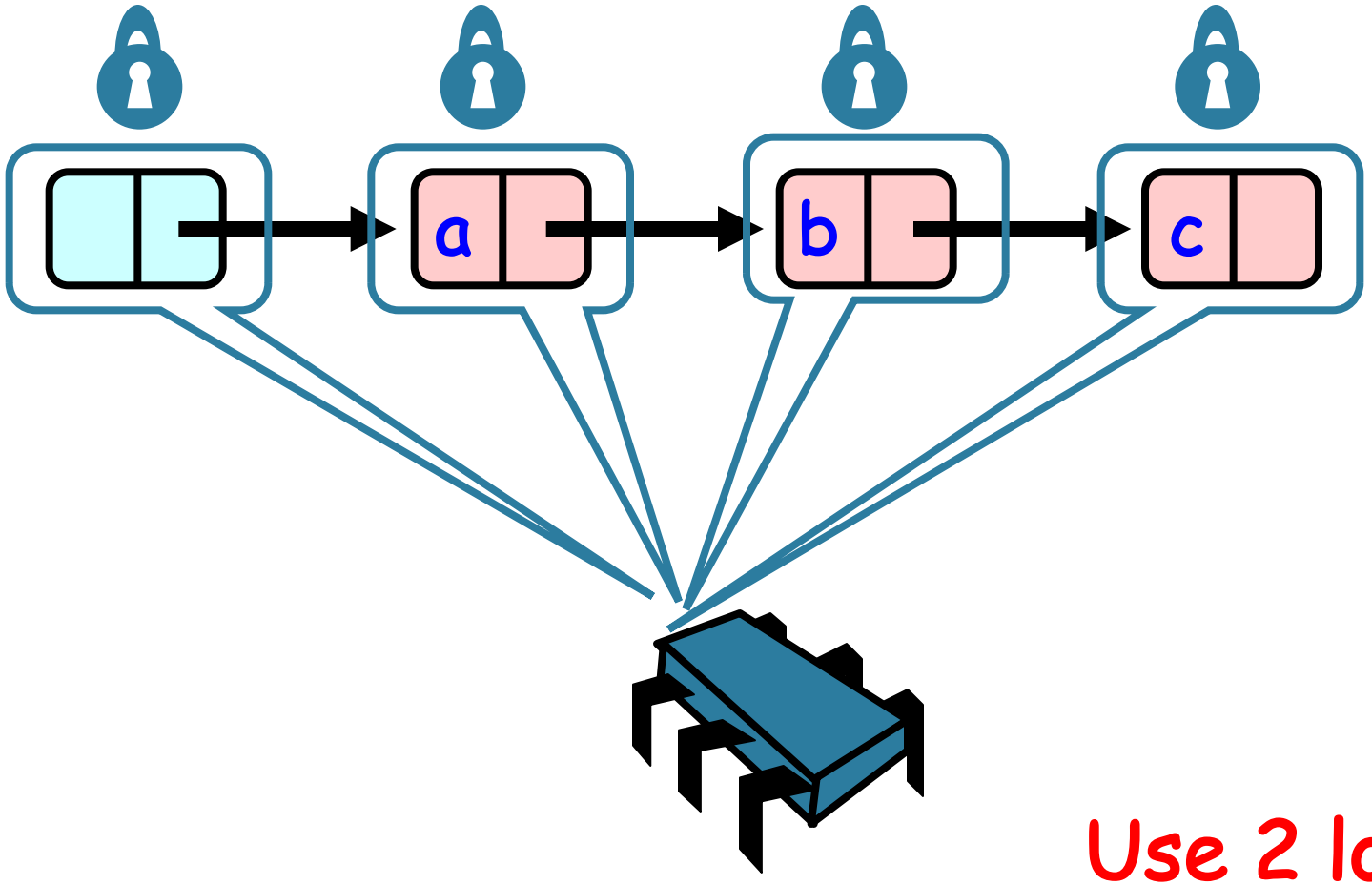
1. Coarse-Grained Synchronization
2. Fine-Grained Synchronization
3. Optimistic Synchronization
4. Lazy Synchronization
5. Lock-Free Synchronization

4 new approaches

1) Fine-grained Locking

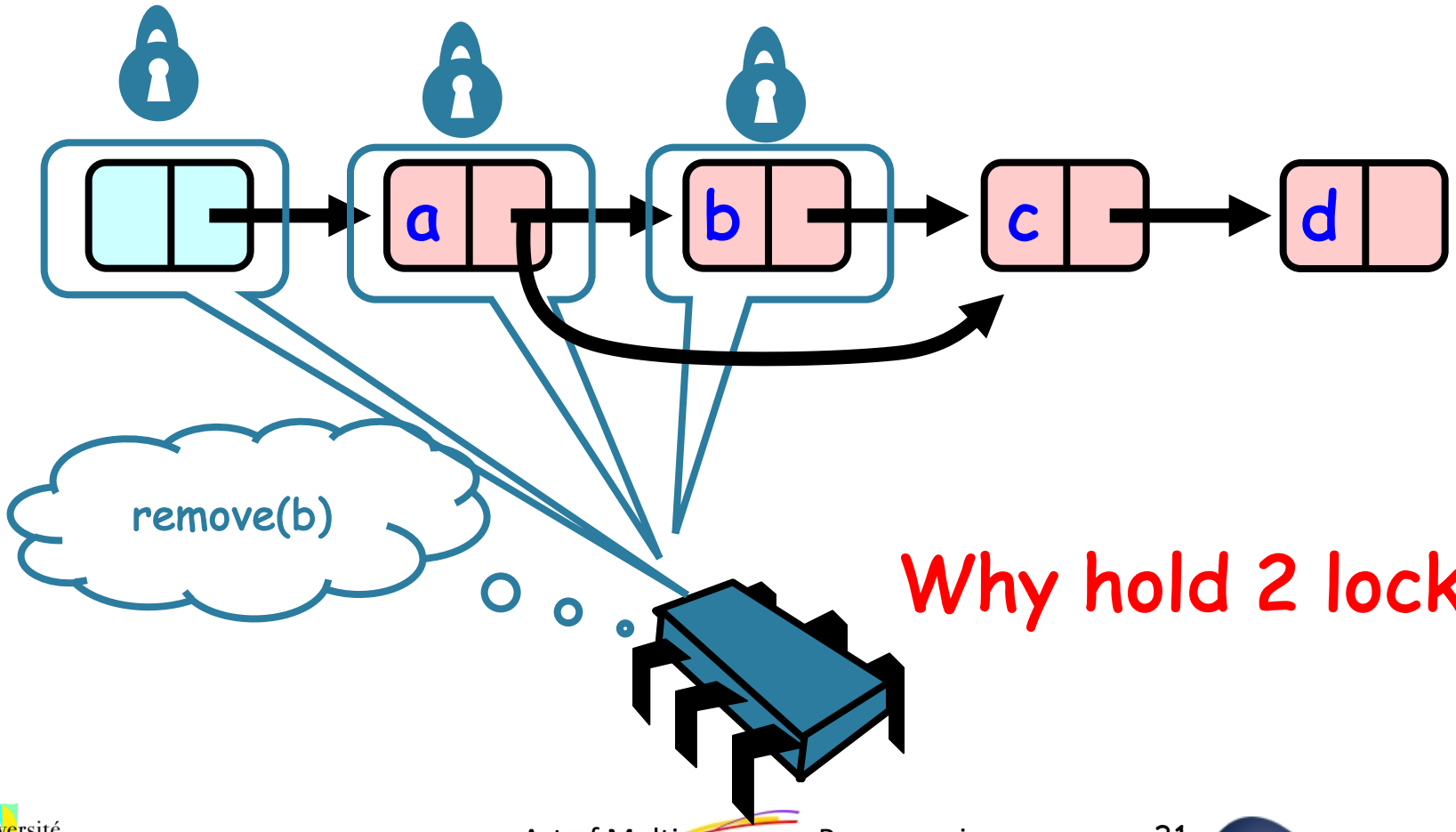
- Requires **careful** thought
- Split object into pieces
  - A List is composed of a set of Node
  - Each piece has own lock
    - Replace a Lock for a List
    - By a set a Lock for a set of Node
  - Methods that work on disjoint pieces need not exclude each other

Hand-over-Hand locking



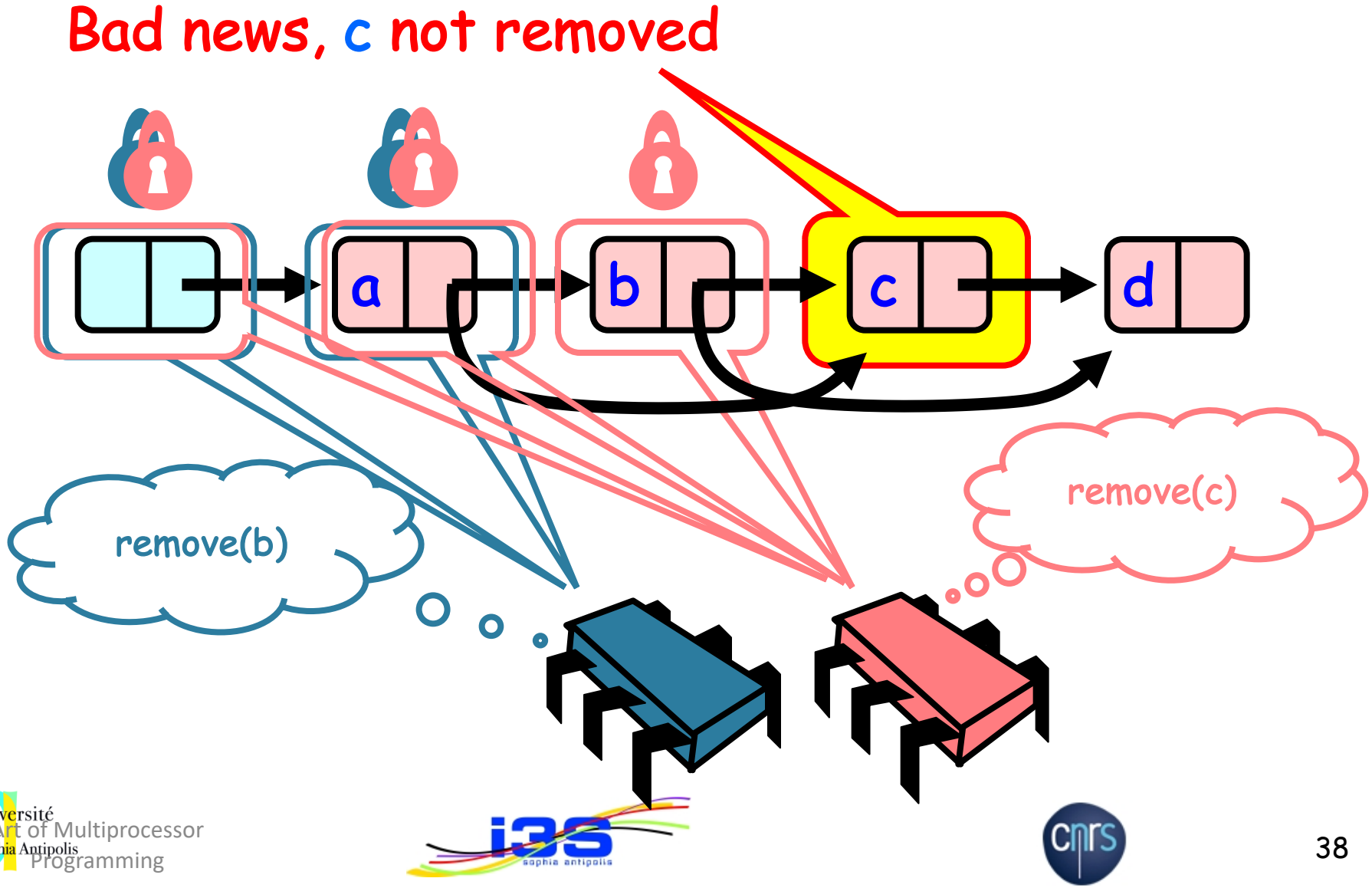
Use 2 locks

Removing a Node



Why hold 2 locks?

