Concepts and Models of Parallel and Data-centric Programming

Shared Memory IX

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

Linked List: coarse-grained synchronization

Coarse-grained Synchronization

- Each method locks the object
  - Avoid contention with optimized lock type
  - Easy to reason about

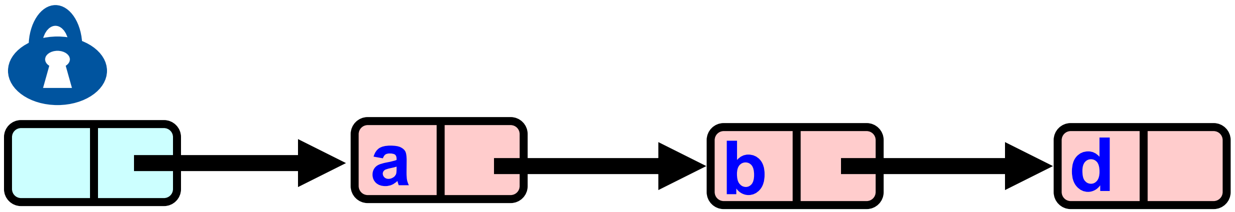


Illustration: add

- Simple …

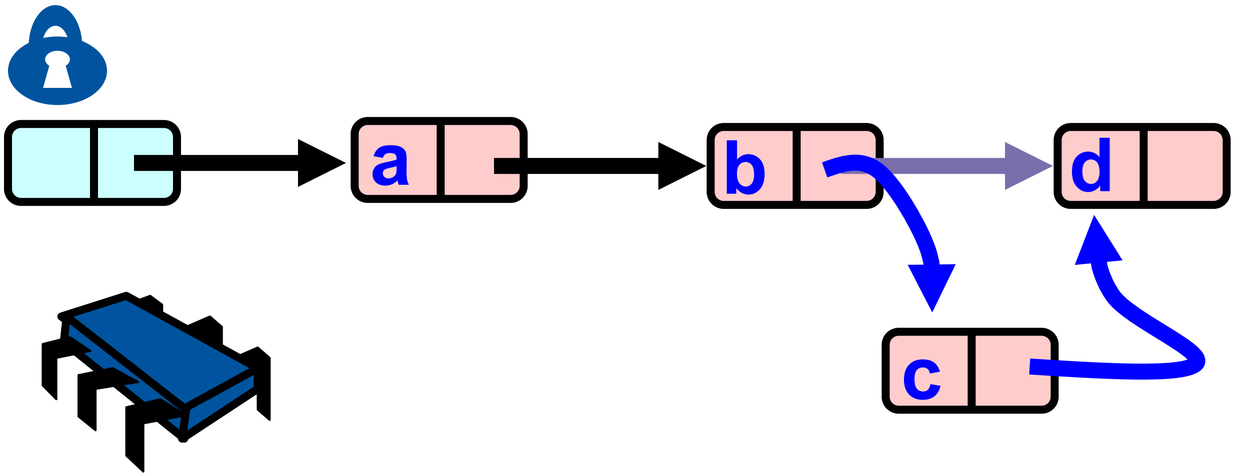
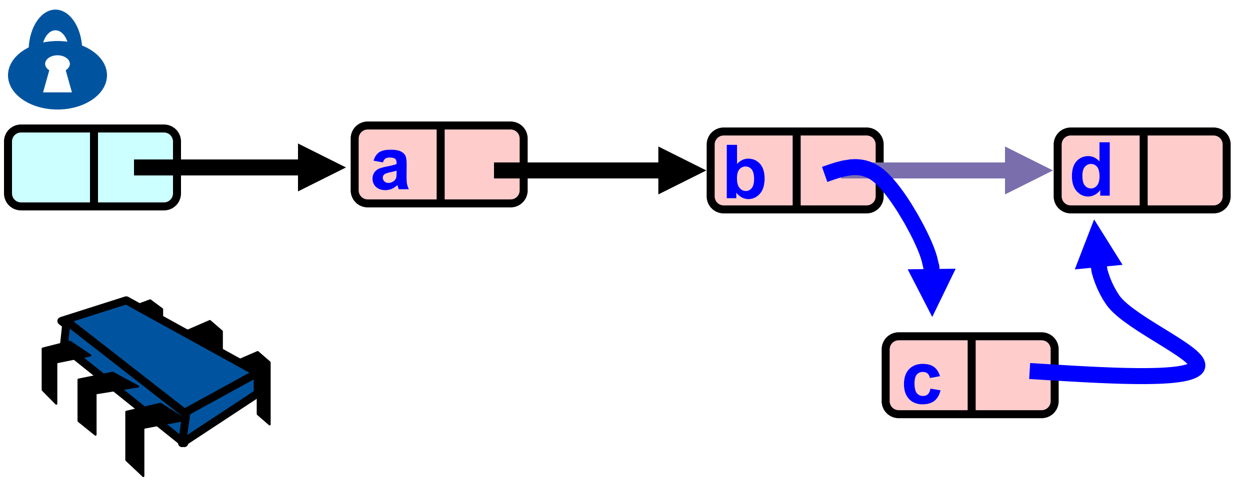


Illustration: add

- Simple …



- … but: bottleneck!

