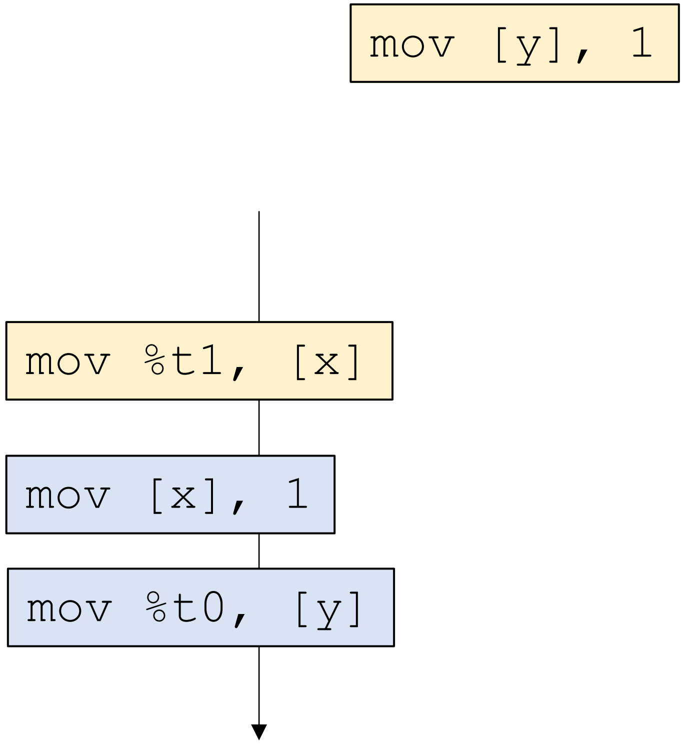
CSE113: Parallel Programming

- **Topics**:
  - Finish up weak memory models



mov [y], 1

mov %t1, [x]

mov [x], 1

mov %t0, [y]

Announcements

- HW 3 was graded last week.

- Today is the last day to turn in HW 4 (and 3 more days)
  - Hopefully you had fun with a taste of GPU programming

- HW 5 will be released this week
  - Last day to turn it in is Dec 3
  - It could be useful to work on for the final

Announcements

Final

- Allowed 3 pages of notes, front and back
- Similar to the midterm
- Time: Dec 12, 12-3pm

Announcements

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

Previous quiz + Review

Memory models

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?`

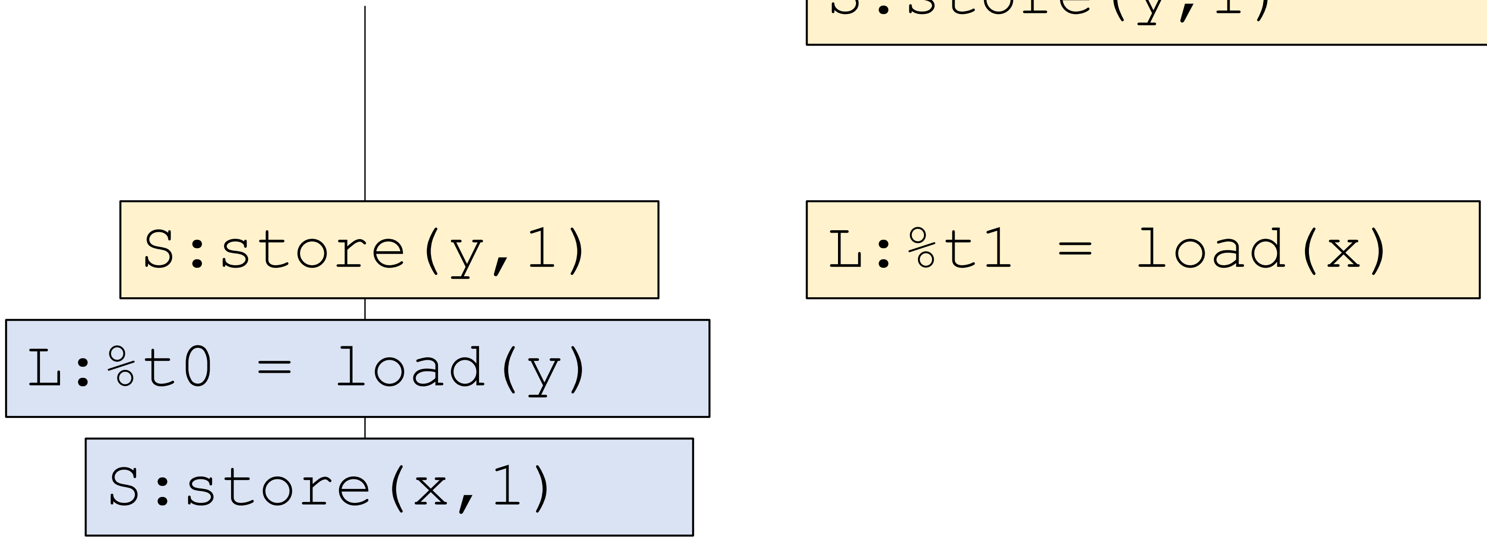
Get out our lego bricks and try for sequential consistency

