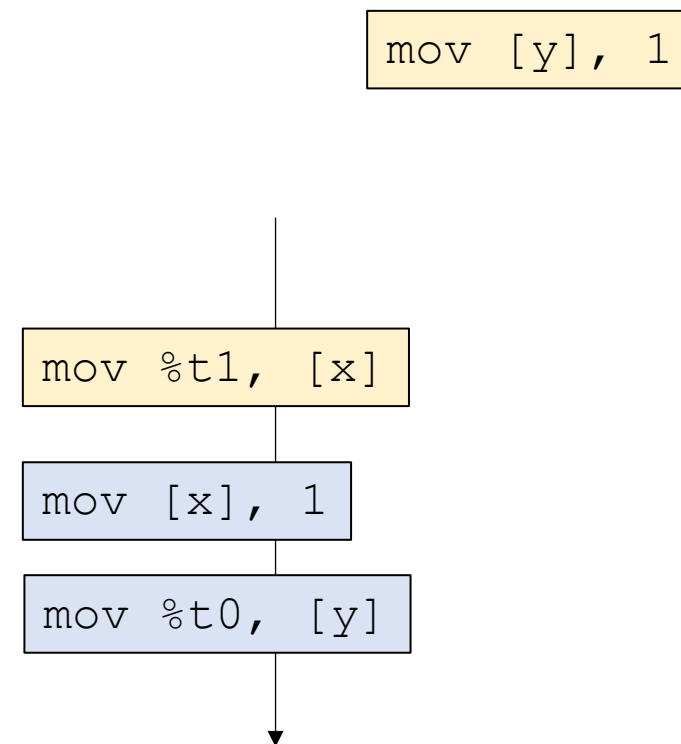


CSE113: Parallel Programming

March 11, 2024

- **Topics:**

- Finish up weak memory models



Announcements

Grading

- HW 2 is graded. Please let us know ASAP if there are any issues.
 - Let us know by Thursday if there are issues
 - Auto grading is difficult here, so don't hesitate to reach out! Make a private post on piazza or see a TA
 - Error in ratio auto grader. This should be fixed by now.
 - Trying to have HW 3 graded by the end of the week

Announcements

- Today is the last day to turn in HW 4
 - Hopefully you had fun with a taste of GPU programming
- HW 5 was released last Friday
 - Due on the day of the final
 - Last day to turn it in is the 21
 - It could be useful to work on for the final

Announcements

- Today is the last day to turn in HW 4
 - Hopefully you had fun with a taste of GPU programming
- HW 5 was released last Friday
 - Due on the day of the final
 - Last day to turn it in is the 21
 - It could be useful to work on for the final

Announcements

Final

- Allowed 3 pages of notes, front and back
- Similar to the midterm (50% longer)
- Time: Monday March 18: 7:30 – 10:30 PM

Announcements

Video games for parallel programming education

- Joint project with the CM department
- Performing a user-study
- Max 160 minutes, \$30 cash
- Play a puzzle video game with parallel programming concepts
- Located in the UC Santa Cruz Silicon Valley Campus
- Includes a tour of the campus, meet & greet with the CM researchers there
- I'll post more information in canvas

Announcements

SETs are out, please do them! It helps us out a lot

Both bad things and good things!

Previous quiz + Review

Previous quiz + Review

There was no quiz last time; thanks to Gurpreet and Jessica for handling the lecture.

Memory models

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test

Can `t0 == t1 == 0`?

Thread 0:

```
mov [x], 1  
mov %t0, [y]
```

```
mov [x], 1
```

```
mov %t0, [y]
```

Thread 1:

```
mov [y], 1  
mov %t1, [x]
```

```
mov [y], 1
```

```
mov %t1, [x]
```



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

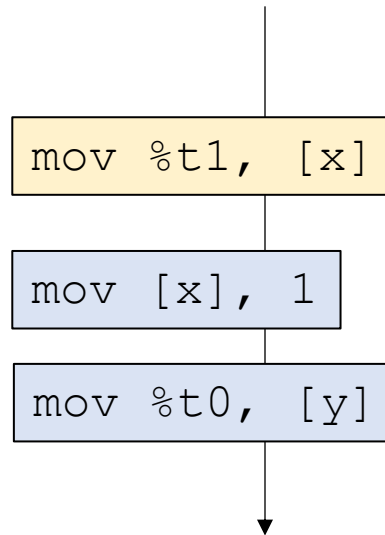
```
mov [x], 1  
mov %t0, [y]
```

Another test

Can `t0 == t1 == 0`?

Thread 1:

```
mov [y], 1  
mov %t1, [x]
```



mov [y], 1

no place for this event!

What if we actually run this code?

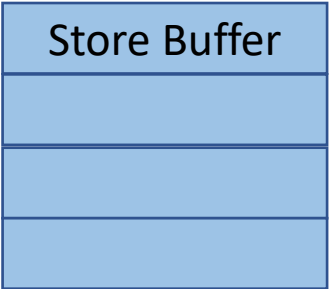
- We'd like to be able to compile atomic instructions just to regular ISA loads and stores

Thread 0:

mov [x], 1

mov %t0, [y]

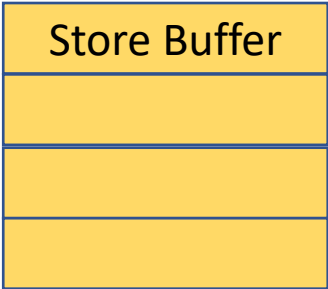
Core 0



Thread 1:

mov [y], 1

mov %t1, [x]

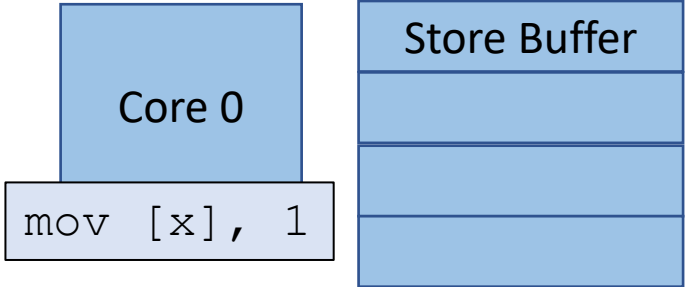


Core 1



Thread 0:

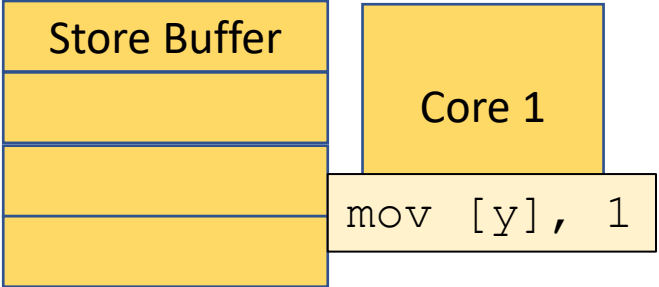
mov %t0, [y]



execute first instruction

Thread 1:

mov %t1, [x]



Thread 0:

mov %t0, [y]

Core 0

Store Buffer
x:1

values get stored in SB

Thread 1:

mov %t1, [x]

Store Buffer
y:1

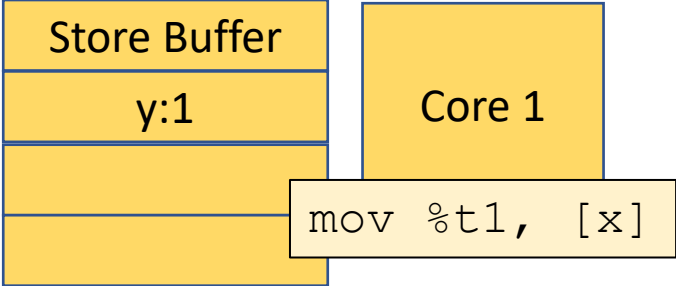
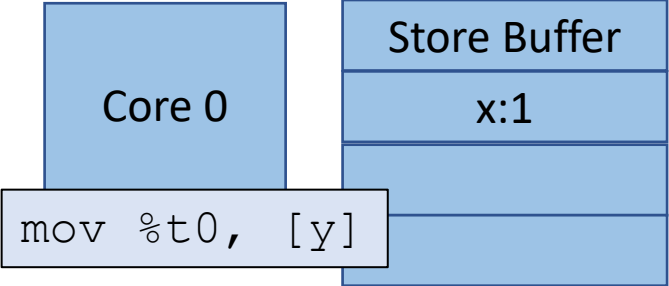
Core 1

x:0	Main Memory
y:0	

Thread 0:

Thread 1:

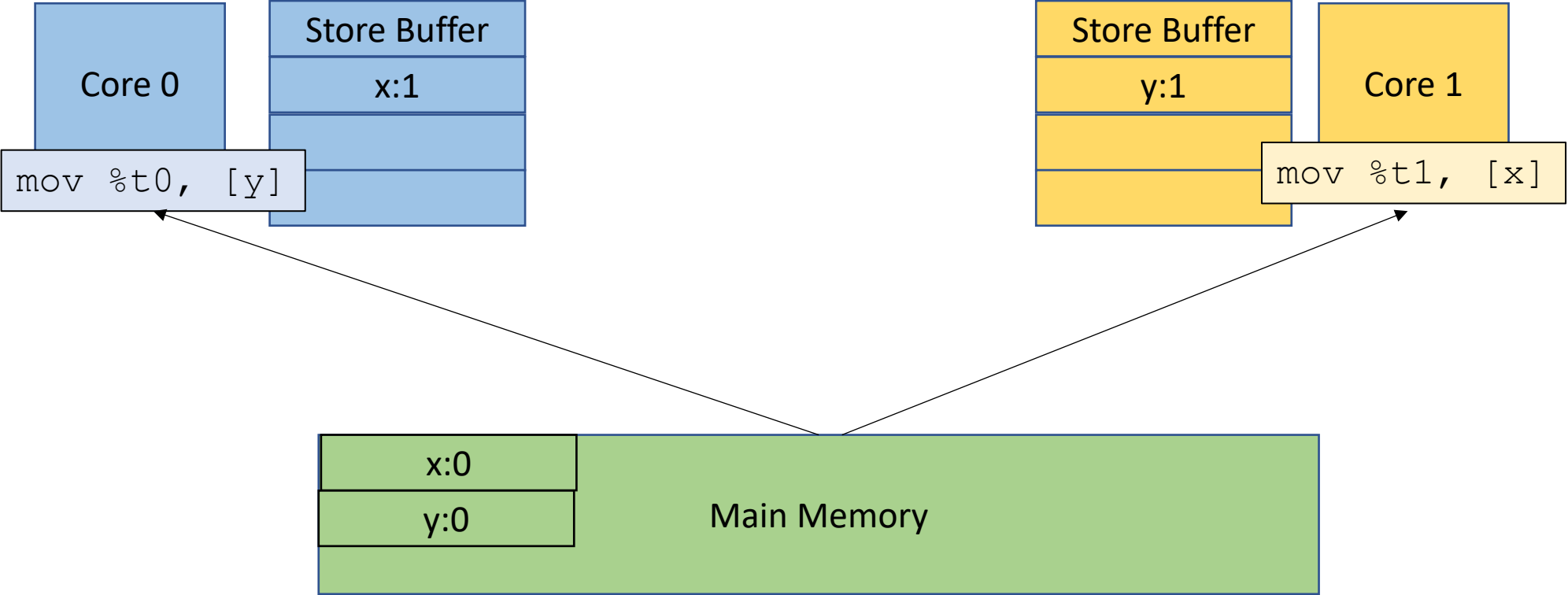
Execute next instruction



Thread 0:

Thread 1:

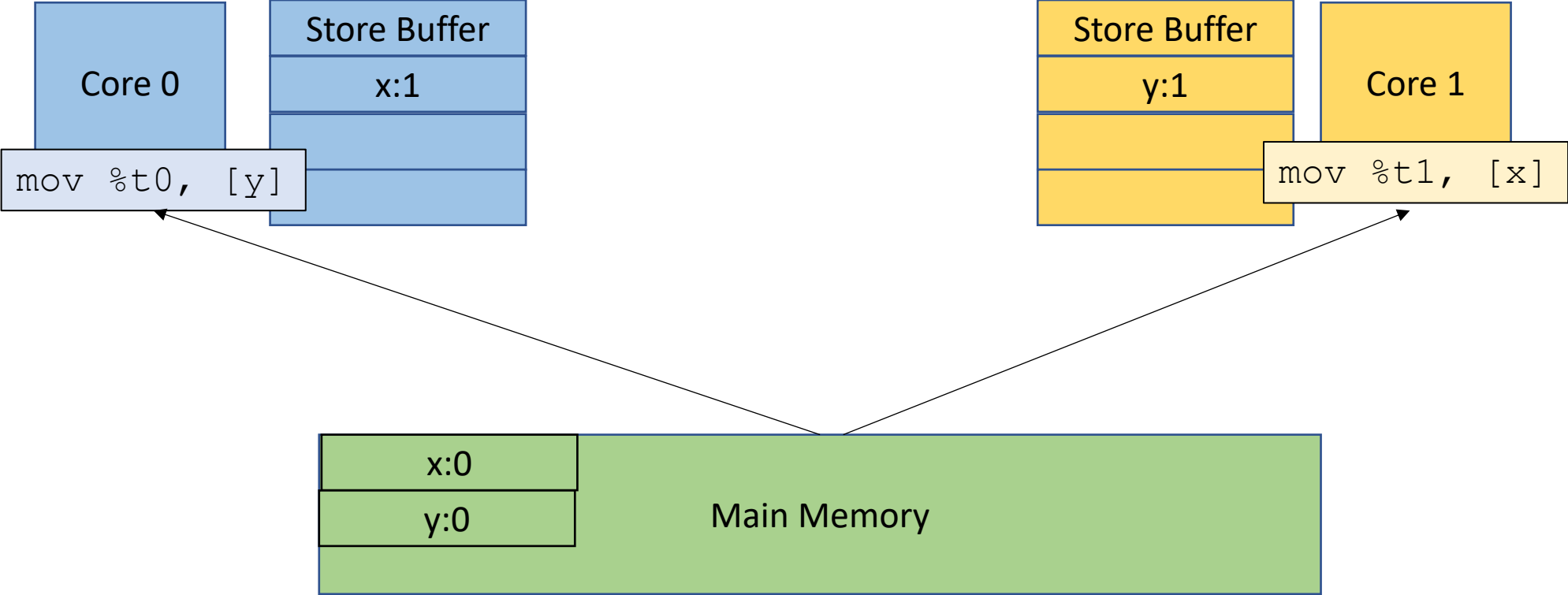
Values get loaded from memory



Thread 0:

Thread 1:

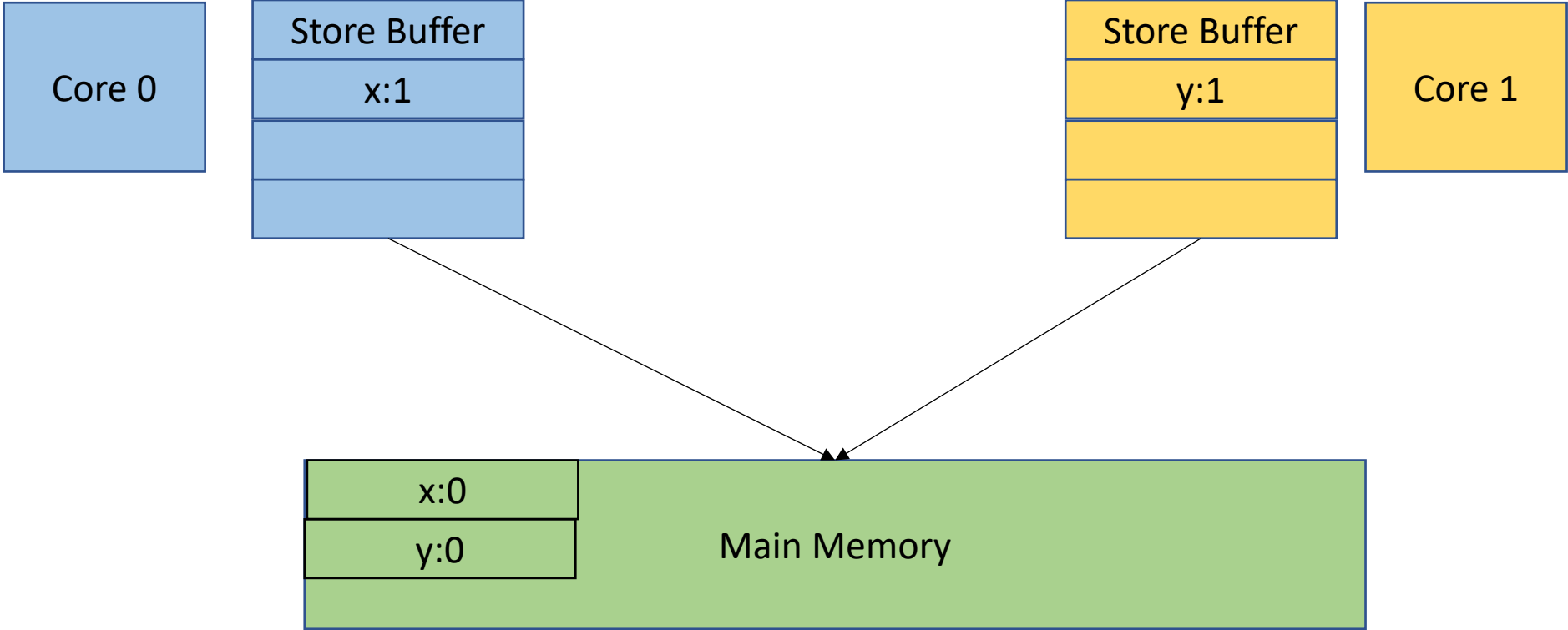
we see `t0 == t1 == 0!`



Thread 0:

Thread 1:

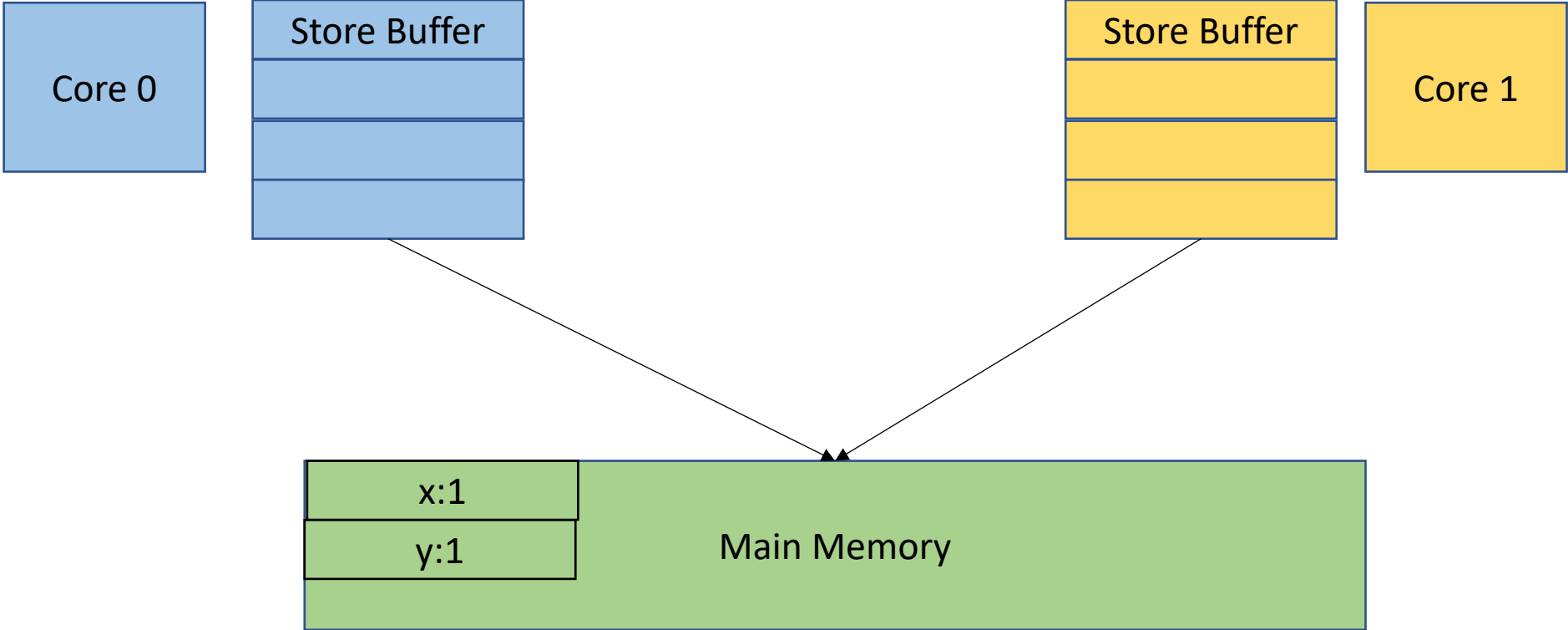
Store buffers are drained eventually



Thread 0:

Thread 1:

Store buffers are drained eventually
but we've already done our loads



Restoring sequential consistency

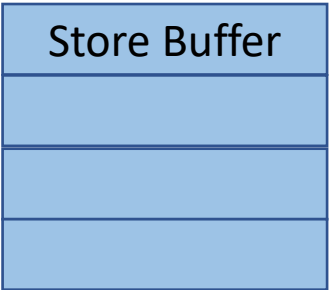
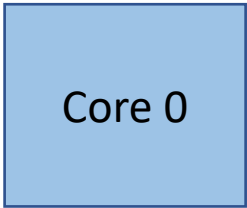
- It is typical that relaxed memory models provide special instructions which can be used to disallow weak behaviors.
- These instructions are called Fences
- The X86 fence is called `mfence`. It flushes the store buffer.

Thread 0:

mov [x], 1

mfence

mov %t0, [y]

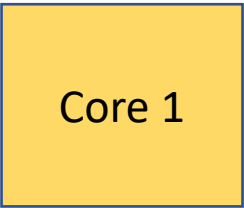
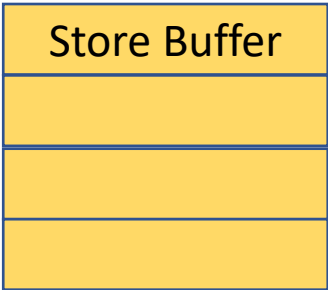


Thread 1:

mov [y], 1

mfence

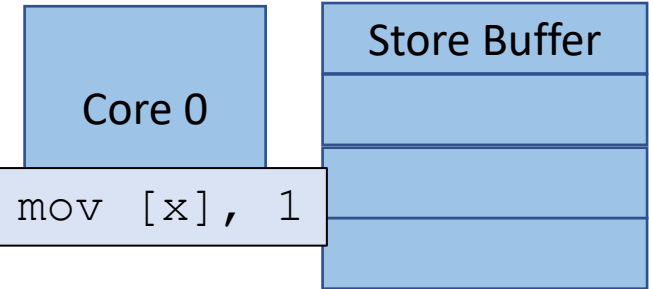
mov %t1, [x]



Thread 0:

mfence

mov %t0, [y]

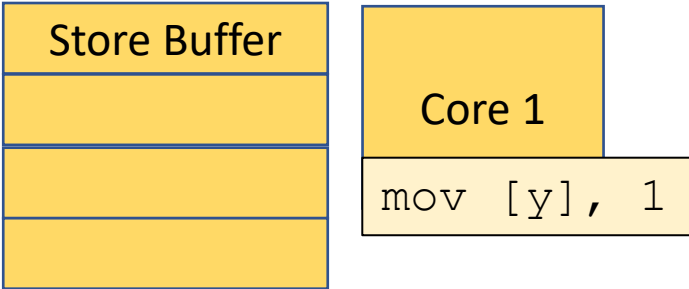


Execute first instruction

Thread 1:

mfence

mov %t1, [x]



Thread 0:

mfence

mov %t0, [y]

Core 0

Store Buffer
x:1

Thread 1:

mfence

mov %t1, [x]

Core 1

Store Buffer
y:1

Values go into the store buffer

x:0	Main Memory
y:0	

Thread 0:

Thread 1:

Execute next instruction

mov %t0, [y]

mov %t1, [x]

Core 0

mfence

Store Buffer

x:1

Store Buffer

y:1

Core 1

mfence

x:0

y:0

Main Memory

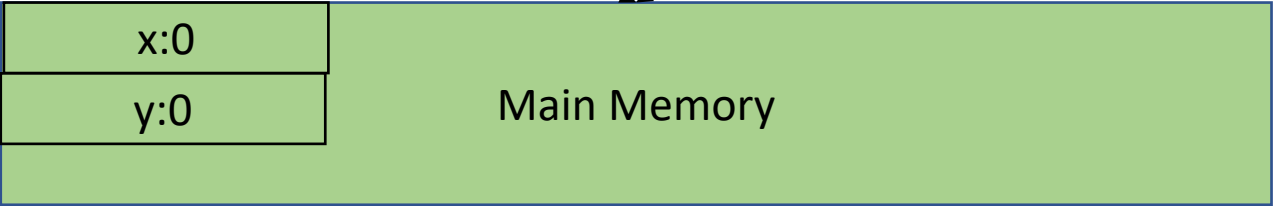
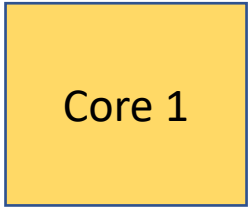
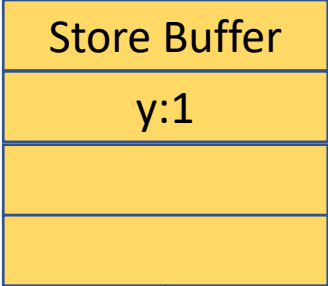
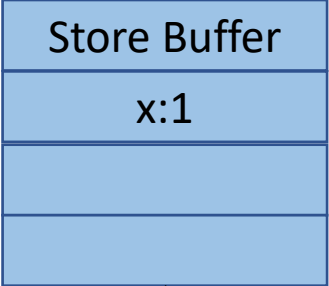
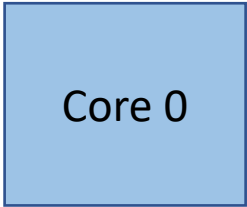
Thread 0:

Thread 1:

store buffers are flushed

```
mov %t0, [y]
```

```
mov %t1, [x]
```



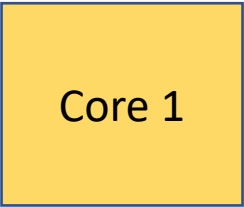
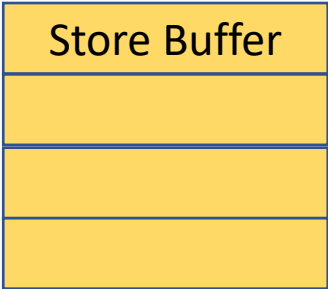
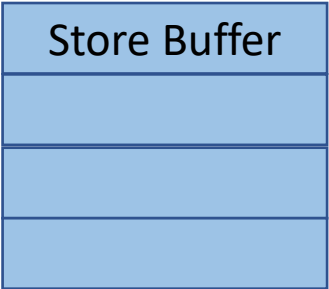
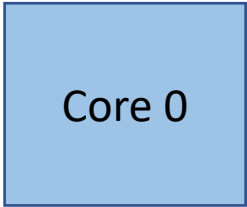
Thread 0:

Thread 1:

store buffers are flushed

```
mov %t0, [y]
```

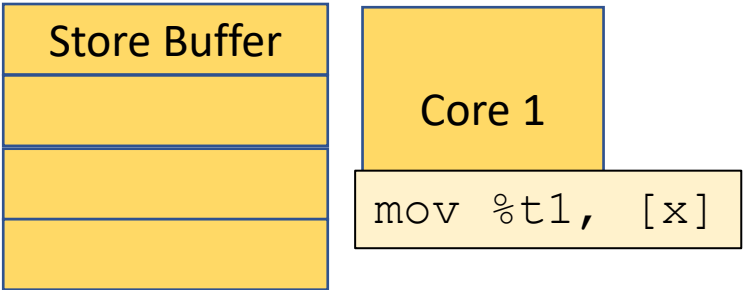
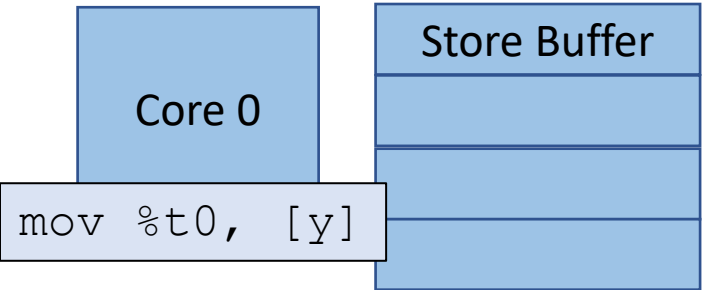
```
mov %t1, [x]
```



Thread 0:

Thread 1:

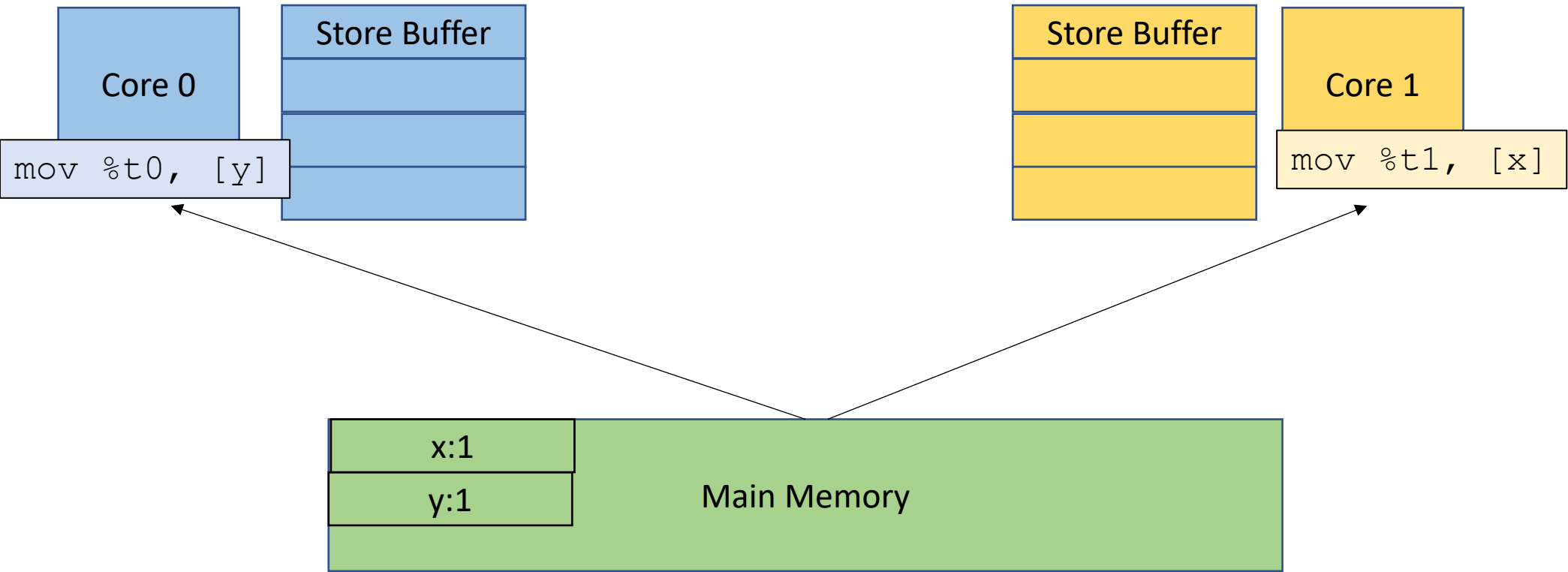
execute next instruction



Thread 0:

Thread 1:

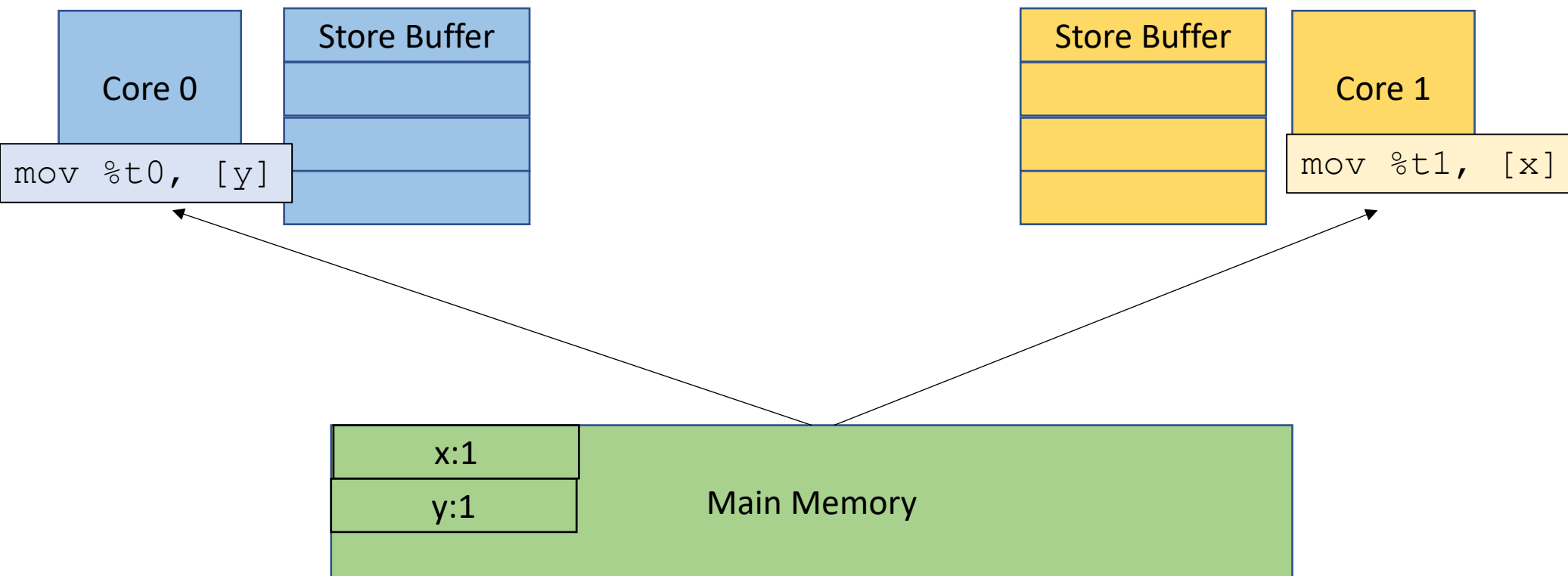
values are loaded from memory



Thread 0:

Thread 1:

We don't get the problematic behavior: $t0 == t1 == 0$



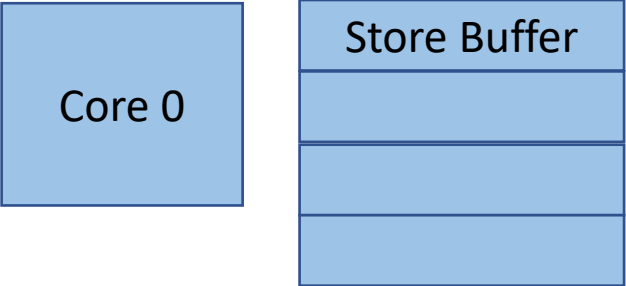
Next example

Thread 0:

mov [x], 1

mov %t0, [x]

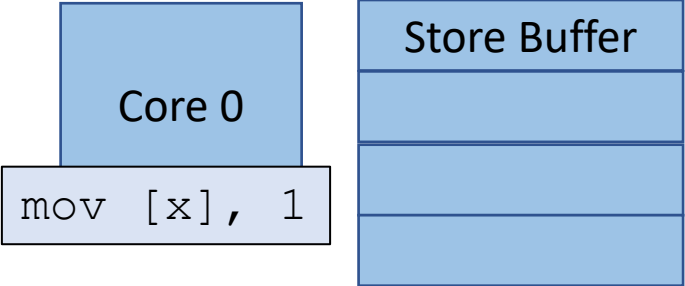
How does this execute?



Thread 0:

execute first instruction

mov %t0, [x]



Thread 0:

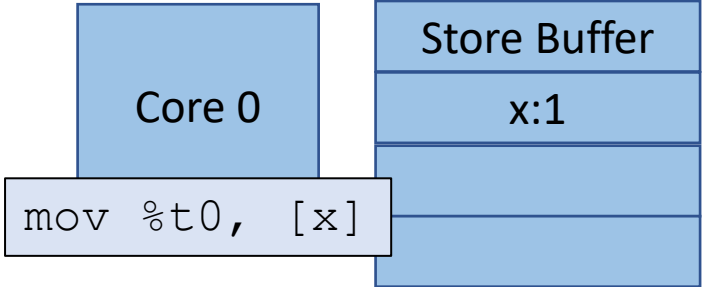
Store the value in the store buffer

```
mov %t0, [x]
```



Thread 0:

Next instruction

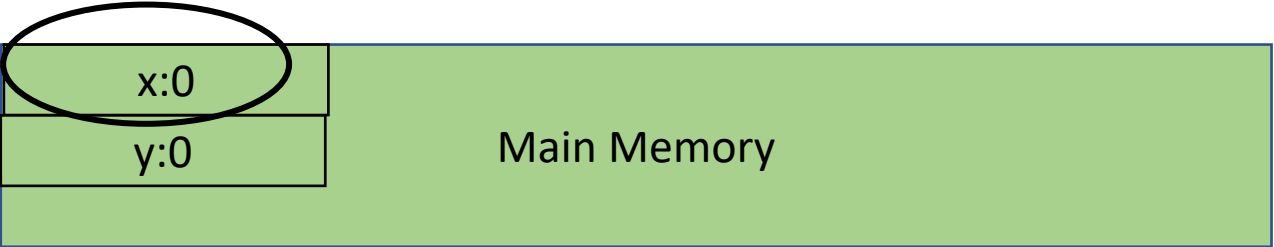
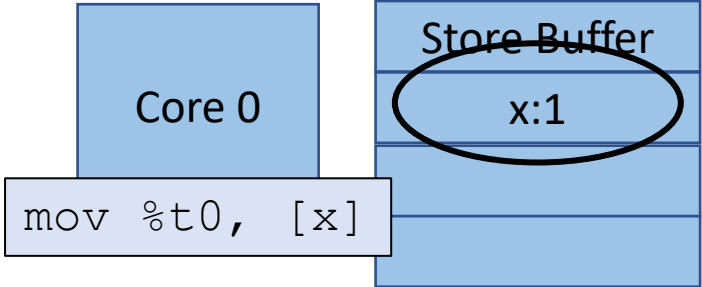


Thread 0:

Where to load??

Store buffer?

Main memory?

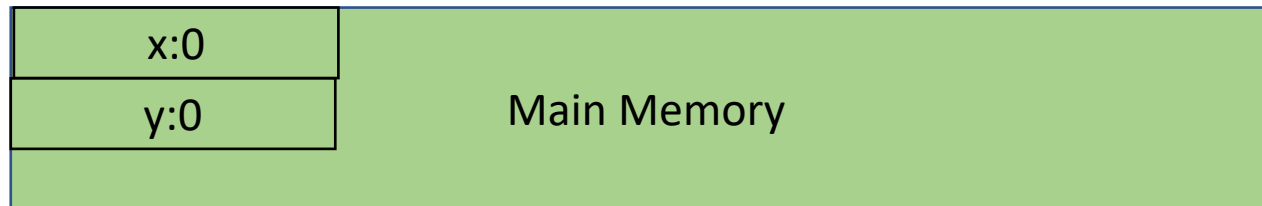
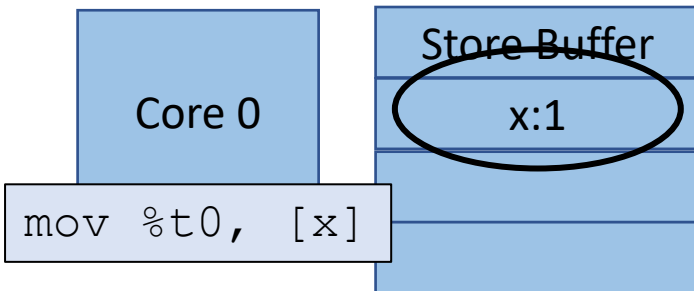


Thread 0:

Where to load??

Threads check store buffer before going to main memory

It is close and cheap to check.



Memory Consistency

- How to specify a relaxed memory model?
- We can do it operationally
 - by constructing a high-level machine and reasoning about operations through the machine.
 - or we can talk about instructions that are allowed to "break" program order.

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test

Can `t0 == t1 == 0`?

Thread 0:

```
mov [x], 1  
mov %t0, [y]
```

Thread 1:

```
mov [y], 1  
mov %t1, [x]
```



We will annotate instructions with S for store, and L for loads

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test

Can `t0 == t1 == 0`?

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```



We will annotate instructions with S for store, and L for loads

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test

Can `t0 == t1 == 0`?

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

```
S:mov [x], 1
```

```
L:mov %t0, [y]
```

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```

```
S:mov [y], 1
```

```
L:mov %t1, [x]
```



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test

Can `t0 == t1 == 0`?

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```

L:mov %t1, [x]

S:mov [x], 1

L:mov %t0, [y]

S:mov [y], 1

Now we make a new rule:

S(tores) followed by a L(oad)
do not have to follow program order

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test

Can `t0 == t1 == 0`?