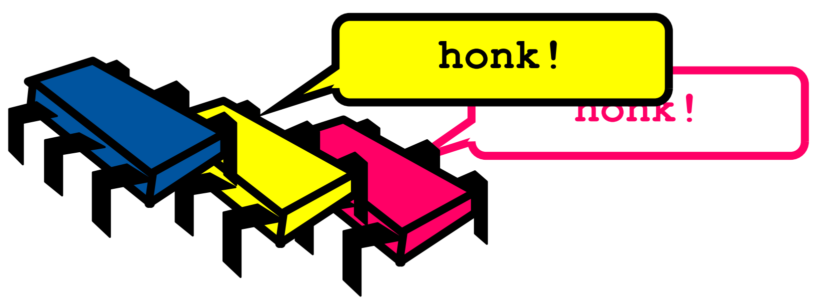
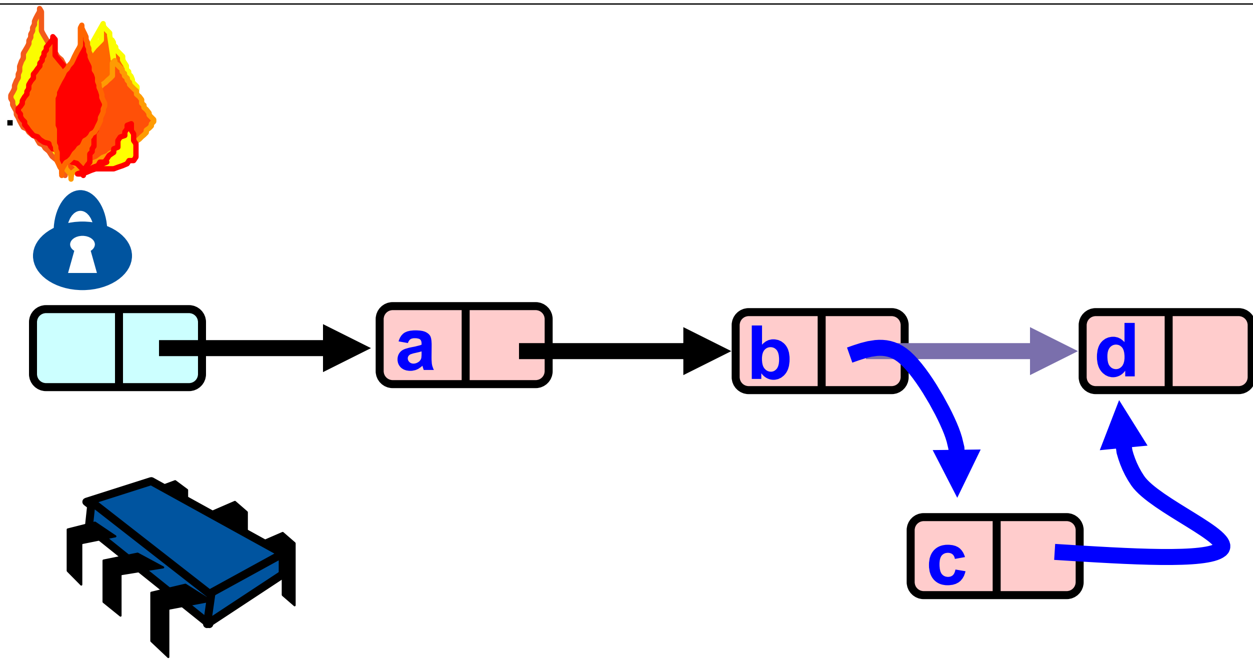
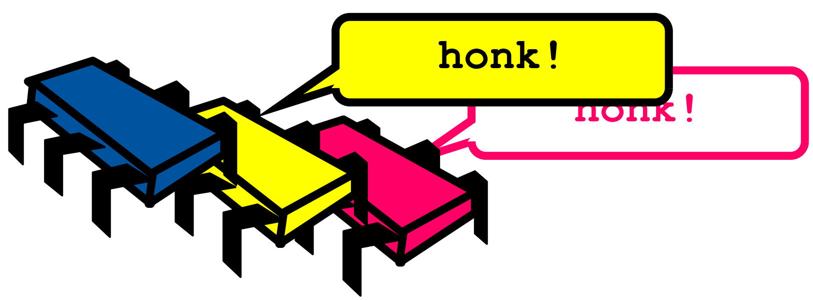


Illustration: add

- Simple …



- … but: bottleneck!



Summary: Coarse-grained Sync.

- Sequential bottleneck
  - Threads “stand in line”
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse
- So why even use a multiprocessor?
  - Simple, clearly correct - deserves some respect!

Linked List: fine-grained synchronization

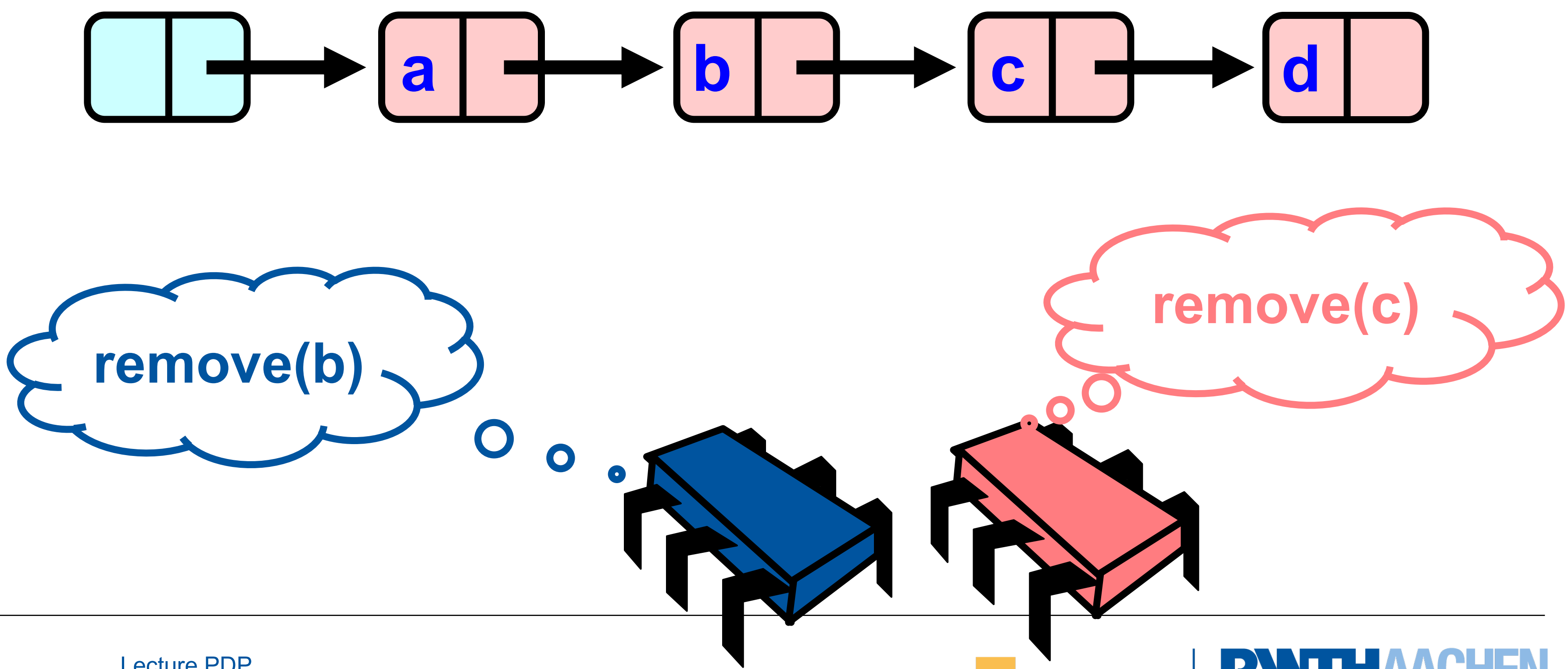
Fine-grained Synchronization

- Instead of using a single lock: split object into
  - Independently-synchronized components
  - Methods conflict when they access the same component at the same time

