Concepts and Models of Parallel and Data-centric Programming

Shared Memory X

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

i12 High Performance Computing
RWTH AACHEN UNIVERSITY

Outline

0. Organization
1. Foundations
2. **Shared Memory**
3. GPU Programming
4. Bulk-Synchronous Parallelism
5. Message Passing
6. Distributed Shared Memory
7. Parallel Algorithms
8. Parallel I/O
9. MapReduce
10. Apache Spark

l. Coarse-grained Synchronization
m. Fine-grained Synchronization
n. Optimistic Synchronization
o. Lazy Synchronization
p. Lock-free Synchronization

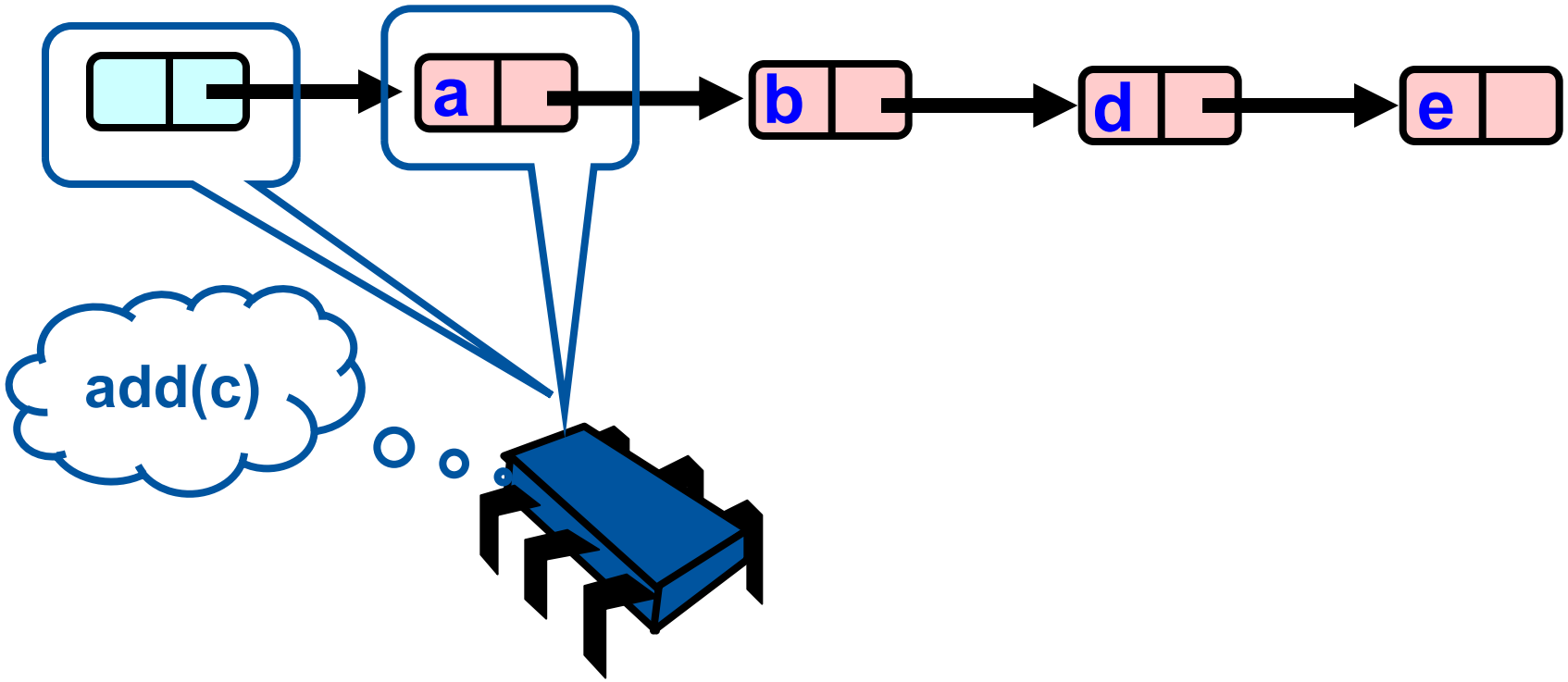
Optimistic Synchronization

Optimistic Synchronization

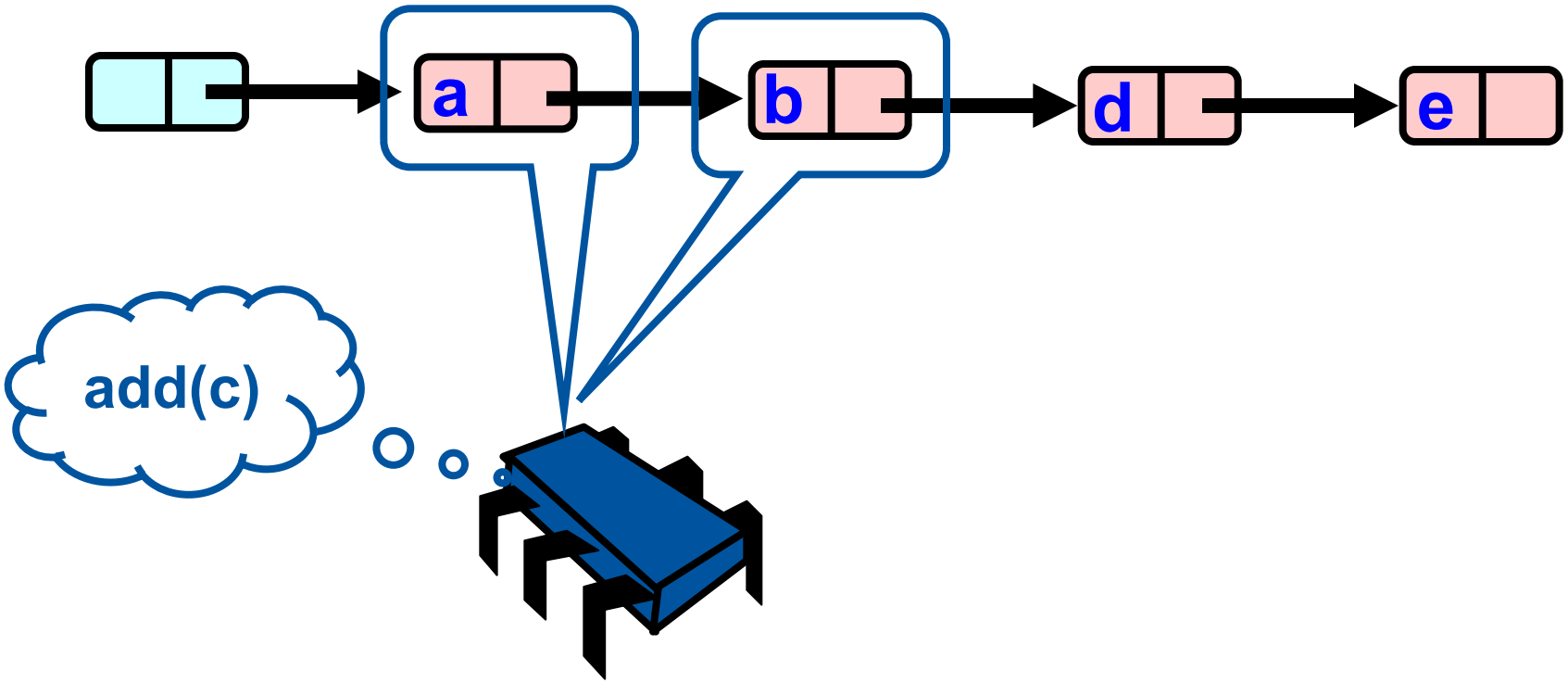
- Search without locking:
  - If you find the element, perform lock and check …
    - OK: we are done
    - Oops: start over

- Evaluation
  - Usually cheaper than locking, but
  - mistakes are expensive (start over: new traversal)