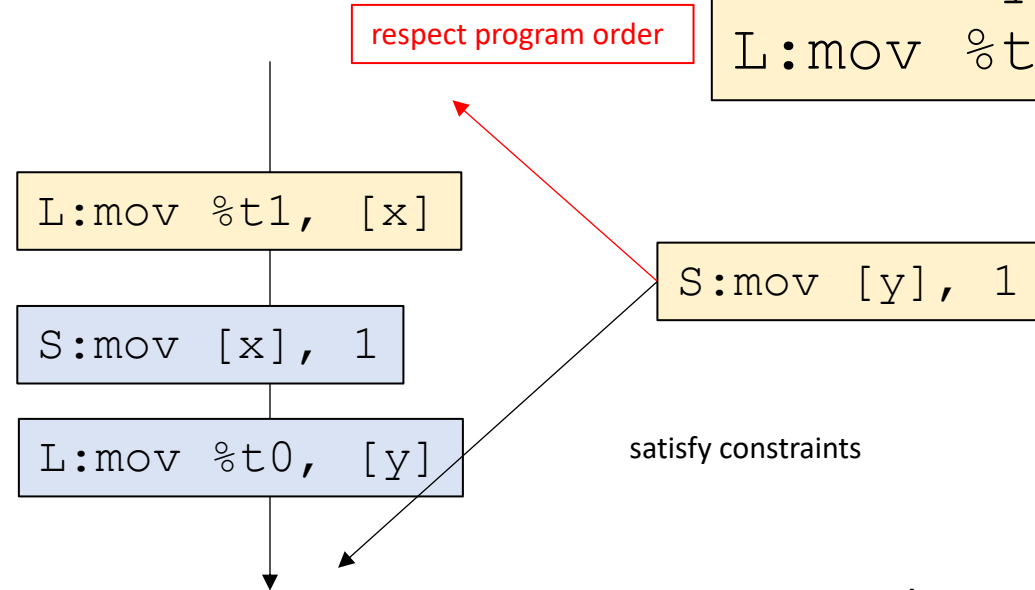
Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```

we can ignore this condition!!



Now we make a new rule:

S(tores) followed by a L(oad)
do not have to follow program order

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test

Can `t0 == t1 == 0`?

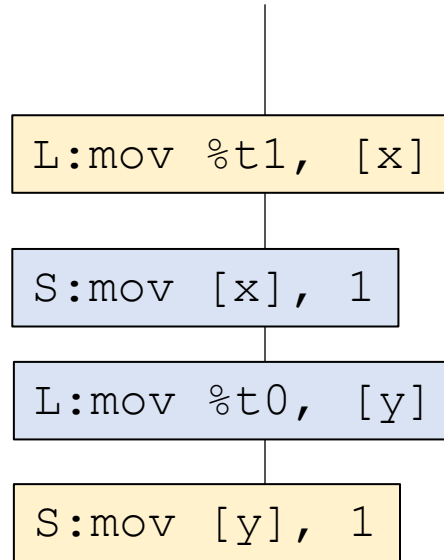
Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```

we can ignore this condition!!



Now we can satisfy the condition!

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

Lets peak under the hood here

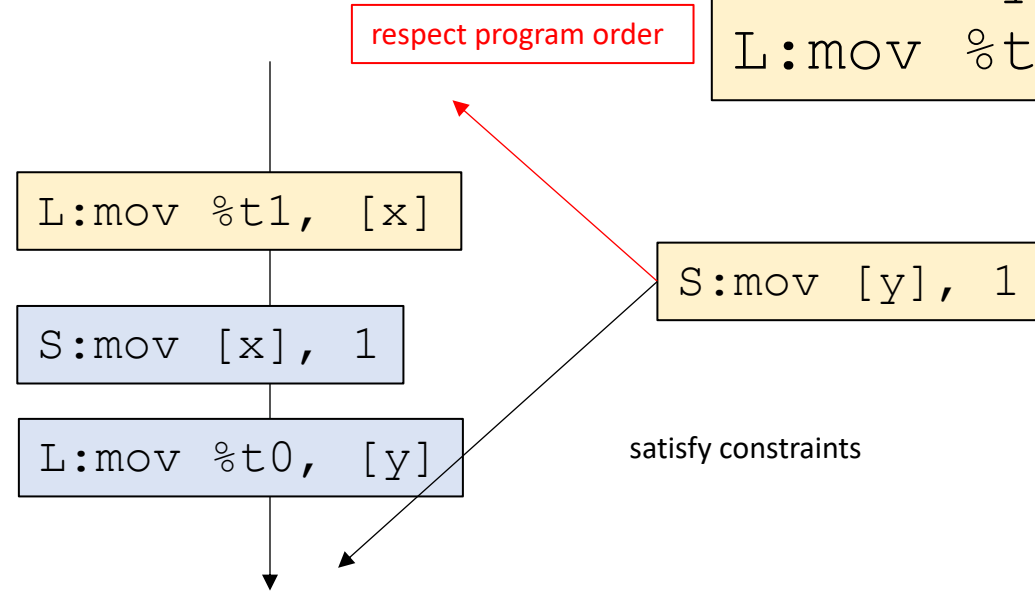
Another test

Can `t0 == t1 == 0`?

Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```

we can ignore this condition!!



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

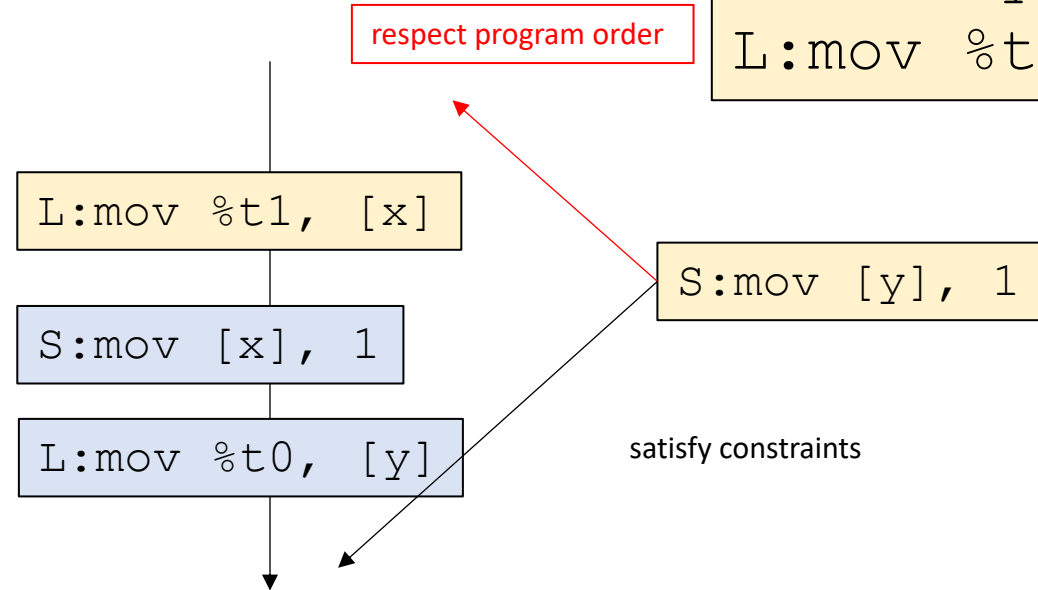
Lets peak under the hood here

Global timeline is when the
Store operation becomes visible
to other threads

Another test

Can `t0 == t1 == 0`?

we can ignore this condition!!



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

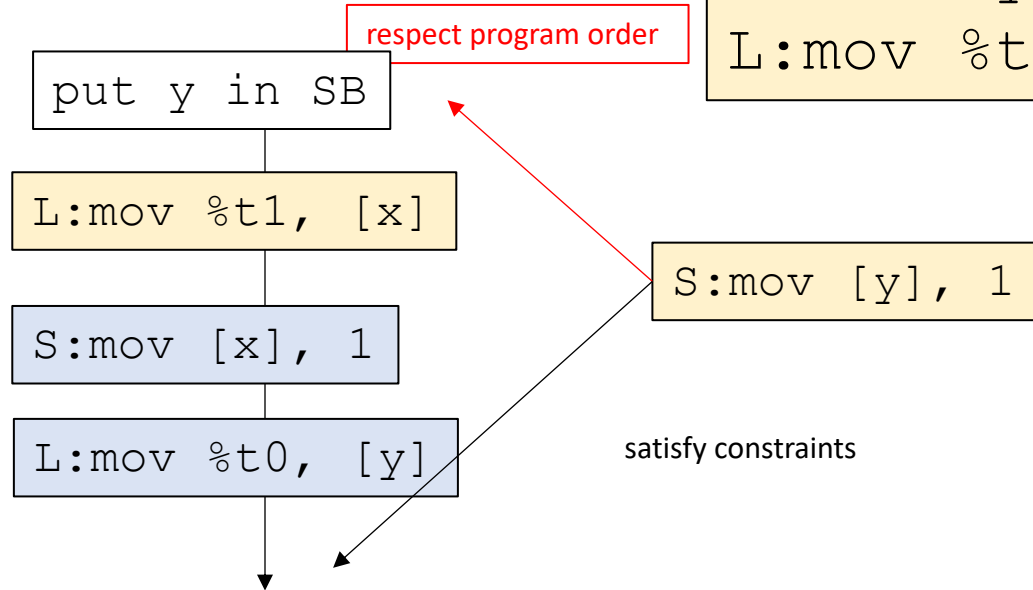
Lets peak under the hood here

Global timeline is when the
Store operation becomes visible
to other threads

Another test

Can `t0 == t1 == 0`?

we can ignore this condition!!



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [y]
```

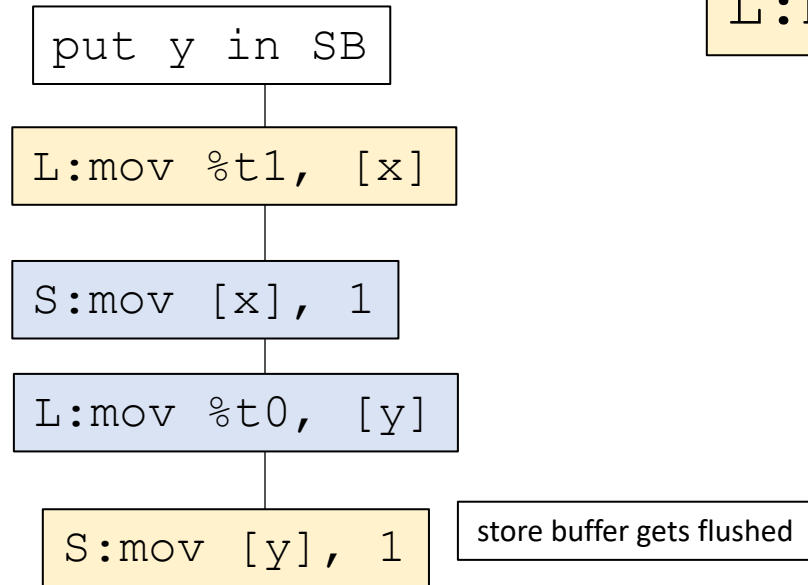
Lets peak under the hood here

Global timeline is when the
Store operation becomes visible
to other threads

Another test

Can `t0 == t1 == 0`?

we can ignore this condition!!



Thread 1:

```
S:mov [y], 1  
L:mov %t1, [x]
```

Questions

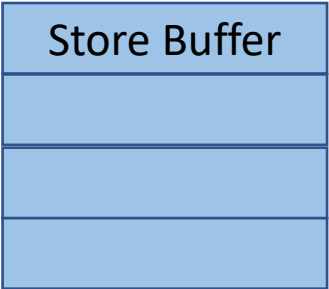
- Can stores be reordered with stores?

Thread 0:

mov [x], 1

mov [y], 1

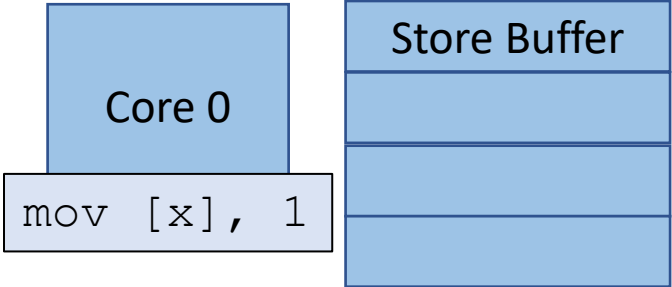
Core 0



Thread 0:

mov [y], 1

execute the first instruction



Thread 0:

mov [y], 1

value goes into store buffer

Core 0

Store Buffer
x:1

x:0	Main Memory
y:0	

Thread 0:

mov [y], 1

execute next instruction

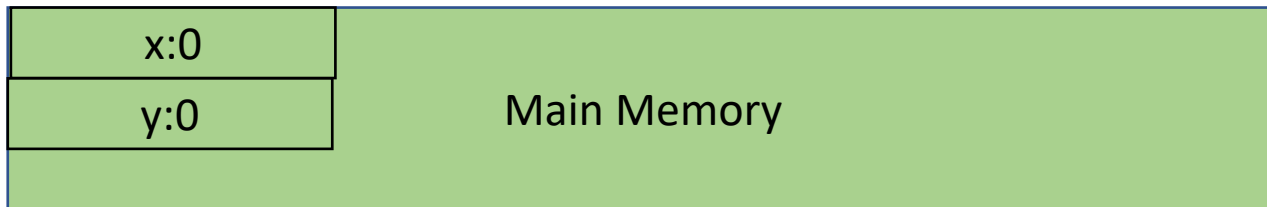
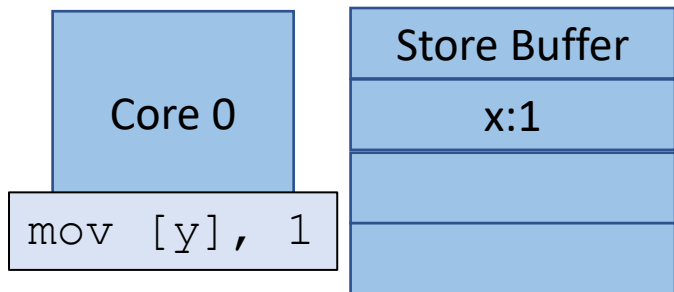
Core 0

Store Buffer
x:1

x:0	Main Memory
y:0	

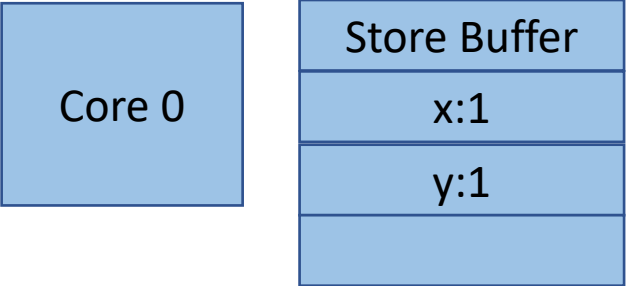
Thread 0:

execute next instruction



Thread 0:

value goes into the store buffer



Thread 0:

Core 0

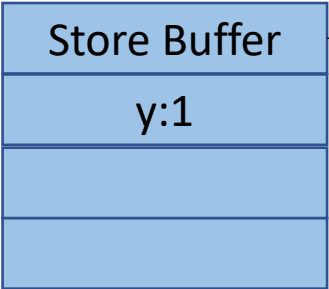
Store Buffer
x:1
y:1

On x86, the store buffer trains in a FIFO way:
thus stores cannot be reordered

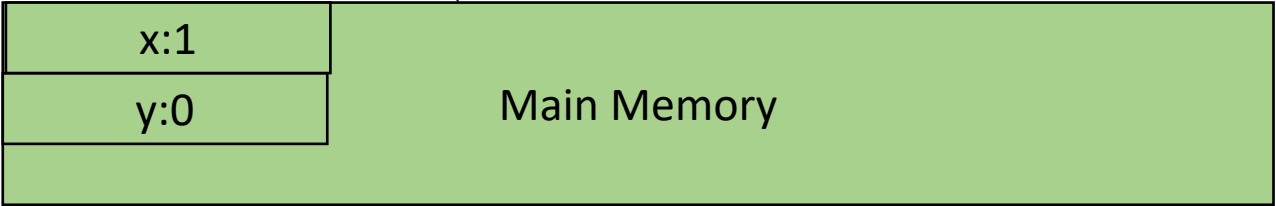
x:0	Main Memory
y:0	

Thread 0:

Core 0

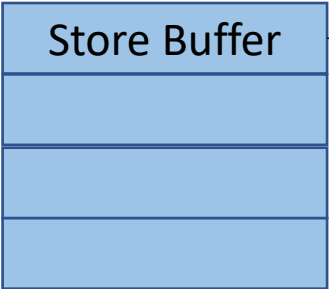


On x86, the store buffer trains in a FIFO way:
thus stores cannot be reordered

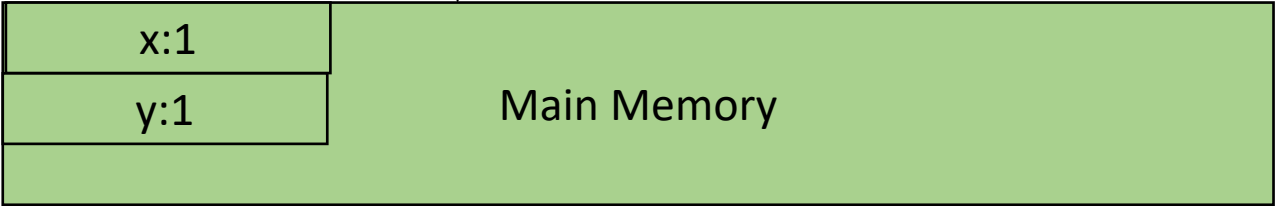


Thread 0:

Core 0



On x86, the store buffer trains in a FIFO way:
thus stores cannot be reordered



Questions

- Can stores be reordered with stores?
- How do we make rules about mfence?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
S:mov [x], 1  
mfence  
L:mov %t0, [y]
```

```
S:mov [x], 1
```

```
mfence
```

```
L:mov %t0, [y]
```

Another test

Can `t0 == t1 == 0`?

Thread 1:

```
S:mov [y], 1  
mfence  
L:mov %t1, [x]
```

```
S:mov [y], 1
```

```
mfence
```

```
L:mov %t1, [x]
```

Rules: S(tores) followed by a L(oad)
do not have to follow program order.

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

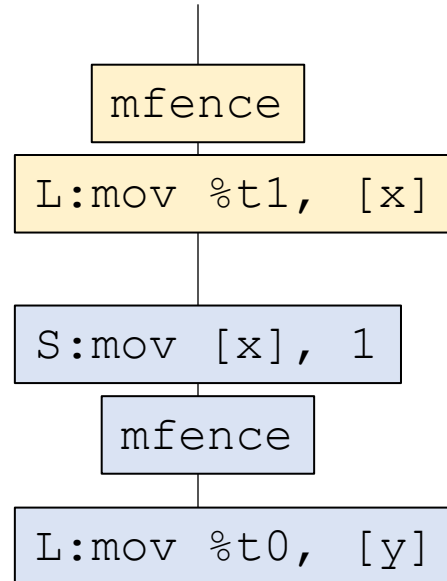
Thread 0:

```
S:mov [x], 1  
mfence  
L:mov %t0, [y]
```

*So we can't
reorder
this instruction
at all!*

Another test

Can `t0 == t1 == 0`?



Thread 1:

```
S:mov [y], 1  
mfence  
L:mov %t1, [x]
```

```
S:mov [y], 1
```

Rules:

S(tores) followed by a L(oad)
do not have to follow program order.

S(tores) cannot be reordered past a fence
in program order

Rules

- Are we done?

Rules:

S(tores) followed by a L(oad)

do not have to follow program order.

S(tores) cannot be reordered past a fence
in program order

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Another test

Can `t0 == 0`?

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [x]
```

```
S:mov [x], 1
```

```
L:mov %t0, [x]
```



Rules:

S(tores) followed by a L(oad)

do not have to follow program order.

S(tores) cannot be reordered past a fence
in program order

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [x]
```

Another test
Can `t0 == 0`?

S:mov [x], 1

where to put this store?

L:mov %t0, [x]

Rules:
S(tores) followed by a L(oad)
do not have to follow program order.

S(tores) cannot be reordered past a fence
in program order

Global variable:

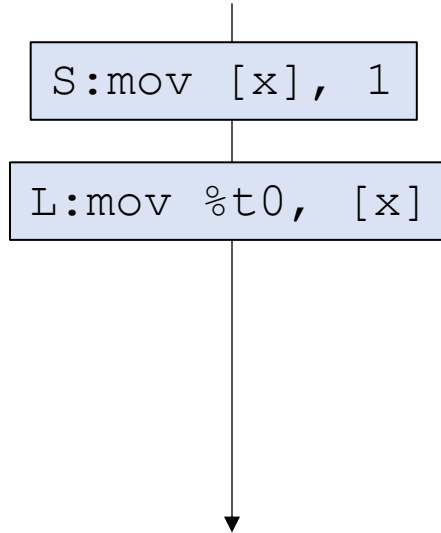
```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:

```
S:mov [x], 1  
L:mov %t0, [x]
```

Another test

Can `t0 == 0`?



where to put this store?

Rules:

S(tores) followed by a L(oad)
do not have to follow program order.

S(tores) cannot be reordered past a fence
in program order

S(tores) cannot be reordered past L(oads)
from the same address

TSO - Total Store Order

Rules:

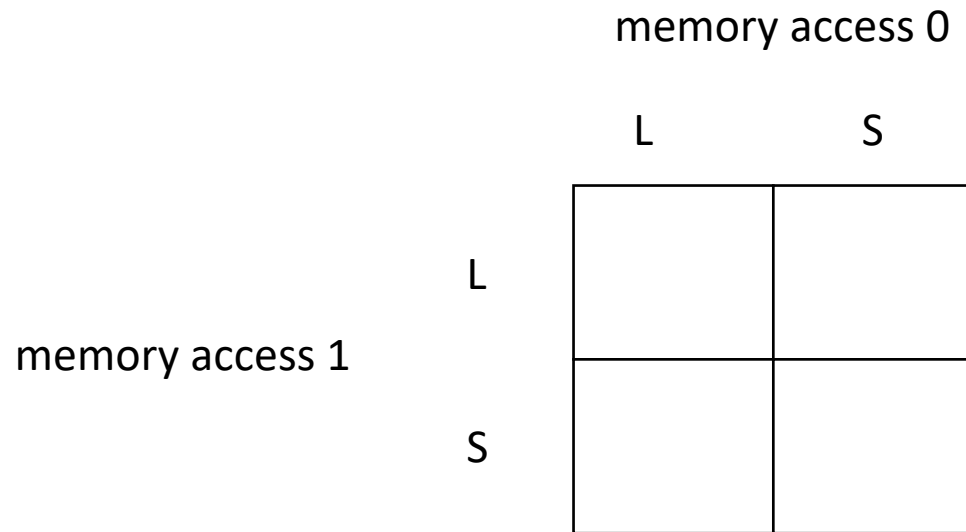
S(tores) followed by a L(oad)
do not have to follow program order.

S(tores) cannot be reordered past a fence
in program order

S(tores) cannot be reordered past L(oads)
from the same address

Other memory models?

- We can specify them in terms of what reorderings are allowed



If memory access 0 appears before memory access 1 in program order, can it bypass program order?

Other memory models?

- We can specify them in terms of what reorderings are allowed

		memory access 0	
		L	S
memory access 1	L	NO	NO
	S	NO	NO

Sequential Consistency

If memory access 0 appears before memory access 1 in program order, can it bypass program order?

Other memory models?

- We can specify them in terms of what reorderings are allowed

		memory access 0	
		L	S
memory access 1	L	NO	Different address
	S	NO	NO

TSO - total store order

If memory access 0 appears before memory access 1 in program order, can it bypass program order?

Other memory models?

- We can specify them in terms of what reorderings are allowed

		memory access 0	
		L	S
memory access 1	L	?	?
	S	?	?

Weaker models?

If memory access 0 appears before memory access 1 in program order, can it bypass program order?

Other memory models?

- We can specify them in terms of what reorderings are allowed

		memory access 0	
		L	S
memory access 1	L	NO	Different address
	S	NO	Different address

PSO - partial store order

If memory access 0 appears before memory access 1 in program order, can it bypass program order?

Allows stores to drain from the store buffer in any order

Other memory models?

- We can specify them in terms of what reorderings are allowed

		memory access 0	
		L	S
memory access 1	L	YES	Different address
	S	Different address	Different address

RMO - Relaxed Memory Order

If memory access 0 appears before memory access 1 in program order, can it bypass program order?

Very relaxed model!

Other memory models?

- FENCE: can always restore order using fences. Accesses cannot be reordered past fences!

		memory access 0	
		L	S
memory access 1	L	NO	NO
	S	NO	NO

Any Memory Model

If memory access 0 appears before memory access 1 in program order, and there is a FENCE between the two accesses, can it bypass program order?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

First thing: change our syntax to pseudo code

Thread 0:

```
L:mov %t0, [y]  
S:mov [x], 1
```

Thread 1:

```
L:mov %t1, [x]  
S:mov [y], 1
```

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

First thing: change our syntax to pseudo code
You should be able to find natural mappings
to any ISA

Thread 0:

```
L:%t0 = load(y)  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

Thread 0:

```
L:%t0 = load(y)  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

Get out our lego bricks and try for sequential consistency

Thread 0:

```
L:%t0 = load(y)  
S:store(x,1)
```

```
L:%t0 = load(y)
```

```
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

```
L:%t1 = load(x)
```

```
S:store(y,1)
```



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can $t0 == t1 == 1$?

Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
S:store(x,1)
```

```
L:%t0 = load(y)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y,1)
```

respect program order

```
S:store(x,1)
```

```
L:%t1 = load(x)
```

```
S:store(y,1)
```

satisfy constraints

Not allowed under sequential consistency!

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

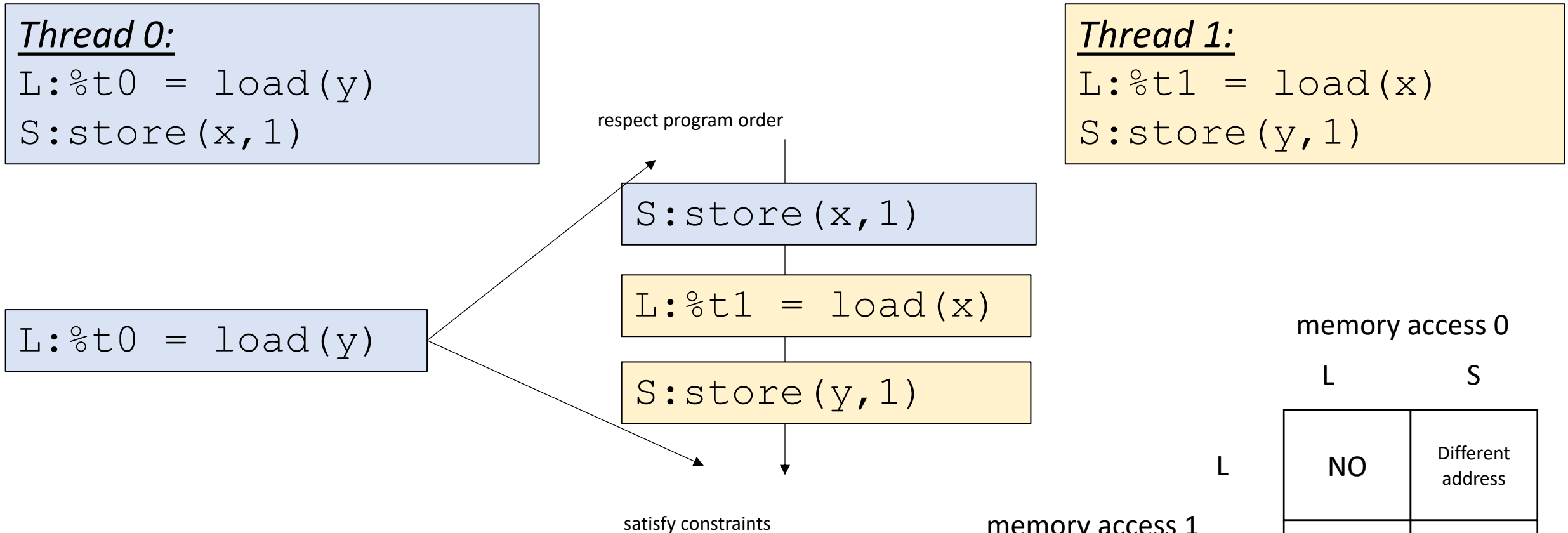
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
S:store(x, 1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y, 1)
```



What about TSO?

memory access 0	
L	S
L	NO
S	NO

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

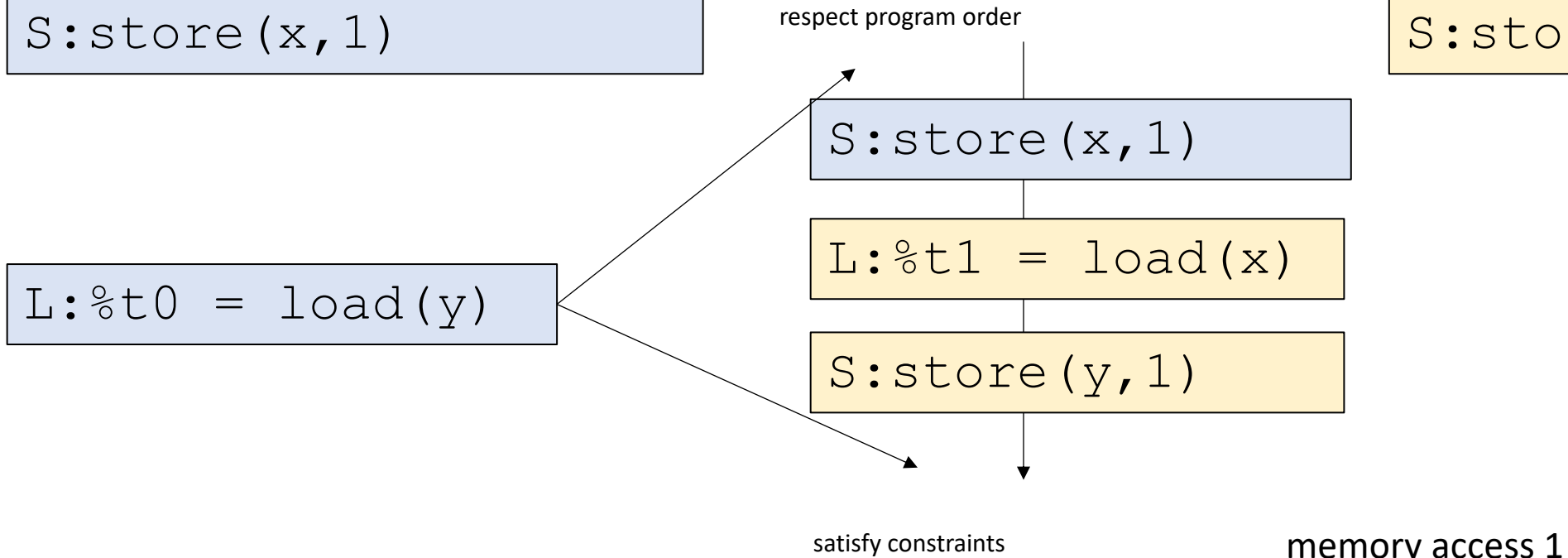
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
S:store(x, 1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y, 1)
```



What about TSO? NOT ALLOWED!

memory access 0	
L	S
L	NO
S	NO

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

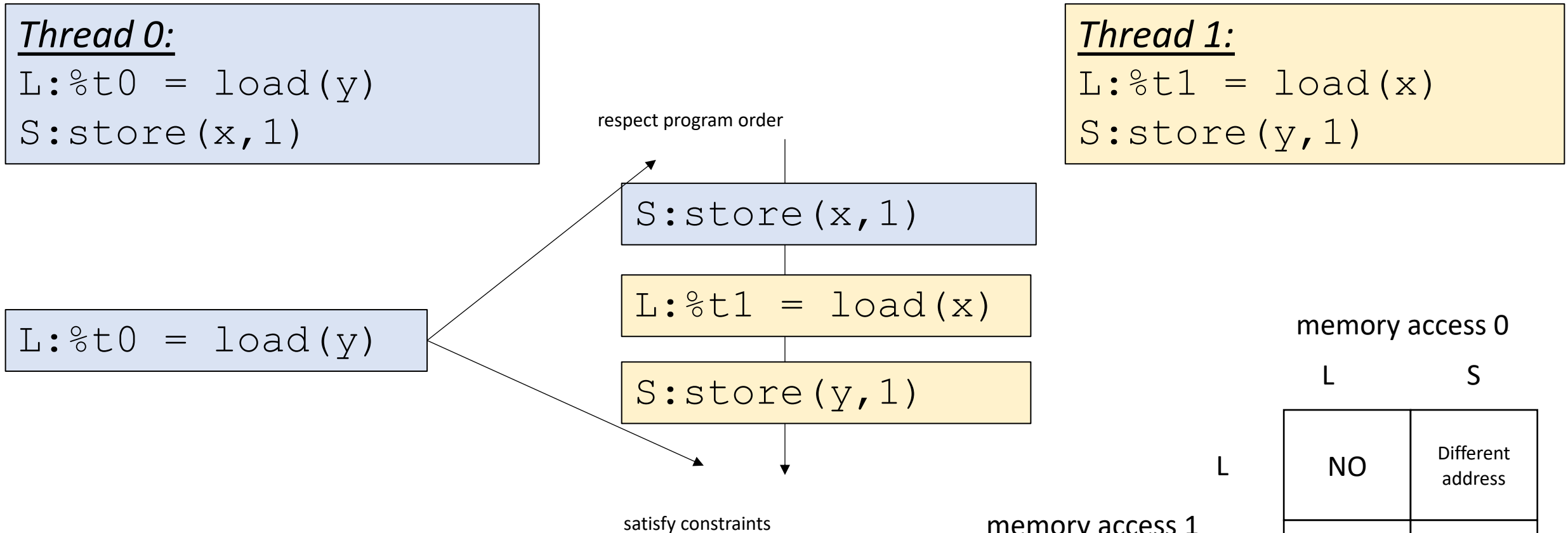
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
S:store(x, 1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y, 1)
```



What about PSO?

memory access 0	
L	S
L NO	Different address
S NO	Different address

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

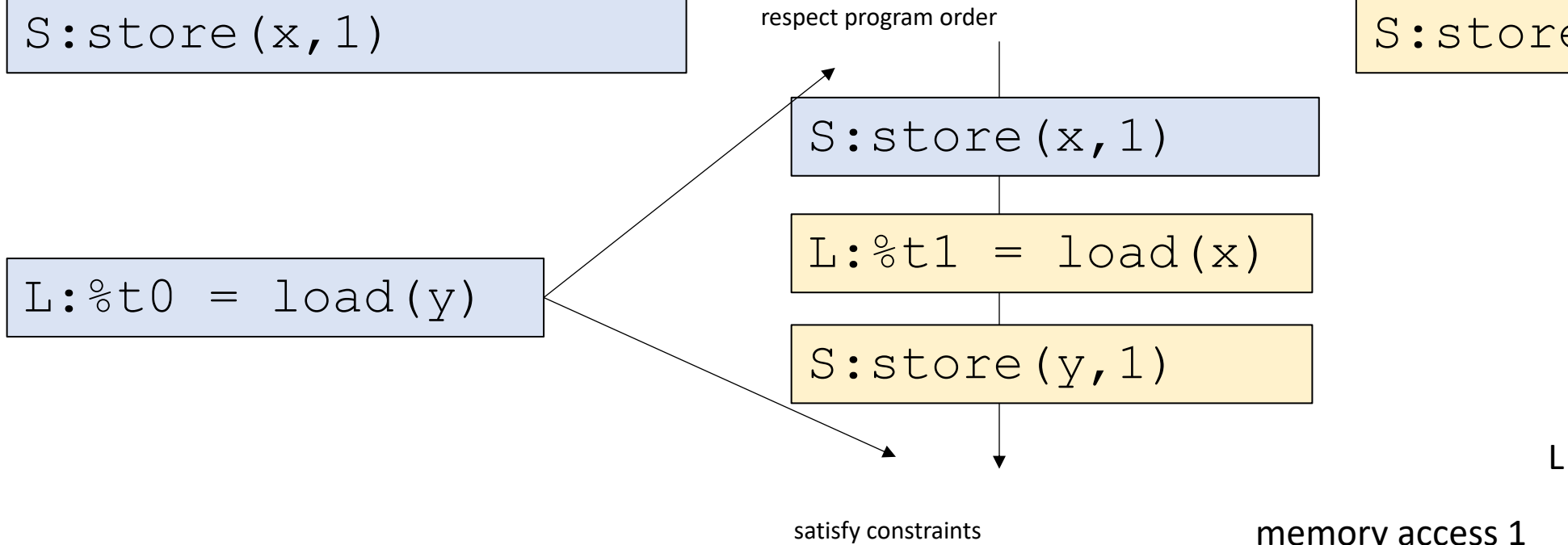
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
S:store(x, 1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y, 1)
```



What about PSO? NO!

memory access 0	
L	S
L NO	Different address
S NO	Different address

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

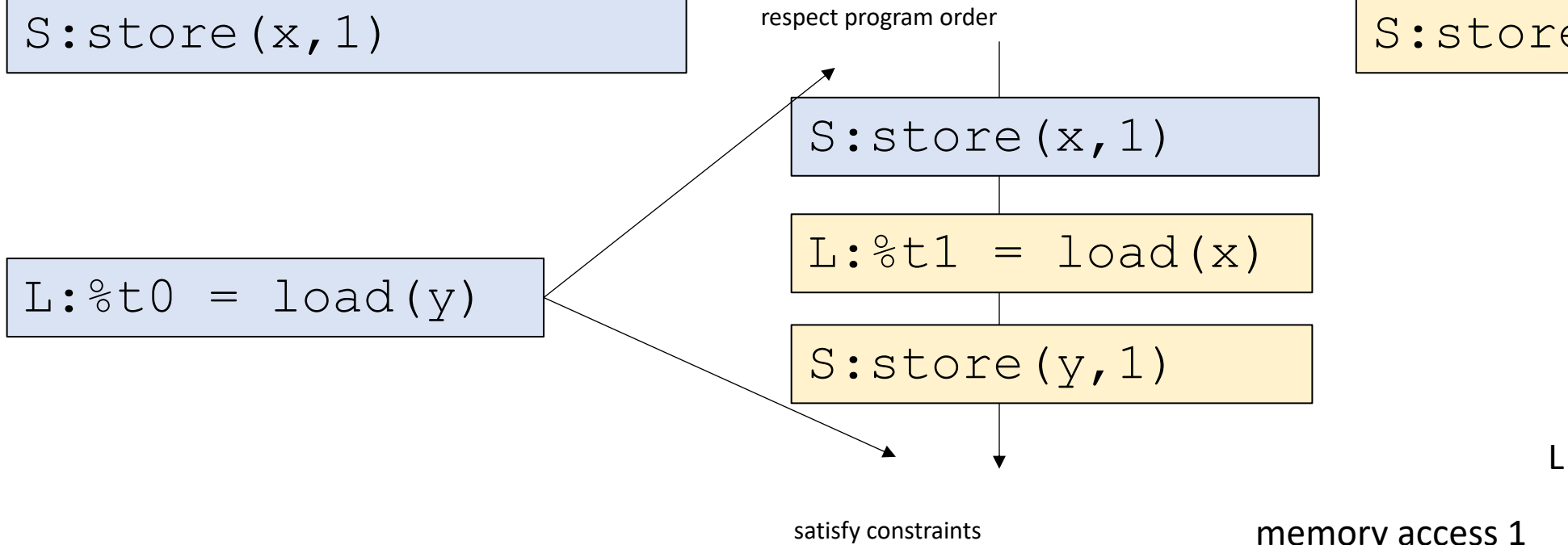
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
S:store(x, 1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y, 1)
```



What about RMO?

memory access 0		
	L	S
L	YES	Different address
S	different address	Different address

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
S:store(x, 1)
```

```
L:%t0 = load(y)
```

respect program order

```
S:store(x, 1)
```

```
L:%t1 = load(x)
```

```
S:store(y, 1)
```

satisfy constraints

Thread 1:

```
L:%t1 = load(x)  
S:store(y, 1)
```

memory access 0

L S

L

YES

Different
address

S

different
address

Different
address

memory access 1

What about RMO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

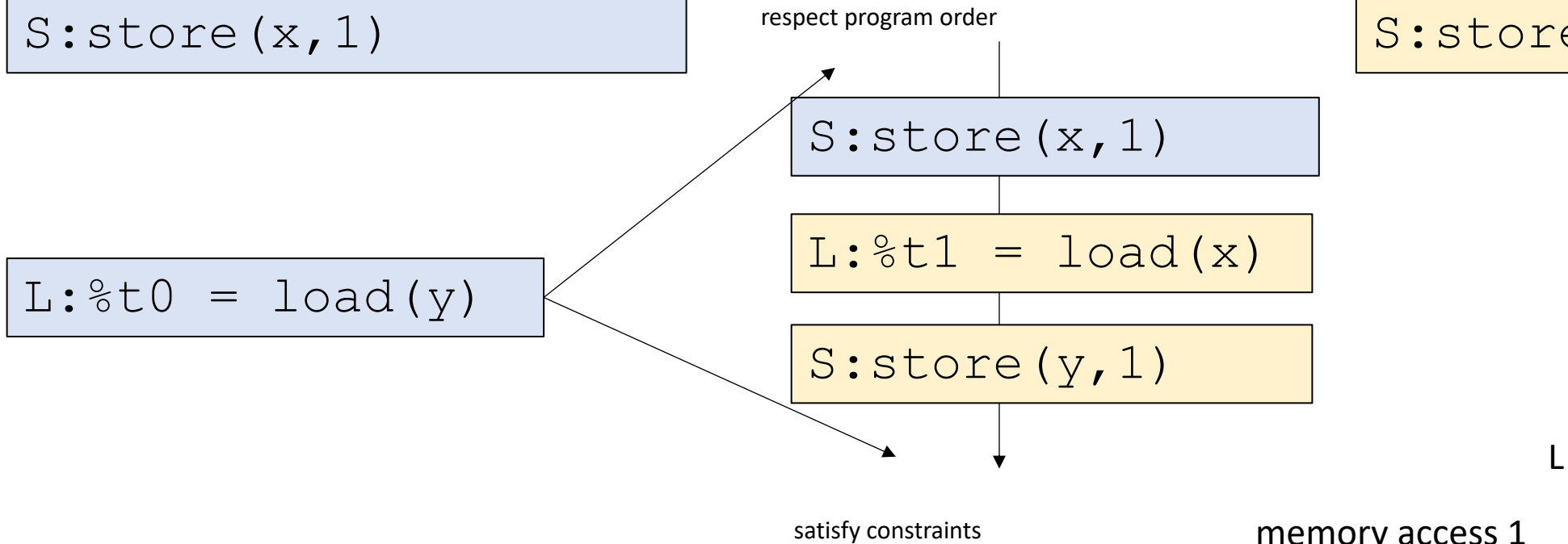
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
S:store(x, 1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y, 1)
```



What about RMO? YES!

memory access 0		
	L	S
L	YES	Different address
S	different address	Different address

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

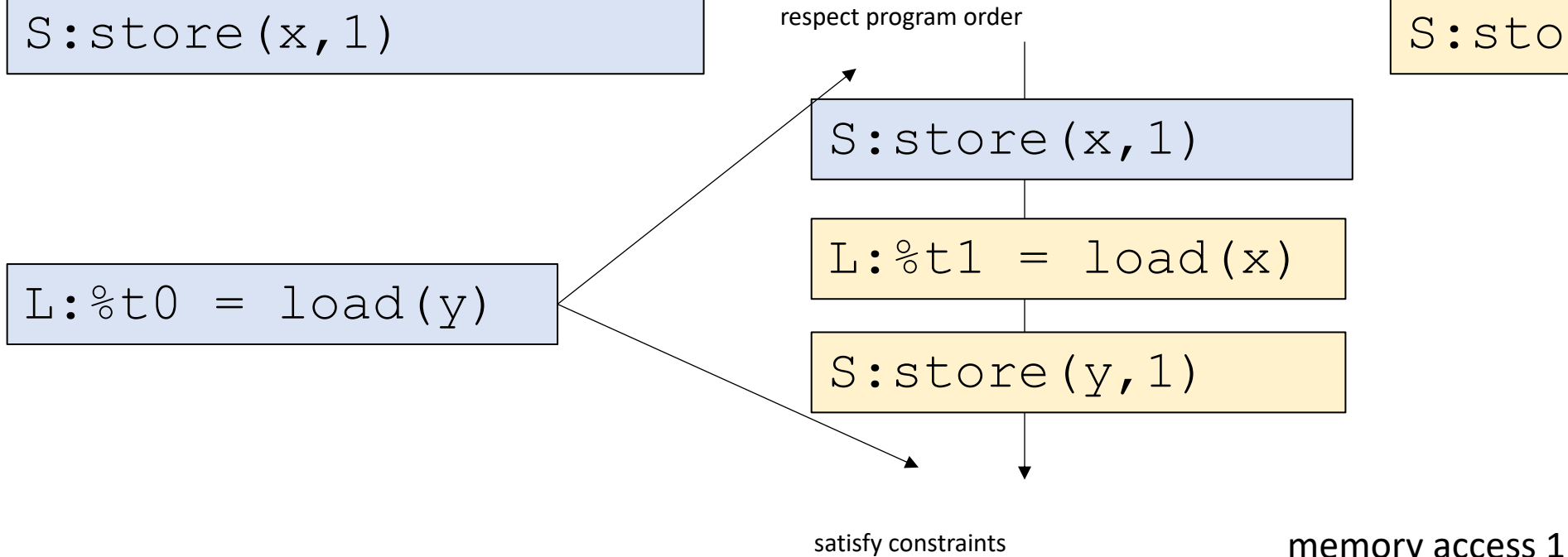
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
S:store(x, 1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y, 1)
```



How do we disallow the behavior in RMO?

memory access 0			
	L		S
L	YES		Different address
S	different address		Different address

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
fence  
S:store(x, 1)
```

```
L:%t0 = load(y)
```

respect program order

```
S:store(x, 1)
```

```
L:%t1 = load(x)
```

```
S:store(y, 1)
```

satisfy constraints

Thread 1:

```
L:%t1 = load(x)  
S:store(y, 1)
```

memory access 0

L S

L

YES

Different
address

S

different
address

Different
address

memory access 1

How do we disallow the behavior in RMO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

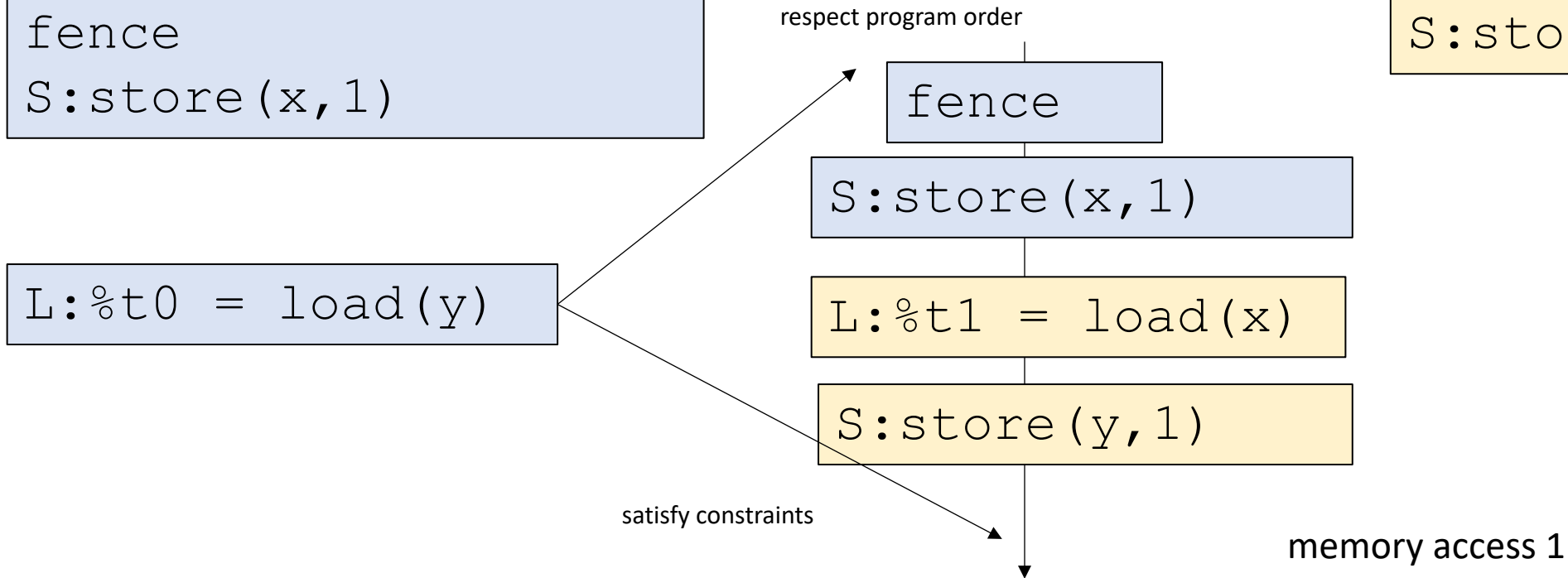
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
fence  
S:store(x, 1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y, 1)
```



How do we disallow the behavior in RMO?

memory access 0

	L	S
L	YES	Different address
S	different address	Different address

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

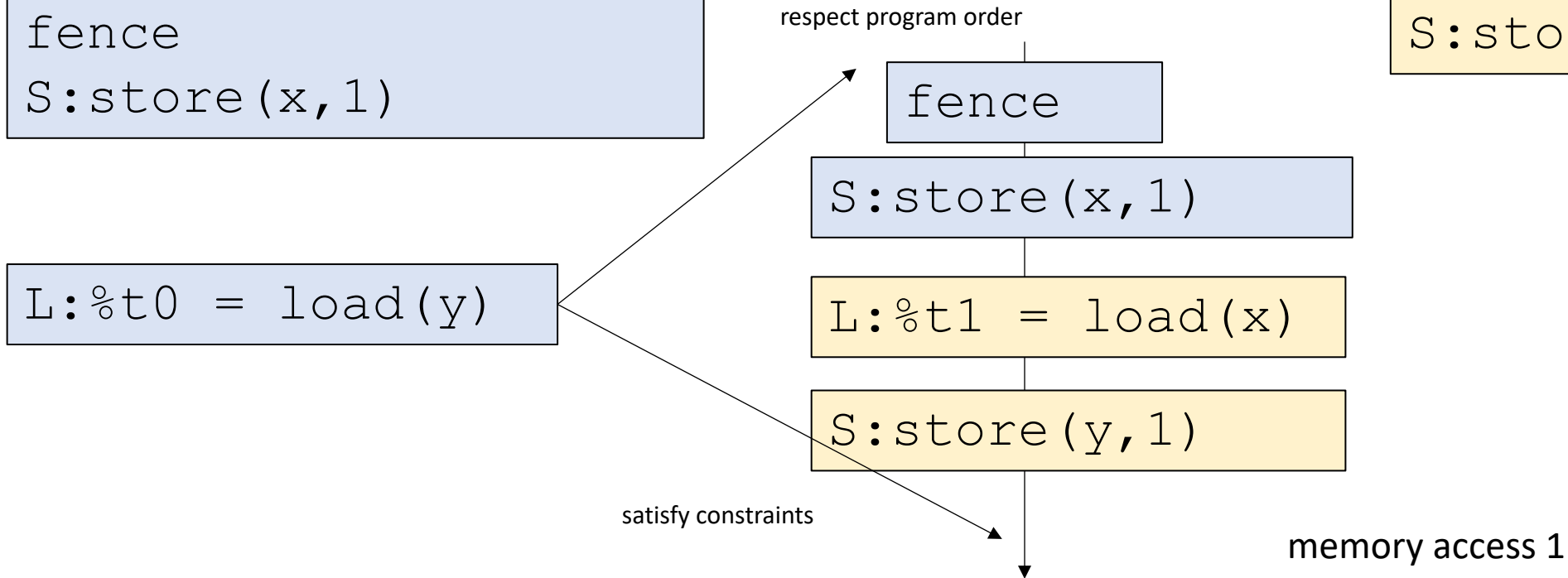
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
fence  
S:store(x, 1)
```

Thread 1:

```
L:%t1 = load(x)  
S:store(y, 1)
```



Now we cannot break program order past the fence!
Are we done?

	L	S
L	YES	Different address
S	different address	Different address

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

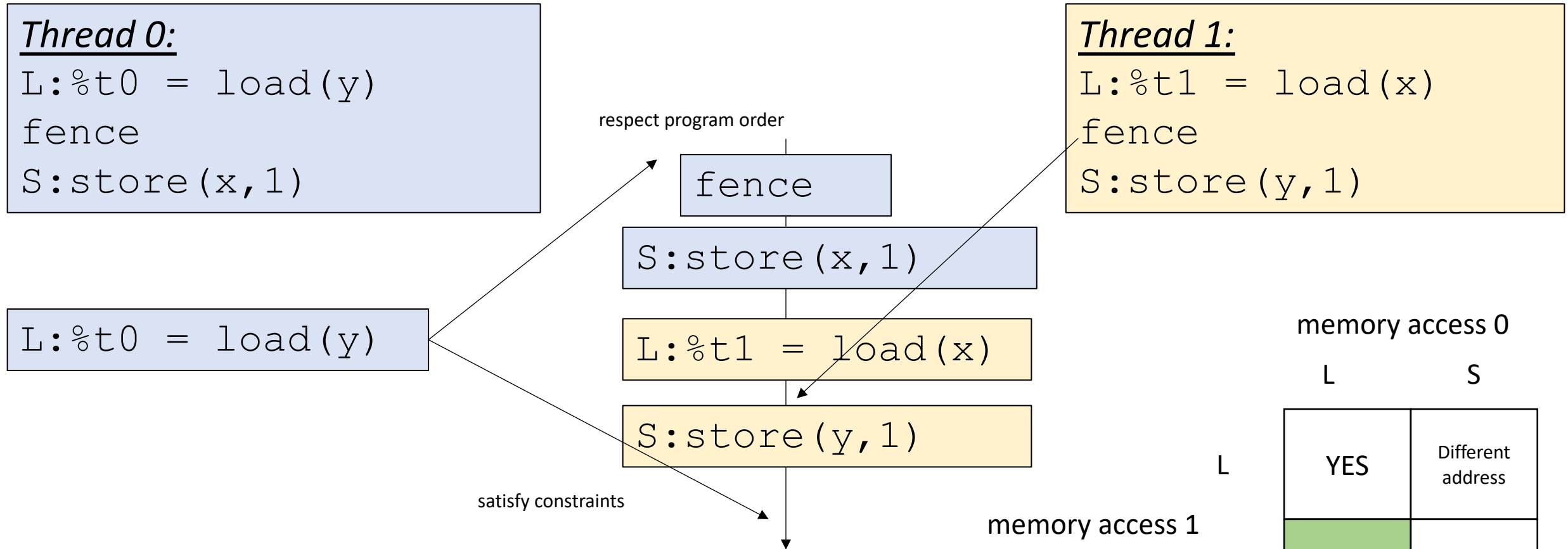
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
fence  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
fence  
S:store(y,1)
```



memory access 0

L S

L

YES

Different
address

S

different
address

Different
address

Now we cannot break program order past the fence!
Are we done?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == t1 == 1`?

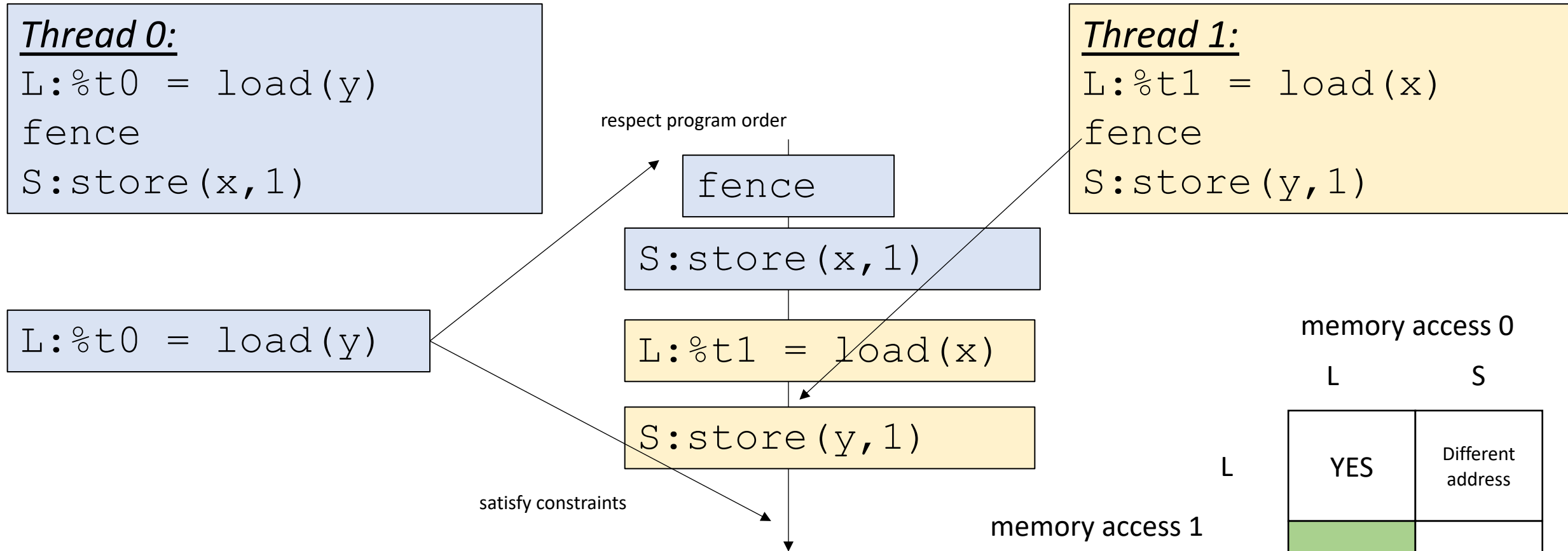
Get out our lego bricks

Thread 0:

```
L:%t0 = load(y)  
fence  
S:store(x,1)
```

Thread 1:

```
L:%t1 = load(x)  
fence  
S:store(y,1)
```



Now we cannot break program order past the fence!
Are we done? The behavior is no longer allowed

	L	S
L	YES	Different address
S	different address	Different address

One more example

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

```
S:store(x,1)
```

```
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```

```
L:%t0 = load(y)
```

```
L:%t1 = load(x)
```



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Thread 0:


```
S:store(x,1)  
S:store(y,1)
```

```
S:store(x,1)
```

```
S:store(y,1)
```

Question: can `t0 == 1` and `t1 == 0`?

start off thinking
about sequential
consistency



Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```

```
L:%t0 = load(y)
```

```
L:%t1 = load(x)
```

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

start off thinking
about sequential
consistency

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```

S:store(x,1)

S:store(y,1)

L:%t0 = load(y)

L:%t1 = load(x)

respect program order

satisfy constraints



Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

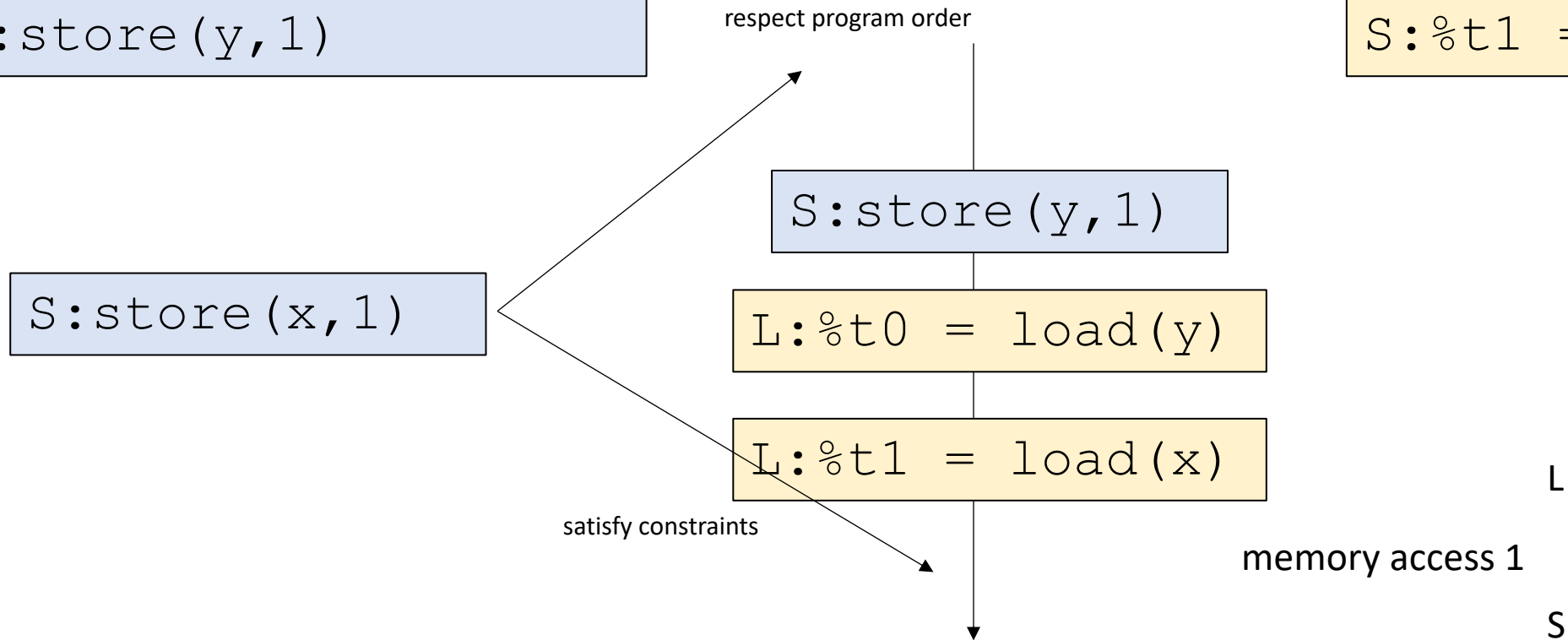
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 0

L S

L

S

L	S
NO	Different address
NO	NO

What about TSO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

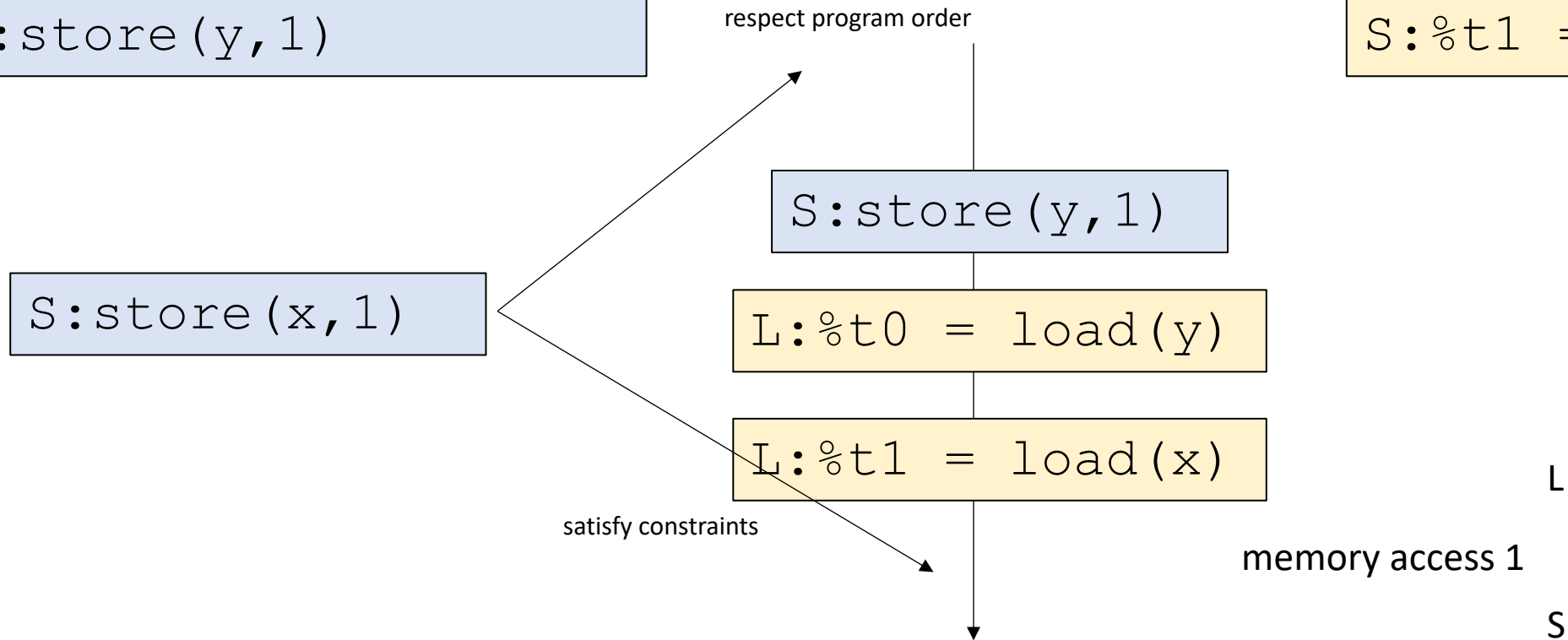
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 0

L S

L

NO

Different
address

S

NO

NO

What about TSO? NO

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

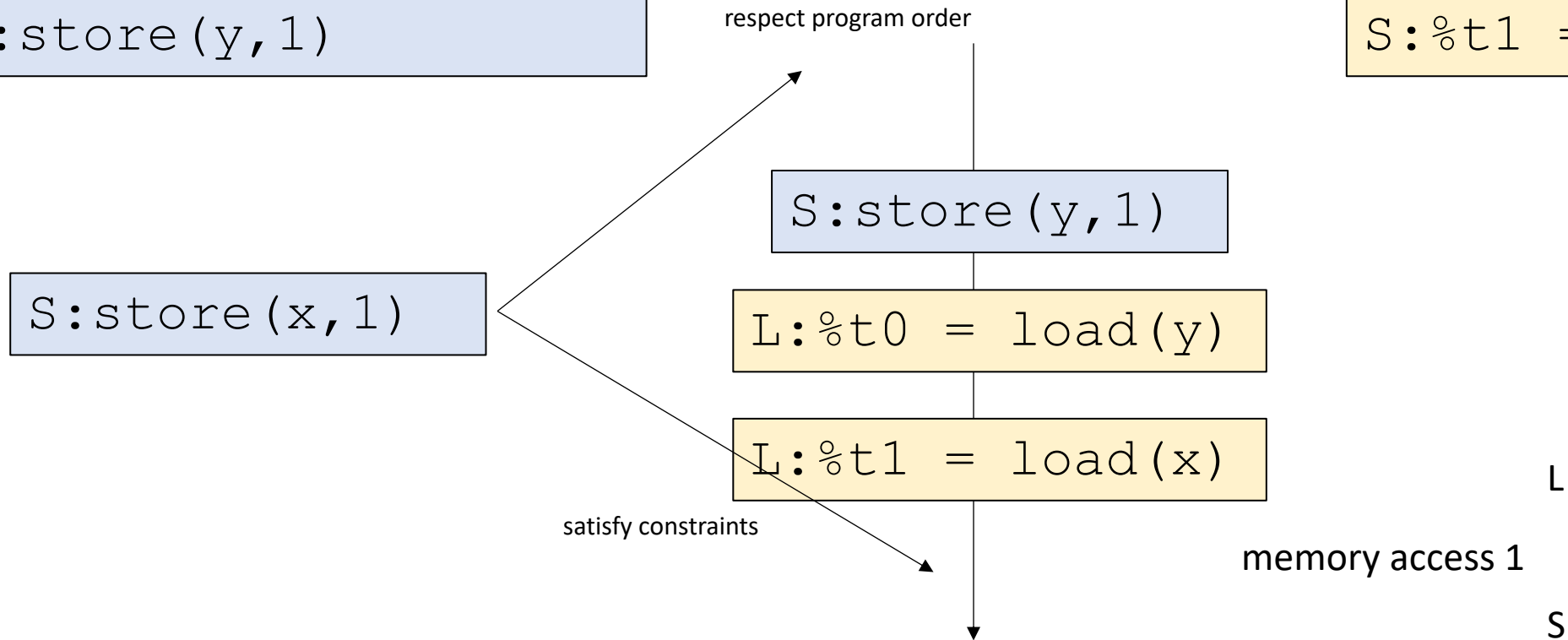
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 0

L S

L

NO

Different
address

S

NO

Different
address

What about PSO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

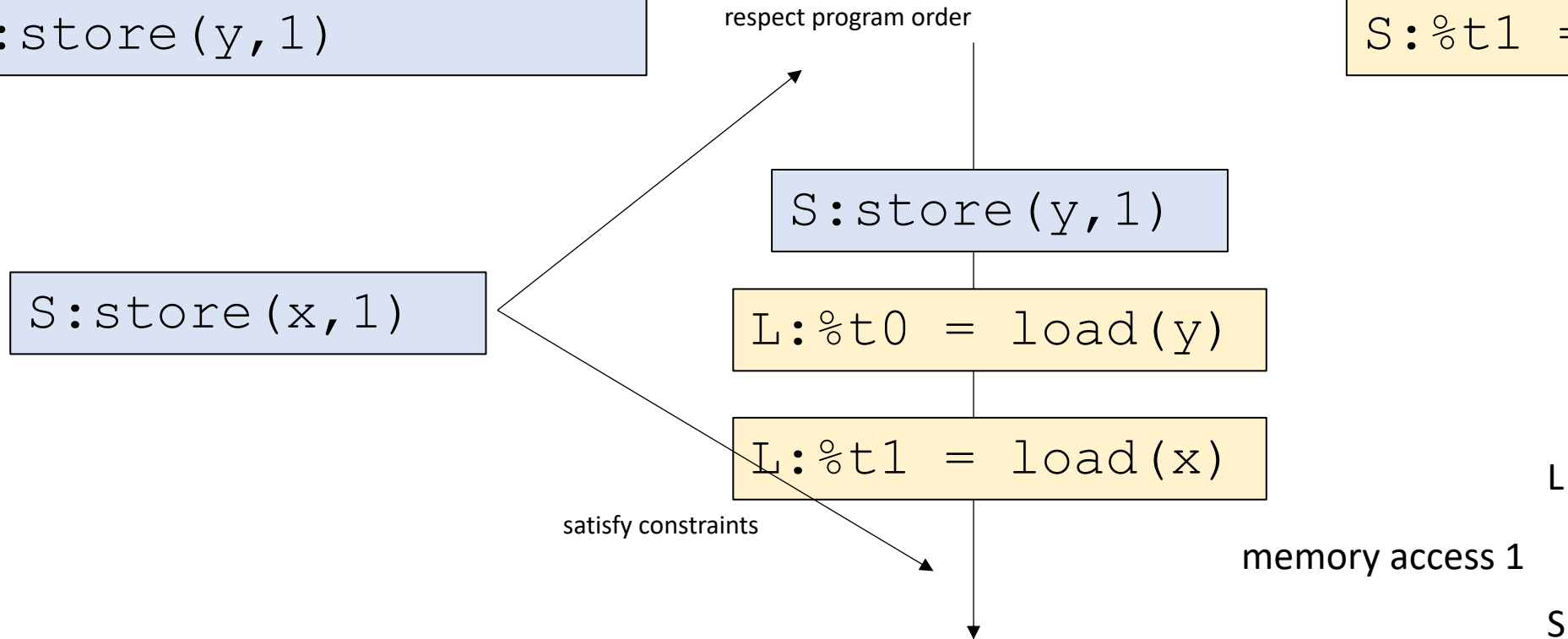
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 0

L S

L

NO

Different
address

S

NO

Different
address

What about PSO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

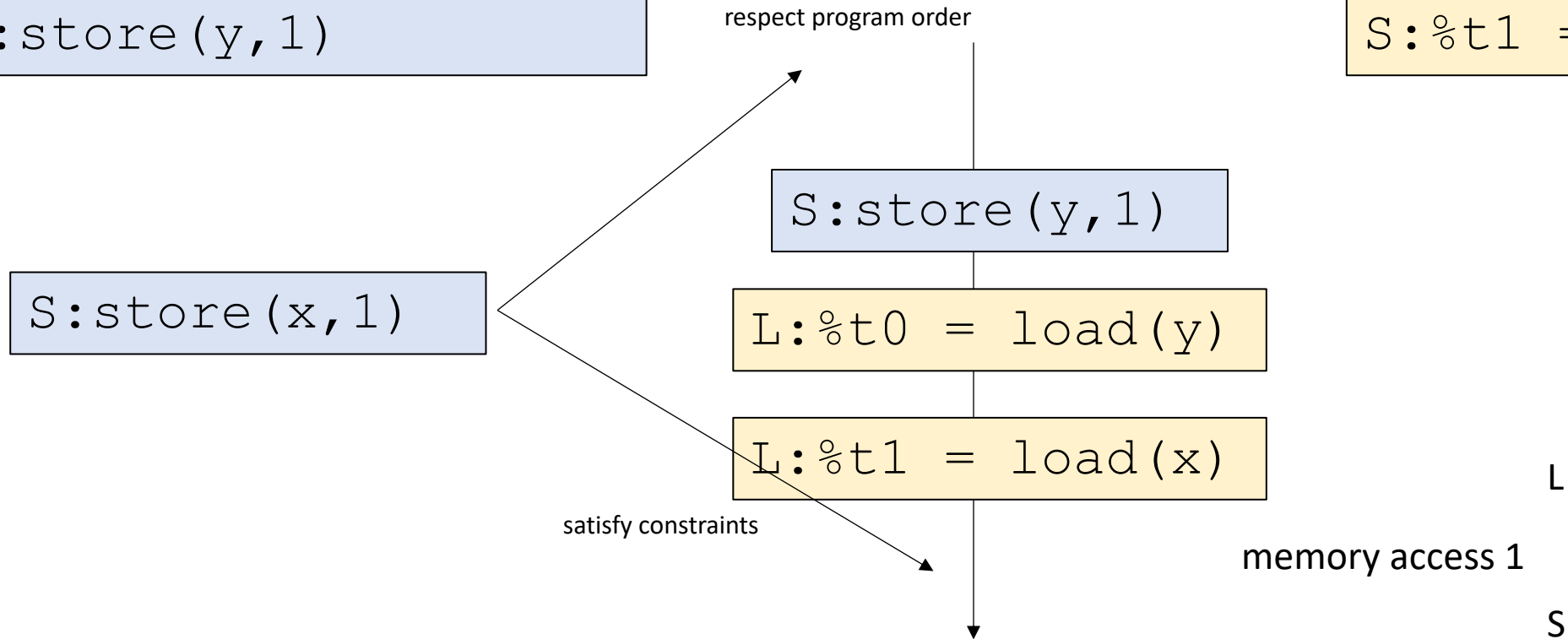
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 0

L S

L

NO

Different address

S

NO

Different address

What about PSO? YES

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

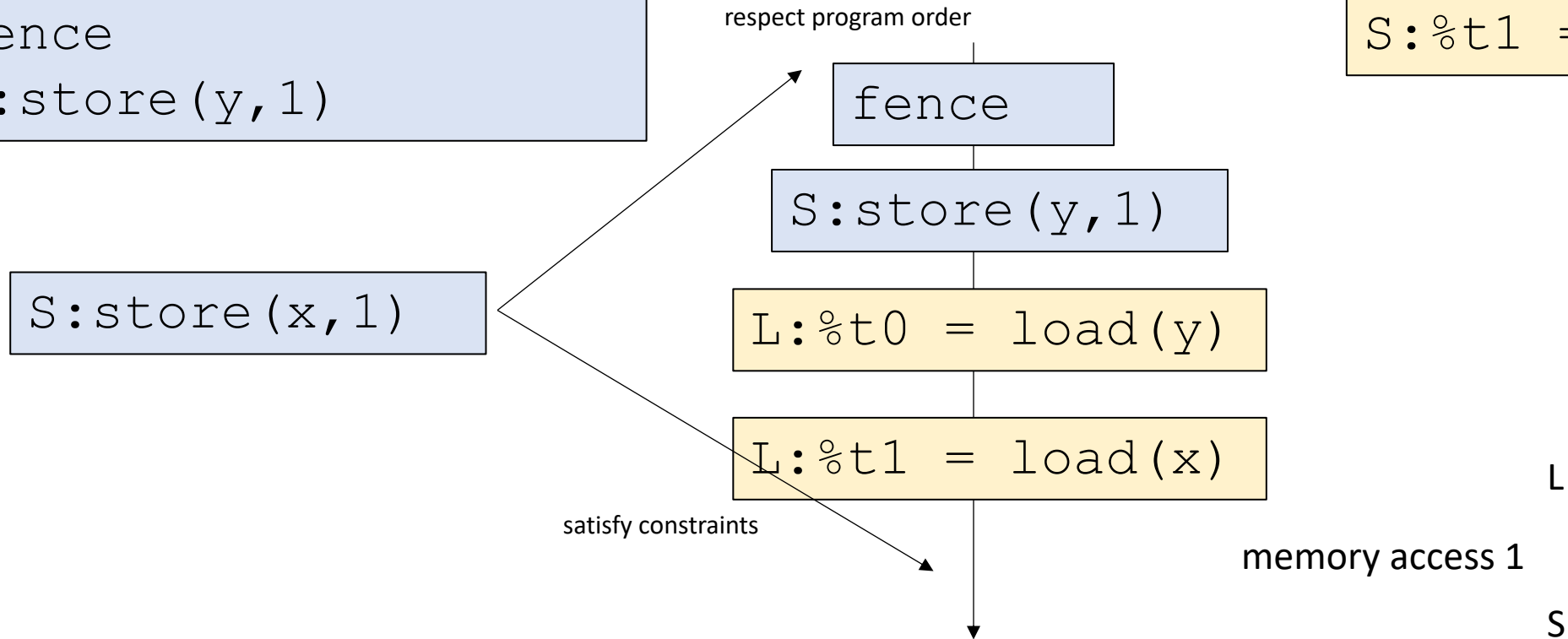
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
fence  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 0

L S

L

NO

Different
address

S

NO

Different
address

Now it is disallowed in PSO

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

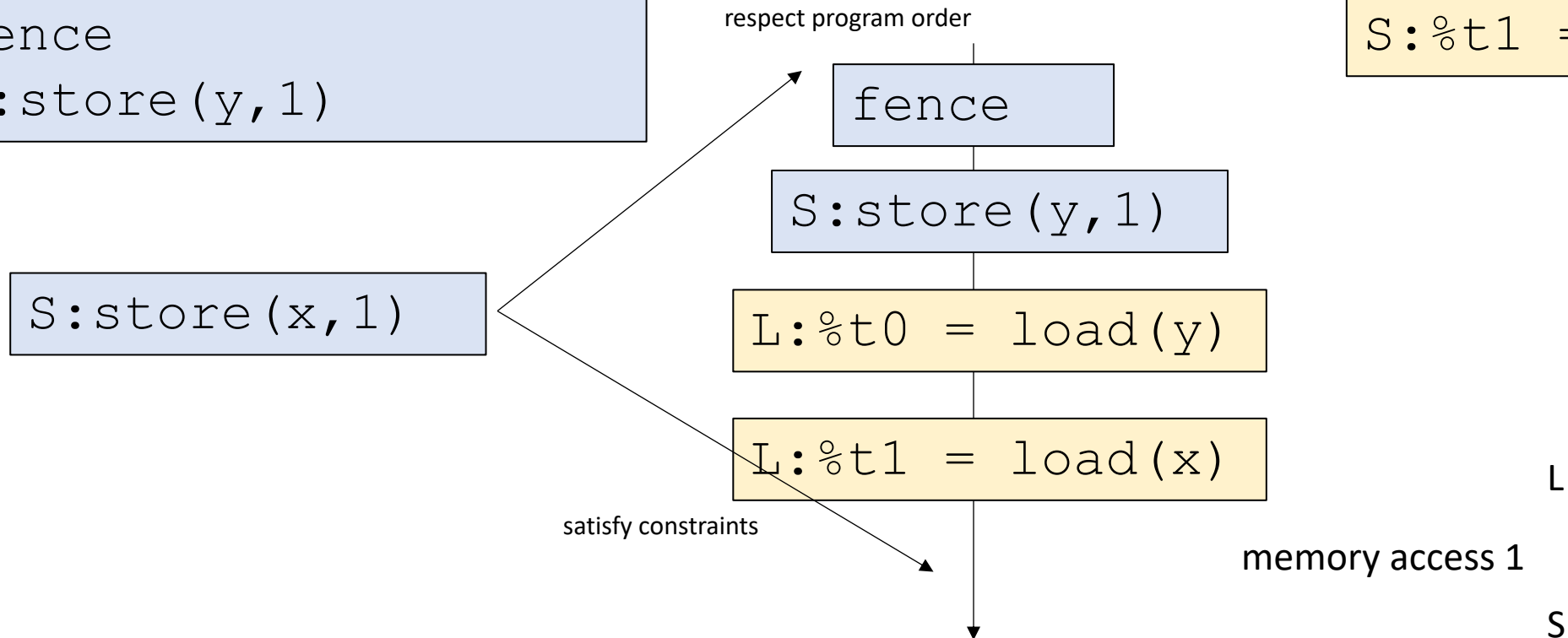
Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
fence  
S:store(y,1)
```

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```



memory access 0

L S

L

S

L	S
YES	Different address
Different address	Different address

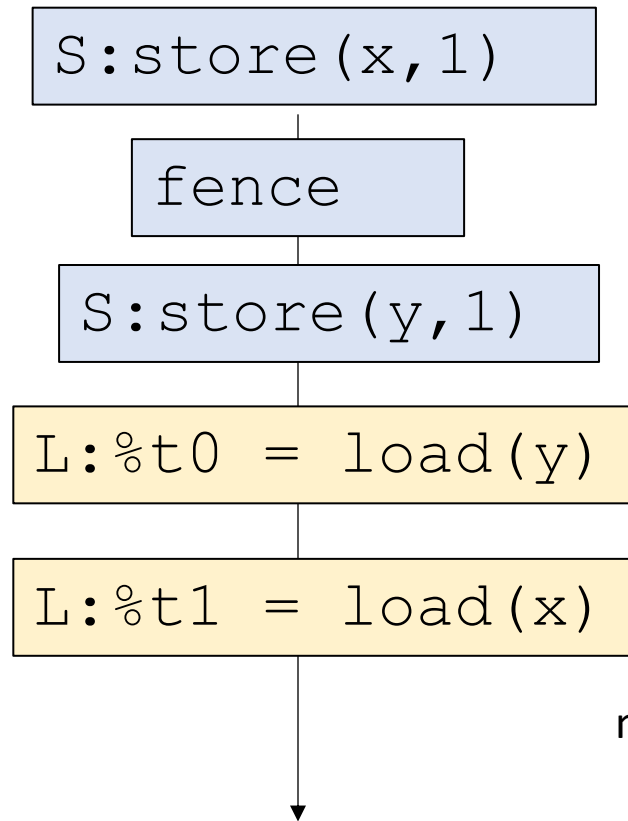
Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
fence  
S:store(y,1)
```



Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```

		memory access 0	
		L	S
memory access 1	L	YES	Different address
	S	Different address	Different address

What about RMO?