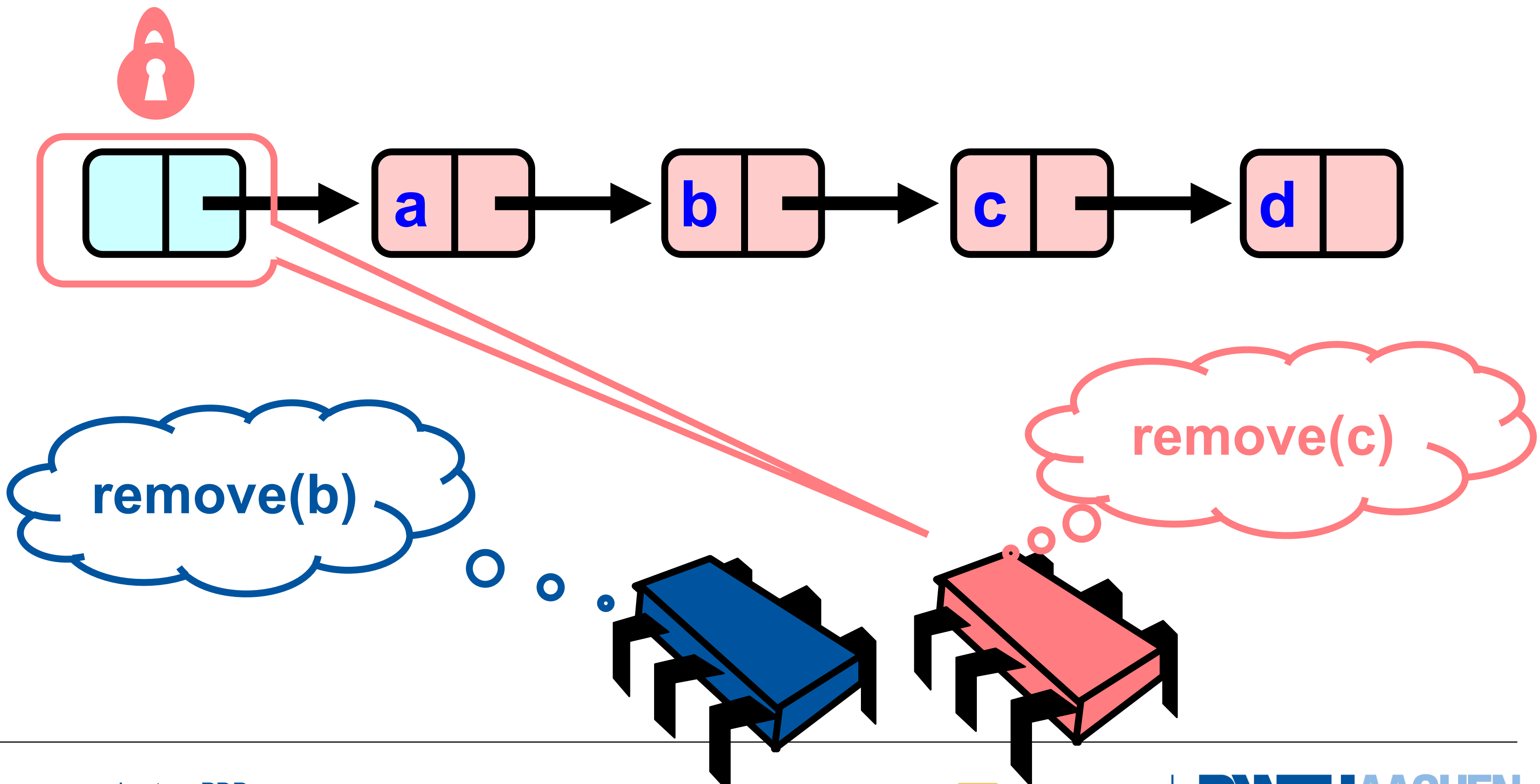
- Improve concurrency by locking individual entries
  - instead of placing a lock on the entire list, add a lock to each entry
  - as a thread traverses the list, it locks each entry when it first visits, and sometime later releases it
  - Methods that work on disjoint pieces need not exclude each other

- Requires careful thought
  “Do not meddle in the affairs of wizards, for they are subtle and quick to anger”