Thread 0:

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

Thread 1:

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



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

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

Global variable:

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

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

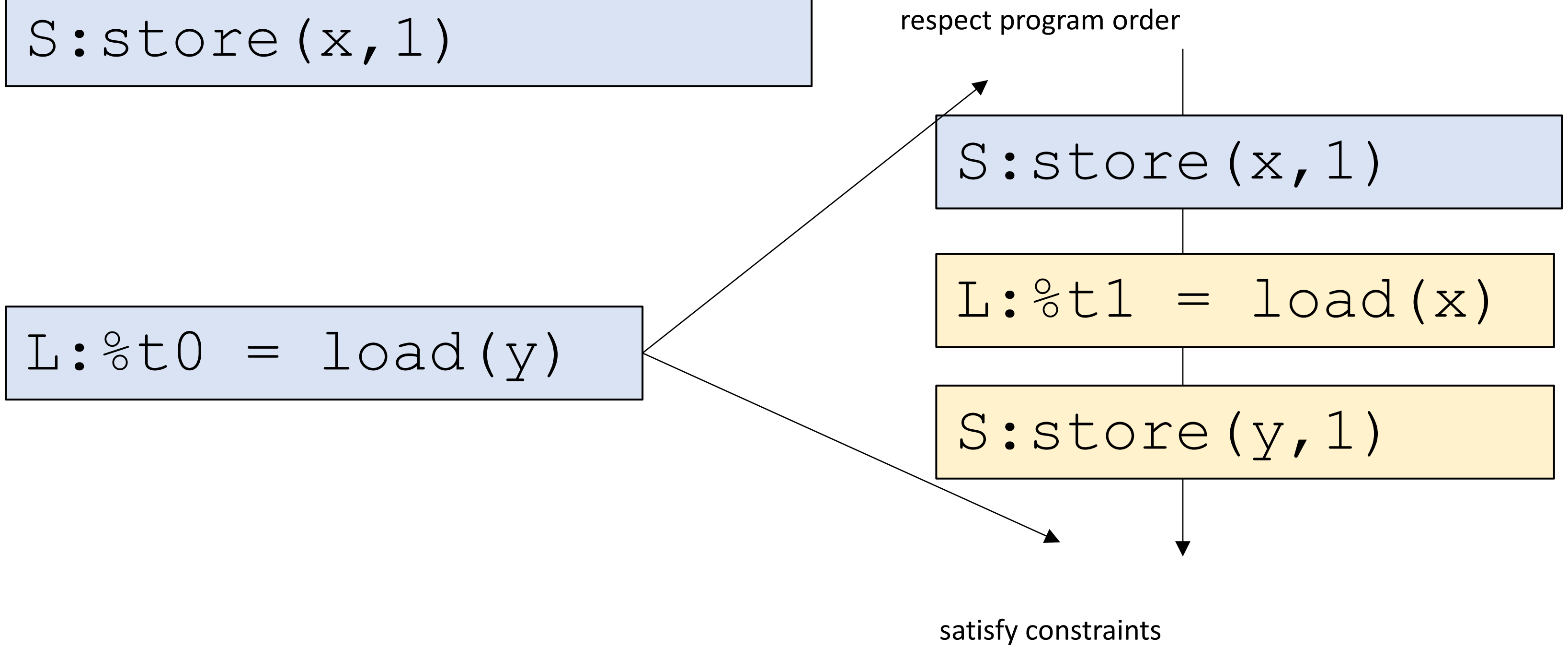
Get out our lego bricks

Thread 0:

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

Thread 1:

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



respect program order