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
fence  
S:store(y,1)
```

L:%t1 = load(x)

S:store(x,1)

fence

S:store(y,1)

L:%t0 = load(y)

Thread 1:

```
L:%t0 = load(y)  
S:%t1 = load(x)
```

memory access 0		L	S
memory access 1	L	YES	Different address
	S	Different address	Different address

What about RMO? The loads can be reordered also!

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == 1` and `t1 == 0`?

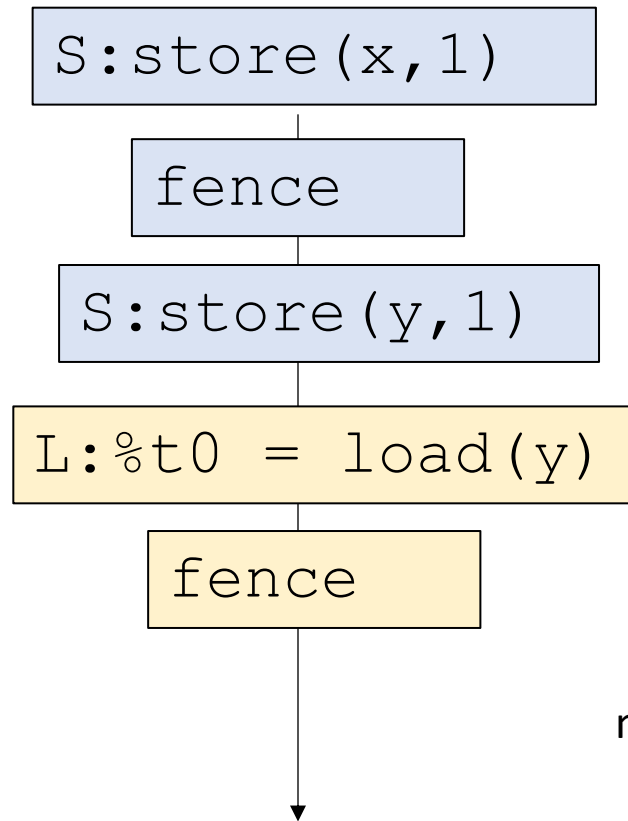
Thread 0:

```
S:store(x,1)  
fence  
S:store(y,1)
```

```
L:%t1 = load(x)
```

Thread 1:

```
L:%t0 = load(y)  
fence  
S:%t1 = load(x)
```



memory access 1

memory access 0

	L	S
L	YES	Different address
S	Different address	Different address

What about RMO? add a fence

Global variable:

```
int x[1] = {0};  
int y[1] = {0};
```

Question: can `t0 == 1` and `t1 == 0`?

Thread 0:

```
S:store(x,1)  
fence  
S:store(y,1)
```

S:store(x,1)

fence

S:store(y,1)

L:%t0 = load(y)

fence

L:%t1 = load(x)

memory access 1

Thread 1:

```
L:%t0 = load(y)  
fence  
S:%t1 = load(x)
```

memory access 0

L S

L

S

	L	S
L	YES	Different address
S	Different address	Different address

Now the relaxed behavior is disallowed

Memory consistency in the real world

- Historic Chips:
 - X86: TSO
 - Surprising robust
 - mutexes and concurrent data structures generally seem to work
 - watch out for store buffering
 - IBM Power and ARM
 - Very relaxed. Similar to RMO with even more rules
 - Mutexes and data structures must be written with care
 - ARM recently strengthened theirs

Memory consistency in the real world

- Historic Chips:
 - X86: TSO
 - Surprising robust
 - mutexes and concurrent data structures generally seem to work
 - watch out for store buffering
 - IBM Power and ARM
 - Very relaxed. Similar to RMO with even more rules
 - Mutexes and data structures must be written with care
 - ARM recently strengthened theirs

Companies have a history of providing insufficient documentation about their rules: academics have then gone and figured it out!

Getting better these days

Memory consistency in the real world

- Modern Chips:
 - RISC-V : two specs: one similar to TSO, one similar to RMO
 - Apple M1: toggles between TSO and weaker
- Vulkan does not provide any fences that provide S - L ordering

Memory consistency in the real world

- PSO and RMO were never implemented widely
 - I have not met anyone who knows of any RMO taped out chip
 - They are part of SPARC ISAs (i.e. RISC-V before it was cool)
 - These memory models might have been part of specialized chips
- Interestingly:
 - Early Nvidia GPUs appeared to informally implement RMO
- Other chips have very strange memory models:
 - Alpha DEC - basically no rules

Where do programming languages fit in?

- One of the highest priorities of a programming language
 - Write once, run everywhere

C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language

C++11 (sequential consistency)

	L	S
L	NO	NO
S	NO	NO

target machine

	L	S
L	?	?
S	?	?

C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language			
C++11 (sequential consistency)			
	L	S	
L	NO	NO	
S	NO	NO	

target machine			
TSO (x86)			
	L	S	
L	NO	different address	
S	NO	No	

C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

	L	S
L	NO	NO
S	NO	NO

find mismatch

target machine
TSO (x86)

	L	S
L	NO	different address
S	NO	No

C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

	L	S
L	NO	NO
S	NO	NO

find mismatch

Two options:

make sure stores
are not reordered
with later loads

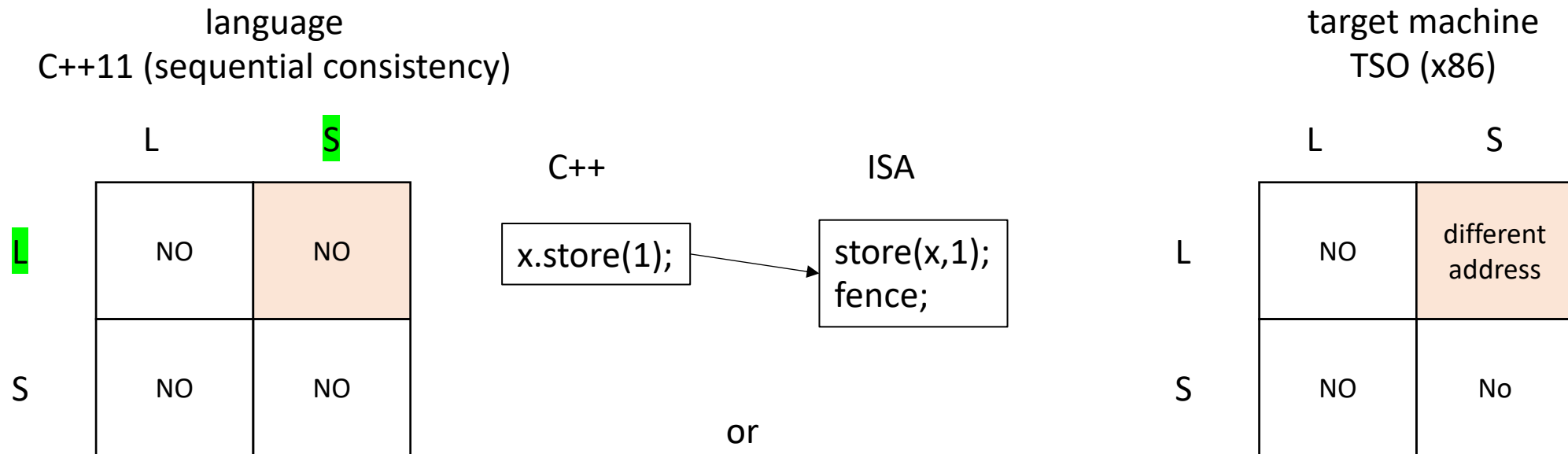
make sure loads
are not reordered
with earlier stores

target machine
TSO (x86)

	L	S
L	NO	different address
S	NO	No

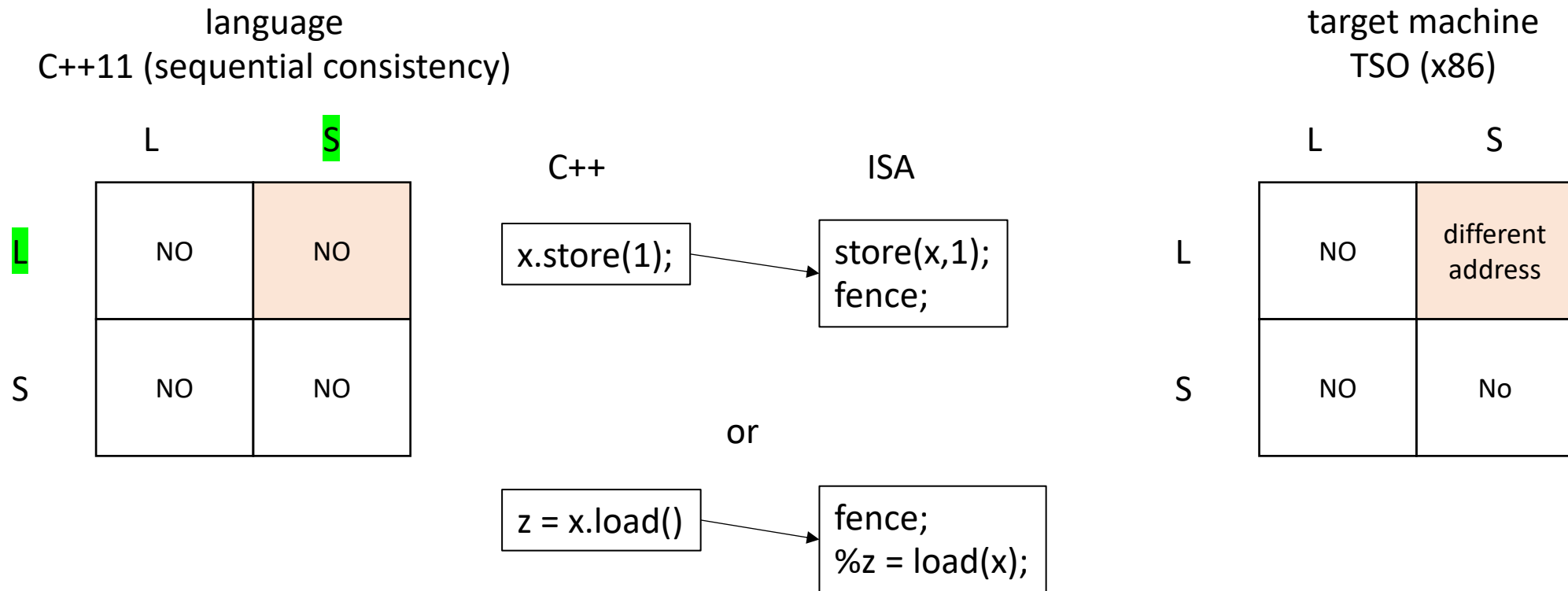
C++11 atomic operation compilation

start with both both of the grids for the two different memory models



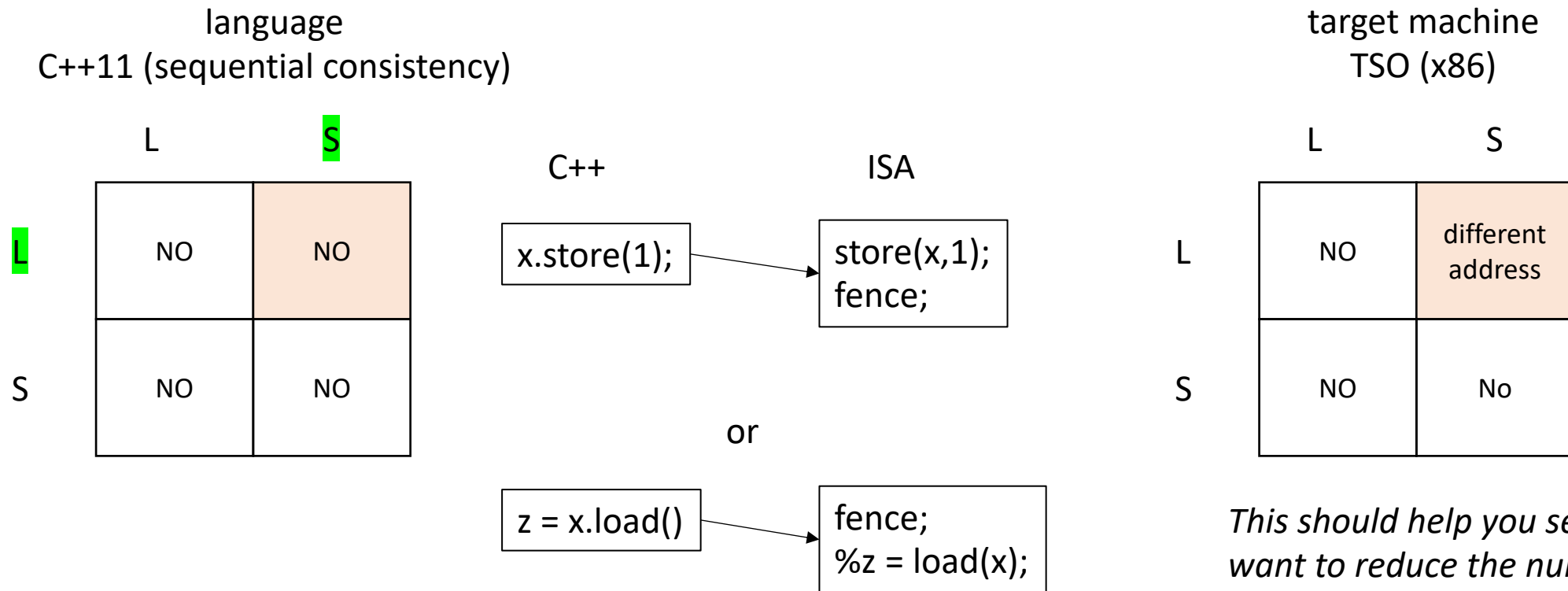
C++11 atomic operation compilation

start with both both of the grids for the two different memory models



C++11 atomic operation compilation

start with both both of the grids for the two different memory models



This should help you see why you want to reduce the number of atomic load/stores in your program

C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

	L	S
L	NO	NO
S	NO	NO

How about this one?

target machine
PSO

	L	S
L	NO	different address
S	NO	different address

C++11 atomic operation compilation

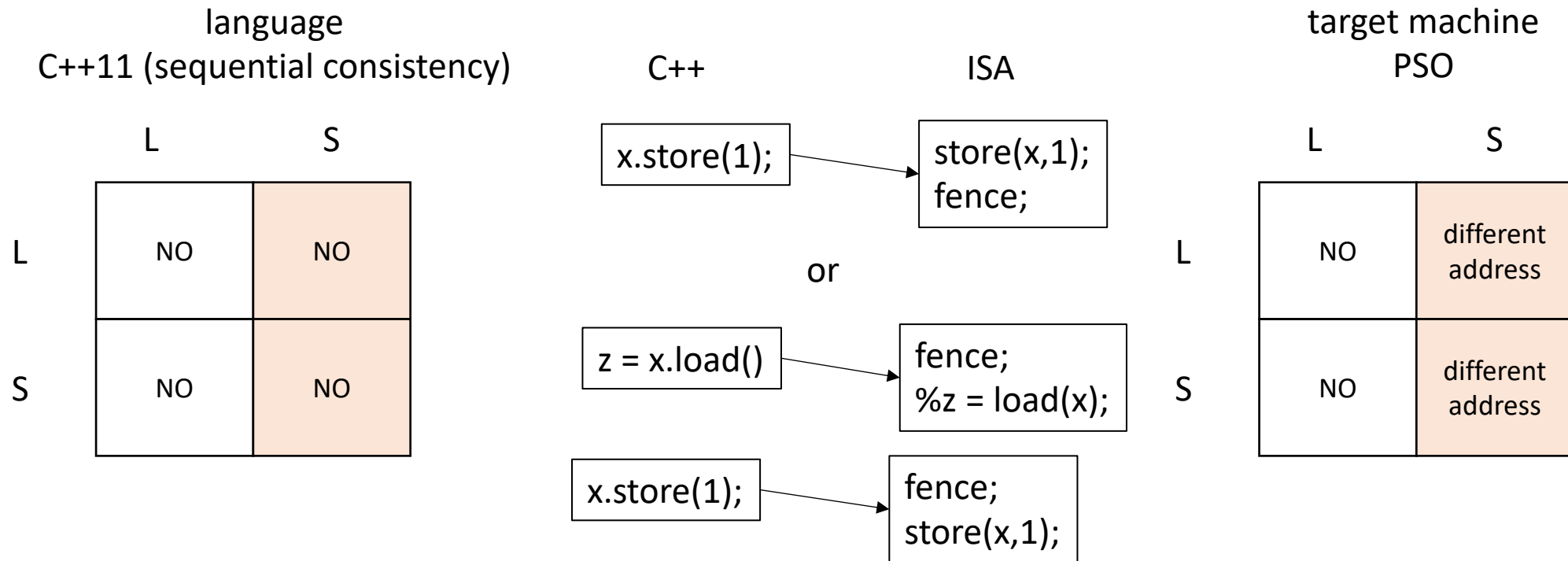
start with both both of the grids for the two different memory models

language			
C++11 (sequential consistency)			
	L	S	
L	NO	NO	
S	NO	NO	

target machine			
PSO			
	L	S	
L	NO	different address	
S	NO	different address	

C++11 atomic operation compilation

start with both both of the grids for the two different memory models



Memory orders

- Atomic operations take an additional “memory order” argument
 - `memory_order_seq_cst` - default
 - `memory_order_relaxed` - weakest

Where have we seen `memory_order_relaxed`?

Relaxed memory order

language
C++11 (sequential consistency)

	L	S
L	NO	NO
S	NO	NO

language
C++11 (memory_order_relaxed)

	L	S
L	different address	different address
S	different address	different address

basically no orderings except for accesses to
the same address

Compiling memory order relaxed

language
C++11 (memory_order_relaxed)

	L	S
L	different address	different address
S	different address	different address

target machine
TSO (x86)

	L	S
L	NO	different address
S	NO	No

Compiling memory order relaxed

language
C++11 (memory_order_relaxed)

	L	S
L	different address	different address
S	different address	different address

lots of mismatches!

target machine
TSO (x86)

	L	S
L	NO	different address
S	NO	No

Compiling memory order relaxed

language
C++11 (memory_order_relaxed)

	L	S
L	different address	different address
S	different address	different address

lots of mismatches!

But language is more relaxed than machine

so no fences are needed

target machine
TSO (x86)

	L	S
L	NO	different address
S	NO	No

Compiling memory order relaxed

Do any of the ISA memory models need any fences for relaxed memory order?

language
C++11 (memory_order_relaxed)

	L	S
L	different address	different address
S	different address	different address

	L	S
L	NO	Different address
S	NO	NO

TSO

	L	S
L	NO	Different address
S	NO	Different address

PSO

	L	S
L	YES	Different address
S	Different address	Different address

RMO

Memory order relaxed

- Very few use-cases! Be very careful when using it
 - Peeking at values (later accessed using a heavier memory order)
 - Counting (e.g. number of finished threads in work stealing)
 - ***DO NOT USE FOR QUEUE INDEXES***

More memory orders: we will not discuss in class

- Atomic operations take an additional “memory order” argument
 - `memory_order_seq_cst` - default
 - `memory_order_relaxed` - weakest
- More memory orders (useful for mutex implementations):
 - `memory_order_acquire`
 - `memory_order_release`
- EVEN MORE memory orders (complicated: in most research it is omitted)
 - `memory_order_consume`

A cautionary tale

*Consider the following example: a graphics program where each thread wants to display a triangle;
the display is a queue (not thread safe)*

Thread 0:

```
m.lock();  
display.enq(triangle0);  
m.unlock();
```

Thread 1:

```
m.lock();  
display.enq(triangle1);  
m.unlock();
```

*Consider the following example: a graphics program where each thread wants to display a triangle;
the display is a queue (not thread safe)*

Thread 0:

```
m.lock();  
display.enq(triangle0);  
m.unlock();
```

Thread 1:

```
m.lock();  
display.enq(triangle1);  
m.unlock();
```

We know how lock and unlock are implemented

*Consider the following example: a graphics program where each thread wants to display a triangle;
the display is a queue (not thread safe)*

Thread 0:

```
SPIN:CAS(mutex, 0, 1);  
display.enq(triangle0);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS(mutex, 0, 1);  
display.enq(triangle1);  
store(mutex, 0);
```

We know how lock and unlock are implemented
We also know how a queue is implemented

*Consider the following example: a graphics program where each thread wants to display a triangle;
the display is a queue (not thread safe)*

Thread 0:

```
SPIN:CAS(mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS(mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

We know how lock and unlock are implemented

We also know how a queue is implemented

What is an execution?

Thread 0:

```
SPIN:CAS (mutex, 0, 1) ;  
%i = load(head) ;  
store(buffer+i, triangle0) ;  
store(head, %i+1) ;  
store(mutex, 0) ;
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1) ;  
%i = load(head) ;  
store(buffer+i, triangle1) ;  
store(head, %i+1) ;  
store(mutex, 0) ;
```

CAS (mutex, 0, 1) ;

*if blue goes first
it gets to complete
its critical section
while thread 1 is spinning*



Thread 0:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

CAS (mutex, 0, 1);

%i = load(head);

store(buffer+i, triangle0);

store(head, %i+1);

store(mutex, 0);



Thread 0:

```
SPIN:CAS(mutex,0,1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex,0);
```

Thread 1:

```
SPIN:CAS(mutex,0,1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex,0);
```

CAS(mutex,0,1);

%i = load(head);

store(buffer+i, triangle0);

store(head, %i+1);

store(mutex,0);

now yellow gets a change to go



Thread 0:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

now yellow gets a change to go

CAS (mutex, 0, 1);

%i = load(head);

store(buffer+i, triangle0);

store(head, %i+1);

store(mutex, 0);

CAS (mutex, 0, 1);

%i = load(head);

store(buffer+i, triangle0);

store(head, %i+1);

store(mutex, 0);



Thread 0:

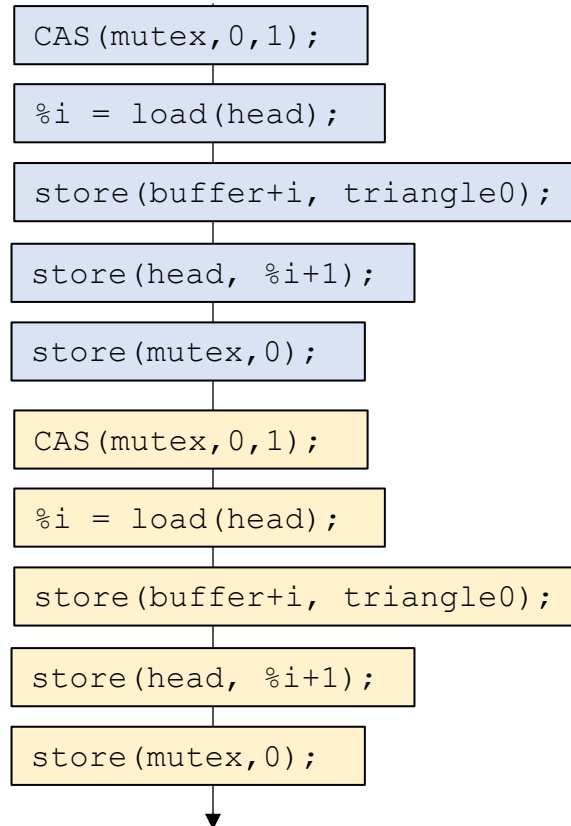
```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

*what can happen in a PSO
memory model?*

	L	S
L	NO	Different address
S	NO	Different address



Thread 0:

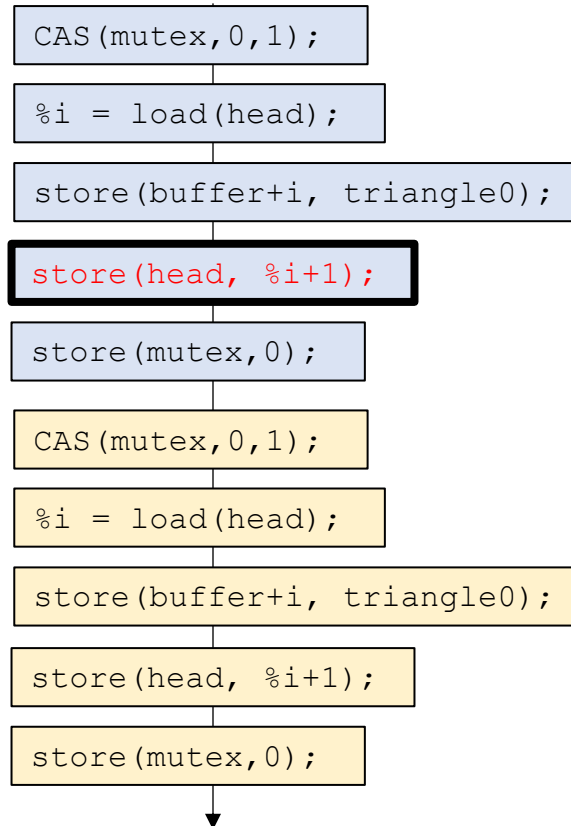
```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

*what can happen in a PSO
memory model?*

	L	S
L	NO	Different address
S	NO	Different address



Thread 0:

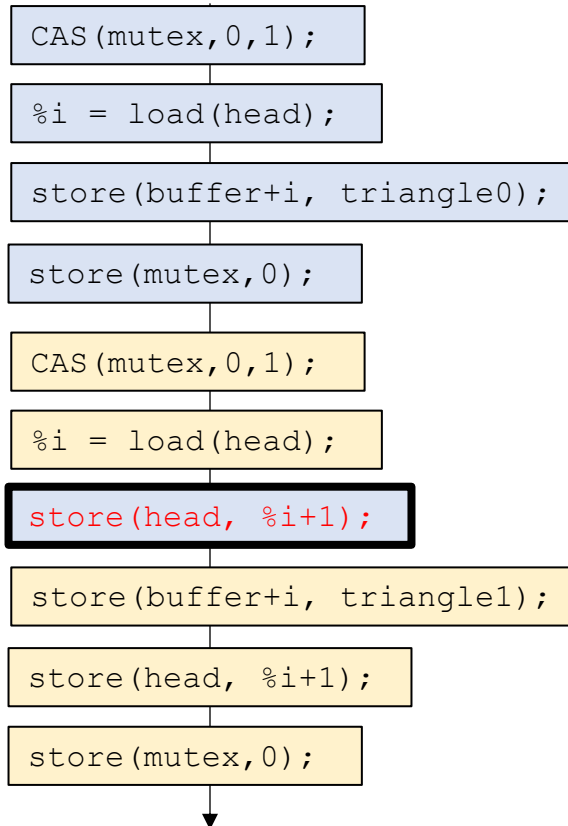
```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

*what can happen in a PSO
memory model?*

	L	S
L	NO	Different address
S	NO	Different address

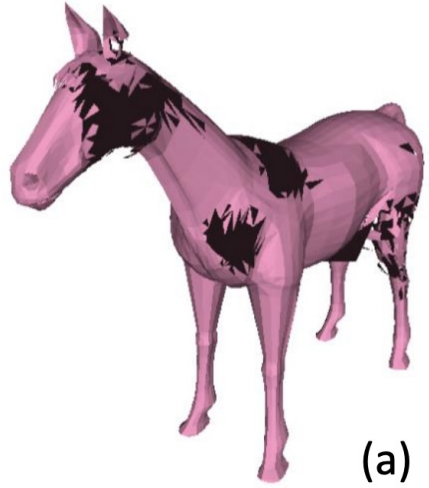


What just happened if this store moves?

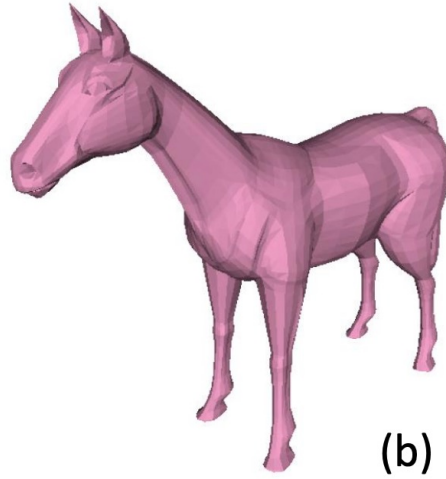
Nvidia in 2015

- Nvidia architects implemented a weak memory model
- Nvidia programmers expected a strong memory model
- Mutexes implemented without fences!

Nvidia in 2015



(a)



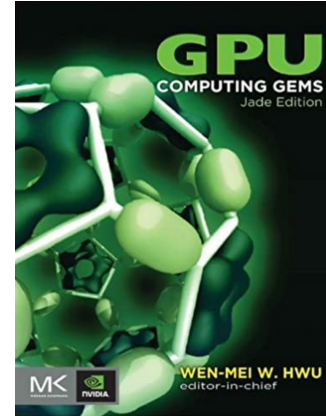
(b)



(c)



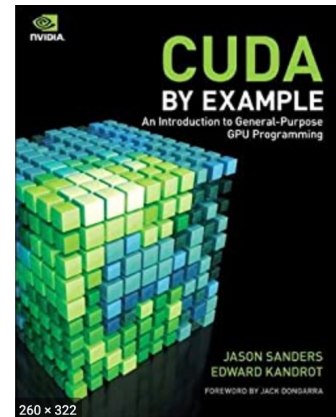
(d)



bug found in two
Nvidia textbooks

We implemented
a side-channel attack
that made the bugs
appear more frequently

These days Nvidia has
a very well-specified
memory model!



Thread 0:

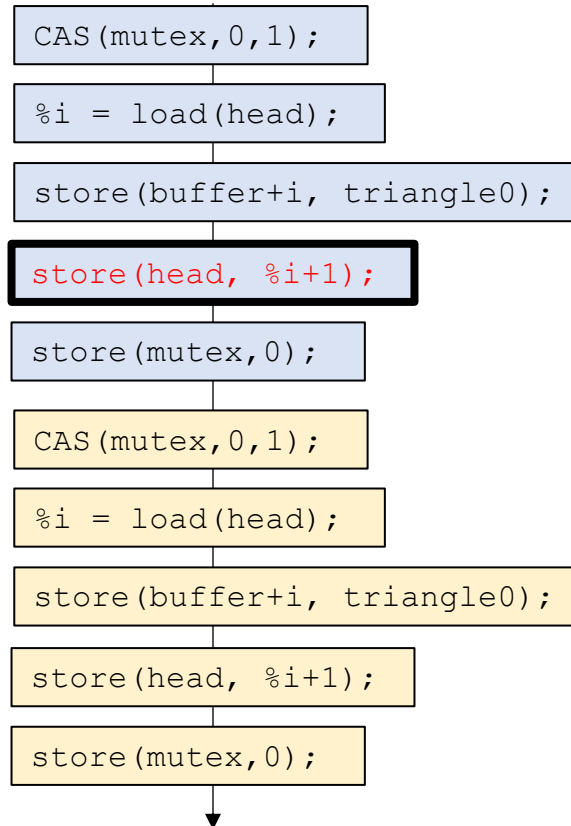
```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
store(mutex, 0);
```

Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
store(mutex, 0);
```

*what can happen in a PSO
memory model?*

	L	S
L	NO	Different address
S	NO	Different address



How to fix the issue?

Thread 0:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
fence;  
store(mutex, 0);
```

*unlock contains fence
before store!*

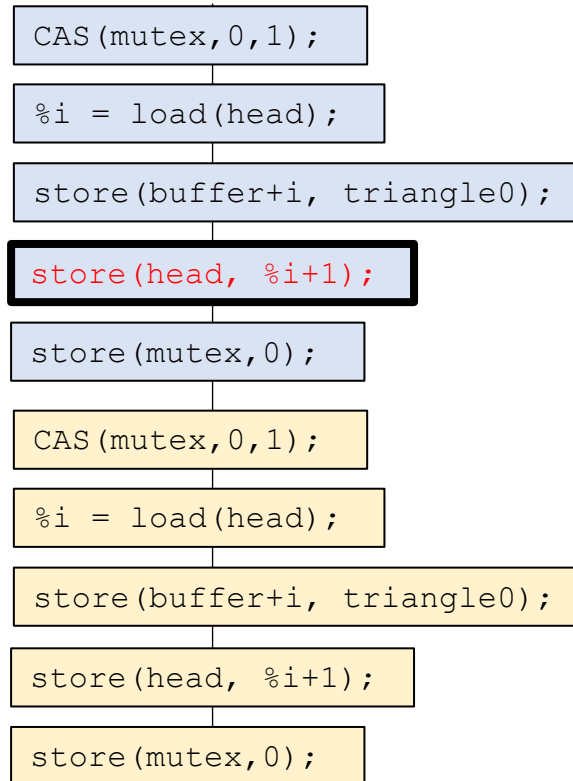
Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
fence;  
store(mutex, 0);
```

*unlock contains fence
before store!*

*what can happen in a PSO
memory model?*

	L	S
L	NO	Different address
S	NO	Different address



How to fix the issue?

your unlock function
should contain a fence!

Thread 0:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle0);  
store(head, %i+1);  
fence;  
store(mutex, 0);
```

*unlock contains fence
before store!*

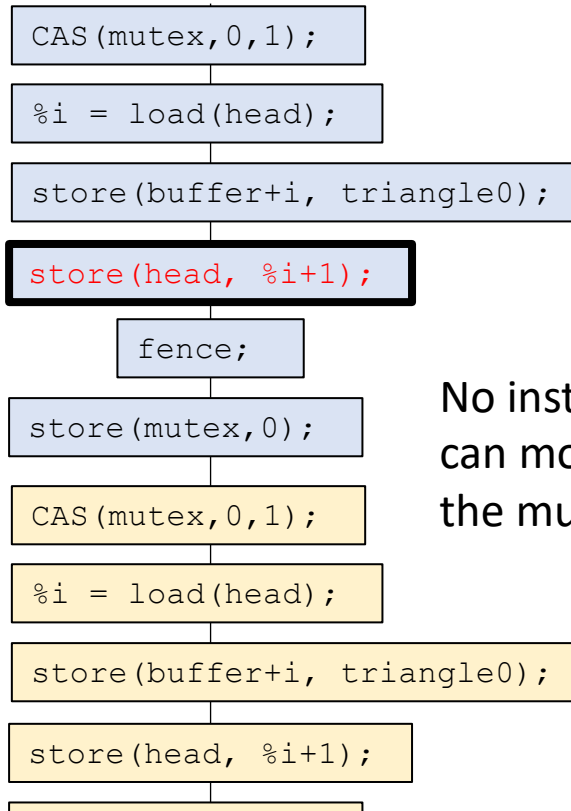
Thread 1:

```
SPIN:CAS (mutex, 0, 1);  
%i = load(head);  
store(buffer+i, triangle1);  
store(head, %i+1);  
fence;  
store(mutex, 0);
```

*unlock contains fence
before store!*

*what can happen in a PSO
memory model?*

	L	S
L	NO	Different address
S	NO	Different address



No instructions
can move after
the mutex store!

How to fix the issue?

your unlock function
should contain a fence!

Memory Model Strength

- If one memory model M0 allows more relaxed behaviors than another memory model M1, then M0 is more *relaxed* (or *weaker*) than M1.
- It is safe to run a program written for M0 on M1. But not vice versa

	L	S
L	NO	Different address
S	NO	NO

TSO

	L	S
L	NO	Different address
S	NO	Different address

PSO

	L	S
L	YES	Different address
S	Different address	Different address

RMO

Memory Model Strength

- Many times specifications are weaker than implementations:
 - A chip might document PSO, but implement TSO:
 - Why?

	L	S
L	NO	Different address
S	NO	NO

TSO

	L	S
L	NO	Different address
S	NO	Different address

PSO

	L	S
L	YES	Different address
S	Different address	Different address

RMO

What memory model does your GPU have?

- Visit the GPU harbor! Demo

If time... General Concurrent Set

Set Interface

- Unordered collection of items
- No duplicates
- We will implement this as a sorted linked list

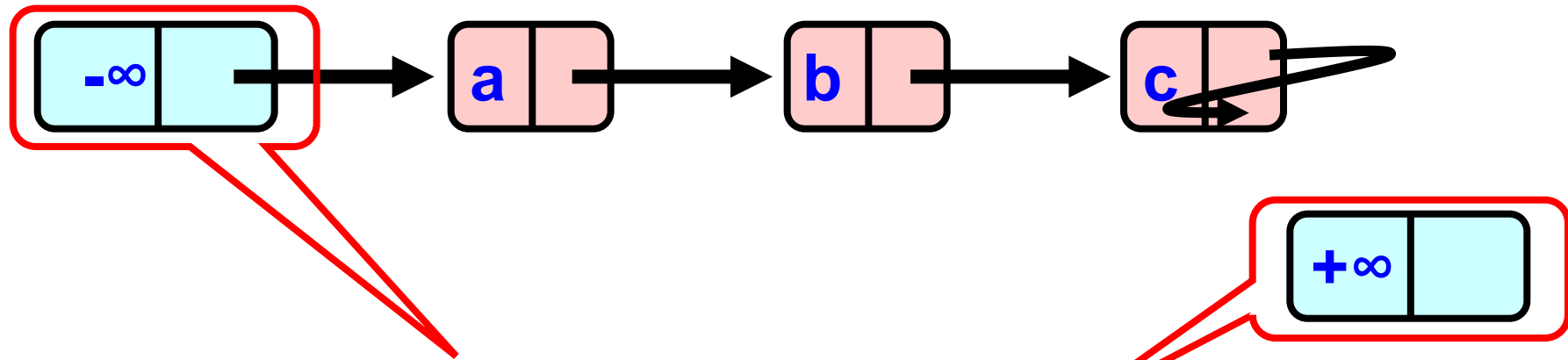
Set Interface

- Unordered collection of items
- No duplicates
- Methods
 - **add (x)** put **x** in set
 - **remove (x)** take **x** out of set
 - **contains (x)** tests if **x** in set

List Node