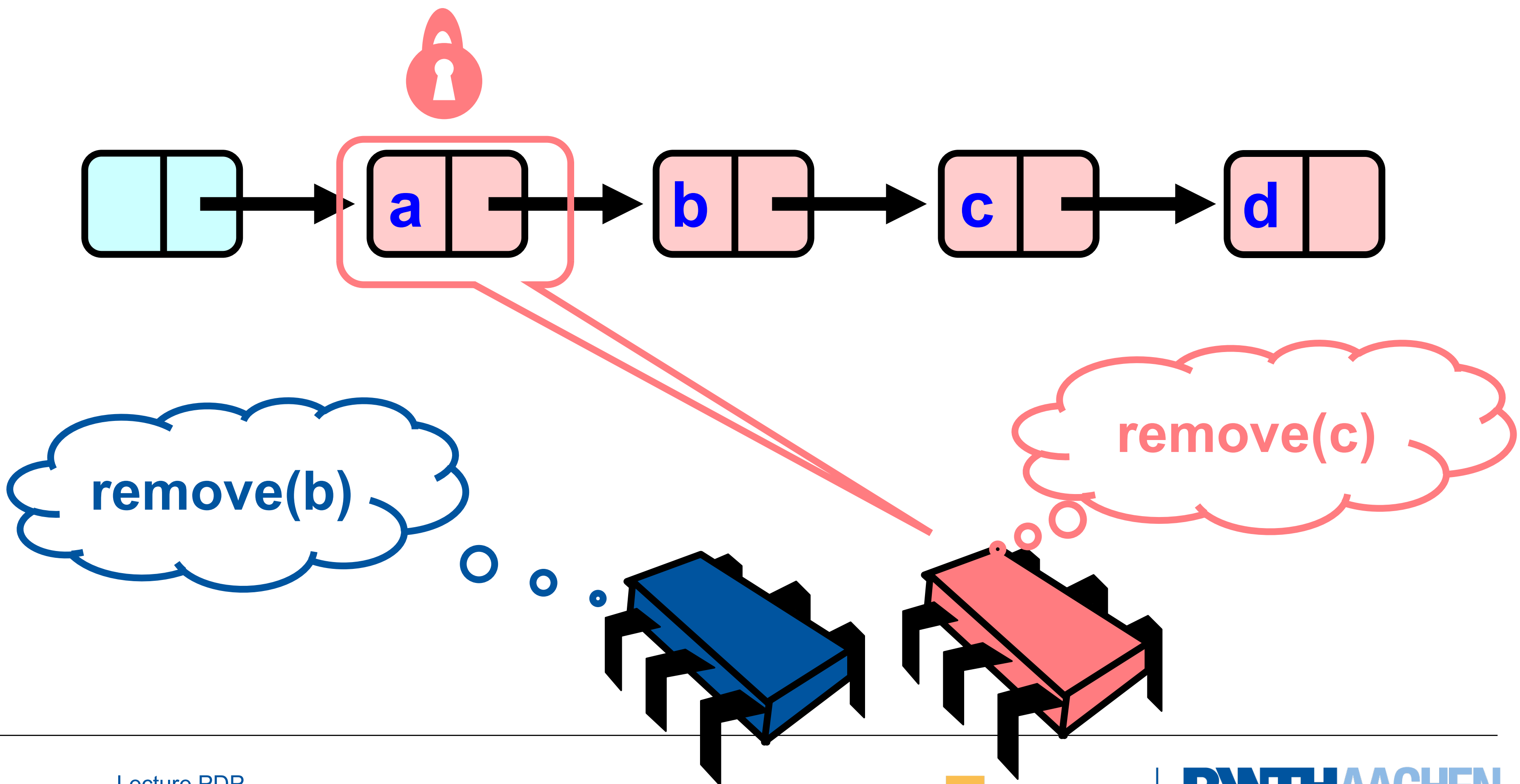
First try of concurrent remove / 1



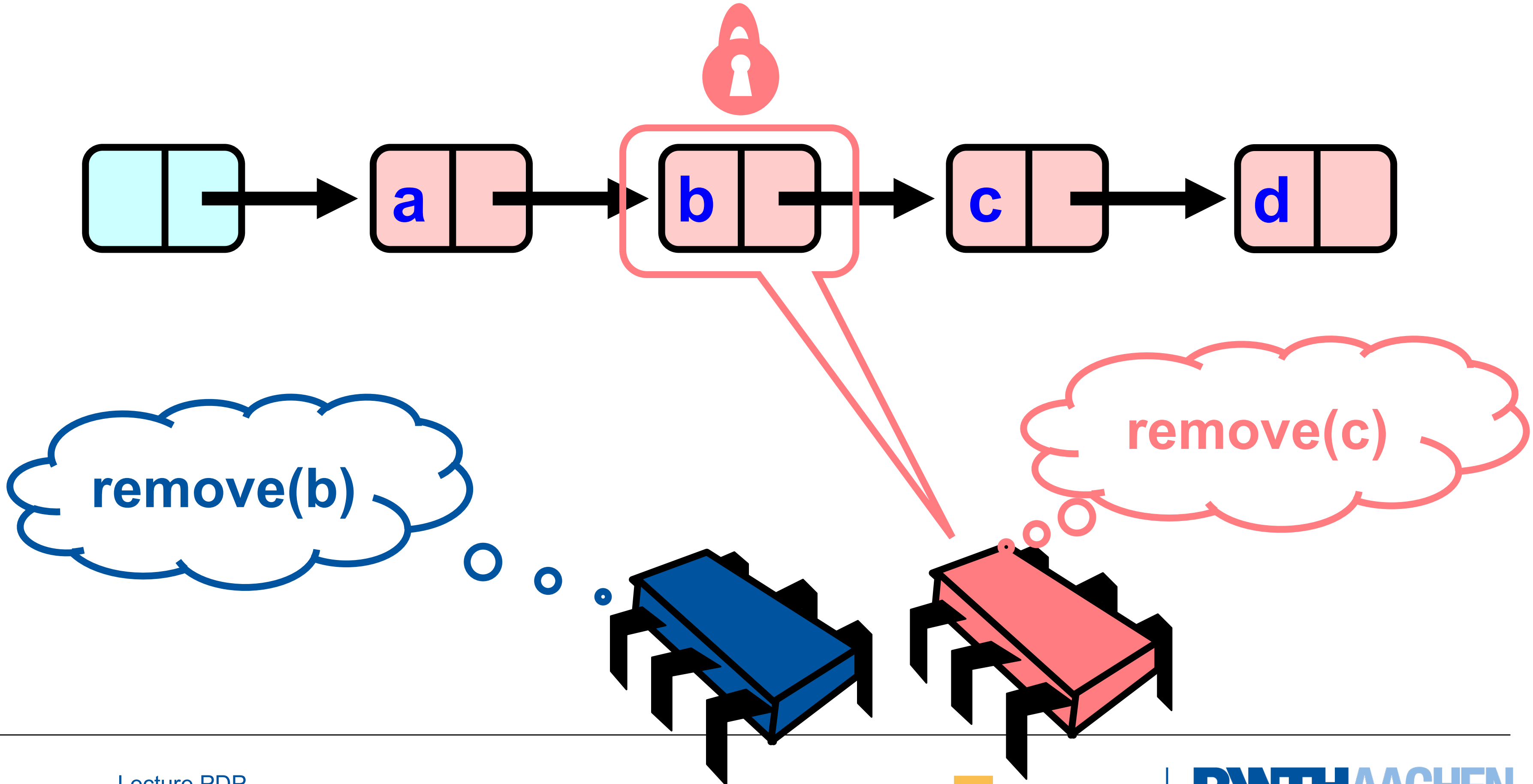
First try of concurrent remove / 2



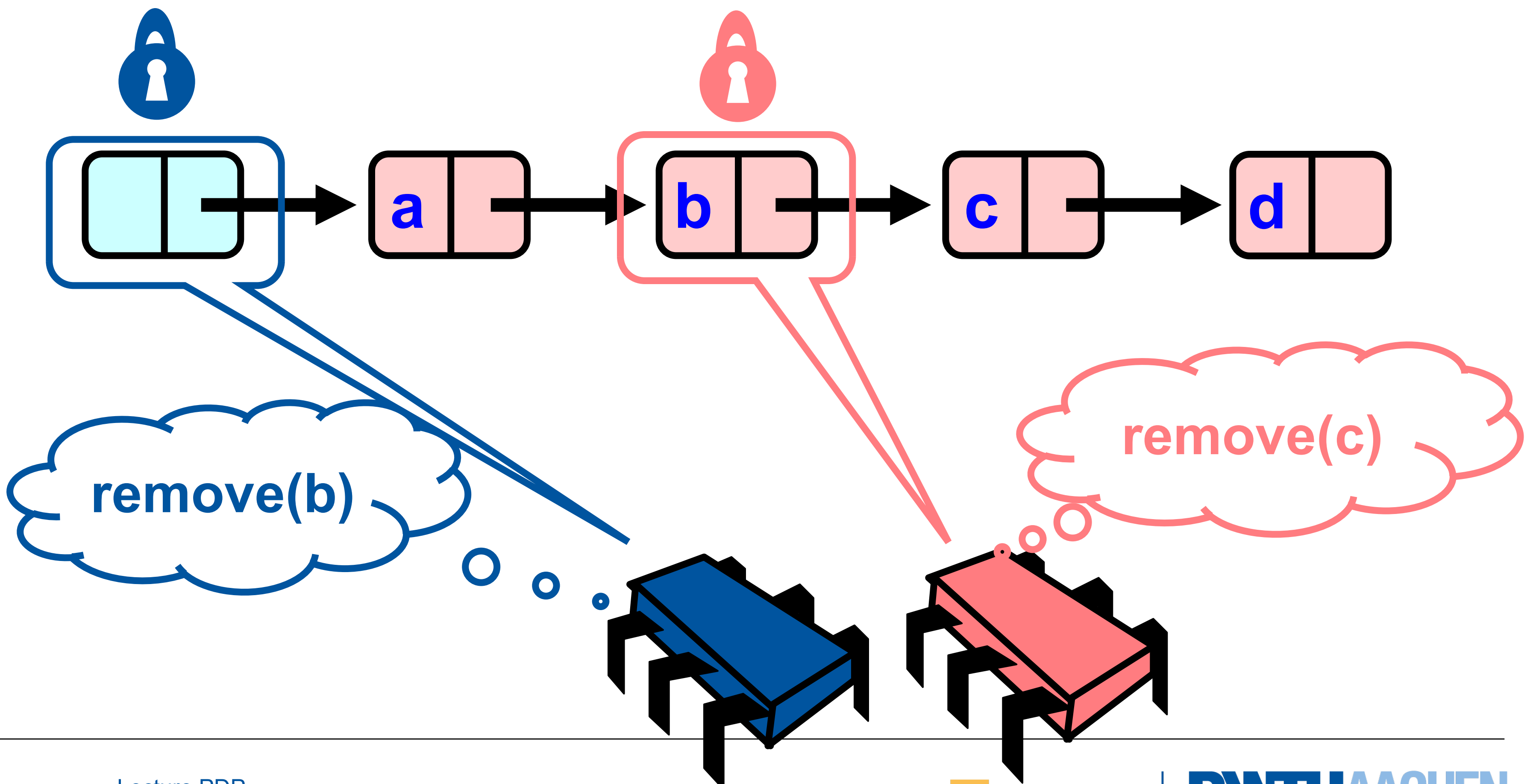