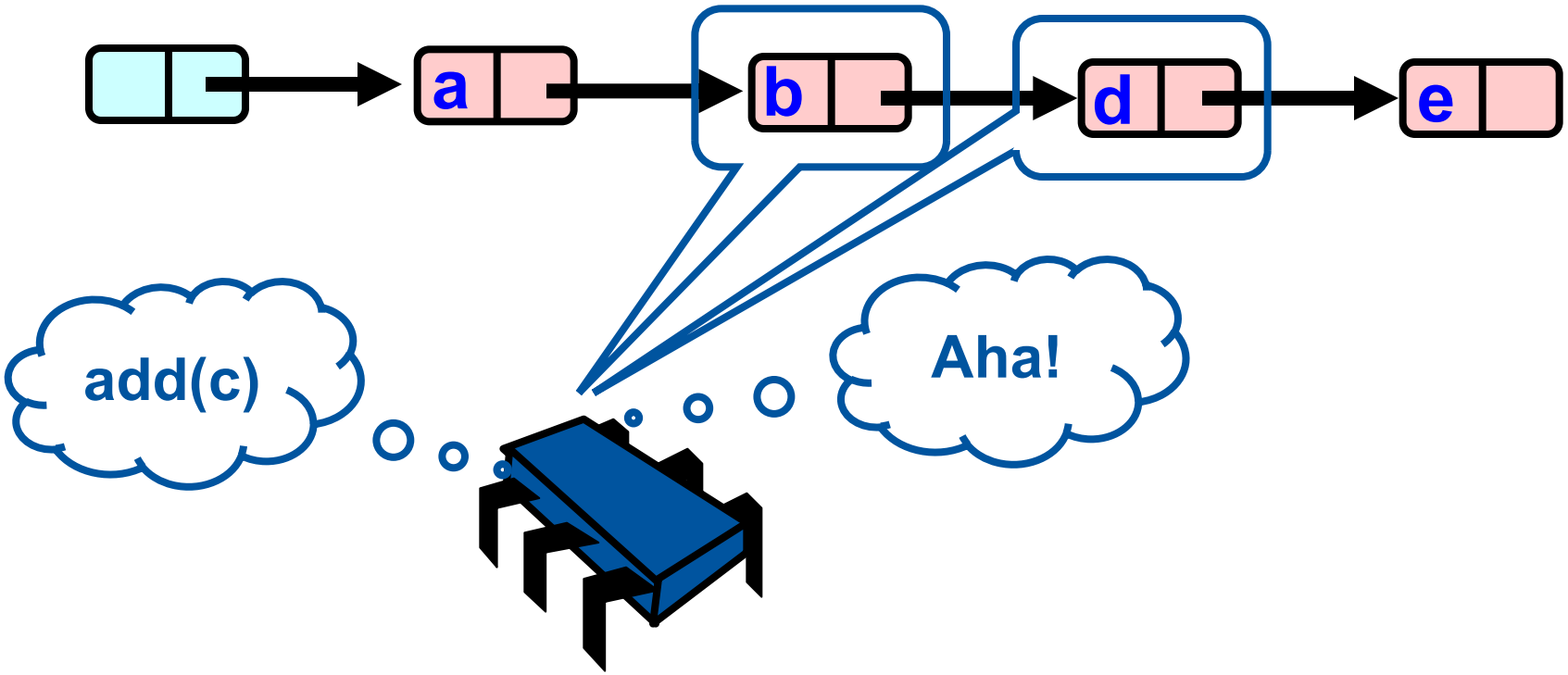
Traversal wo/ Locking



Traversal wo/ Locking

