Design of Parallel and High-Performance Computing

Fall 2016

Lecture: Locks and Lock-Free

Motivational video: *https://www.youtube.com/watch?v=jhApQIPQquw*

Instructor: Torsten Hoefler & Markus Püschel

TA: Salvatore Di Girolamo

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Administrivia

- **Final project presentation: Monday 12/19 during last lecture**
  - Report will be due in January!

    Still, starting to write early is very helpful --- write – rewrite – rewrite (no joke!)

  - Some more ideas what to talk about:

    What tools/programming language/parallelization scheme do you use?

    Which architecture? (we only offer access to Xeon Phi, you may use different)

    How to verify correctness of the parallelization?

    How to argue about performance (bounds, what to compare to?)

    (Somewhat) realistic use-cases and input sets?

    What are the key concepts employed?

    What are the main obstacles?

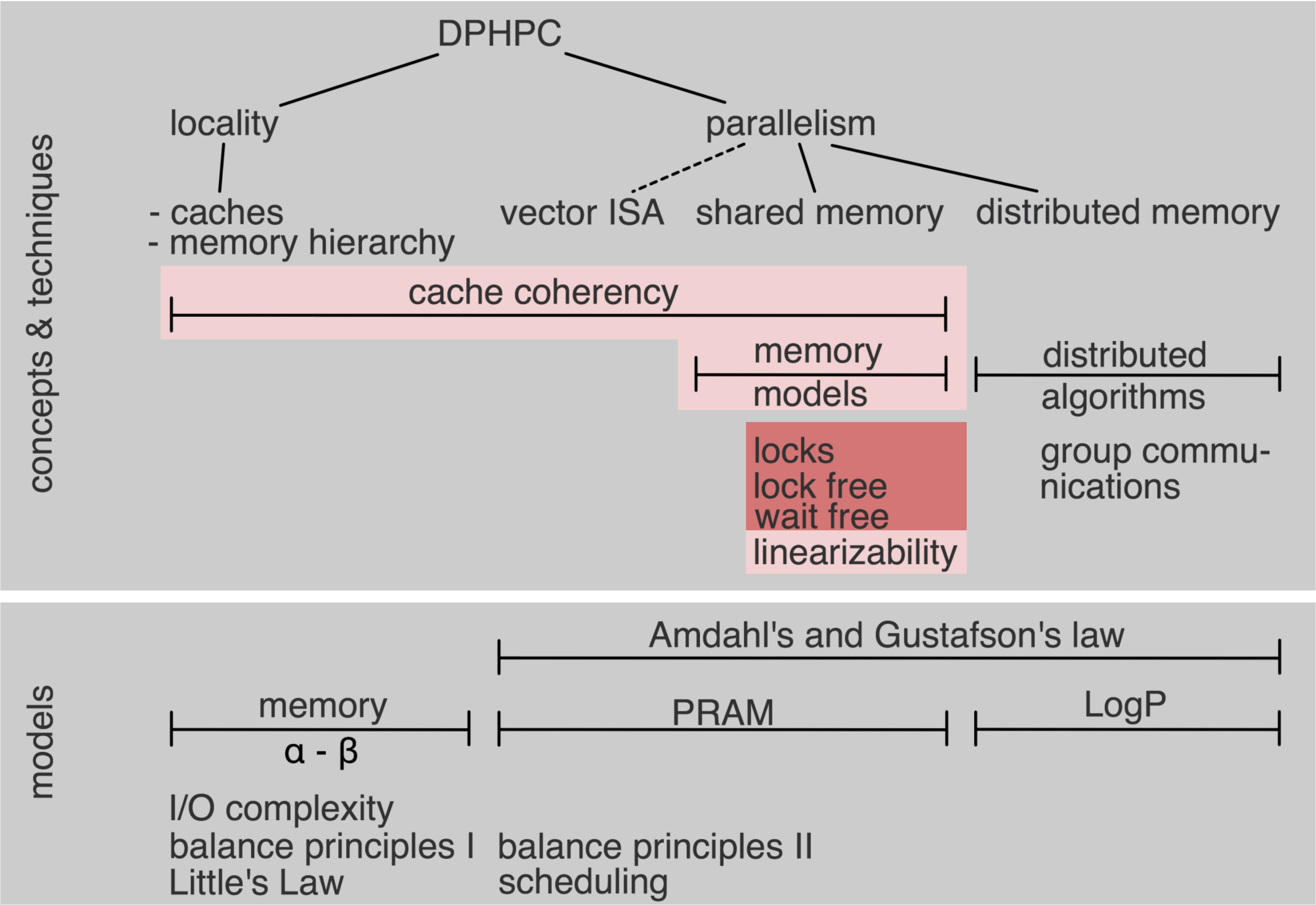
Review of last lecture

- **Language memory models**
  - Java/C++ memory model overview
  - Synchronized programming

- **Locks**
  - Broken two-thread locks
  - Peterson
  - N-thread locks (filter lock)
  - Many different locks, strengths and weaknesses
  - Lock options and parameters

- **Formal proof methods**
  - Correctness (mutual exclusion as condition)
  - Progress

DPHPC Overview



concepts & techniques

DPHPC

locality

parallelism

- caches
- memory hierarchy

vector ISA

shared memory

distributed memory

cache coherency

memory models

distributed algorithms

locks
lock free
wait free

linearizability

group communications

models

Amdahl's and Gustafson's law

memory

α - β

PRAM

LogP

I/O complexity
balance principles I
Little's Law

balance principles II
scheduling

Goals of this lecture

- **More N-thread locks!**
  - Hardware operations for concurrency control
- **More on locks (using advanced operations)**
  - Spin locks
  - Various optimized locks
- **Even more on locks (issues and extended concepts)**
  - Deadlocks, priority inversion, competitive spinning, semaphores
- **Case studies**
  - Barrier, reasoning about semantics
- **Locks in practice: a set structure**

Lock Fairness

- **Starvation freedom provides no guarantee on how long a thread waits or if it is “passed”!**
- **To reason about fairness, we define two sections of each lock algorithm:**
  - Doorway D (bounded # of steps)
  - Waiting W (unbounded # of steps)

```
void lock() {
  int j = 1 - tid;
  flag[tid] = true; // I’m interested
  victim = tid;      // other goes first
  while (flag[j] && victim == tid) {};
}
```

- **FIFO locks:**
  - If T_A finishes its doorway before T_B the CR_A → CR_B
  - Implies fairness

Lamport’s Bakery Algorithm (1974)

- Is a FIFO lock (and thus fair)
- Each thread takes a number in doorway and threads enter in the order of their number!

```
volatile int flag[n] = {0,0,…,0};
volatile int label[n] = {0,0,….,0};

void lock() {
  flag[tid] = 1; // request
  label[tid] = max(label[0], ...,label[n-1]) + 1; // take ticket
  while ((∃k != tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {};
}
public void unlock() {
  flag[tid] = 0;
}
```

Lamport's Bakery Algorithm

- **Advantages:**
  - Elegant and correct solution
  - Starvation free, even FIFO fairness

- **Not used in practice!**
  - Why?
  - Needs to read/write N memory locations for synchronizing N threads
  - Can we do better?

    Using only atomic registers/memory

A Lower Bound to Memory Complexity

- Theorem 5.1 in [1]: *“If S is a [atomic] read/write system with at least two processes and S solves mutual exclusion with global progress [deadlock-freedom], then S must have at least as many variables as processes”*

- So we’re doomed! Optimal locks are available and they’re fundamentally non-scalable. Or not?

[1] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. Information and Computation, 107(2):171–184, December 1993

Hardware Support?

- **Hardware atomic operations:**
  - Test&Set

    Write const to memory while returning the old value
  - Atomic swap

    Atomically exchange memory and register
  - Fetch&Op

    Get value and apply operation to memory location
  - Compare&Swap

    Compare two values and swap memory with register if equal
  - Load-linked/Store-Conditional LL/SC

    Loads value from memory, allows operations, commits only if no other updates committed → mini-TM
  - Intel TSX (transactional synchronization extensions)

    Hardware-TM (roll your own atomic operations)

Relative Power of Synchronization

- **Design-Problem I: Multi-core Processor**
  - Which atomic operations are useful?
- **Design-Problem II: Complex Application**
  - What atomic should I use?
- **Concept of “consensus number” C if a primitive can be used to solve the “consensus problem” in a finite number of steps (even if threads stop)**
  - atomic registers have C=1 (thus locks have C=1!)
  - TAS, Swap, Fetch&Op have C=2
  - CAS, LL/SC, TM have C=∞

Test-and-Set Locks

- **Test-and-Set semantics**
  - Memoize old value
  - Set fixed value TASval (true)
  - Return old value
- **After execution:**
  - Post-condition is a fixed (constant) value!

```
bool test_and_set (bool *flag) {
  bool old = *flag;
  *flag = true;
  return old;
} // all atomic!
```

Test-and-Set Locks

- **Assume TASval indicates “locked”**
- **Write something else to indicate “unlocked”**
- **TAS until return value is != TASval**

- **When will the lock be granted?**
- **Does this work well in practice?**

```
volatile int lock = 0;

void lock() {
  while (TestAndSet(&lock) == 1);
}

void unlock() {
  lock = 0;
}
```

Contention

- **On x86, the XCHG instruction is used to implement TAS**
  - For experts: x86 LOCK is superfluous!
- **Cacheline is read and written**
  - Ends up in exclusive state, invalidates other copies
  - Cacheline is “thrown” around uselessly
  - High load on memory subsystem

    x86 bus lock is essentially a full memory barrier ☹

```
movl    $1, %eax
xchg    %eax, (%ebx)
```

Schweizer, Besta, Hoefler: “Evaluating the Cost of Atomic Operations on Modern Architectures”, PACT’15

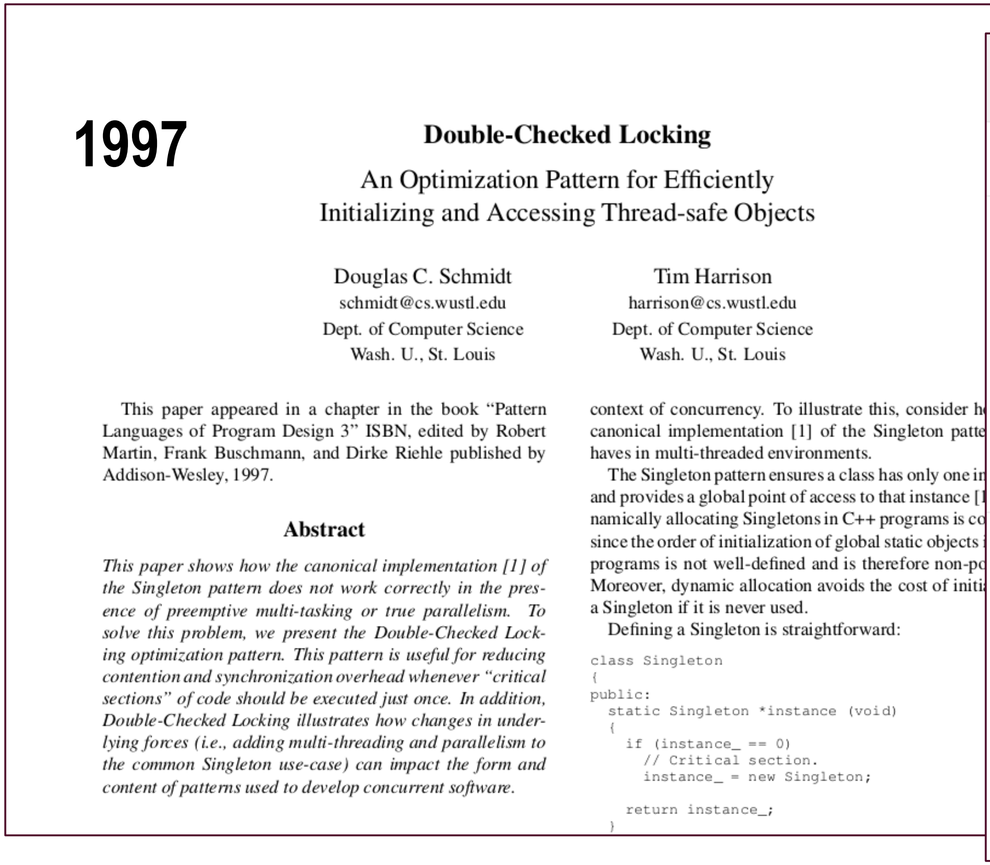
Test-and-Test-and-Set (TATAS) Locks

- **Spinning in TAS is not a good idea**
- **Spin on cache line in shared state**
  - All threads at the same time, no cache coherency/memory traffic
- **Danger!**
  - Efficient but use with great care!
  - Generalizations are dangerous

```
volatile int lock = 0;

void lock() {
  do {
    while (lock == 1);
  } while (TestAndSet(&lock) == 1);
}

void unlock() {
  lock = 0;
}
```

Warning: Even Experts get it wrong!

- **Example: Double-Checked Locking**



1997

Double-Checked Locking

An Optimization Pattern for Efficiently
Initializing and Accessing Thread-safe Objects

Douglas C. Schmidt
schmidt@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

Tim Harrison
harrison@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

This paper appeared in a chapter in the book "Pattern Languages of Program Design 3" ISBN, edited by Robert Martin, Frank Buschmann, and Dirke Riehle published by Addison-Wesley, 1997.

Abstract

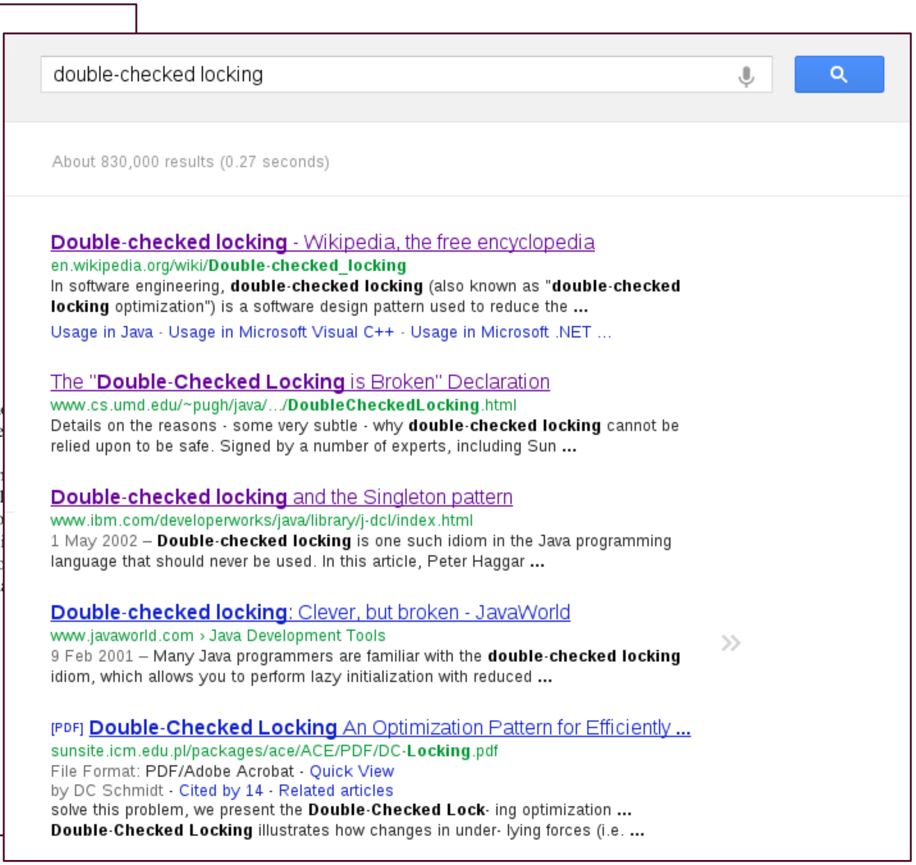
This paper shows how the canonical implementation [1] of the Singleton pattern does not work correctly in the presence of preemptive multi-tasking or true parallelism. To solve this problem, we present the Double-Checked Locking optimization pattern. This pattern is useful for reducing contention and synchronization overhead whenever "critical sections" of code should be executed just once. In addition, Double-Checked Locking illustrates how changes in underlying forces (i.e., adding multi-threading and parallelism to the common Singleton use-case) can impact the form and content of patterns used to develop concurrent software.

context of concurrency. To illustrate this, consider h
canonical implementation [1] of the Singleton patte
haves in multi-threaded environments.

The Singleton pattern ensures a class has only one in
and provides a global point of access to that instance [
namically allocating Singletons in C++ programs is co
since the order of initialization of global static objects i
programs is not well-defined and is therefore non-po
Moreover, dynamic allocation avoids the cost of initi
a Singleton if it is never used.

Defining a Singleton is straightforward:

```
class Singleton
{
public:
  static Singleton *instance (void)
  {
    if (instance_ == 0)
      // Critical section.
      instance_ = new Singleton;

    return instance_;
  }
```



About 830,000 results (0.27 seconds)

Double-checked locking - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/**Double-checked_locking**
In software engineering, **double-checked locking** (also known as "**double-checked locking** optimization") is a software design pattern used to reduce the **...**
Usage in Java · Usage in Microsoft Visual C++ · Usage in Microsoft .NET ...

The "**Double-Checked Locking** is Broken" Declaration
www.cs.umd.edu/~pugh/java/.../**DoubleCheckedLocking**.html
Details on the reasons - some very subtle - why **double-checked locking** cannot be relied upon to be safe. Signed by a number of experts, including Sun **...**

Double-checked locking and the Singleton pattern
www.ibm.com/developerworks/java/library/j-dcl/index.html
1 May 2002 – **Double-checked locking** is one such idiom in the Java programming language that should never be used. In this article, Peter Haggar **...**

Double-checked locking: Clever, but broken - JavaWorld
www.javaworld.com › Java Development Tools
9 Feb 2001 – Many Java programmers are familiar with the **double-checked locking** idiom, which allows you to perform lazy initialization with reduced **...**

[PDF] **Double-Checked Locking** An Optimization Pattern for Efficiently ...
sunsite.icm.edu.pl/packages/ace/ACE/PDF/DC-**Locking**.pdf
File Format: PDF/Adobe Acrobat - Quick View
by DC Schmidt - Cited by 14 - Related articles
solve this problem, we present the **Double-Checked Lock**- ing optimization **...**
Double-Checked Locking illustrates how changes in under- lying forces (i.e. **...**

Problem: Memory ordering leads to race-conditions!

Contention?

- **Do TATAS locks still have contention?**
- **When lock is released, k threads fight for cache line ownership**
  - One gets the lock, all get the CL exclusively (serially!)
  - What would be a good solution? (think “collision avoidance”)

```
volatile int lock = 0;

void lock() {
  do {
    while (lock == 1);
  } while (TestAndSet(&lock) == 1);
}

void unlock() {
  lock = 0;
}
```

Schweizer, Besta, Hoefler: “Evaluating the Cost of Atomic Operations on Modern Architectures”, PACT’15

TAS Lock with Exponential Backoff

- **Exponential backoff eliminates contention statistically**
  - Locks granted in unpredictable order
  - Starvation possible but unlikely

    How can we make it even less likely?

```
volatile int lock = 0;

void lock() {
  while (TestAndSet(&lock) == 1) {
     wait(time);
     time *= 2; // double waiting time
  }
}

void unlock() {
  lock = 0;
}
```

Similar to: T. Anderson: “The performance of spin lock alternatives for shared-memory multiprocessors”, TPDS, Vol. 1 Issue 1, Jan 1990

TAS Lock with Exponential Backoff

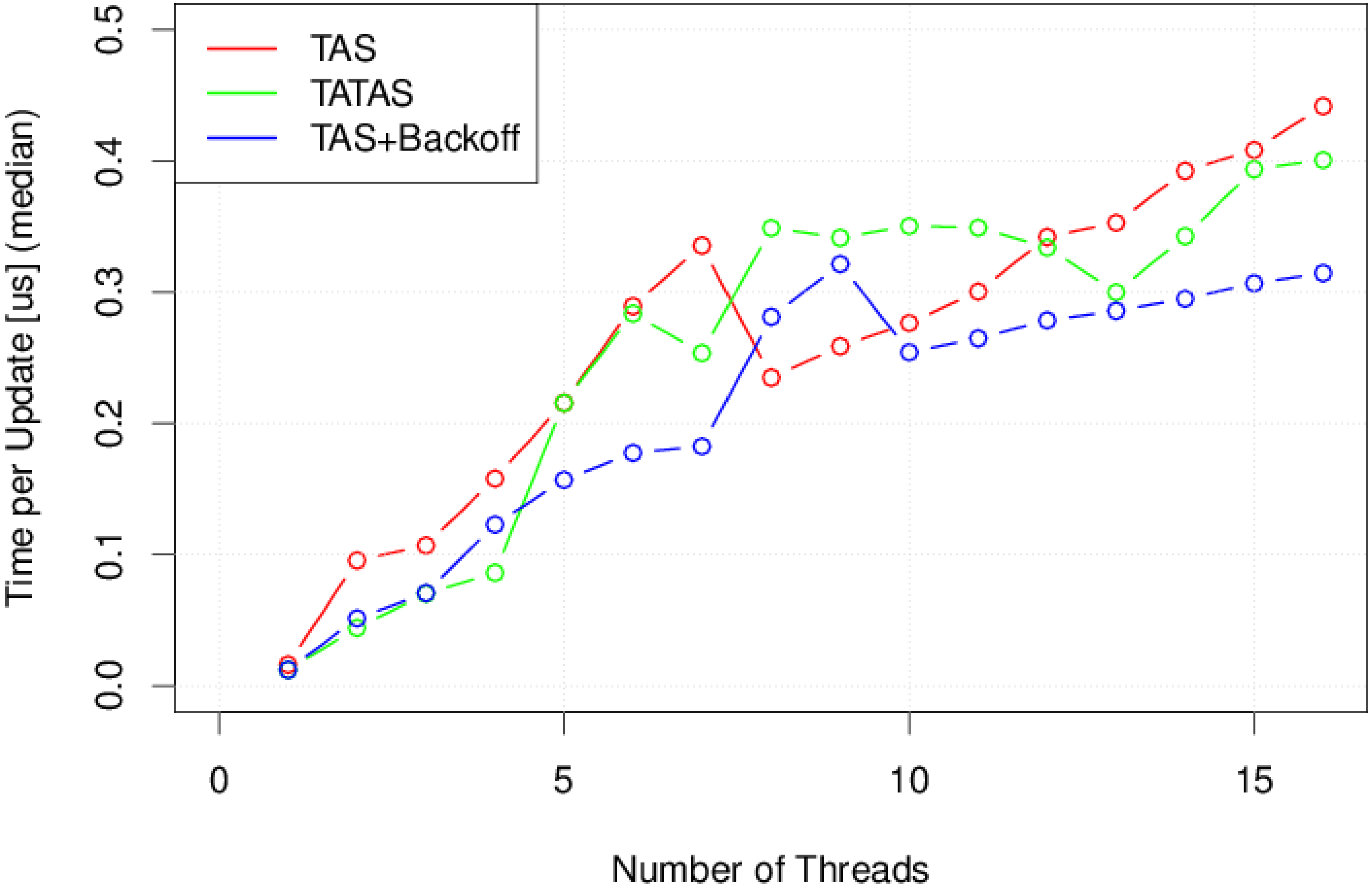
- **Exponential backoff eliminates contention statistically**
  - Locks granted in unpredictable order
  - Starvation possible but unlikely

    Maximum waiting time makes it less likely

```
volatile int lock = 0;
const int maxtime=1000;

void lock() {
  while (TestAndSet(&lock) == 1) {
     wait(time);
     time = min(time * 2, maxtime);
  }
}

void unlock() {
  lock = 0;
}
```

Similar to: T. Anderson: “The performance of spin lock alternatives for shared-memory multiprocessors”, TPDS, Vol. 1 Issue 1, Jan 1990

Comparison of TAS Locks



Improvements?

- **Are TAS locks perfect?**
  - What are the two biggest issues?
  - Cache coherency traffic (contending on same location with expensive atomics)

    -- or --

  - Critical section underutilization (waiting for backoff times will delay entry to CR)
- **What would be a fix for that?**
  - How is this solved at airports and shops (often at least)?
- **Queue locks -- Threads enqueue**
  - Learn from predecessor if it's their turn
  - Each threads spins at a different location
  - FIFO fairness

Array Queue Lock

- **Array to implement queue**
  - Tail-pointer shows next free queue position
  - Each thread spins on own location

    CL padding!
  - index[] array can be put in TLS
- **So are we done now?**
  - What’s wrong?
  - Synchronizing M objects requires Θ(NM) storage
  - What do we do now?

```
volatile int array[n] = {1,0,…,0};
volatile int index[n] = {0,0,…,0};
volatile int tail = 0;

void lock() {
  index[tid] = GetAndInc(tail) % n;
  while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
  array[index[tid]] = 0; // I release my lock
  array[(index[tid] + 1) % n] = 1; // next one
}
```

CLH Lock (1993)

- **List-based (same queue principle)**
- **Discovered twice by Craig, Landin, Hagersten 1993/94**
- **2N+3M words**
  - N threads, M locks
- **Requires thread-local qnode pointer**
  - Can be hidden!

```
typedef struct qnode {
  struct qnode *prev;
  int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
  qn->succ_blocked = 1;
  qn->prev = FetchAndSet(lck, qn);
  while (qn->prev->succ_blocked);
}

void unlock(qnode **qn) {
  qnode *pred = (*qn)->prev;
  (*qn)->succ_blocked = 0;
  *qn = pred;
}
```

CLH Lock (1993)

- **Qnode objects represent thread state!**
  - succ_blocked == 1 if waiting or acquired lock
  - succ_blocked == 0 if released lock
- **List is implicit!**
  - One node per thread
  - Spin location changes

    NUMA issues (cacheless)
- **Can we do better?**

```
typedef struct qnode {
  struct qnode *prev;
  int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
  qn->succ_blocked = 1;
  qn->prev = FetchAndSet(lck, qn);
  while (qn->prev->succ_blocked);
}

void unlock(qnode **qn) {
  qnode *pred = (*qn)->prev;
  (*qn)->succ_blocked = 0;
  *qn = pred;
}
```

MCS Lock (1991)

- **Make queue explicit**
  - Acquire lock by appending to queue
  - Spin on own node until locked is reset
- **Similar advantages as CLH but**
  - Only 2N + M words
  - Spinning position is fixed!

    Benefits cache-less NUMA
- **What are the issues?**
  - Releasing lock spins
  - More atomics!

```
typedef struct qnode {
  struct qnode *next;
  int succ_blocked;
} qnode;

qnode *lck = NULL;

void lock(qnode *lck, qnode *qn) {
  qn->next = NULL;
  qnode *pred = FetchAndSet(lck, qn);
  if(pred != NULL) {
    qn->locked = 1;
    pred->next = qn;
    while(qn->locked);
} }

void unlock(qnode * lck, qnode *qn) {
  if(qn->next == NULL) { // if we’re the last waiter
    if(CAS(lck, qn, NULL)) return;
    while(qn->next == NULL); // wait for pred arrival
  }
  qn->next->locked = 0; // free next waiter
  qn->next = NULL;
}
```

Lessons Learned!

- **Key Lesson:**
  - Reducing memory (coherency) traffic is most important!
  - Not always straight-forward (need to reason about CL states)

- **MCS: 2006 Dijkstra Prize in distributed computing**
  - *“an outstanding paper on the principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade”*
  - *“probably the most influential practical mutual exclusion algorithm ever”*
  - *“vastly superior to all previous mutual exclusion algorithms”*
  - fast, fair, scalable → widely used, always compared against!

Time to Declare Victory?

- **Down to memory complexity of 2N+M**
  - Probably close to optimal
- **Only local spinning**
  - Several variants with low expected contention
- **But: we assumed sequential consistency ☹**
  - Reality causes trouble sometimes
  - Sprinkling memory fences may harm performance
  - Open research on minimally-synching algorithms!

    Come and talk to me if you're interested

More Practical Optimizations

- **Let’s step back to “data race”**
  - (recap) two operations A and B on the same memory cause a data race if one of them is a write (“conflicting access”) and neither A→B nor B→A
  - So we put conflicting accesses into a CR and lock it!

    This also guarantees memory consistency in C++/Java!

- **Let’s say you implement a web-based encyclopedia**
  - Consider the “average two accesses” – do they conflict?

Reader-Writer Locks

- **Allows multiple concurrent reads**
  - Multiple reader locks concurrently in CR
  - Guarantees mutual exclusion between writer and writer locks and reader and writer locks

- **Syntax:**
  - read_(un)lock()
  - write_(un)lock()

A Simple RW Lock

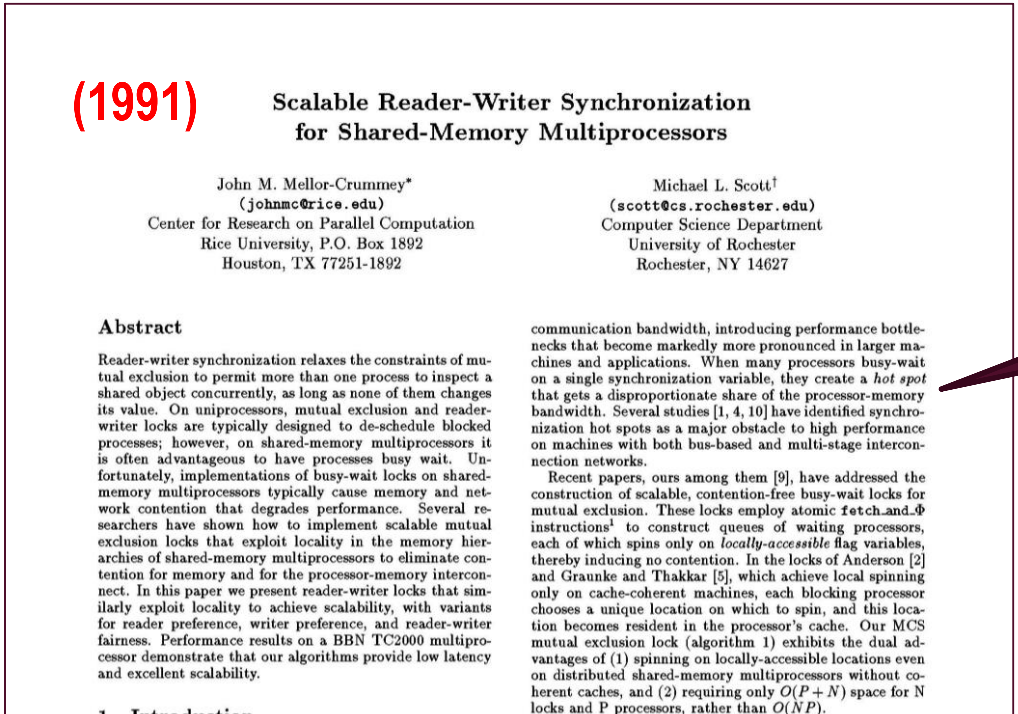
- **Seems efficient!?**
  - Is it? What’s wrong?
  - Polling CAS!

- **Is it fair?**
  - Readers are preferred!
  - Can always delay writers (again and again and again)

```
const W = 1;
const R = 2;
volatile int lock=0; // LSB is writer flag!

void read_lock(lock_t lock) {
  AtomicAdd(lock, R);
  while(lock & W);
}

void write_lock(lock_t lock) {
  while(!CAS(lock, 0, W));
}

void read_unlock(lock_t lock) {
  AtomicAdd(lock, -R);
}

void write_unlock(lock_t lock) {
  AtomicAdd(lock, -W);
}
```

Fixing those Issues?

- **Polling issue:**
  - Combine with MCS lock idea of queue polling
- **Fairness:**
  - Count readers and writers



(1991)

Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors

John M. Mellor-Crummey*
(johnmc@rice.edu)
Center for Research on Parallel Computation
Rice University, P.O. Box 1892
Houston, TX 77251-1892

Michael L. Scott†
(scott@cs.rochester.edu)
Computer Science Department
University of Rochester
Rochester, NY 14627

Abstract

Reader-writer synchronization relaxes the constraints of mutual exclusion to permit more than one process to inspect a shared object concurrently, as long as none of them changes its value. On uniprocessors, mutual exclusion and reader-writer locks are typically designed to de-schedule blocked processes; however, on shared-memory multiprocessors it is often advantageous to have processes busy wait. Unfortunately, implementations of busy-wait locks on shared-memory multiprocessors typically cause memory and network contention that degrades performance. Several researchers have shown how to implement scalable mutual exclusion locks that exploit locality in the memory hierarchies of shared-memory multiprocessors to eliminate contention for memory and for the processor-memory interconnect. In this paper we present reader-writer locks that similarly exploit locality to achieve scalability, with variants for reader preference, writer preference, and reader-writer fairness. Performance results on a BBN TC2000 multiprocessor demonstrate that our algorithms provide low latency and excellent scalability.

1 Introduction

communication bandwidth, introducing performance bottlenecks that become markedly more pronounced in larger machines and applications. When many processors busy-wait on a single synchronization variable, they create a *hot spot* that gets a disproportionate share of the processor-memory bandwidth. Several studies [1, 4, 10] have identified synchronization hot spots as a major obstacle to high performance on machines with both bus-based and multi-stage interconnection networks.

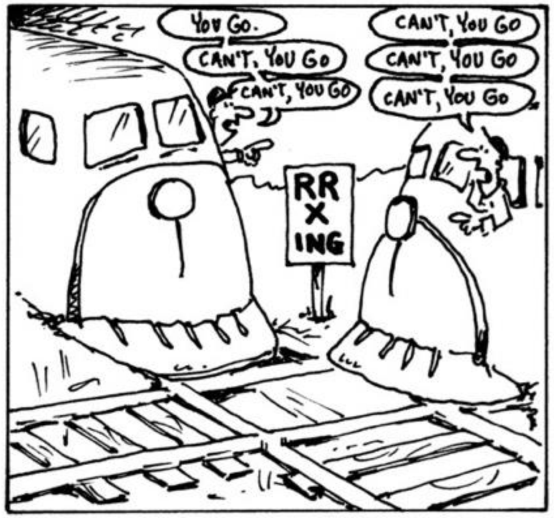
Recent papers, ours among them [9], have addressed the construction of scalable, contention-free busy-wait locks for mutual exclusion. These locks employ atomic fetch_and_Φ instructions[1] to construct queues of waiting processors, each of which spins only on *locally-accessible* flag variables, thereby inducing no contention. In the locks of Anderson [2] and Graunke and Thakkar [5], which achieve local spinning only on cache-coherent machines, each blocking processor chooses a unique location on which to spin, and this location becomes resident in the processor's cache. Our MCS mutual exclusion lock (algorithm 1) exhibits the dual advantages of (1) spinning on locally-accessible locations even on distributed shared-memory multiprocessors without coherent caches, and (2) requiring only $O(P+N)$ space for N locks and P processors, rather than $O(NP)$.

The final algorithm (Alg. 4) has a flaw that was corrected in 2003!

Deadlocks

- **Kansas state legislature: *"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."***

[according to Botkin, Harlow "A Treasury of Railroad Folklore" (pp. 381)]



What are necessary conditions for deadlock?

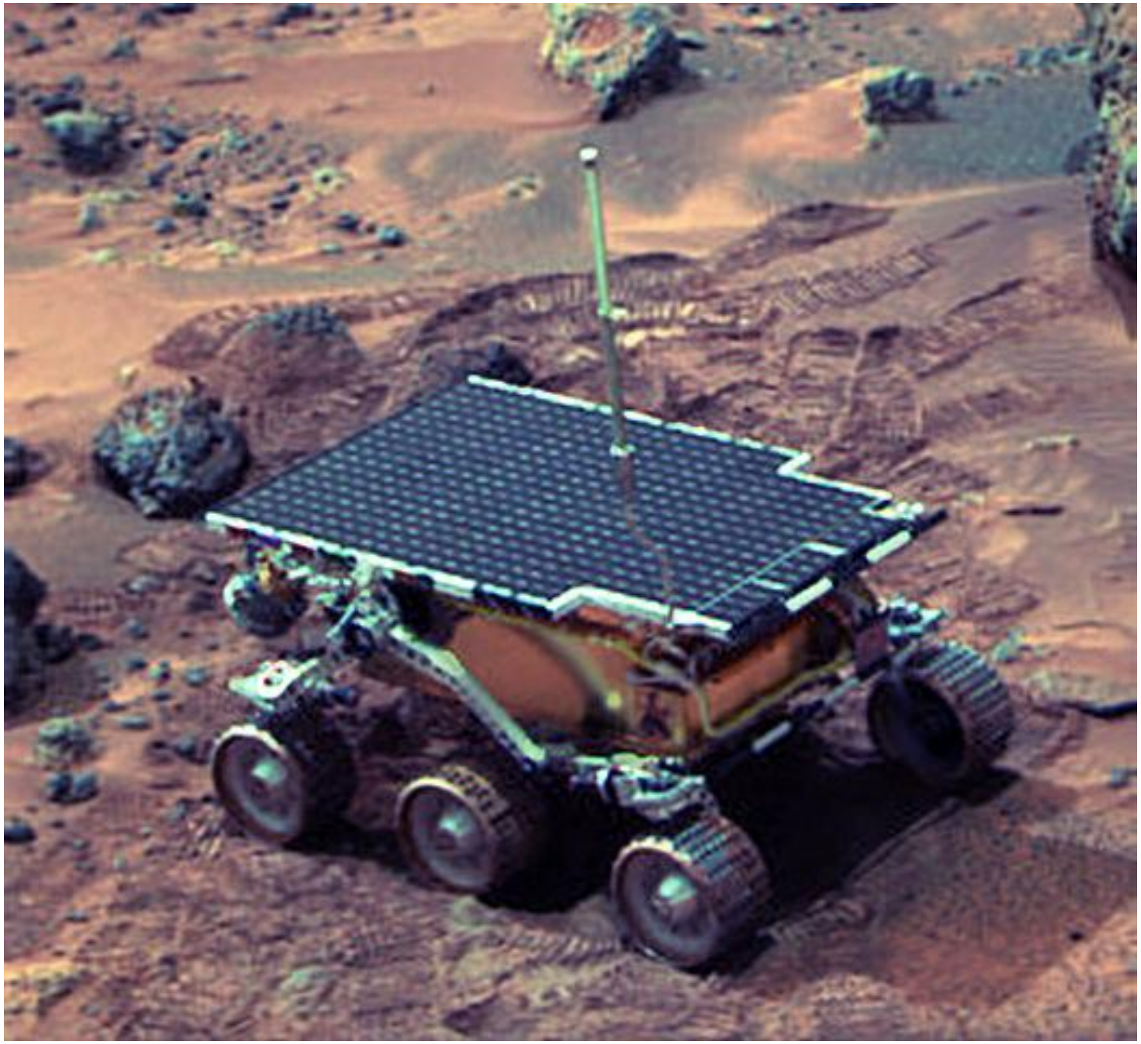
Deadlocks

- **Necessary conditions:**
  - Mutual Exclusion
  - Hold one resource, request another
  - No preemption
  - Circular wait in dependency graph
- **One condition missing will prevent deadlocks!**
  - →Different avoidance strategies (which?)

Issues with Spinlocks

- **Spin-locking is very wasteful**
  - The spinning thread occupies resources
  - Potentially the PE where the waiting thread wants to run → requires context switch!
- **Context switches due to**
  - Expiration of time-slices (forced)
  - Yielding the CPU

What is this?



Why is the 1997 Mars Rover in our lecture?

- **It landed, received program, and worked … until it spuriously rebooted!**
  - → watchdog
- **Scenario (vxWorks RT OS):**
  - Single CPU
  - Two threads A,B sharing common bus, using locks
  - (independent) thread C wrote data to flash
  - Priority: A→C→B (A highest, B lowest)
  - Thread C would run into a lifelock (infinite loop)
  - Thread B was preempted by C while holding lock
  - Thread A got stuck at lock ☹

[http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html]

Priority Inversion

- **If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!**
- **Can be fixed with the help of the OS**
  - E.g., mutex priority inheritance (temporarily boost priority of task in CR to highest priority among waiting tasks)

Condition Variables

- **Allow threads to yield CPU and leave the OS run queue**
  - Other threads can get them back on the queue!
- **cond_wait(cond, lock) – yield and go to sleep**
- **cond_signal(cond) – wake up sleeping threads**
- **Wait and signal are OS calls**
  - Often expensive, which one is more expensive?

    Wait, because it has to perform a full context switch

Condition Variable Semantics

- **Hoare-style:**
  - Signaler passes lock to waiter, signaler suspended
  - Waiter runs immediately
  - Waiter passes lock back to signaler if it leaves critical section or if it waits again
- **Mesa-style (most used):**
  - Signaler keeps lock
  - Waiter simply put on run queue
  - Needs to acquire lock, may wait again

When to Spin and When to Block?

- **Spinning consumes CPU cycles but is cheap**
  - “Steals” CPU from other threads
- **Blocking has high one-time cost and is then free**
  - Often hundreds of cycles (trap, save TCB …)
  - Wakeup is also expensive (latency)

    Also cache-pollution
- **Strategy:**
  - Poll for a while and then block

When to Spin and When to Block?

- **What is a “while”?**
- **Optimal time depends on the future**
  - When will the active thread leave the CR?
  - Can compute optimal offline schedule
  - Actual problem is an online problem
- **Competitive algorithms**
  - An algorithm is c-competitive if for a sequence of actions x and a constant a holds:

    $C(x) \leq c*C_{opt}(x) + a$
  - What would a good spinning algorithm look like and what is the competitiveness?

Competitive Spinning

- **If T is the overhead to process a wait, then a locking algorithm that spins for time T before it blocks is 2-competitive!**
  - Karlin, Manasse, McGeoch, Owicki: “Competitive Randomized Algorithms for Non-Uniform Problems”, SODA 1989
- **If randomized algorithms are used, then e/(e-1)-competitiveness (~1.58) can be achieved**
  - See paper above!