Concurrent Removes whith one lock



Remove method (fine grain locking)

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   pred = this.head; pred.lock();
   curr = pred.next; curr.lock();
   while (curr.key <= key) {
      if (item == curr.item) {
         pred.next = curr.next;
         return true;
      }
      pred.unlock();
      pred = curr; curr = curr.next;
      curr.lock();
    }
    return false;
 } finally {
    curr.unlock();
    pred.unlock();
 }}
```

Minimum locking

Why does this work?

- To remove node e
  - Must lock e
  - Must lock e's predecessor
- Therefore, if you lock a node
  - It can't be removed
  - And neither can its successor
- To add node e
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted
  - (Is successor lock actually required?)

Evaluation

- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Inefficient

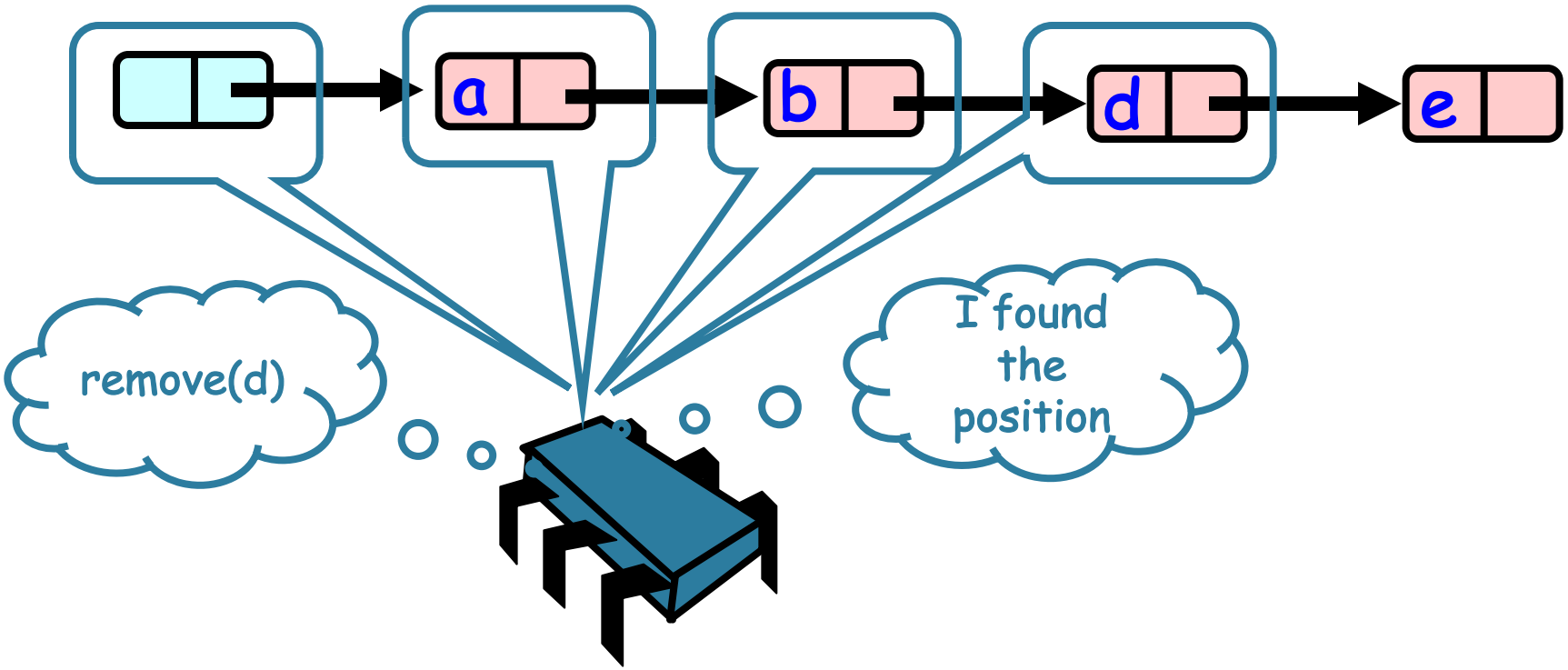
1. Coarse-Grained Synchronization
2. Fine-Grained Synchronization
3. Optimistic Synchronization
4. Lazy Synchronization
5. Lock-Free Synchronization

4 new approaches

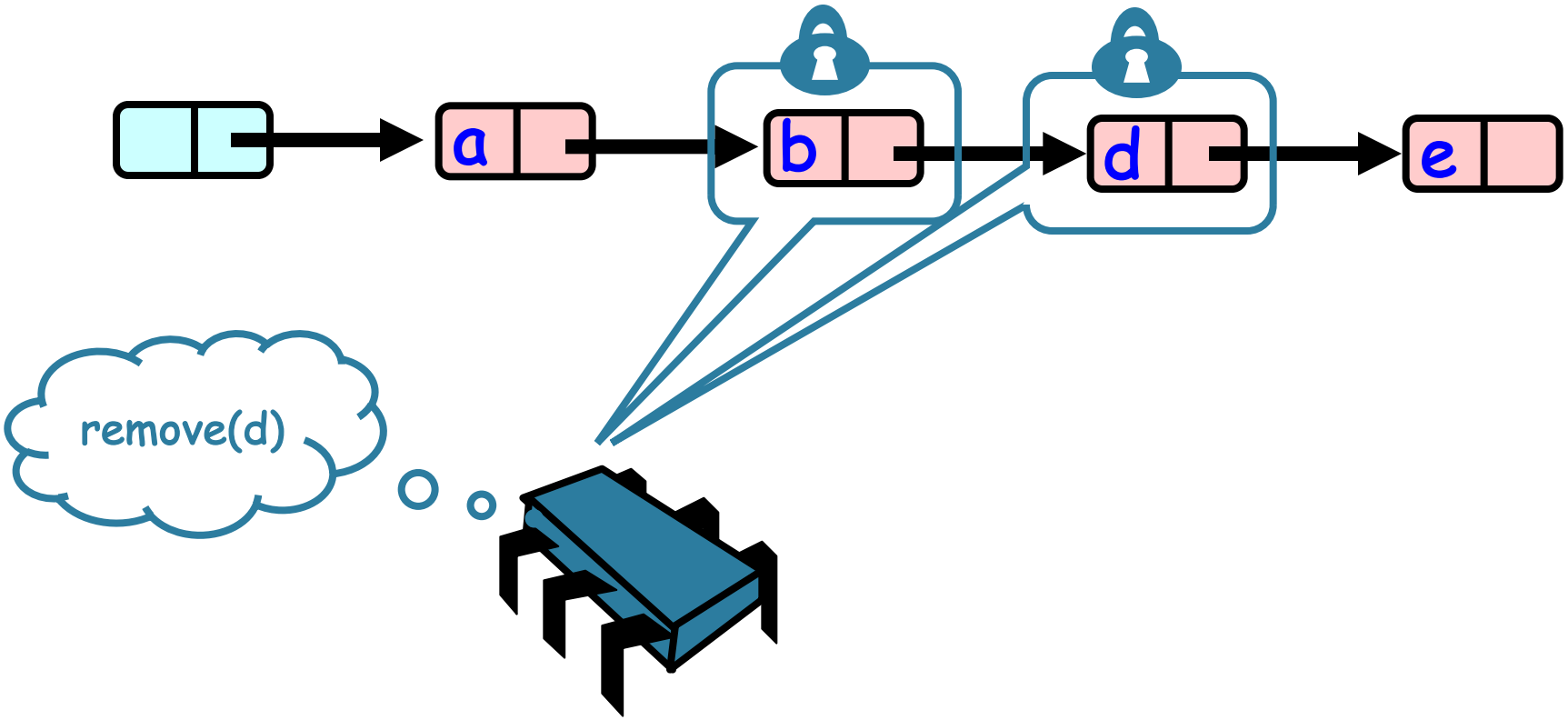
3) Optimistic Synchronization

- Find nodes without locking
- Lock nodes
- Check that everything is OK for the operation
- Do the operation

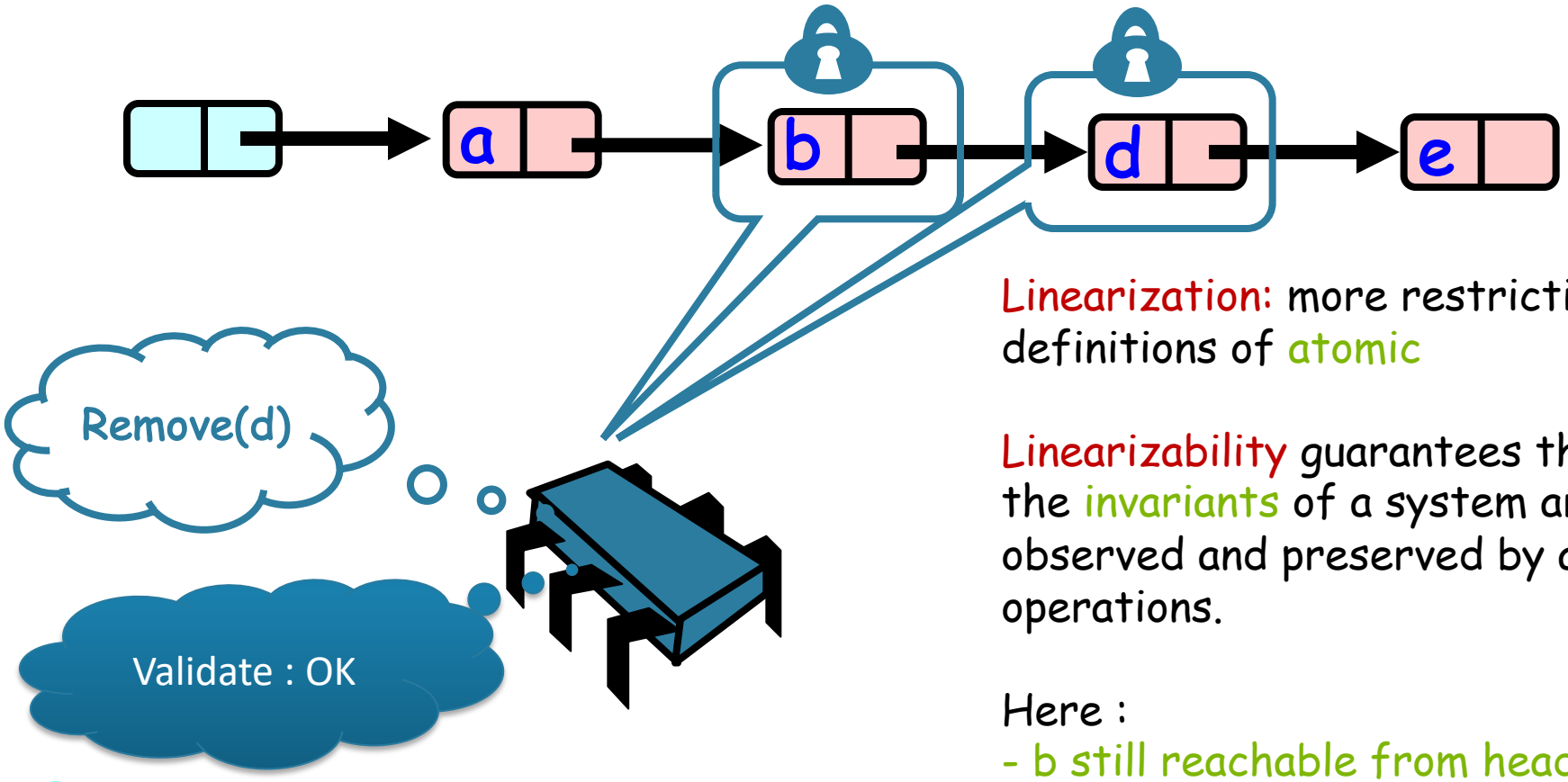
Optimistic: 1) Traverse without Locking



Optimistic: 2) Lock



Optimistic: 3) Check that everything is OK

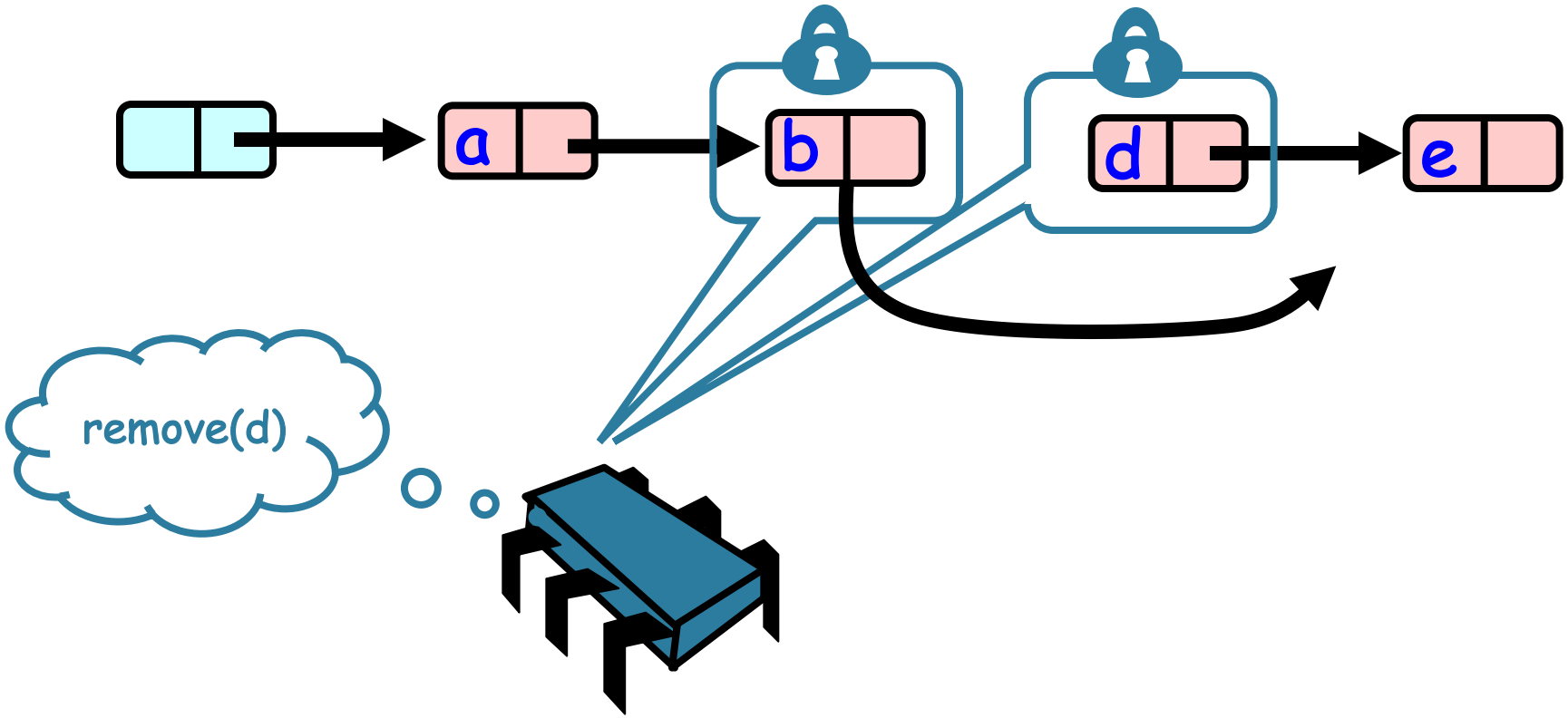


Linearization: more restrictive definitions of atomic

Linearizability guarantees that the invariants of a system are observed and preserved by all operations.

Here :
- b still reachable from head
- b still points to d

Optimistic: 4) Do the operation

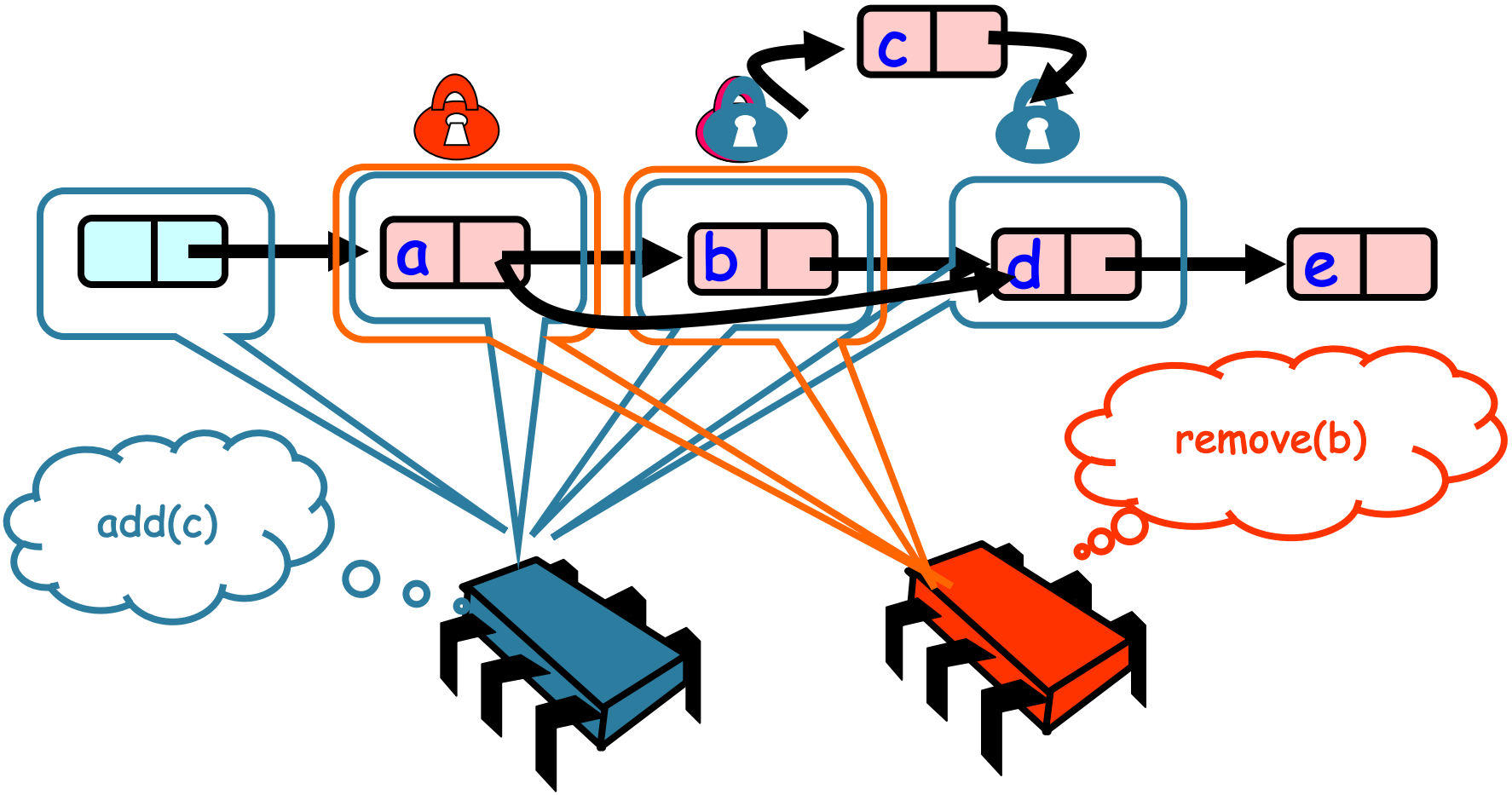


Remove method (optimistic synchronization)