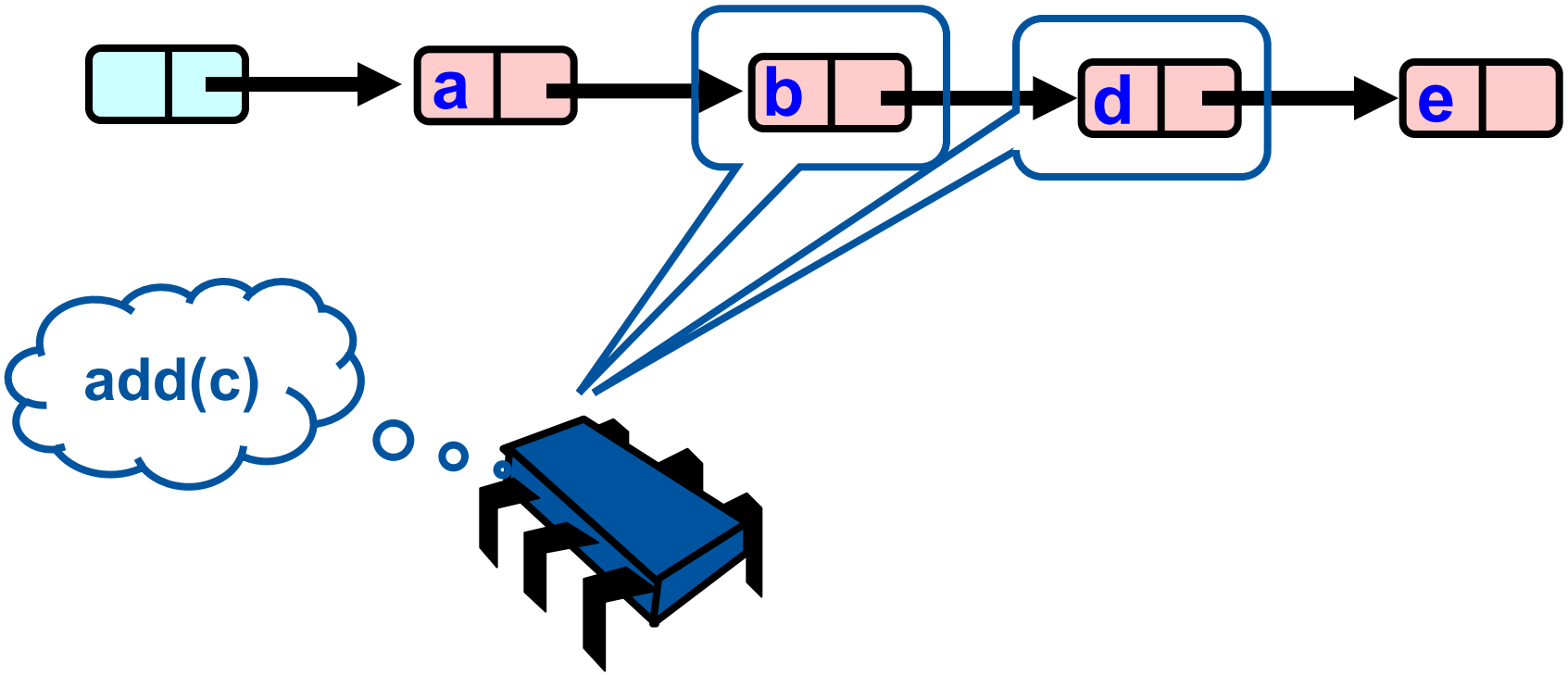


Traversal wo/ Locking