Generalized Locks: Semaphores

- **Controlling access to more than one resource**
  - Described by Dijkstra 1965
- **Internal state is an atomic counter C**
- **Two operations:**
  - P() – block until C>0; decrement C (atomically)
  - V() – signal and increment C
- **Binary or 0/1 semaphore equivalent to lock**
  - C is always 0 or 1, i.e., V() will not increase it further
- **Trivia:**
  - If you're lucky (aehem, speak Dutch), mnemonics:

    V*erhogen (increment) and* ***P****rolaag (probeer te verlagen = try to reduce)*

Semaphore Implementation

- **Can be implemented with mutual exclusion!**
  - And can be used to implement mutual exclusion ☺
- **… or with test and set and many others!**
- **Also has fairness concepts:**
  - Order of granting access to waiting (queued) threads
  - strictly fair (starvation impossible, e.g., FIFO)
  - weakly fair (starvation possible, e.g., random)

Case Study 1: Barrier

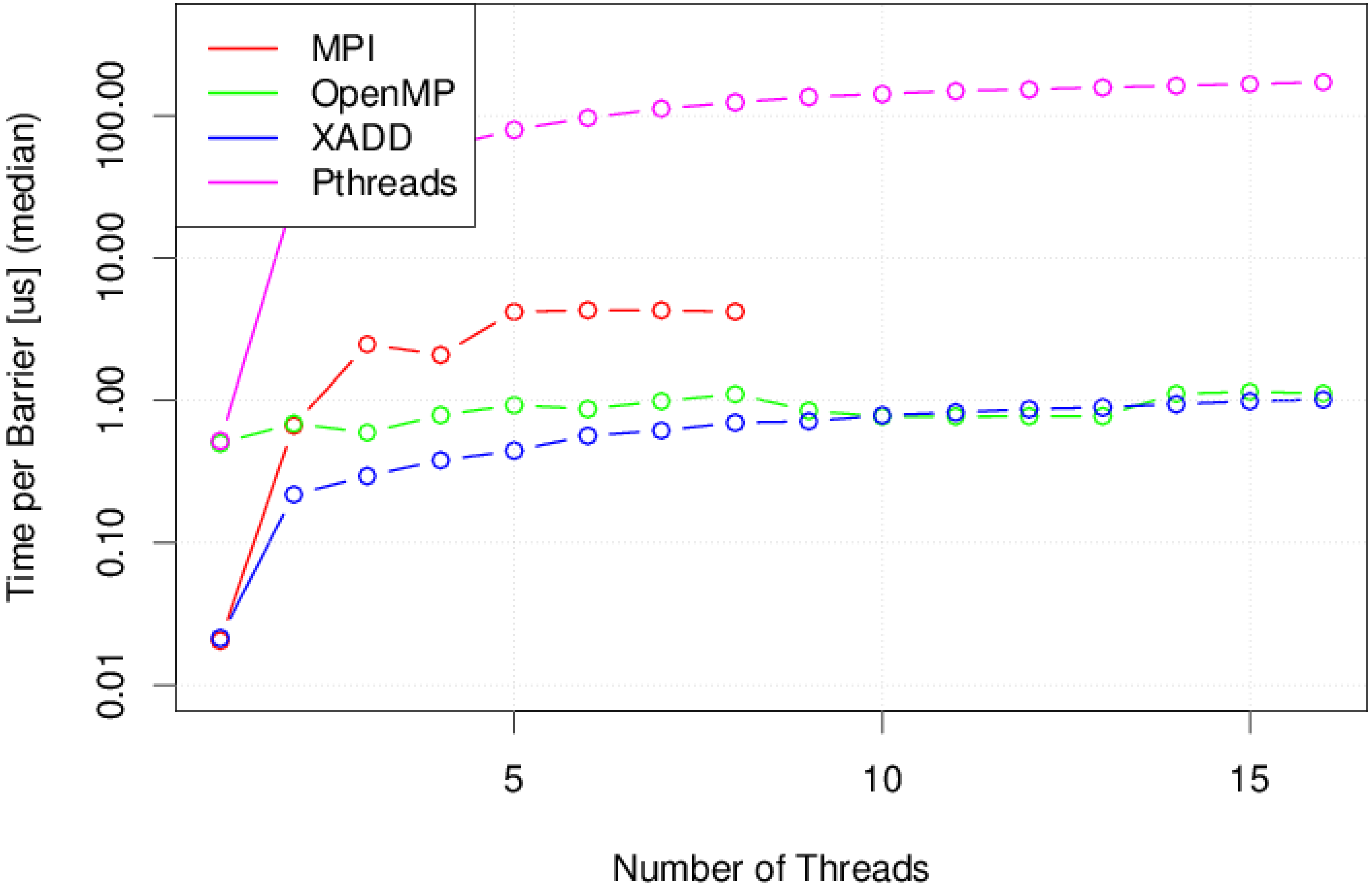
- **Barrier semantics:**
  - No process proceeds before all processes reached barrier
  - Similar to mutual exclusion but not exclusive, rather “synchronized”
- **Often needed in parallel high-performance programming**
  - Especially in SPMD programming style
- **Parallel programming “frameworks” offer barrier semantics (pthread, OpenMP, MPI)**
  - MPI_Barrier() (process-based)
  - pthread_barrier
  - #pragma omp barrier
  - …
- **Simple implementation: lock xadd + spin**

  Problem: when to re-use the counter?

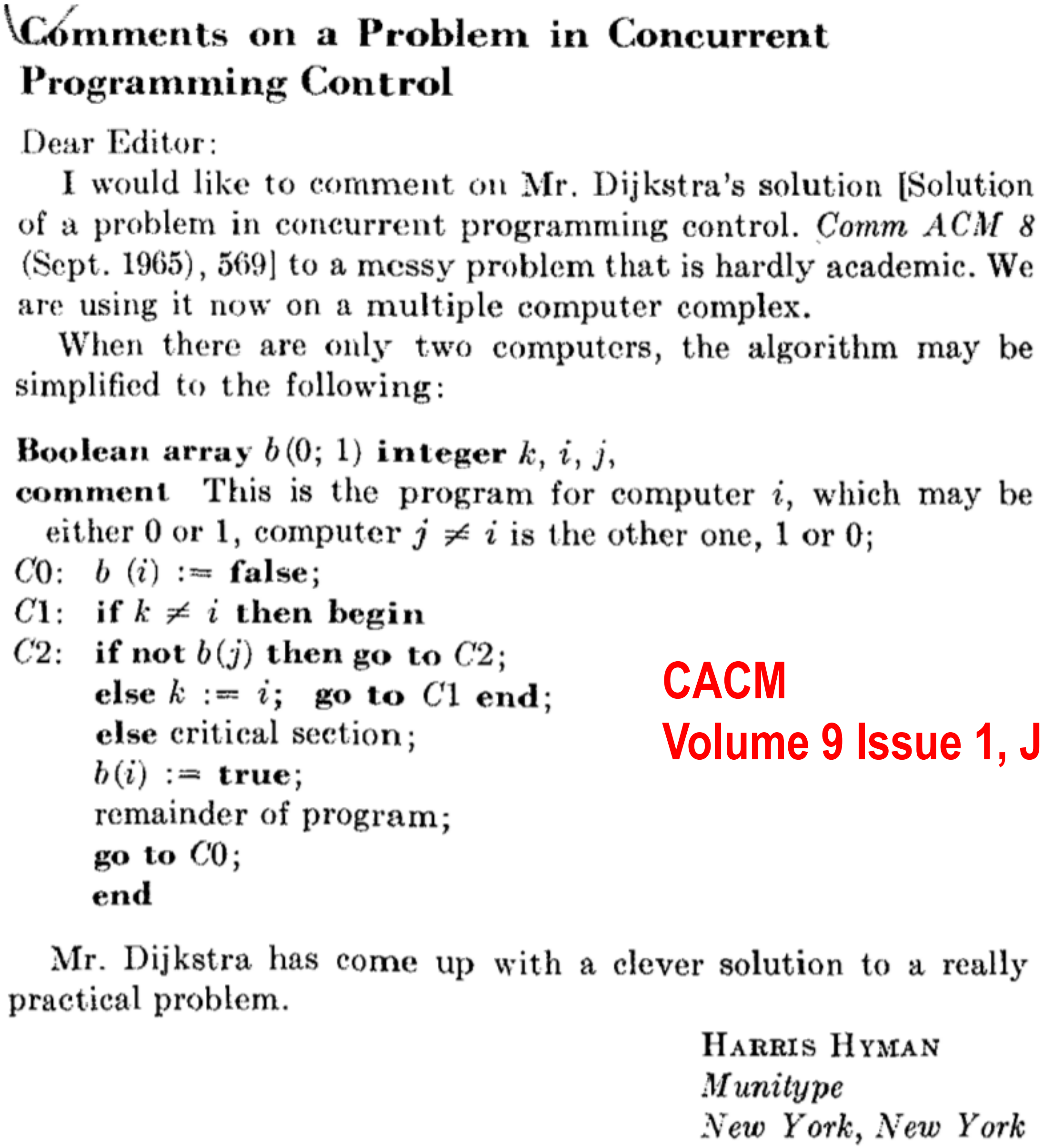
  Cannot just set it to 0 ☹ → Trick: “lock xadd -1” when done ☺

[cf. http://www.spiral.net/software/barrier.html]

Barrier Performance



Case Study 2: Reasoning about Semantics



Comments on a Problem in Concurrent Programming Control

Dear Editor:

I would like to comment on Mr. Dijkstra's solution [Solution of a problem in concurrent programming control. *Comm ACM 8* (Sept. 1965), 569] to a messy problem that is hardly academic. We are using it now on a multiple computer complex.

When there are only two computers, the algorithm may be simplified to the following:

```
Boolean array b(0; 1) integer k, i, j,
comment  This is the program for computer i, which may be
   either 0 or 1, computer j ≠ i is the other one, 1 or 0;
C0:  b (i) := false;
C1:  if k ≠ i then begin
C2:  if not b(j) then go to C2;
     else k := i;  go to C1 end;
     else critical section;
     b(i) := true;
     remainder of program;
     go to C0;
     end
```

Mr. Dijkstra has come up with a clever solution to a really practical problem.

Harris Hyman
Munitype
New York, New York

CACM
Volume 9 Issue 1, Jan. 1966

Case Study 2: Reasoning about Semantics

- **Is the proposed algorithm correct?**
  - We may proof it manually

    Using tools from the last lecture

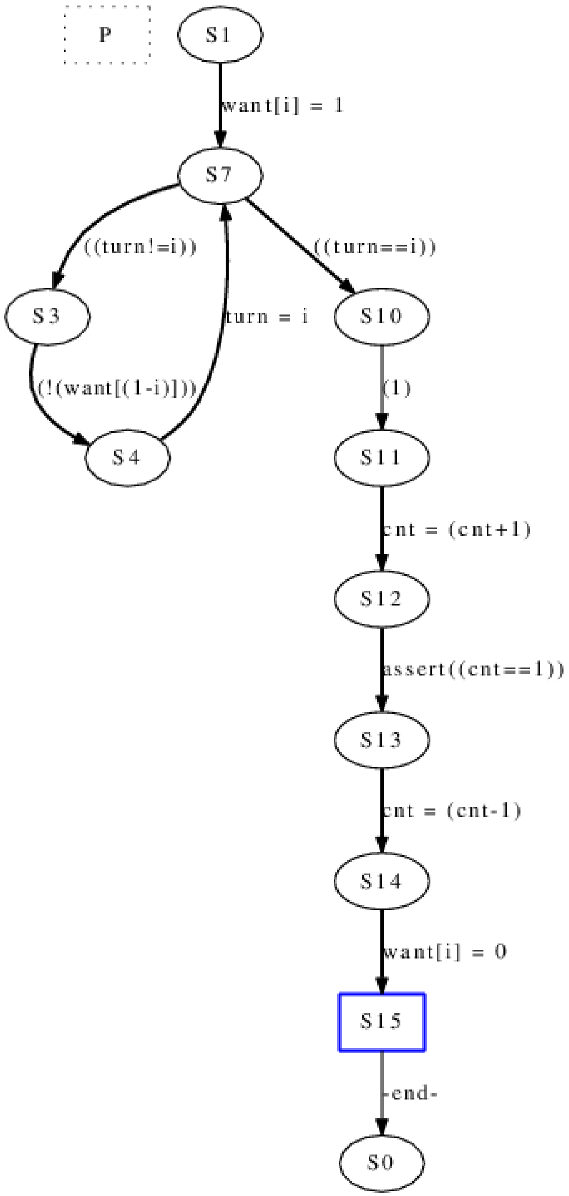
    → reason about the state space of H
  - Or use automated proofs (model checking)

    E.g., SPIN (Promela syntax)

```
bool want[2];
bool turn;
byte cnt;

proctype P(bool i)
{
  want[i] = 1;
  do
  :: (turn != i) ->
      (!want[1-i]);
      turn = i
  :: (turn == i) ->
      break
  od;
  skip; /* critical section */
  cnt = cnt+1;
  assert(cnt == 1);
  cnt = cnt-1;
  want[i] = 0
}

init { run P(0); run P(1) }
```

Case Study 2: Reasoning about Semantics

- **Spin tells us quickly that it found a problem**
  - A sequentially consistent order that violates mutual exclusion!
- **It's not always that easy**
  - This example comes from the SPIN tutorial
  - More than two threads make it much more demanding!
- **More in the recitation!**



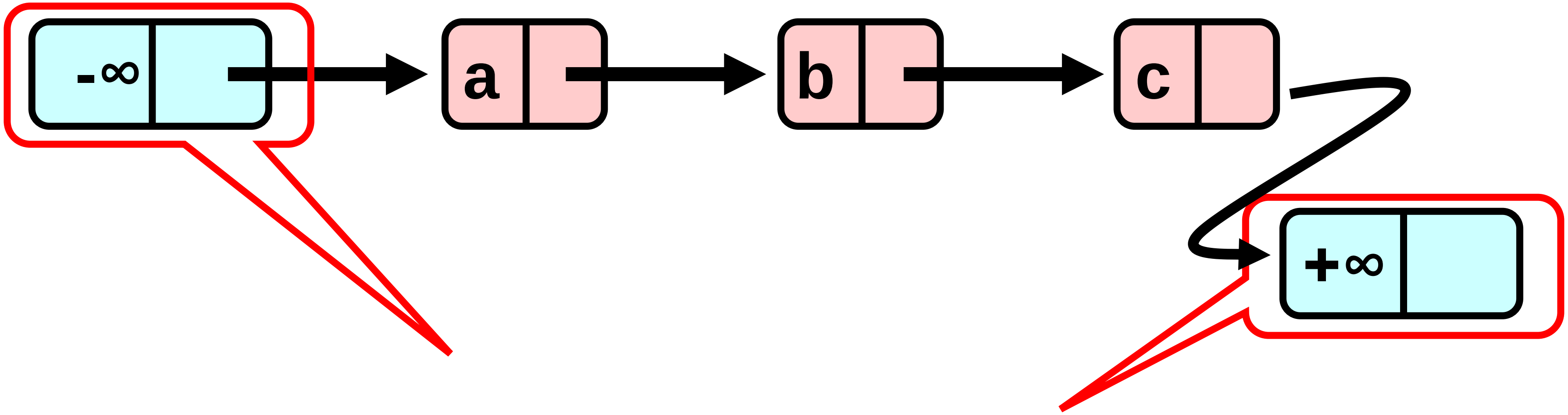
Locks in Practice

- **Running example: List-based set of integers**
  - S.insert(v) – return true if v was inserted
  - S.remove(v) – return true if v was removed
  - S.contains(v) – return true iff v in S
- **Simple ordered linked list**
  - Do not use this at home (poor performance)
  - Good to demonstrate locking techniques

    E.g., skip lists would be faster but more complex

Set Structure in Memory

- This and many of the following illustrations are provided by Maurice Herlihy in conjunction with the book “The Art of Multiprocessor Programming”



Sorted with Sentinel nodes
(min & max possible keys)

Sequential Set

```
boolean add(S, x) {
  node *pred = S.head;
  node *curr = pred.next;
  while(curr.key < x) {
    pred = curr;
    curr = pred.next;
  }
  if(curr.key == x)
    return false;
  else {
    node n = new node();
    n.key = x;
    n.next = curr;
    pred.next = n;
  }
  return true;
}
```

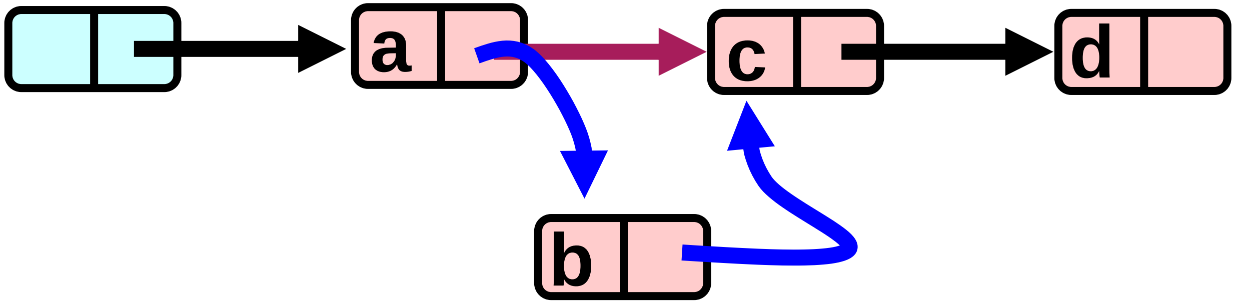
```
boolean remove(S, x) {
  node *pred = S.head;
  node *curr = pred.next;
  while(curr.key < x) {
    pred = curr;
    curr = pred.next;
  }
  if(curr.key == x) {
    pred.next = curr.next;
    free(curr);
    return true;
  }
  return false;
}
```

```
boolean contains(S, x) {
  int *curr = S.head;
  while(curr.key < x)
    curr = curr.next;
  if(curr.key == x)
    return true;
  return false;
}
```

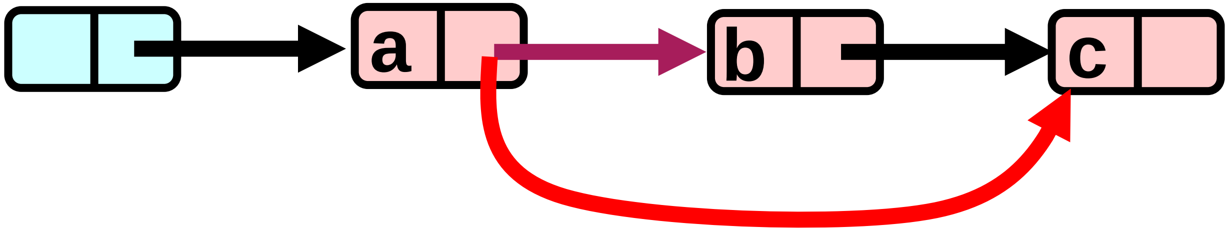
```
typedef struct {
  int key;
  node *next;
} node;
```

Sequential Operations

add ( )



remove ( )



Concurrent Sets

- **What can happen if multiple threads call set operations at the “same time”?**
  - Operations can conflict!
- **Which operations conflict?**
  - (add, remove), (add, add), (remove, remove), (remove, contains) will conflict
  - (add, contains) may miss update (which is fine)
  - (contains, contains) does not conflict
- **How can we fix it?**

Coarse-grained Locking

```
boolean add(S, x) {
 lock(S);
 node *pred = S.head;
 node *curr = pred.next;
 while(curr.key < x) {
   pred = curr;
   curr = pred.next;
 }
 if(curr.key == x)
   unlock(S);
   return false;
 else {
   node node = malloc();
   node.key = x;
   node.next = curr;
   pred.next = node;
 }
 unlock(S);
 return true;
}
```

```
boolean remove(S, x) {
 lock(S);
 node *pred = S.head;
 node *curr = pred.next;
 while(curr.key < x) {
   pred = curr;
   curr = pred.next;
 }
 if(curr.key == x) {
   pred.next = curr.next;
   unlock(S);
   free(curr);
   return true;
 }
 unlock(S);
 return false;
}
```

```
boolean contains(S, x) {
 lock(S);
 int *curr = S.head;
 while(curr.key < x)
   curr = curr.next;
 if(curr.key == x)  {
   unlock(S);
   return true;
 }
 unlock(S);
 return false;
}
```

Coarse-grained Locking

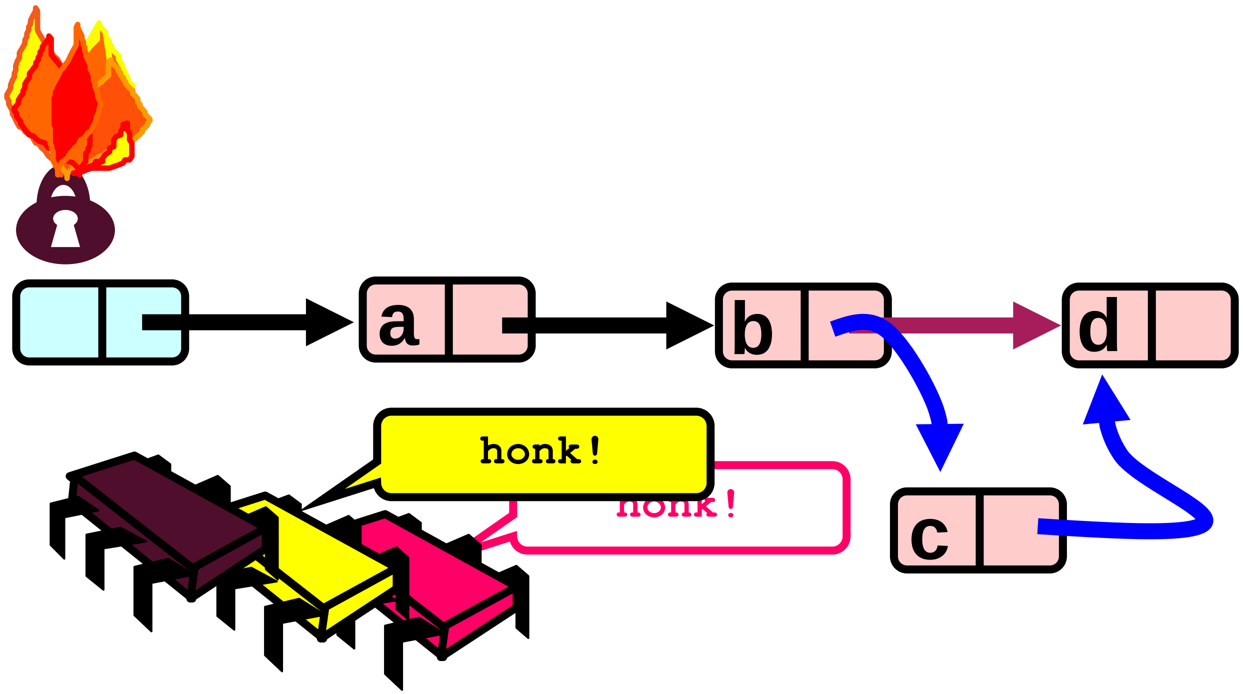
- **Correctness proof?**
  - Assume sequential version is correct

    Alternative: define set of invariants and proof that initial condition as well as all transformations adhere (pre- and post conditions)
  - Proof that all accesses to shared data are in CRs

    This may prevent some optimizations
- **Is the algorithm deadlock-free? Why?**
  - Locks are acquired in the same order (only one lock)
- **Is the algorithm starvation-free and/or fair? Why?**
  - It depends on the properties of the used locks!

Coarse-grained Locking

- Is the algorithm performing well with many concurrent threads accessing it?



Simple but hotspot + bottleneck

Coarse-grained Locking

- **Is the algorithm performing well with many concurrent threads accessing it?**
  - No, access to the whole list is serialized
- **BUT: it's easy to implement and proof correct**
  - Those benefits should **never** be underestimated
  - May be just good enough
  - *"We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified"* — Donald Knuth (in *Structured Programming with Goto Statements*)

How to Improve?

- **Will present some “tricks”**
  - Apply to the list example
  - But often generalize to other algorithms
  - Remember the trick, not the example!
- **See them as “concurrent programming patterns” (not literally)**
  - Good toolbox for development of concurrent programs
  - They become successively more complex

Tricks Overview

1. **Fine-grained locking**
   - Split object into “lockable components”
   - Guarantee mutual exclusion for conflicting accesses to same component
2. **Reader/writer locking**
3. **Optimistic synchronization**
4. **Lazy locking**
5. **Lock-free**

Tricks Overview

1. **Fine-grained locking**
2. **Reader/writer locking**
   - Multiple readers hold lock (traversal)
   - contains() only needs read lock
   - Locks may be upgraded during operation

   Must ensure starvation-freedom for writer locks!
3. **Optimistic synchronization**
4. **Lazy locking**
5. **Lock-free**

Tricks Overview

1. **Fine-grained locking**
2. **Reader/writer locking**
3. **Optimistic synchronization**
   - Traverse without locking

     Need to make sure that this is correct!
   - Acquire lock if update necessary

     May need re-start from beginning, tricky
4. **Lazy locking**
5. **Lock-free**

Tricks Overview

1. **Fine-grained locking**
2. **Reader/writer locking**
3. **Optimistic synchronization**
4. **Lazy locking**
   - Postpone hard work to idle periods
   - Mark node deleted

     Delete it physically later
5. **Lock-free**

Tricks Overview

1. **Fine-grained locking**
2. **Reader/writer locking**
3. **Optimistic synchronization**
4. **Lazy locking**
5. **Lock-free**
   - Completely avoid locks
   - Enables wait-freedom
   - Will need atomics (see later why!)
   - Often very complex, sometimes higher overhead

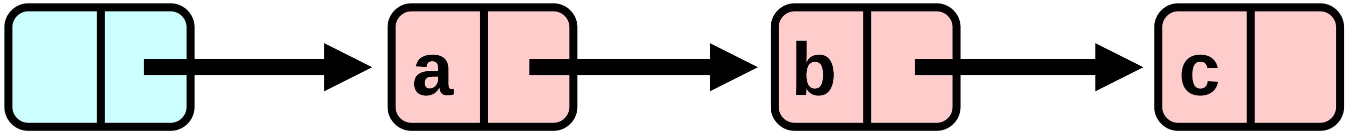
Trick 1: Fine-grained Locking

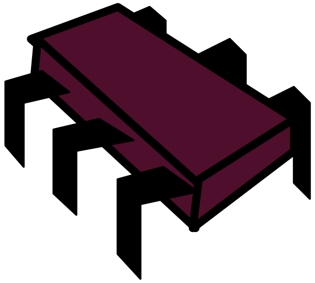
- **Each element can be locked**
  - High memory overhead
  - Threads can traverse list concurrently like a pipeline
- **Tricky to prove correctness**
  - And deadlock-freedom
  - Two-phase locking (acquire, release) often helps
- **Hand-over-hand (coupled locking)**
  - Not safe to release x’s lock before acquiring x.next’s lock

    will see why in a minute
  - Important to acquire locks in the same order

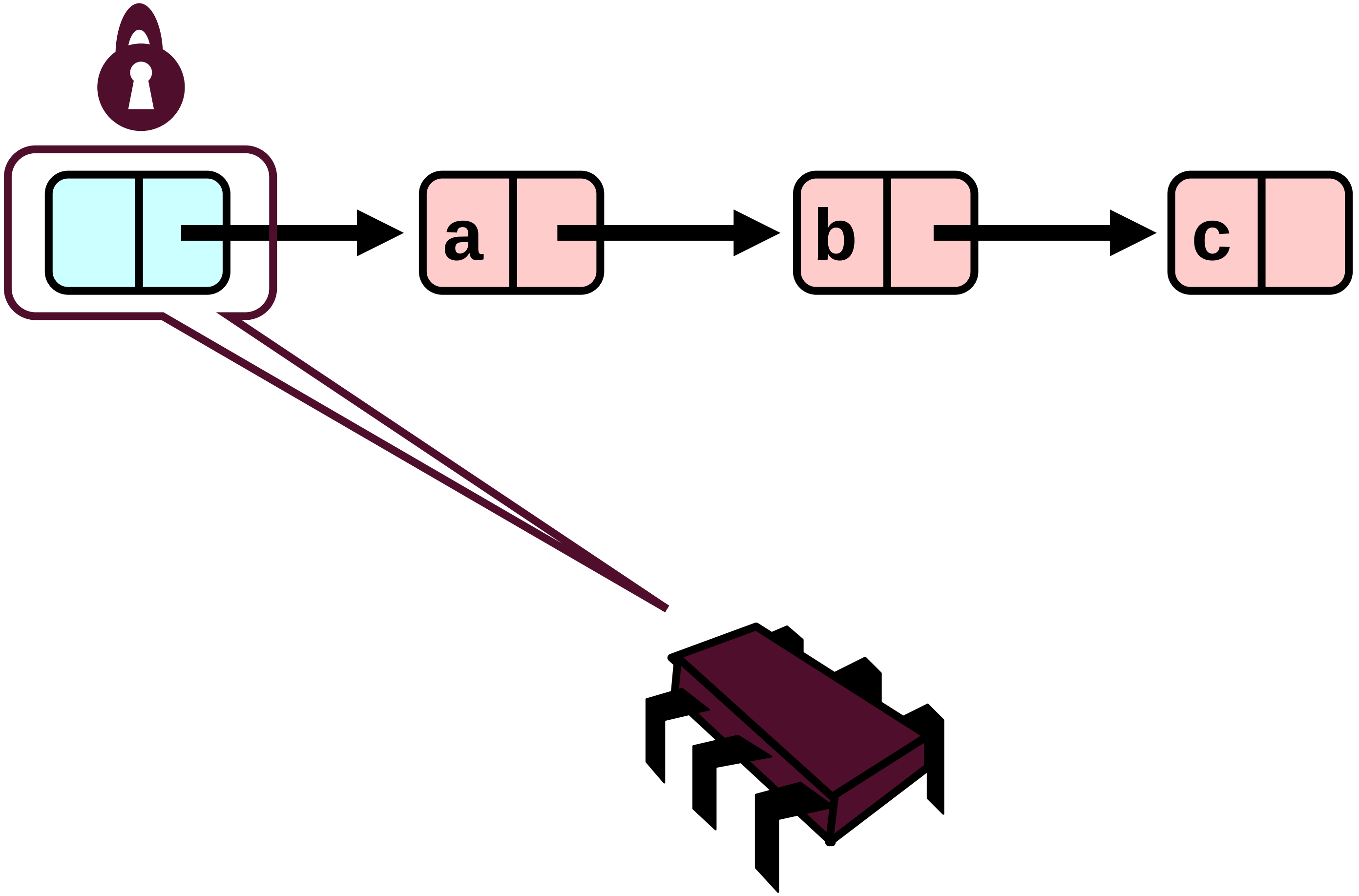
```
typedef struct {
  int key;
  node *next;
  lock_t lock;
} node;
```