```
public boolean remove(Item item) {
  int key = item.hashCode();
  Node pred = this.head;
  Node curr = pred.next;
  while (curr.key <= key) {
    if (item == curr.item) break;
    pred = curr; curr = curr.next;
  }
  try {
    pred.lock(); curr.lock();
    if (validate(pred,curr)) {
      if (curr.key == key) {
        pred.next = curr.next;
        return true;
      } else {
        return false;
  }}} finally {
    pred.unlock();
    curr.unlock();
}}
```

```
private boolean
  validate(Node pred,
           Node curr) {
  Node node = head;
  while (node.key
           <= pred.key) {
    if (node == pred)
      return pred.next==curr;
    node = node.next;
  }
  return false;
}
```

Is a validate operation necessary ?
Some possible execution without validation.



Evaluation

- OK for for
  - add(), remove(), contains()
- Much less lock acquisition/release
  - Traversals are wait-free
  - Performance
  - Concurrency
- Problems
  - Need to traverse list twice
  - contains() method acquires locks
    - In many apps, contains represent 90% of call

Gain ? Optimistic is effective if

- cost of scanning twice without locks

  is less than

- cost of scanning once with locks

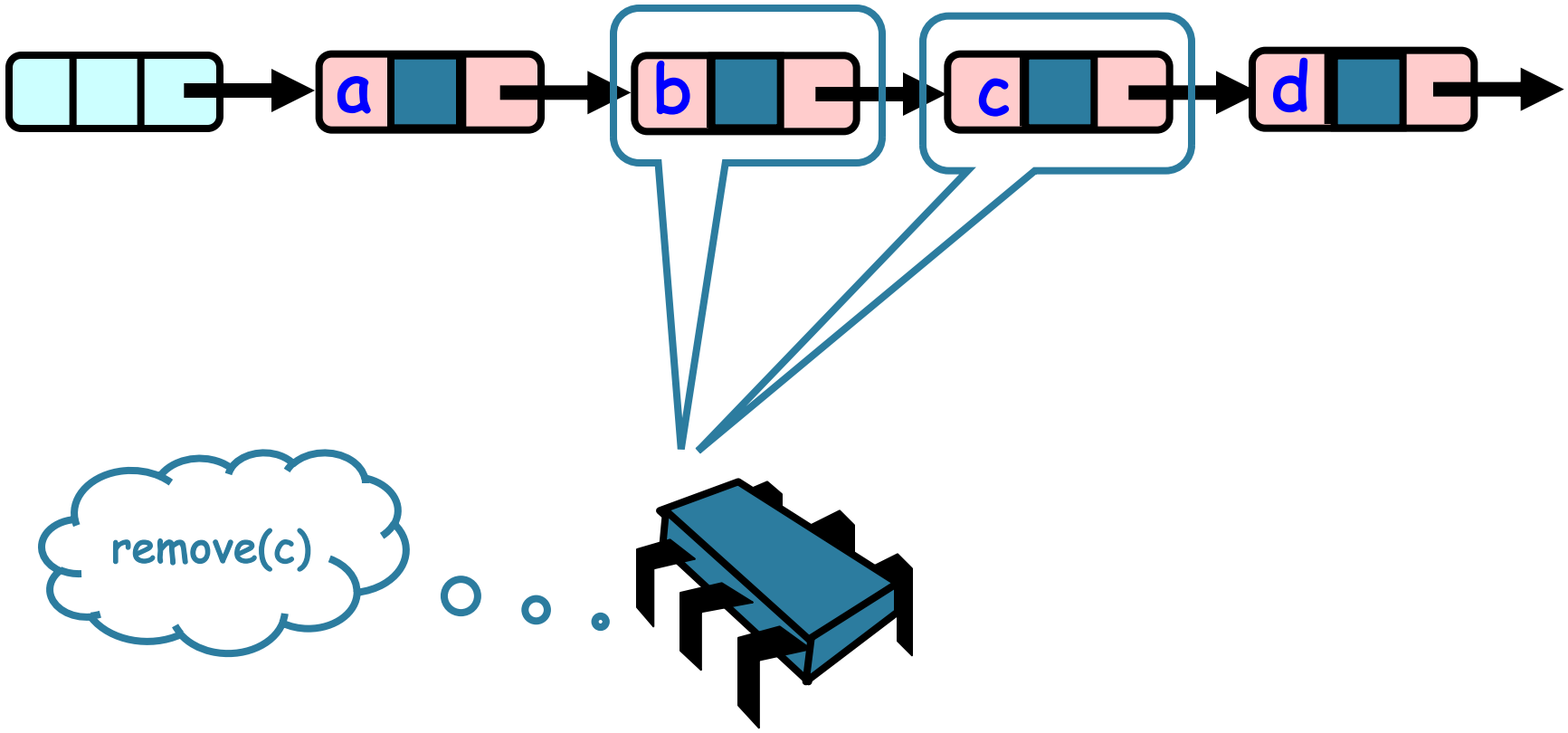
1. Coarse-Grained Synchronization
2. Fine-Grained Synchronization
3. Optimistic Synchronization
4. Lazy Synchronization
5. Lock-Free Synchronization

4 new approaches

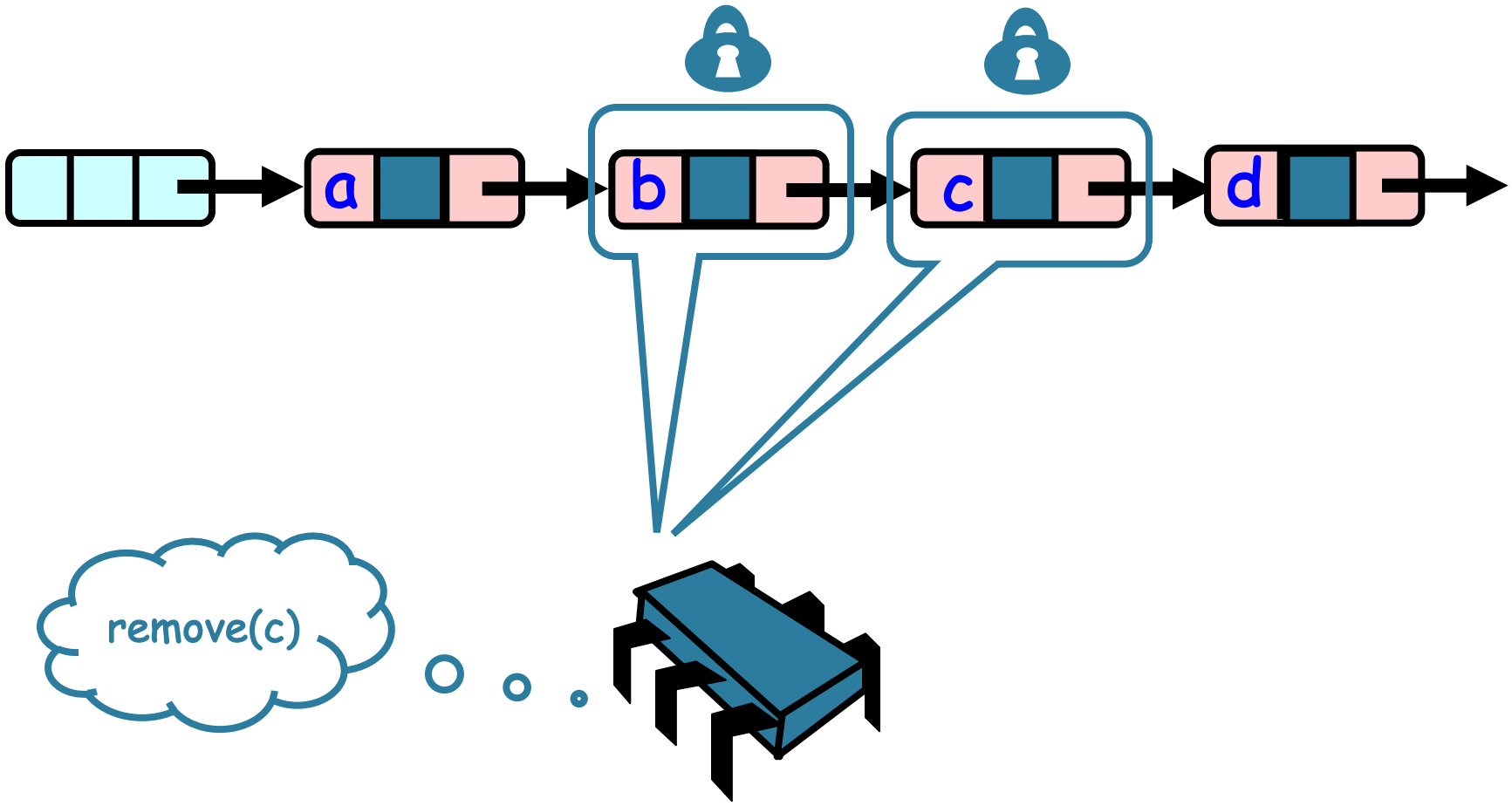
4) Lazy synchronisation

- `remove()`
  - Scans list (as before)
  - Locks predecessor & current (as before)
  - Validation
    - Check that pred is not marked
    - Check that curr is not marked
    - Check that pred points to curr
  - Logical delete : Marks current node as removed (new!)
  - Physical delete : Redirects predecessor's next (as before)

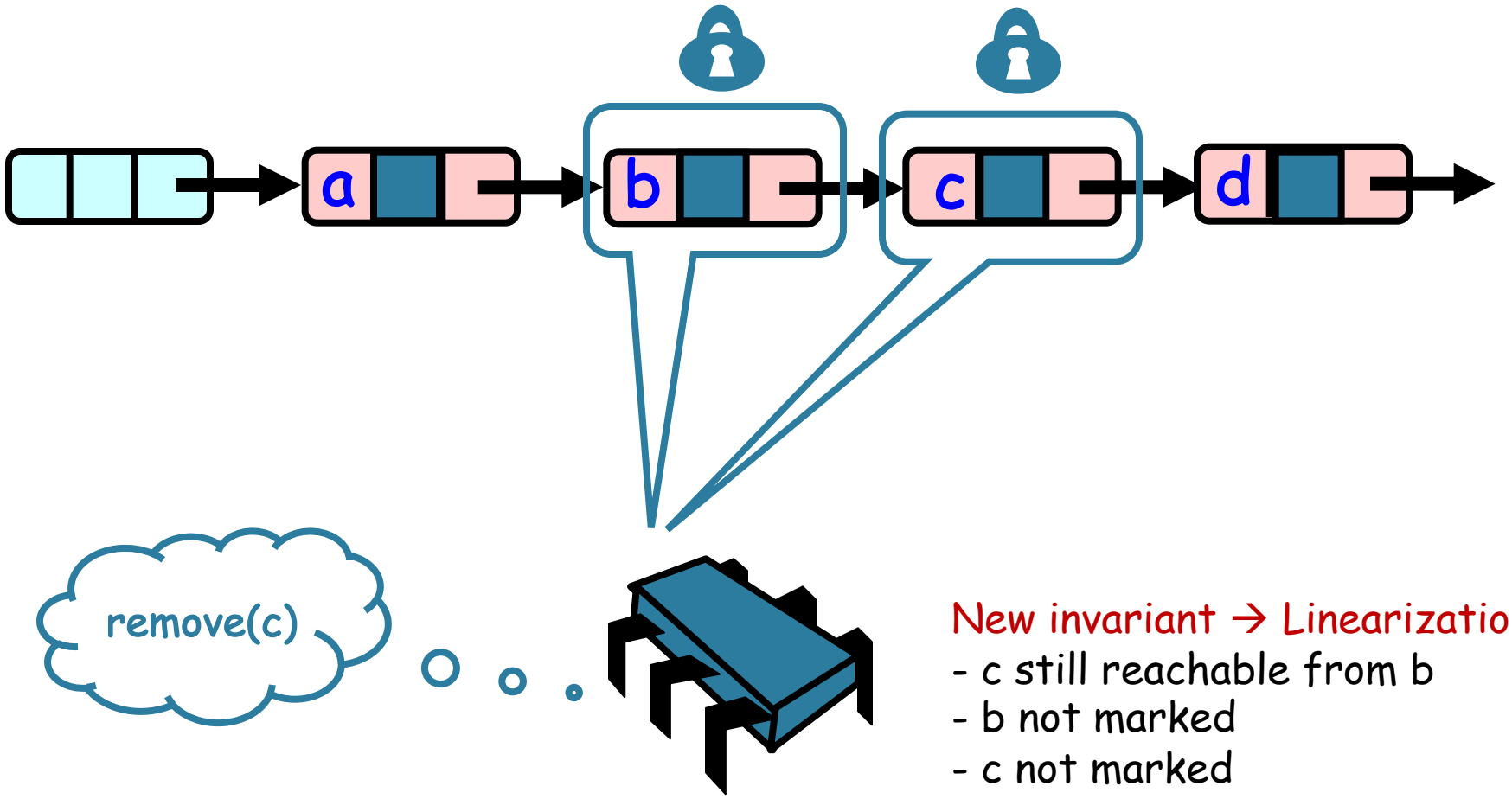
Lazy removal : 1) Traverse without Locking



Lazy removal : 2) Lock

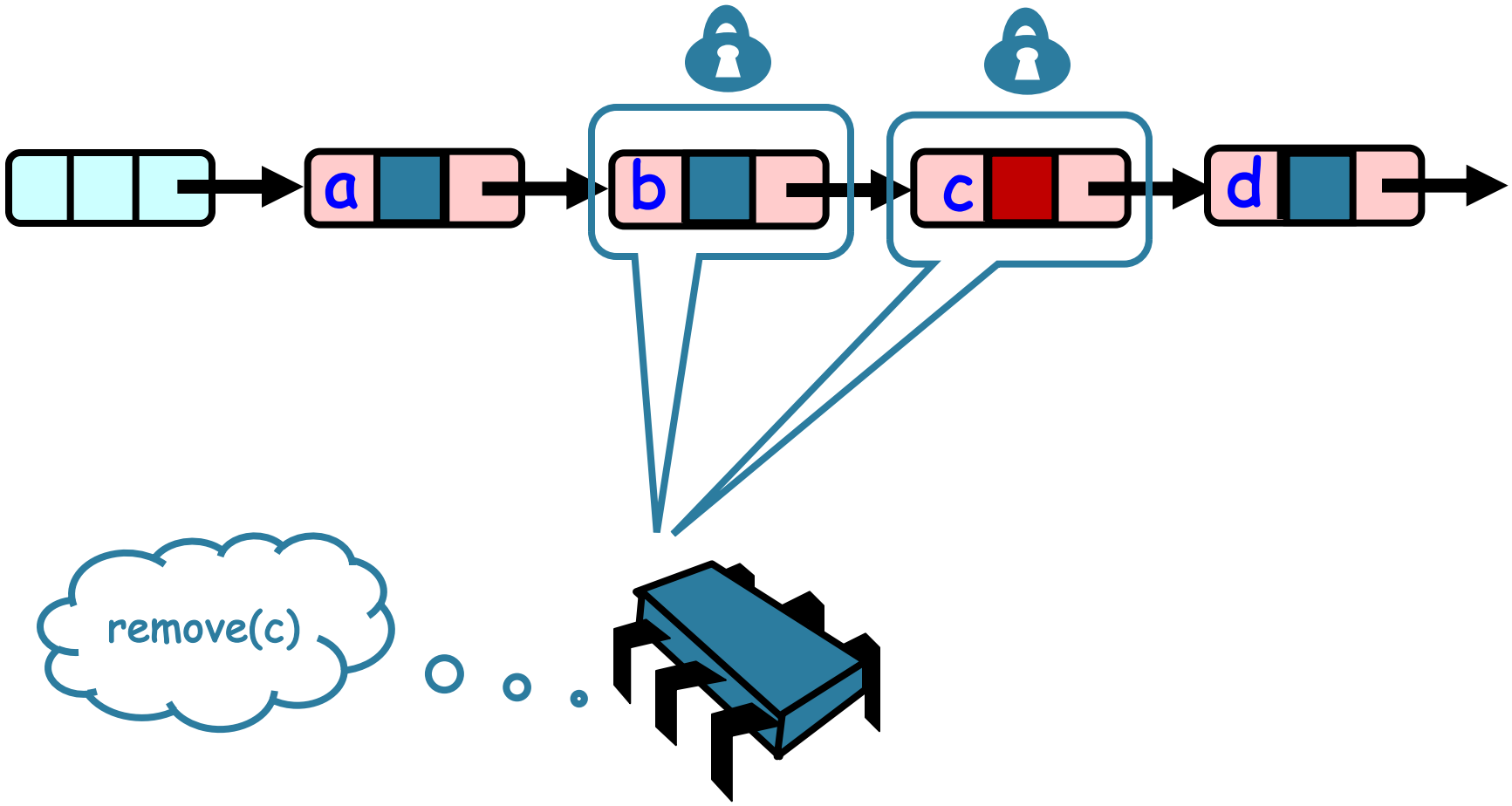


Lazy removal : 3) Check everything is OK (new)

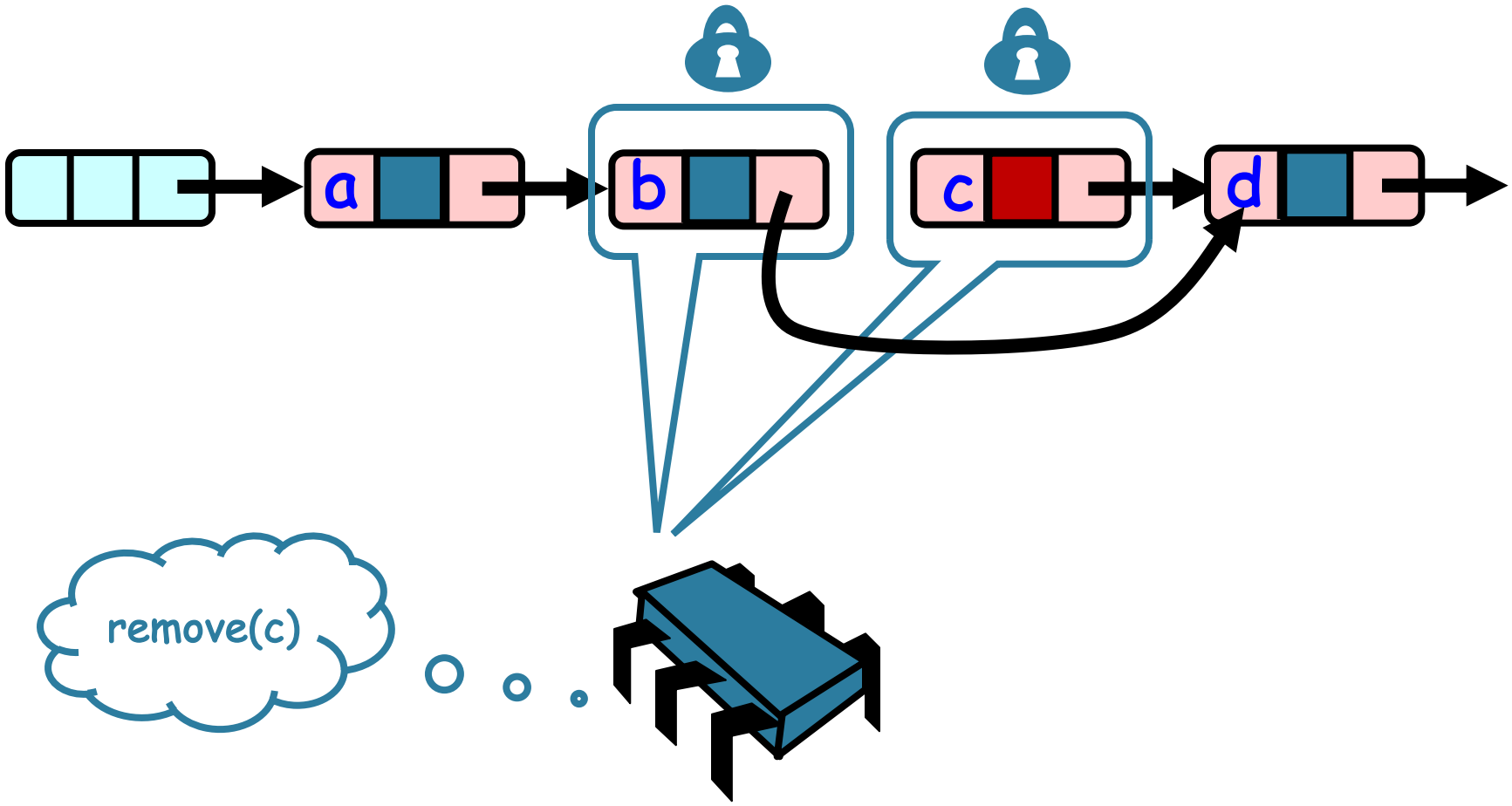


New invariant → Linearization:
- c still reachable from b
- b not marked
- c not marked

Lazy removal : 4) Logical delete (new)



Lazy removal : 4) Physical delete



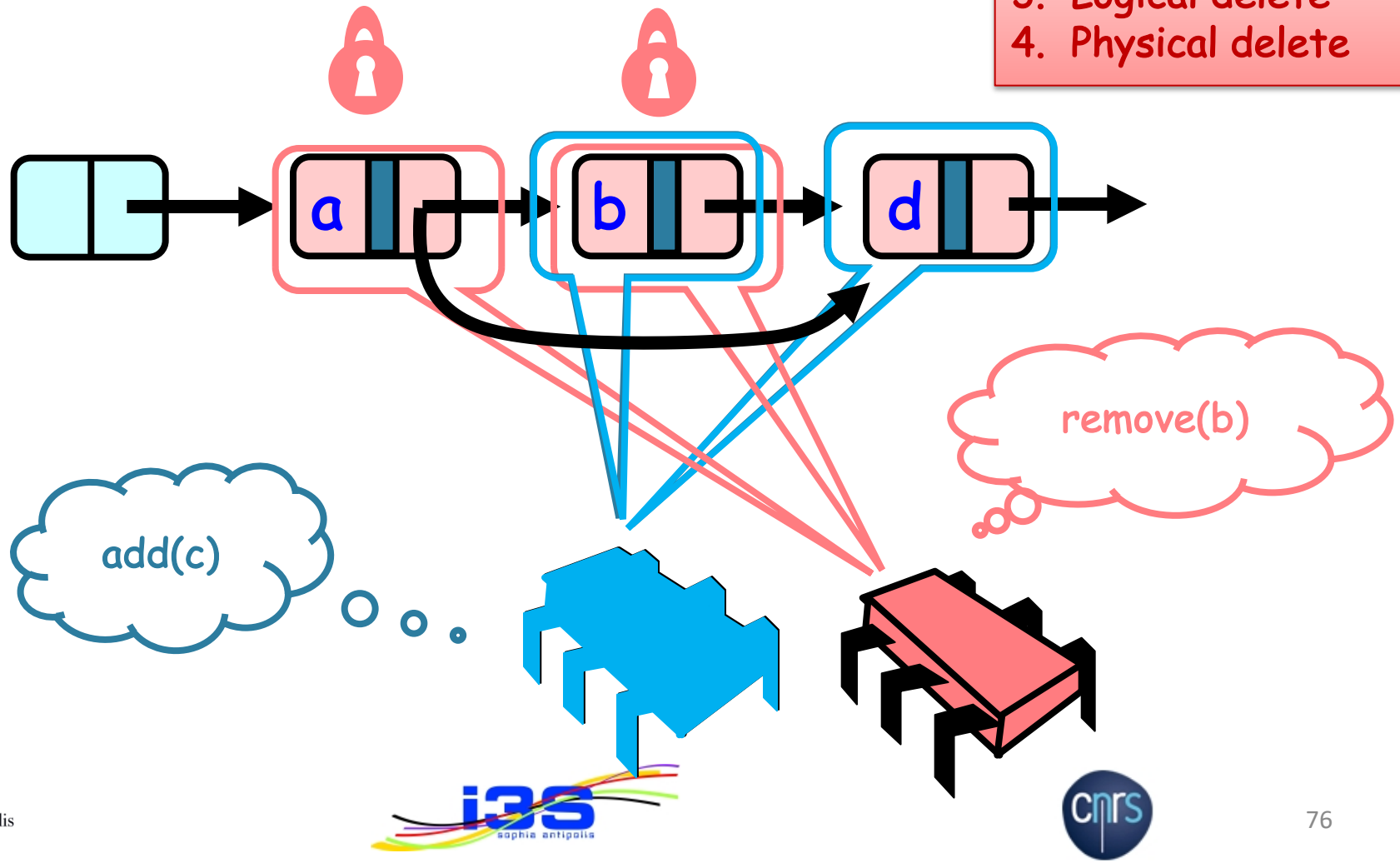
Remove method (lazy synchronization)