First try of concurrent remove / 3



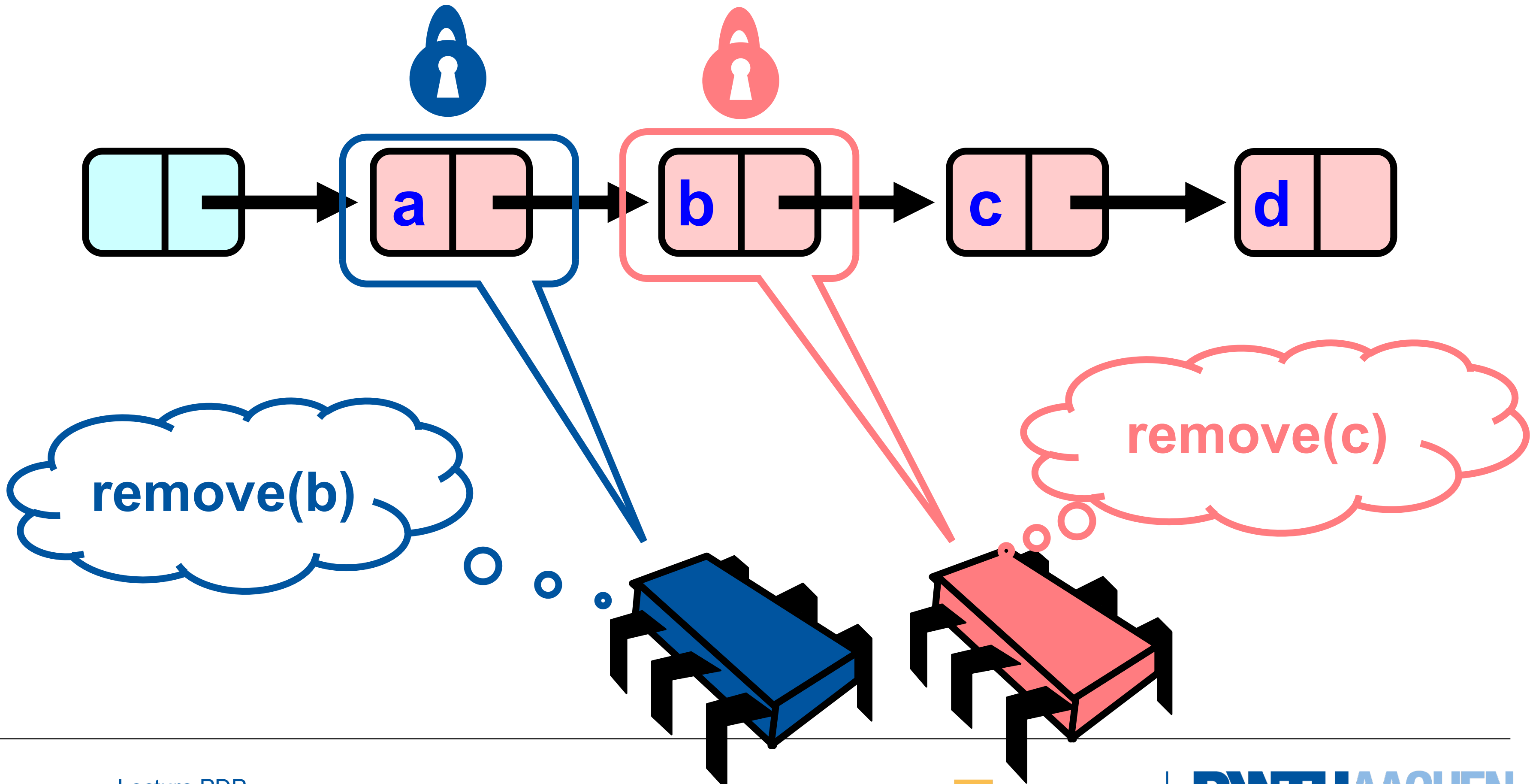
First try of concurrent remove / 4



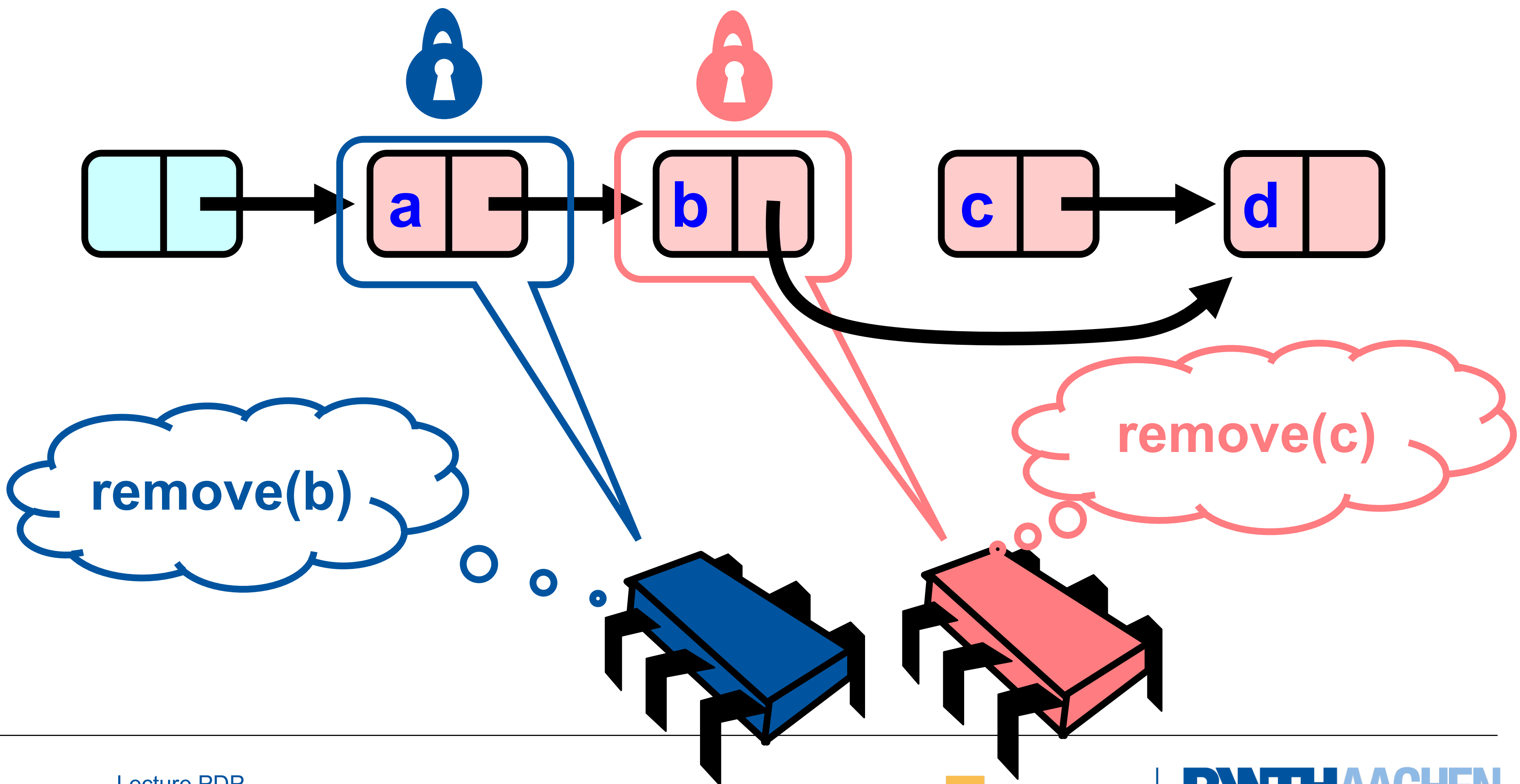
First try of concurrent remove / 5



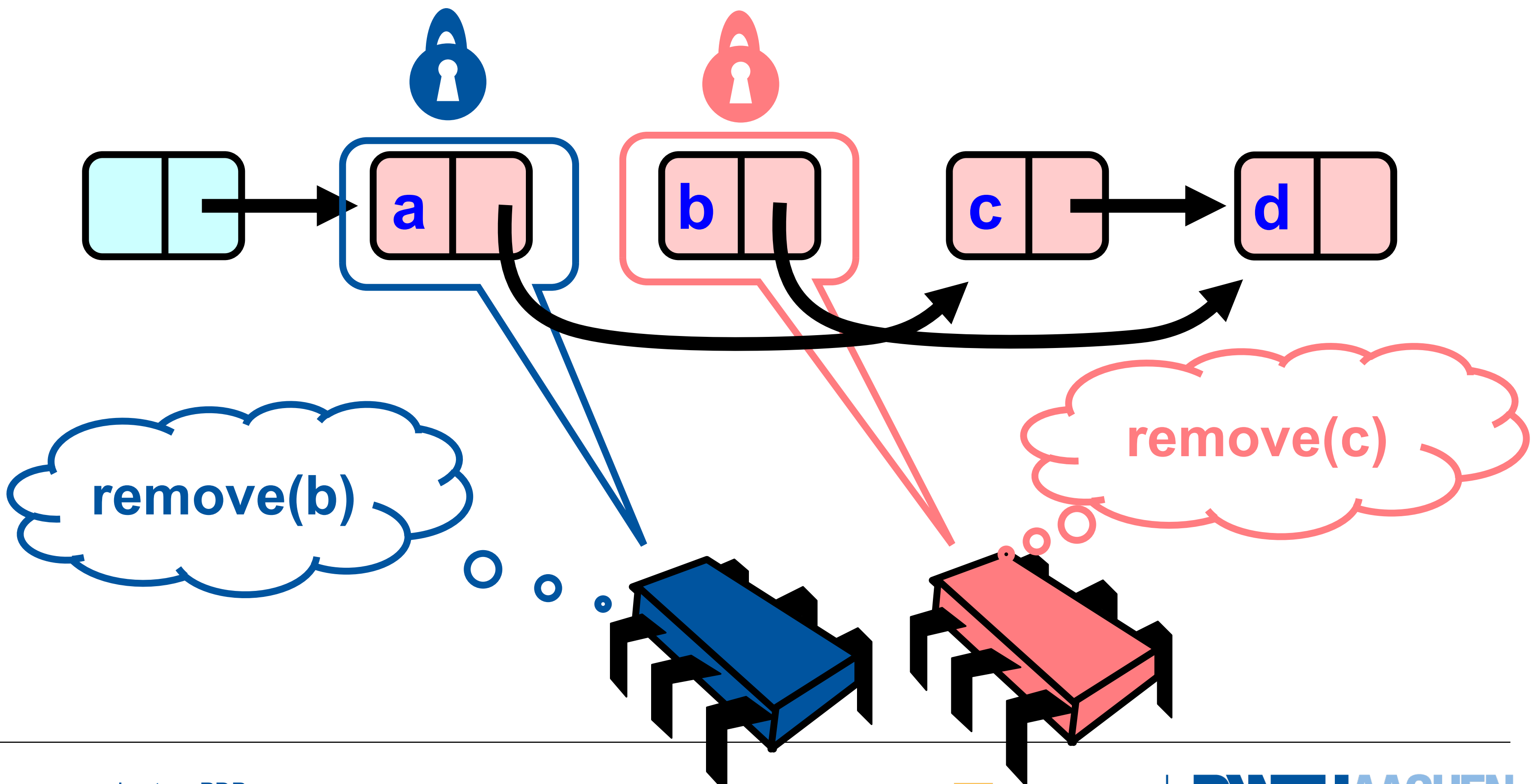