Hand-over-Hand (fine-grained) locking

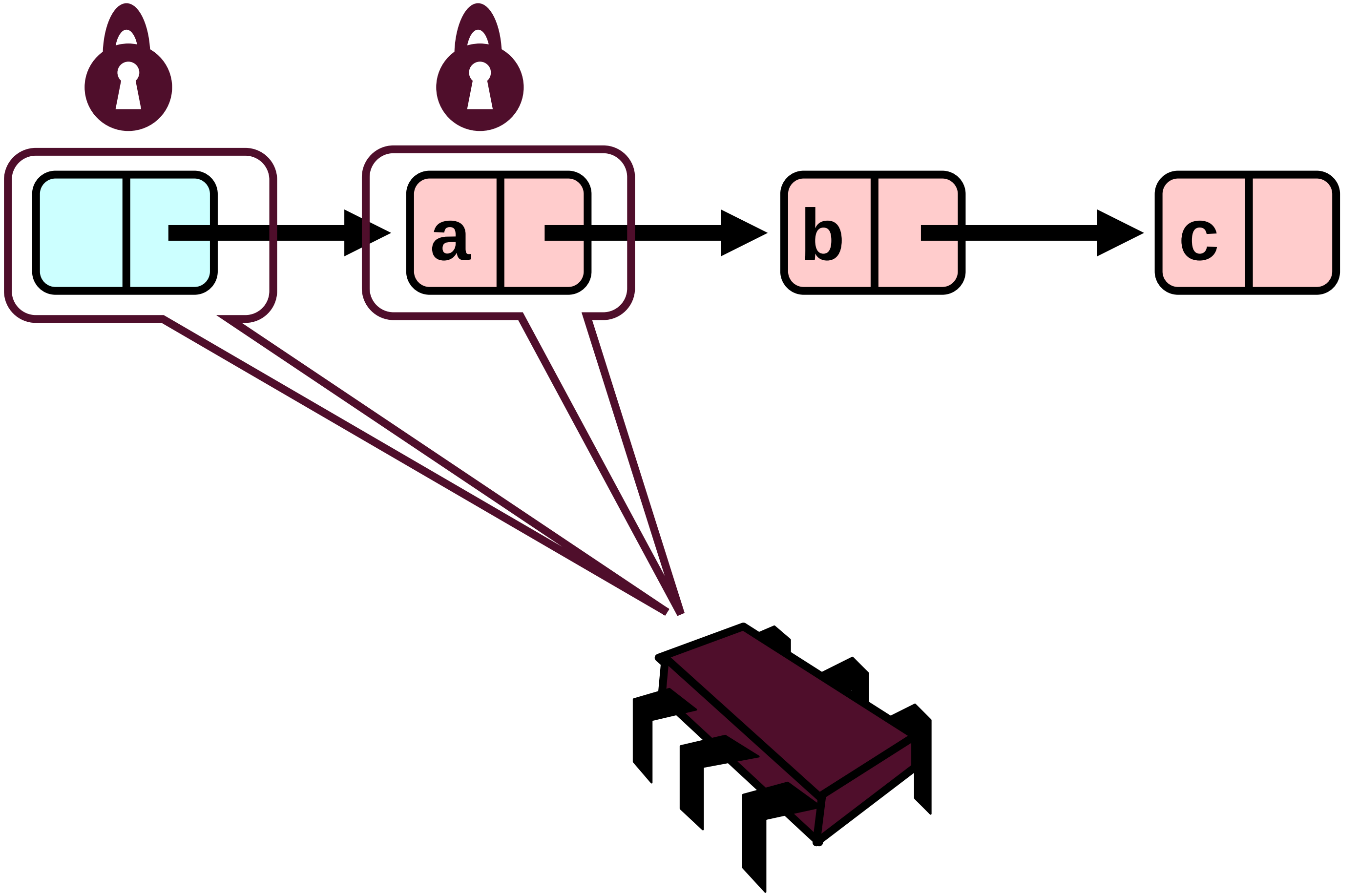




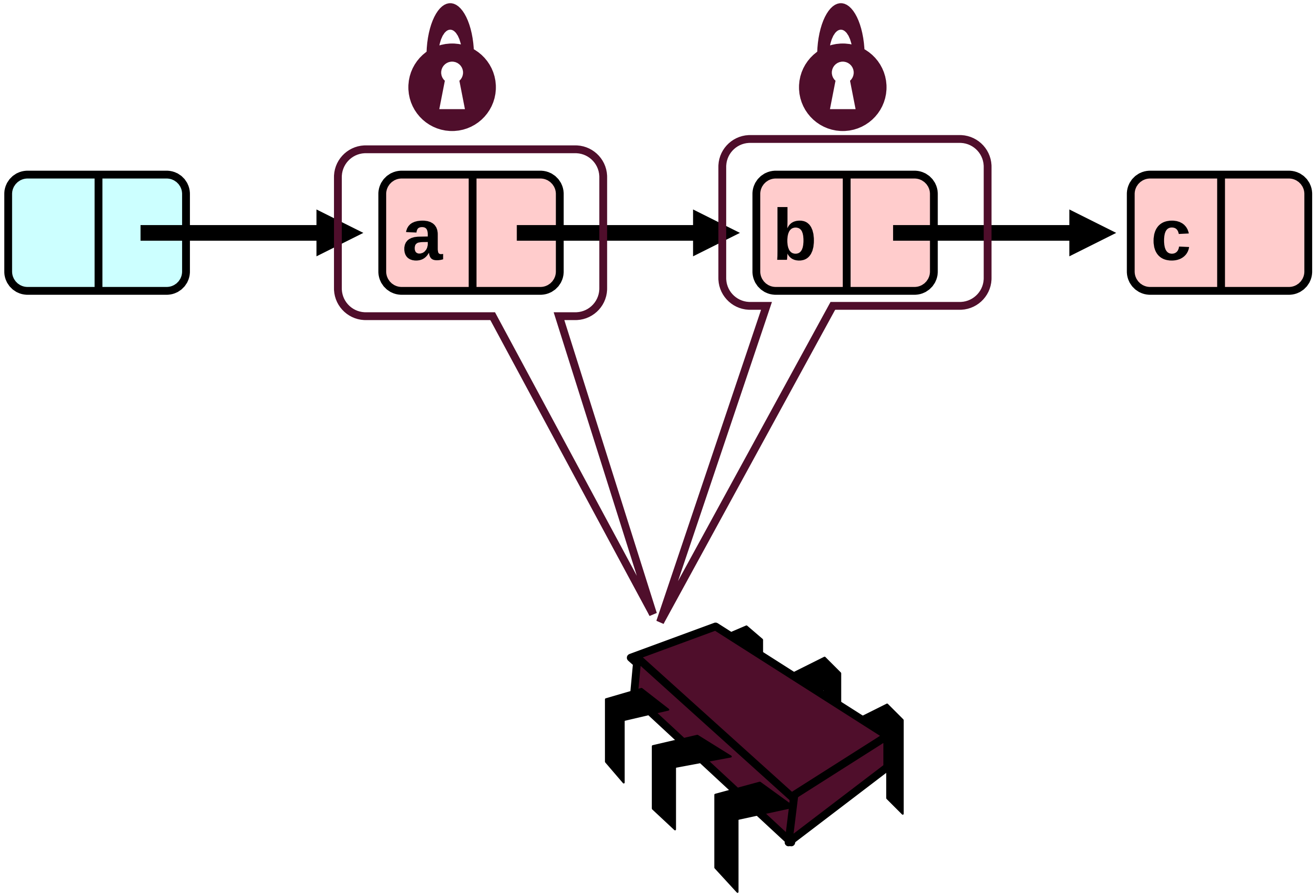
Hand-over-Hand (fine-grained) locking



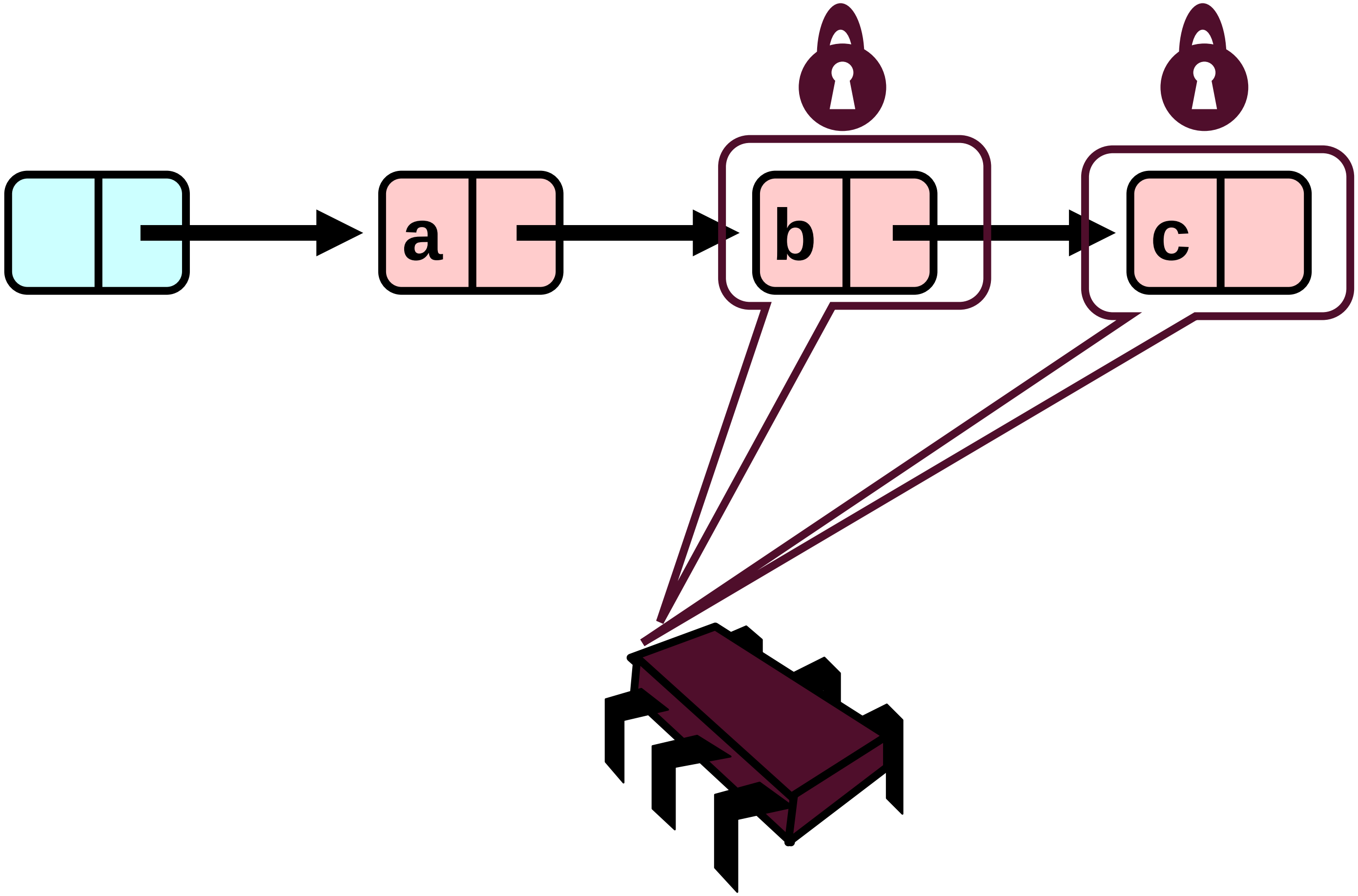
Hand-over-Hand (fine-grained) locking



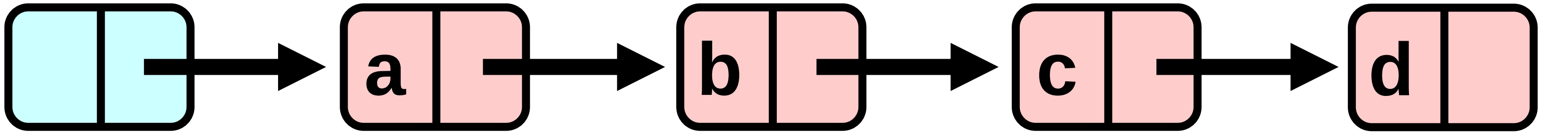
Hand-over-Hand (fine-grained) locking

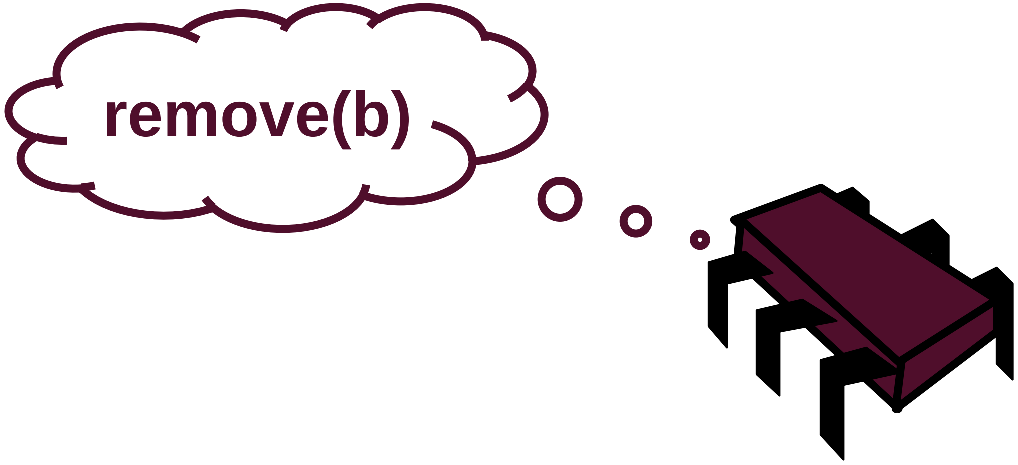


Hand-over-Hand (fine-grained) locking

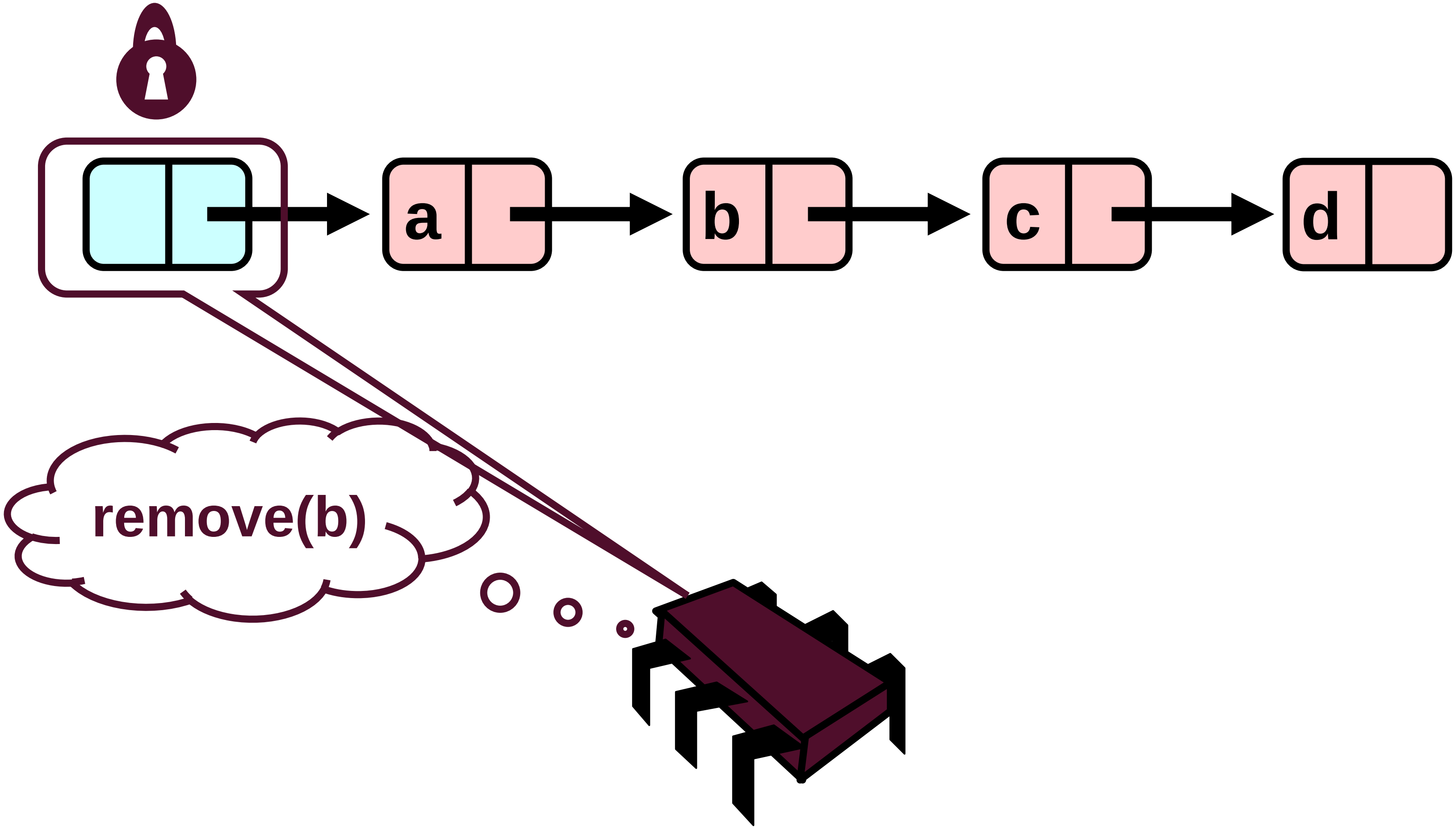


Removing a Node

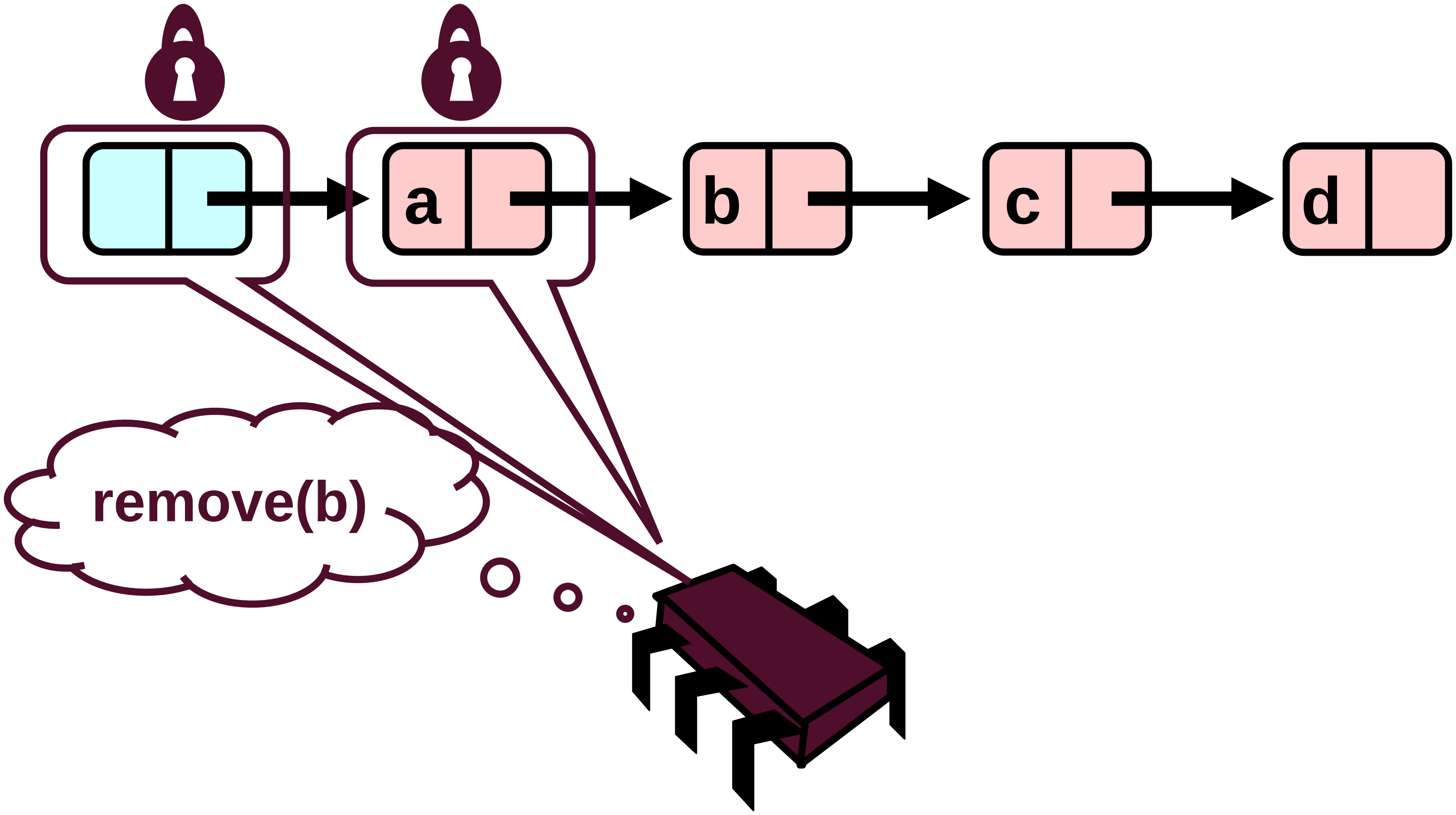




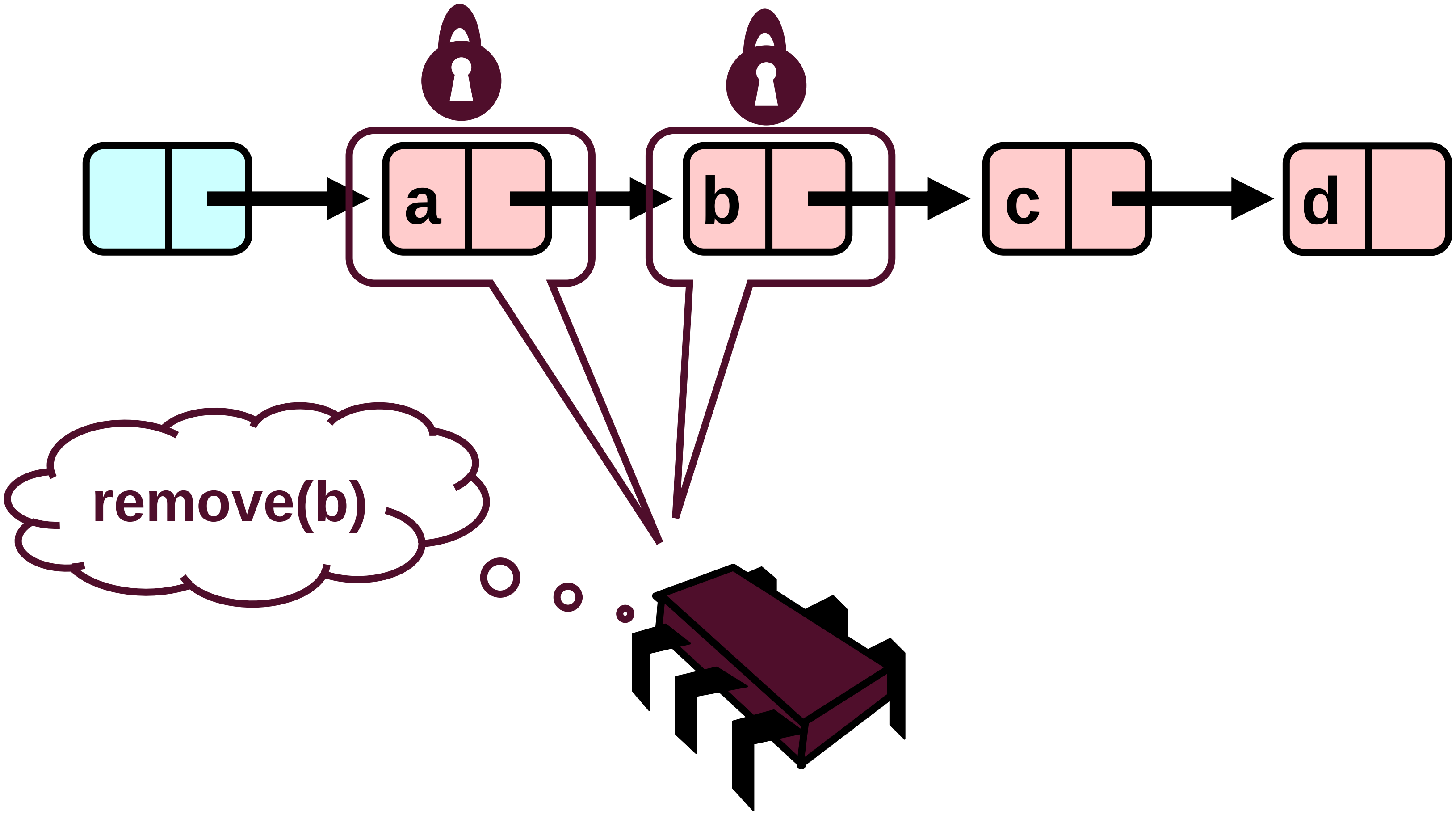
Removing a Node



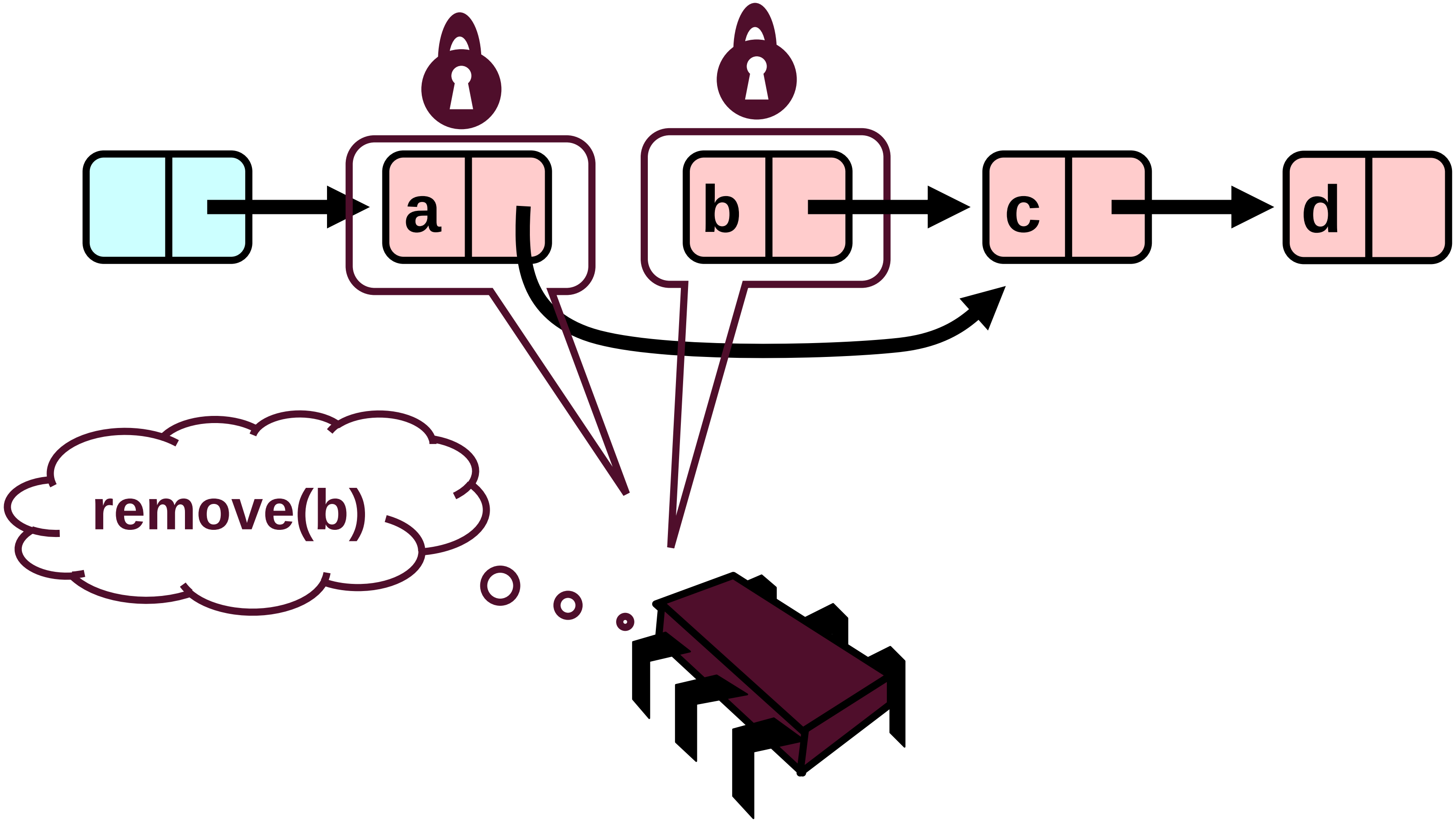
Removing a Node



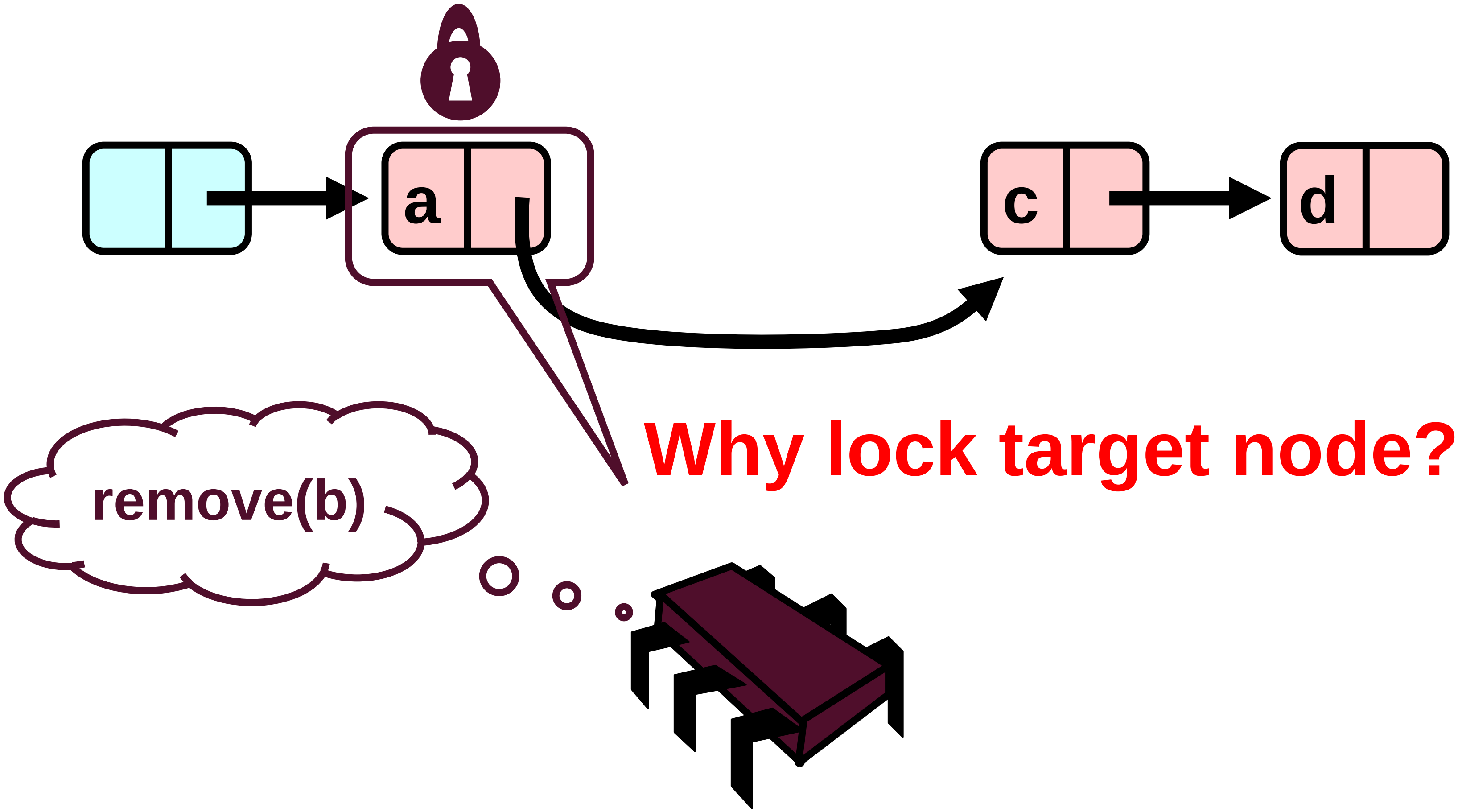
Removing a Node



Removing a Node

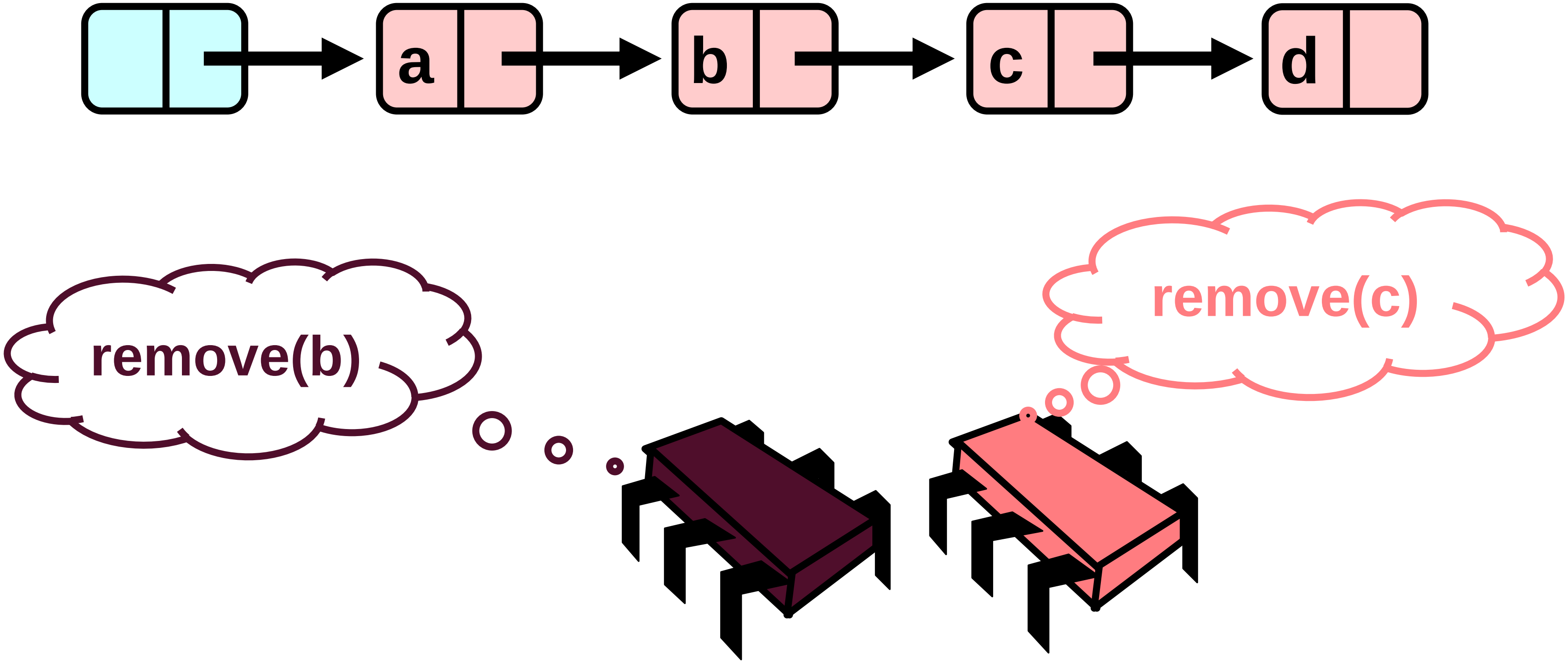


Removing a Node

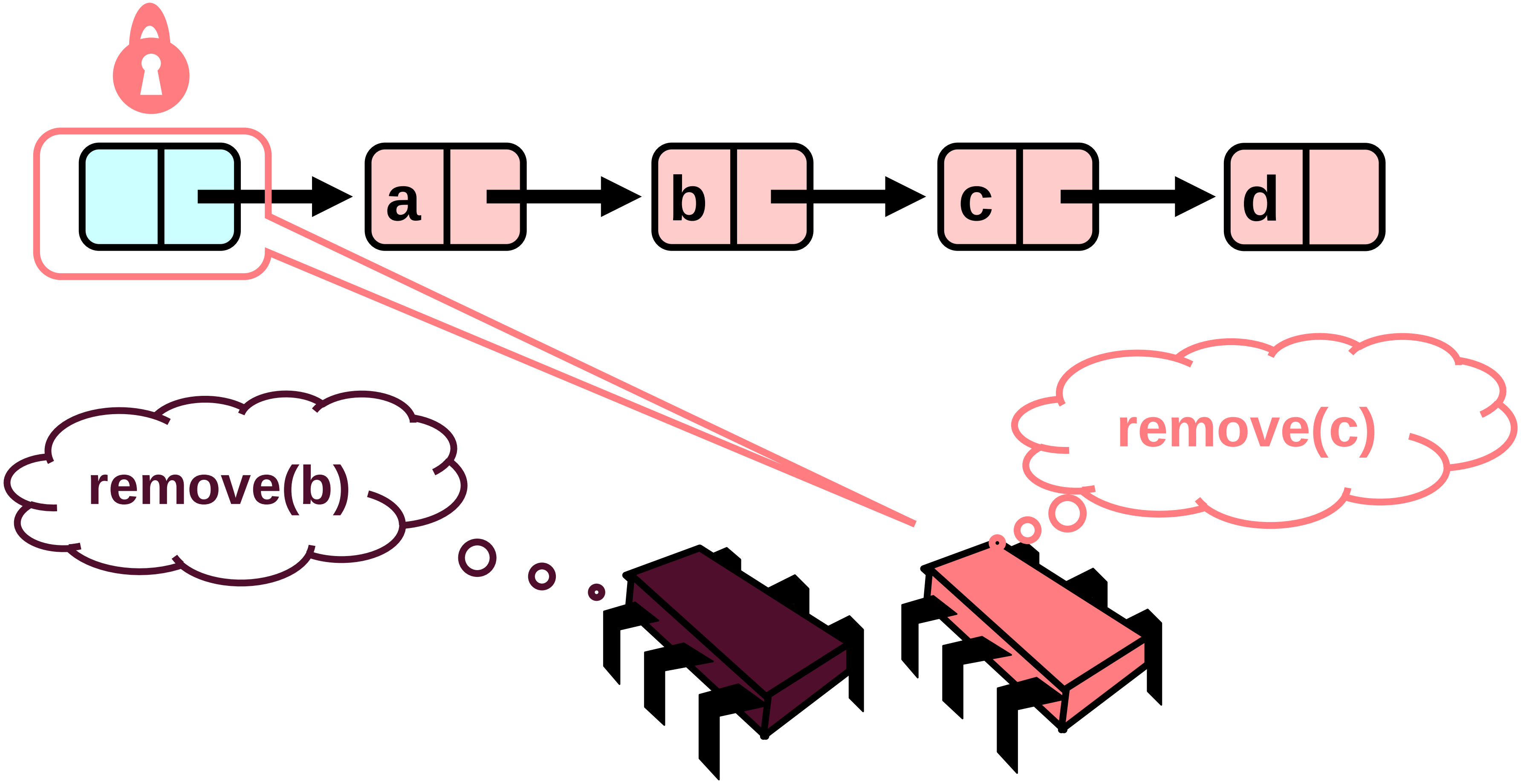


Why lock target node?