```
public boolean remove(Item item) {
  int key = item.hashCode();
  Node pred = this.head;
  Node curr = pred.next;
  while (curr.key <= key) {
    if (item == curr.item) break;
    pred = curr; curr = curr.next;
  }
  try {
    pred.lock(); curr.lock();
    if (validate(pred,curr)) {
      if (curr.item == item) {
        curr.marked = true;
        pred.next = curr.next;
        return true;
      } else {
        return false;
  }}} finally {
    pred.unlock();
    curr.unlock();
}}
```

```
private boolean
  validate(Node pred,
           Node curr) {
  Node node = head;
  return !pred.marked &&
    !curr.marked &&
    pred.next = curr;
  }
}
```

Is logical removal necessary ?
Some possible execution without.

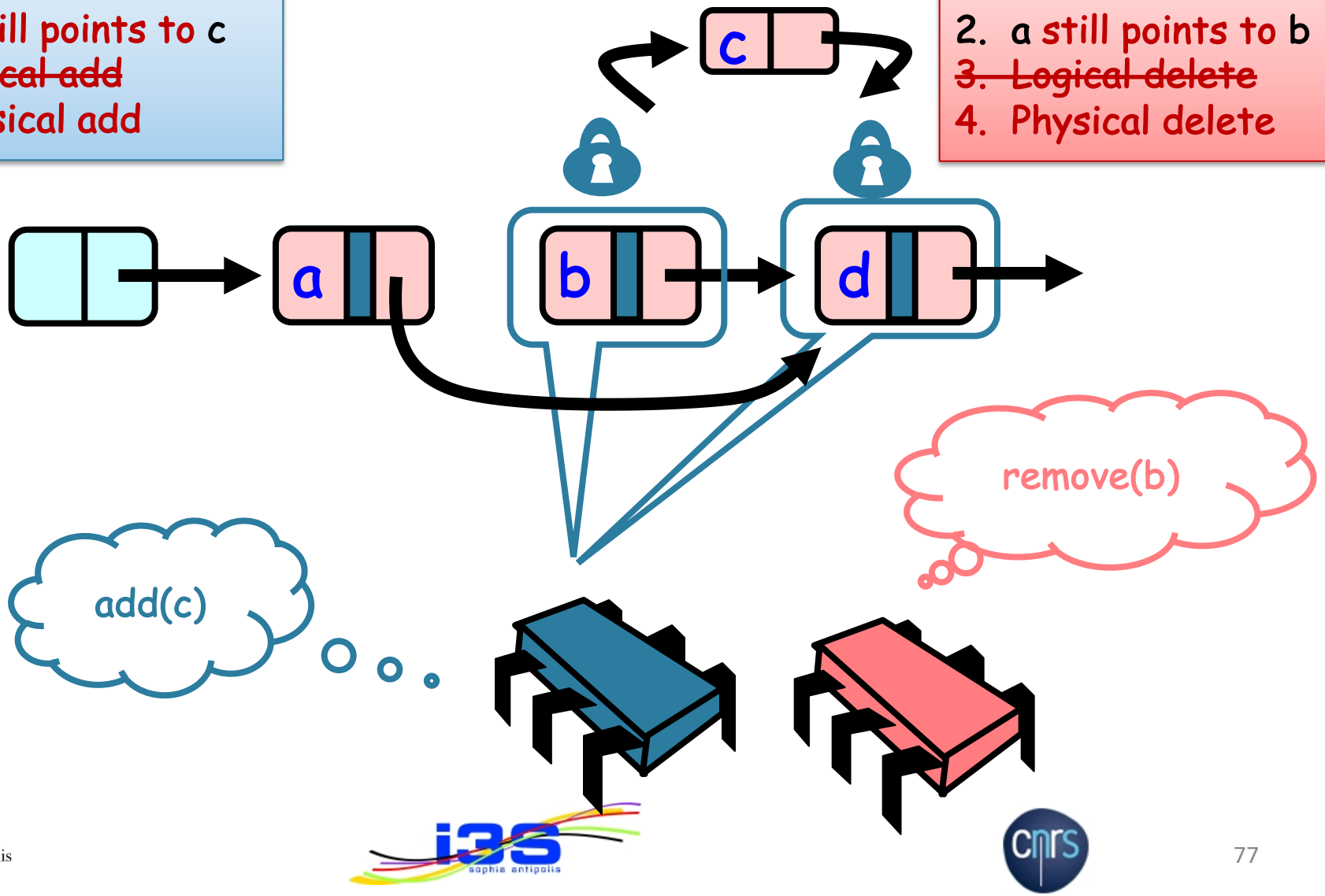
1. a & b not marked
2. a still points to b
3. ~~Logical delete~~
4. Physical delete



Is logical removal necessary ?
Some possible execution without.

1. b & c not marked
2. b still points to c
3. ~~Logical add~~
4. Physical add

1. a & b not marked
2. a still points to b
3. ~~Logical delete~~
4. Physical delete



Evaluation
Lazy add() and remove()

- Good:
  - contains() doesn't lock
  - In fact, its wait-free!
  - Good because typically high % contains()
  - Uncontended calls don't re-traverse
- Bad
  - Contended add() and remove() calls do re-traverse
  - Traffic jam if one thread
    - Enters critical section
    - And "eats the big muffin"
      - Cache miss, page fault, descheduled …

But wait-free contains()

Use Mark bit + list ordering

1. Not marked → in the set
2. Marked or missing → not in the set

1. Coarse-Grained Synchronization
2. Fine-Grained Synchronization
3. Optimistic Synchronization
4. Lazy Synchronization
5. Lock-Free Synchronization