```
class Node {  
    public:  
        Value v;  
        int key;  
        Node *next;  
}
```

The List-Based Set



Sorted with Sentinel nodes
(min & max possible keys)

Sequential List Based Set

add(b)

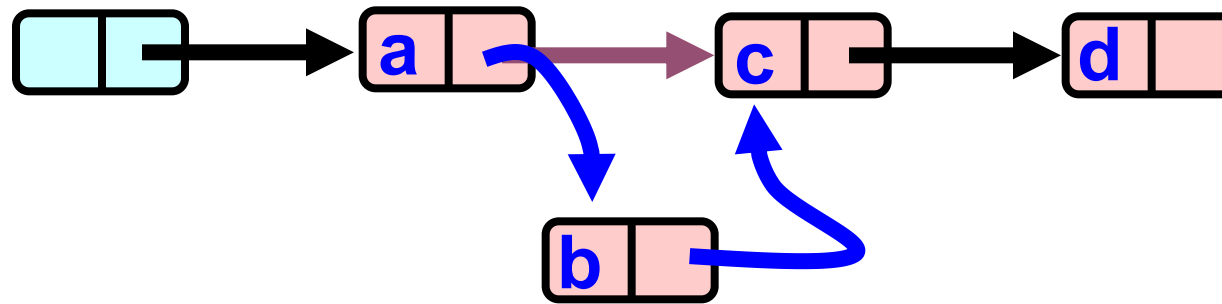


remove(b)

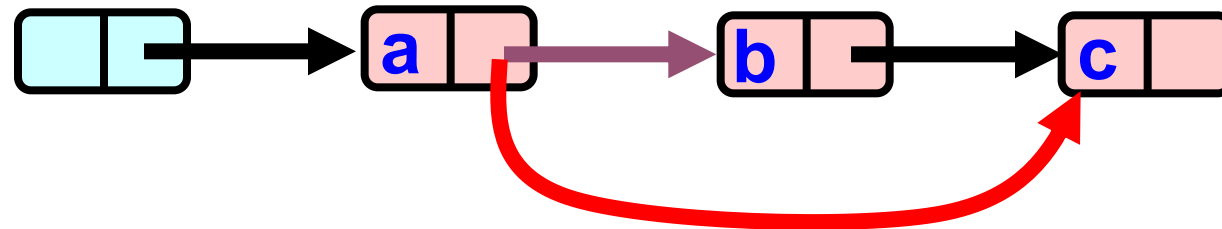


Sequential List Based Set

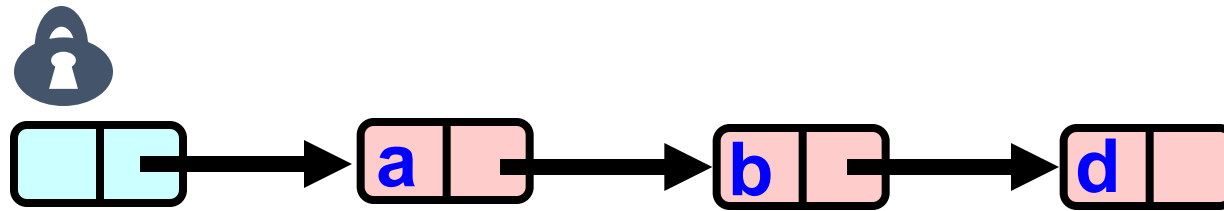
add(b)



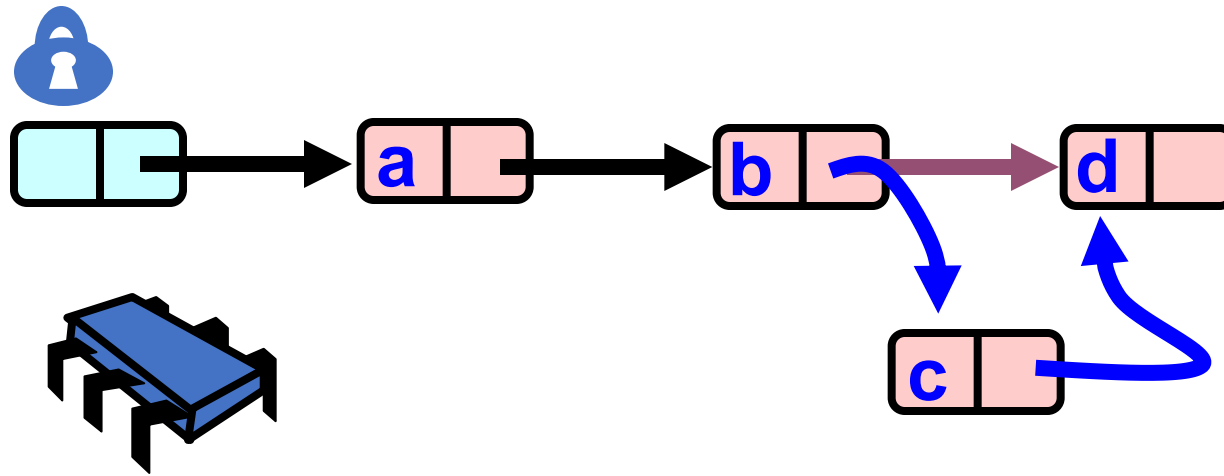
remove(b)



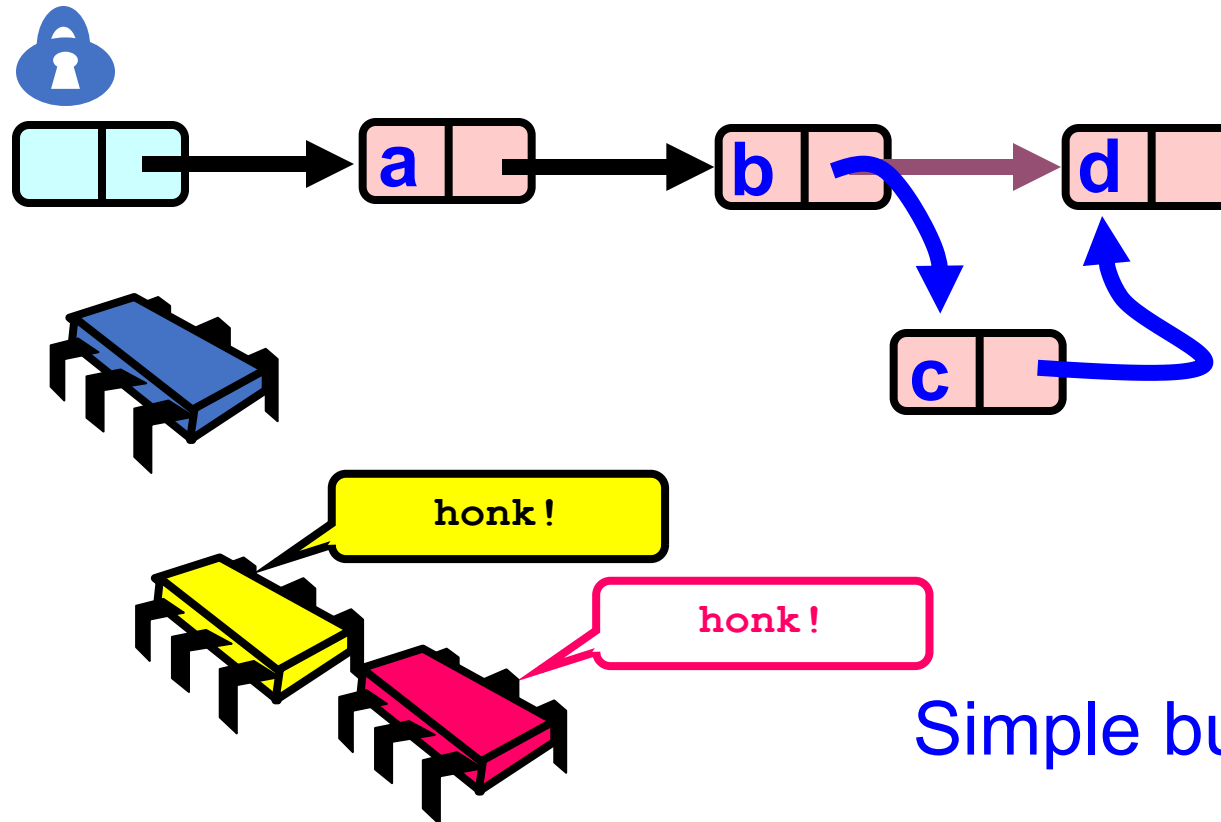
Coarse-Grained Locking



Coarse-Grained Locking



Coarse-Grained Locking



Simple but inefficient!

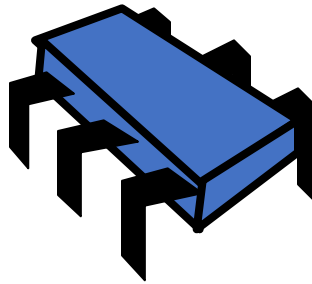
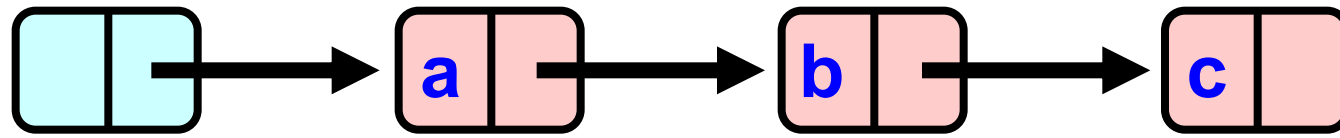
Schedule

- Concurrent set
 - Coarse-grained lock
 - **fine-grained lock**
 - optimistic locking

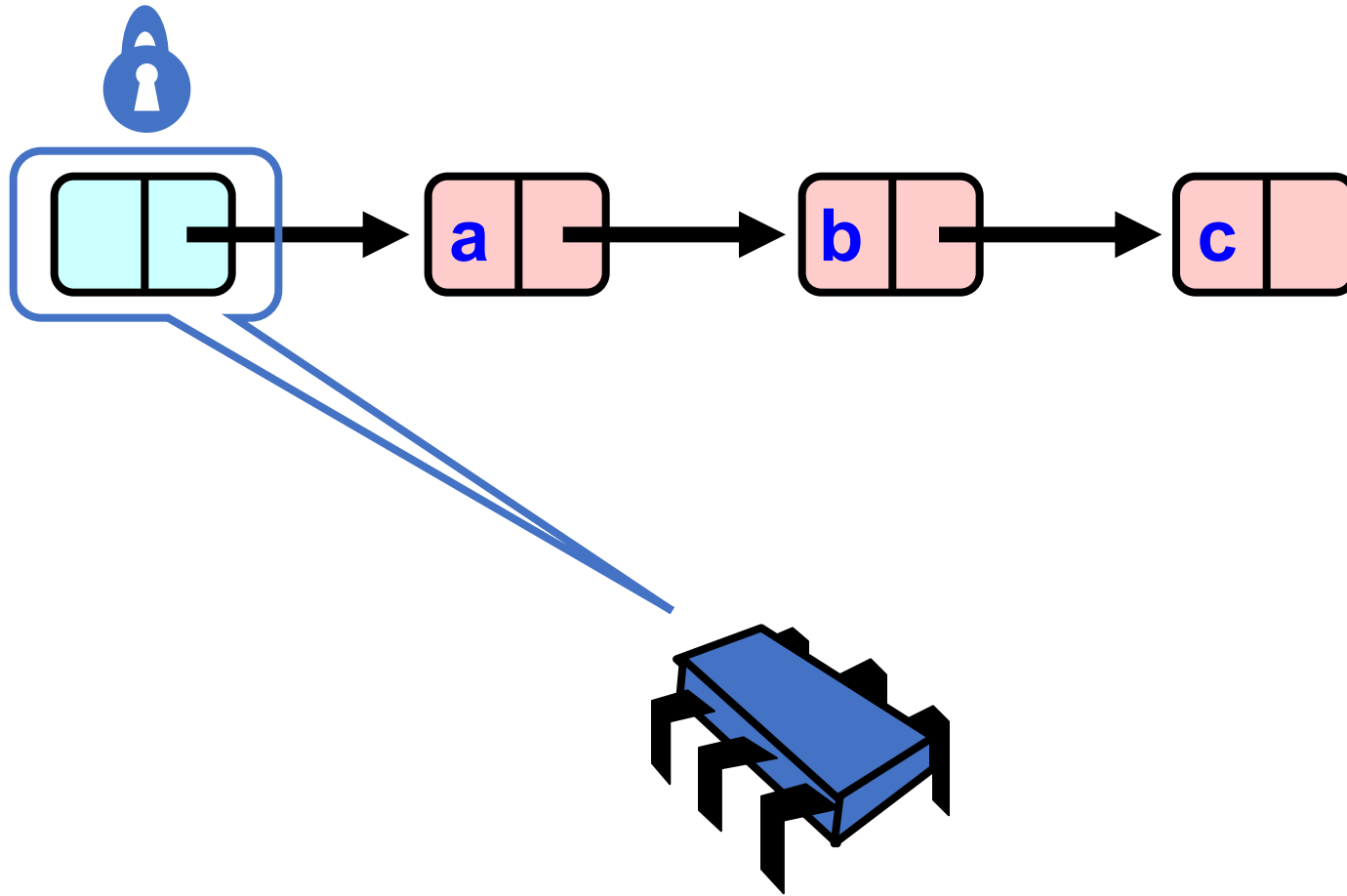
Fine-grained Locking

- Requires **careful** thought
- Split object into pieces
 - Each piece has own lock
 - Methods that work on disjoint pieces need not exclude each other

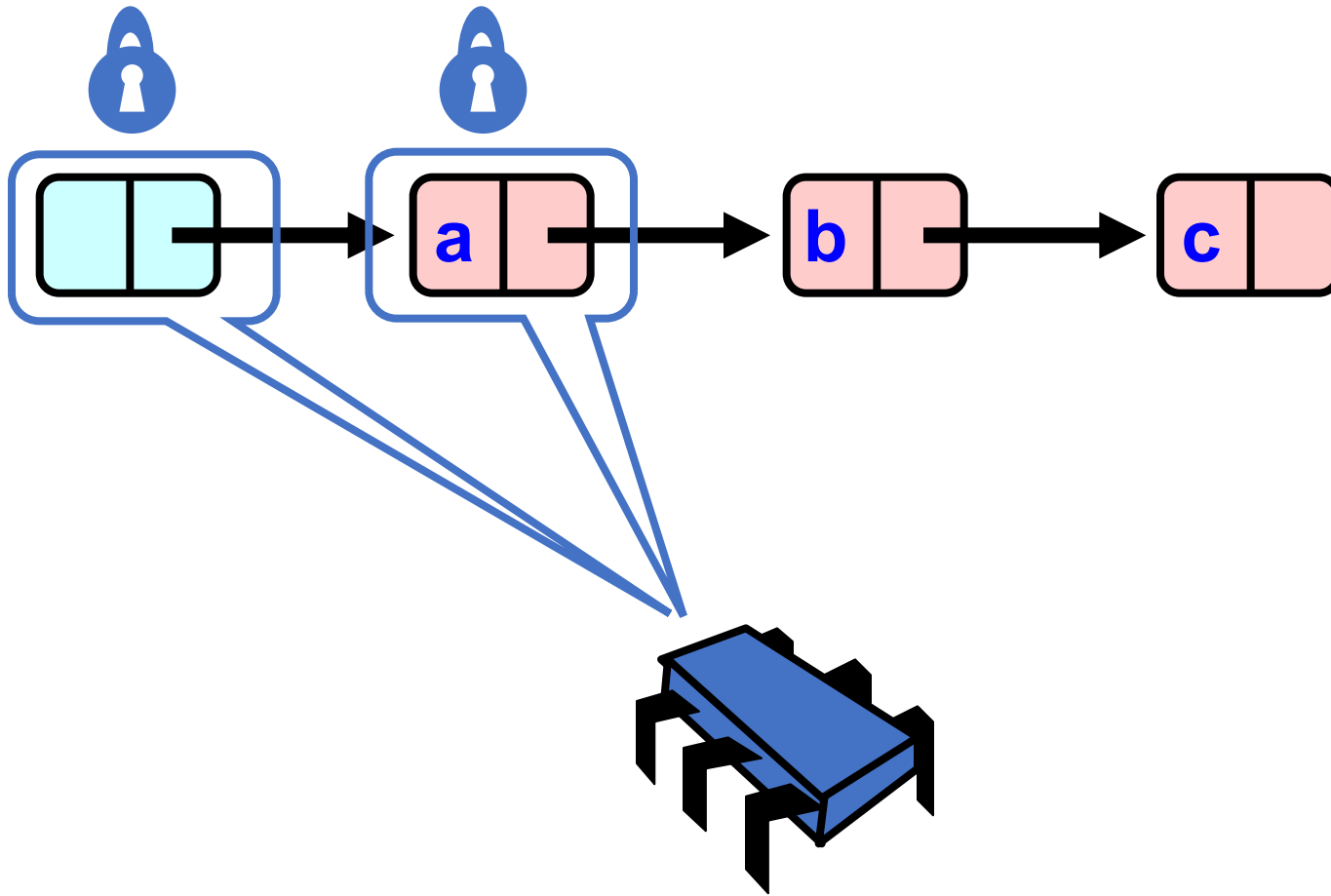