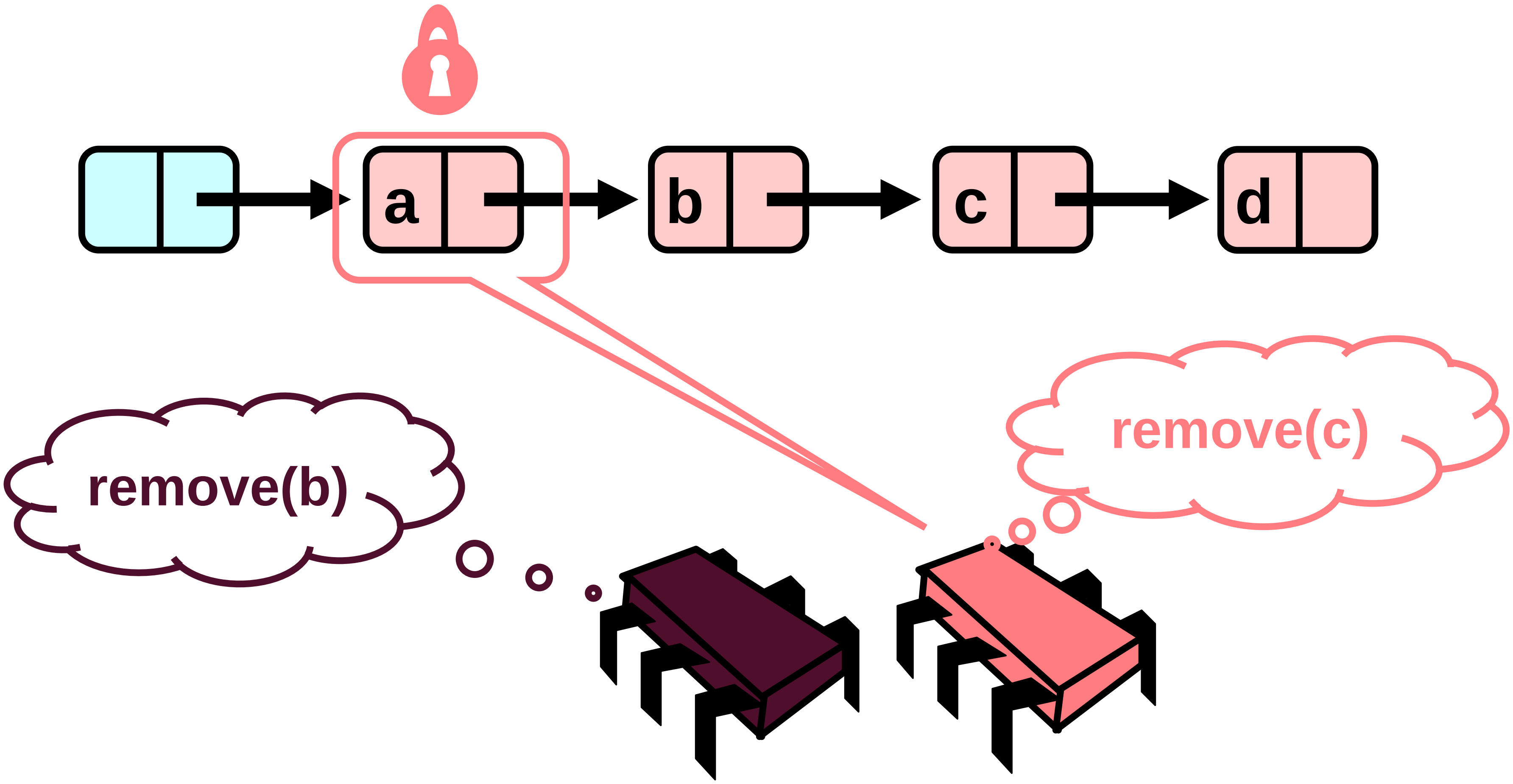
Concurrent Removes



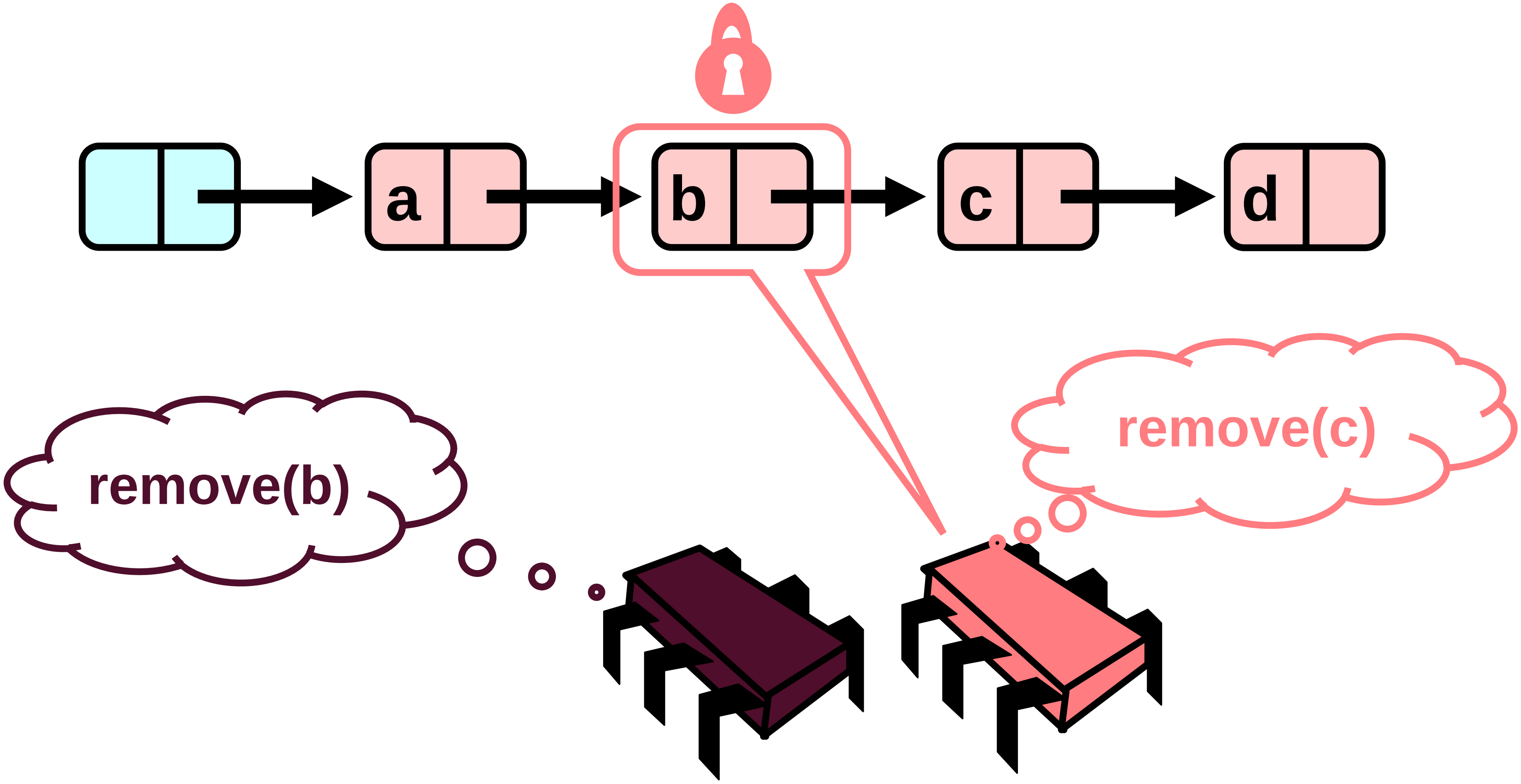
Concurrent Removes



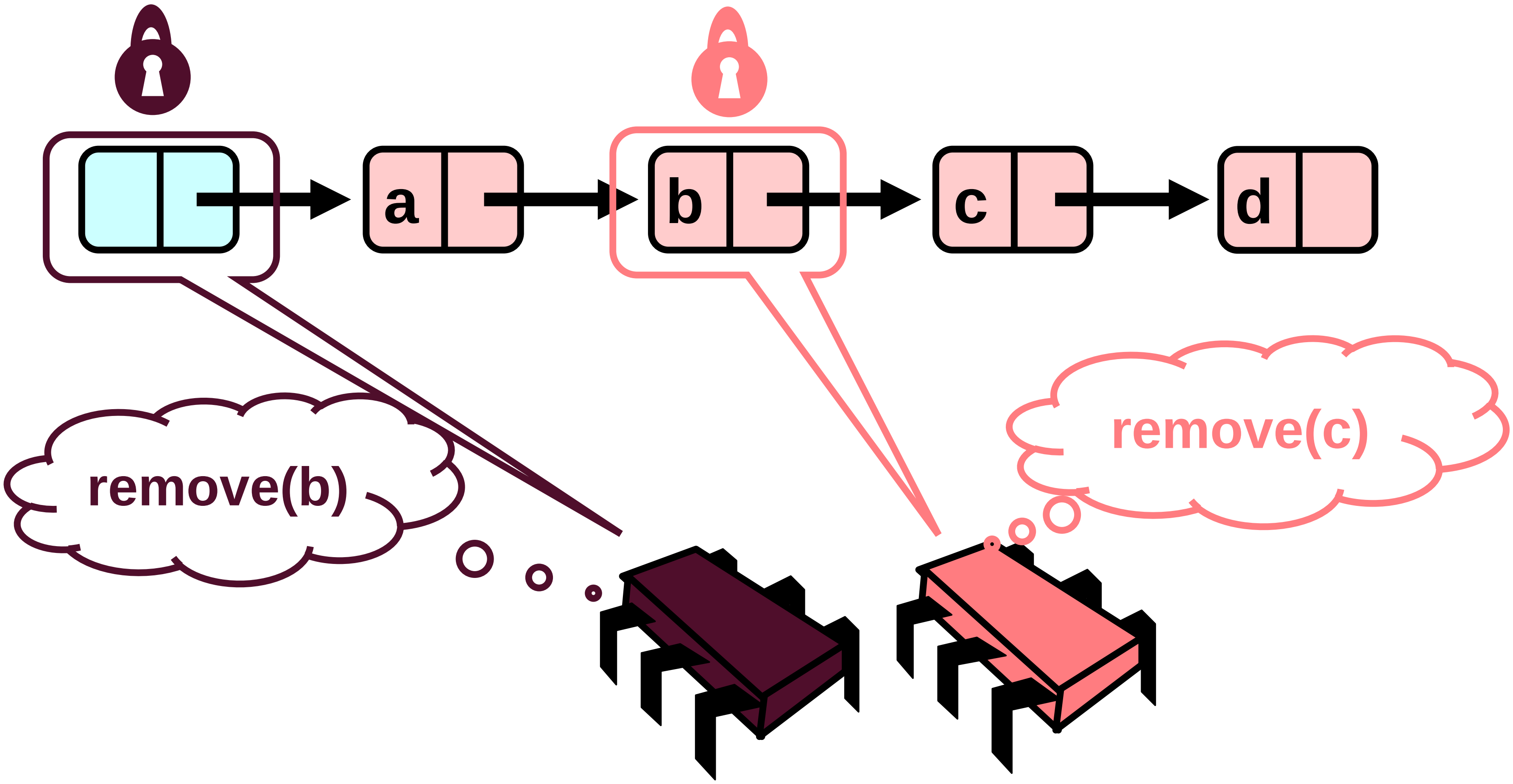
Concurrent Removes



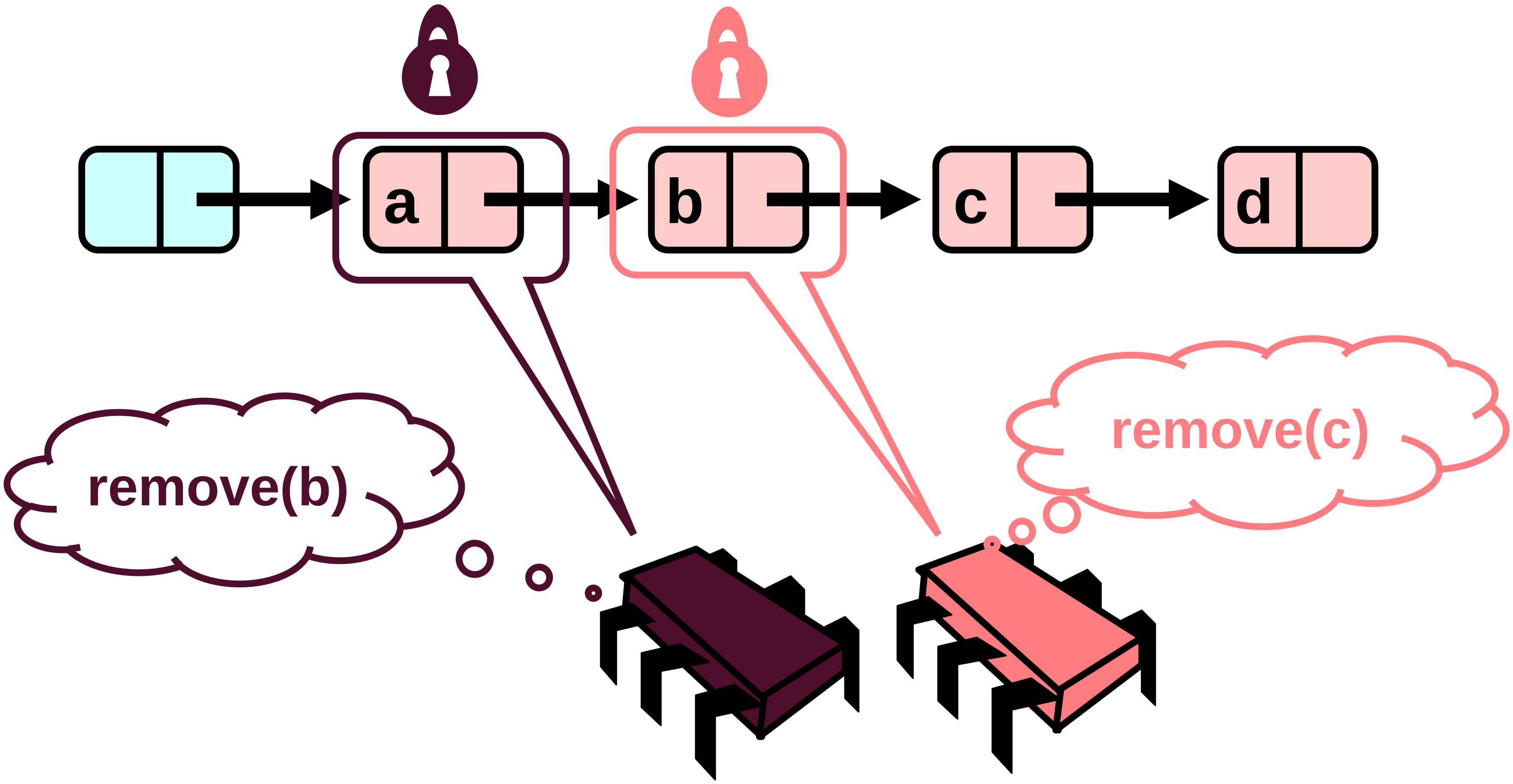
Concurrent Removes



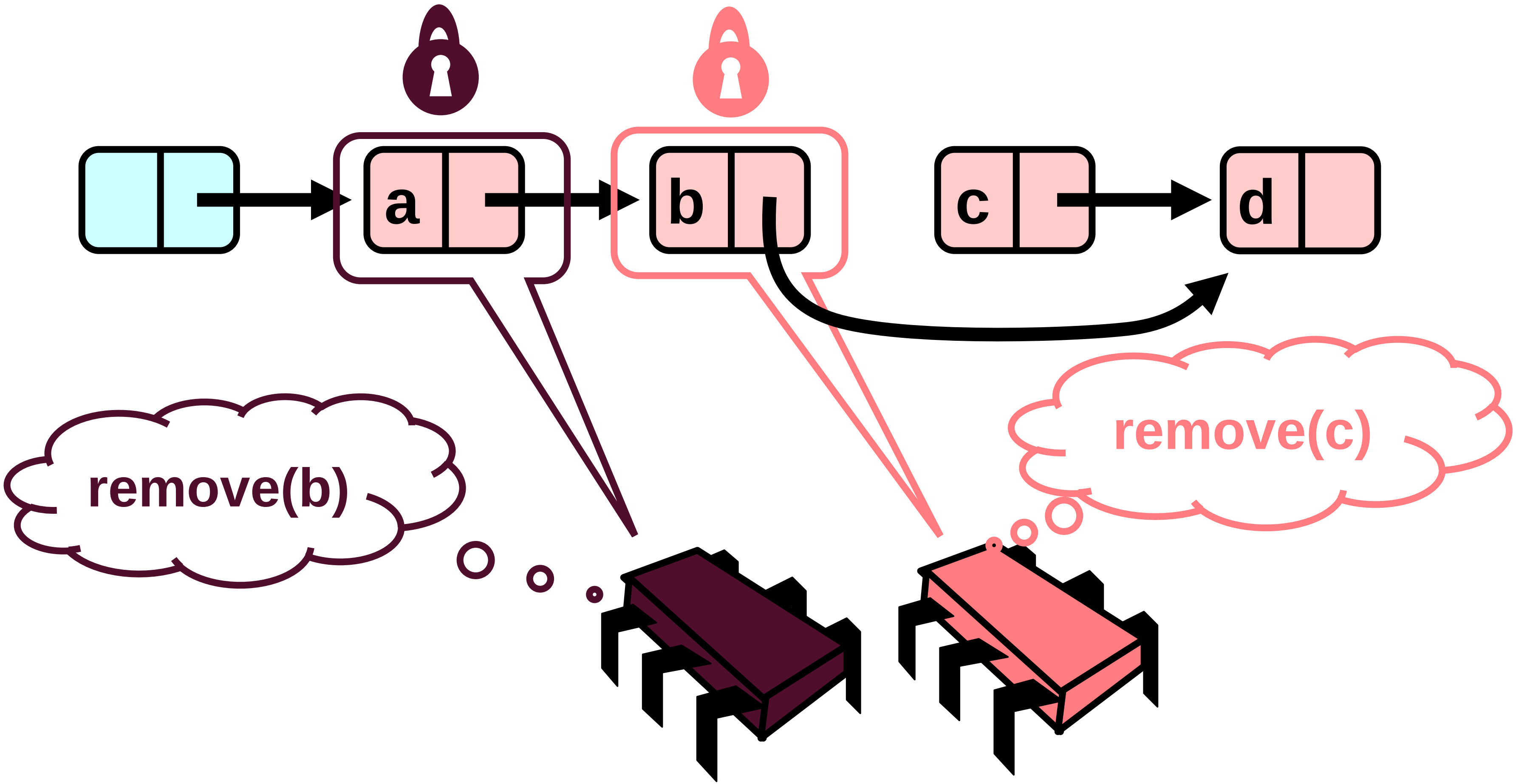
Concurrent Removes



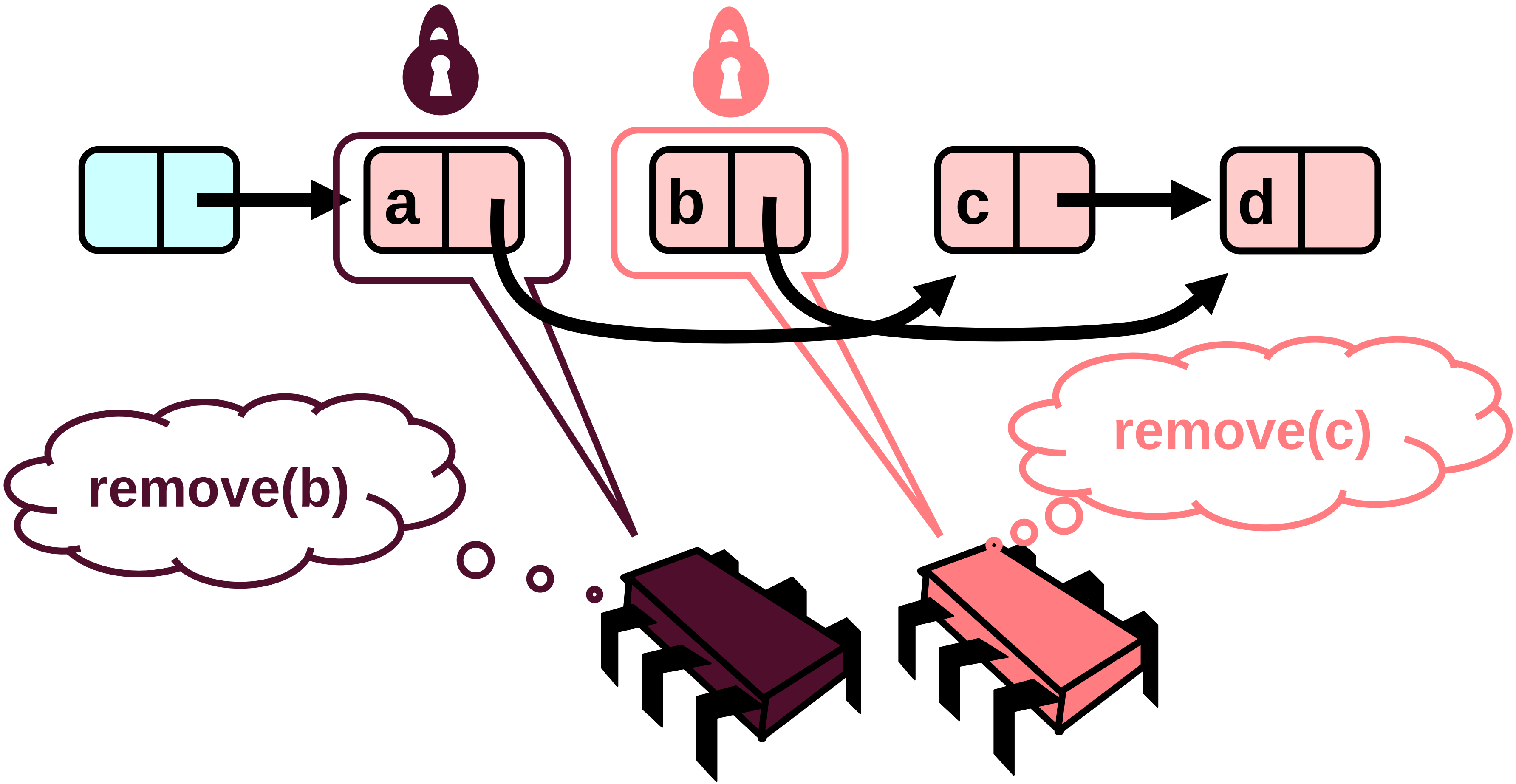
Concurrent Removes



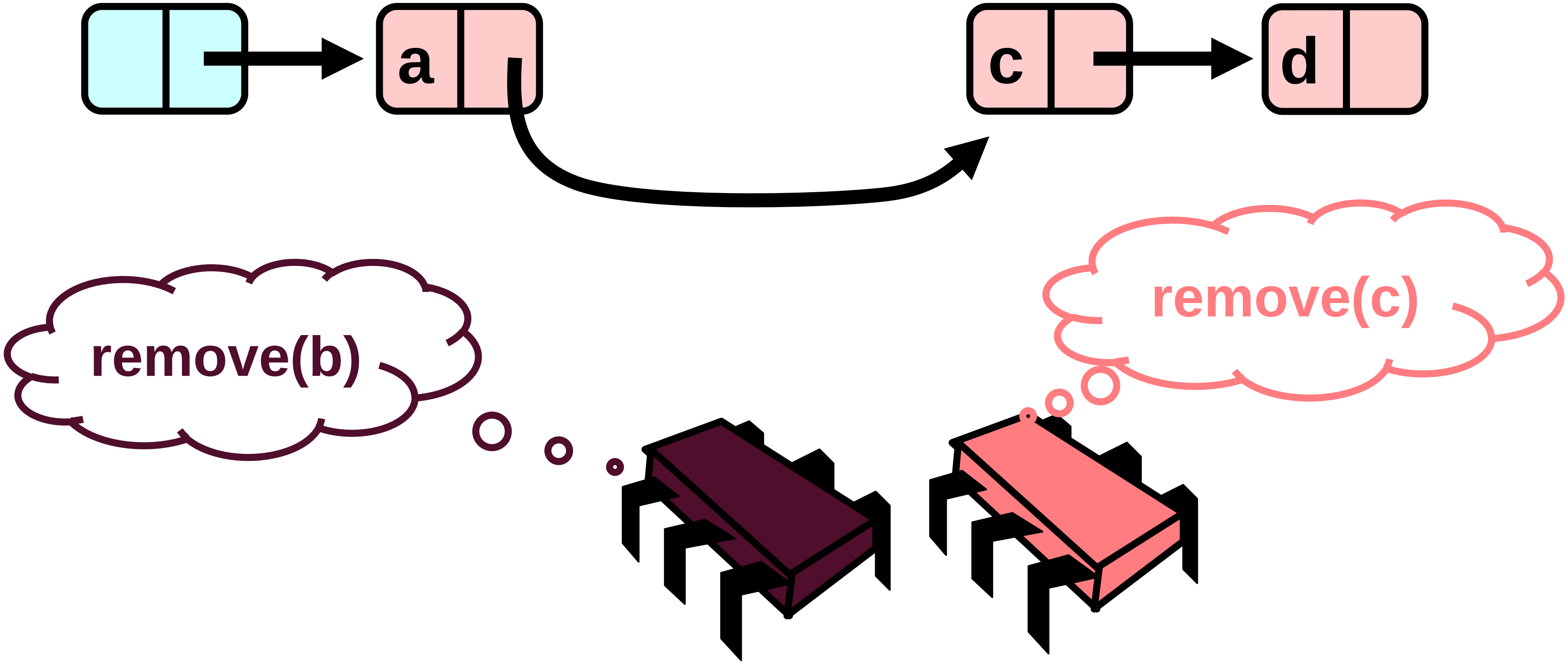
Concurrent Removes



Concurrent Removes

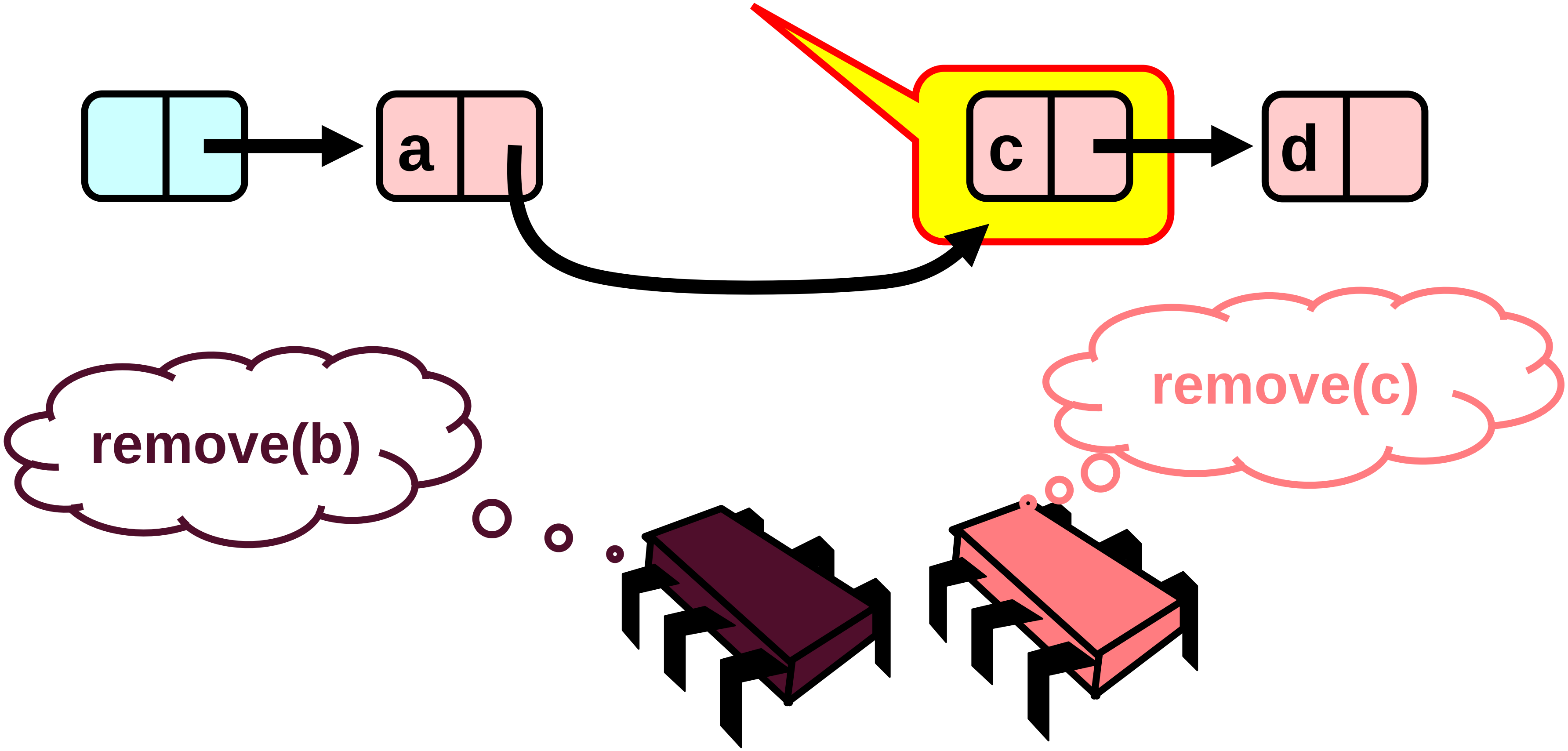


Uh, Oh



Uh, Oh

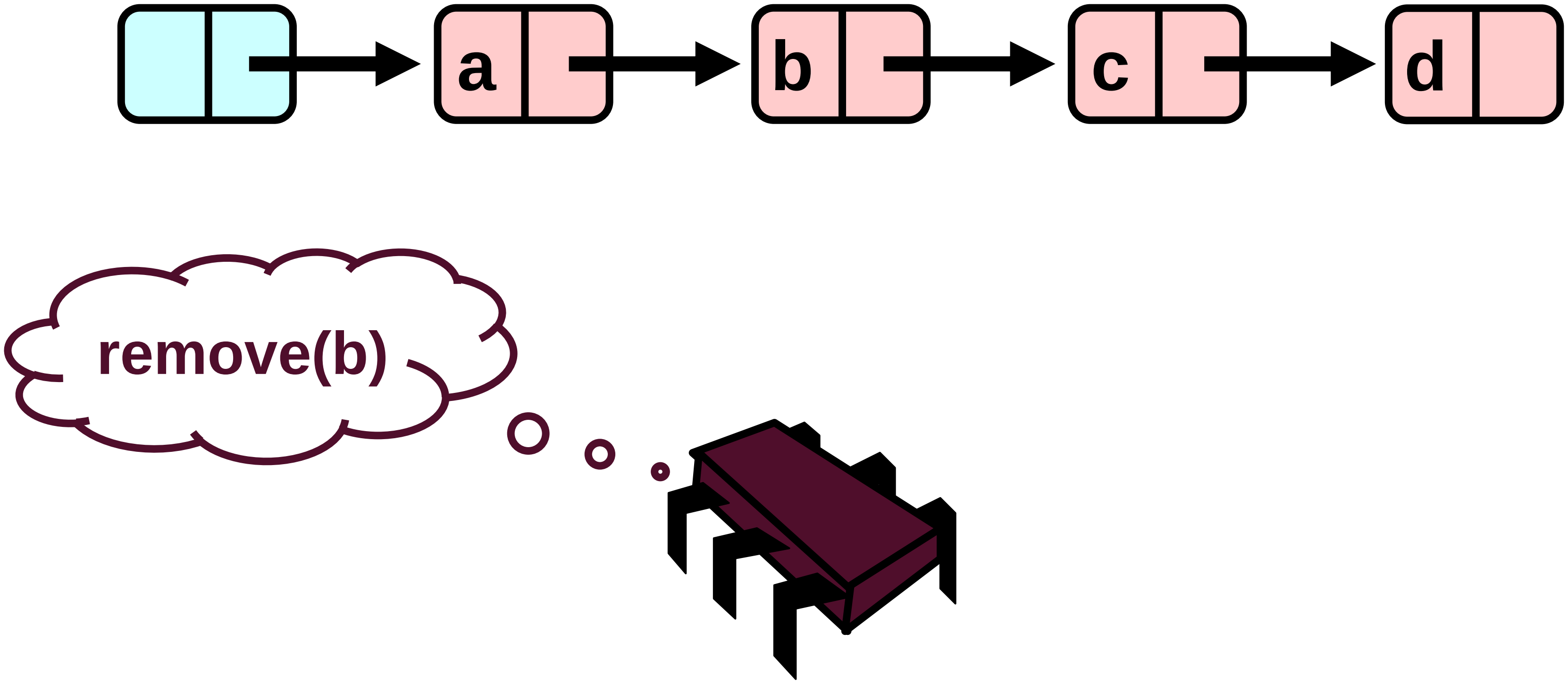
Bad news, c not removed



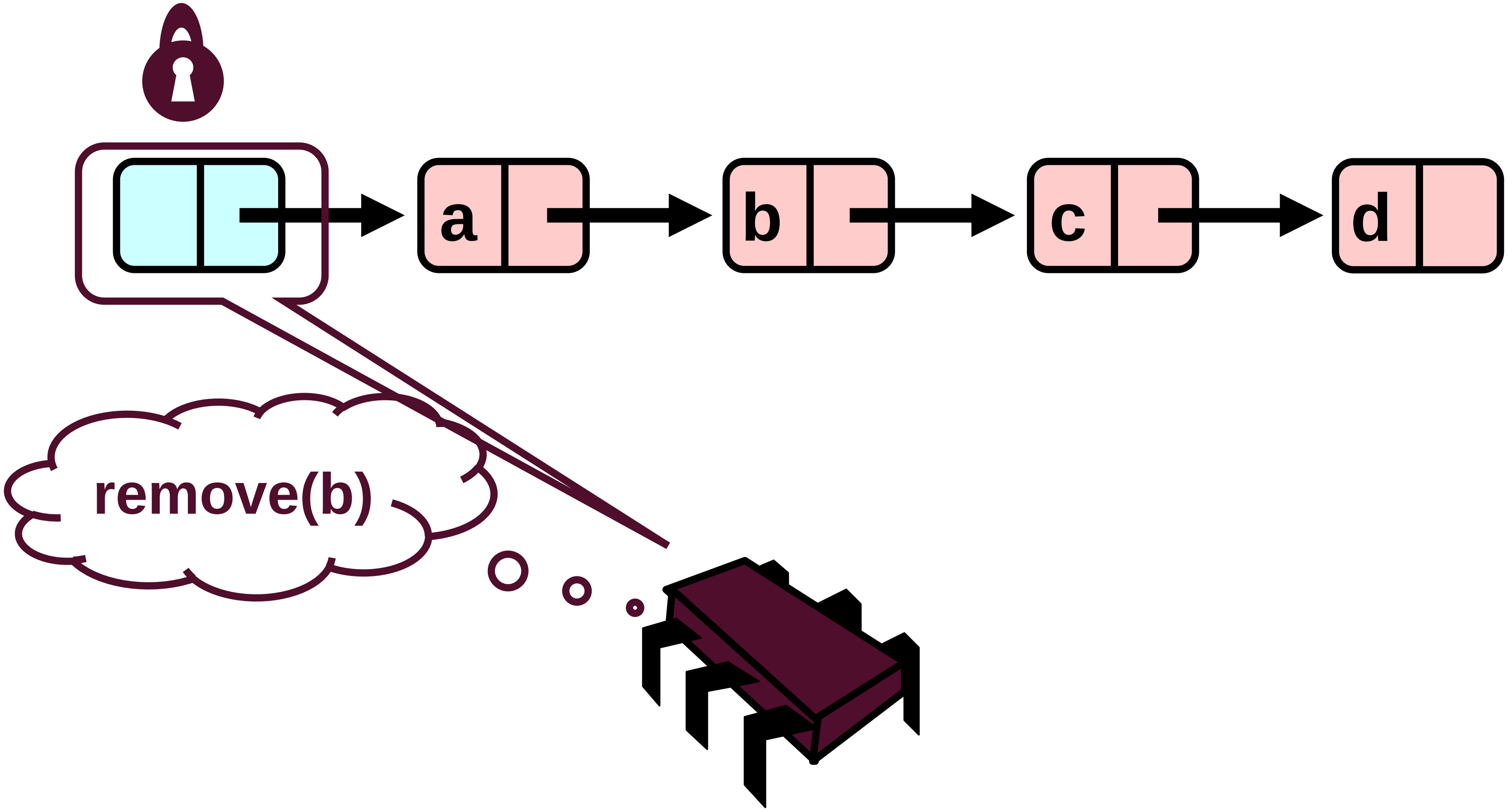
Insight

- **If a node x is locked**
  - Successor of x cannot be deleted!
- **Thus, safe locking is**
  - Lock node to be deleted
  - And its predecessor!
  - → hand-over-hand locking

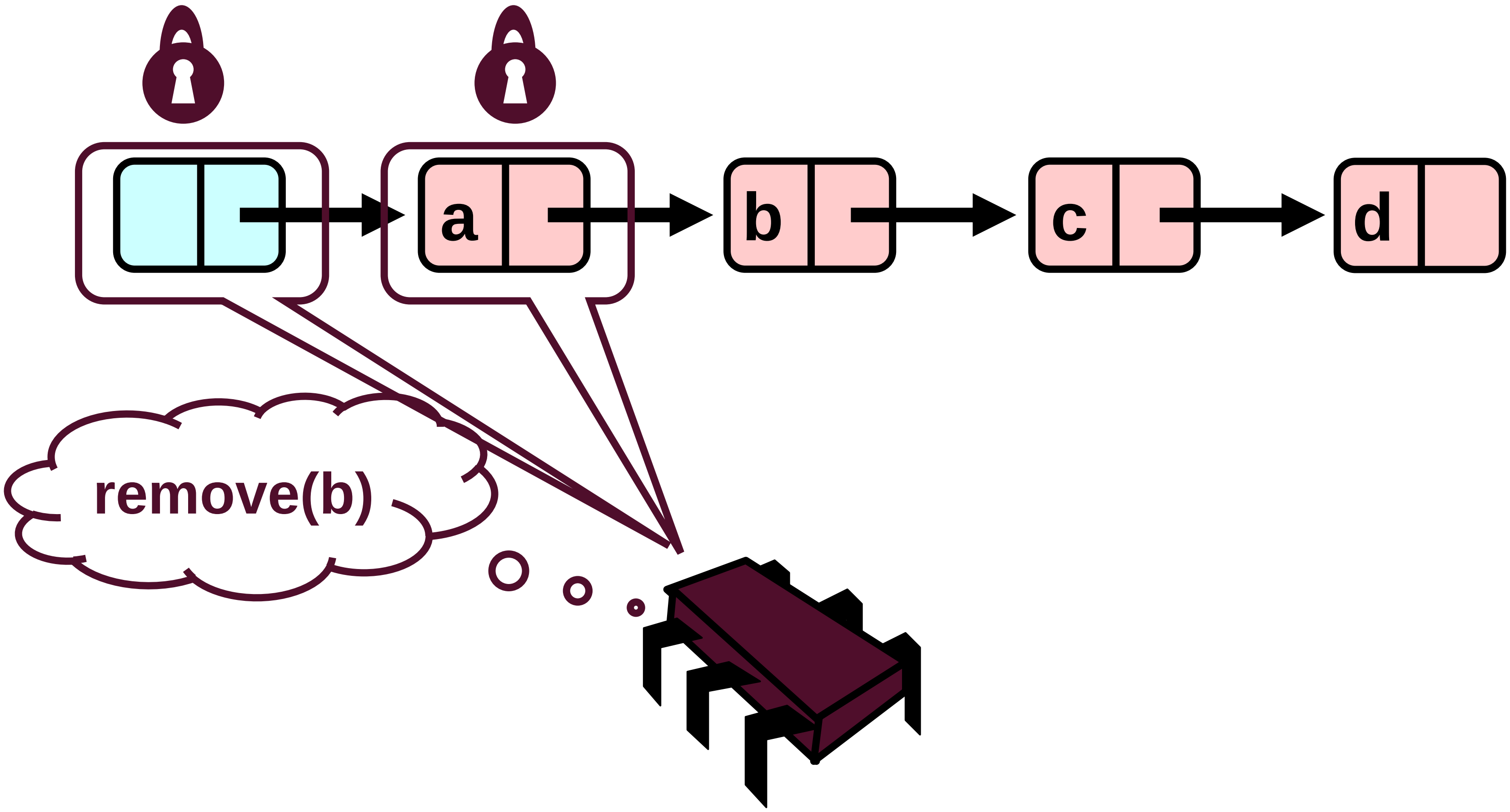
Hand-Over-Hand Again



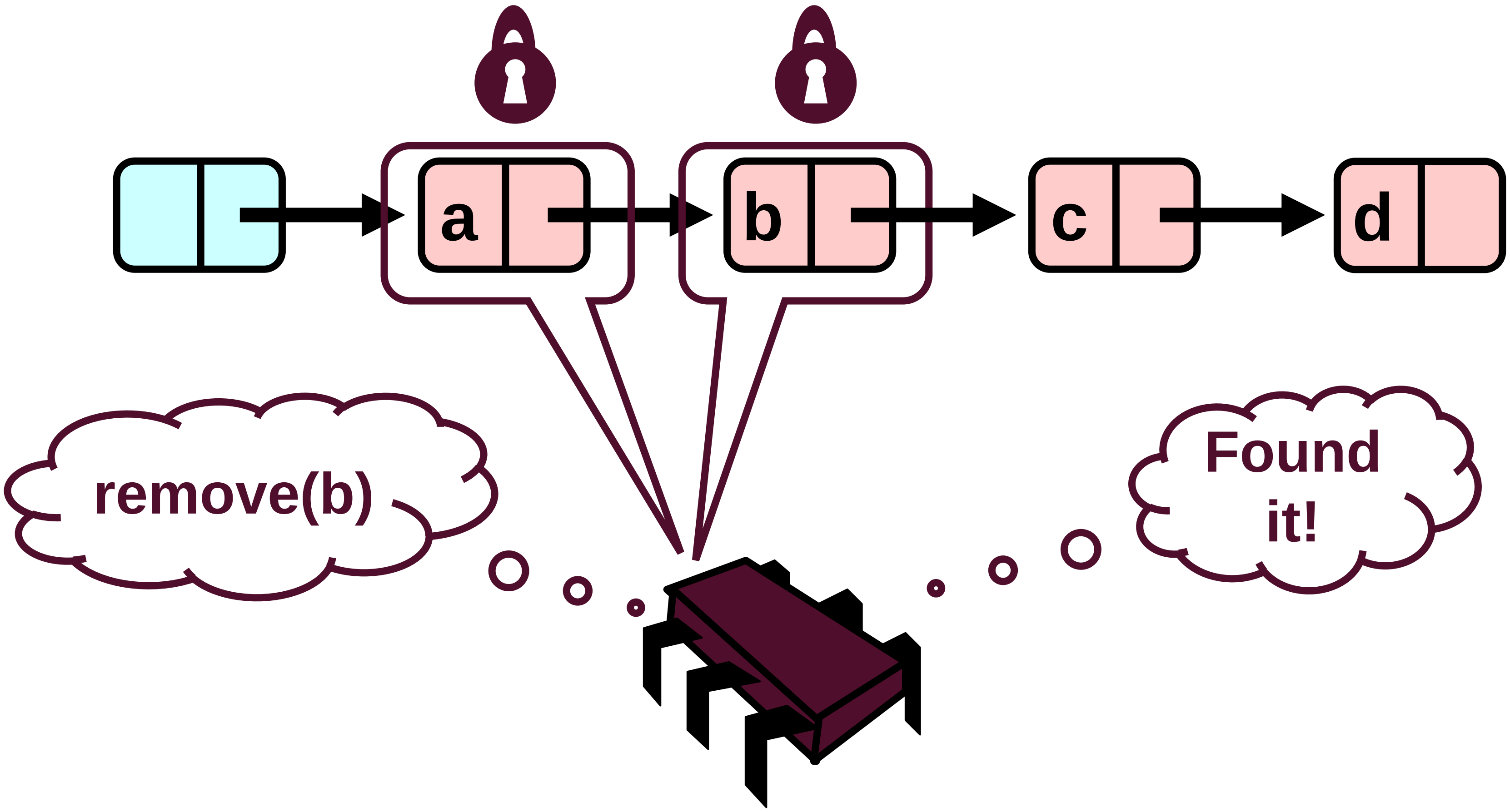
Hand-Over-Hand Again



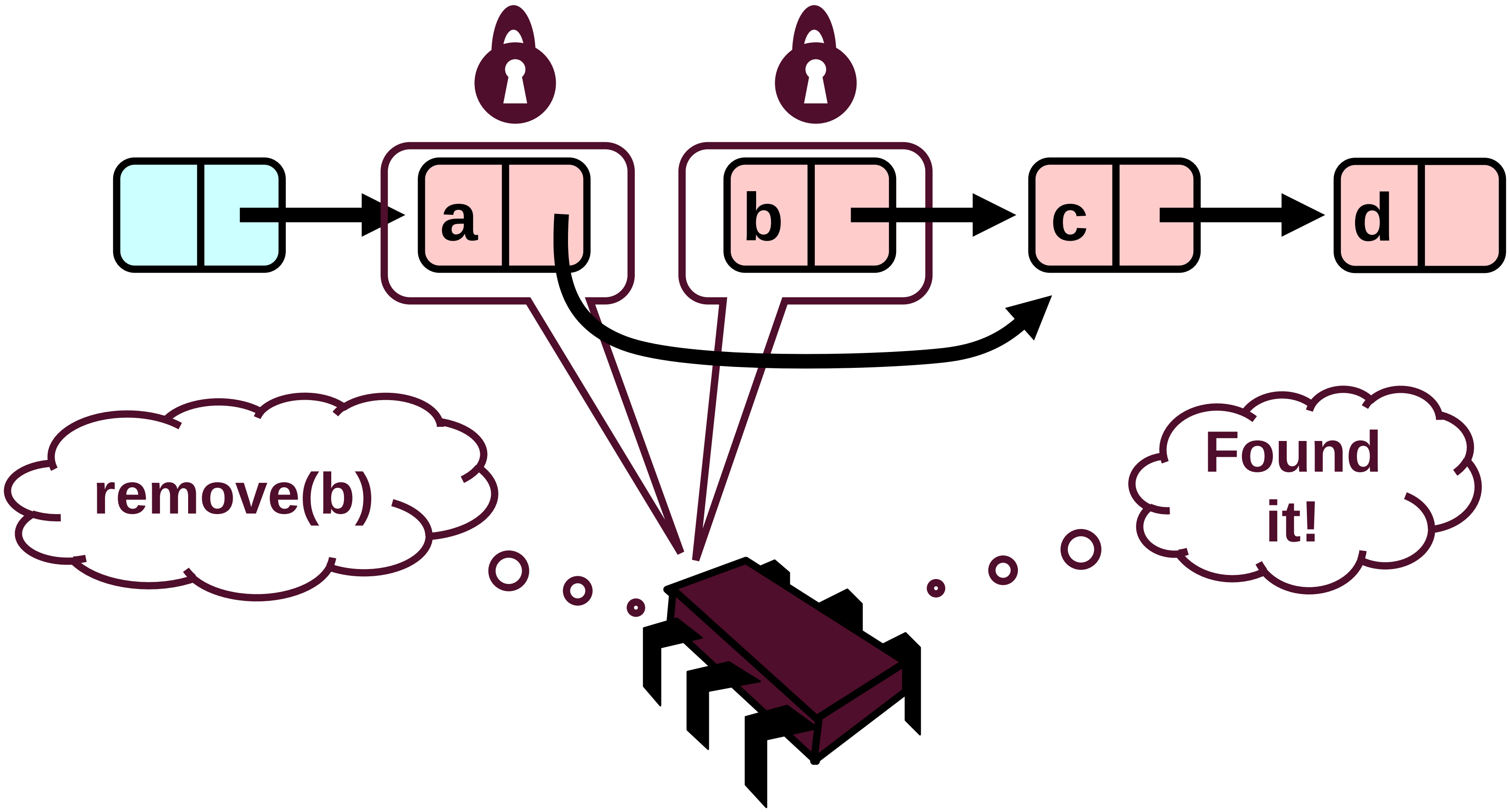
Hand-Over-Hand Again



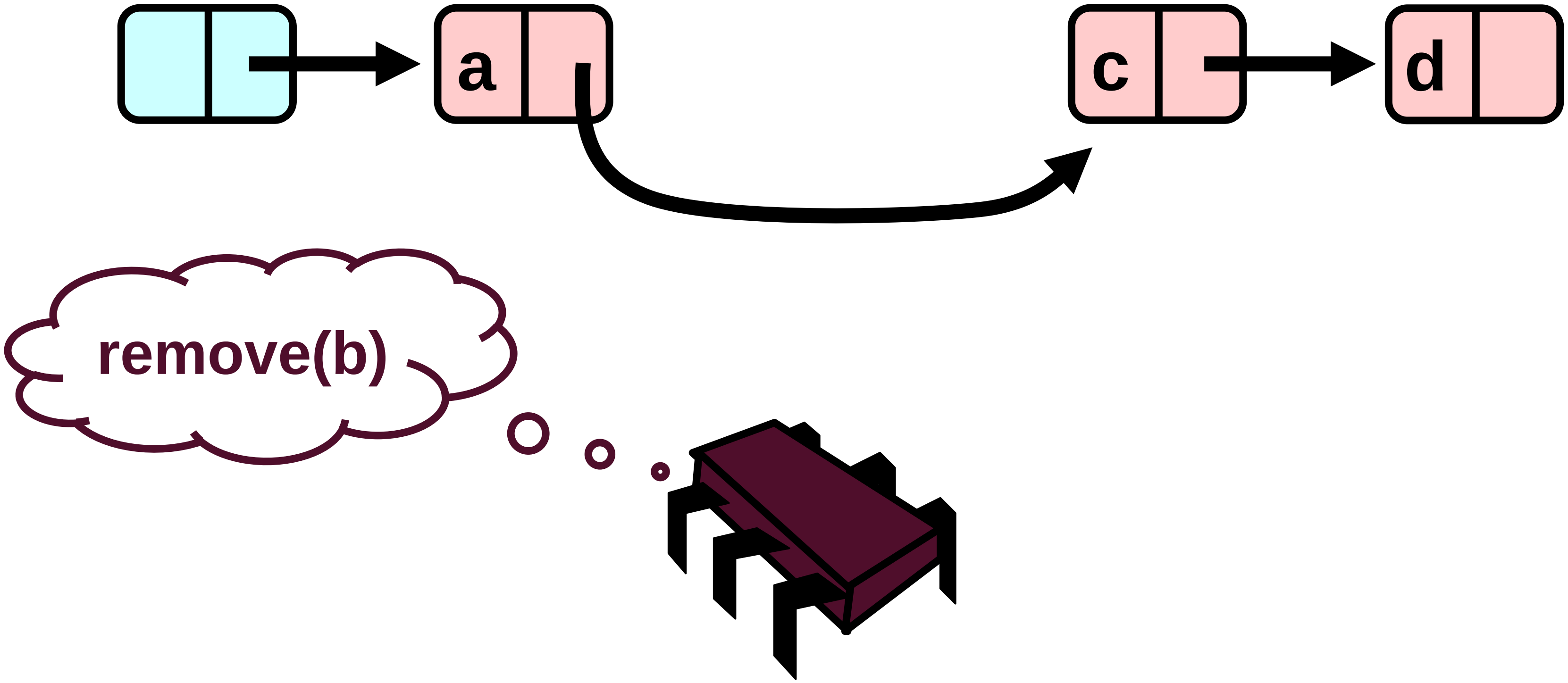
Hand-Over-Hand Again



Hand-Over-Hand Again



Hand-Over-Hand Again



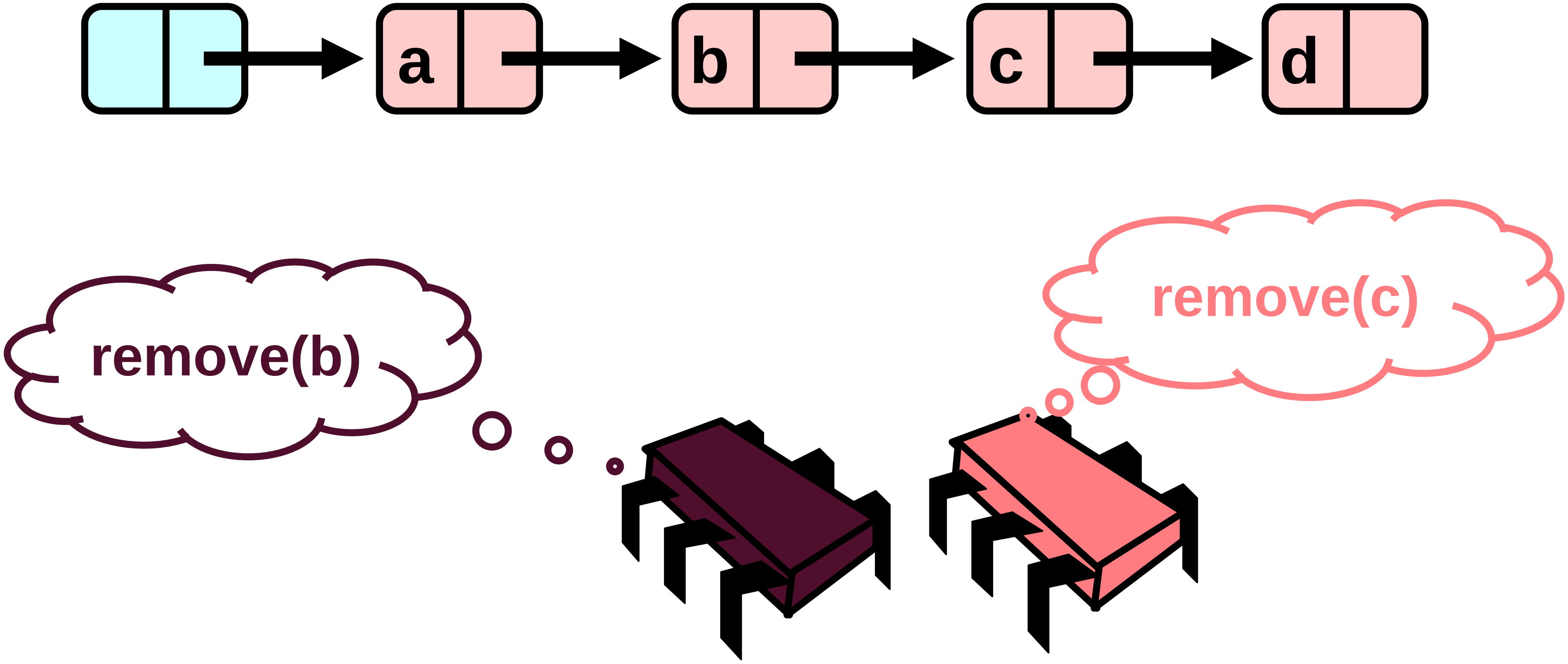
a

c

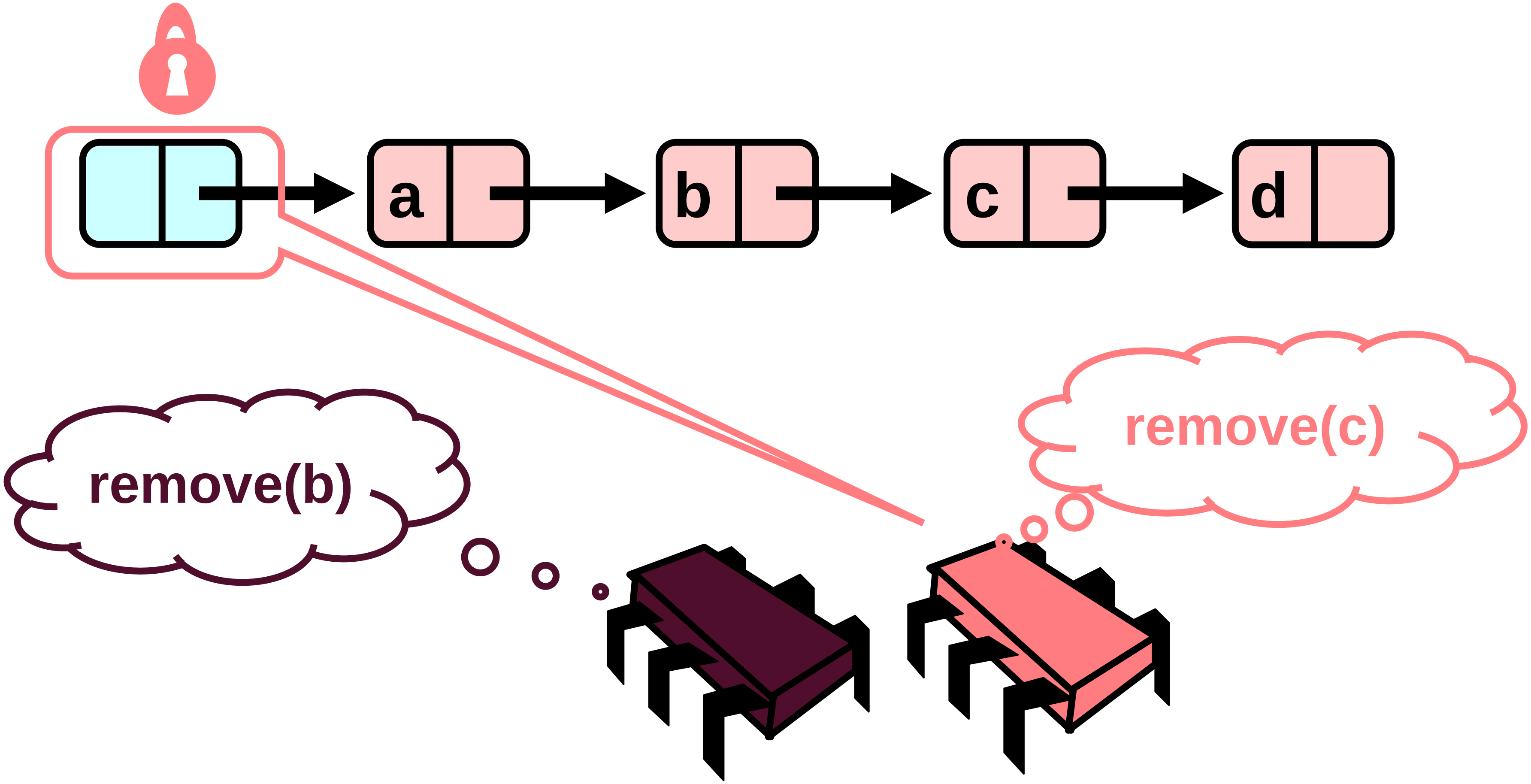
d

remove(b)