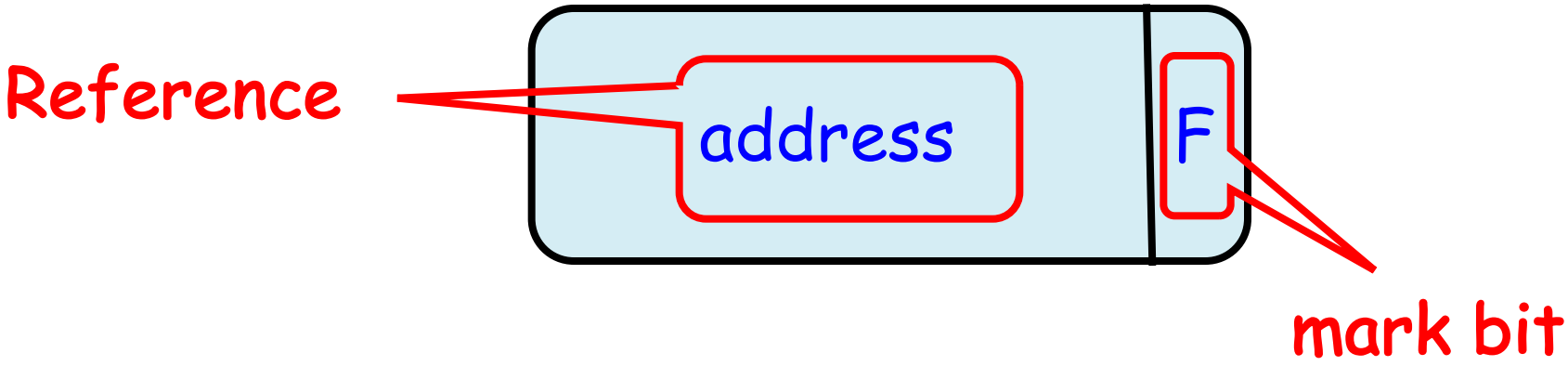
4 new approaches

5) Lock-free Lists

- Next logical step
  - Wait-free contains()
  - lock-free add() and remove()
- Solution use `AtomicMarkableReference`
  - Atomically
    - Swing reference and
    - Update flag
  - Remove in two steps
    - Set mark bit in next field
    - Redirect predecessor's pointer
- ATTENTION :
  - don't use `AtomicReference`
  - or `compareAndSet()` from `AtomicReference`

Marking a Node

- AtomicMarkableReference class
  - Java.util.concurrent.atomic package



Marking a Node

- Extracting Reference & Mark
  - `public Object get(boolean[] marked);`

Returns reference

Returns mark at array index 0!

- Extracting Mark Only
  - `public boolean isMarked();`

Value of mark

Marking a Node

- Changing State
  - Public boolean compareAndSet(
    Object expectedRef,
    Object updateRef,
    boolean expectedMark,
    boolean updateMark);

If this is the current reference …

…then change to this new reference …

And this is the current mark …

… and this new mark

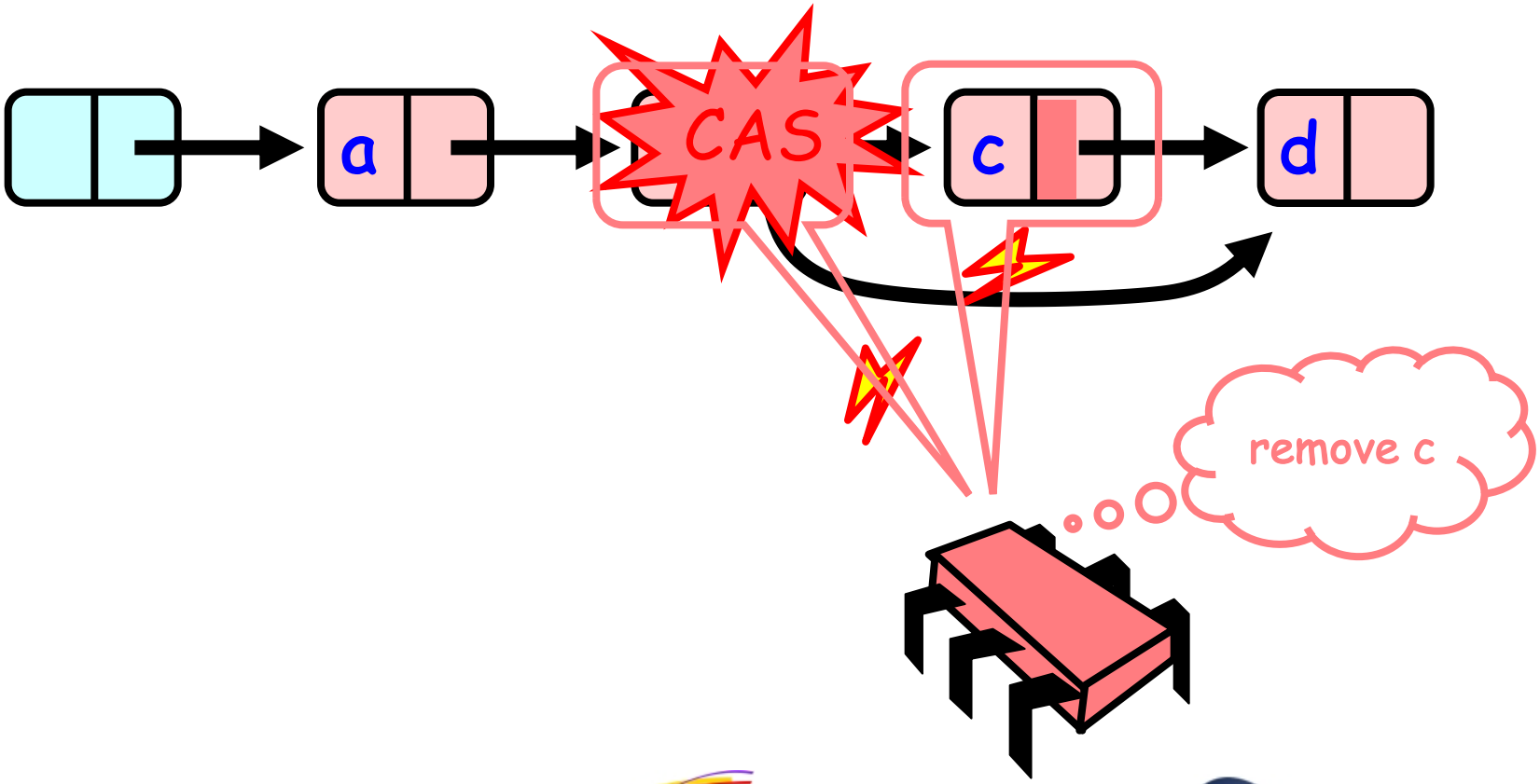
  - public boolean attemptMark(
    Object expectedRef,
    boolean updateMark);

If this is the current reference …

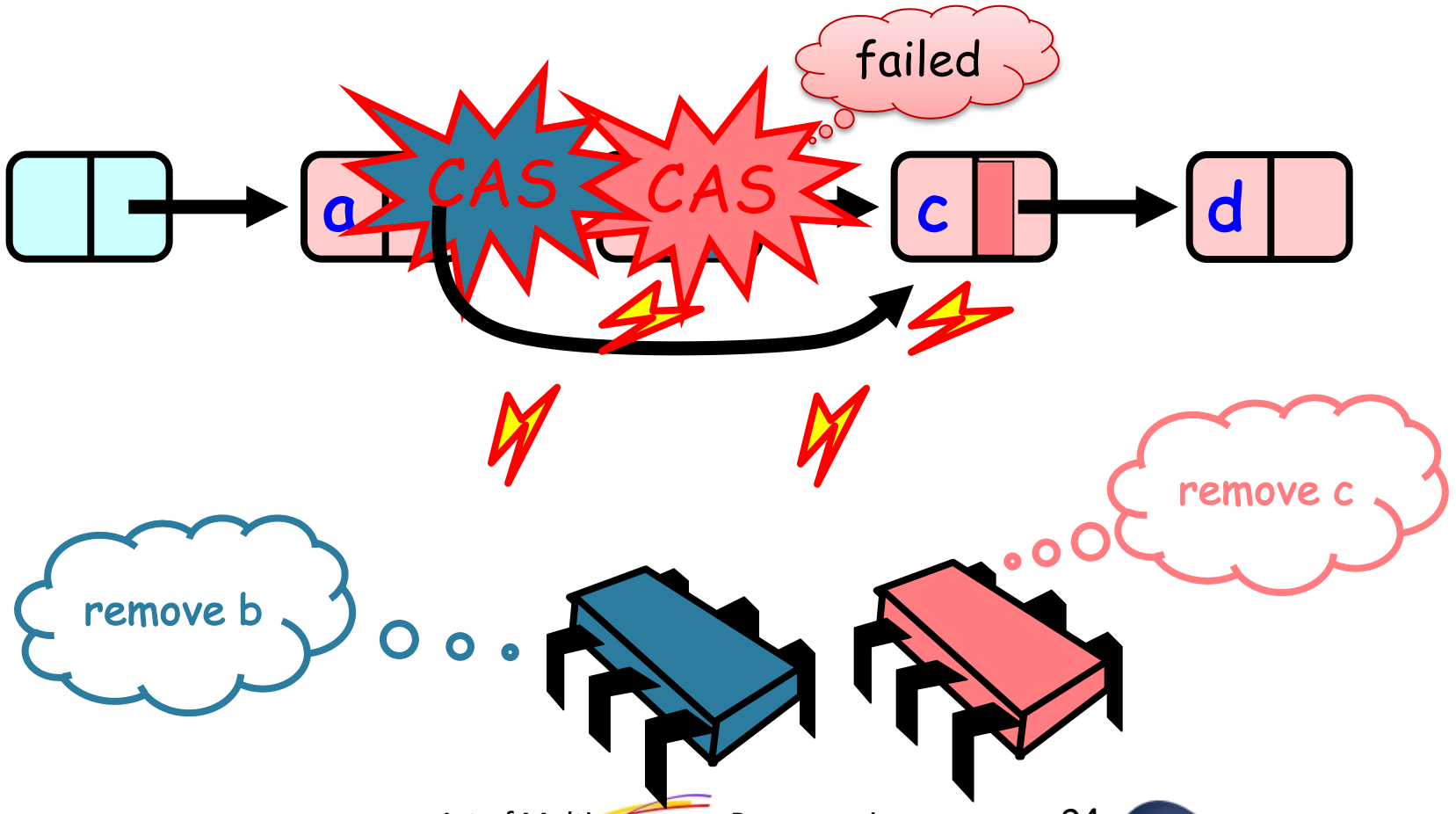
.. then change to this new mark.

Removing a Node

1. Found the position
2. AttempMark()
3. CompareAndSet()



Removing a Node : trying with two thread

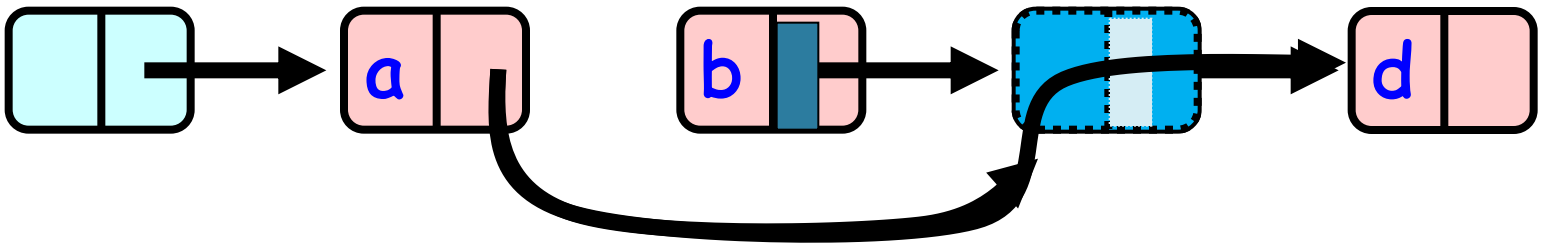


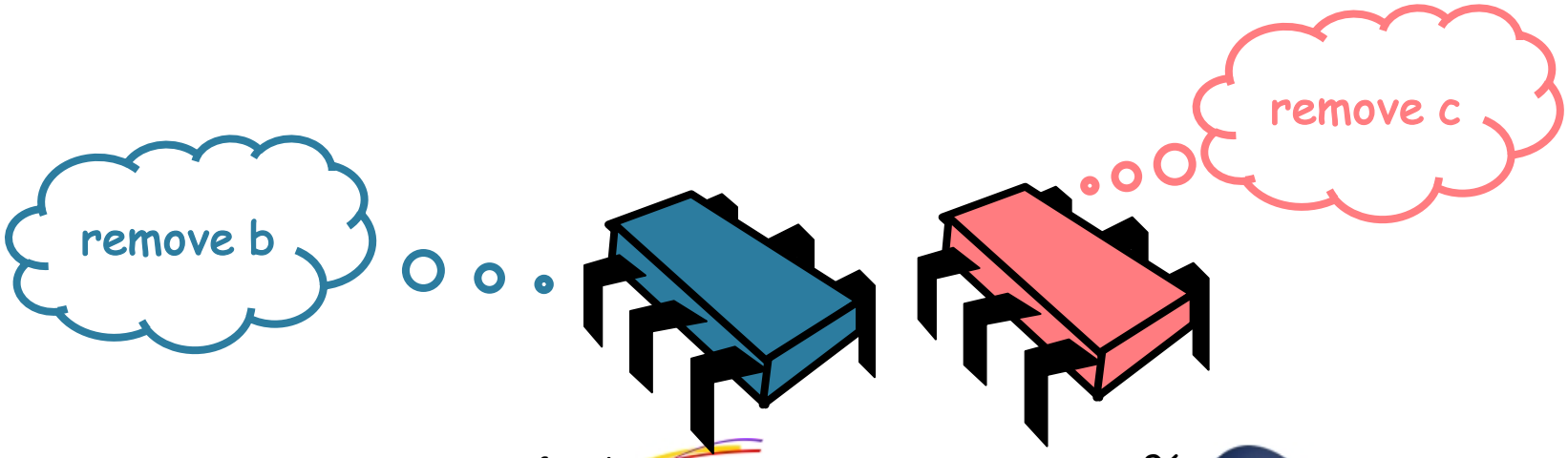
Removing a Node : trying with two thread

Traversing the List

- Q: what do you do when you find a "logically" deleted node in your path?
- A: finish the job.
  - CAS the predecessor's next field
  - Proceed (repeat as needed)

Removing a Node : trying with two thread





Window class