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

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

satisfy constraints

Not allowed under sequential consistency!

Global variable:

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

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

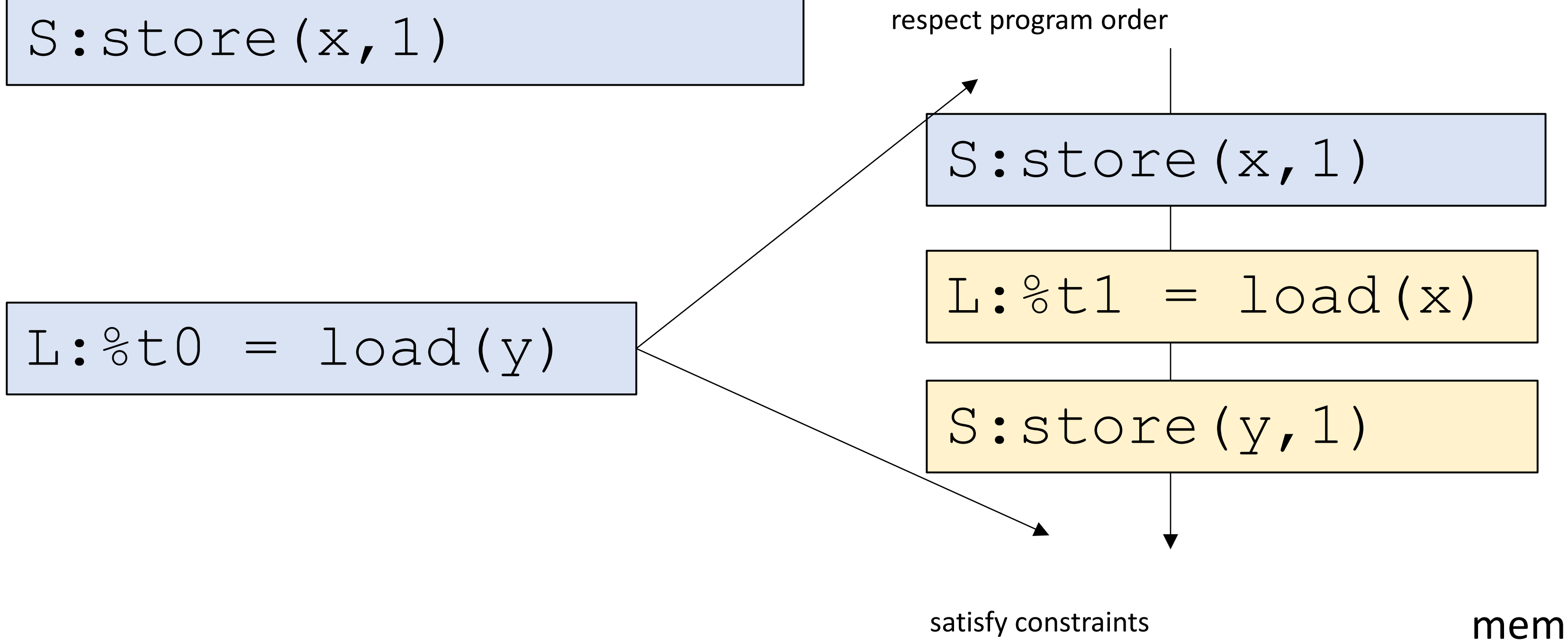
Get out our lego bricks

Thread 0:

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

Thread 1:

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



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

What about TSO?