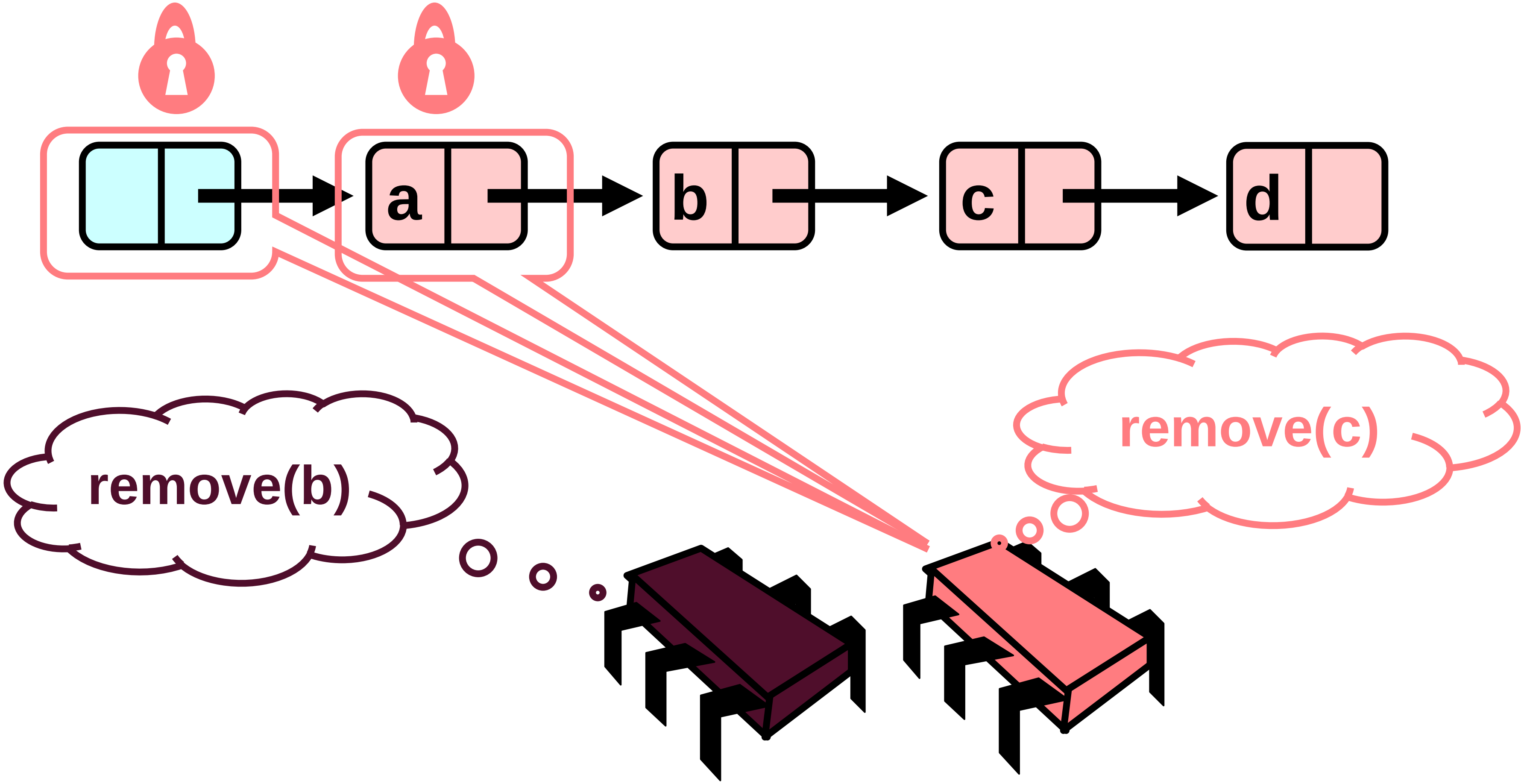
Removing a Node



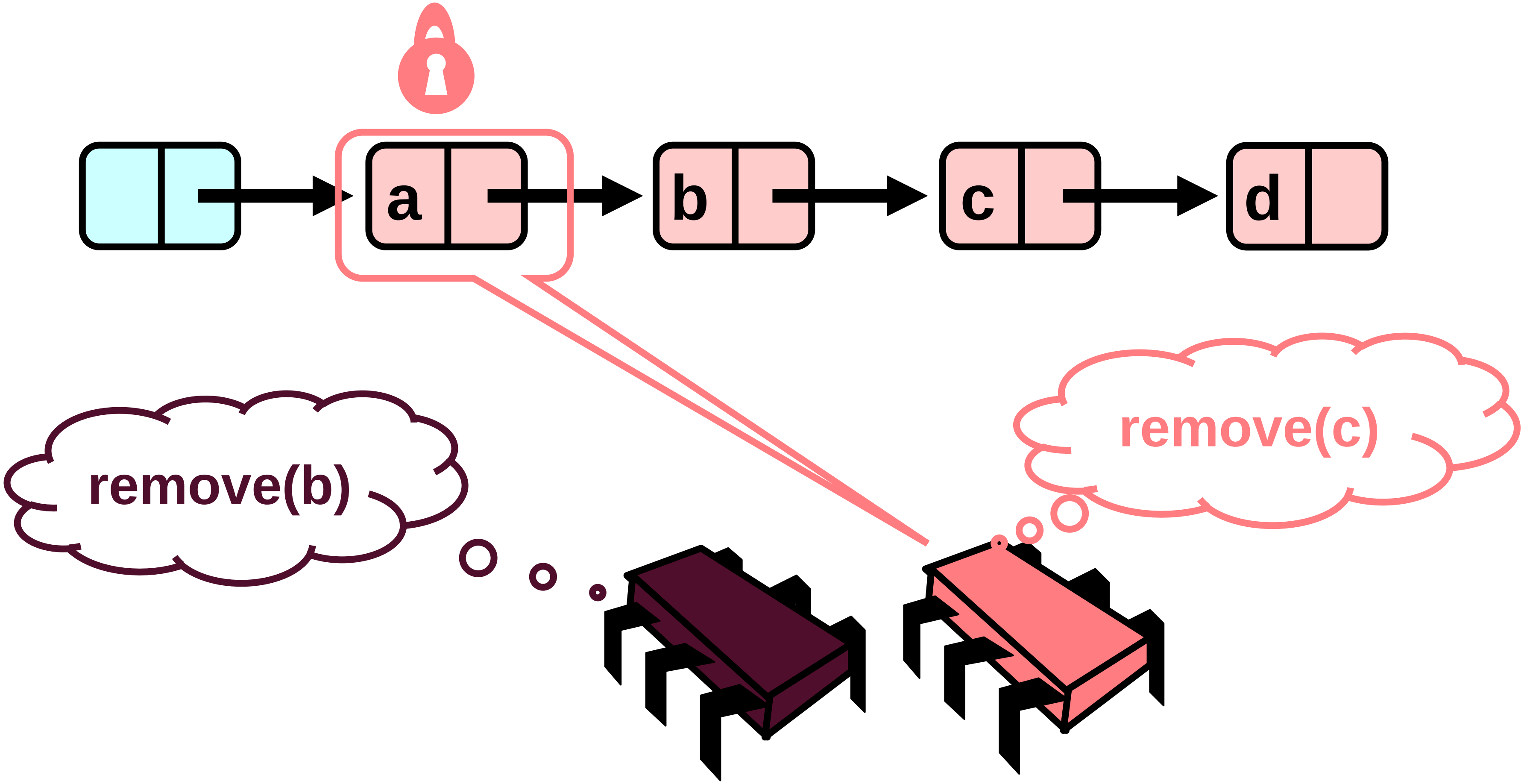
Removing a Node



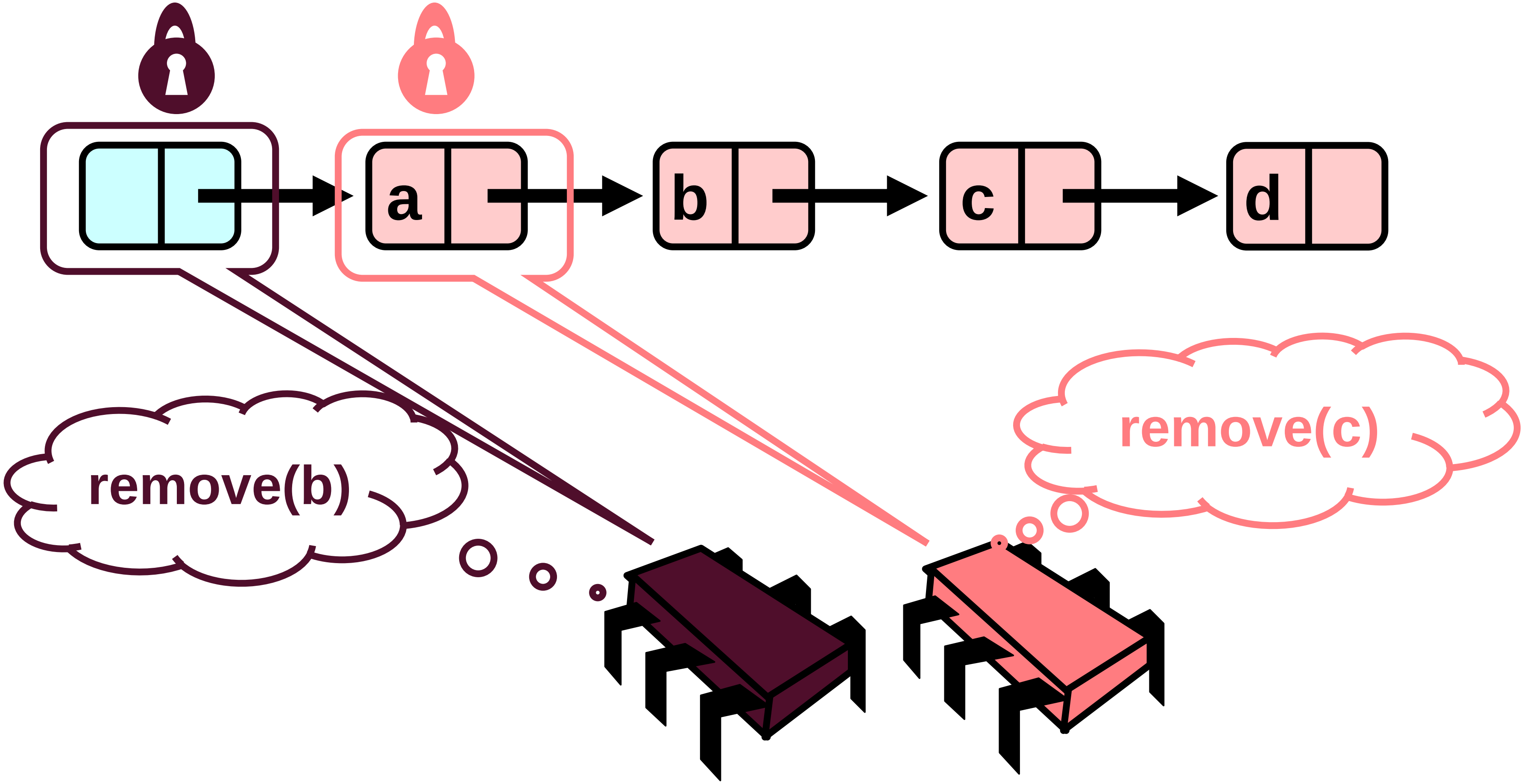
Removing a Node



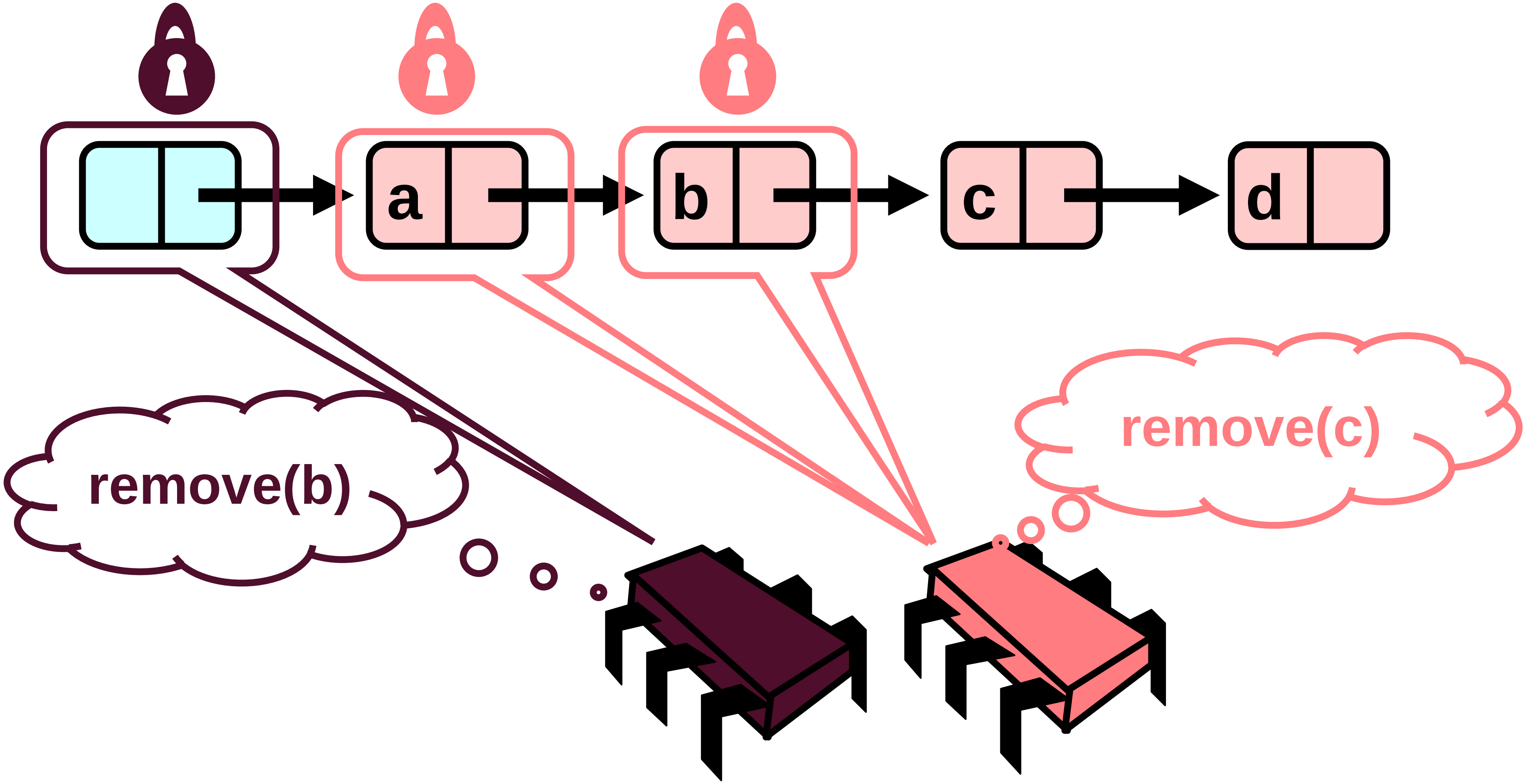
Removing a Node



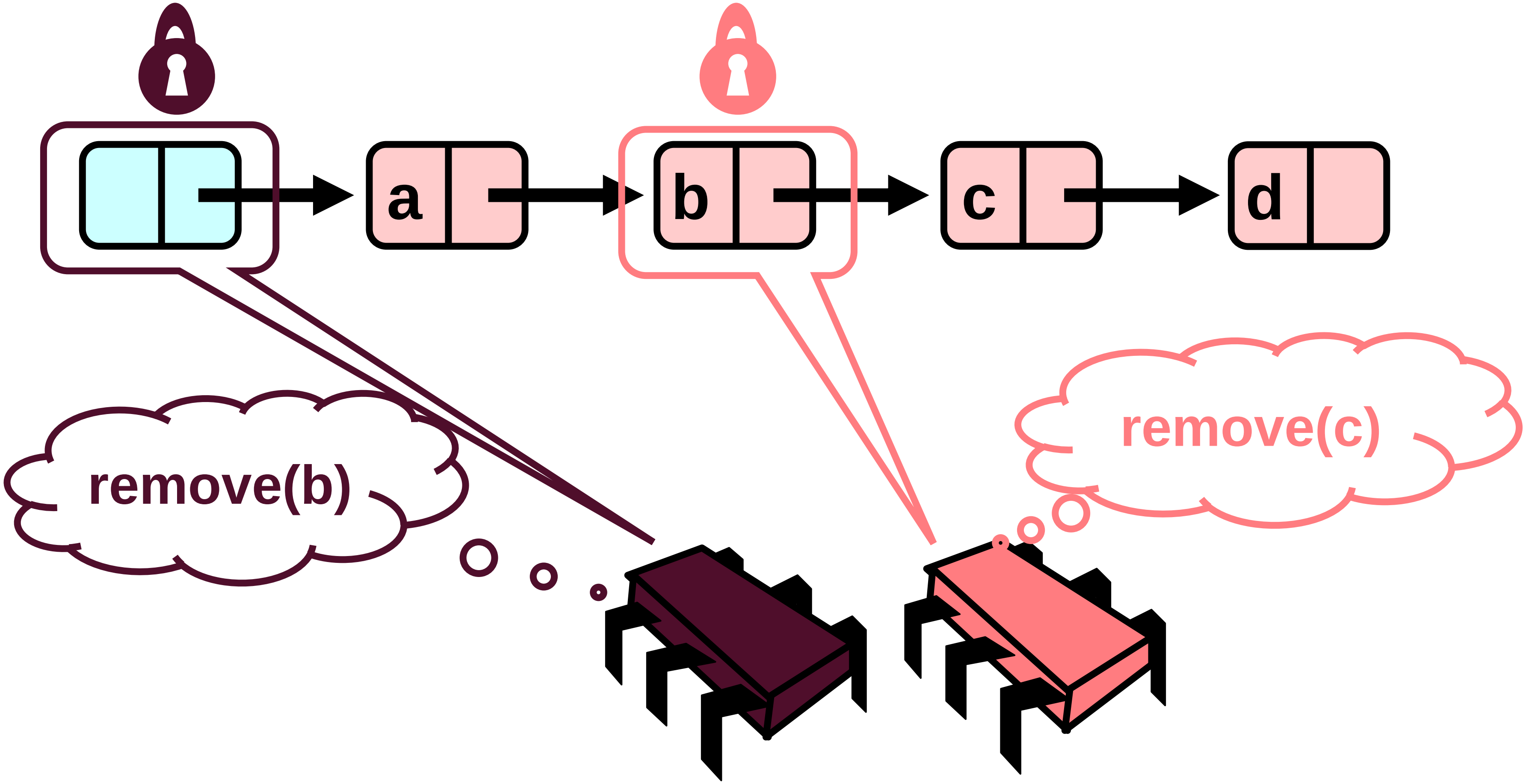
Removing a Node



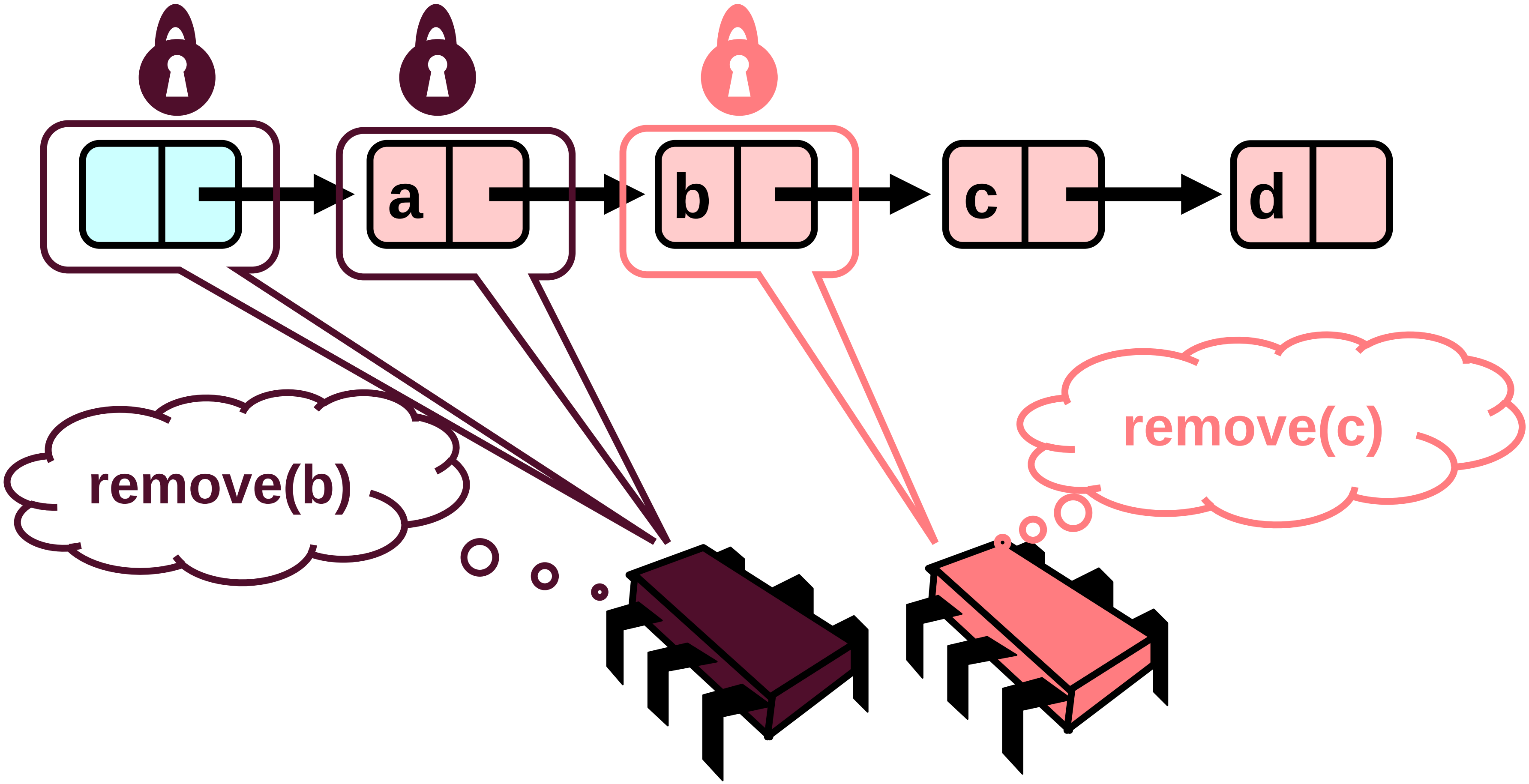
Removing a Node



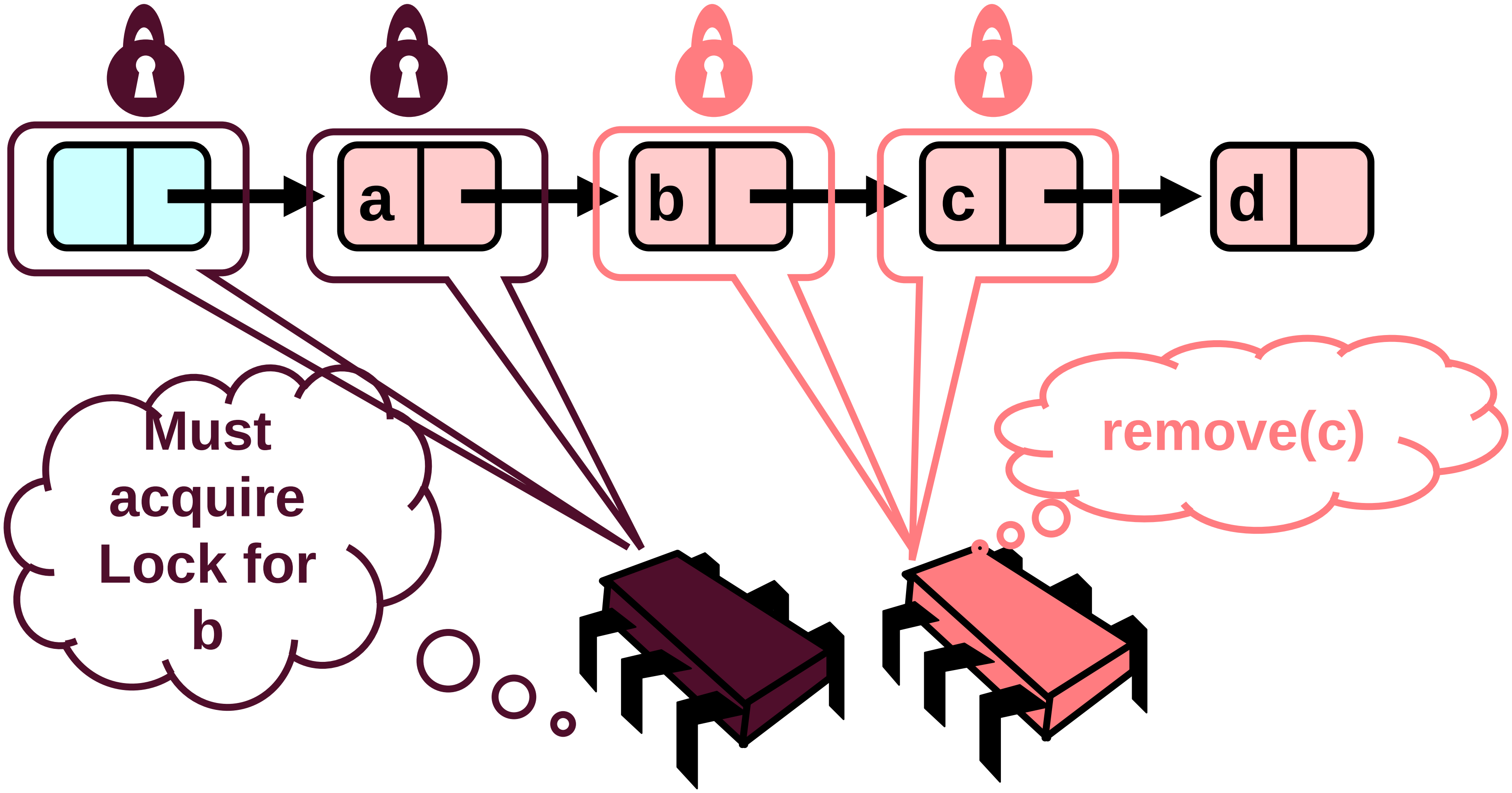
Removing a Node



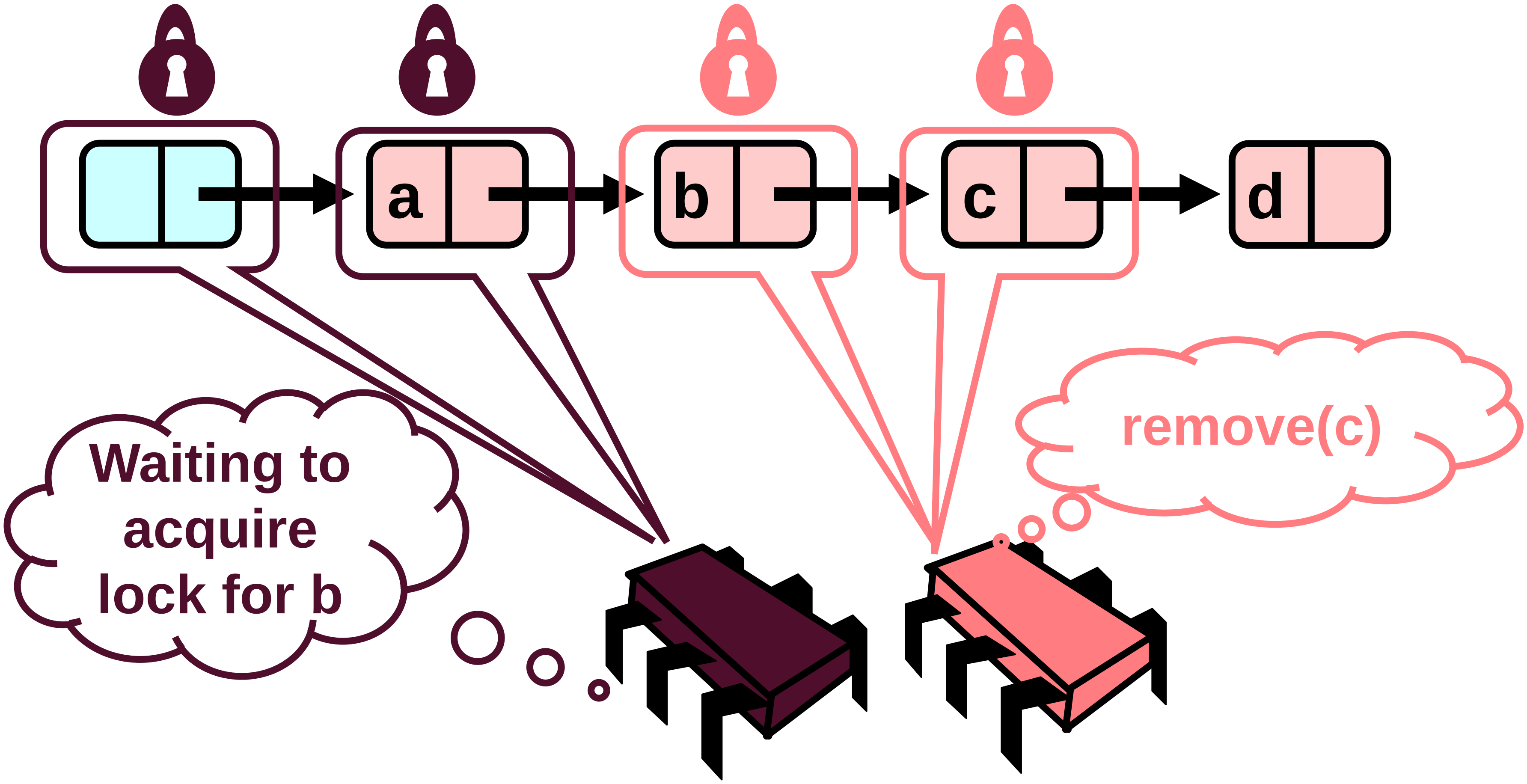
Removing a Node



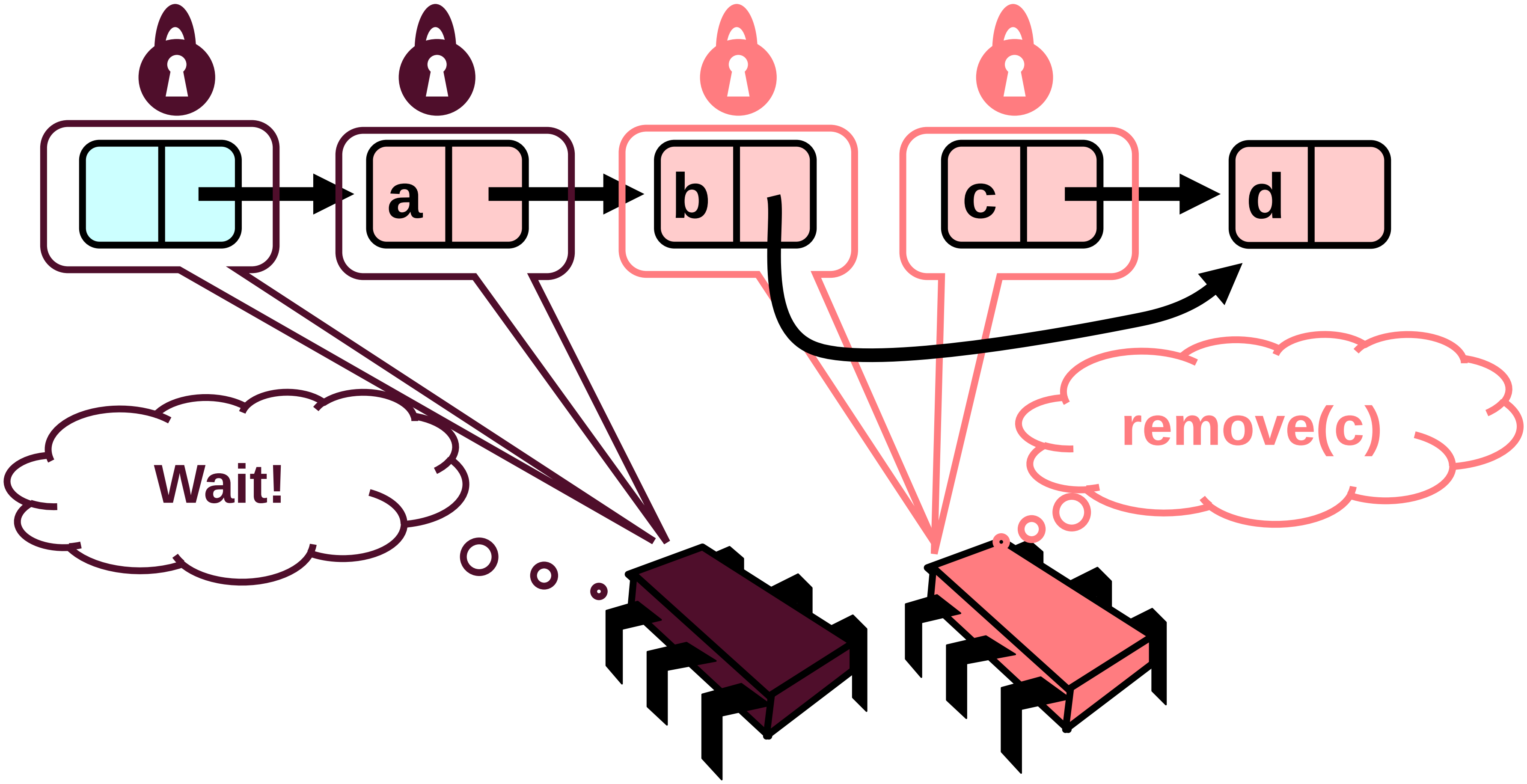
Removing a Node



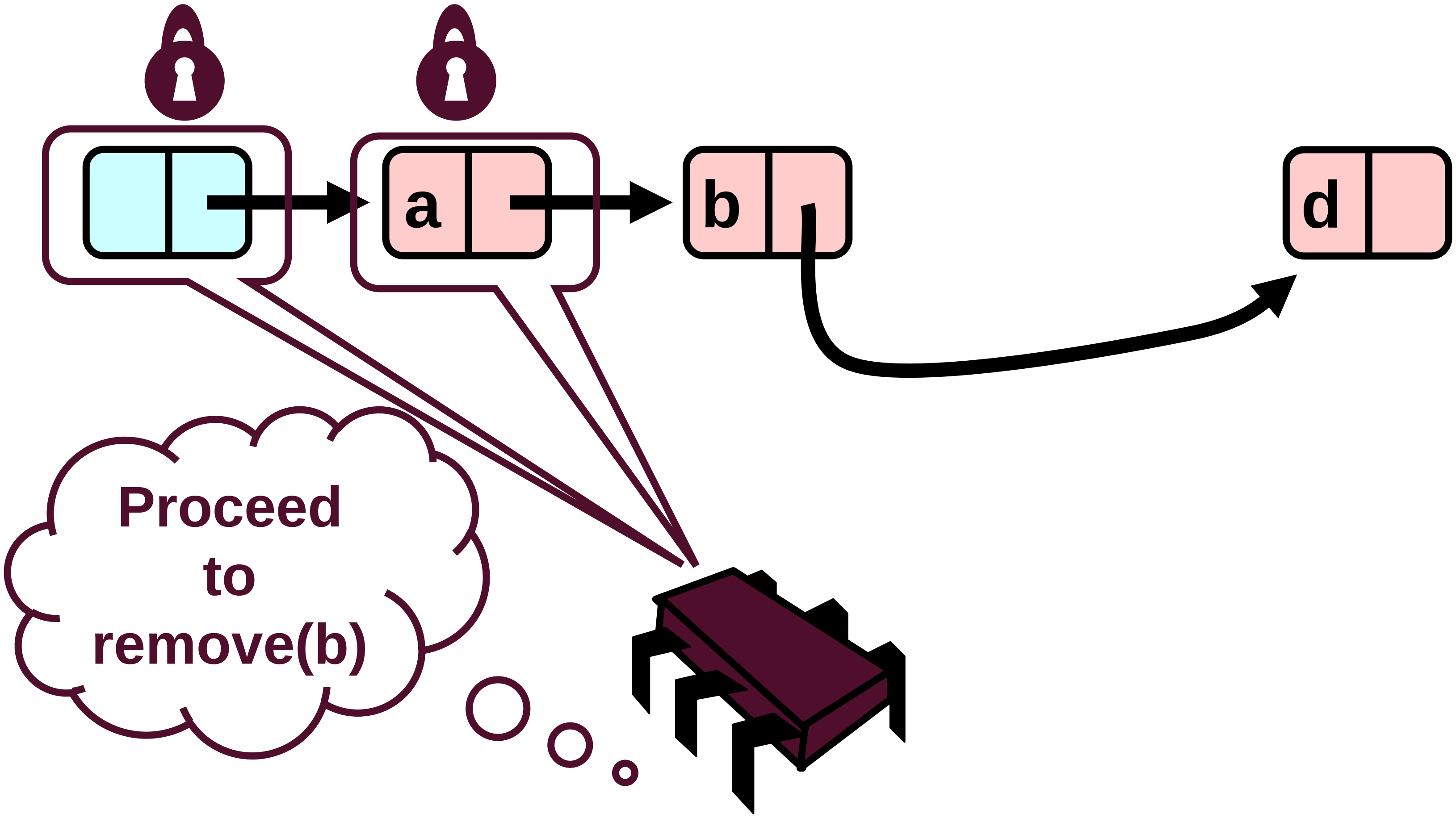
Removing a Node



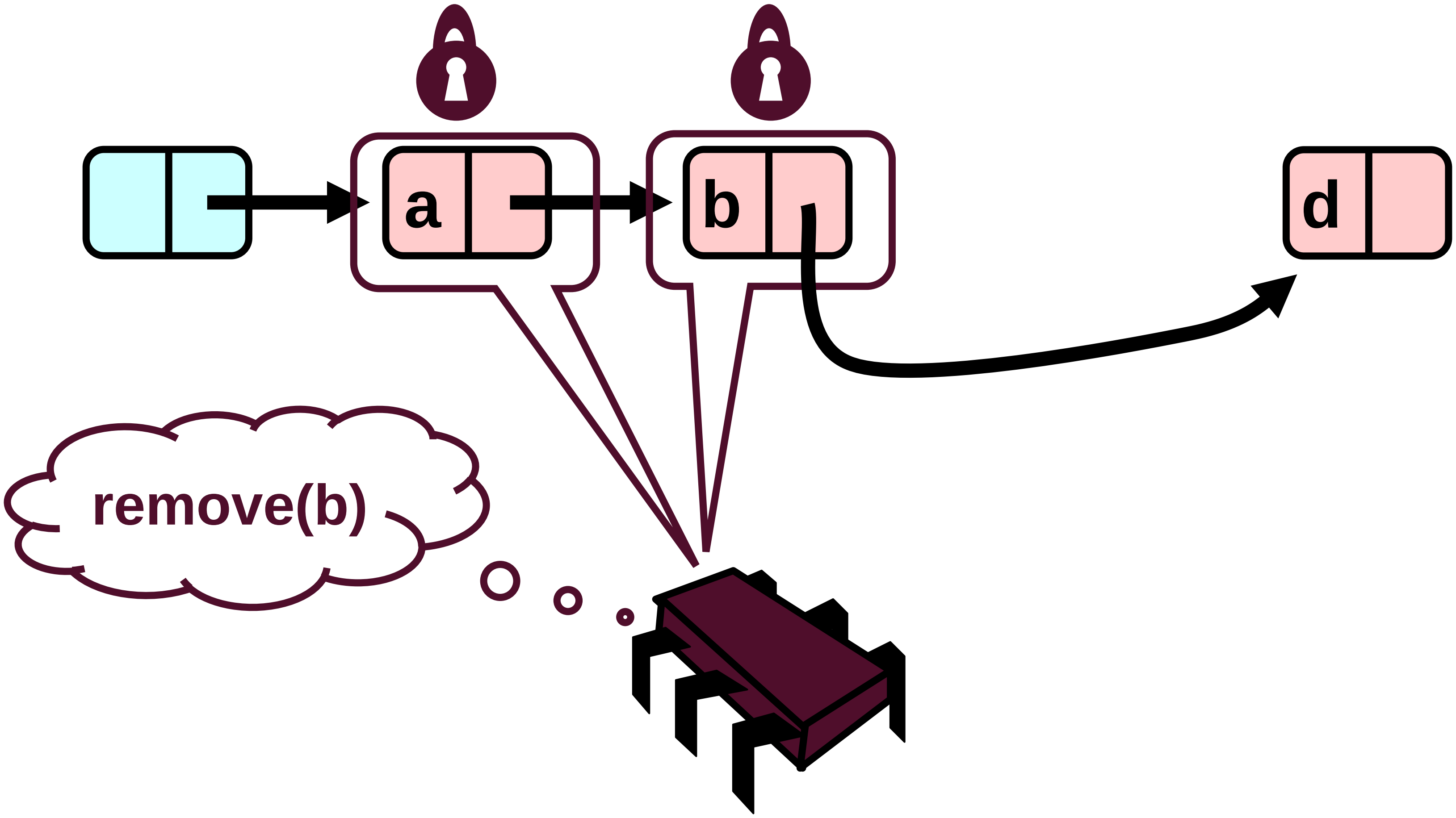
Removing a Node



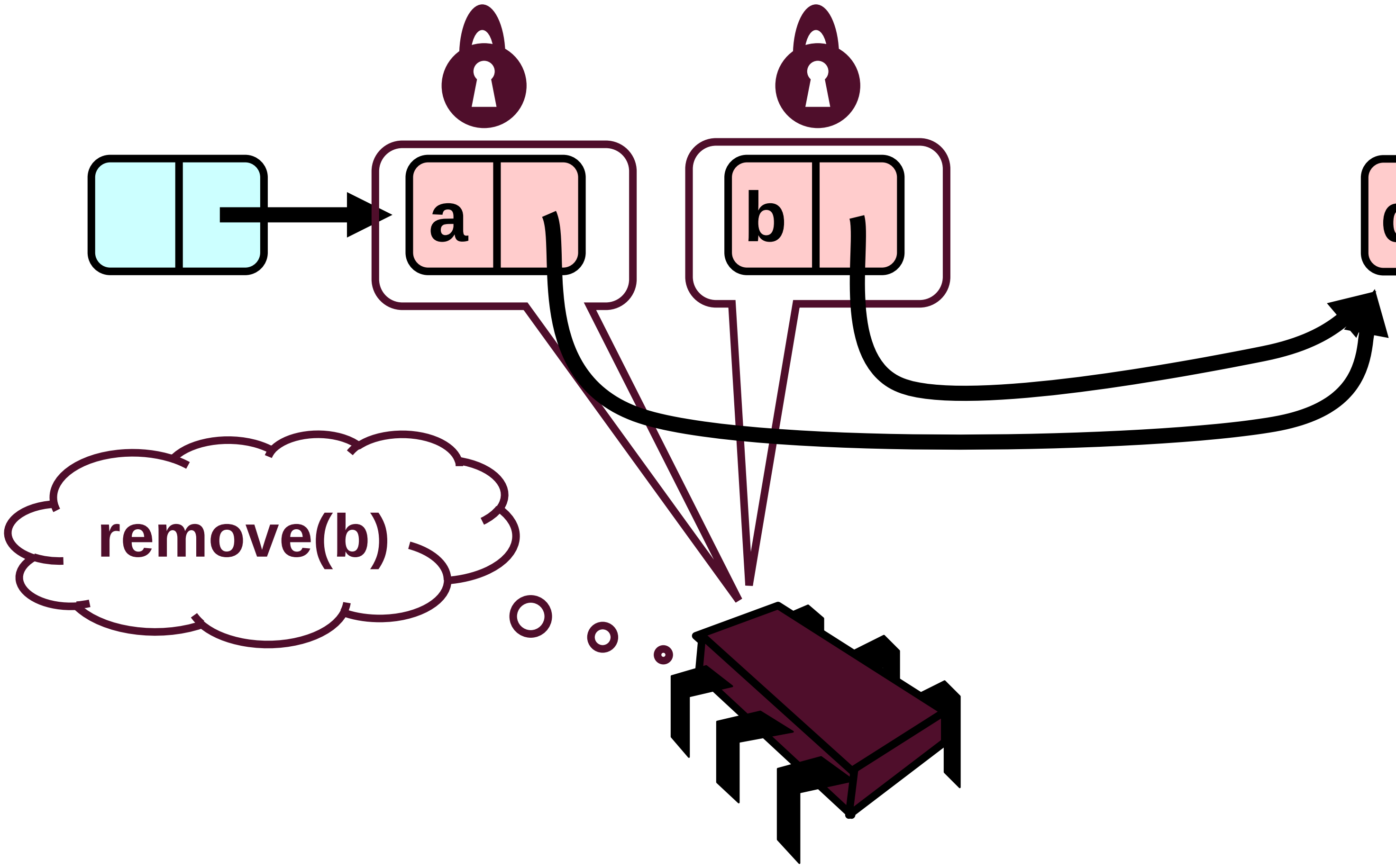
Removing a Node



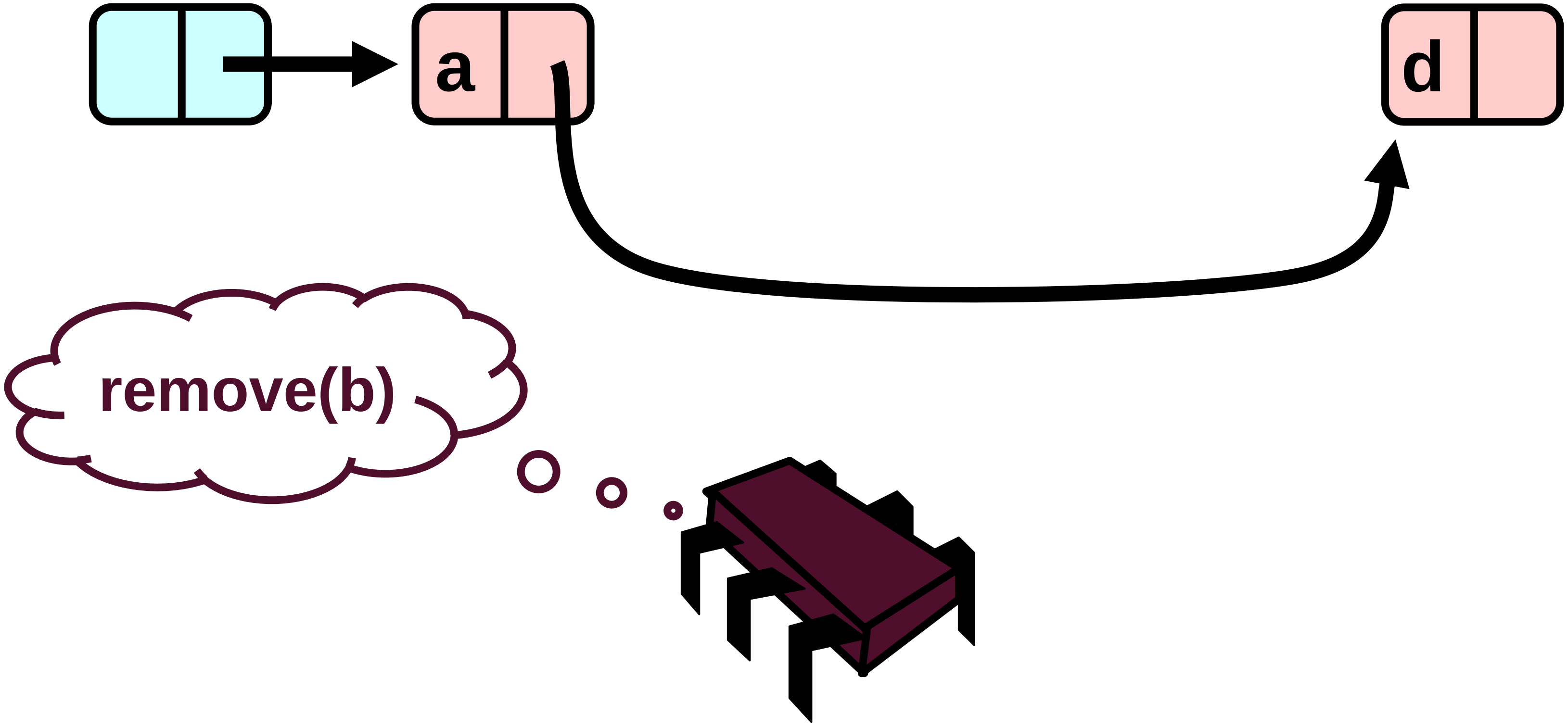
Removing a Node



Removing a Node



Removing a Node



What are the Issues?