```
Class Window {
  public Node pred;
  public Node curr;
  Window (Node pred, Node curr) {
    this.pred = pred;
    this.curr = curr;
  }
}
```

Remove method (lock-free synchronization)

```
public boolean remove(Item item) {
  boolean snip;
  int key = item.hashCode();
  while (true) {
    Window window = find(this.head, key);
    Node pred = window.pred;
    Node curr = window.curr;
    if (curr.key != key) {
       return false;
    } else {
      Node succ = curr.next.getReference();
      snip = curr.next.attempMark(succ, true);
      if (snip) {
        pred.next.comparAndSet(curr, succ,
                               false, false);
        return true;
      }
    }
  }
}
```

Find neighbors

Try to mark node as deleted

delete

Add (lock-free synchronization)

```
public boolean add(T item) {
  boolean splice;
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred;
    Node curr = window.curr;
    if (curr.key == key) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = new AtomicMarkableRef(curr, false);
      if (pred.next.compareAndSet
            (curr, node, false, false)) {
        return true;
      }
    }
  }
}
```

Find neighbors

Create node

Install the new node

Lock-free Find

```
public Window find(Node head, int key) {
  Node pred = null;
  Node curr = null;
  Node succ = null;
  boolean[] marked = {false};
  boolean snip;
  retry: while (true) {
    pred = head;
    curr = pred.next.getReference();
    while (true) {
      succ = curr.next.get(marked);
      while (marked[0]) {
      …
      }
      if (curr.key >= key) return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
  }
}
```

If list changes while traversed, start over

Lock-Free because we start over only if someone else makes progress

Try to remove deleted nodes in path…

code details soon

Lock-free Find

```
while (marked[0]) {
  snip = pred.next.compareAndSet(curr, succ,
                                 false, false);

  if (!snip) continue retry;

  curr = succ;
  succ = curr.next.get(marked);
}
```

Try to snip out node

if predecessor's next field changed must retry whole traversal

Otherwise move on to check if next node deleted

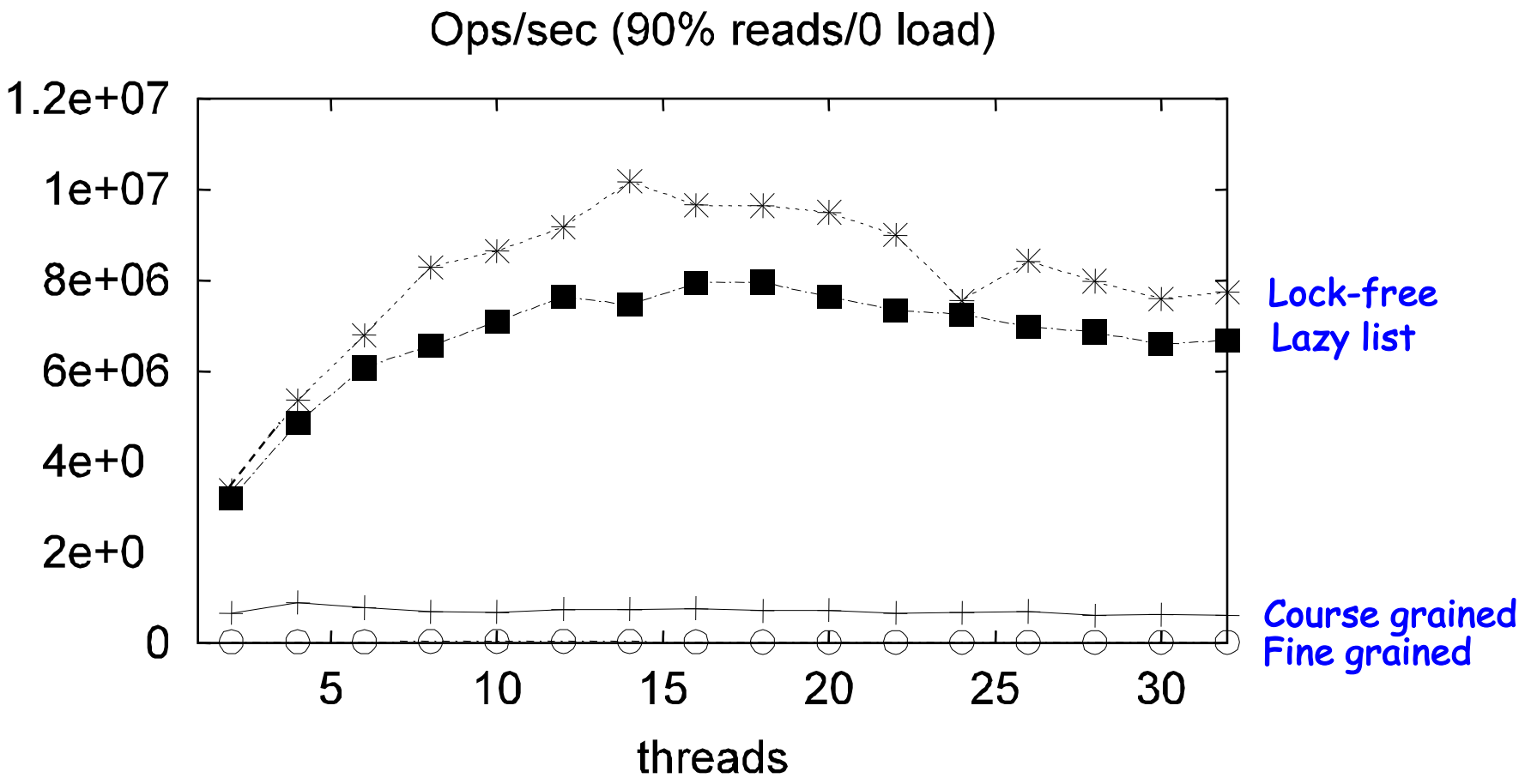
Contains (wait-free synchronization)