Hand-over-Hand locking



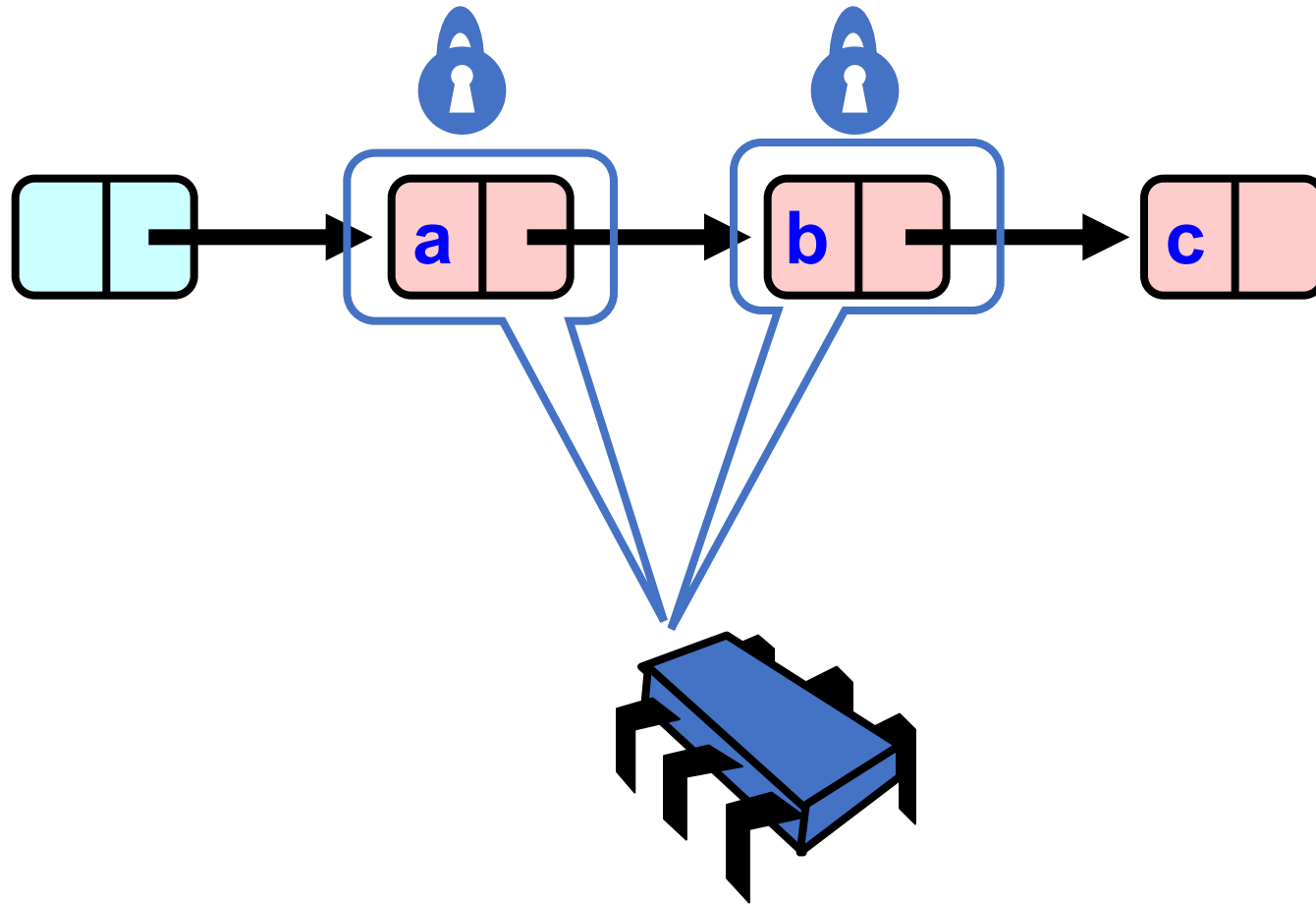
Hand-over-Hand locking



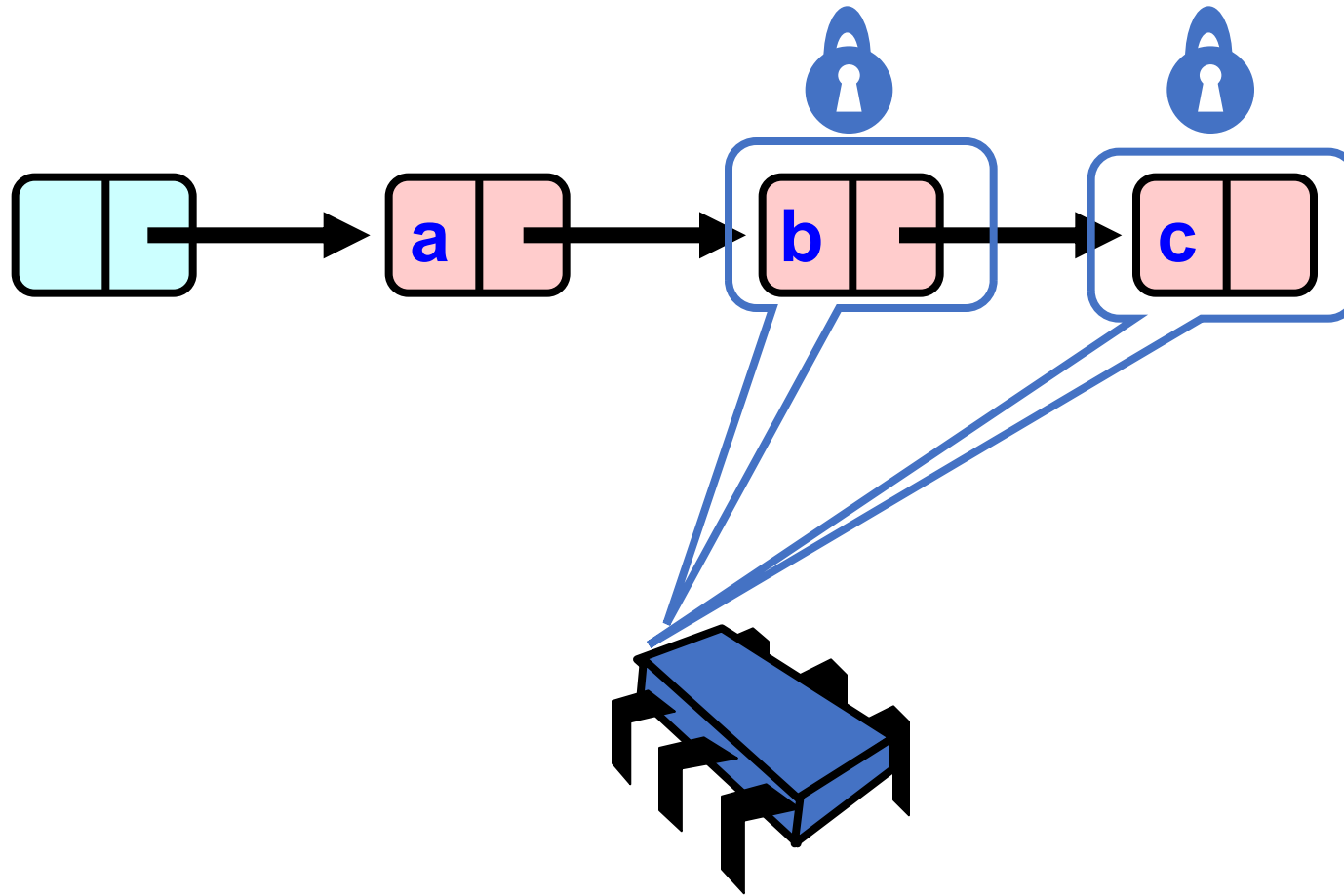
Hand-over-Hand locking



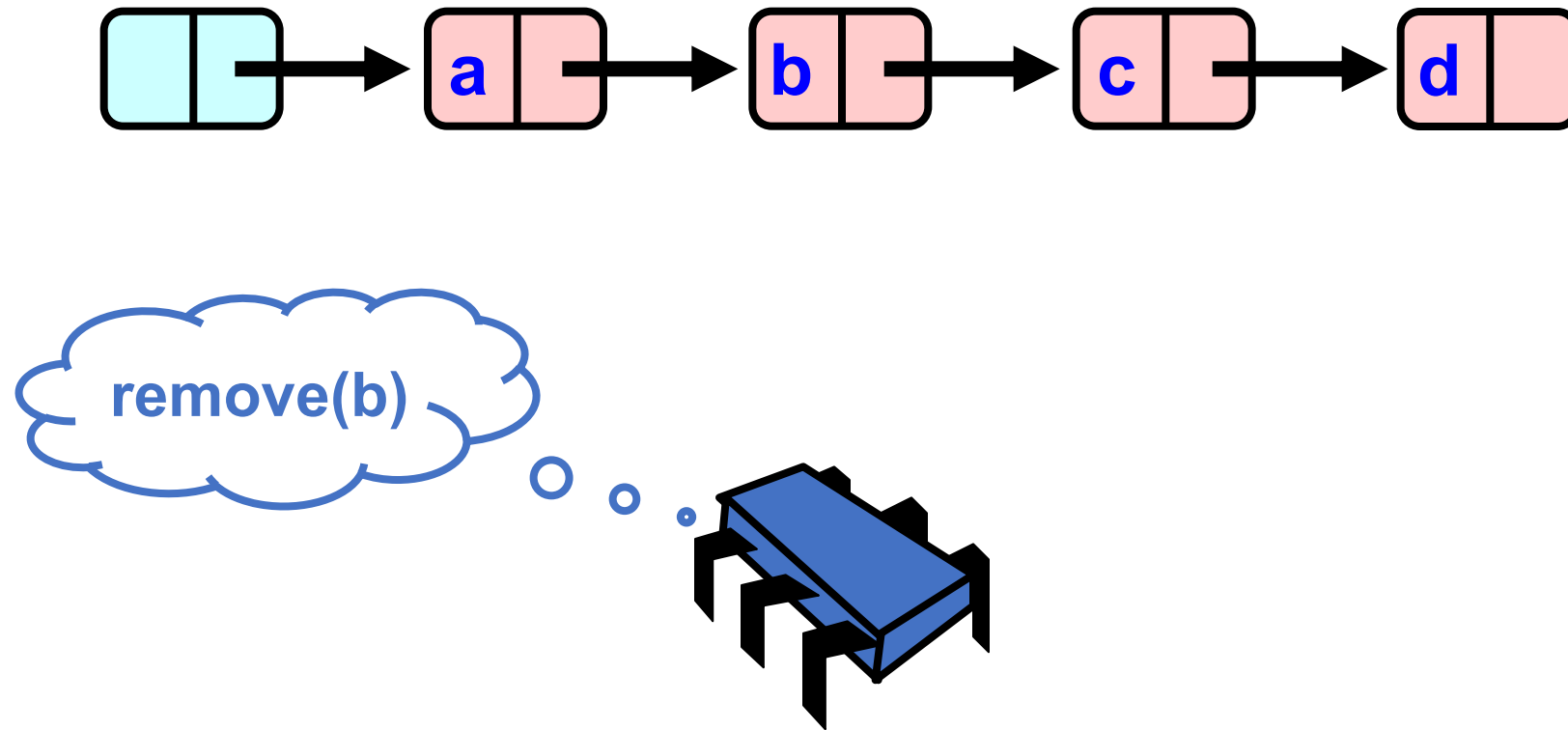
Hand-over-Hand locking



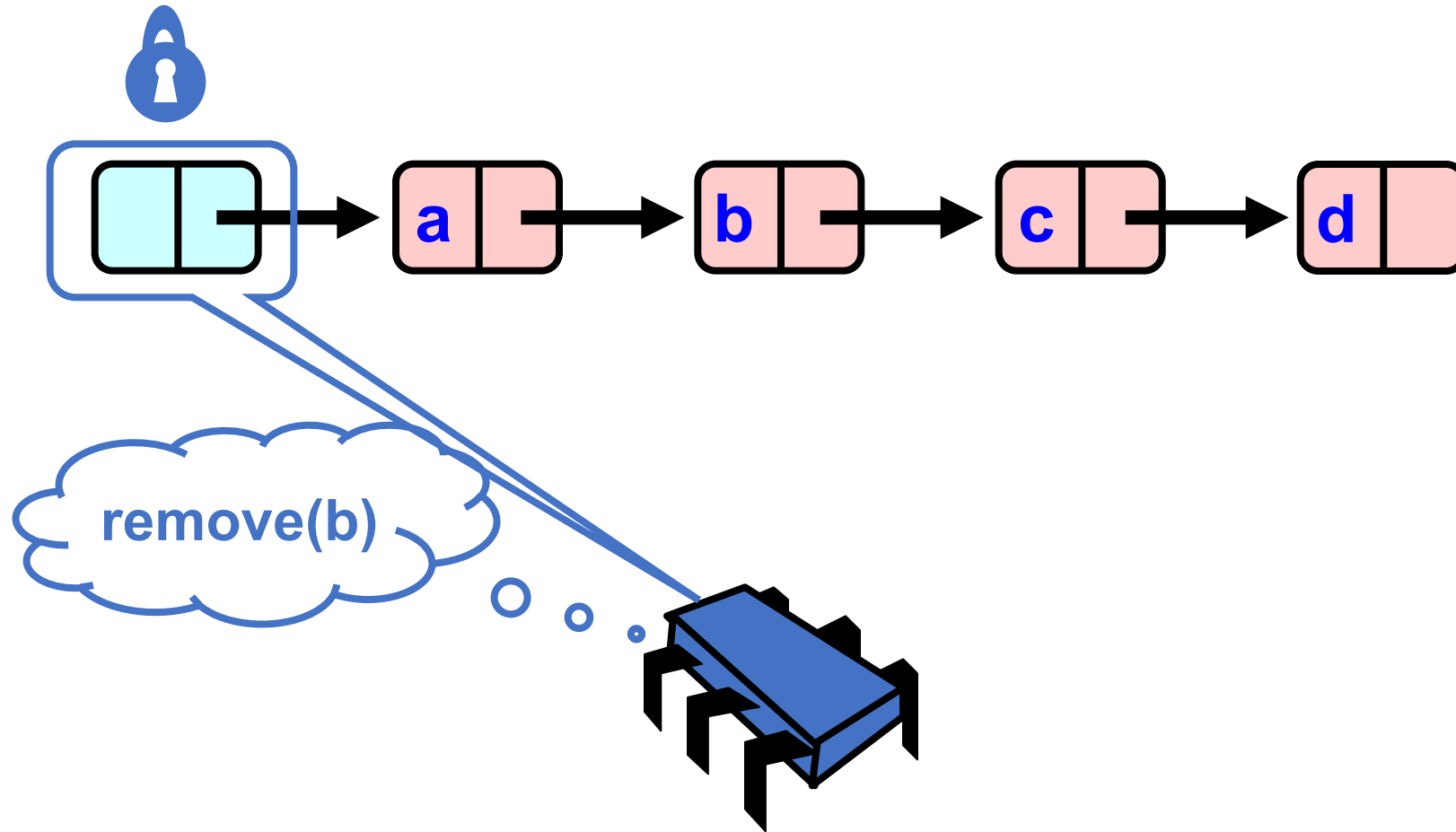
Hand-over-Hand locking



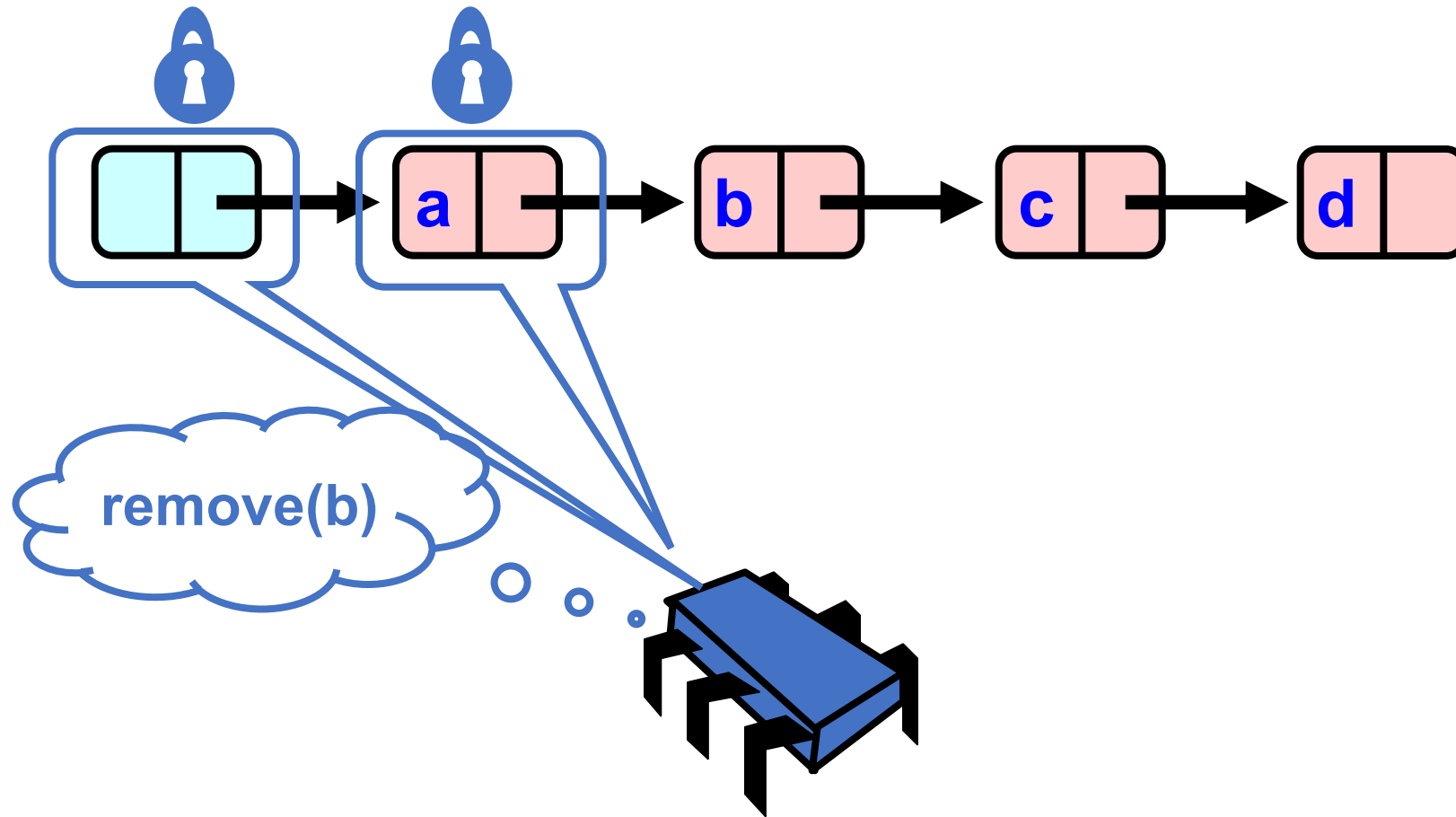
Removing a Node



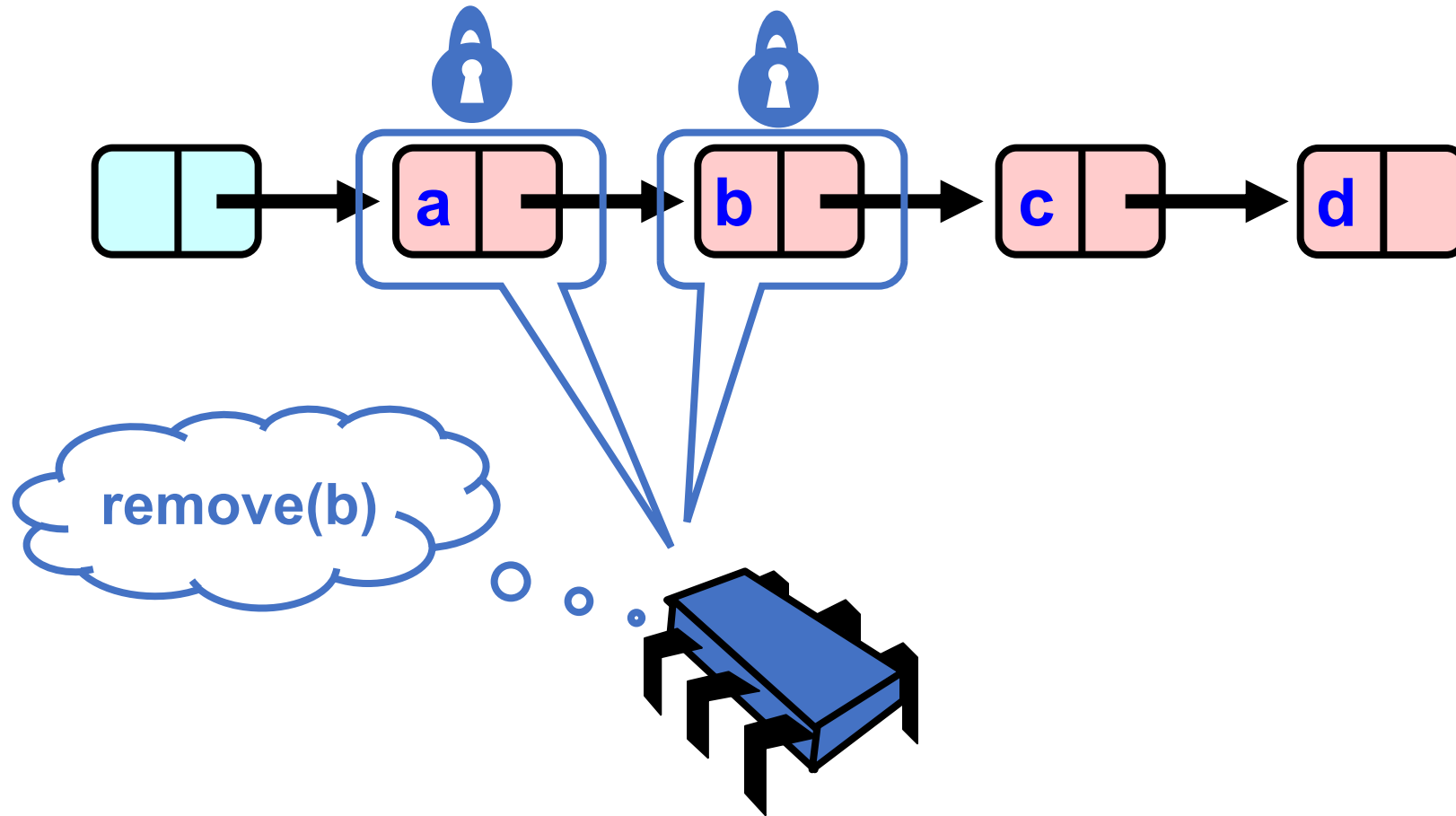
Removing a Node



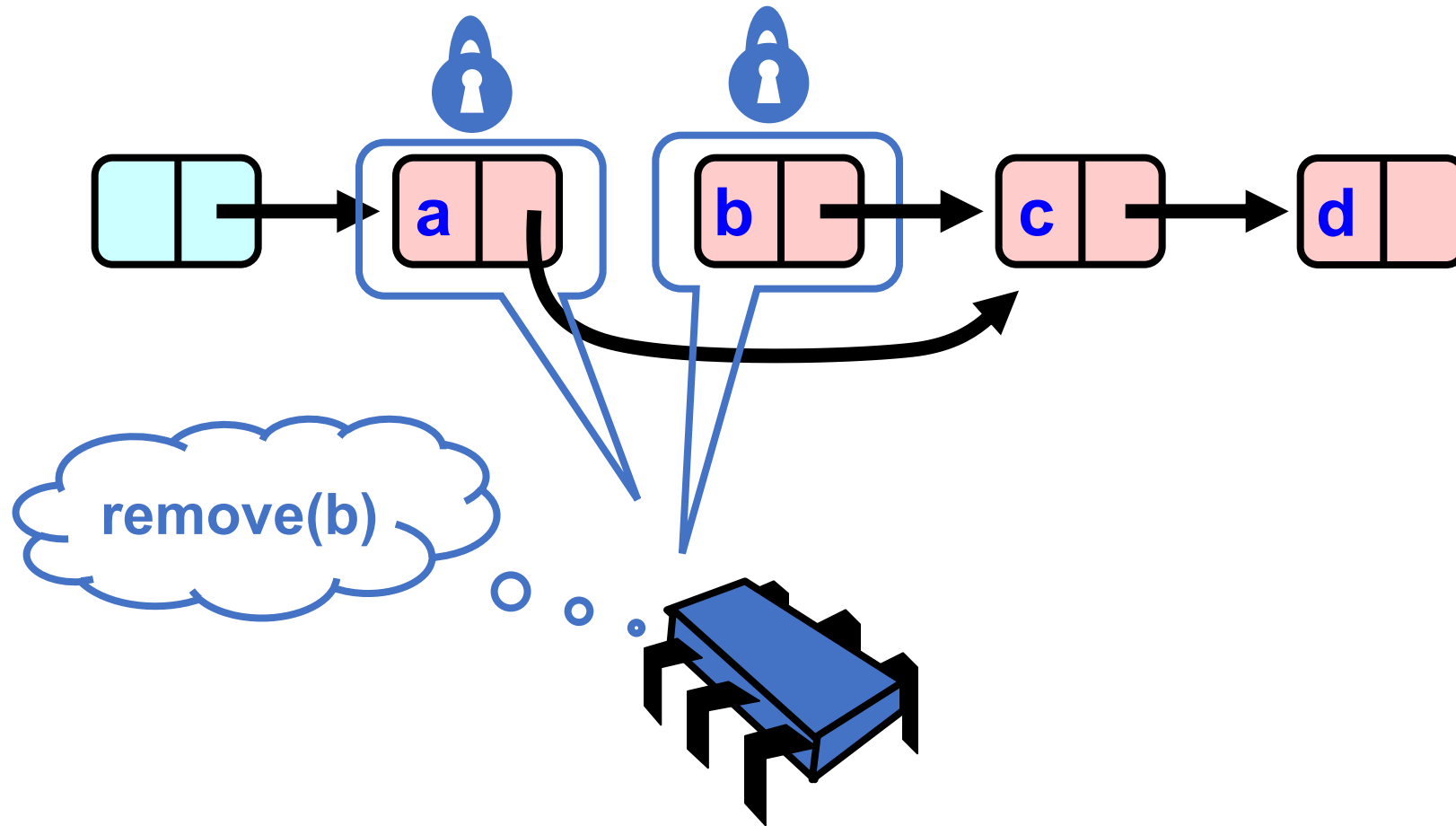
Removing a Node



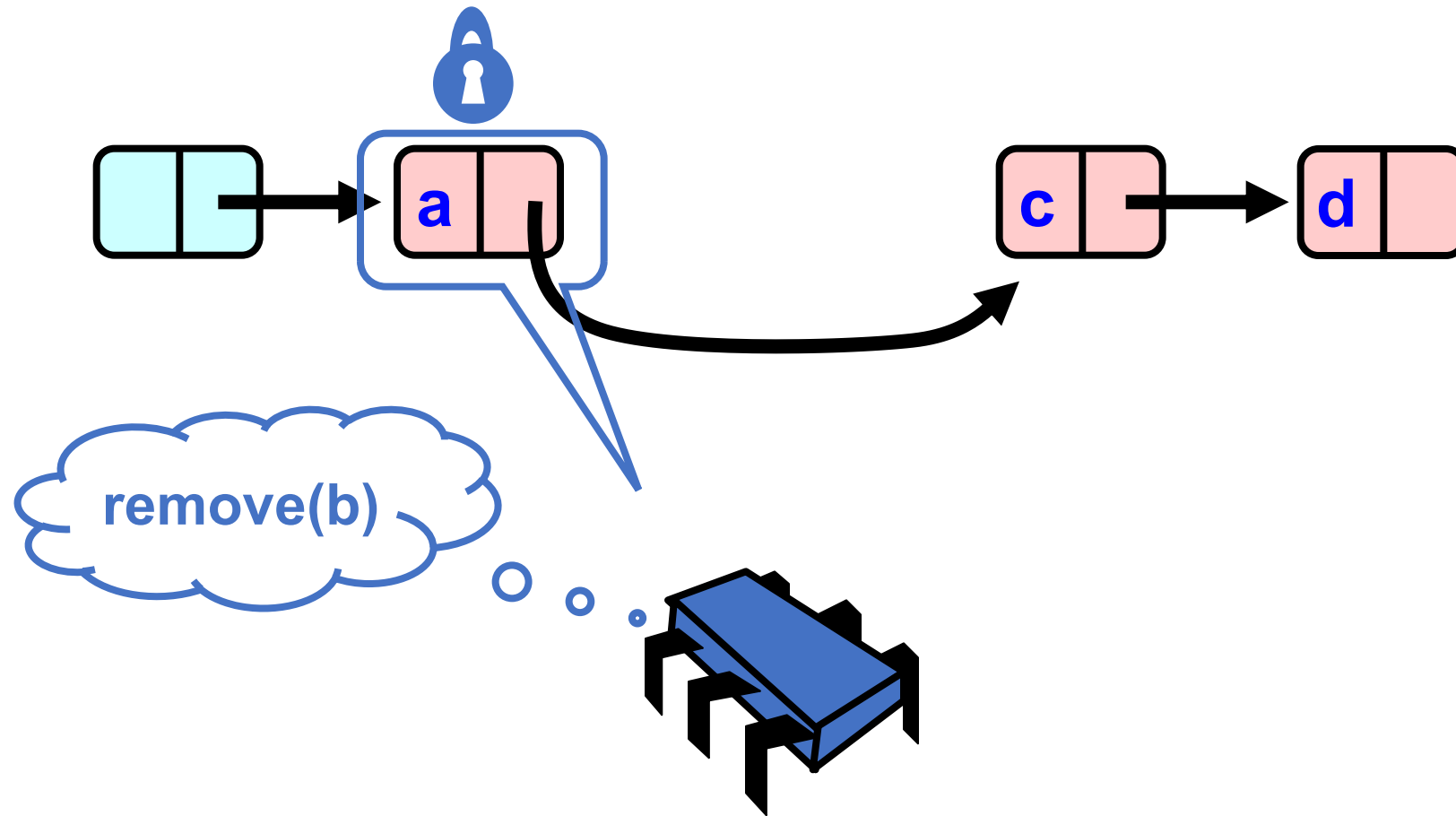
Removing a Node



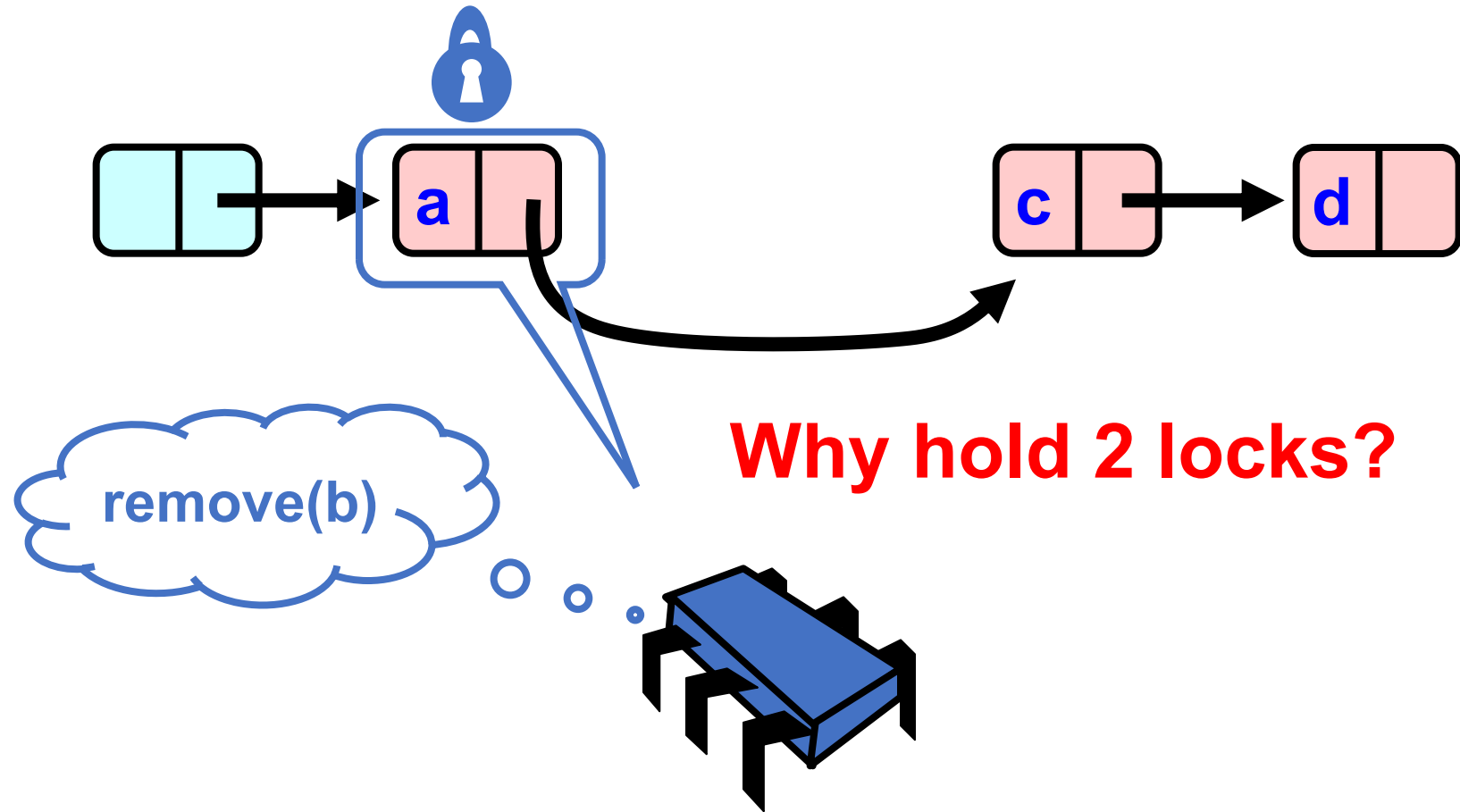
Removing a Node



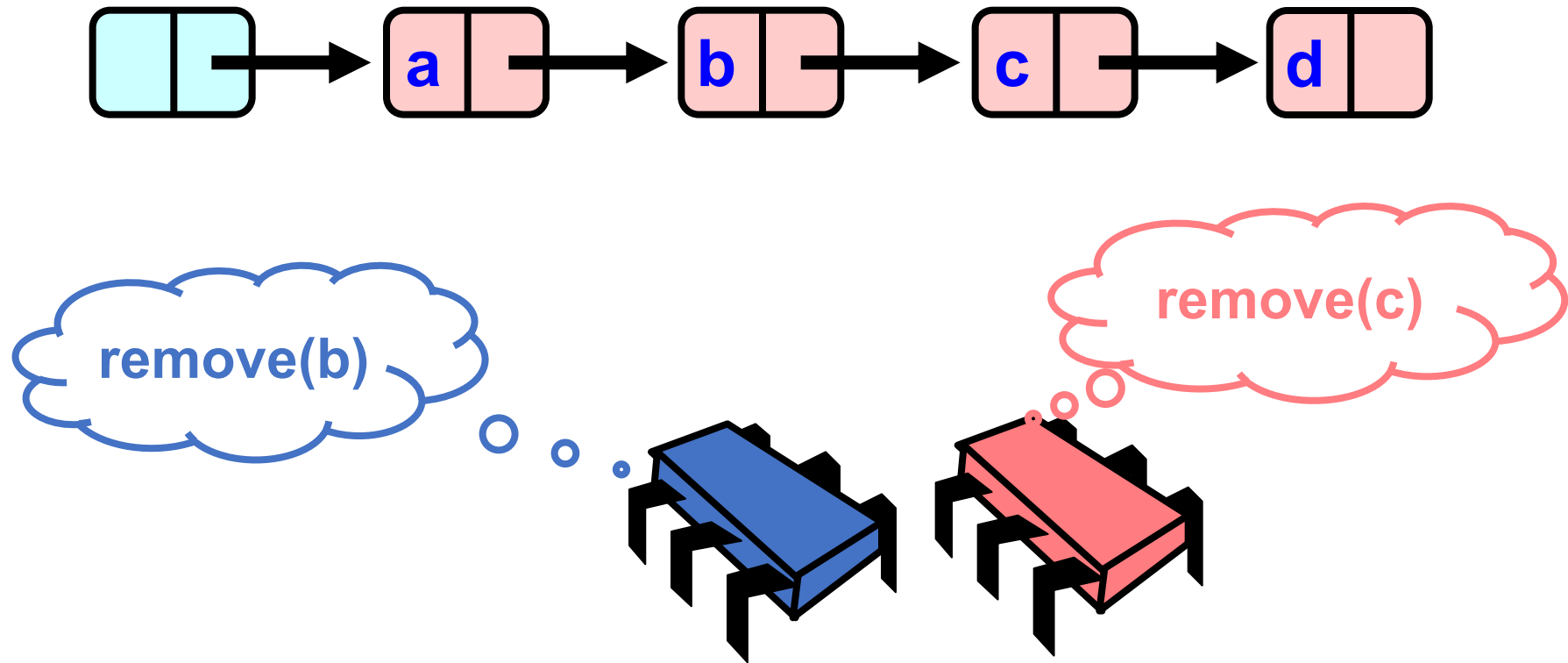
Removing a Node



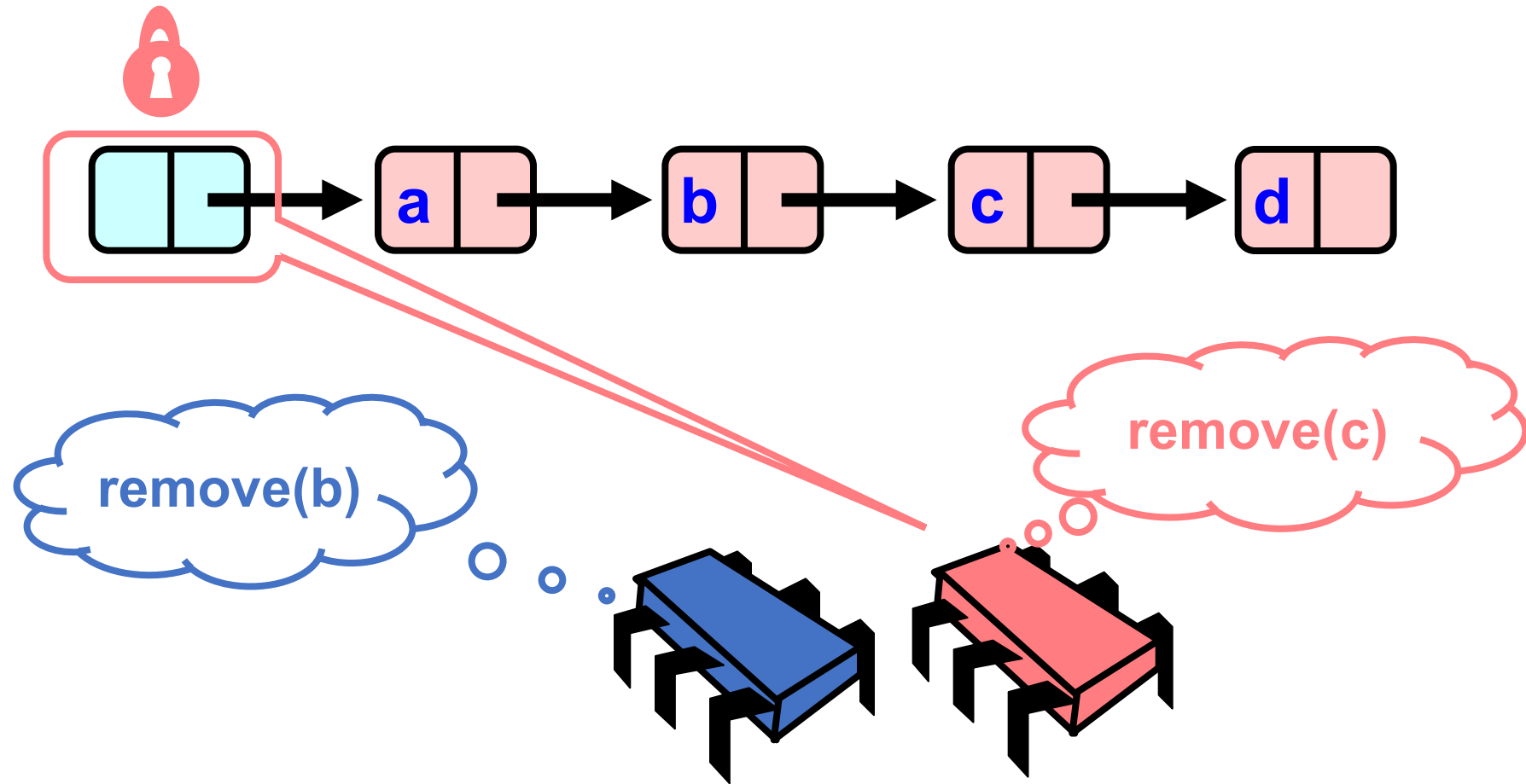
Removing a Node



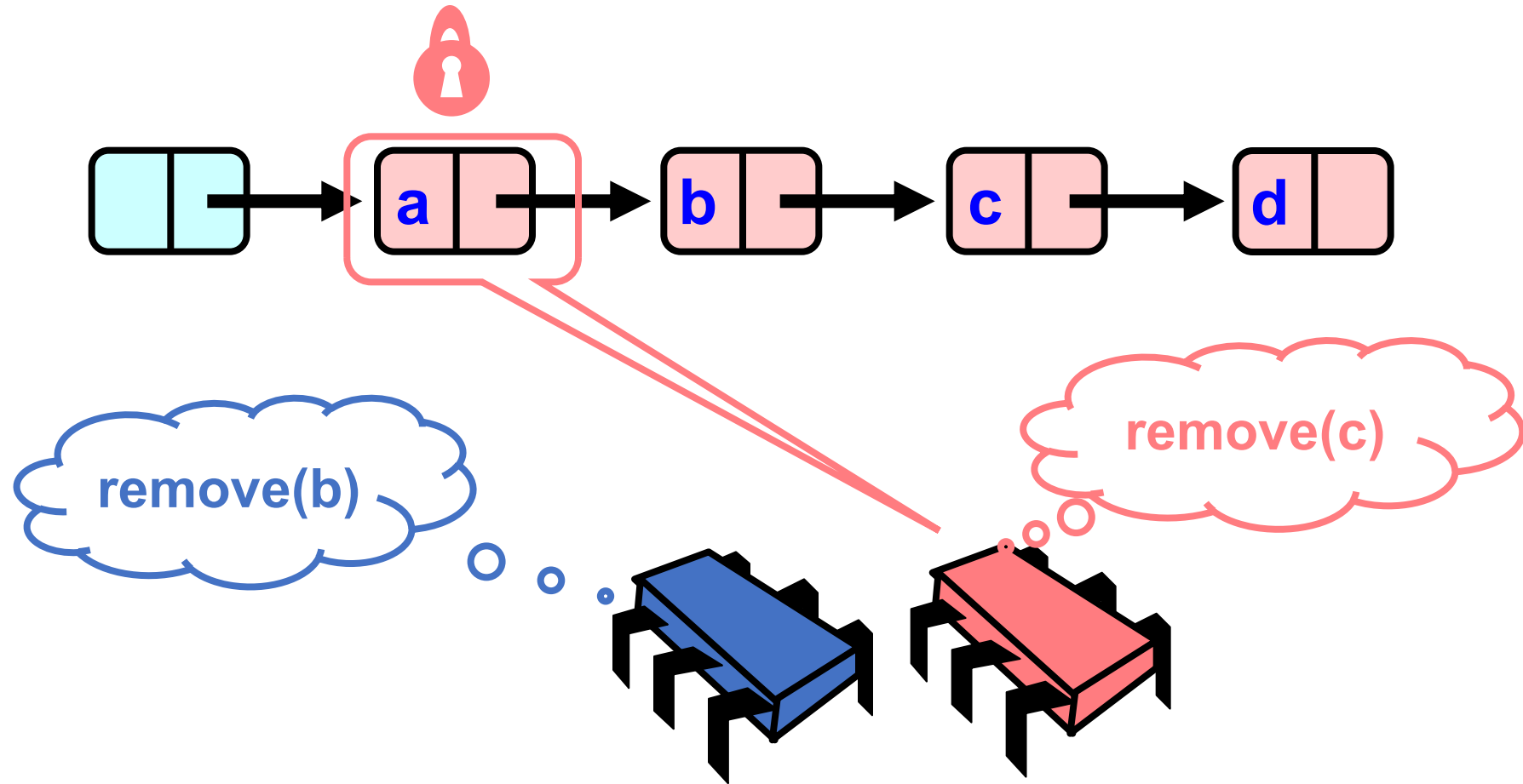
Concurrent Removes



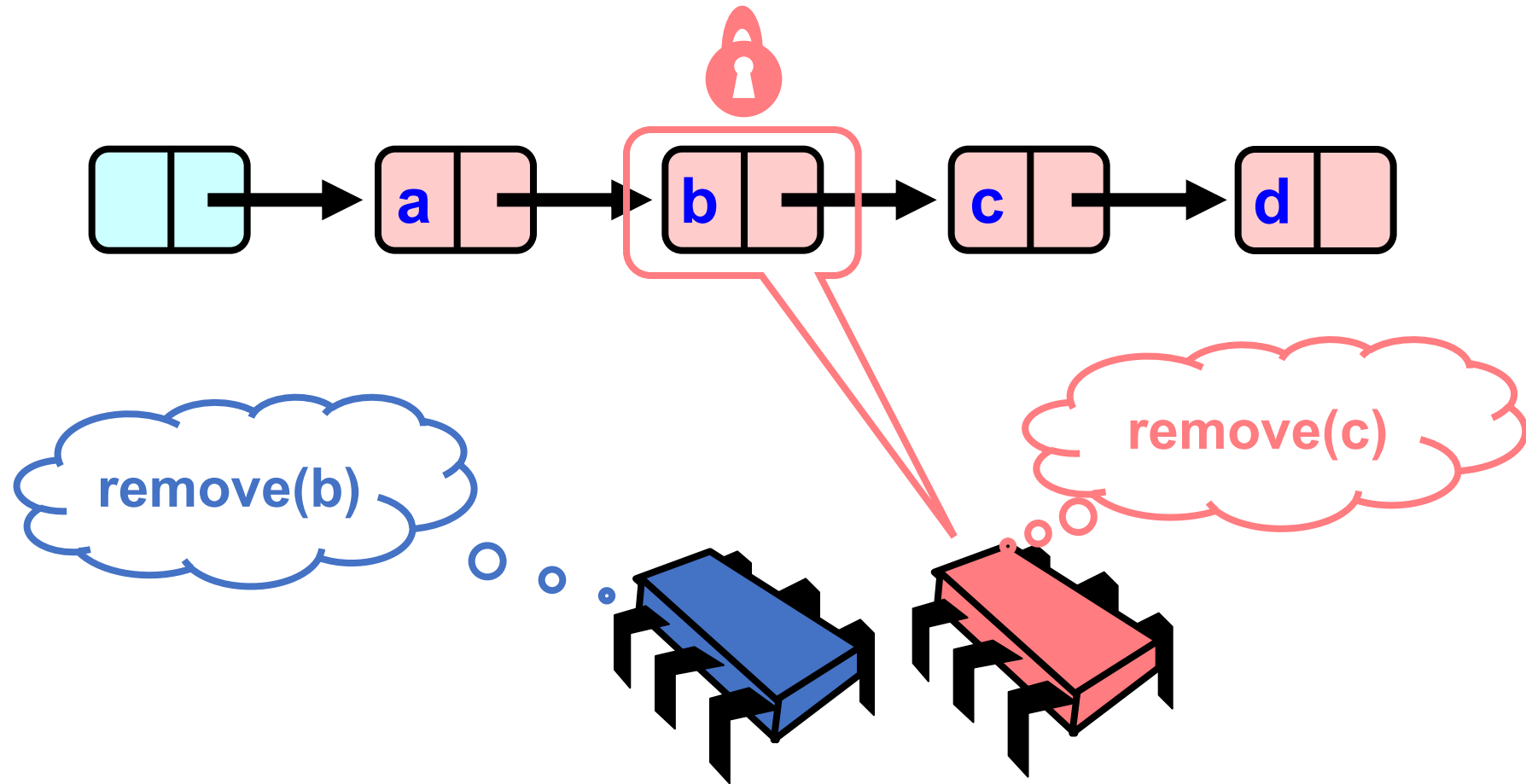
Concurrent Removes



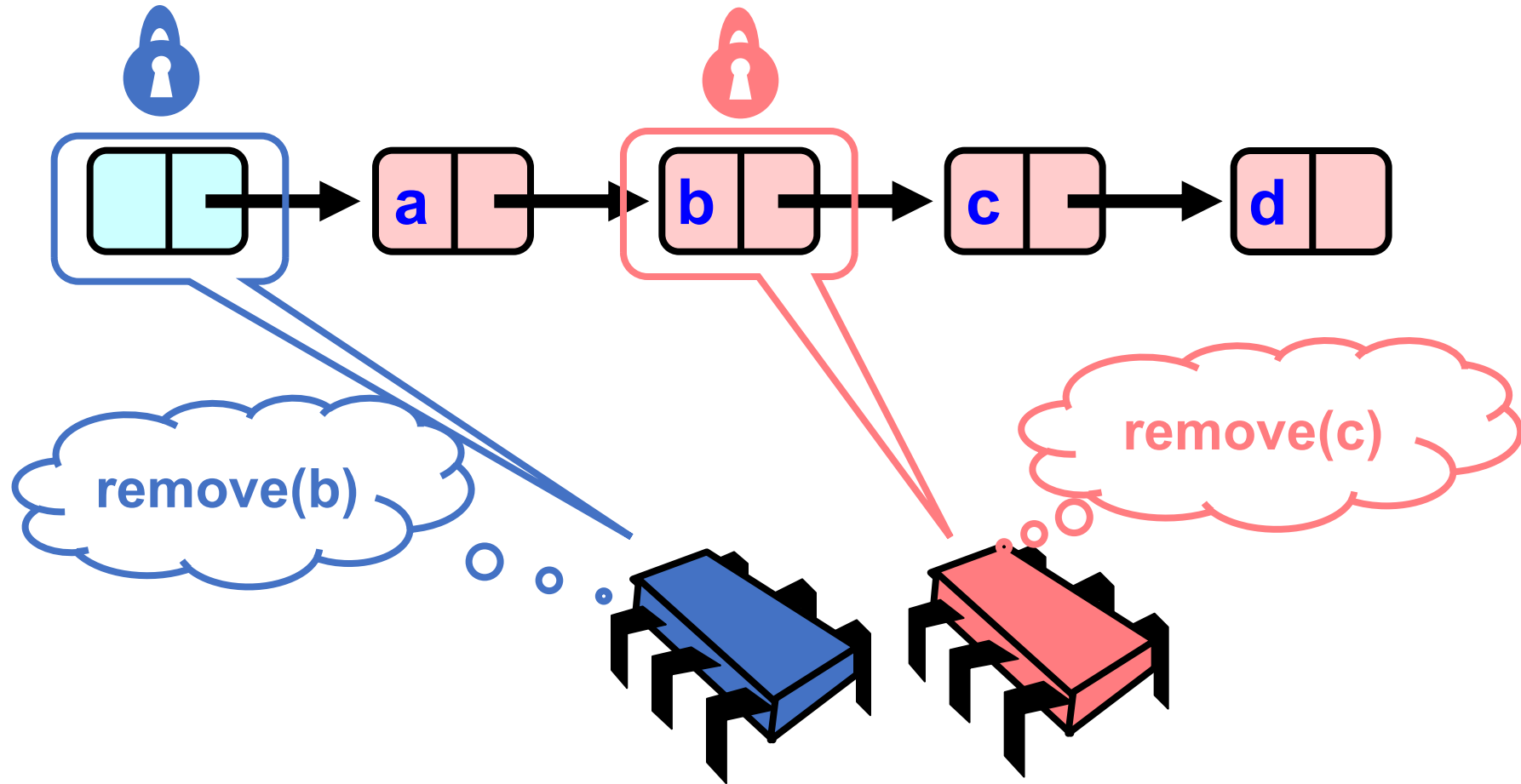
Concurrent Removes



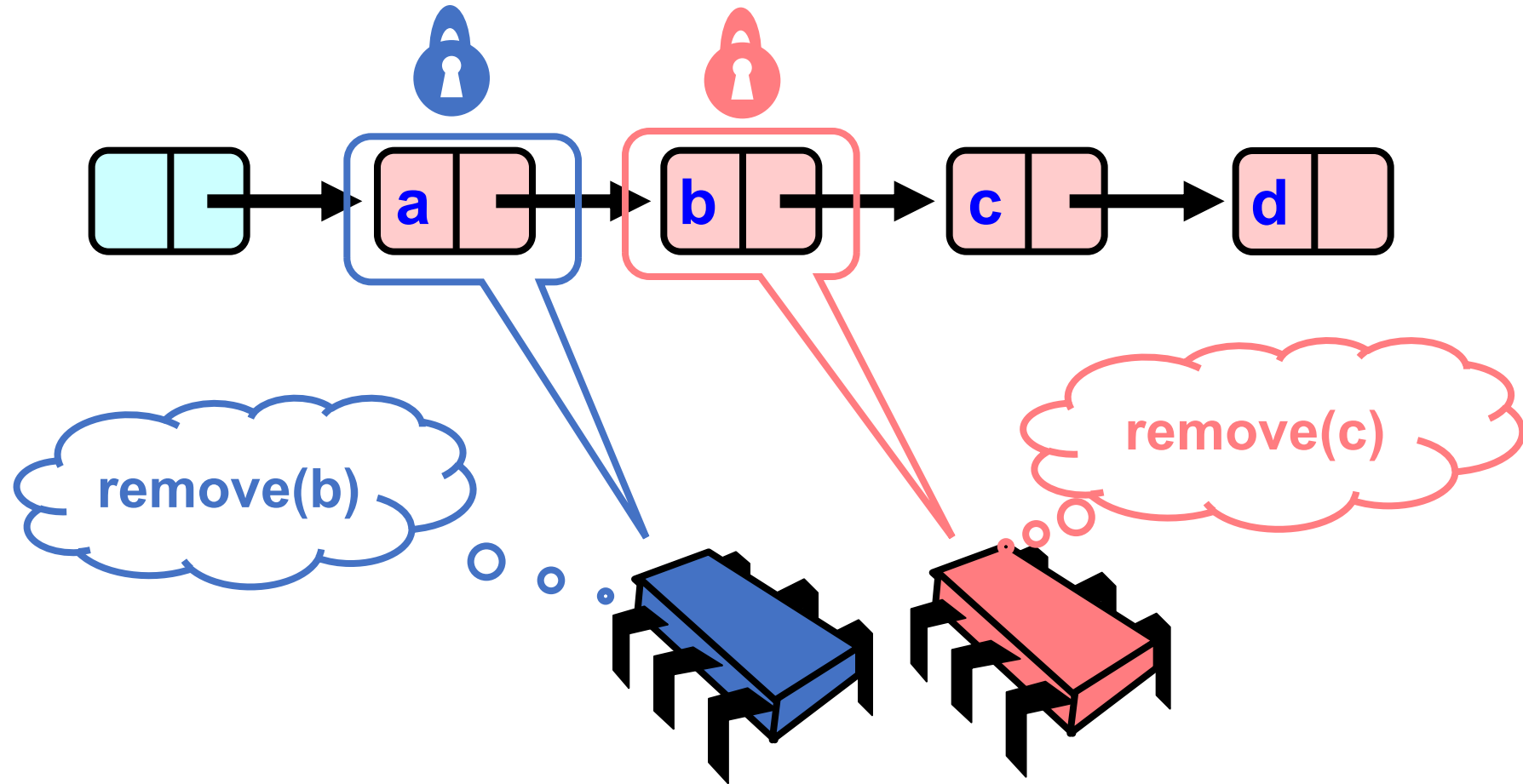
Concurrent Removes



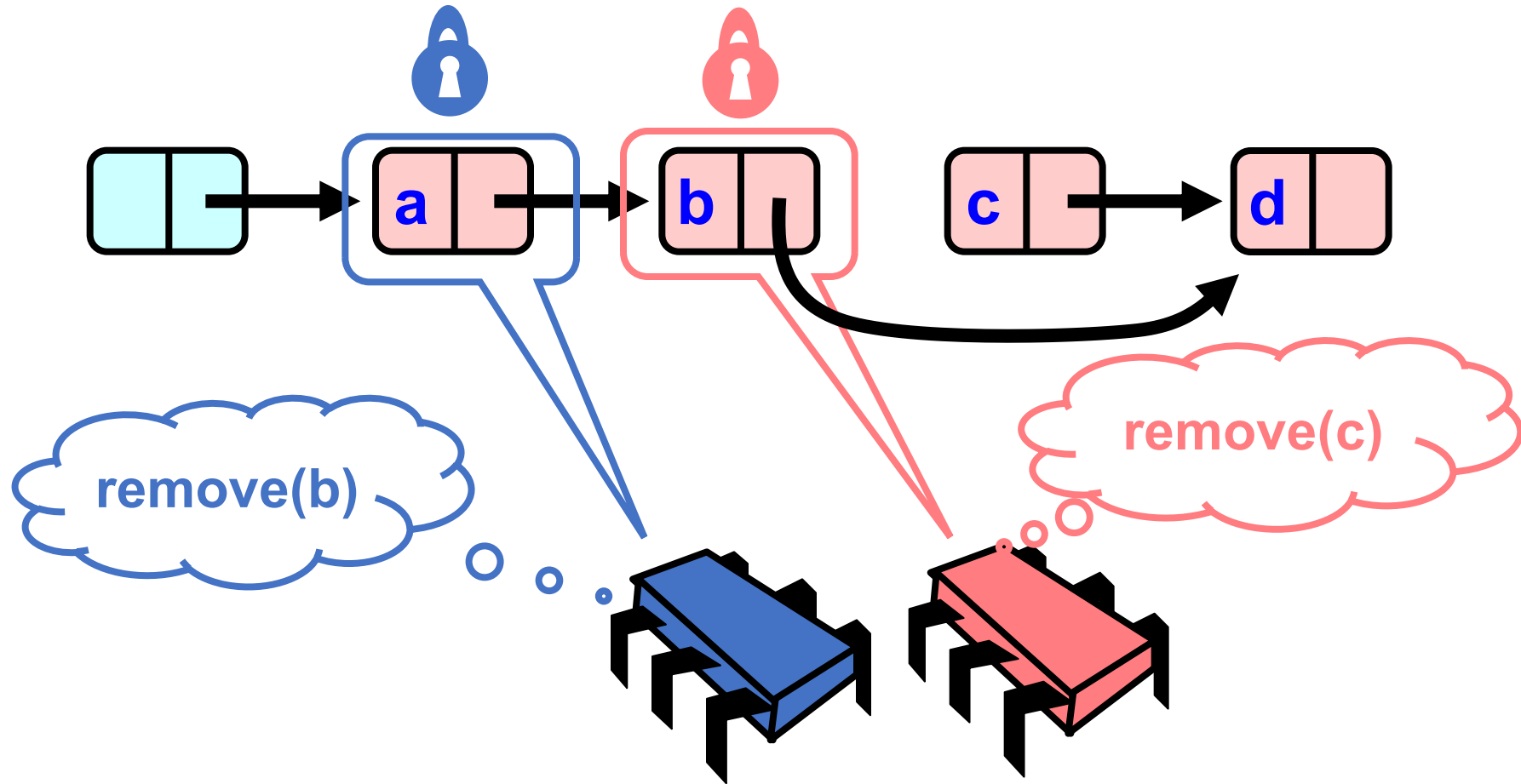