- **We have fine-grained locking, will there be contention?**
  - Yes, the list can only be traversed sequentially, a remove of the 3rd item will block all other threads!
  - This is essentially still serialized if the list is short (since threads can only pipeline on list elements)
- **Other problems, ignoring contention?**
  - Must acquire O(|S|) locks

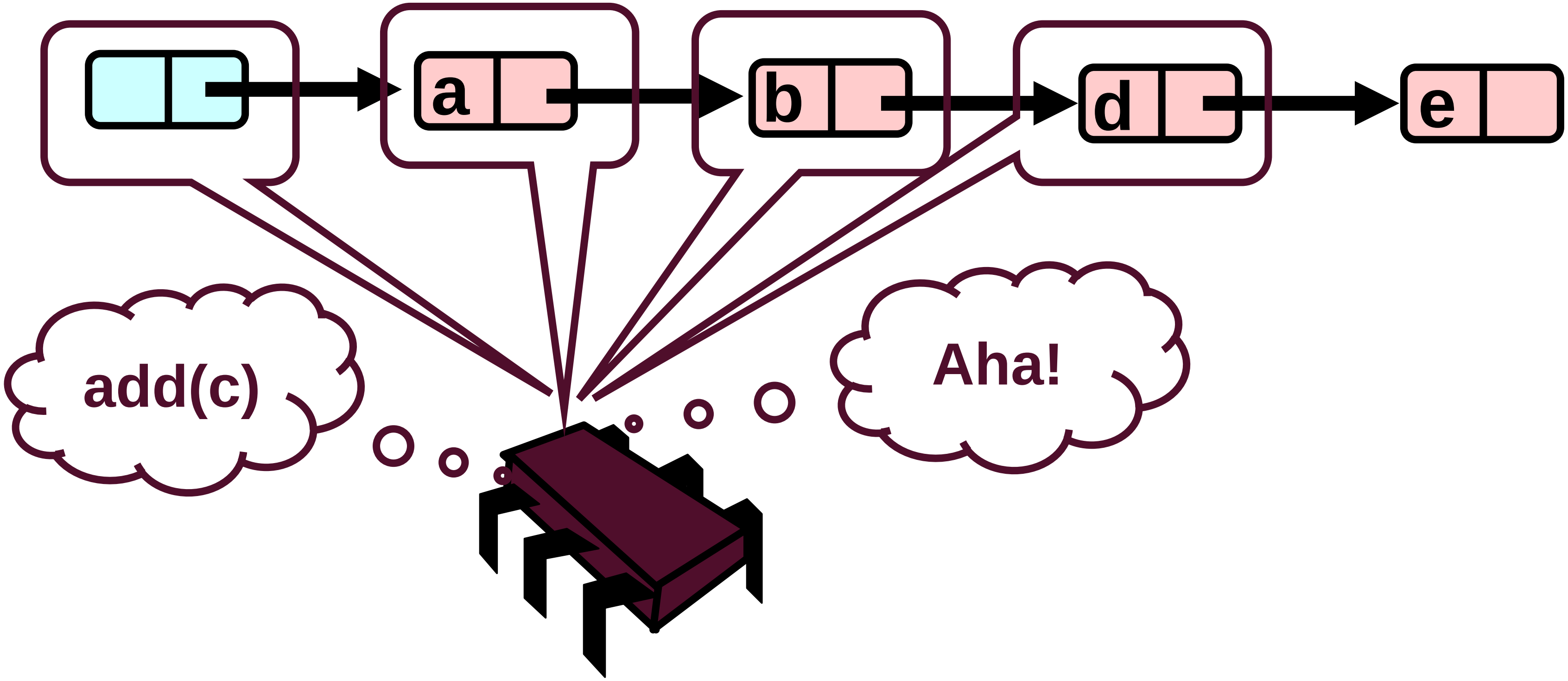
Trick 2: Reader/Writer Locking

- **Same hand-over-hand locking**
  - Traversal uses reader locks
  - Once add finds position or remove finds target node, upgrade **both** locks to writer locks
  - Need to guarantee deadlock and starvation freedom!
- **Allows truly concurrent traversals**
  - Still blocks behind writing threads
  - Still O(|S|) lock/unlock operations

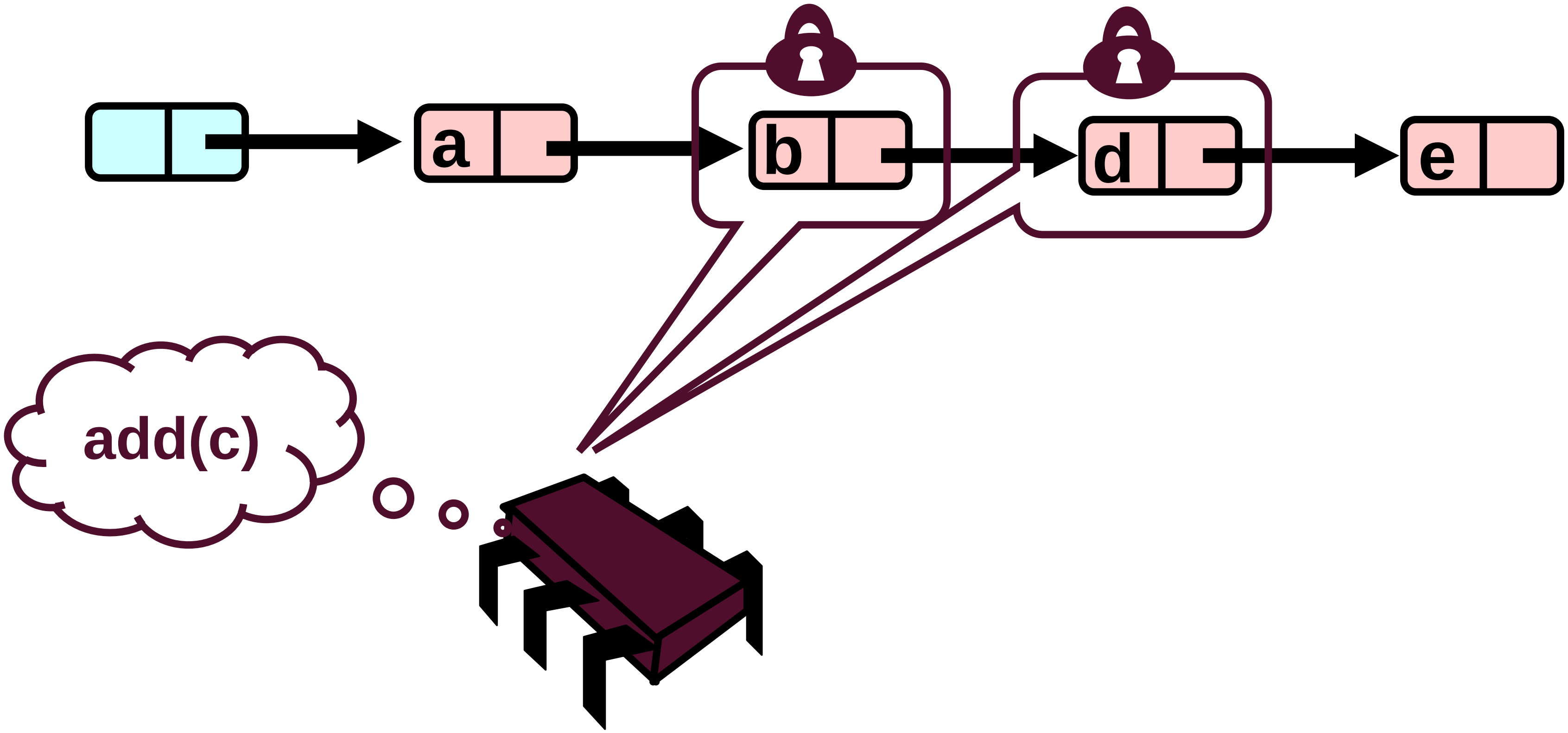
Trick 3: Optimistic synchronization

- **Similar to reader/writer locking but traverse list without locks**
  - Dangerous! Requires additional checks.
- **Harder to proof correct**

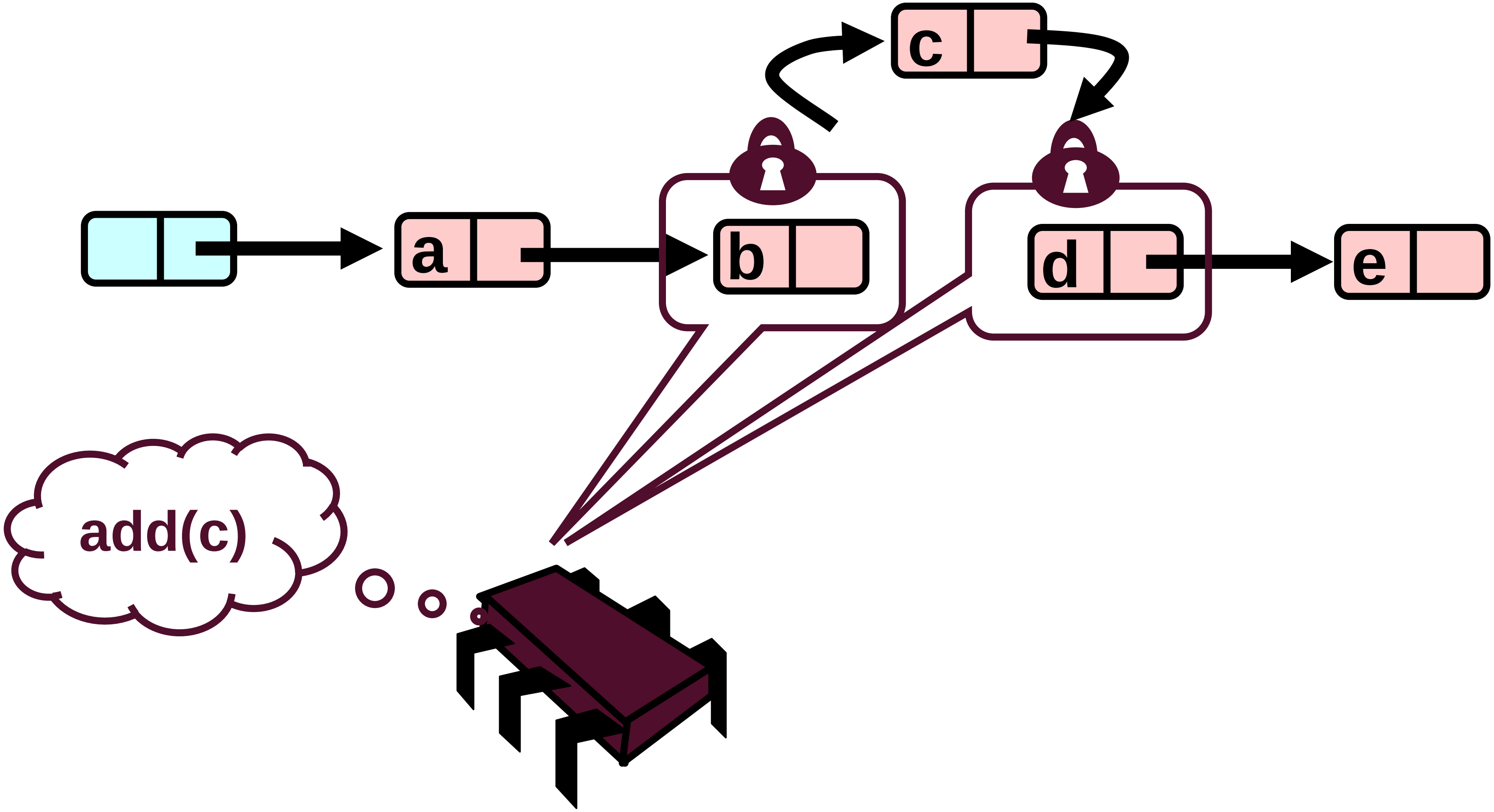
Optimistic: Traverse without Locking



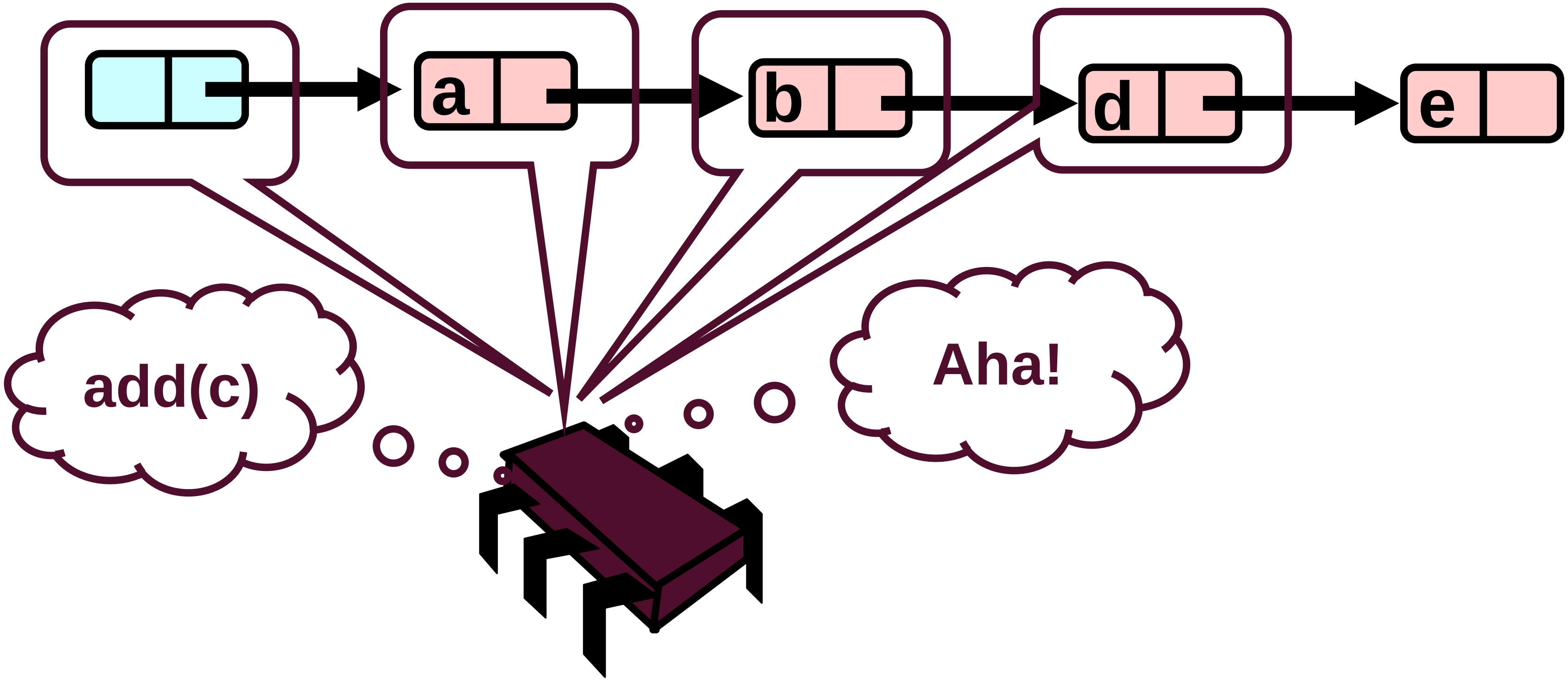
Optimistic: Lock and Load



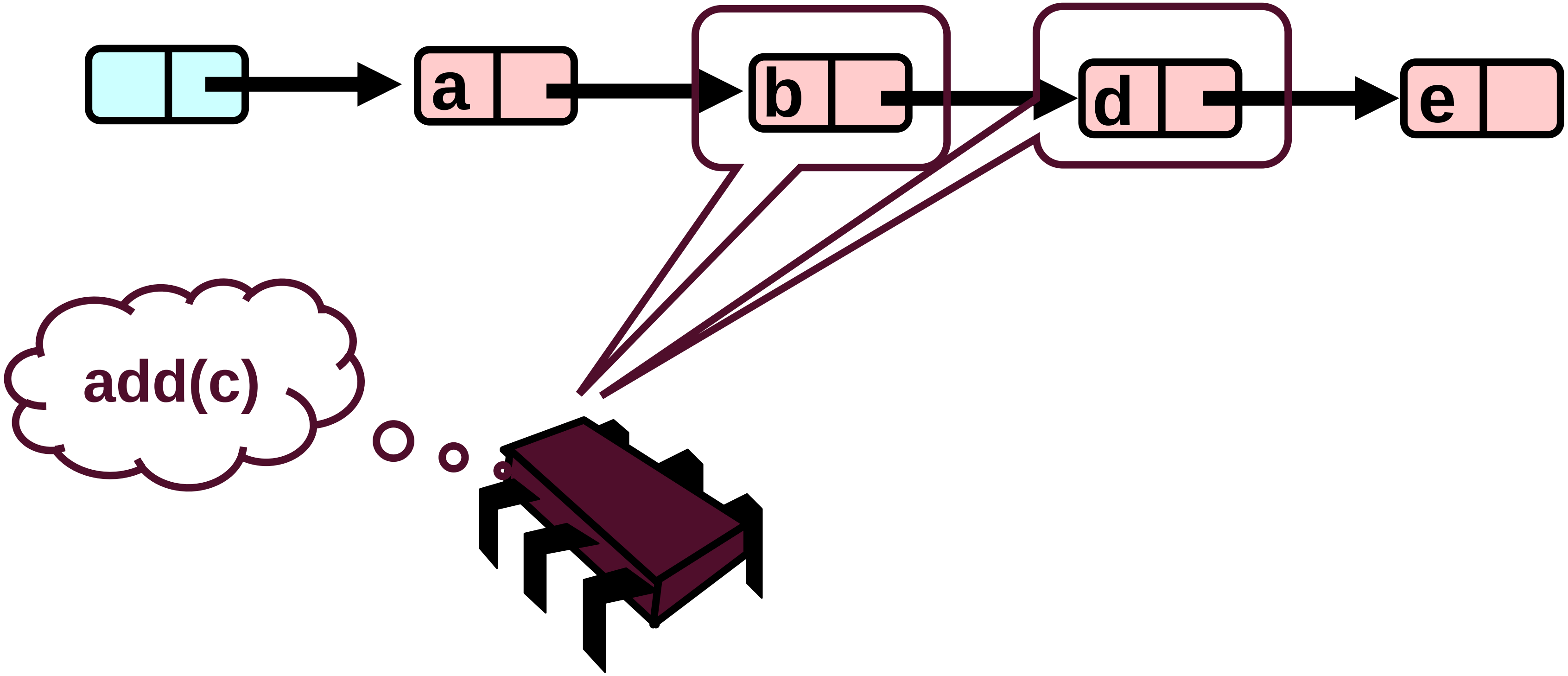
Optimistic: Lock and Load



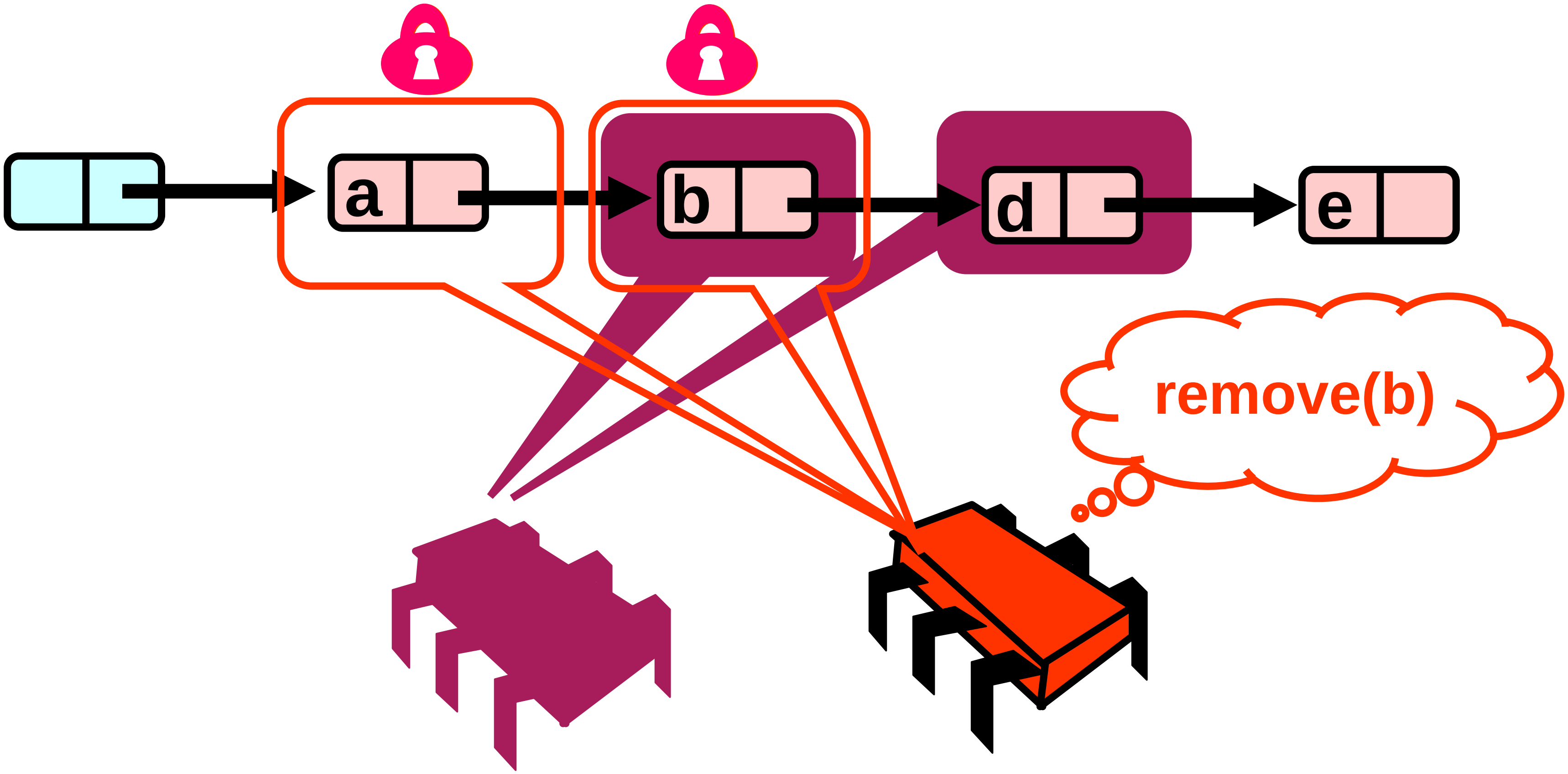
What could go wrong?



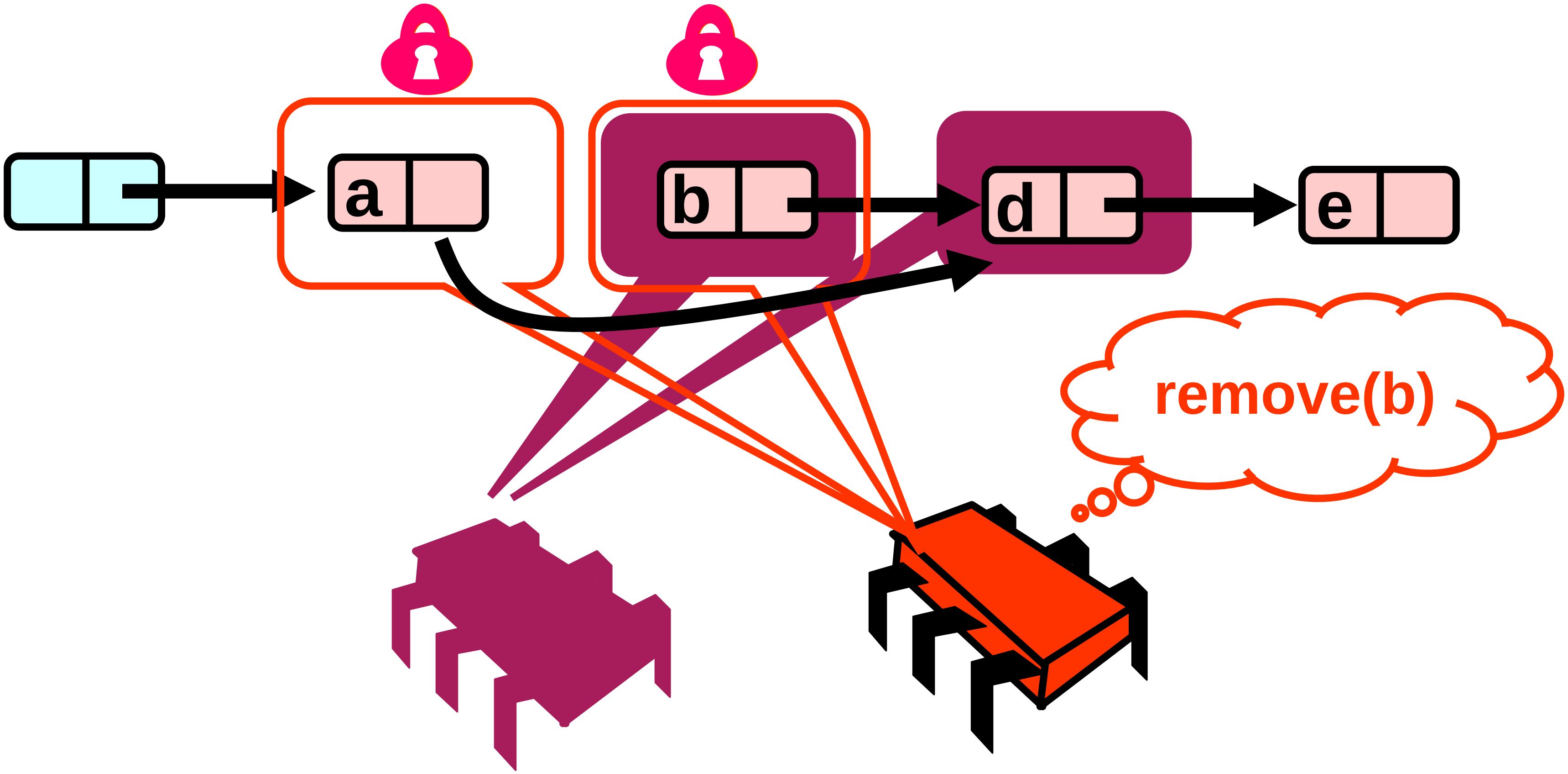
What could go wrong?



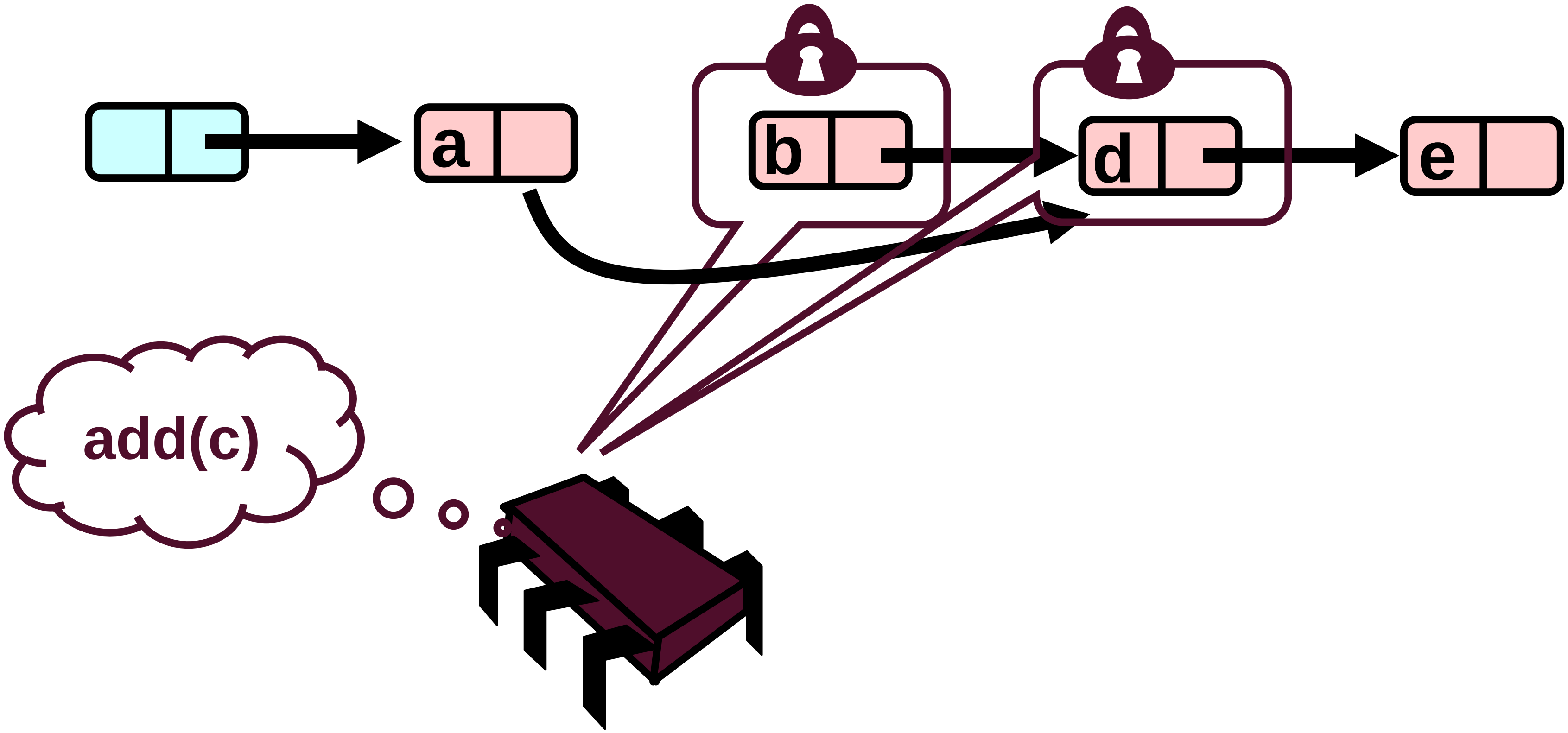
What could go wrong?



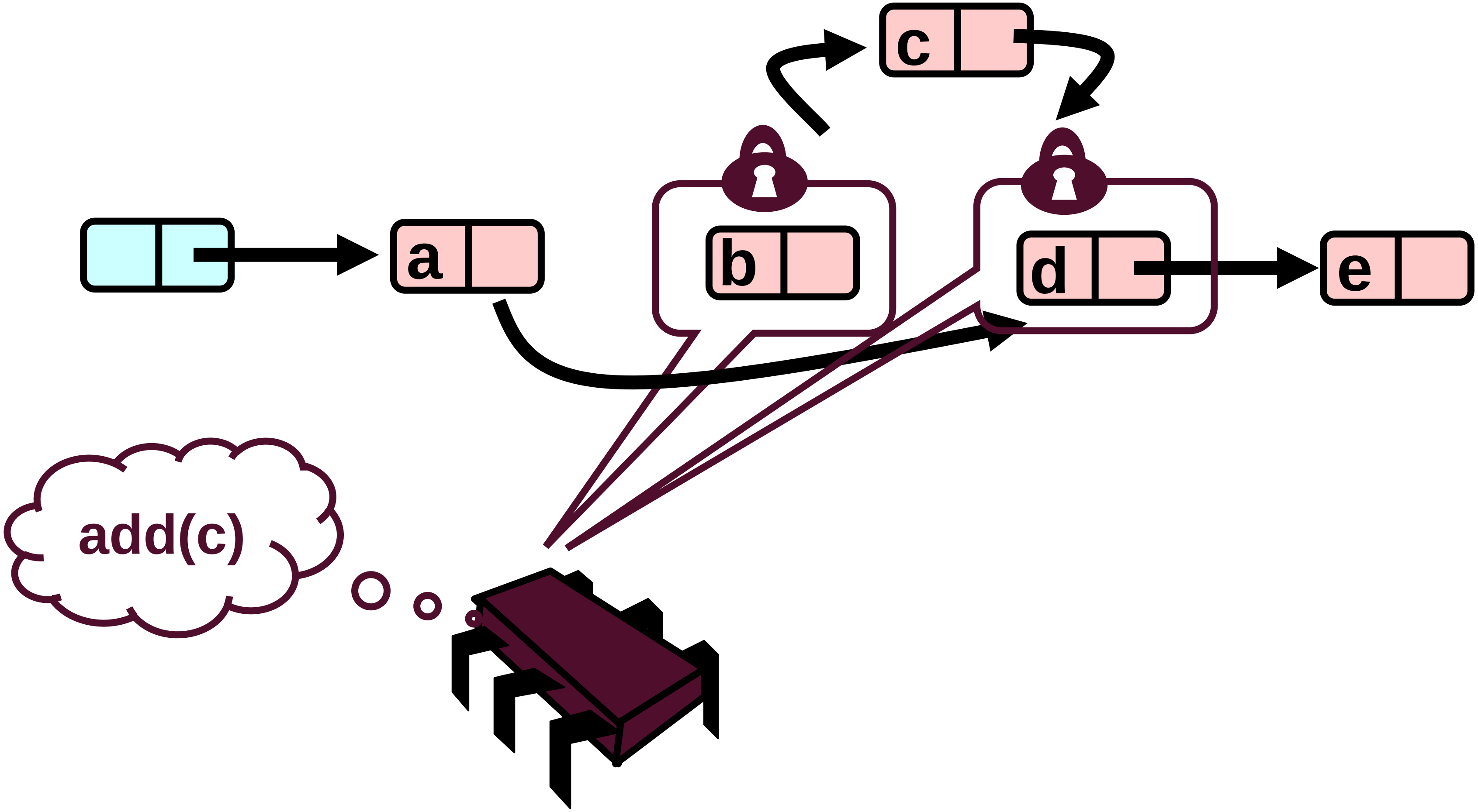
What could go wrong?



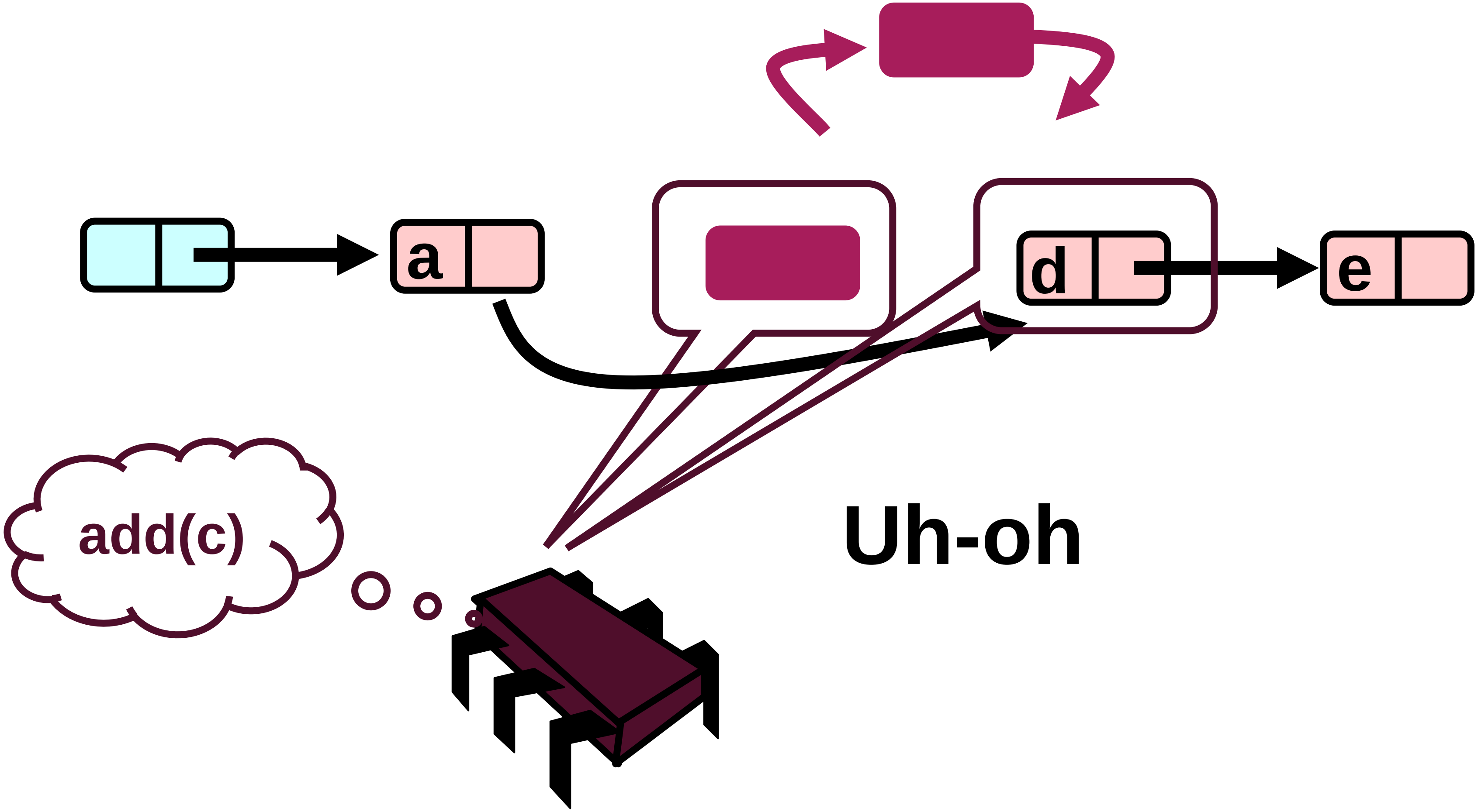
What could go wrong?



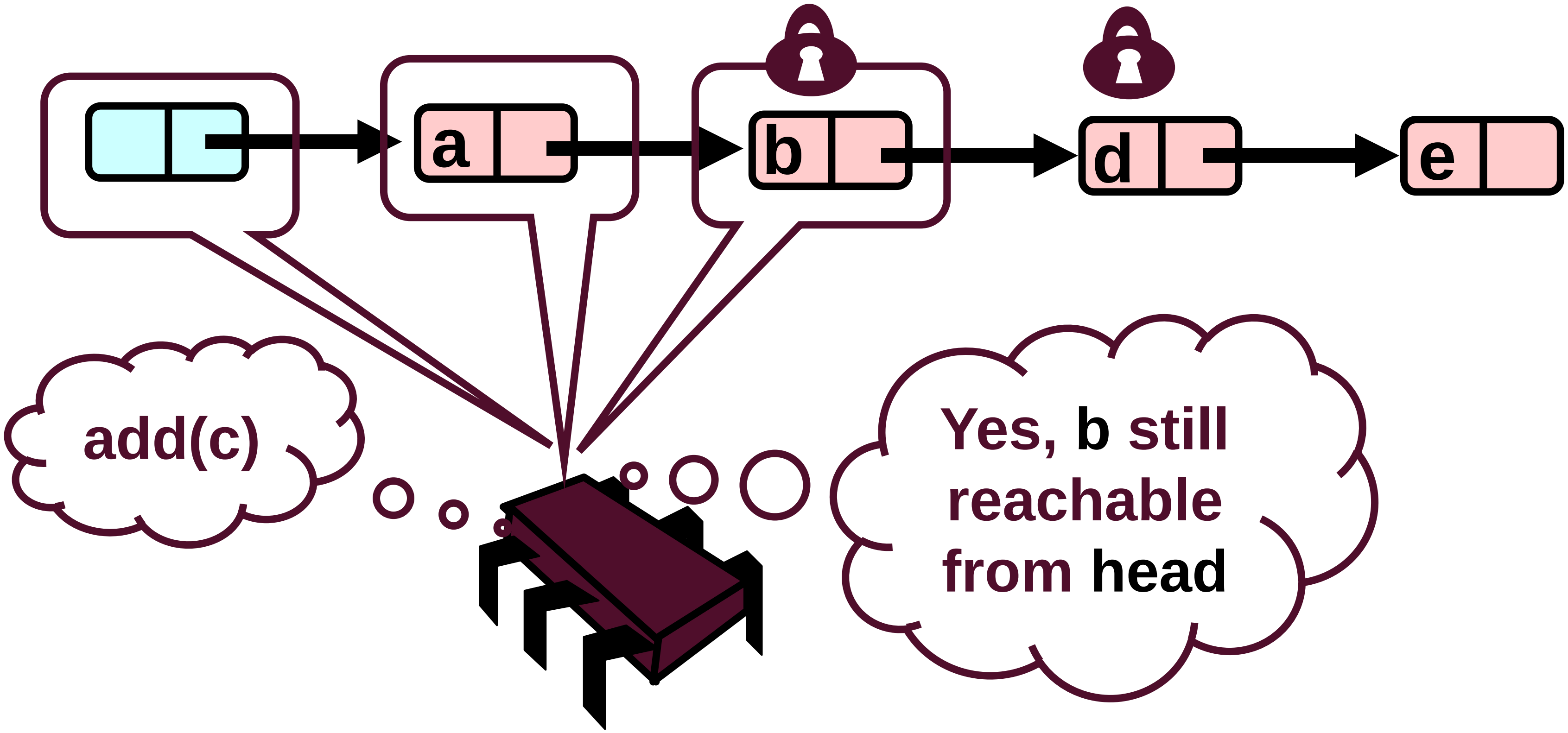
What could go wrong?



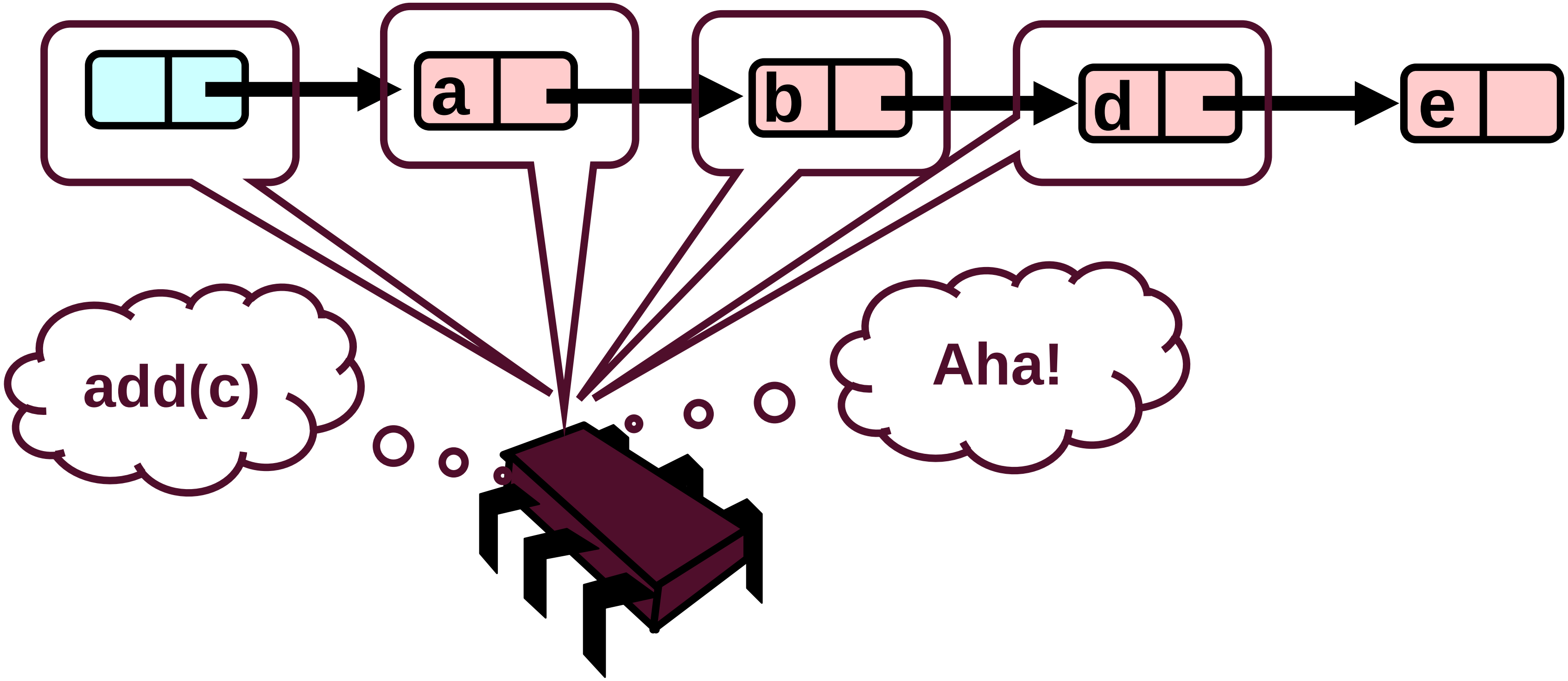
What could go wrong?