```
public boolean contains(Node item) {
  boolean marked;
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) curr = curr.next;
  Node succ = curr.next.get(marked);
  return (curr.key == key && !marked[0])
}
```

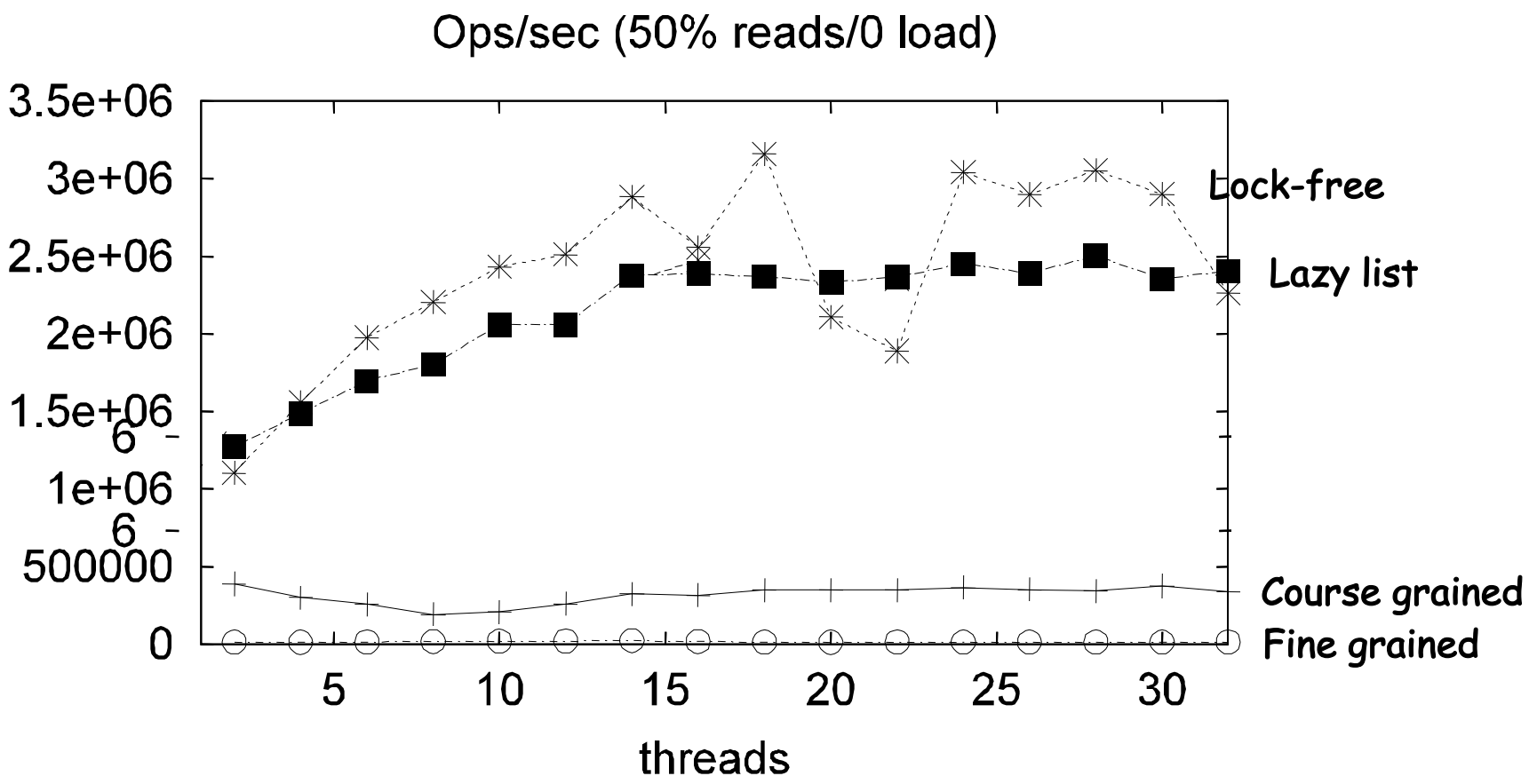
Performance studies

- On 16 node shared memory machine
  - Benchmark throughput of Java List-based Set
  - algs. vary % of Contains() method Calls.

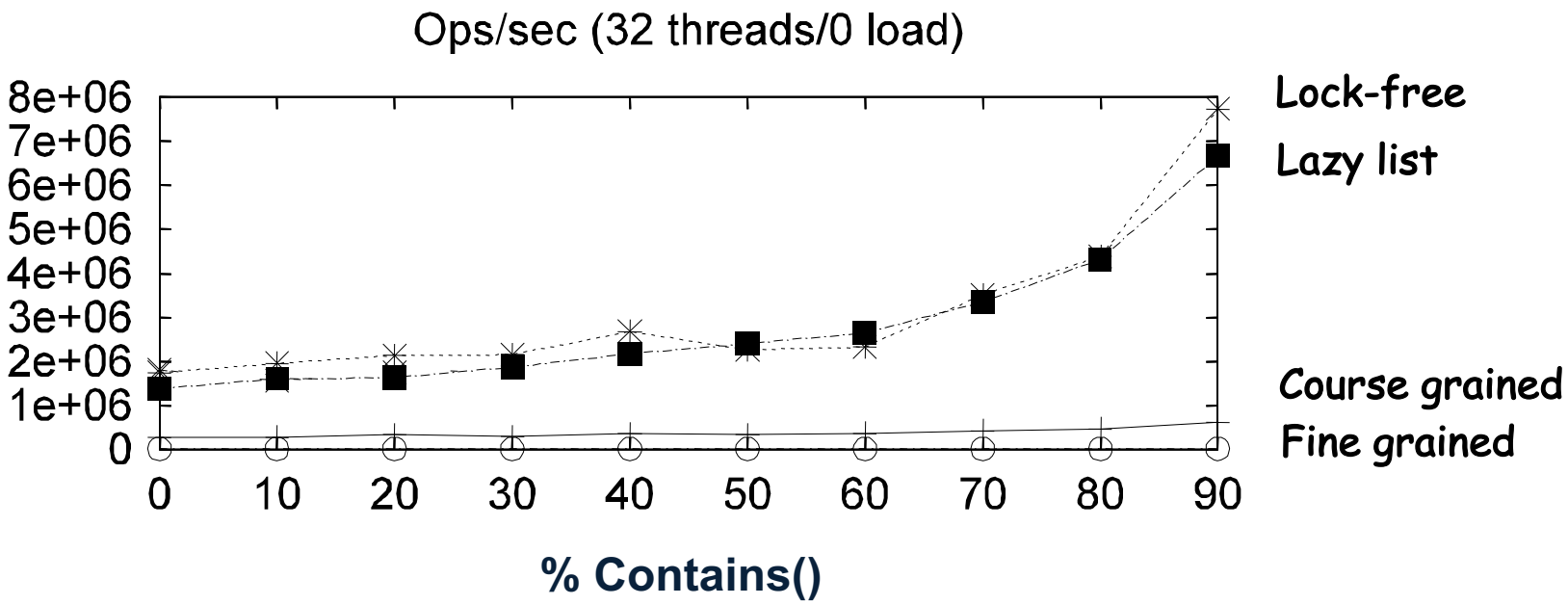
High Contains Ratio



Low Contains Ratio



As Contains Ratio Increases



Summary

- We have looked inside
  0. Coarse-grained locking
  1. Fine-grained locking
  2. Optimistic synchronization
  3. Lazy synchronization
  4. Lock-free synchronization
- Not all algorithms are presented
  - but they are in the Maurice Herlihy web site

		Pro	Cons
Concurrent Data Structure	With lock	simple' programming mode	false conflicts sequential bottleneck
	Lock free	Resilient to failures More efficient	Often complex Memory consuming Sometime, weak progress condition

Summary

Q. Est-ce réellement utilisé ou est-ce un jeu pour les chercheurs ?

R. Pas de réponse générique mais :

- Des bibliothèques pour différentes structures de données usuelles sont construites ou en cours de construction (file, pile, queue, arbre, etc).
  - Exemple : http://www.google.com/patents/US7533138
  - Attention : il ne s'agit pas d'un brevet google mais d'un site google qui recense les brevets
- Plusieurs OS sont proposés en particulier lorsque les verrouillages ne sont pas possibles (traitement des exceptions)
  - OS pour le traitement de l'audio par exemple
  - http://www.rossbencina.com/code/lockfree

Q. Que se passe-t-il dans le projet si l'on calcule avec plusieurs threads et que l'on supprime la synchronisation (i.e. lectures non cohérentes) ?

R. On parle alors de calcul asynchrone (études sur la convergence dans Journal of Parallel and Distributed Computing en 1996 et Mathematics of Computation en 1998)

- En Java
- http://howtodoinjava.com/core-java/multi-threading/non-blocking-thread-safe-list-concurrentlinkeddeque-example/
  - Deque : A linear collection that supports element insertion and removal at both ends.
  - ArrayDeque : Resizable-array implementation of the Deque interface
  - LinkedList : Doubly-linked list implementation of the List and Deque interfaces.
  - LinkedBlockingDeque : An optionally-bounded blocking deque based on linked nodes.
  - ConcurrentLinkedDeque : An unbounded concurrent deque based on linked nodes.

Q&A

http://www.i3s.unice.fr/~riveill

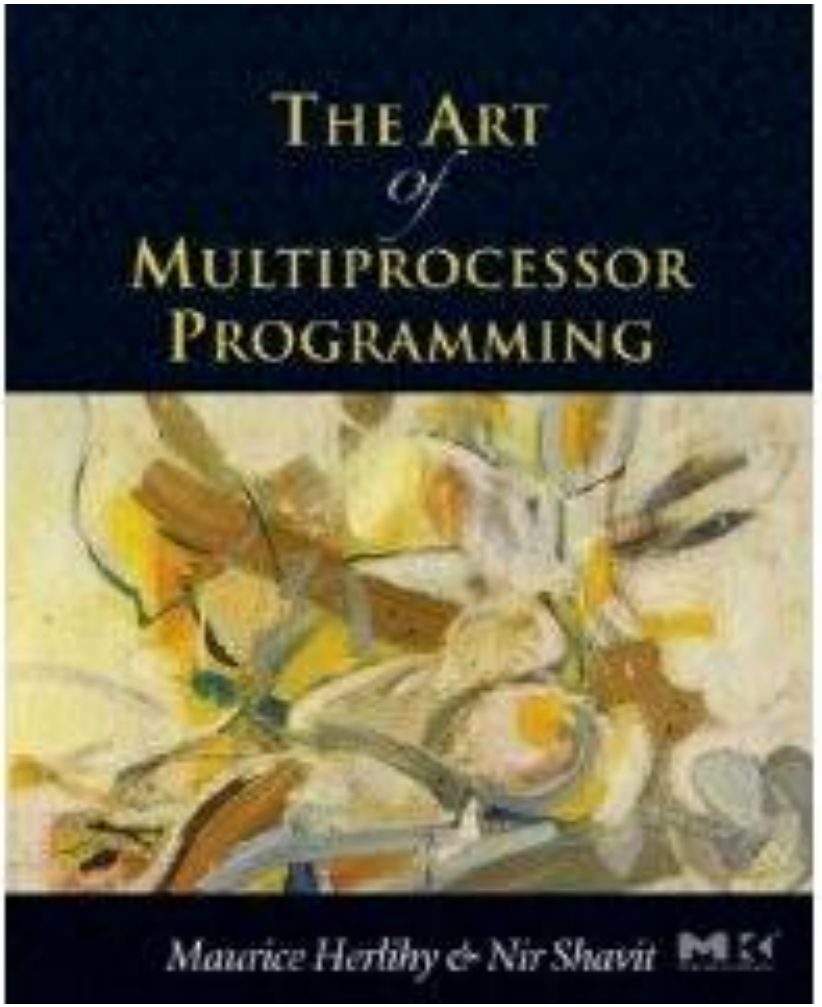
Université Nice Sophia Antipolis

cnrs

Livres utilisés

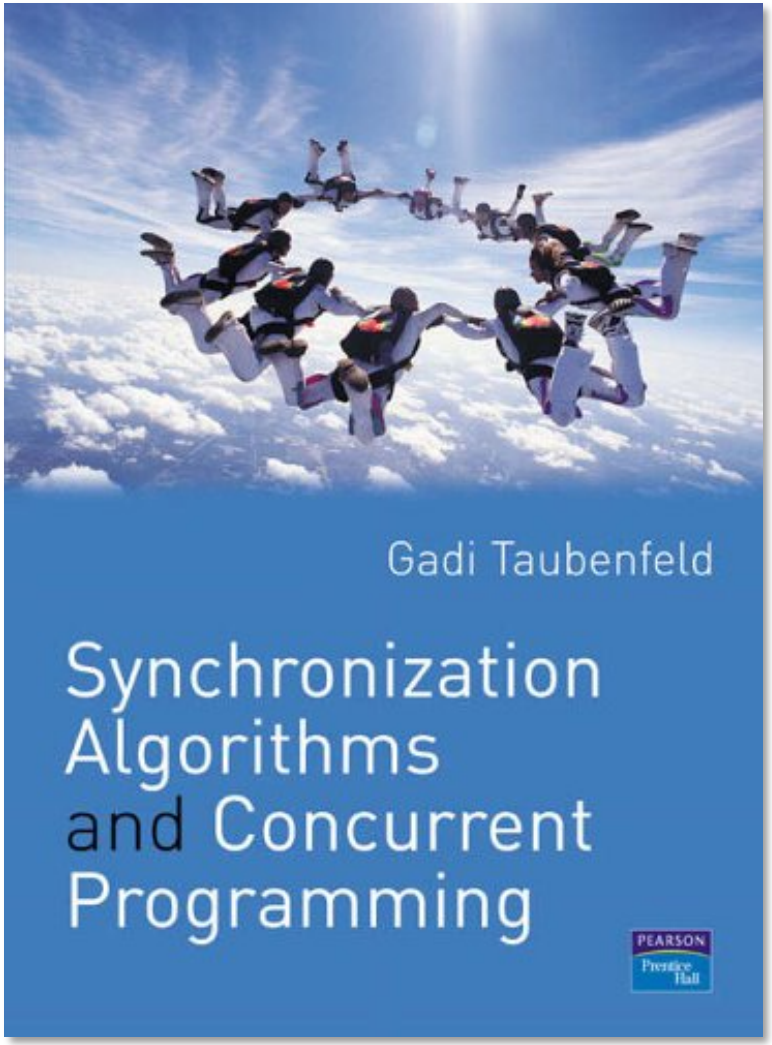
- Maurice Herlihy
- Professor
- Brown University, USA

http://cs.brown.edu/~mph/



Livres utilisés

- *Gadi Taubenfeld*
- Professor
- ICH, Israel



To get the most updated version of these slides go to:

http://www.faculty.idc.ac.il/gadi/book.htm

Université Nice Sophia Antipolis

i3s sophia antipolis

cnrs