Global variable:

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

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

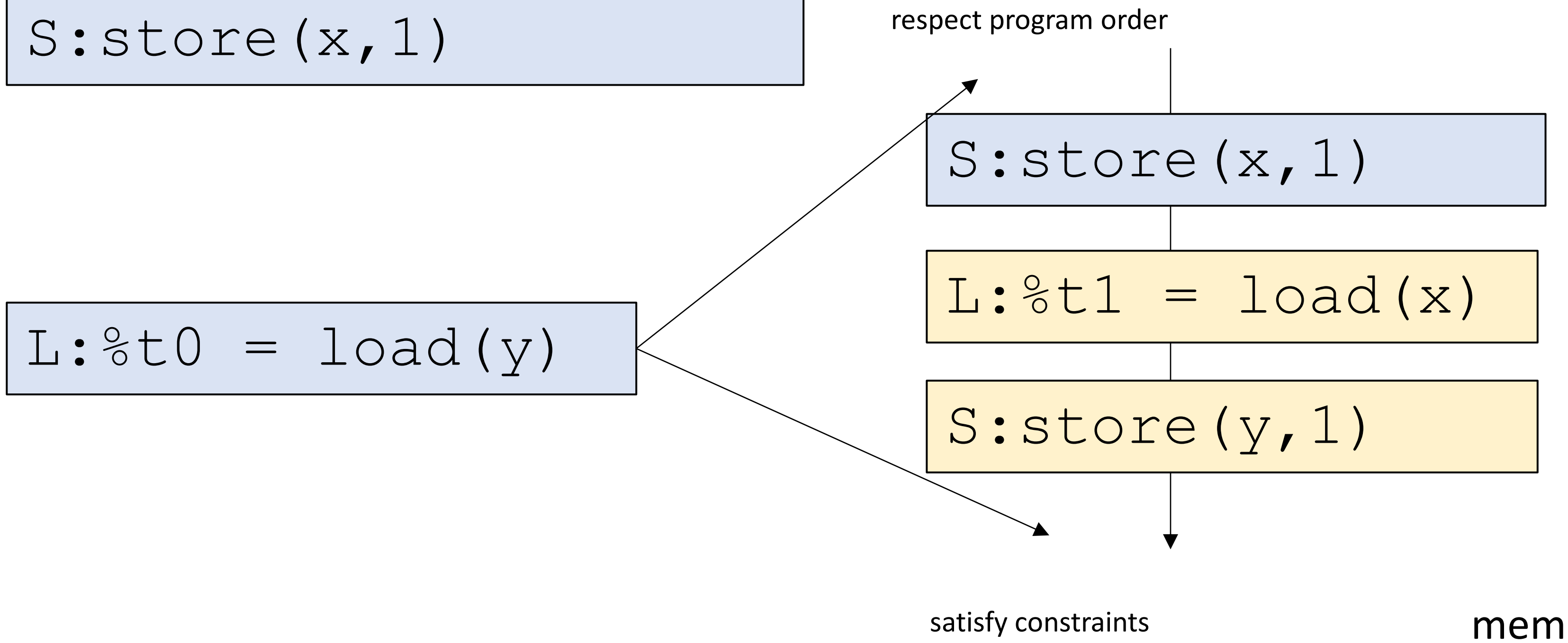
Get out our lego bricks

Thread 0:

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

Thread 1:

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



respect program order

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

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

satisfy constraints

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

What about TSO? NOT ALLOWED!