Concurrent Removes



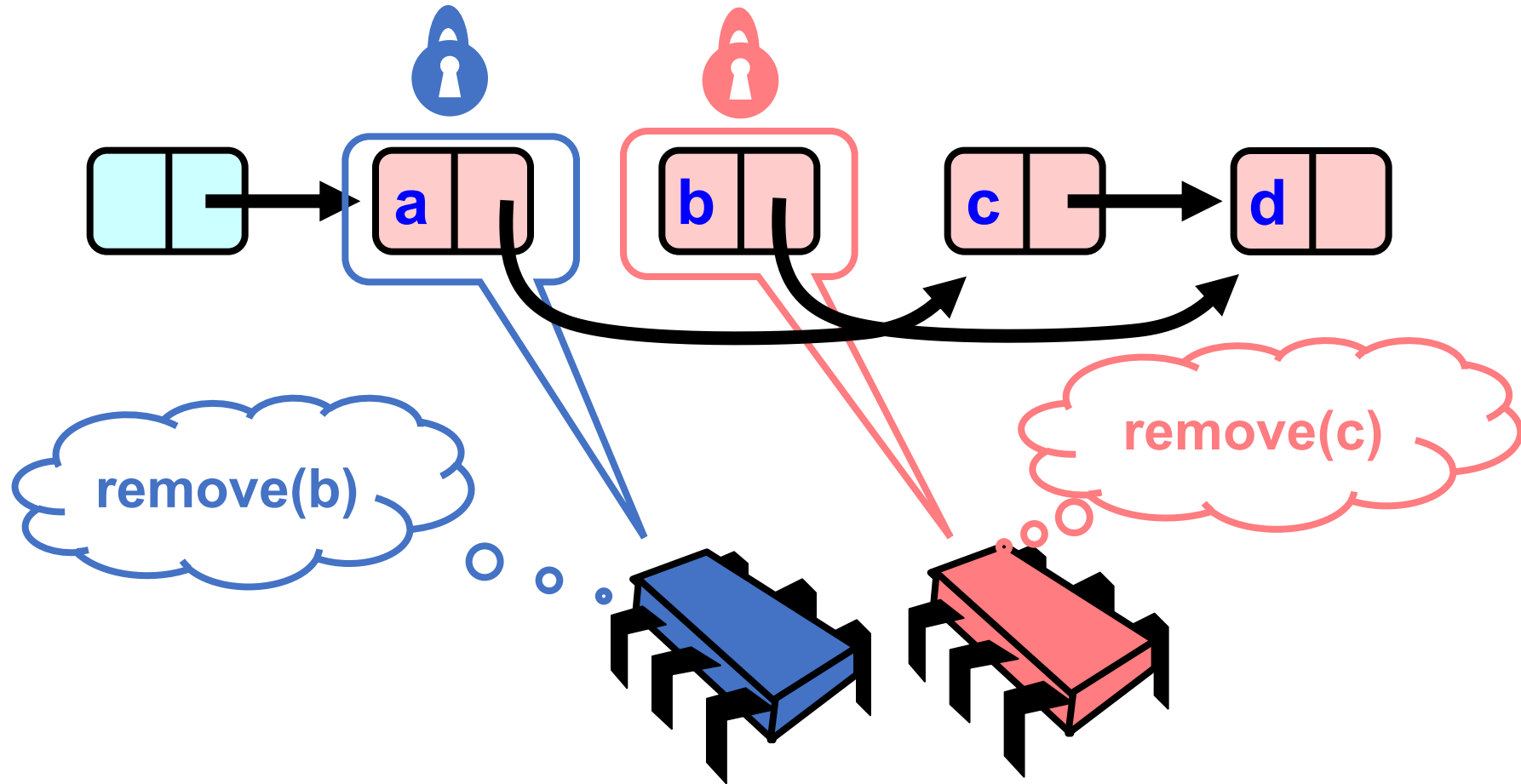
Concurrent Removes



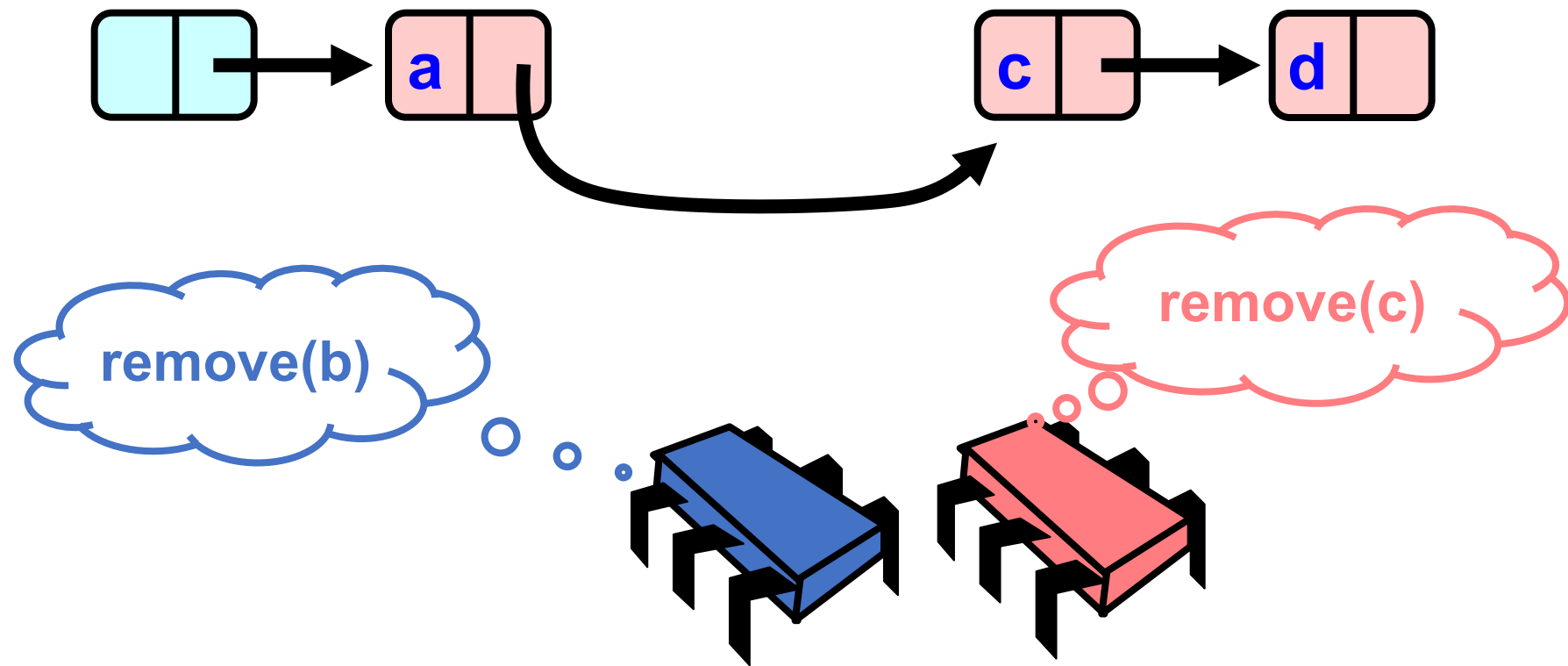
Concurrent Removes



Concurrent Removes

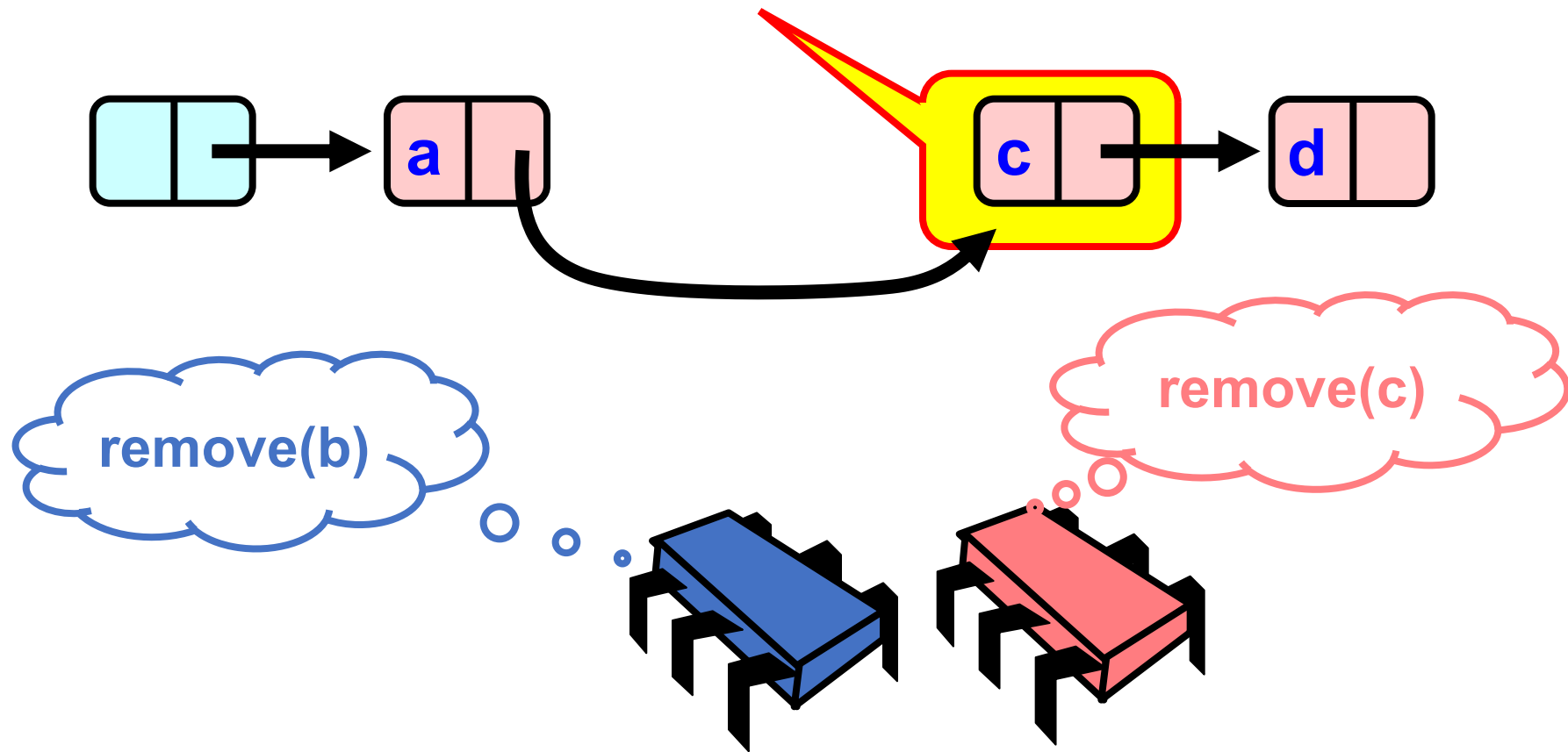


Uh, Oh



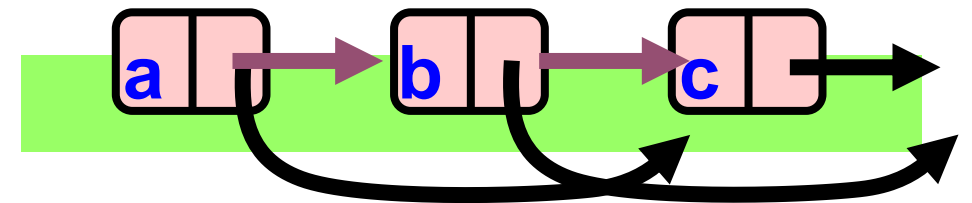
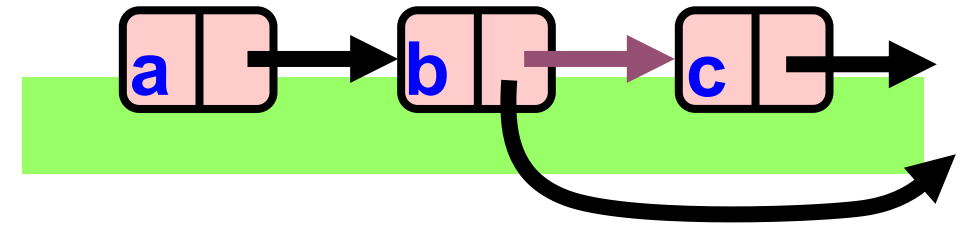
Uh, Oh

Bad news, c not removed

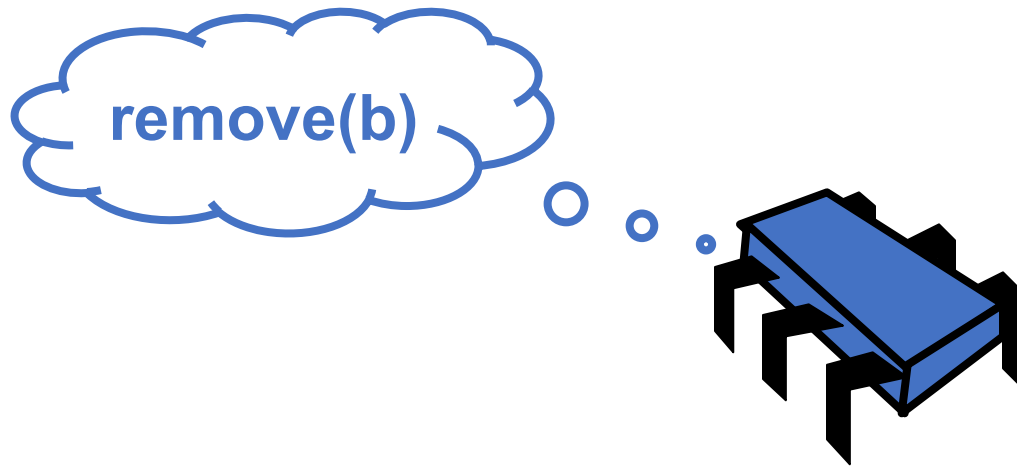
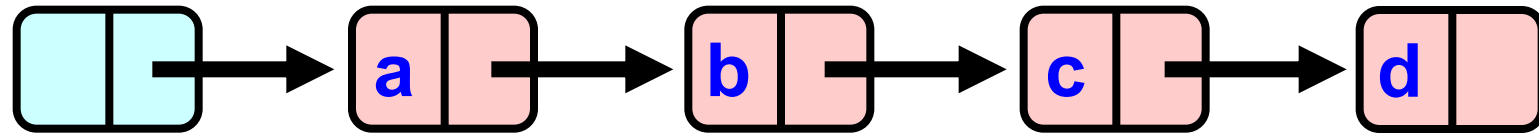


Problem

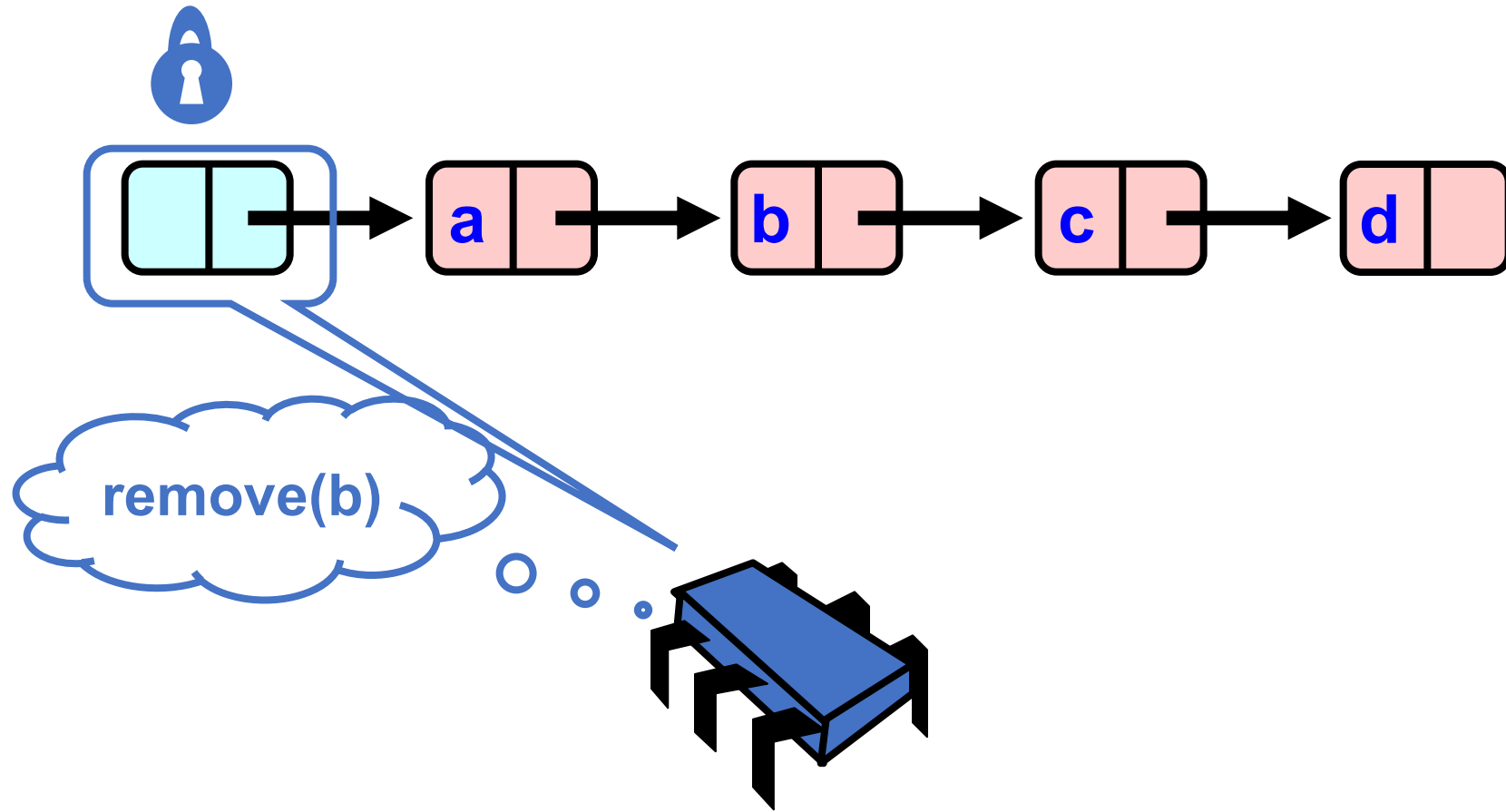
- To delete node c
 - Swing node b's next field to d
- Problem is,
 - ***Data conflict:***
 - Someone deleting b concurrently could direct a pointer to C



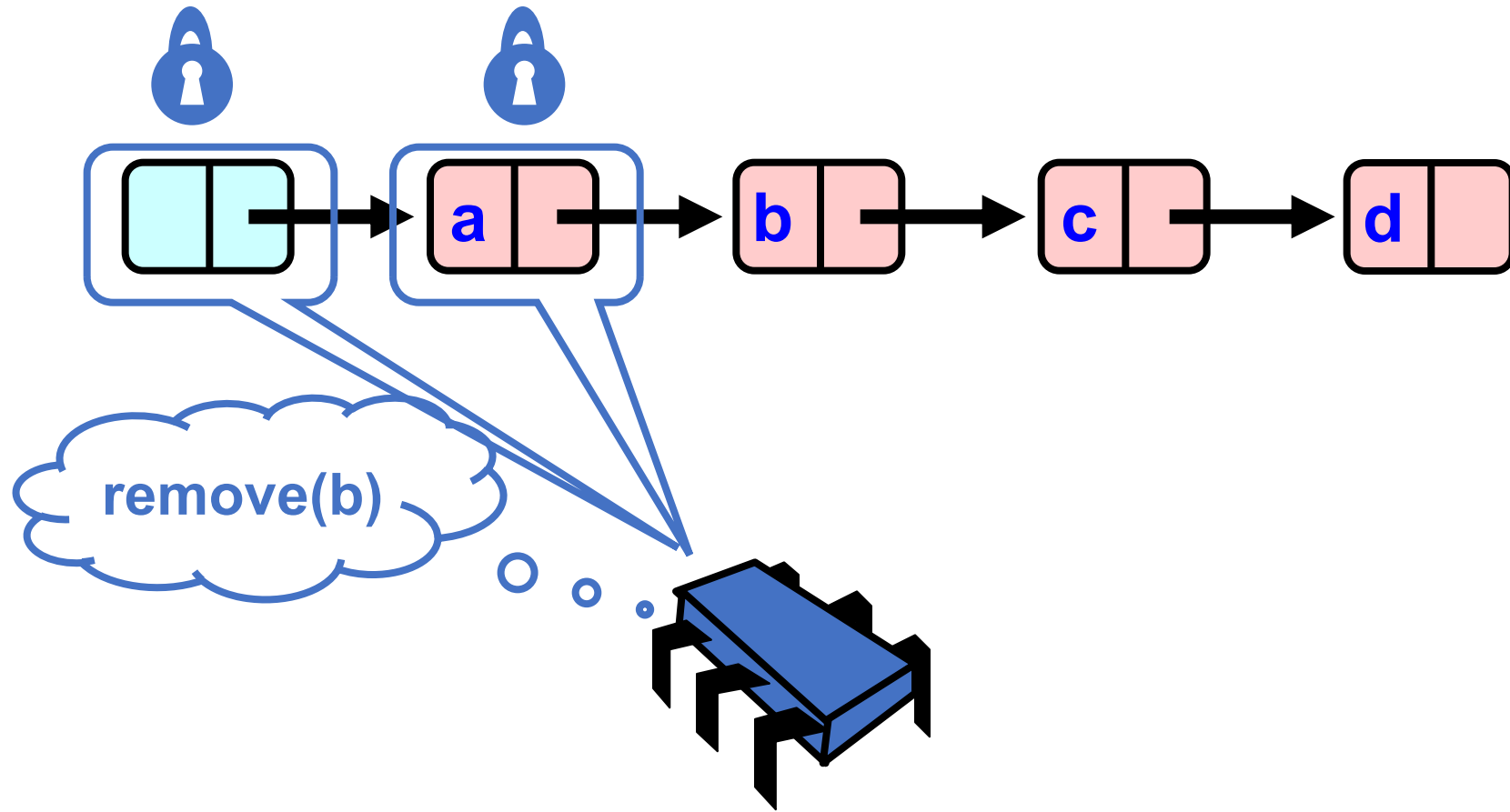
Hand-Over-Hand Again



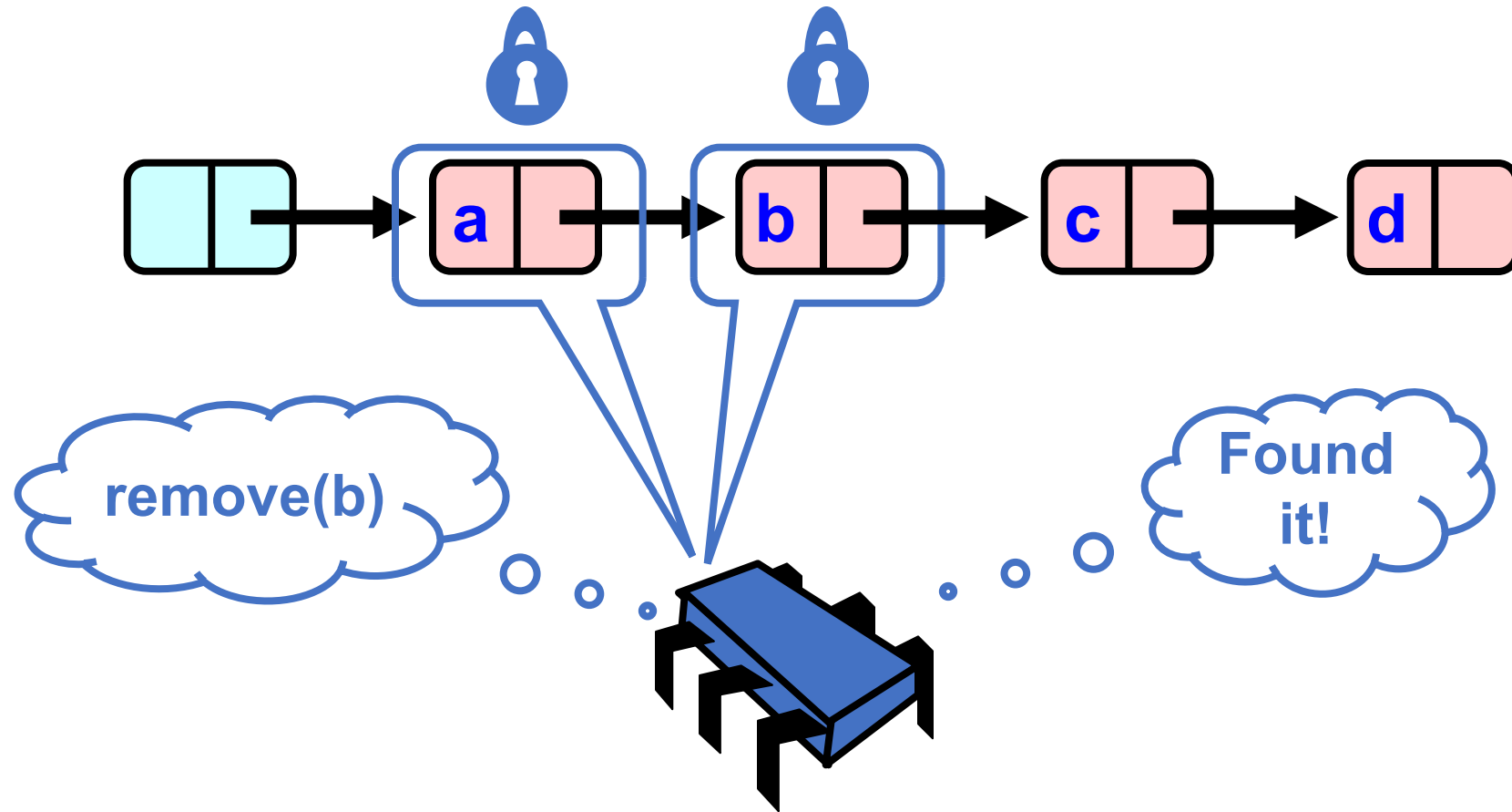
Hand-Over-Hand Again



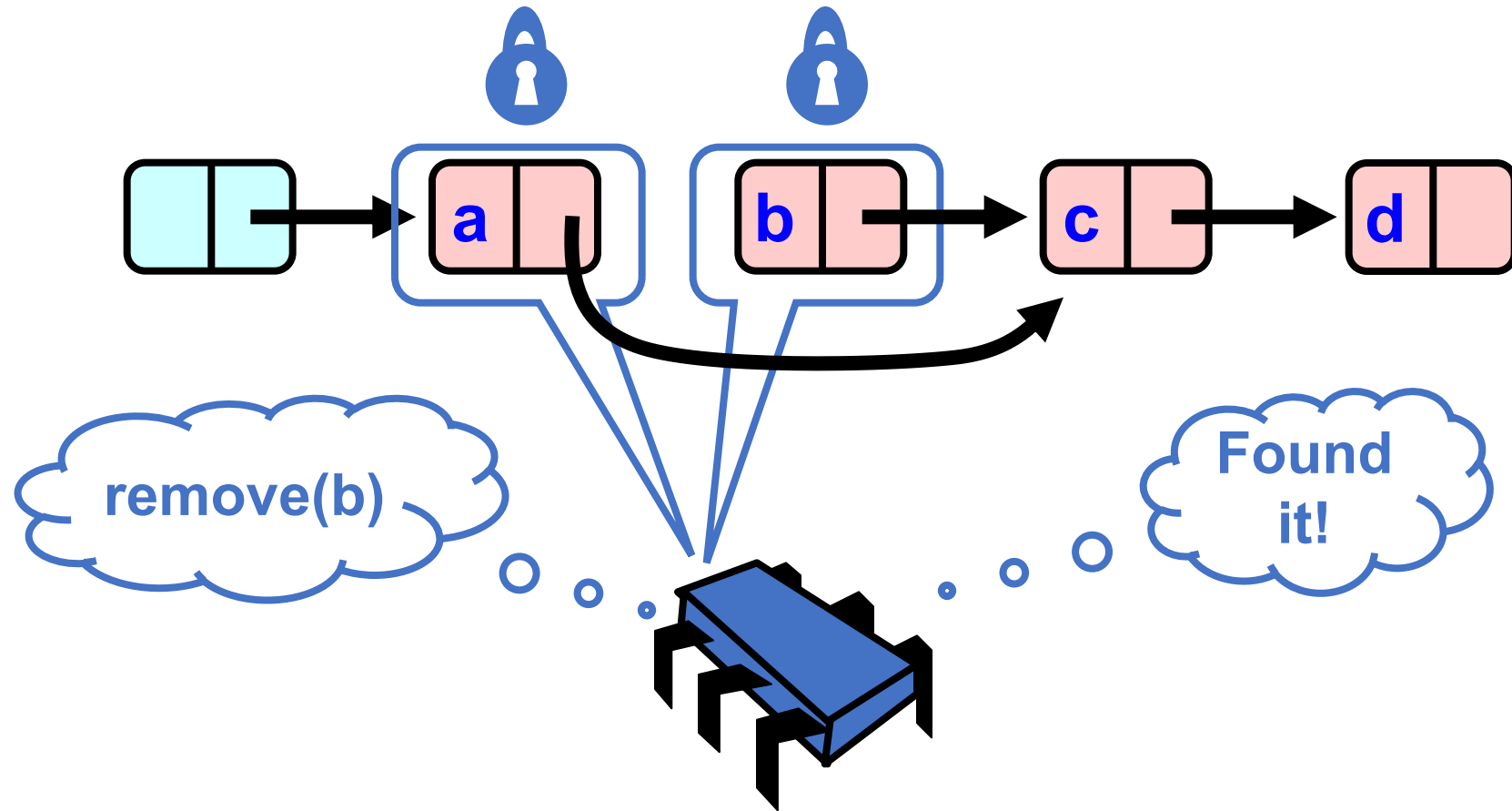
Hand-Over-Hand Again



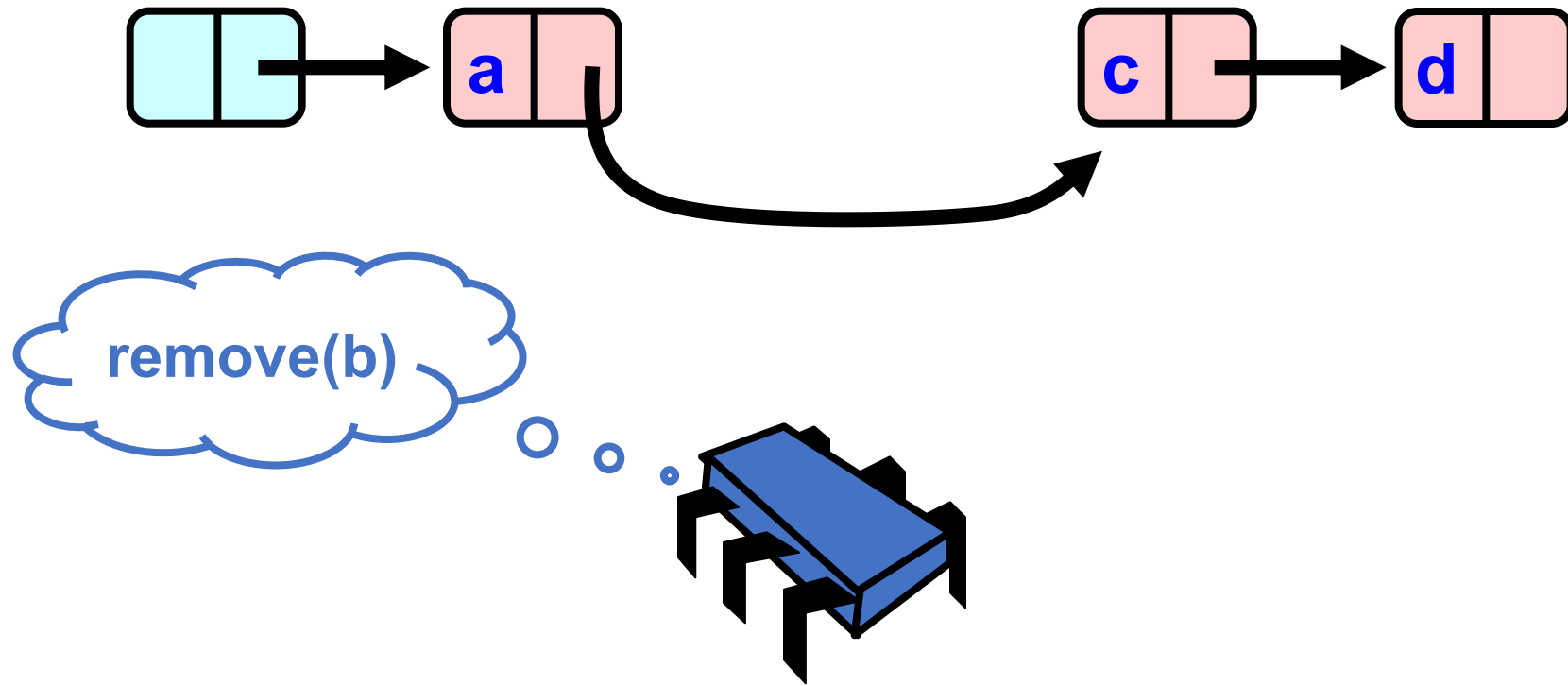
Hand-Over-Hand Again



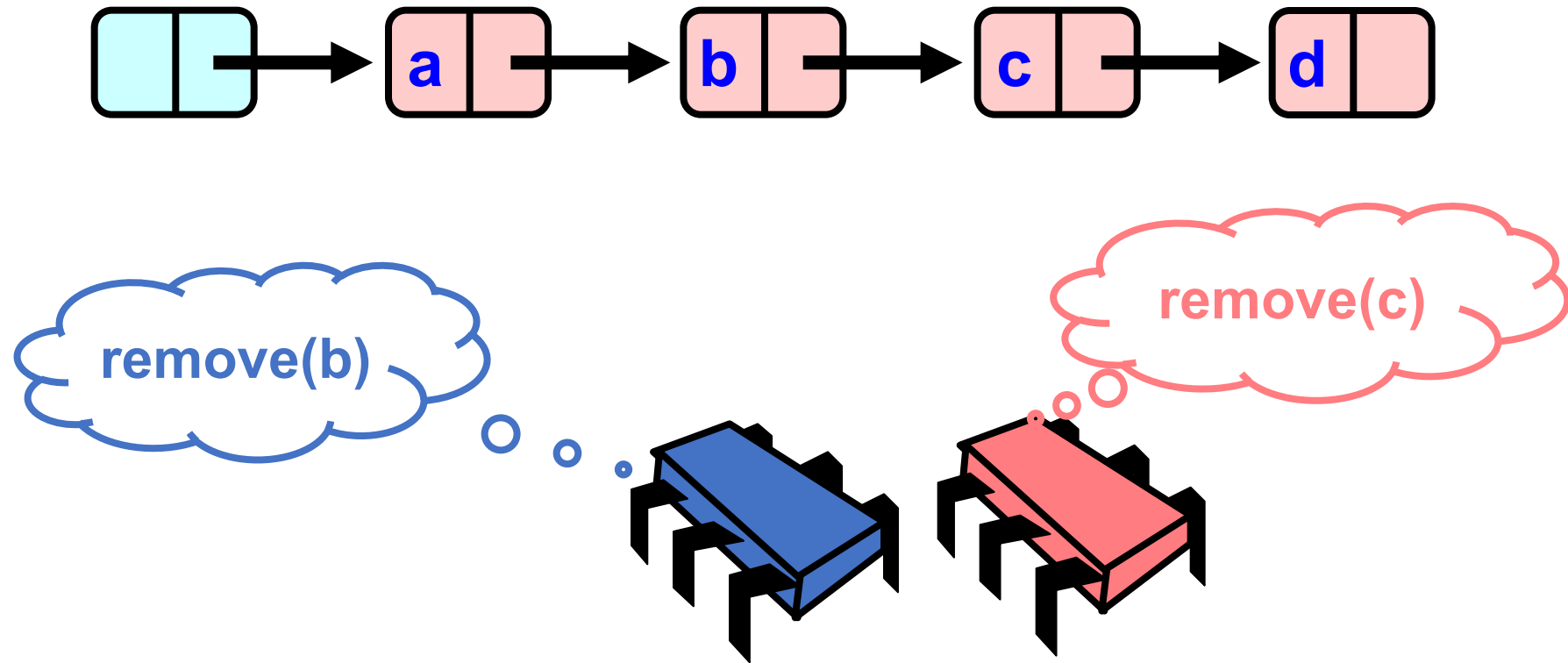
Hand-Over-Hand Again



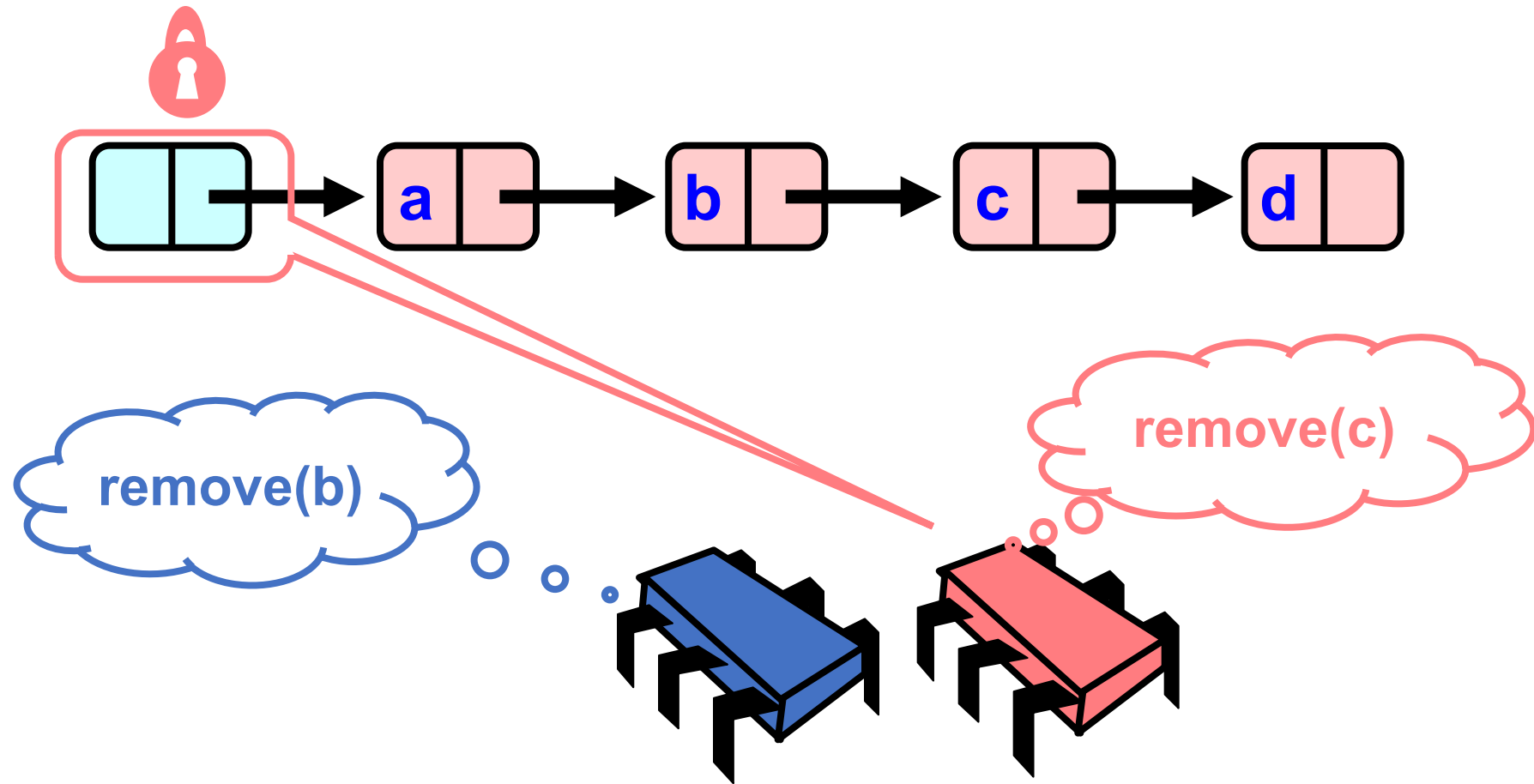