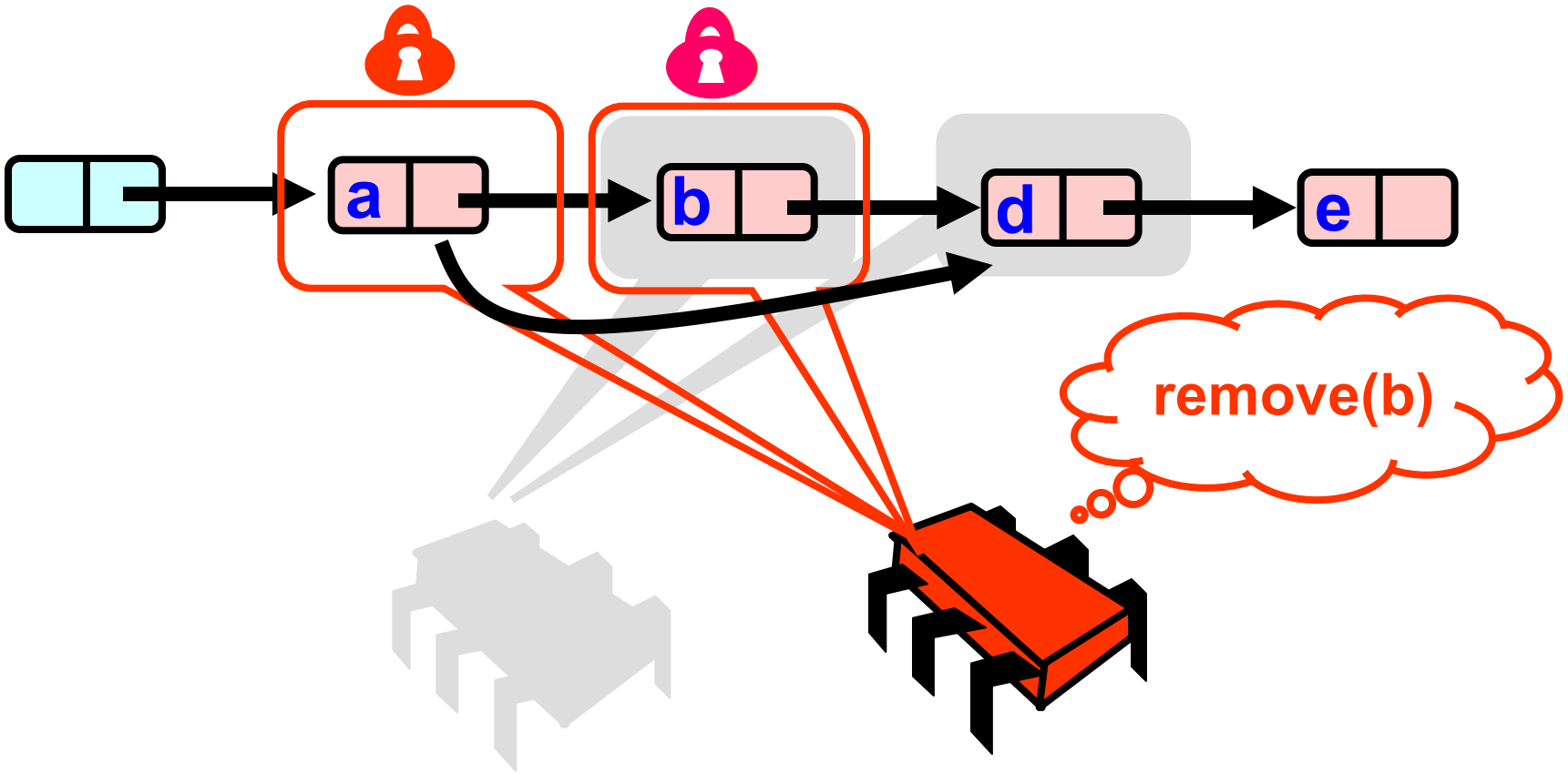


Add w/ Locks

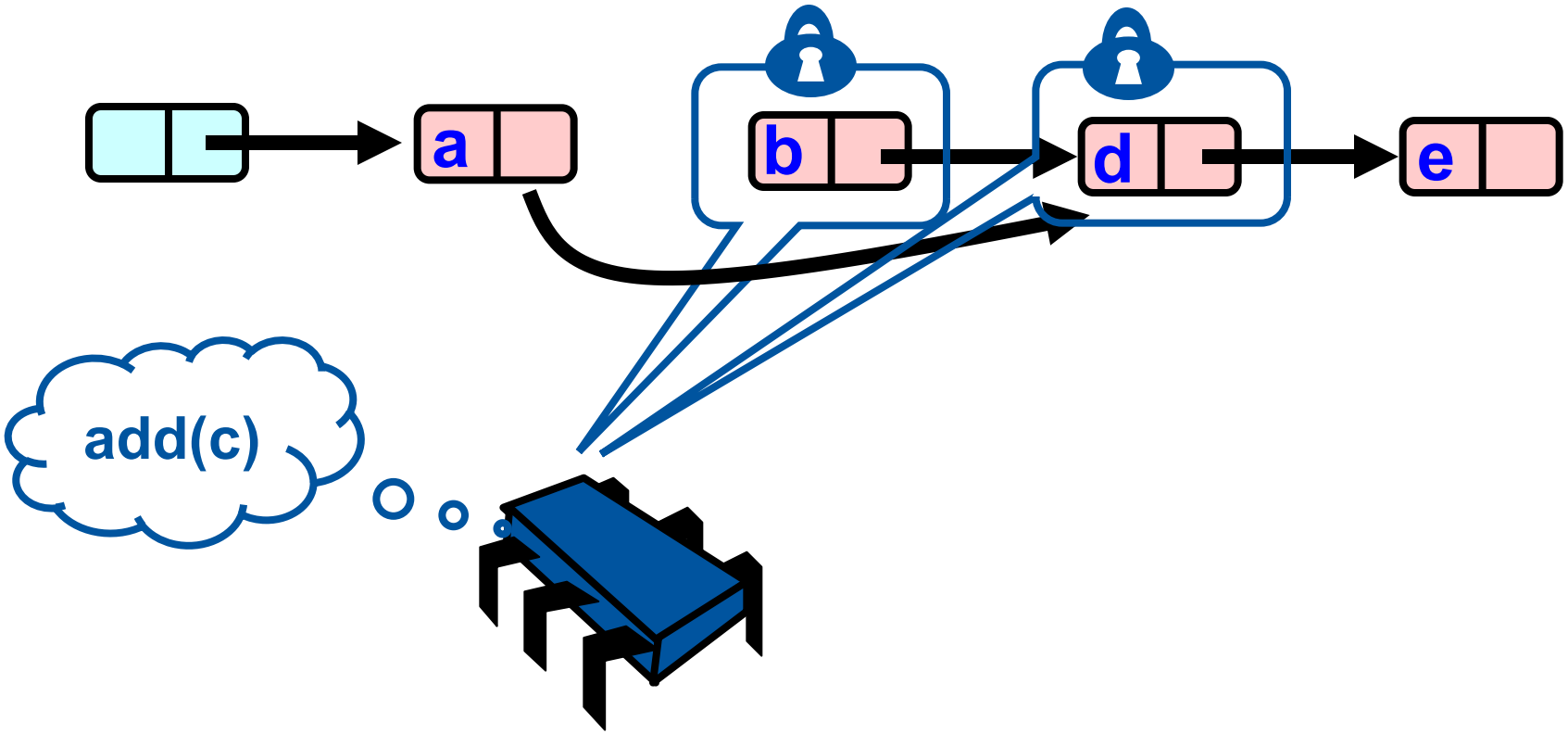
- What could go wrong? Acquire the lock and then perform the add…