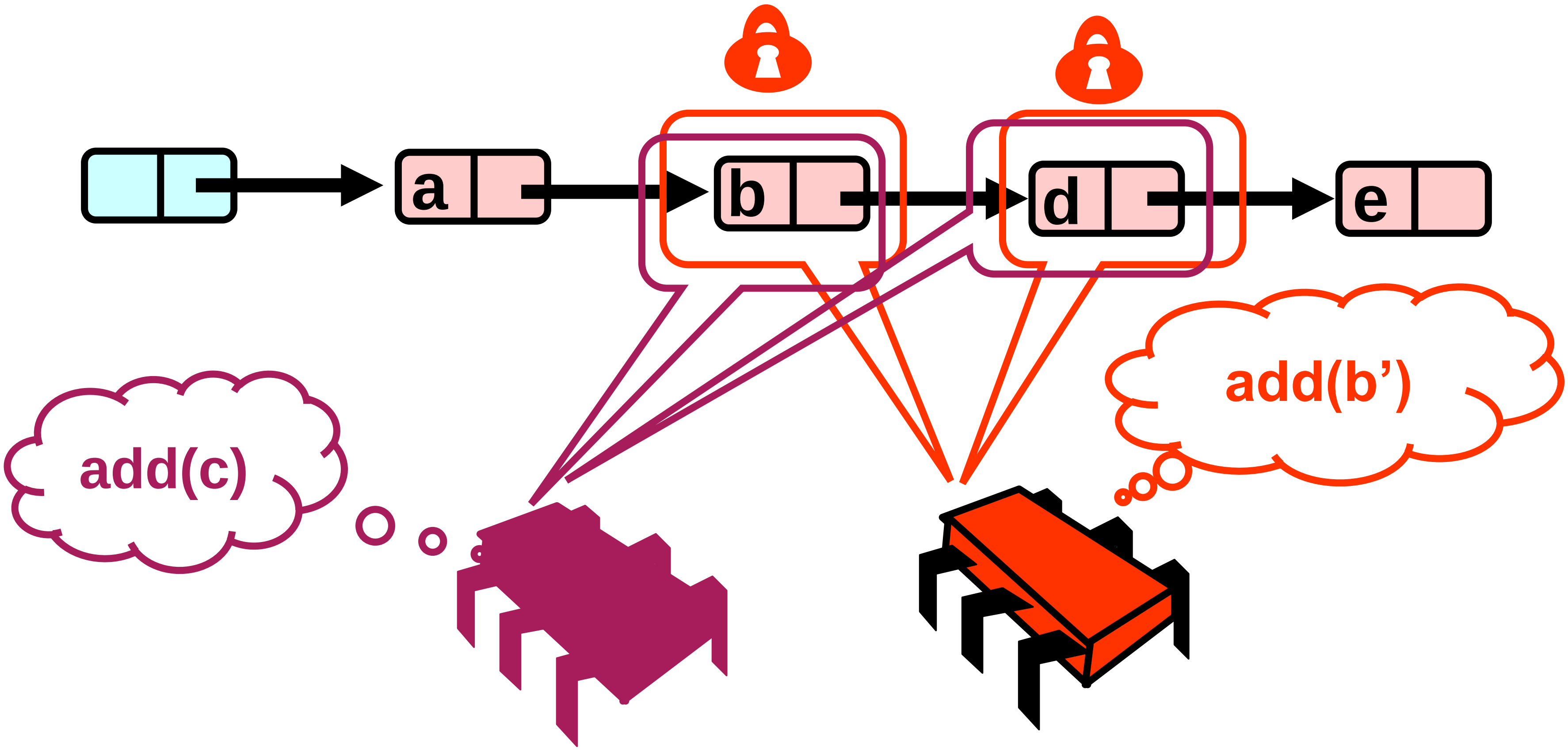
Validate – Part 1



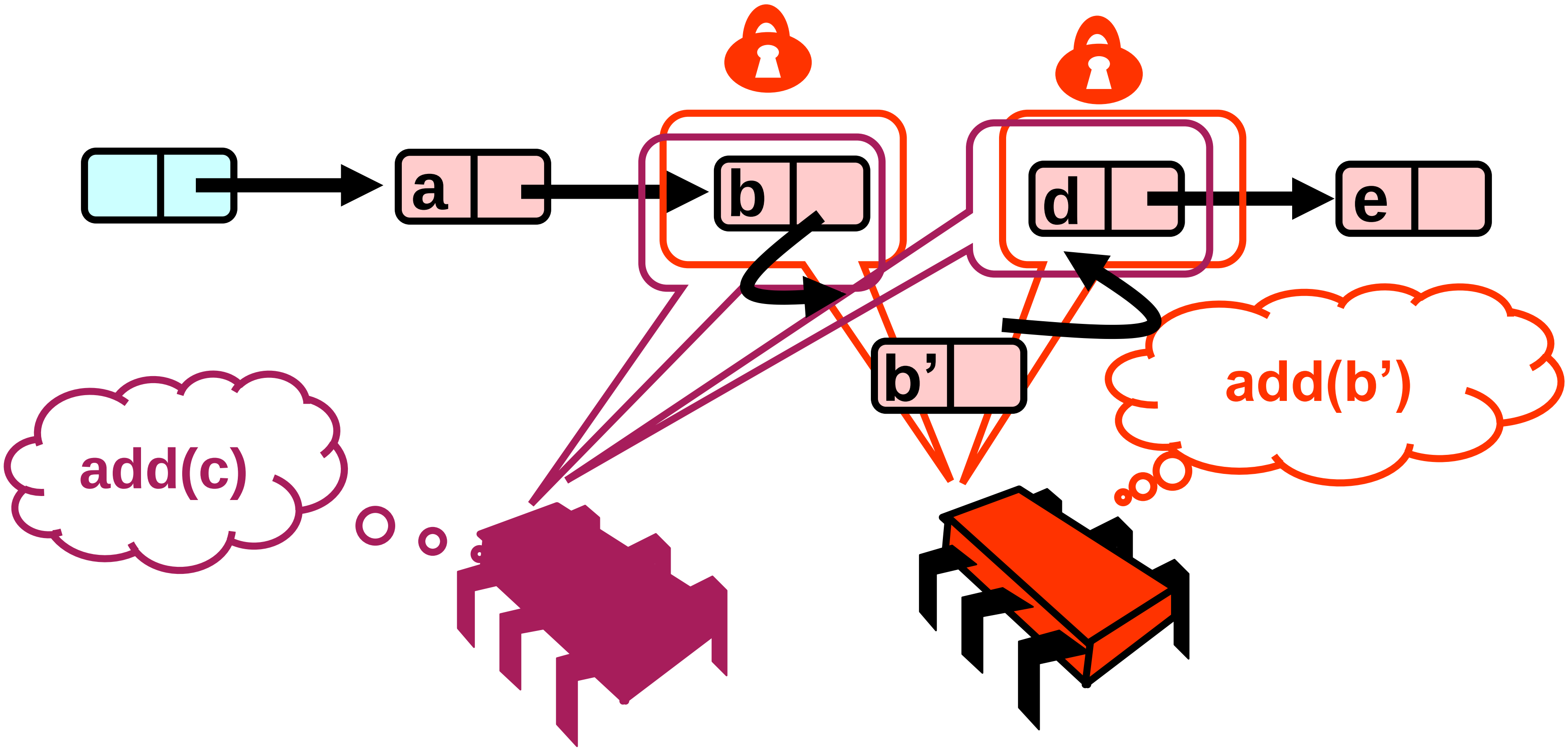
What Else Could Go Wrong?



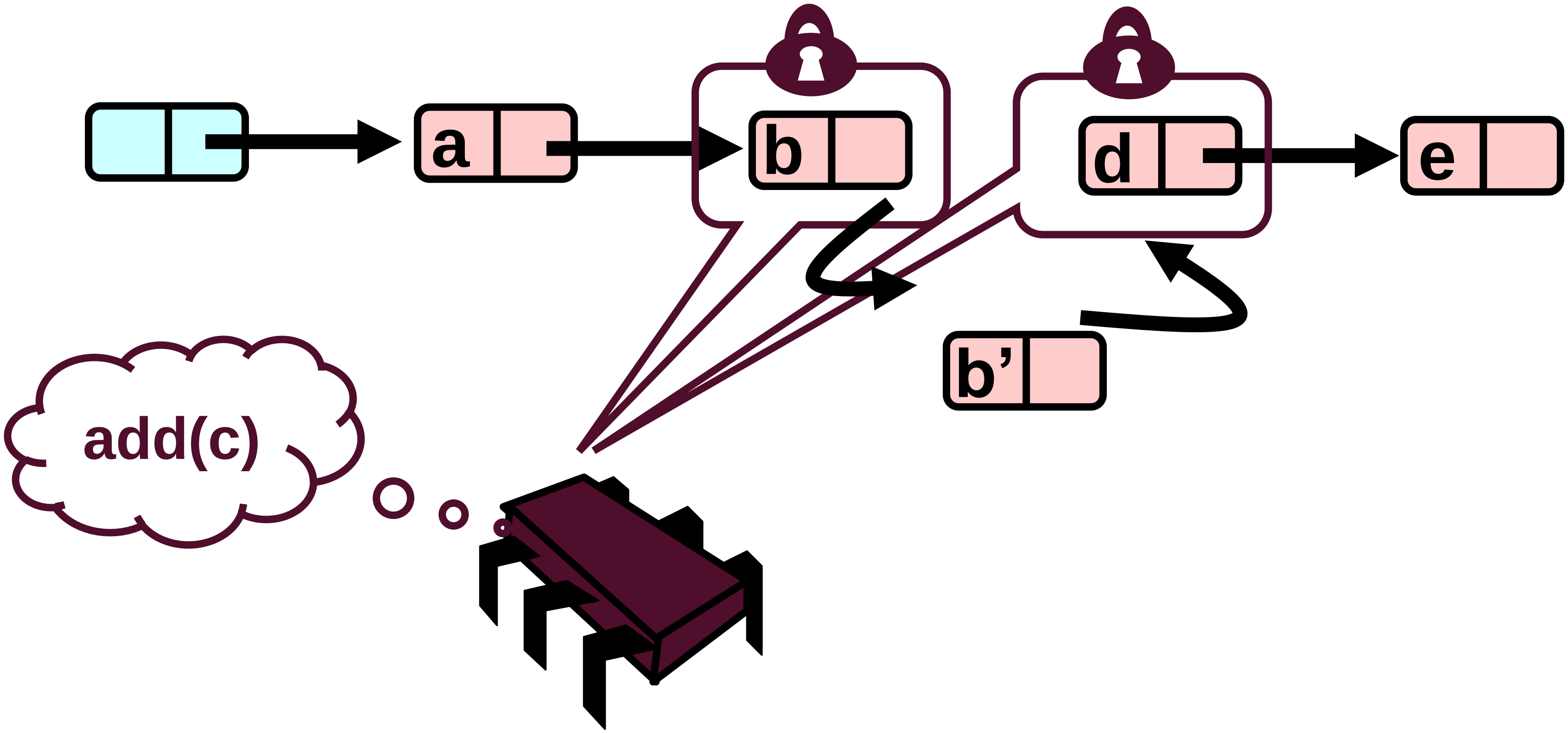
What Else Could Go Wrong?



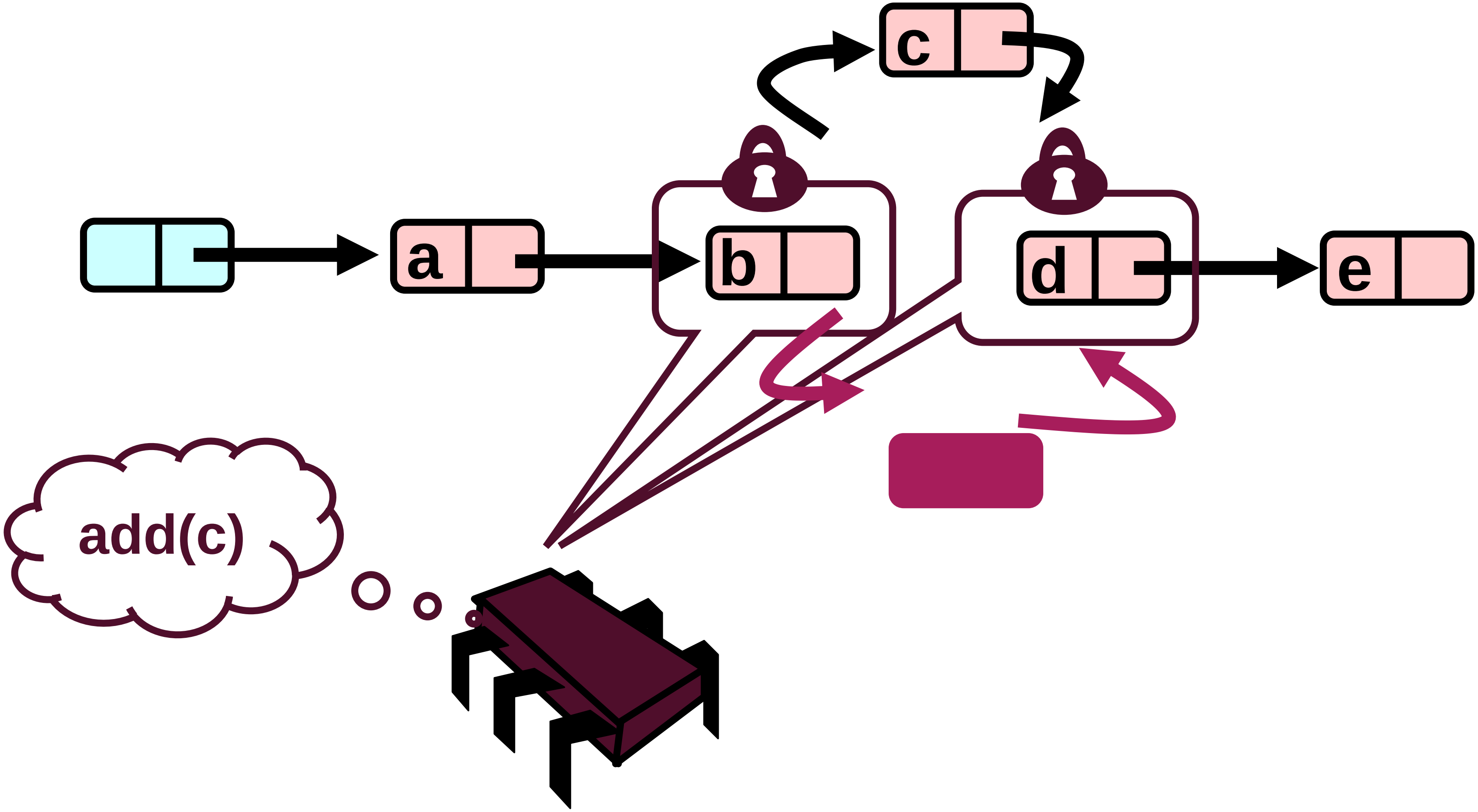
What Else Could Go Wrong?



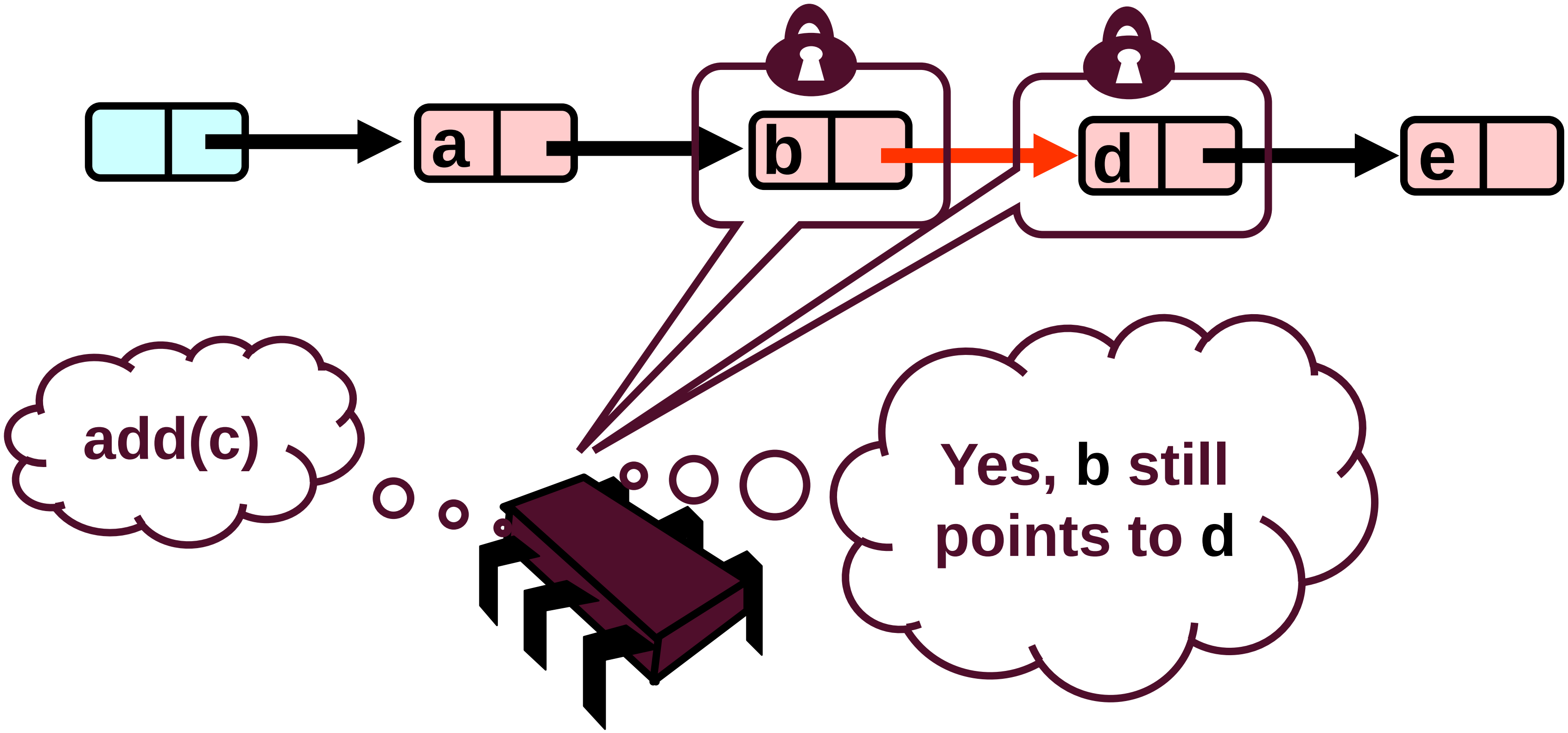
What Else Could Go Wrong?



What Else Could Go Wrong?



Validate Part 2 (while holding locks)



Yes, b still points to d

Optimistic synchronization

- **One MUST validate AFTER locking**
  1. Check if the path how we got there is still valid!
  2. Check if locked nodes are still connected
  - If any of those checks fail?

    Start over from the beginning (hopefully rare)

- **Not starvation-free**
  - A thread may need to abort forever if nodes are added/removed
  - Should be rare in practice!

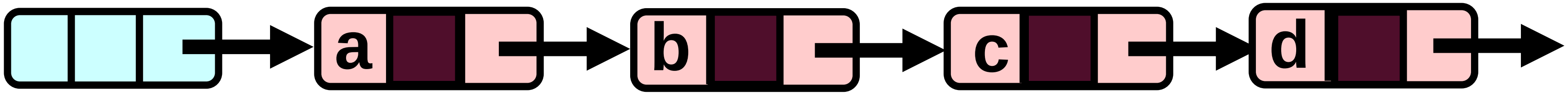
- **Other disadvantages?**
  - All operations requires two traversals of the list!
  - Even contains() needs to check if node is still in the list!

Trick 4: Lazy synchronization

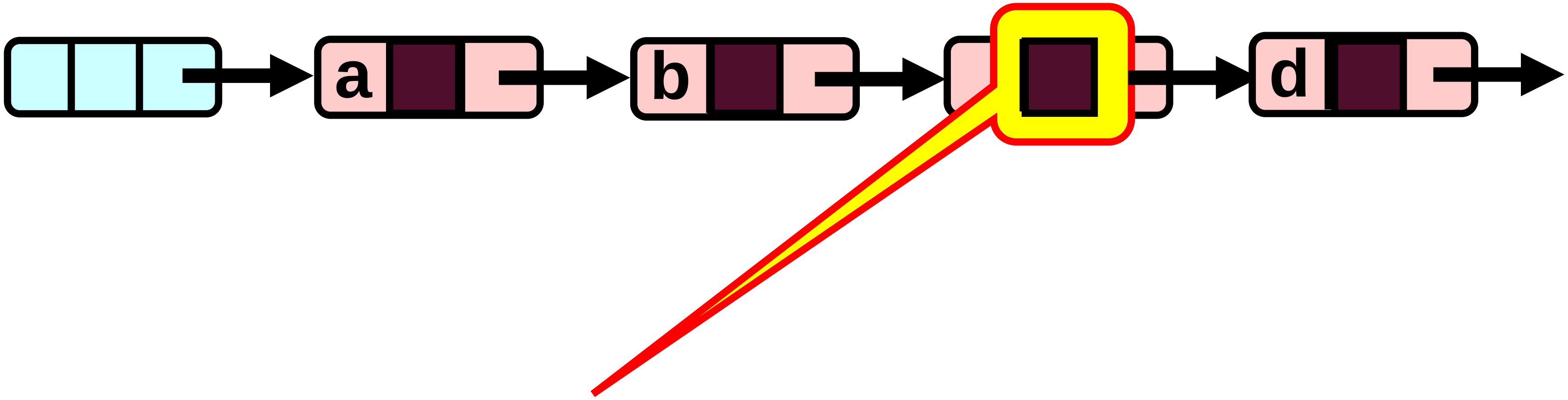
- **We really want one list traversal**
- **Also, contains() should be wait-free**
  - Is probably the most-used operation
- **Lazy locking is similar to optimistic**
  - Key insight: removing is problematic
  - Perform it “lazily”
- **Add a new “valid” field**
  - Indicates if node is still in the set
  - Can remove it without changing list structure!
  - Scan once, contains() never locks!

```
typedef struct {
  int key;
  node *next;
  lock_t lock;
  boolean valid;
} node;
```

Lazy Removal

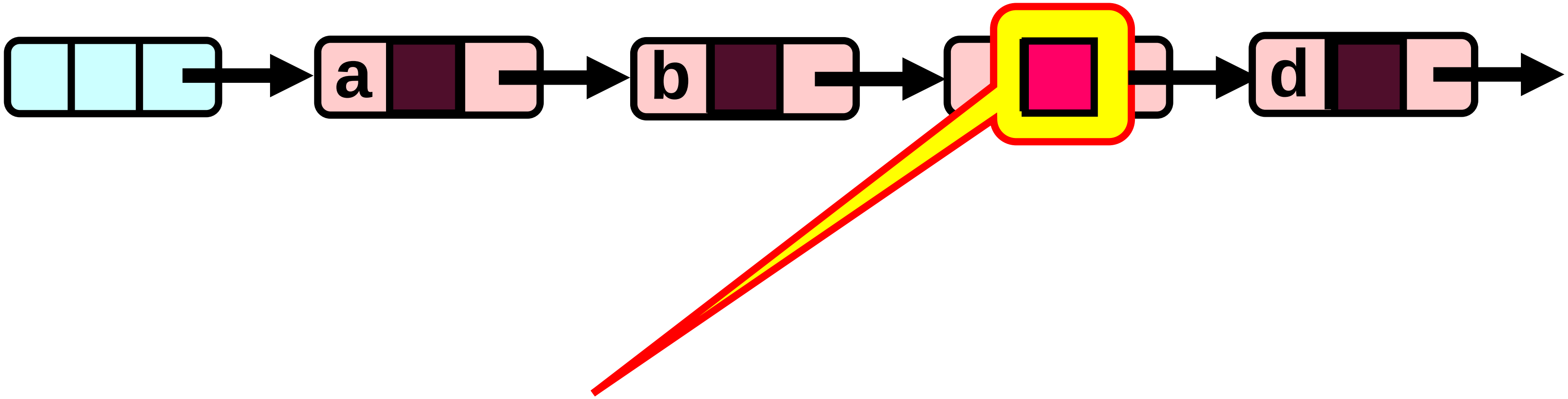


Lazy Removal



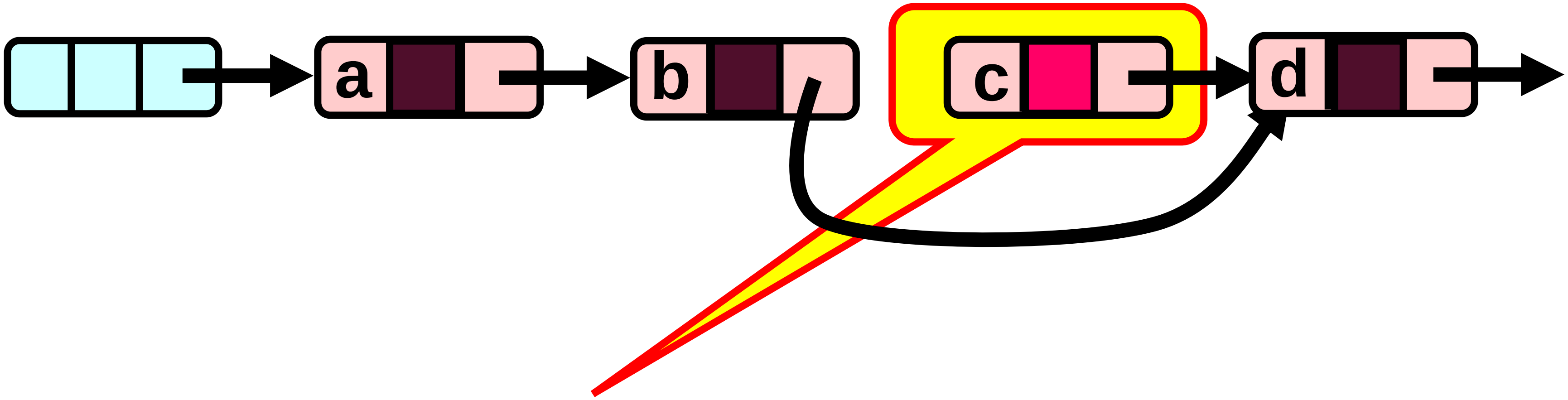
Present in list

Lazy Removal



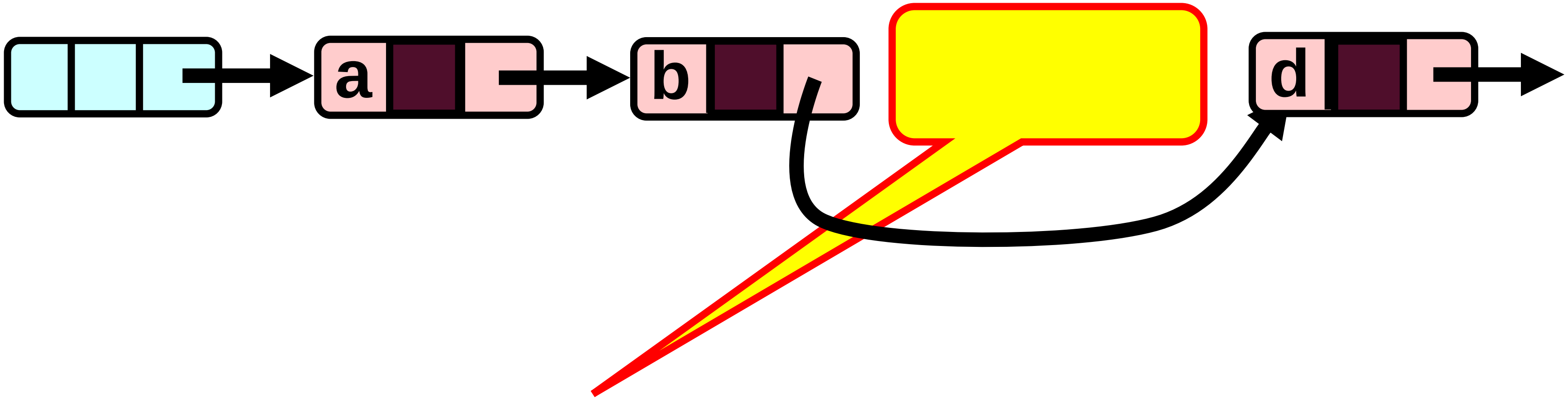
Logically deleted

Lazy Removal



Physically deleted

Lazy Removal



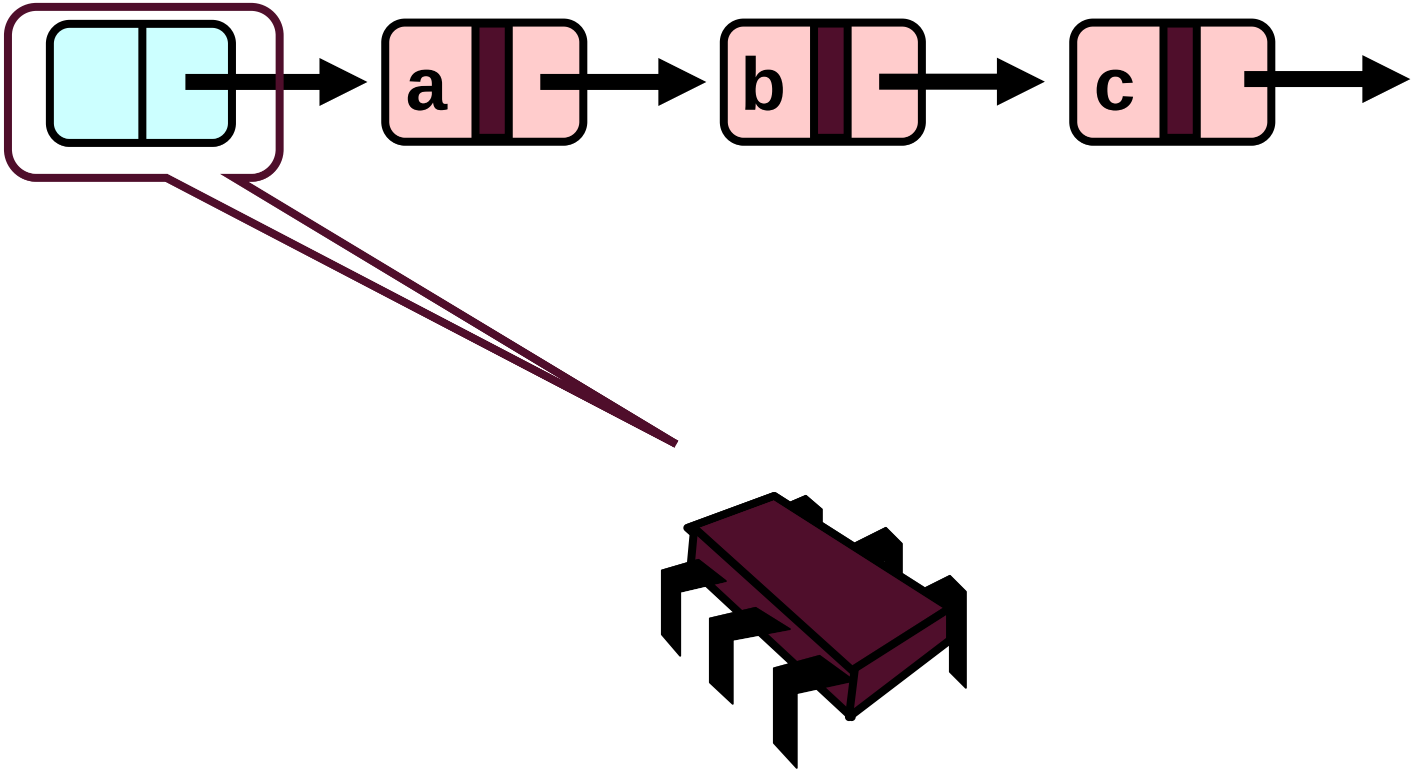
Physically deleted

How does it work?

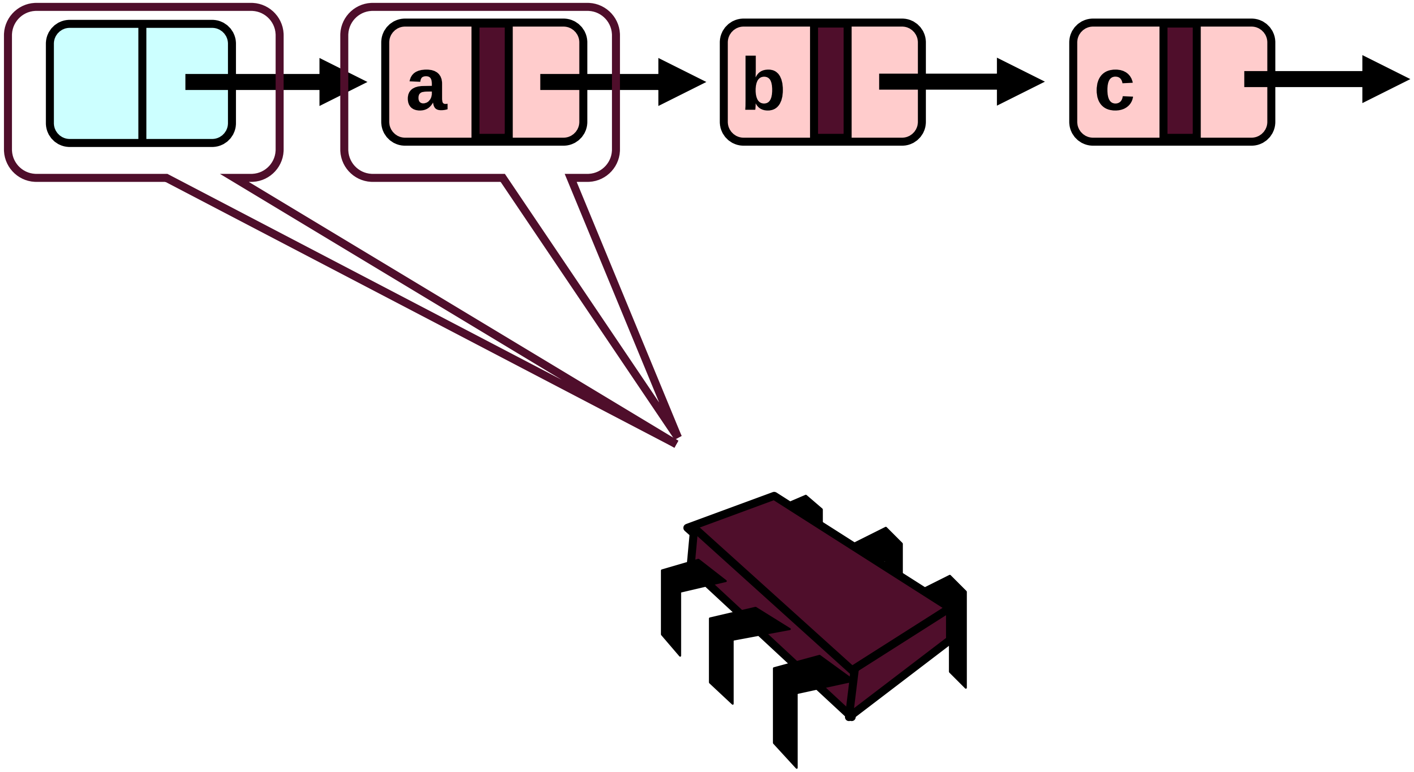
- **Eliminates need to re-scan list for reachability**
  - Maintains invariant that every **unmarked** node is reachable!
- **Contains can now simply traverse the list**
  - Just check marks, not reachability, no locks
- **Remove/Add**
  - Scan through locked and marked nodes
  - Removing does not delay others
  - Must only lock when list structure is updated