First try of concurrent remove / 6



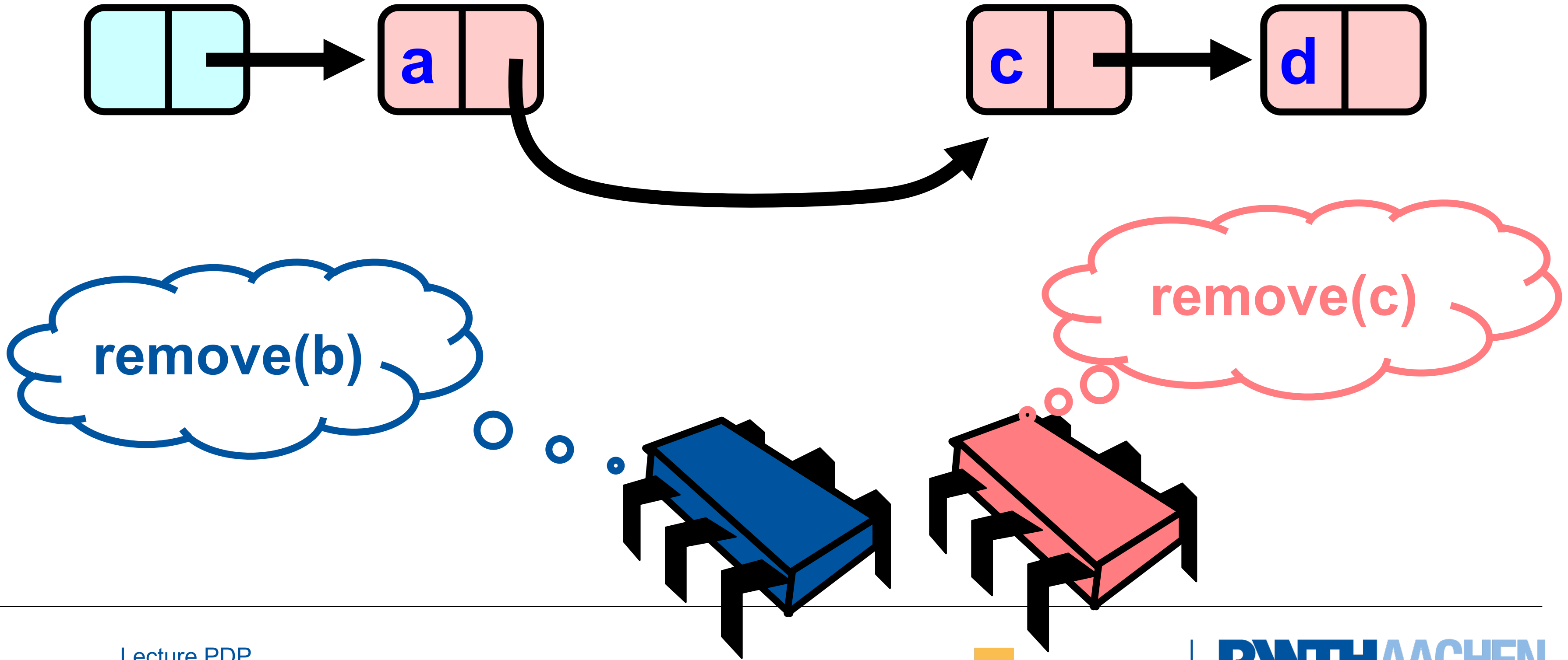
First try of concurrent remove / 7



First try of concurrent remove / 8

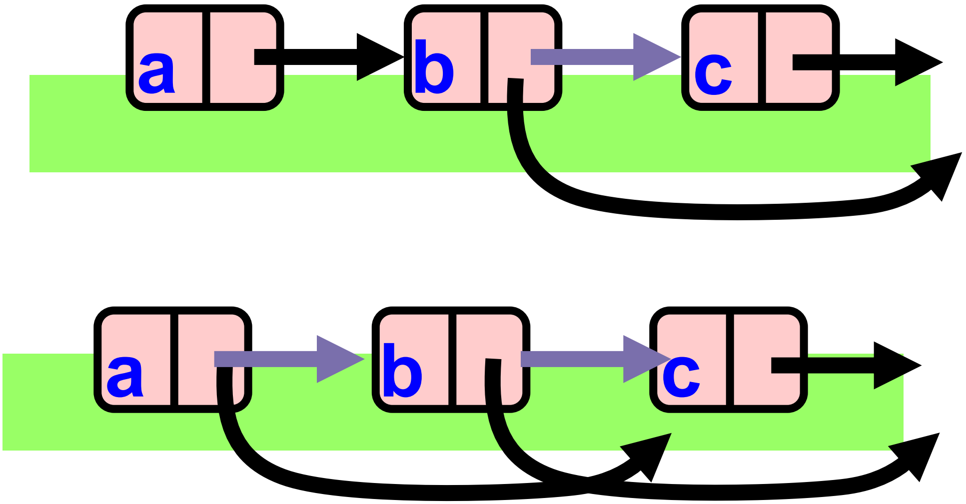