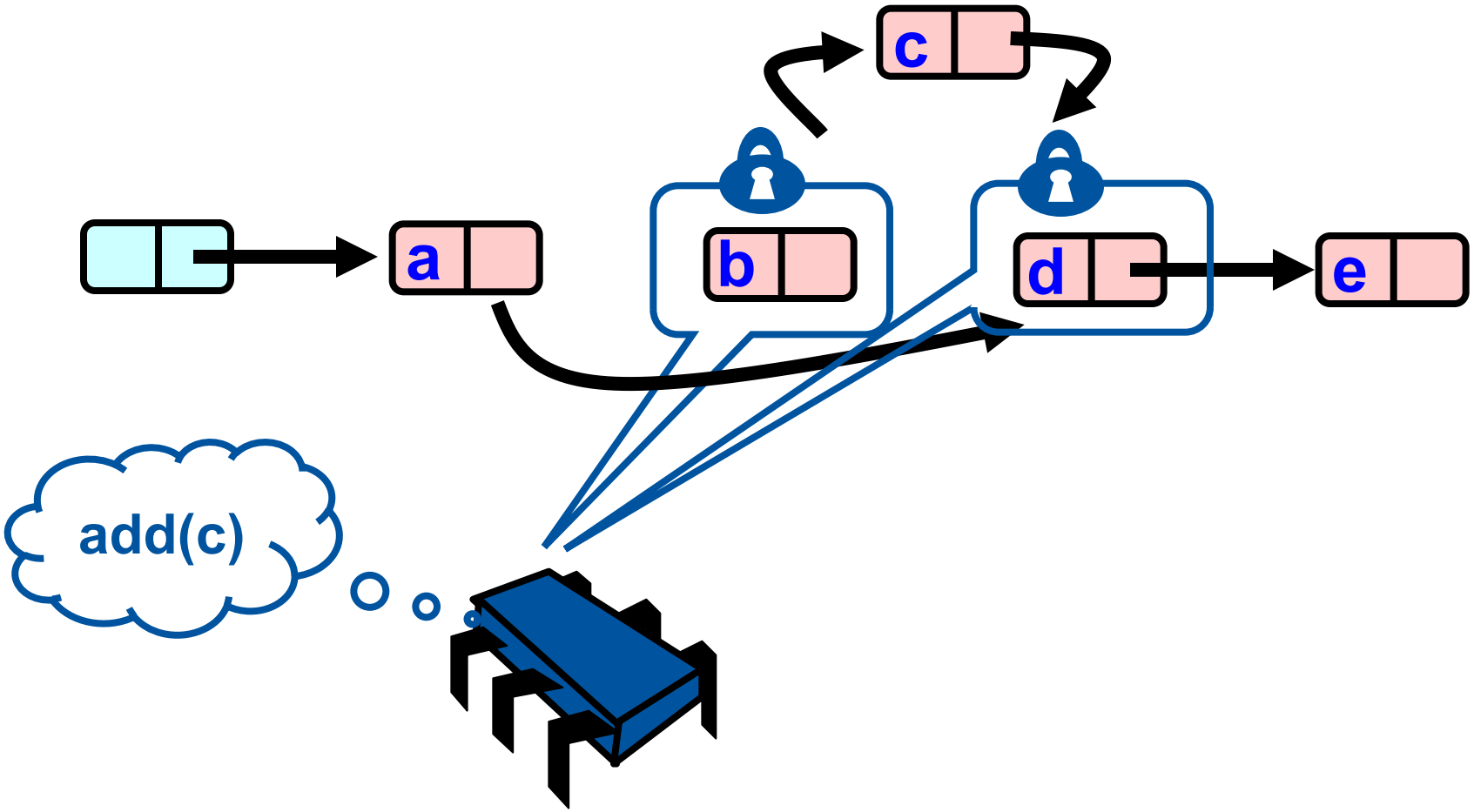
Illustration…



Illustration…



Illustration…



Solution

- Problem: we may traverse deleted nodes!
  - Because we did
    - traversal until we found the node
    - then acquired the lock
    - meanwhile the world might have turned further

Solution

- Problem: we may traverse deleted nodes!
  - Because we did
    - traversal until we found the node
    - then acquired the lock
    - meanwhile the world might have turned further

- Implementation approach:
  - Validation after locking target nodes
    - Verifies correct state of the list, such as:
      - Pred element reachable from head
      - Pred element points to curr element
    - Otherwise: retry (`while`-loop in the corresponding operation)

Illustration: validation

```
bool validate(Node *pred, Node *curr)
{
   Node *node = head;
   while (node->key <= pred->key)
   {
      if (node->key == pred->key)
         return pred->next->key == curr->key;
      node = node->next;
   }
   return false;
}
```

Summary: Optimistic Sync.

- Much less lock acquisition/release
  - Performance + Concurrency

- Optimistic is effective if
  - cost of scanning twice without locks is less than
  - cost of scanning once with locks

- Drawback
  - contains() acquires locks (without, it could be derailed)
  - 90% of calls in many apps