Hand-Over-Hand Again



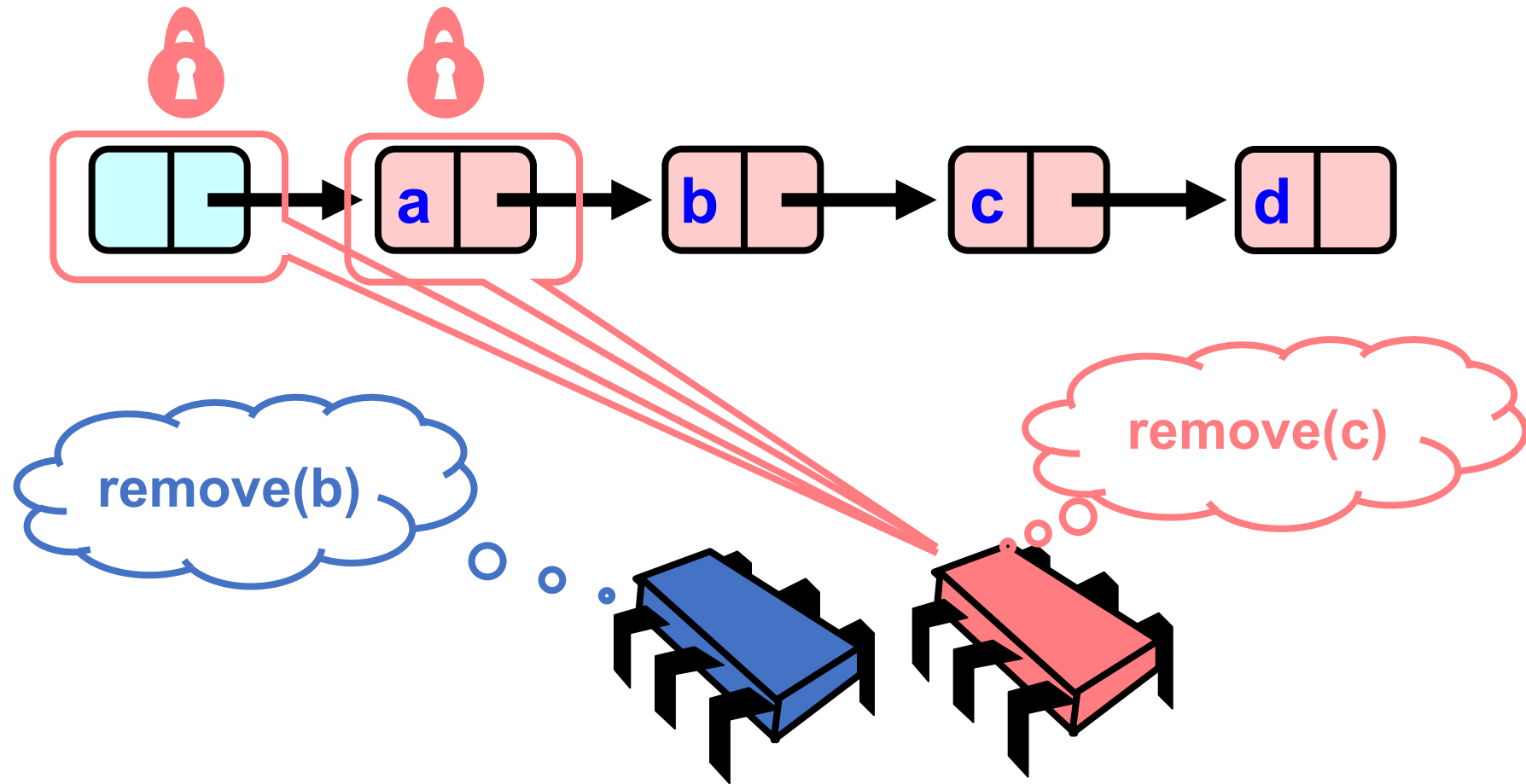
Removing a Node



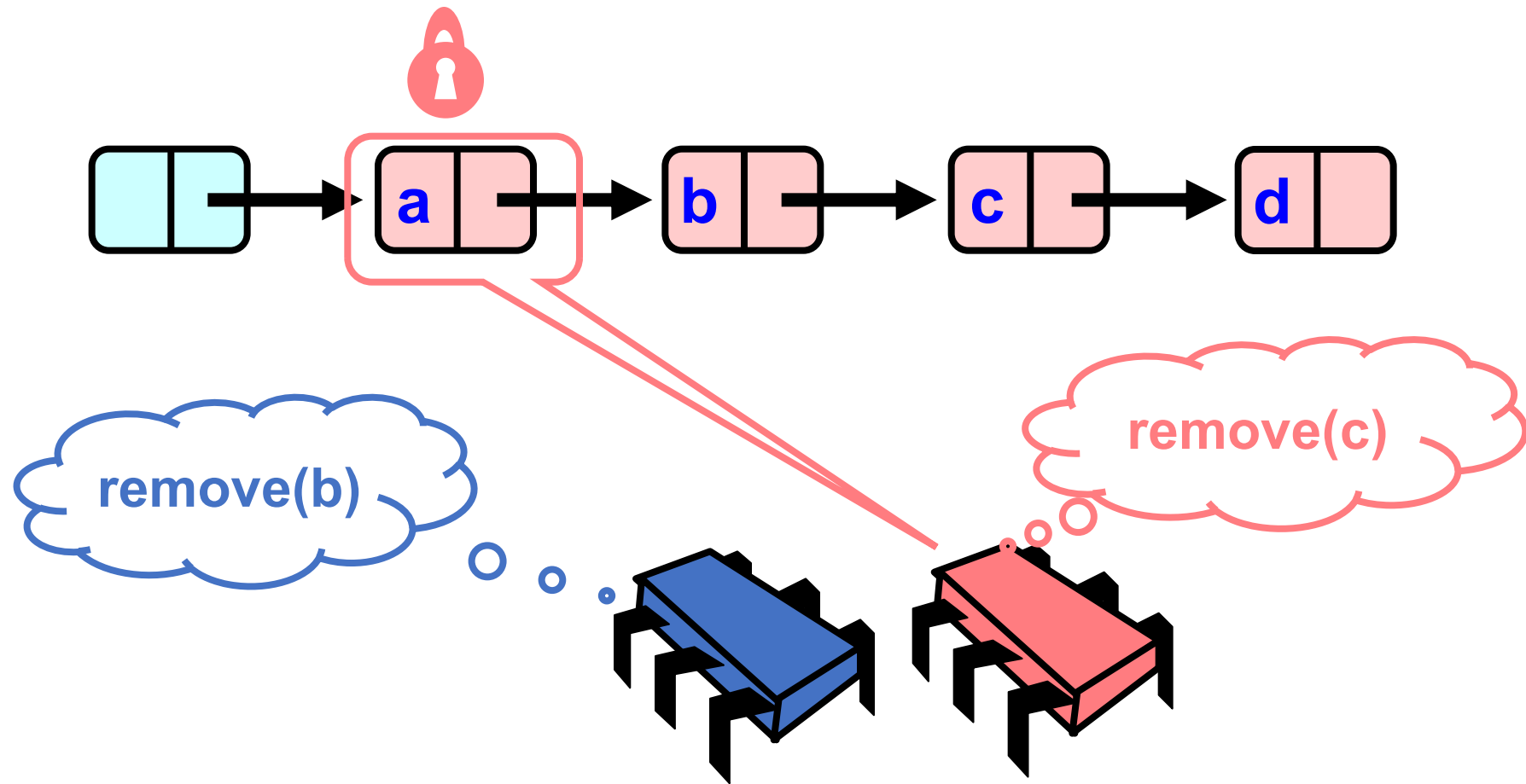
Removing a Node



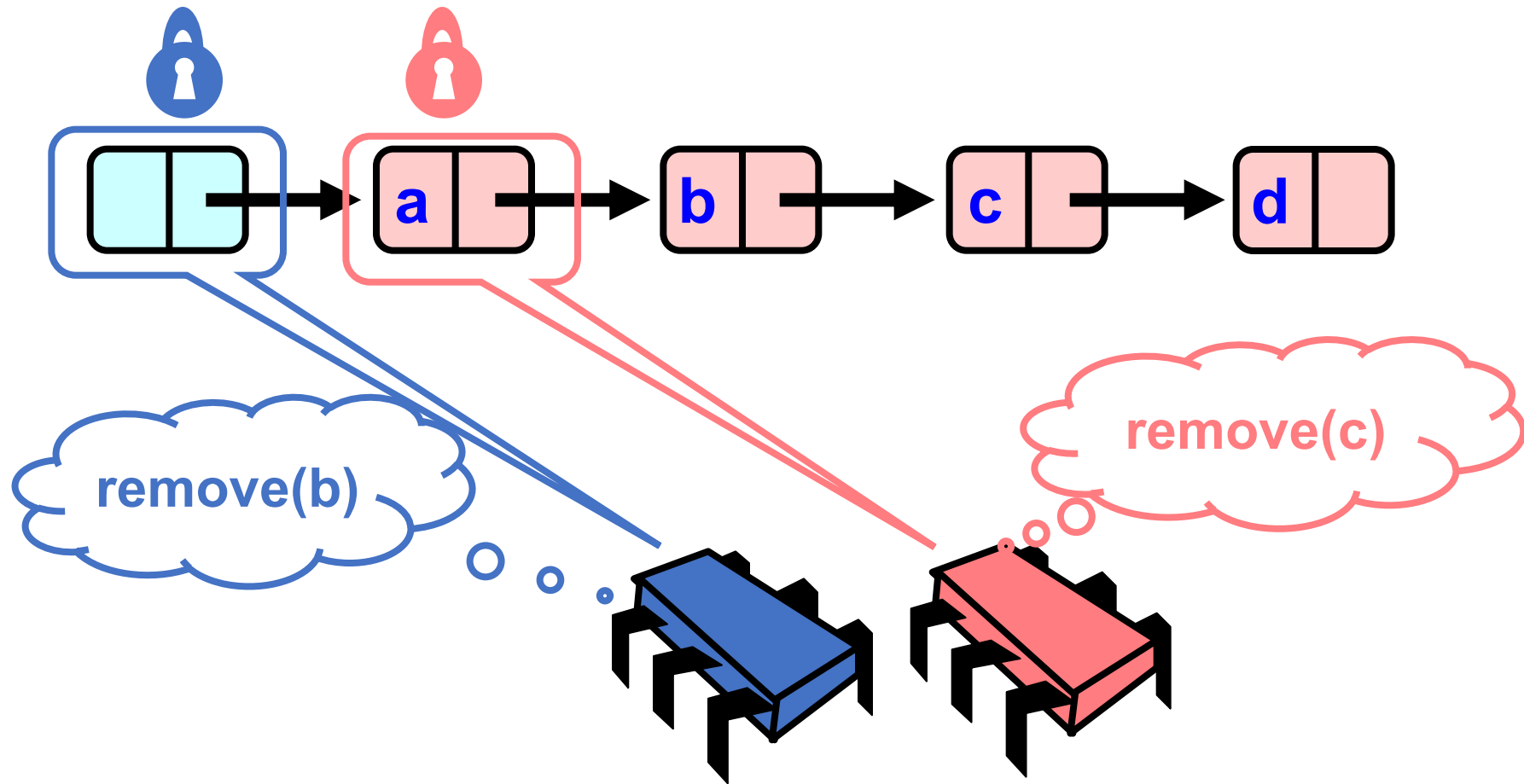
Removing a Node



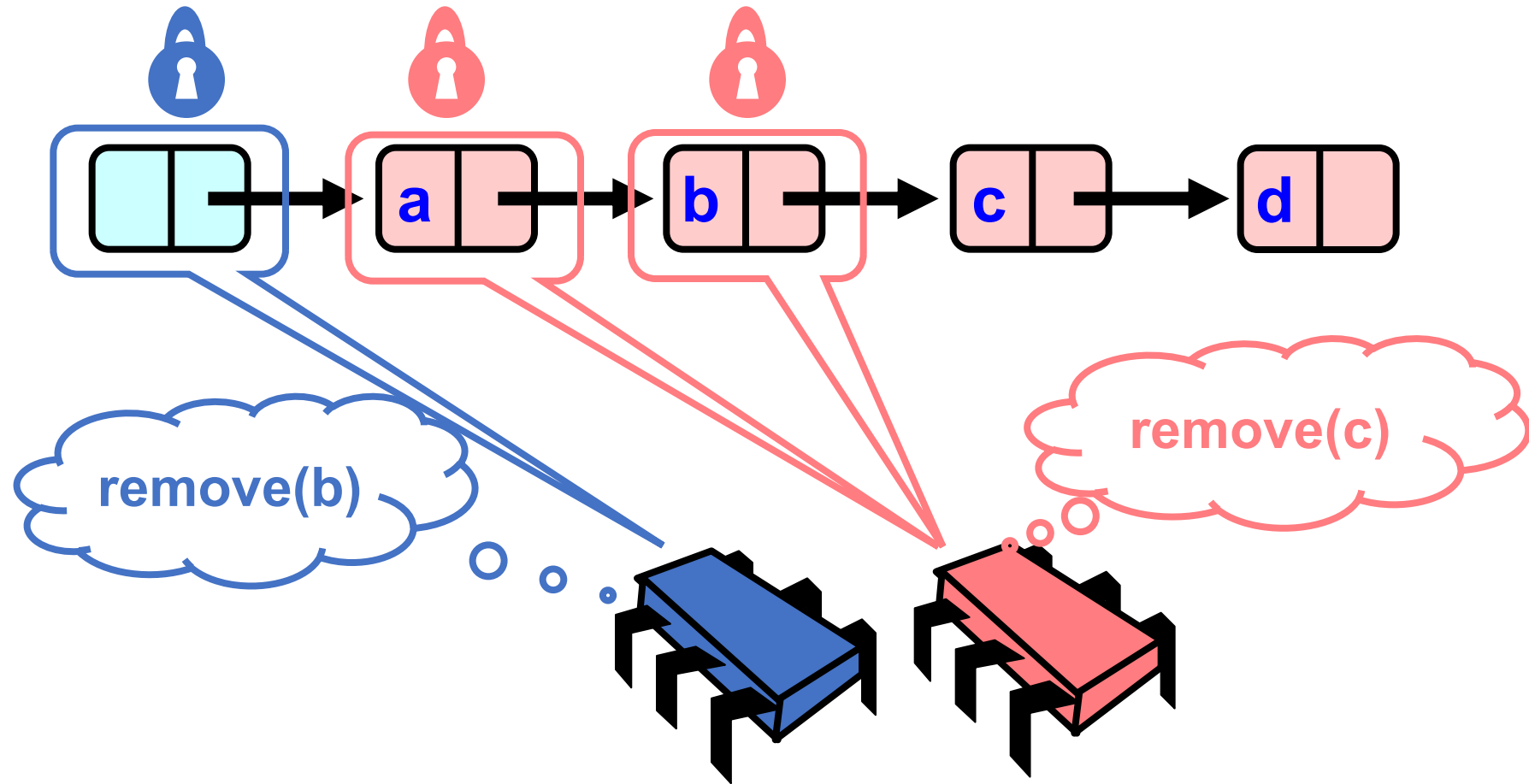
Removing a Node



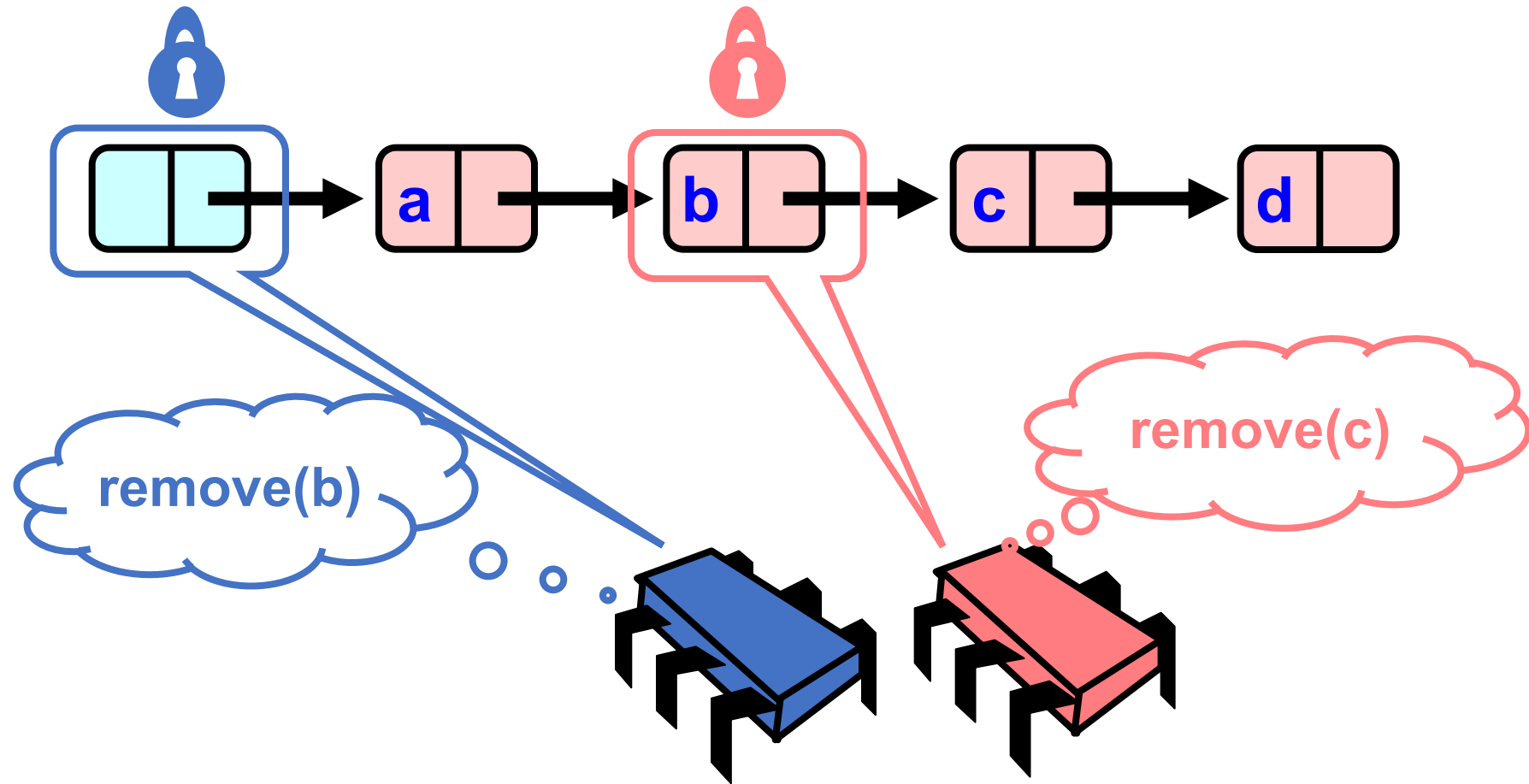
Removing a Node



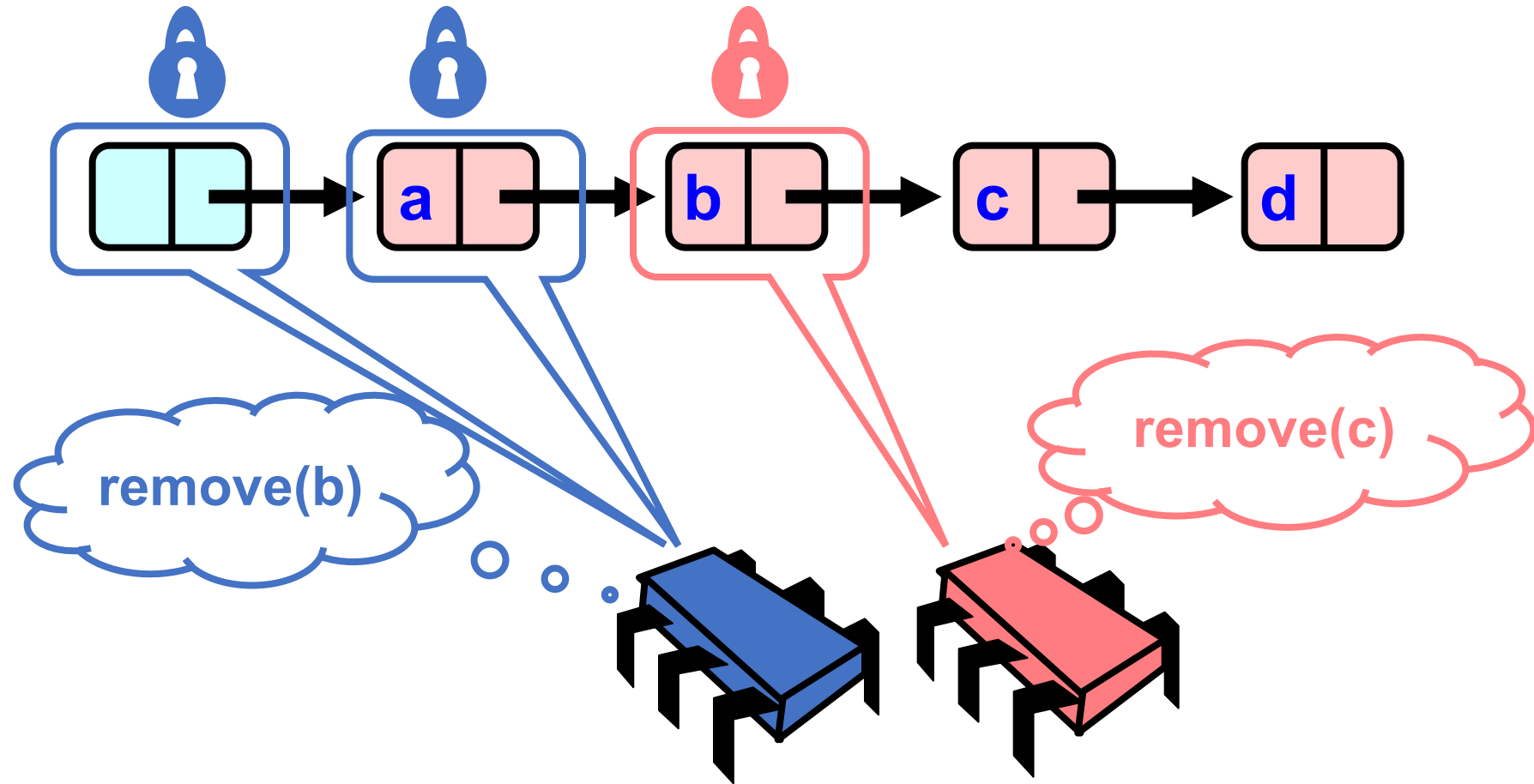
Removing a Node



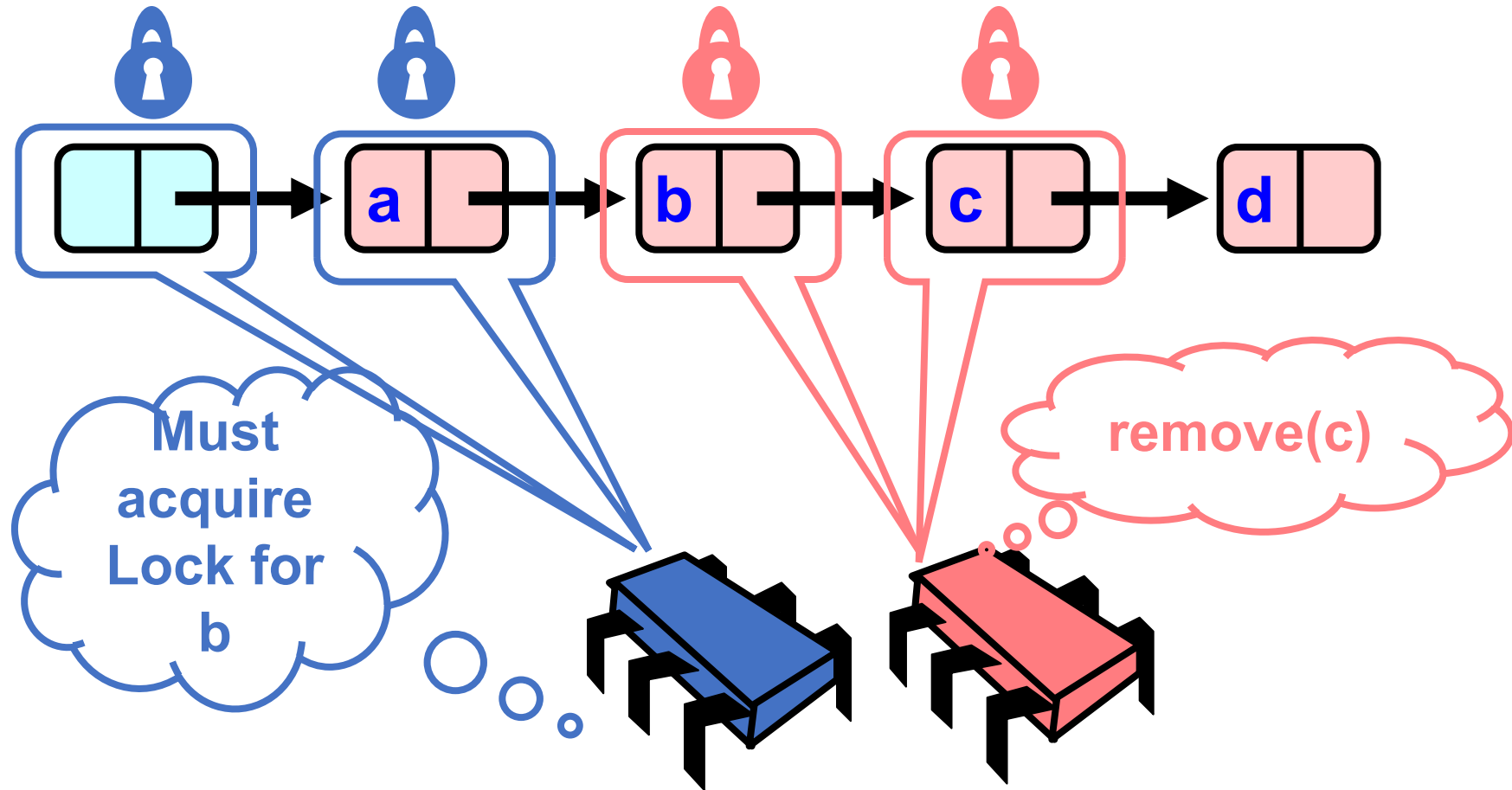
Removing a Node



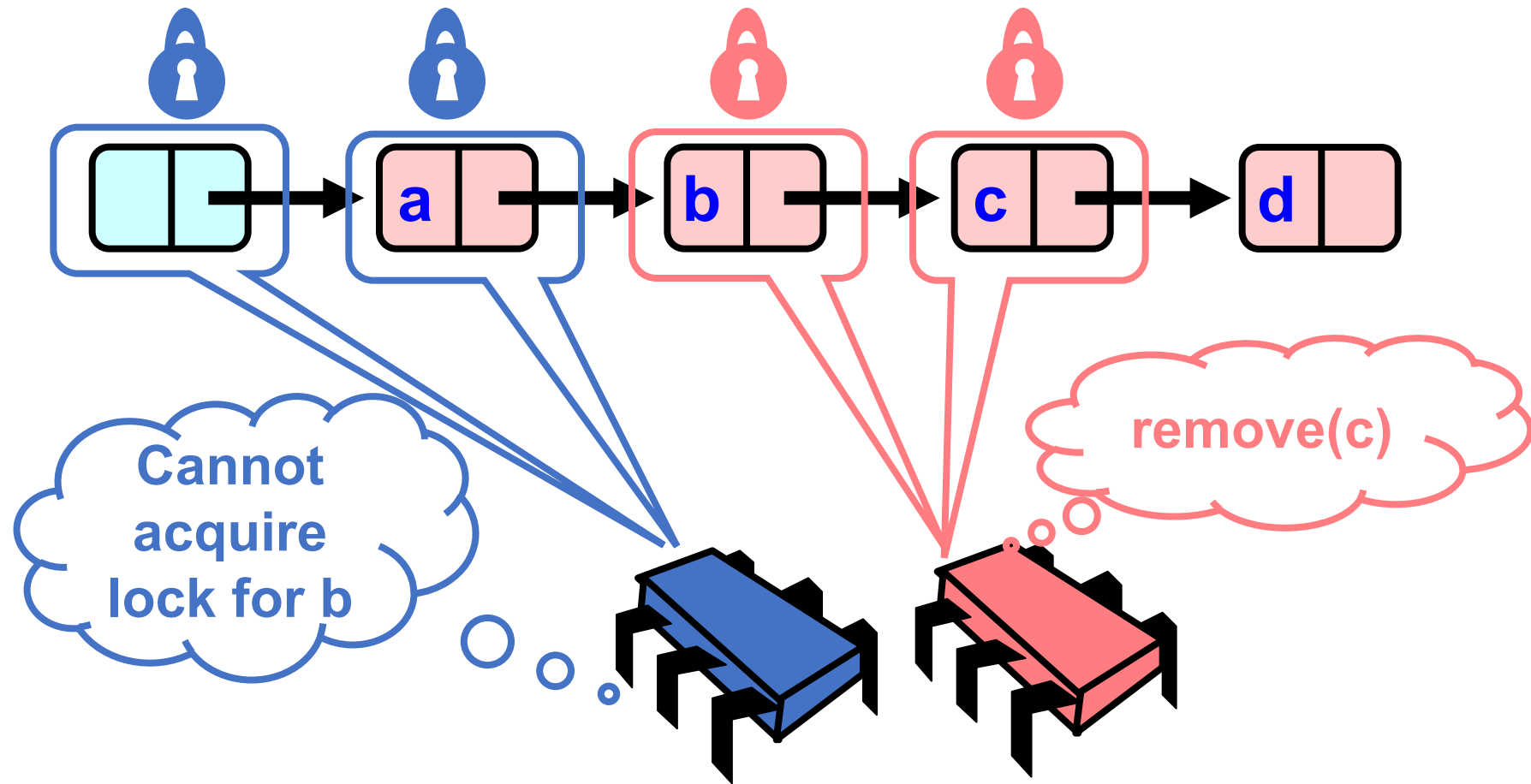
Removing a Node



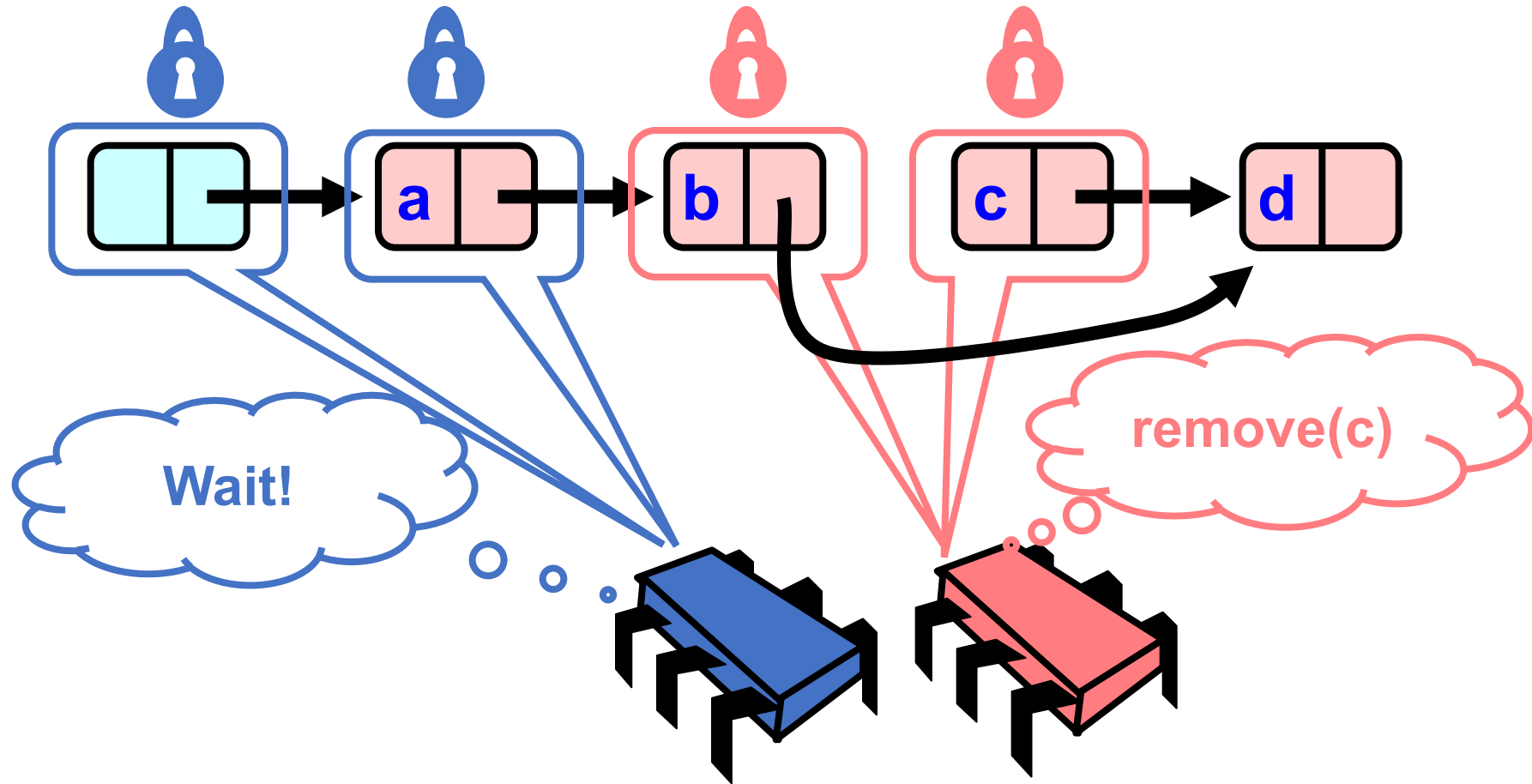
Removing a Node



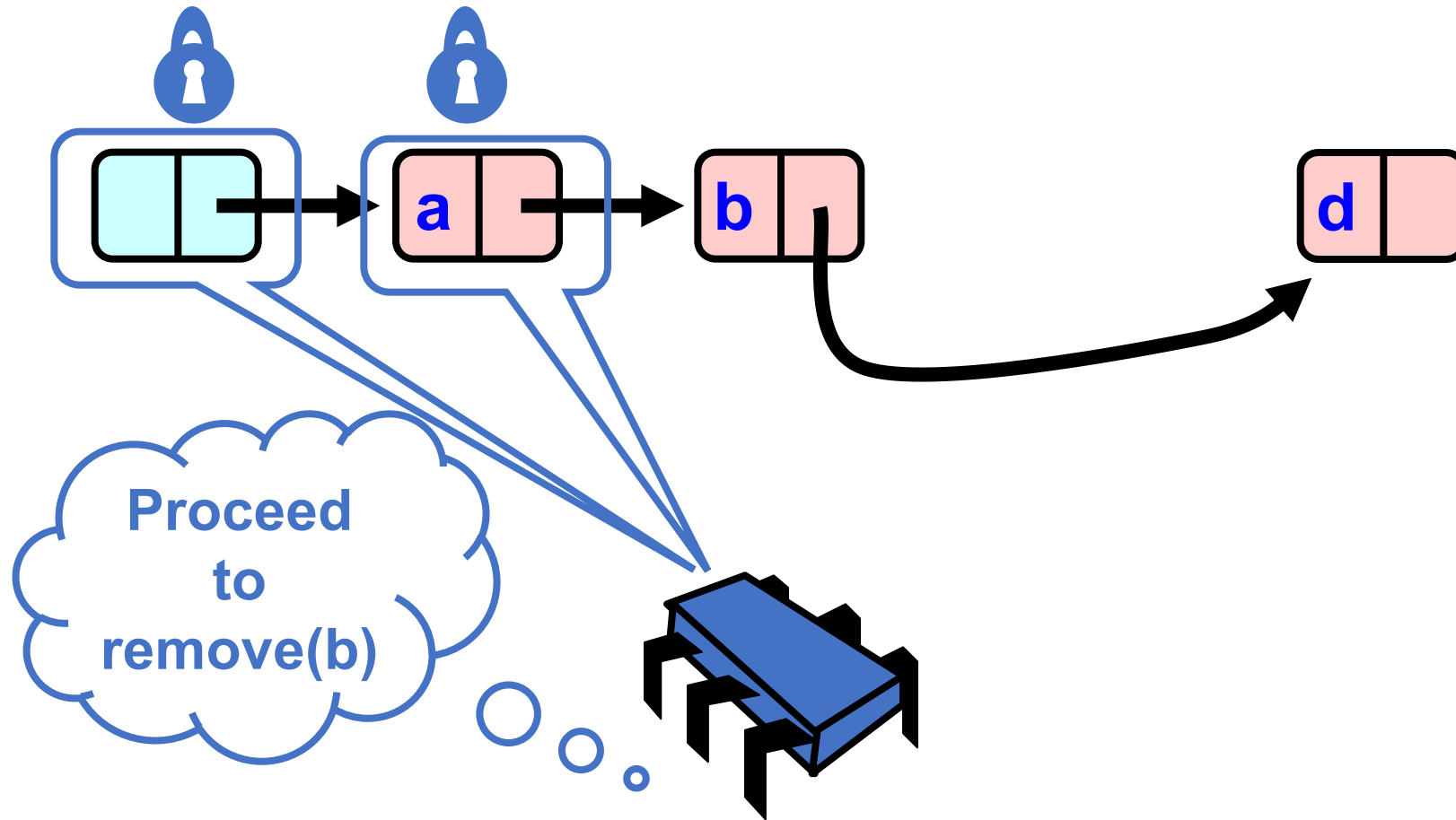
Removing a Node



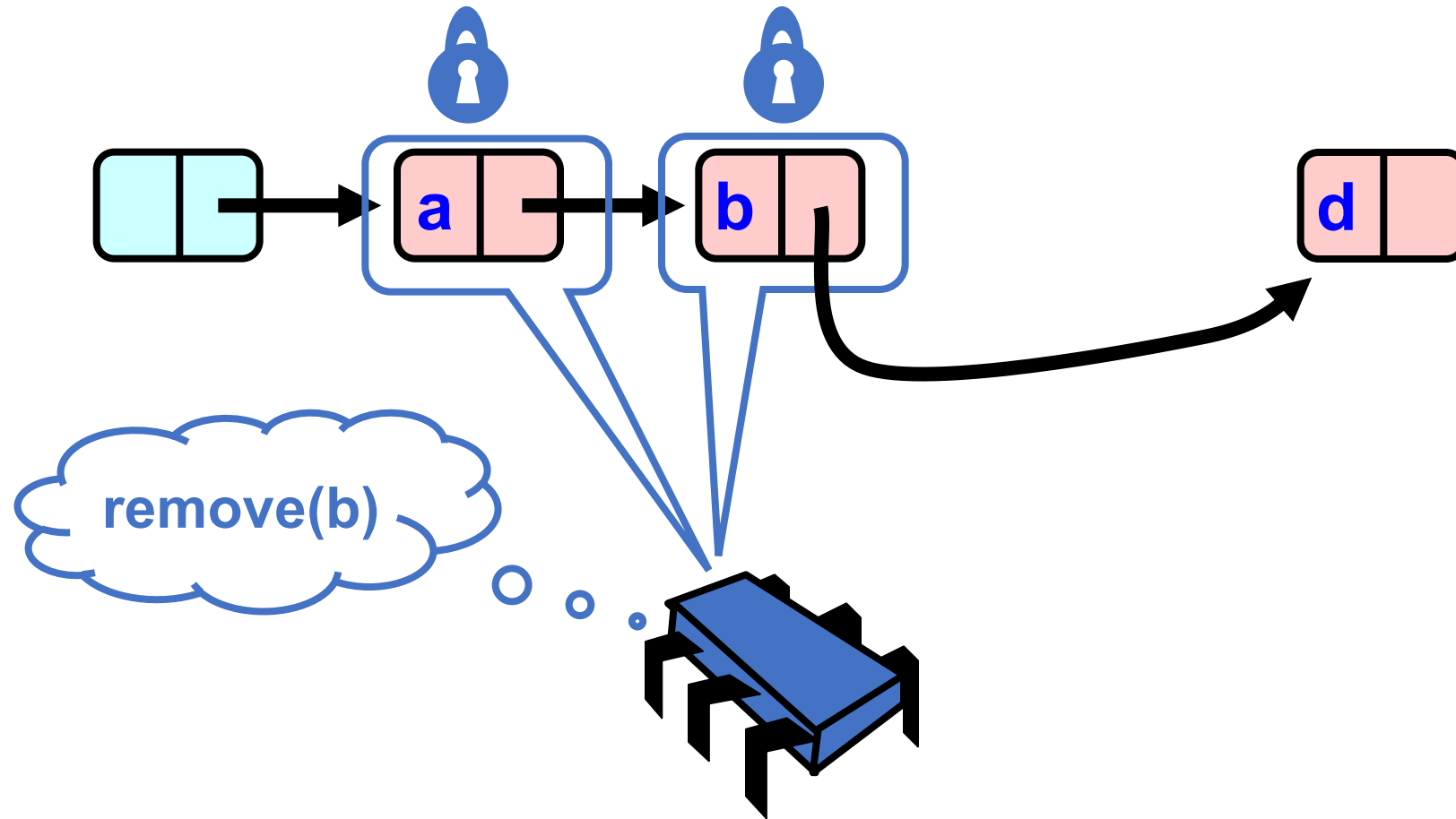
Removing a Node



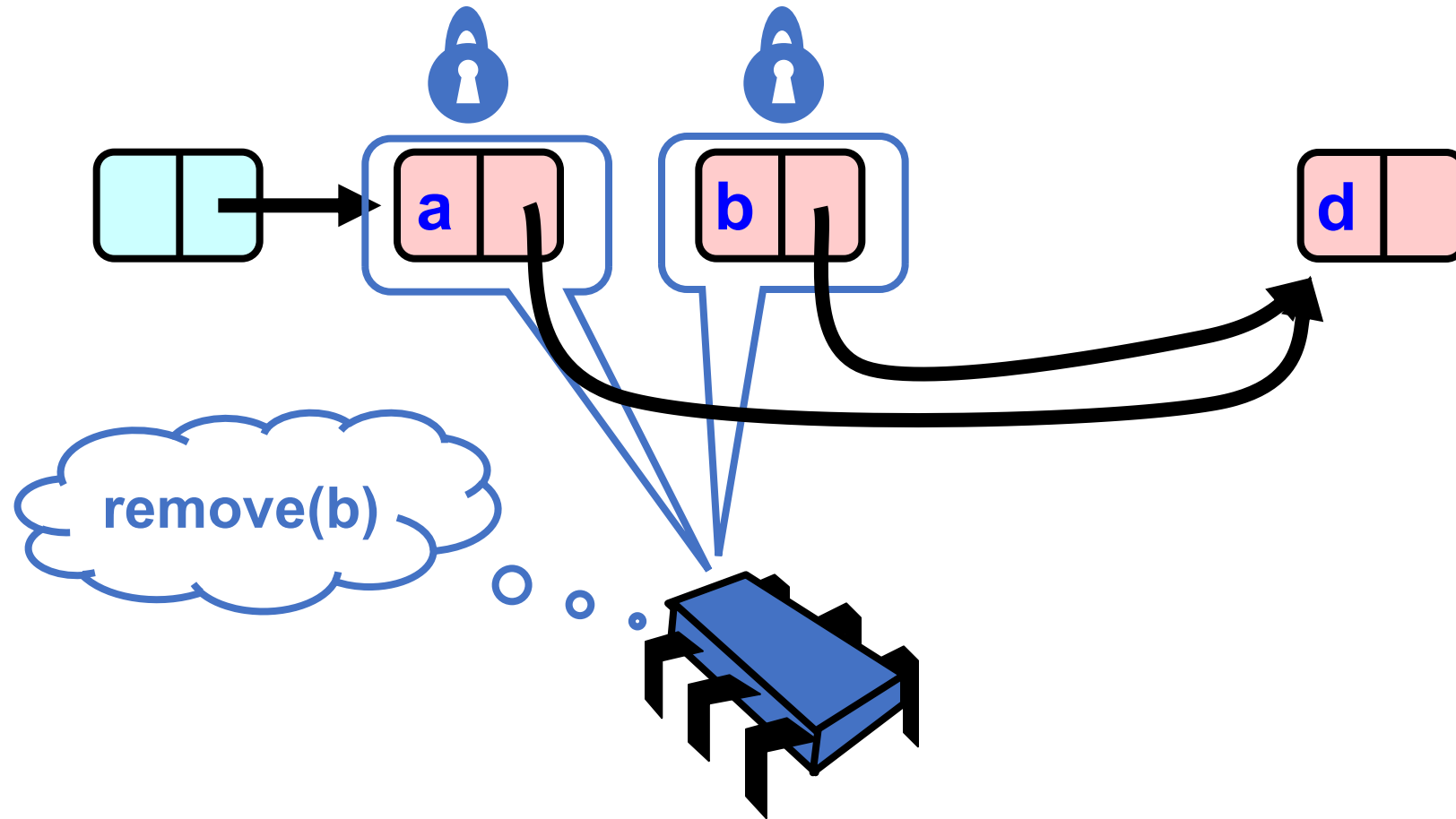
Removing a Node



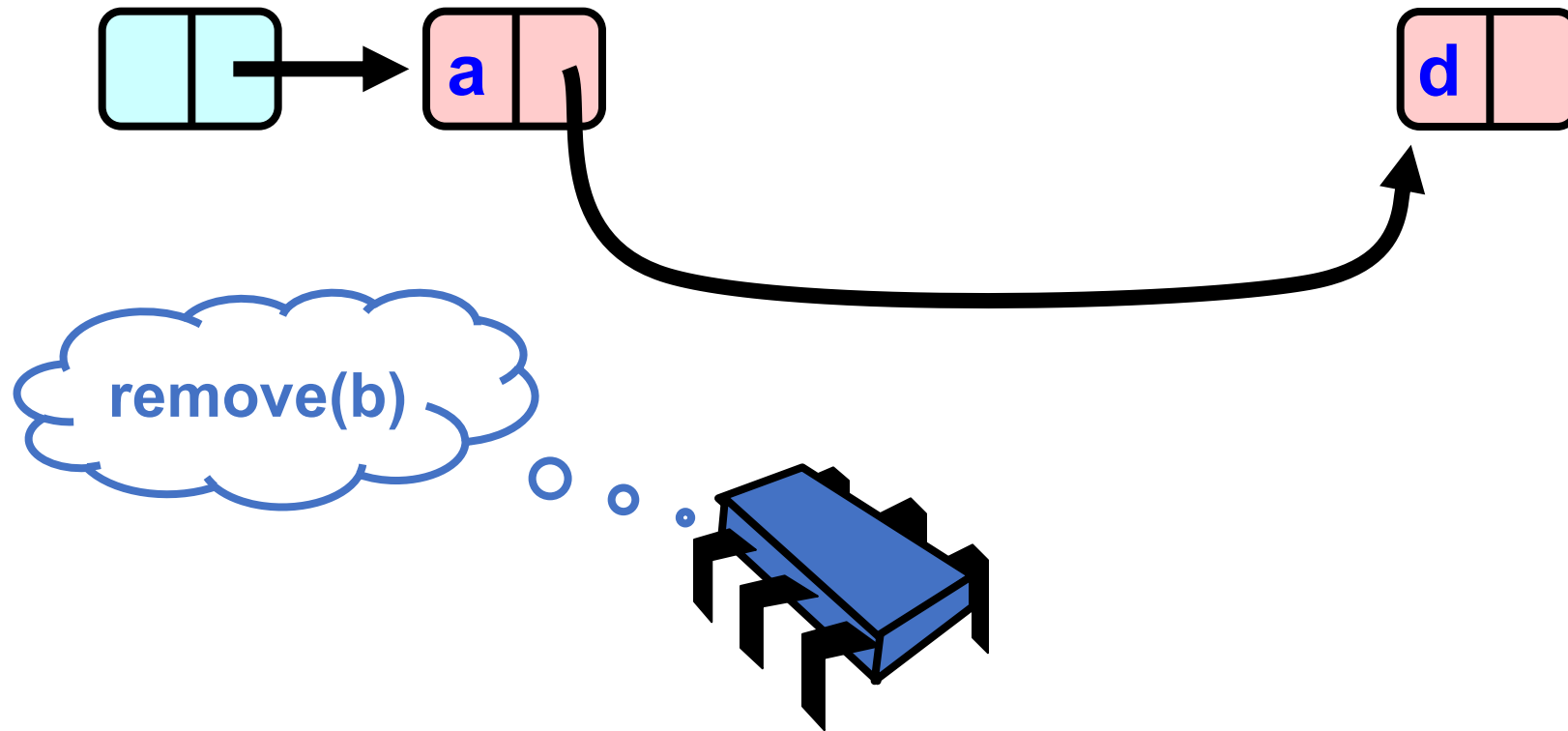
Removing a Node



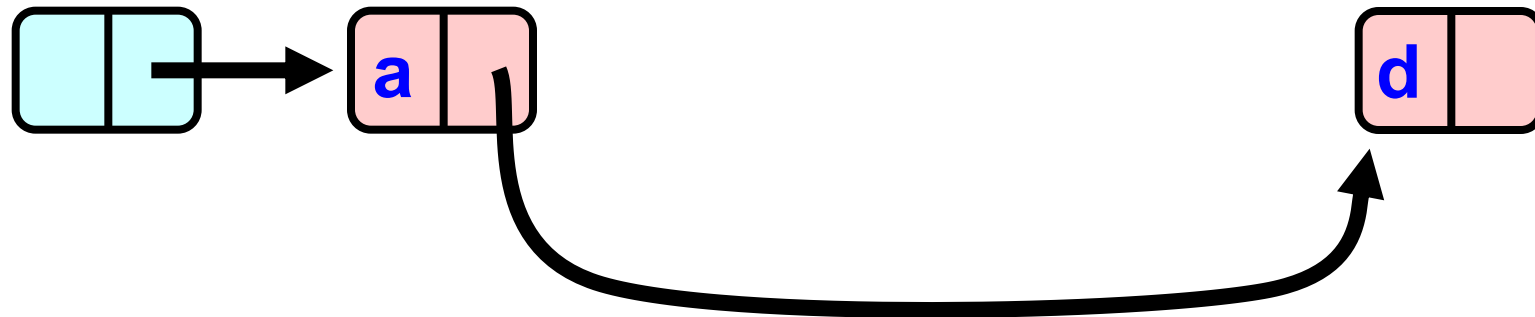
Removing a Node



Removing a Node



Removing a Node



Adding Nodes

- To add node e
 - Must lock predecessor
 - Must lock successor
- Neither can be deleted