    Check if neither pred nor curr are marked, pred.next == curr

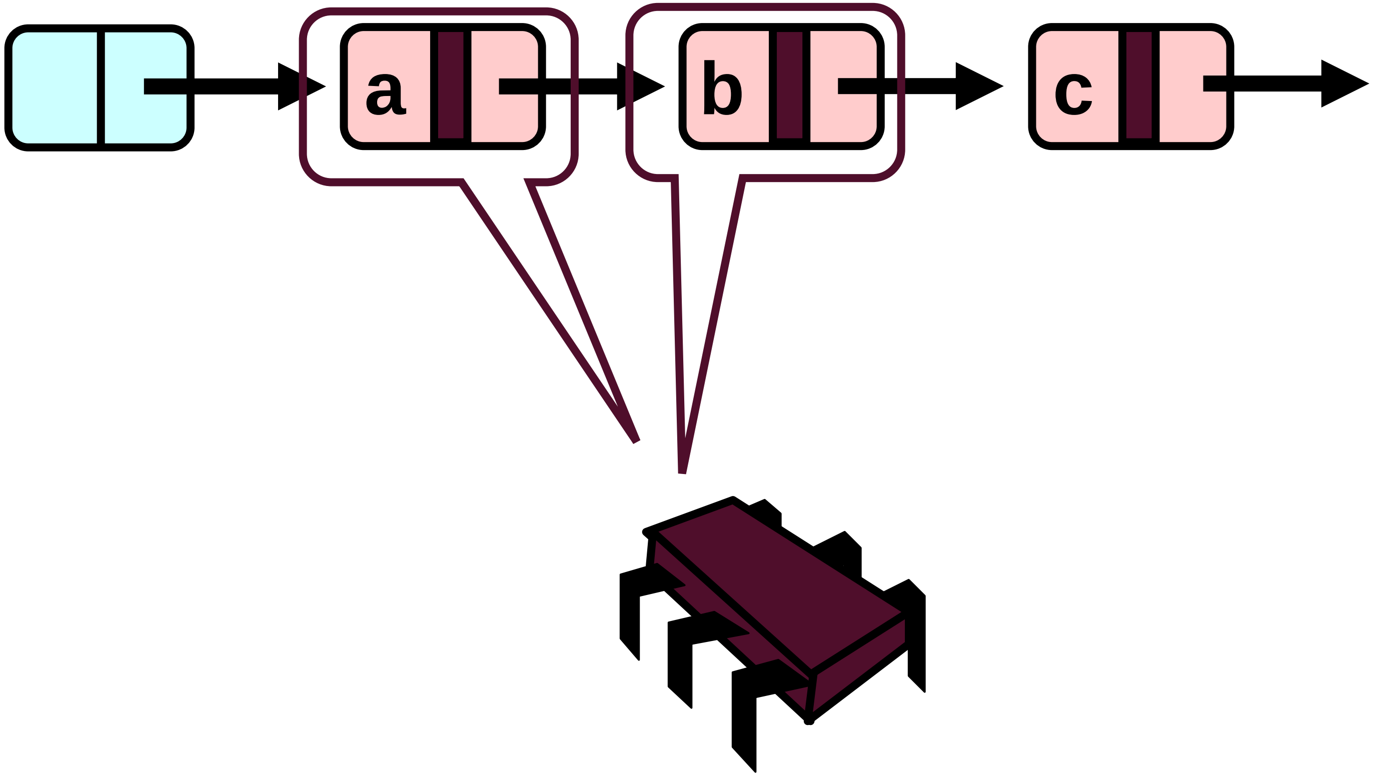
Business as Usual



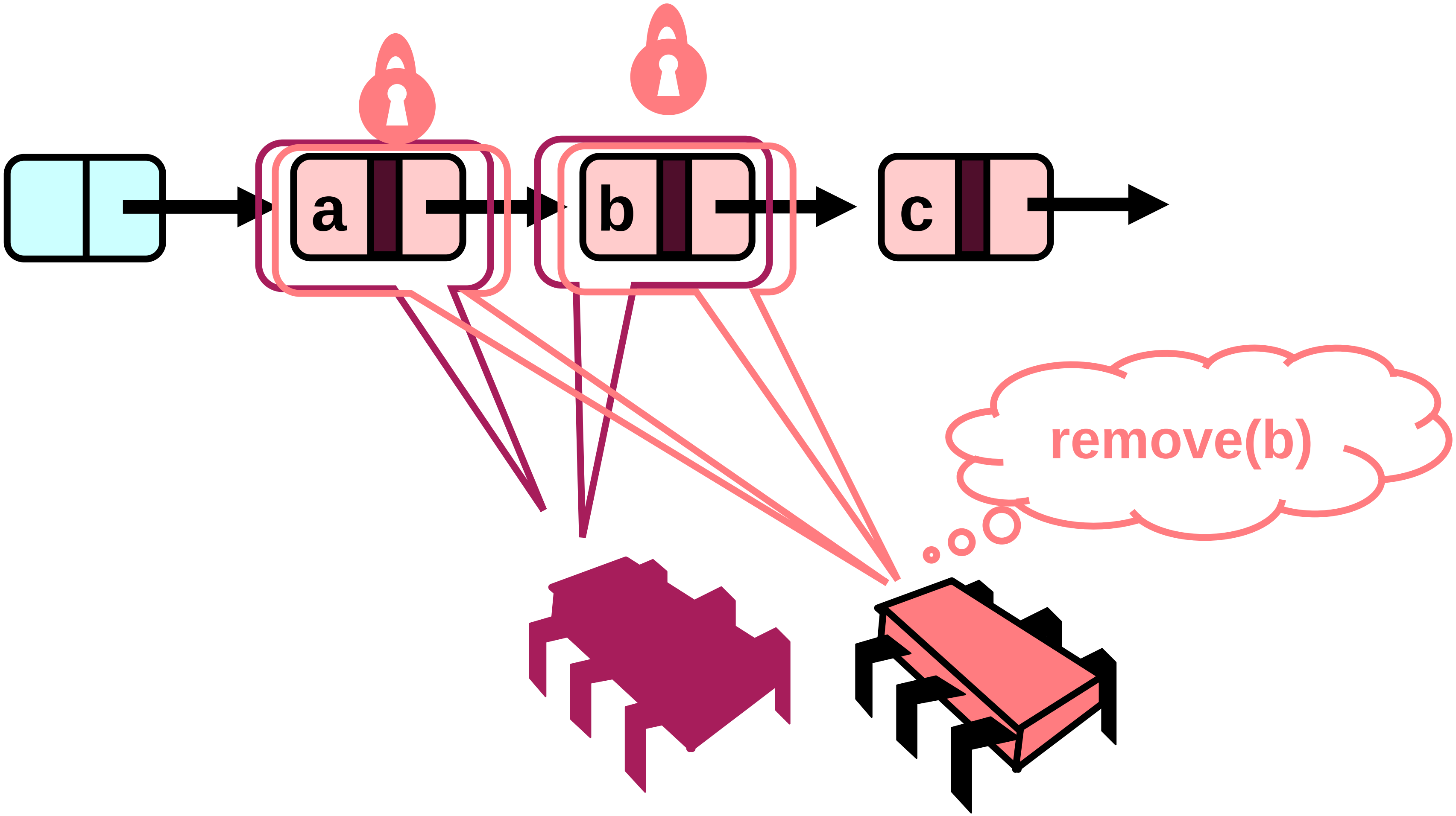
Business as Usual



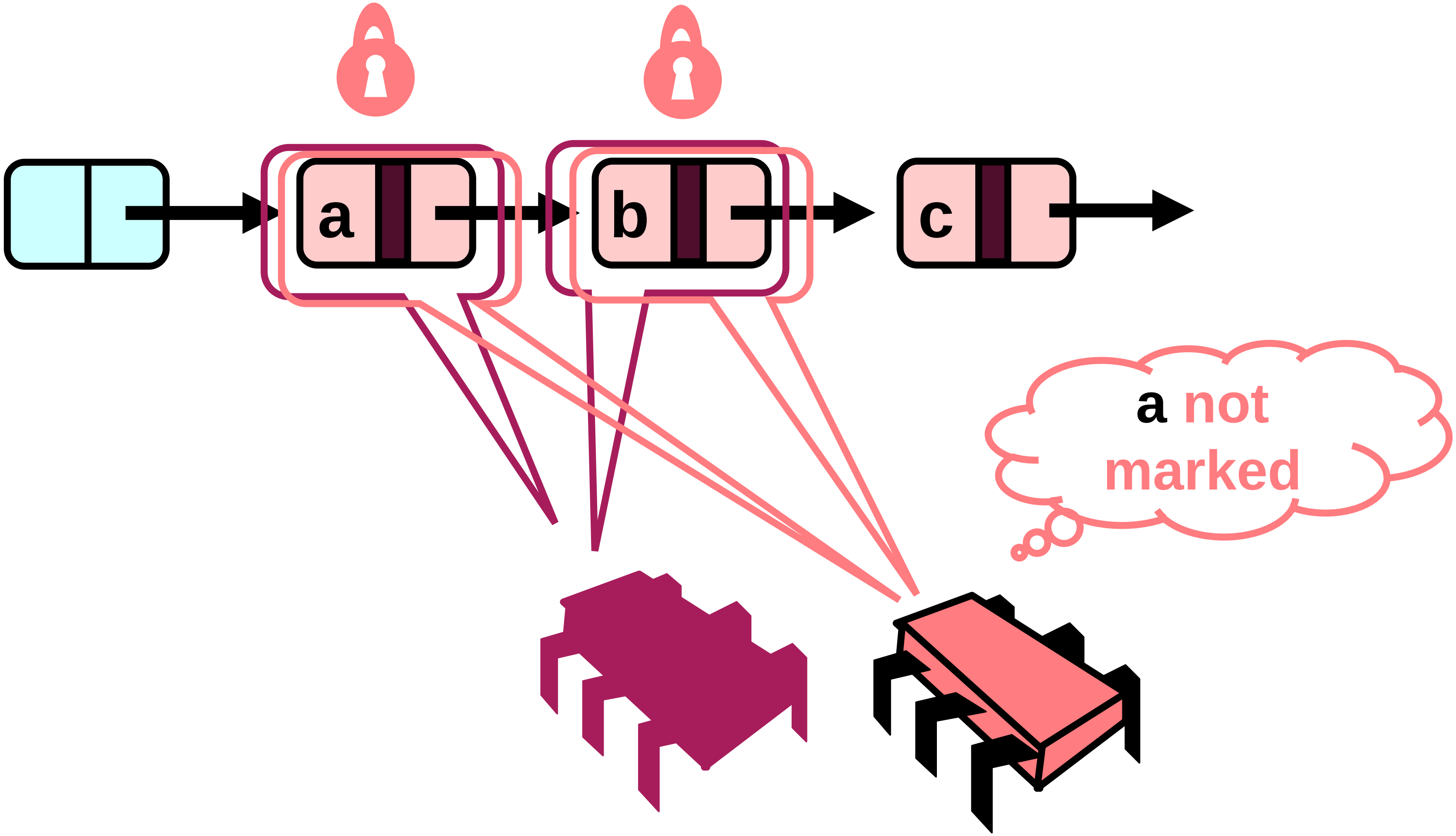
Business as Usual



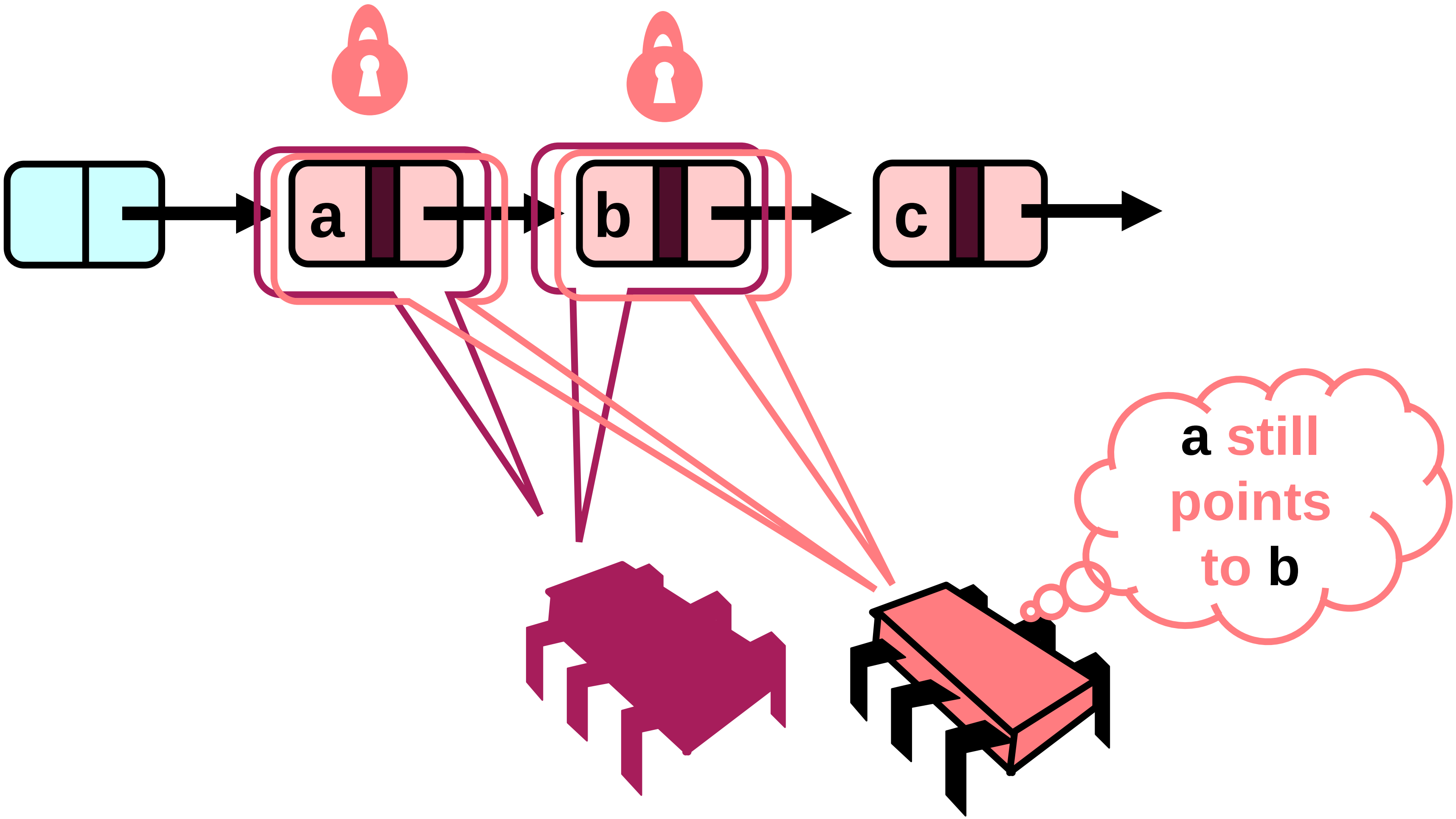
Business as Usual



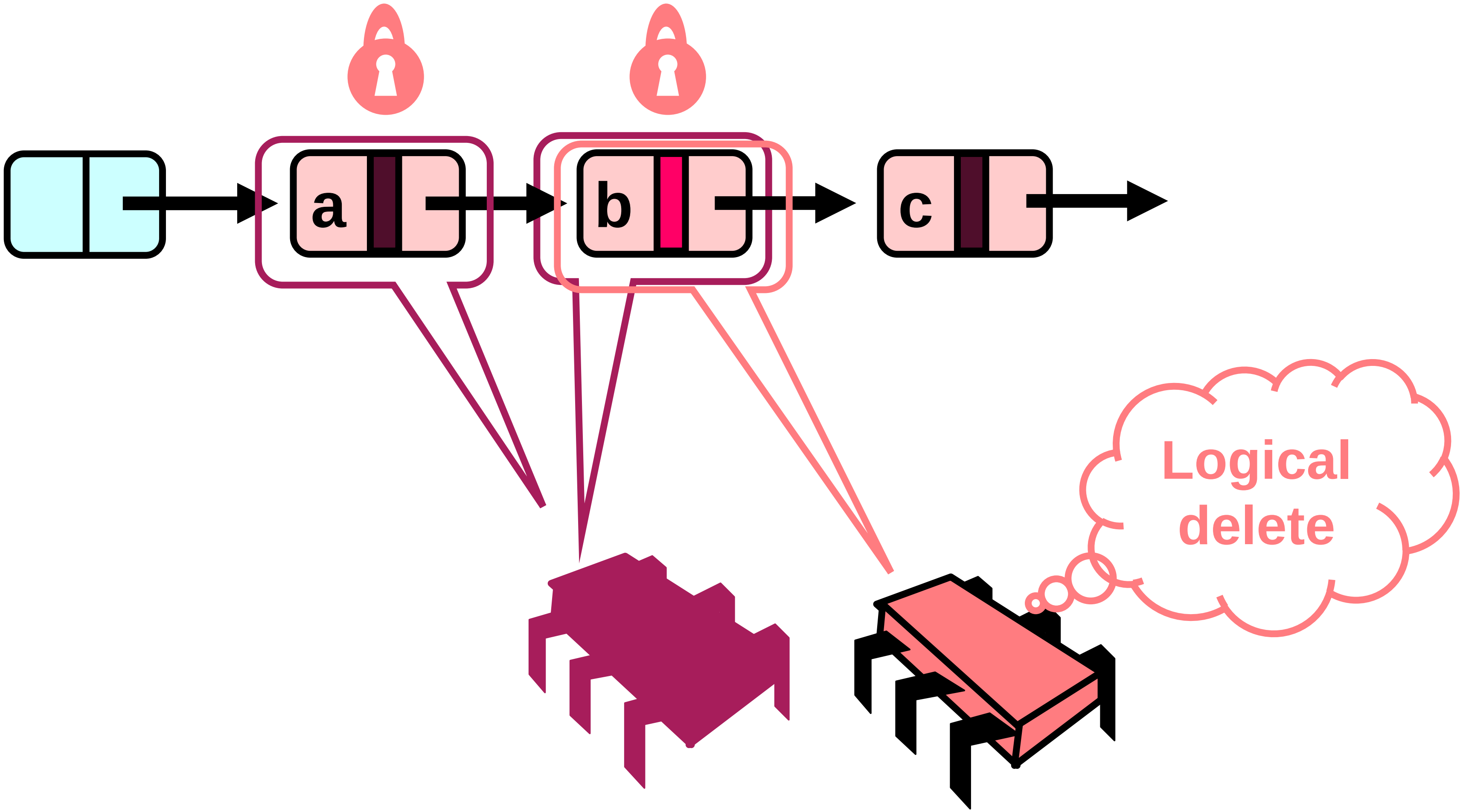
Business as Usual



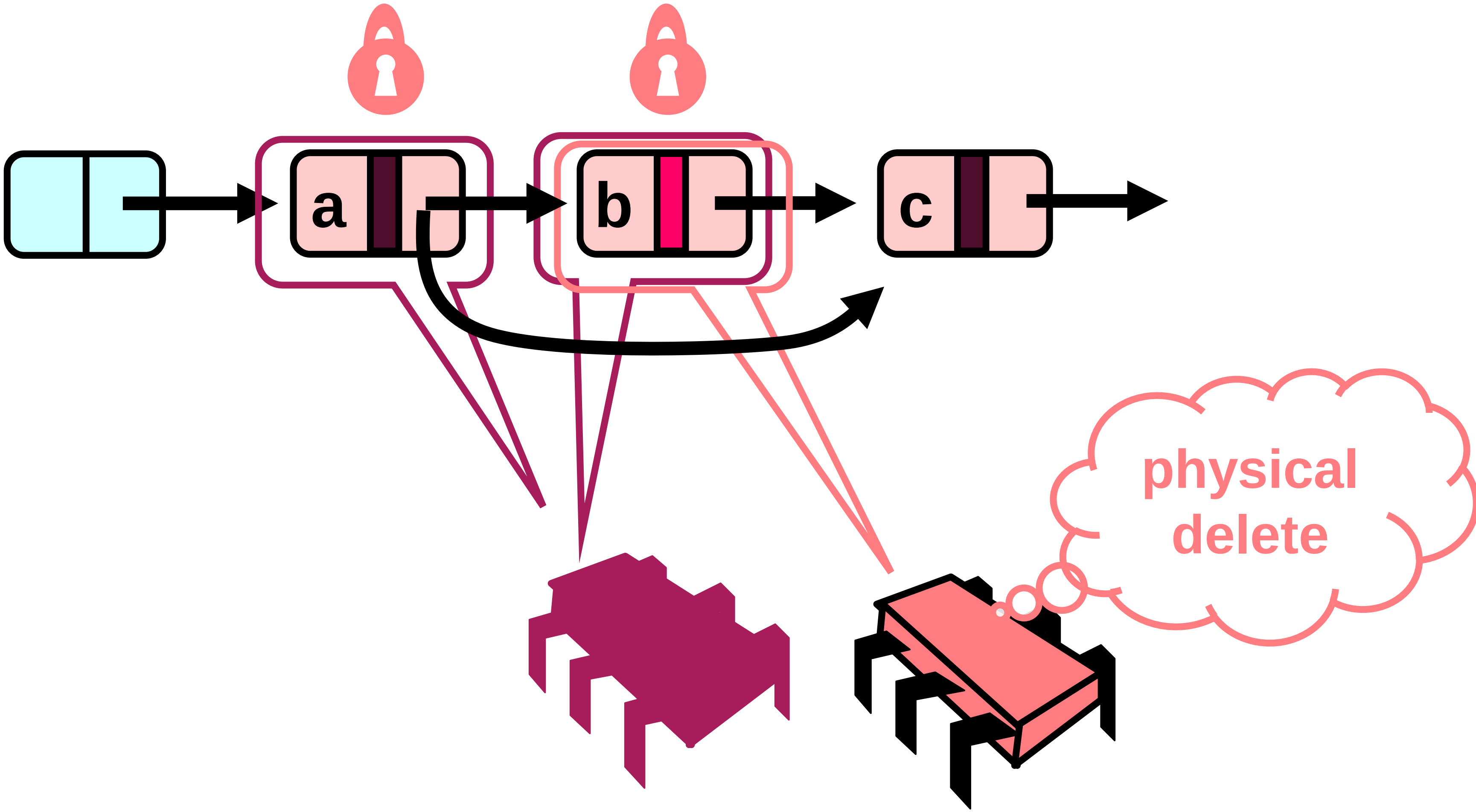
Business as Usual



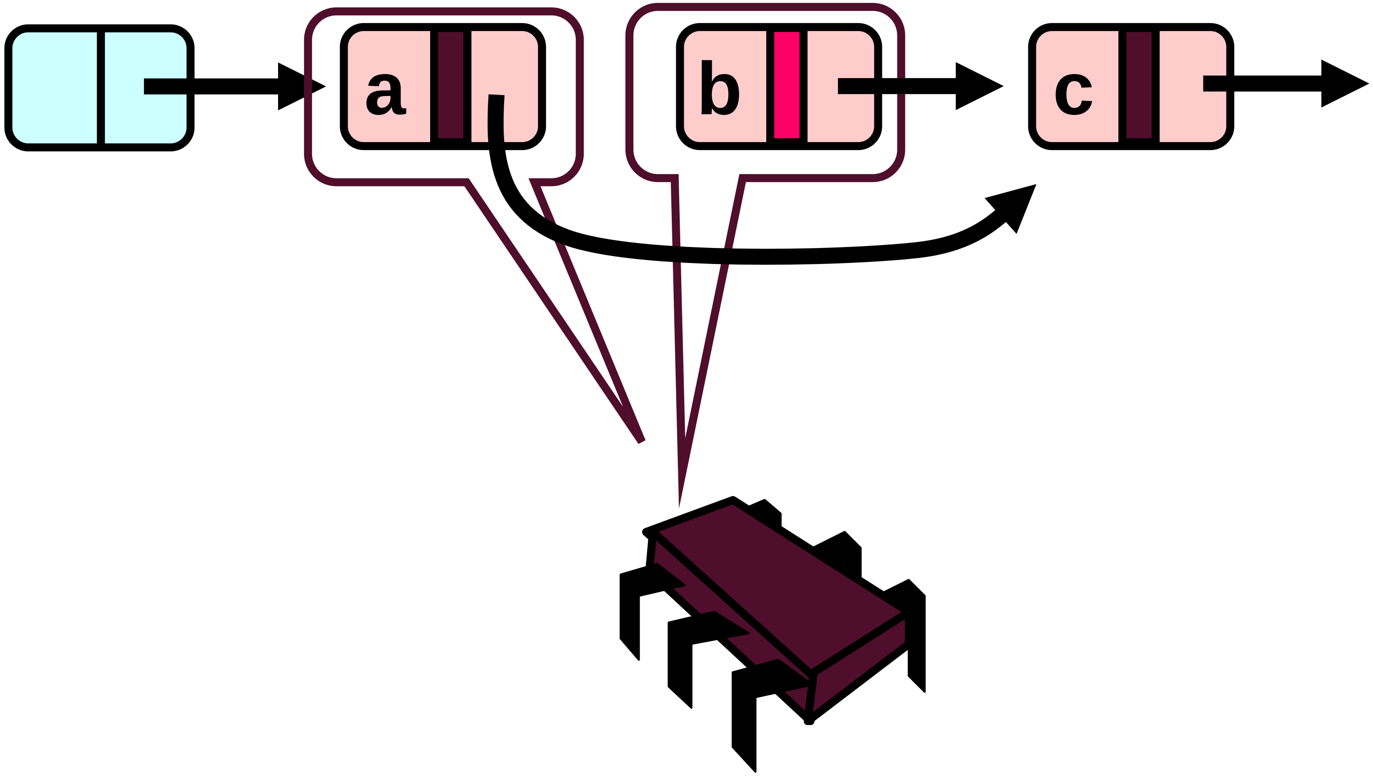
Business as Usual



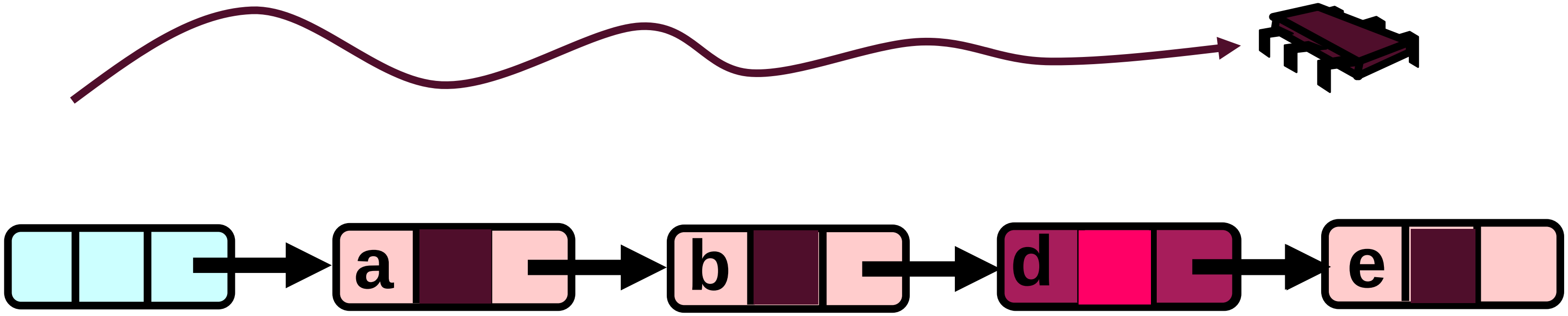
Business as Usual



Business as Usual



Summary: Wait-free Contains



Use Mark bit + list ordering

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

Lazy add() and remove() + Wait-free contains()

Problems with Locks

- **What are the fundamental problems with locks?**
- **Blocking**
  - Threads wait, fault tolerance
  - Especially when things like page faults occur in CR
- **Overheads**
  - Even when not contended
  - Also memory/state overhead
- **Synchronization is tricky**
  - Deadlock, other effects are hard to debug
- **Not easily composable**

Lock-free Methods

- **No matter what:**
  - Guarantee minimal progress

    I.e., some thread will advance
  - Threads may halt at bad times (no CRs! No exclusion!)

    I.e., cannot use locks!
  - Needs other forms of synchronization

    E.g., atomics (discussed before for the implementation of locks)

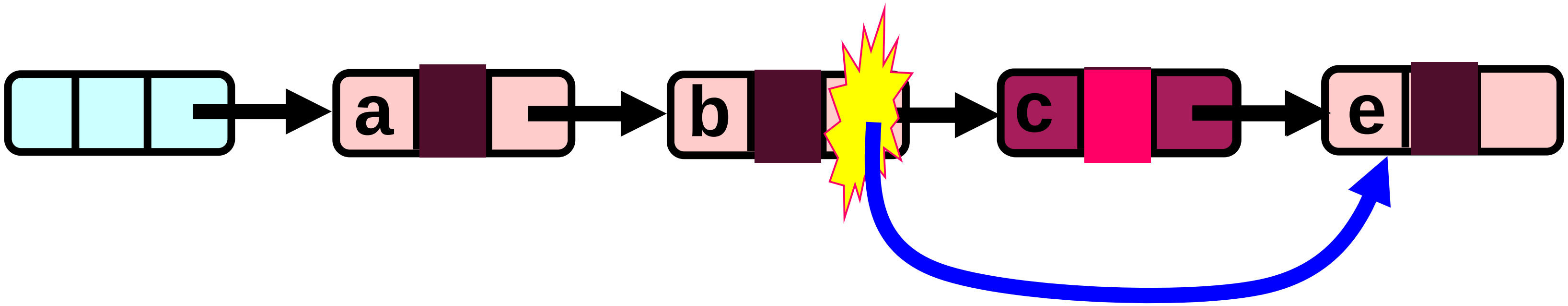
    Techniques are astonishingly similar to guaranteeing mutual exclusion

Trick 5: No Locking

- **Make list lock-free**
- **Logical succession**
  - We have wait-free contains
  - Make add() and remove() lock-free!

    Keep logical vs. physical removal
- **Simple idea:**
  - Use CAS to verify that pointer is correct before moving it

Lock-free Lists



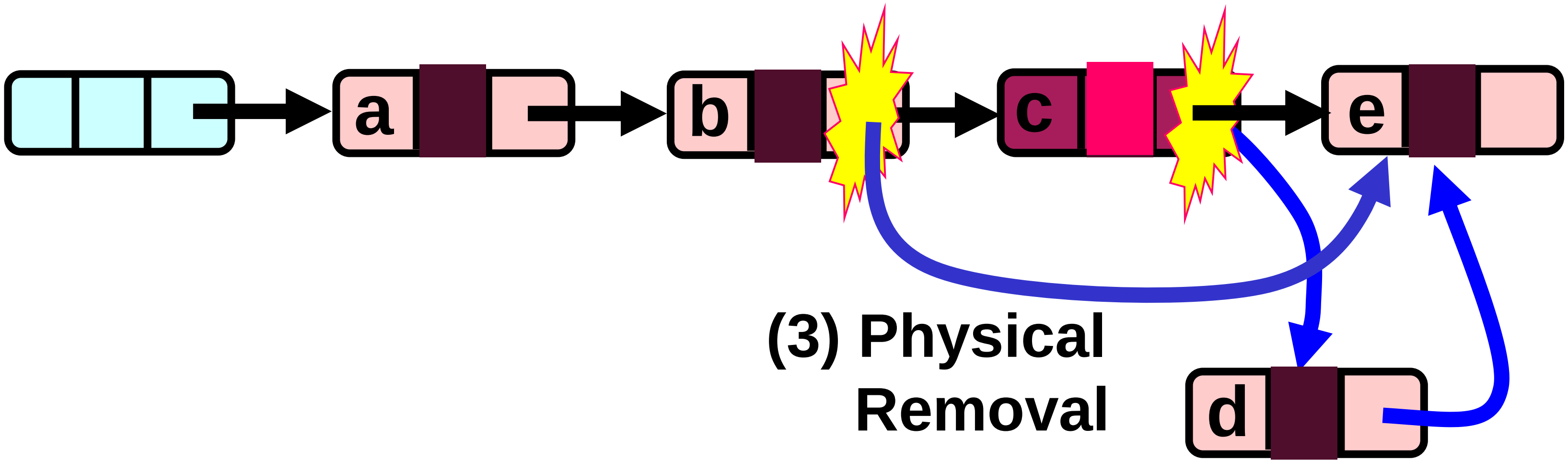
(1) Logical Removal

Use CAS to verify pointer is correct

(2) Physical Removal

Not enough! Why?

Problem…

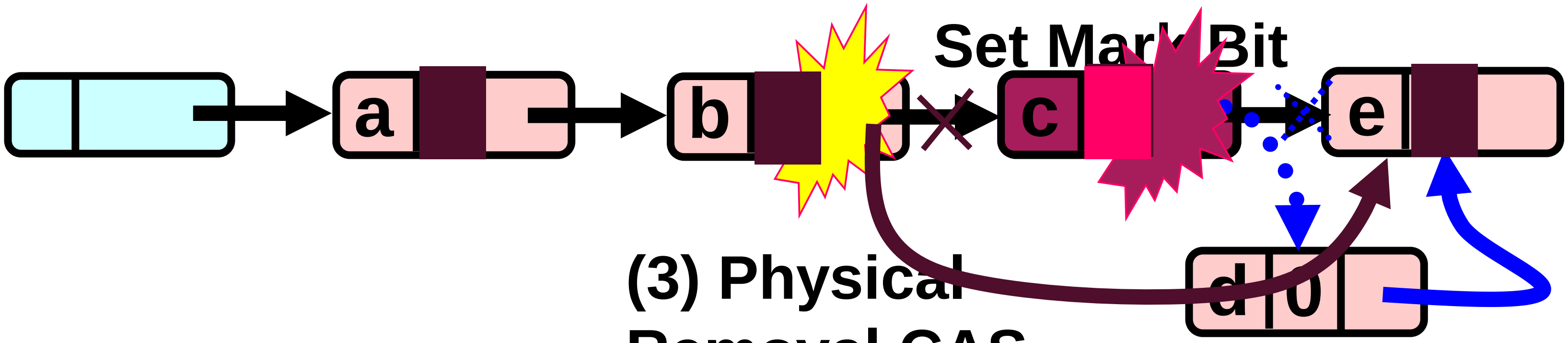


(1) Logical Removal

(3) Physical Removal

(2) Node added

The Solution: Combine Mark and Pointer



(1) Logical Removal
=
Set Mark Bit

(3) Physical
Removal CAS

(2) Fail CAS: Node not
added after logical
Removal

Mark-Bit and Pointer
are CASed together!

Practical Solution(s)

- **Option 1:**
  - Introduce “atomic markable reference” type
  - “Steal” a bit from a pointer
  - Rather complex and OS specific ☹
- **Option 2:**
  - Use Double CAS (or CAS2) ☺

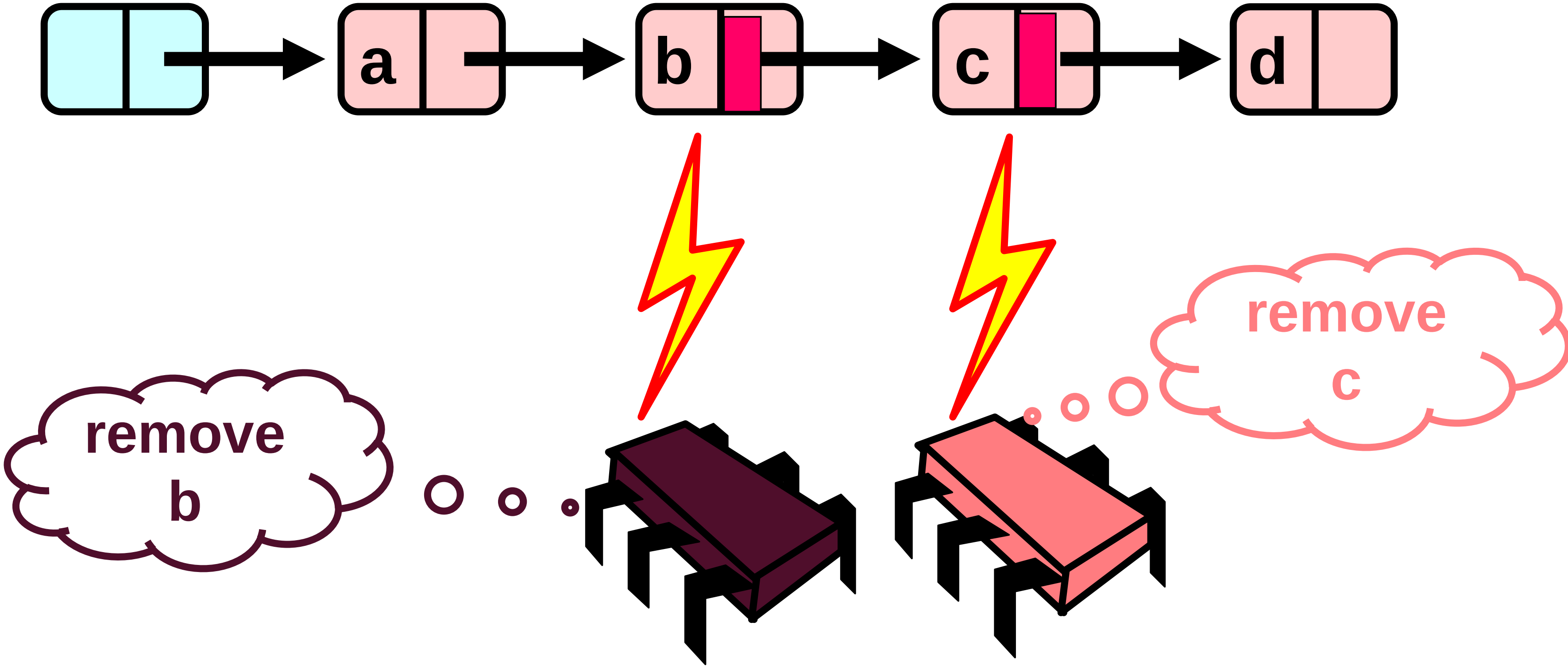
    CAS of two noncontiguous locations
  - Well, not many machines support it ☹

    Any still alive?
- **Option 3:**
  - Our favorite ISA (x86) offers double-width CAS

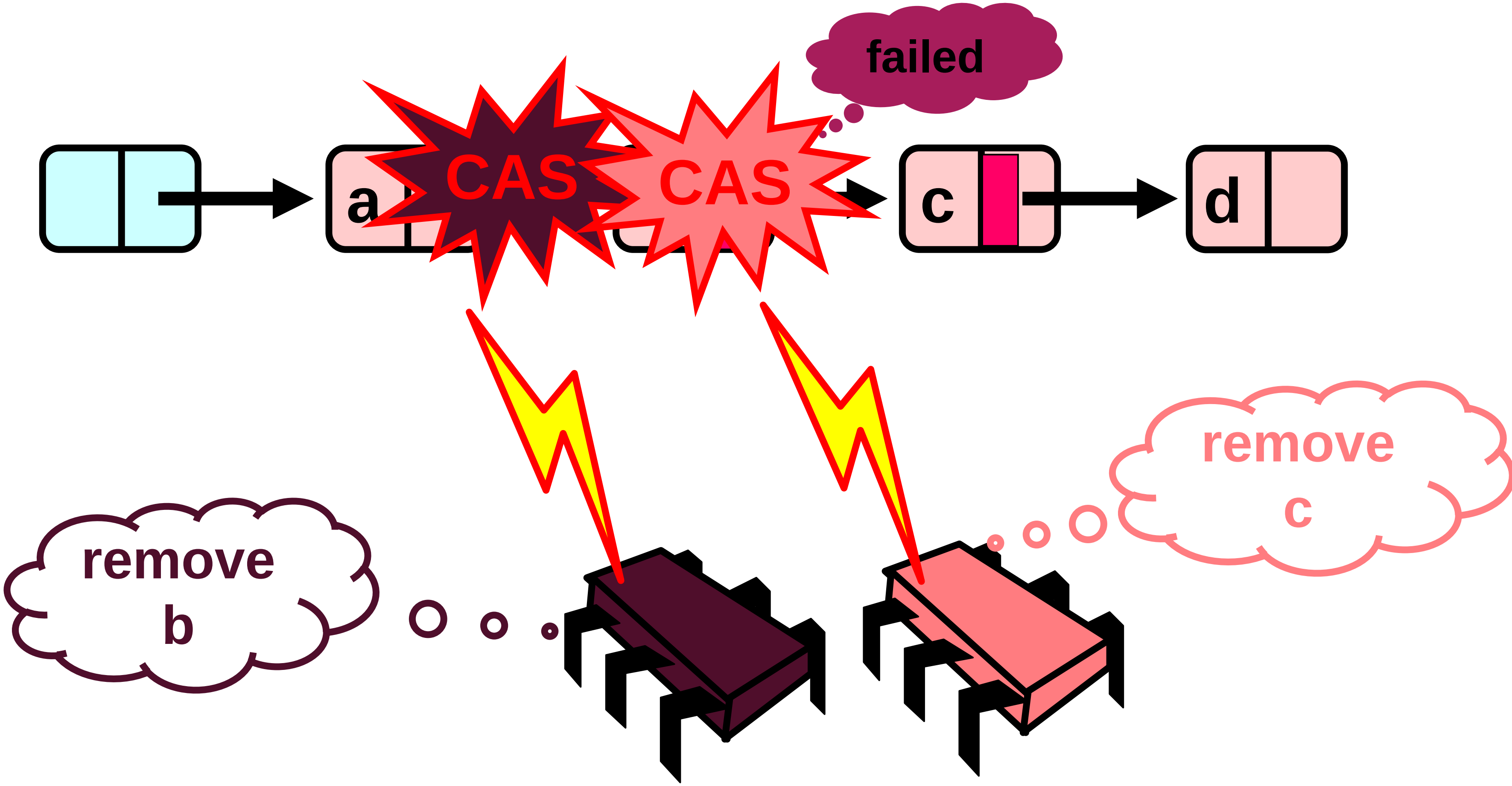
    Contiguous, e.g., lock cmpxchg16b (on 64 bit systems)
- **Option 4:**
  - TM!

    E.g., Intel’s TSX (essentially a cmpxchg64b (operates on a cache line))

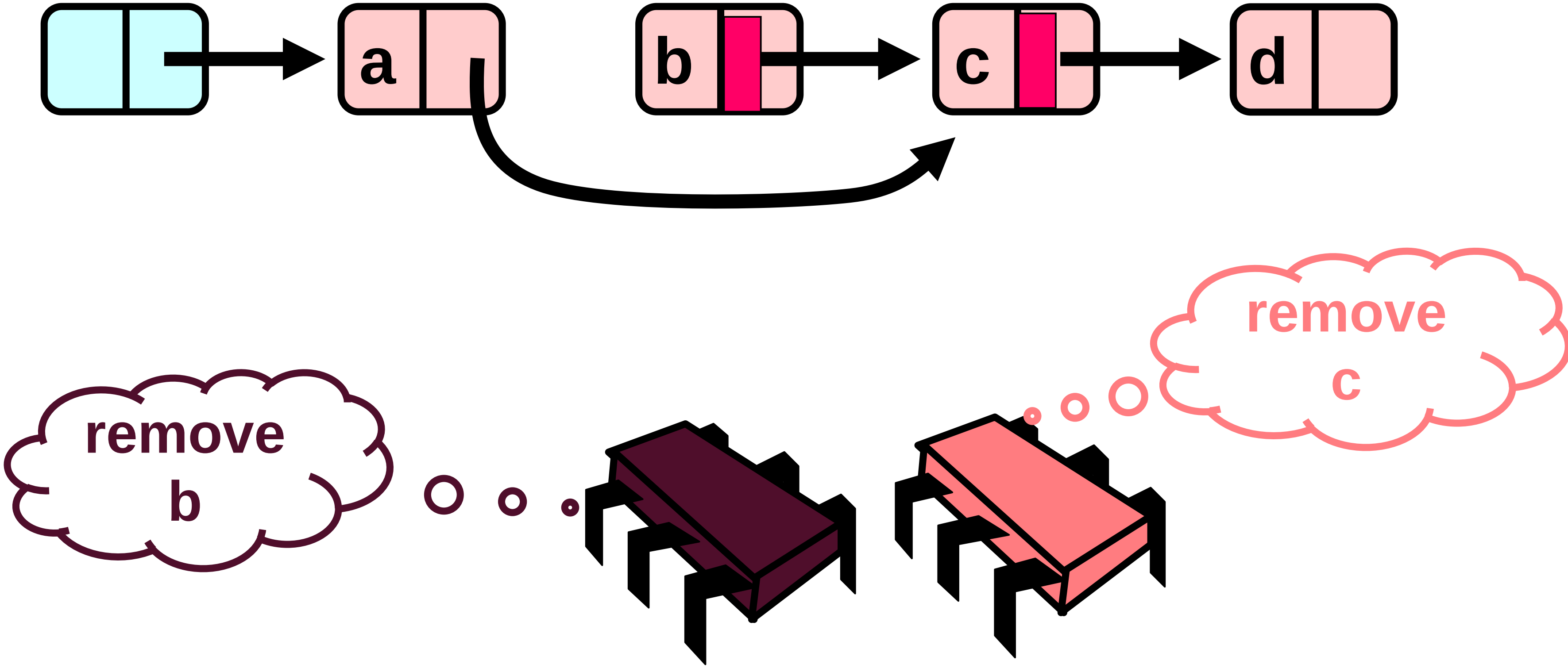
Removing a Node



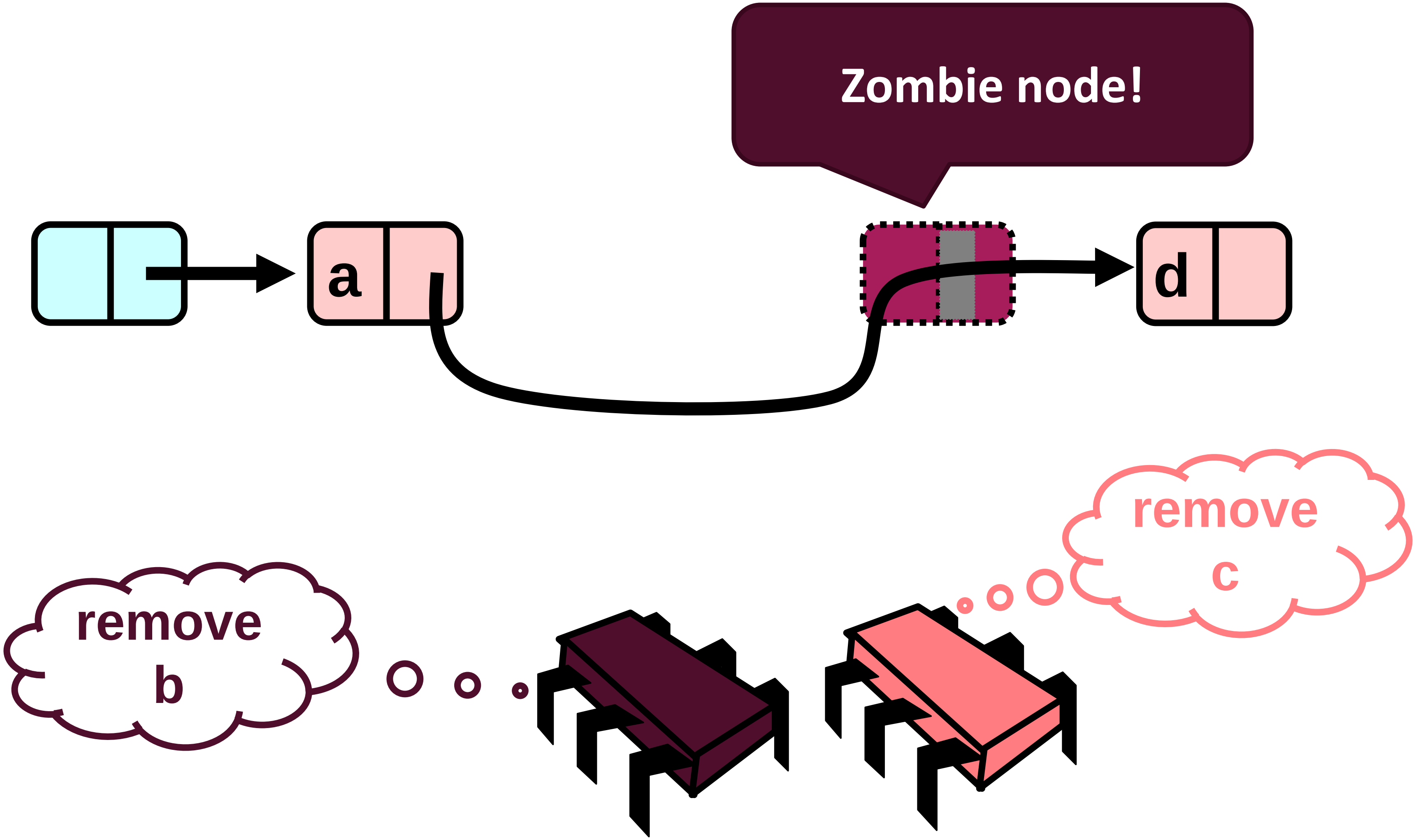
Removing a Node



Removing a Node



Uh oh – node marked but not removed!



Dealing With Zombie Nodes

- **Add() and remove() "help to clean up"**
  - Physically remove any marked nodes on their path
  - I.e., if curr is marked: CAS (pred.next, mark) to (curr.next, false) and remove curr

    If CAS fails, restart from beginning!
- **"Helping" is often needed in wait-free algs**
- **This fixes all the issues and makes the algorithm correct!**

Comments

- **Atomically updating two variables (CAS2 etc.) has a non-trivial cost**
- **If CAS fails, routine needs to re-traverse list**
  - Necessary cleanup may lead to unnecessary contention at marked nodes
- **More complex data structures and correctness proofs than for locked versions**
  - But guarantees progress, fault-tolerant and maybe even faster (that really depends)

More Comments

- **Correctness proof techniques**
  - Establish invariants for initial state and transformations

    E.g., head and tail are never removed, every node in the set has to be reachable from head, …
  - Proofs are similar to those we discussed for locks

    Very much the same techniques (just trickier)

    Using sequential consistency (or consistency model of your choice ☺)

    Lock-free gets somewhat tricky
- **Source-codes can be found in Chapter 9 of "The Art of Multiprocessor Programming"**