First try of concurrent remove / 9



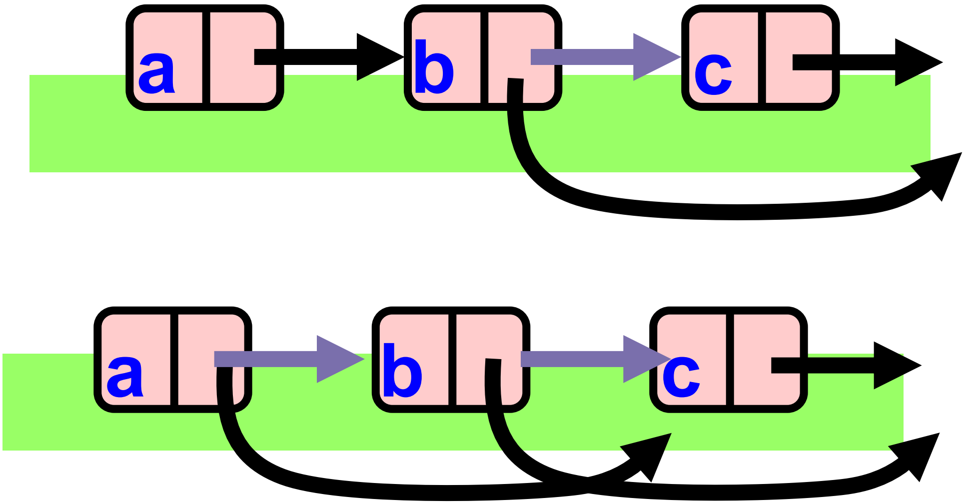
Problem of first try

- To delete node c
  - Swing node b's next field to d
- Problem is,
  - Someone deleting b concurrently could direct a pointer to c



Problem of first try

- To delete node c
  - Swing node b's next field to d
- Problem is,
  - Someone deleting b concurrently could direct a pointer to c



- If a node is locked
  - No one can delete node's *successor*
- If a thread locks
  - Node to be deleted
  - And its predecessor
  - Then it works

Concurrent remove / 1

- Boilerplate

```
bool Node::remove(int item_key)
{
    Node *pred, *curr;
    std::unique_lock<std::mutex> lpred(pred->mut,
            std::defer_lock);
    std::unique_lock<std::mutex> lcurr(curr->mut,
            std::defer_lock);
    {
      ...
    }
}
```

Ensure that locks are released

Everything else

- Note: a `std::mutex` has to be added to the Node class

Concurrent remove / 1

- Boilerplate

```
bool Node::remove(int item_key)
{
    Node *pred, *curr;
    std::unique_lock<std::mutex> lpred(pred->mut,
            std::defer_lock);
    std::unique_lock<std::mutex> lcurr(curr->mut,
            std::defer_lock);
    {
        ...
    }
}
```

Ensure that locks are released

Everything else

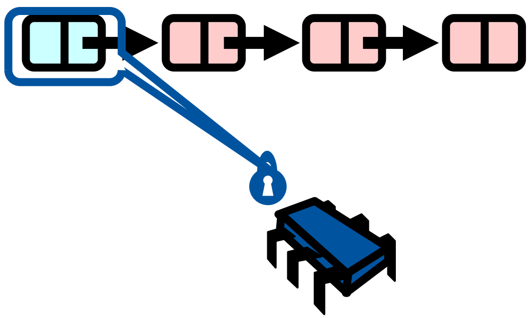
- Note: a `std::mutex` has to be added to the Node class
- Q: why do we have to ensure that locks are released?

Concurrent remove / 2

- Remove method

```
pred = this->head;
lpred.lock();
curr = pred->next;
lcurr.lock();
...
```

Lock pred == head



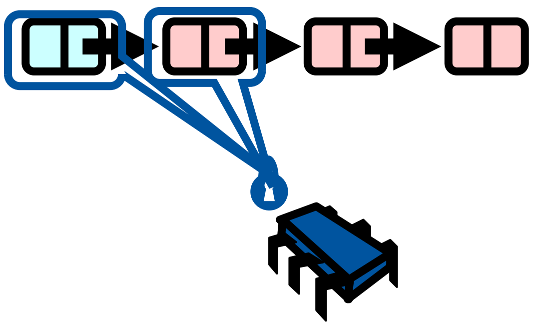
Concurrent remove / 2

- Remove method

```
pred = this->head;
lpred.lock();
curr = pred->next;
lcurr.lock();
...
```

Lock current

List traversal

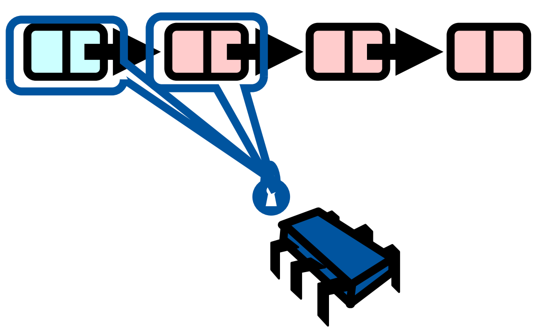


Concurrent remove / 3

- List traversal

```
while (curr->key <= item_key) {
  if (item_key == curr->key) {
   pred->next = curr->next;
   return true;
  }
  lpred.unlock();
  pred = curr;
  curr = curr->next;
  lcurr.lock();
 }
return false;
```

At start of each loop:
curr and pred locked

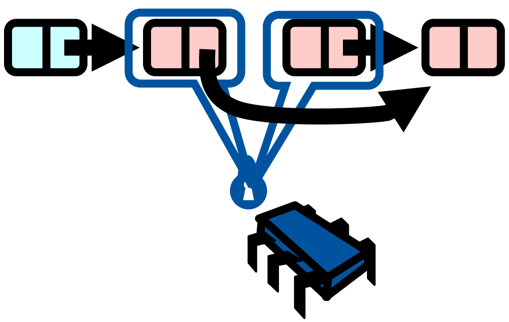


Concurrent remove / 3

- List traversal

```
while (curr->key <= item_key) {
  if (item_key == curr->key) {
   pred->next = curr->next;
   return true;
  }
  lpred.unlock();
  pred = curr;
  curr = curr->next;
  lcurr.lock();
 }
return false;
```

If item found: remove

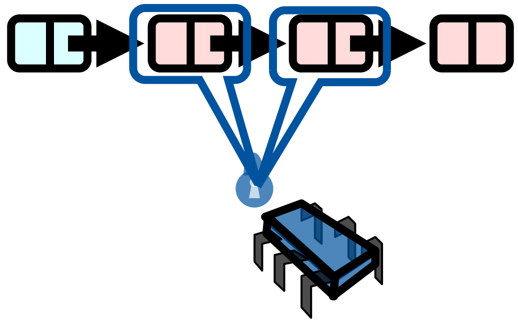


Concurrent remove / 3

- List traversal

```
while (curr->key <= item_key) {
  if (item_key == curr->key) {
   pred->next = curr->next;
   return true;
  }
  lpred.unlock();
  pred = curr;
  curr = curr->next;
  lcurr.lock();
 }
return false;
```

Proceed



Concurrent remove / 3

- List traversal

```
while (curr->key <= item_key) {
  if (item_key == curr->key) {
   pred->next = curr->next;
   return true;
  }
  lpred.unlock();
  pred = curr;
  curr = curr->next;
  lcurr.lock();
 }
return false;
```

Element not present

Adding an element

- To add node e
  - Must lock predecessor
  - ~~Must lock successor~~

- Neither can be deleted

Summary: Fine-grained Sync.

- Easy to check that
  - tail always reachable from head
  - nodes sorted, no duplicates
- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Inefficient