Global variable:

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

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

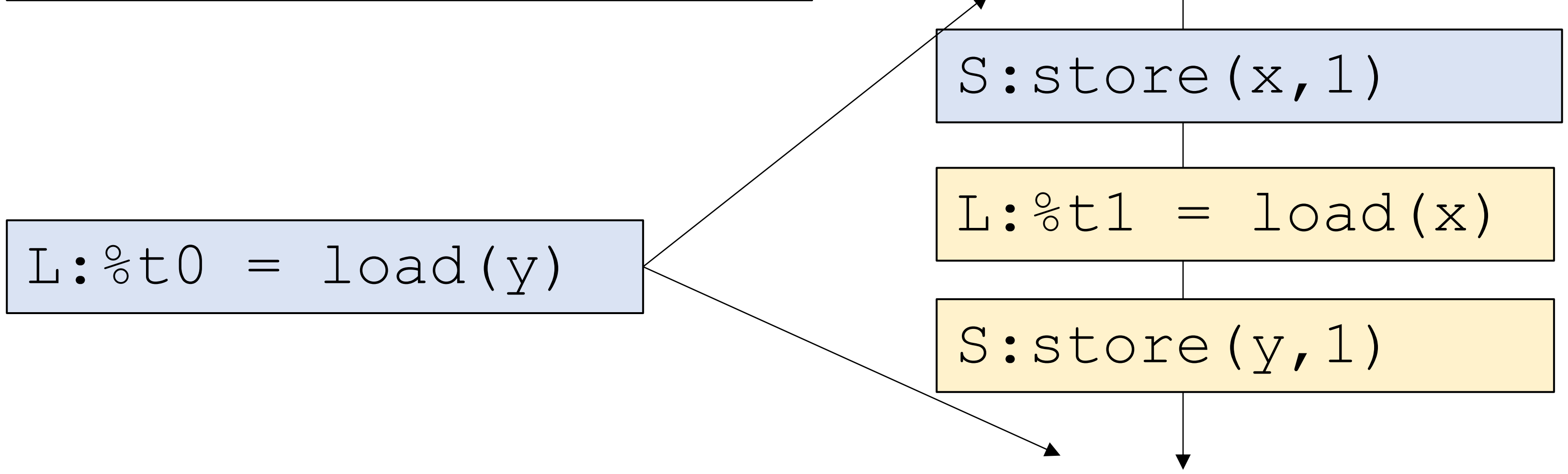
Get out our lego bricks

Thread 0:

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

Thread 1:

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



respect program order

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

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

satisfy constraints

memory access 0

memory access 1

	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 == t1 == 1?`

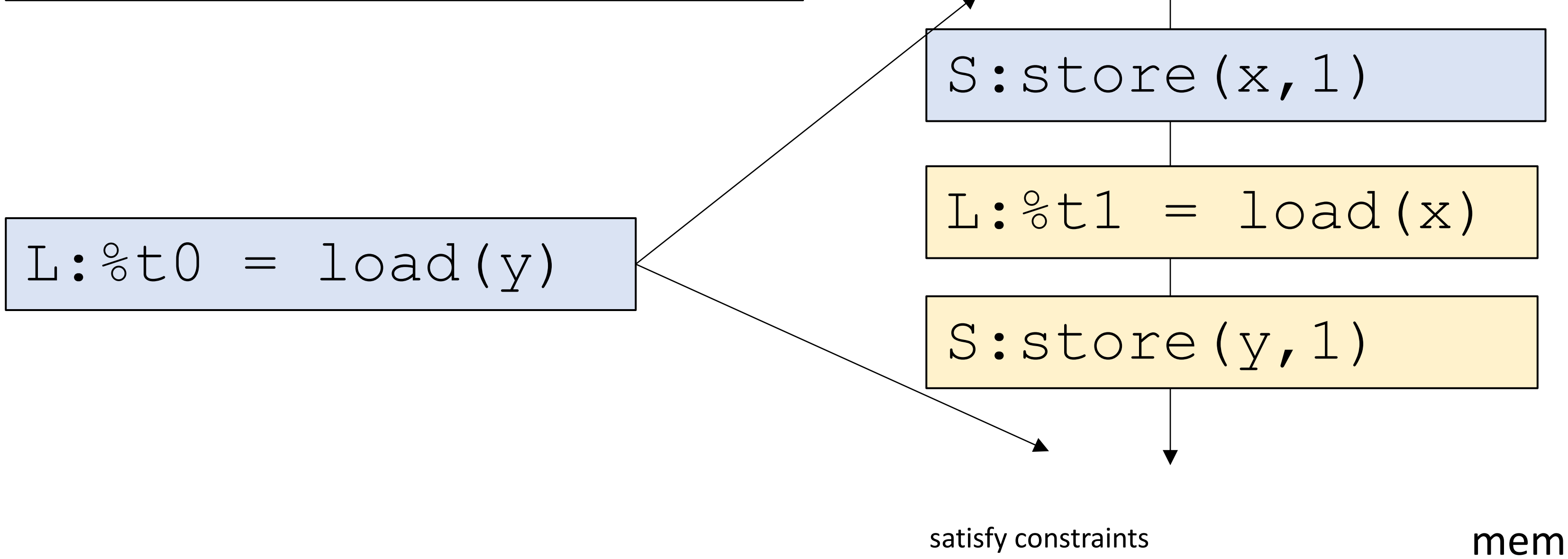
Get out our lego bricks

Thread 0:

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

Thread 1:

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



respect program order

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

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

satisfy constraints

memory access 0

memory access 1

	L	S
L	NO	Different address
S	NO	Different address

What about PSO? NO!

Global variable:

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

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

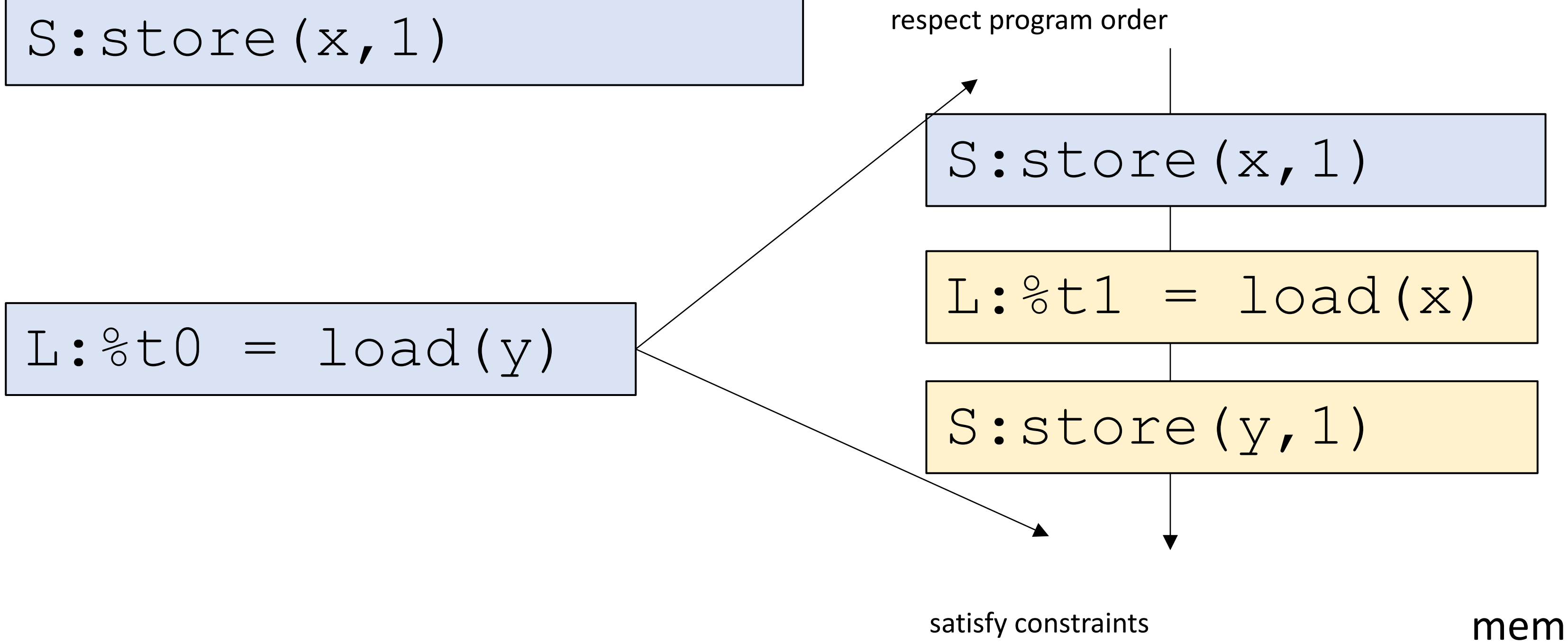
Get out our lego bricks

Thread 0:

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

Thread 1:

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



respect program order

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

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

satisfy constraints

		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 == t1 == 1?`

Get out our lego bricks

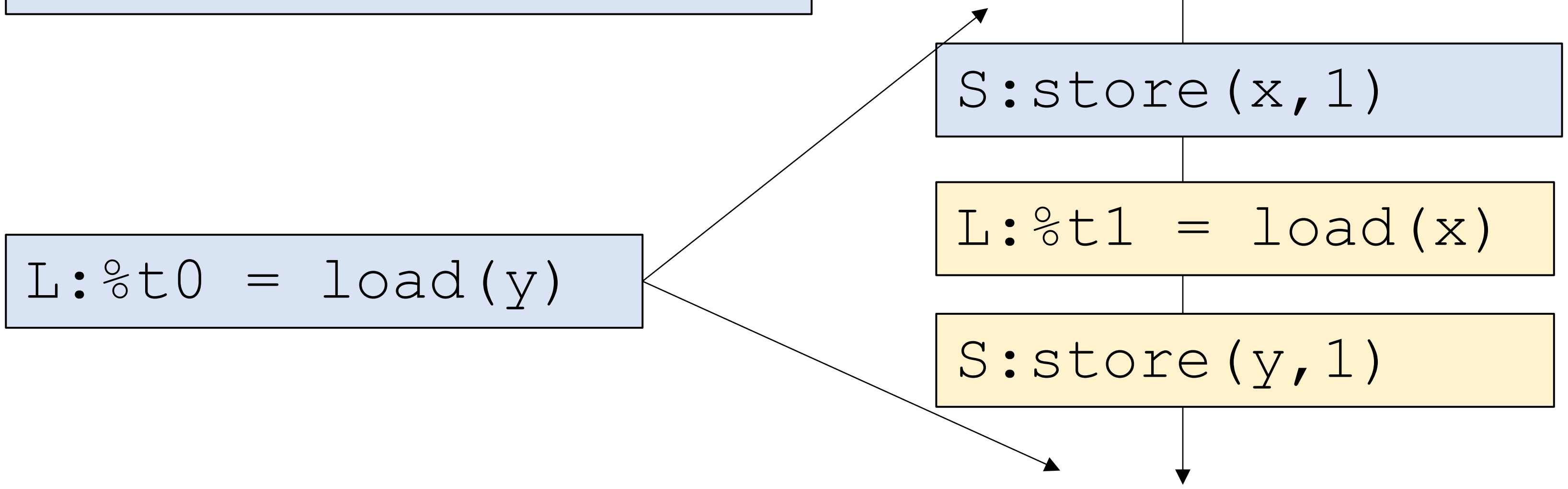
Thread 0:

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

Thread 1:

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

respect program order



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

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

satisfy constraints

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 == t1 == 1?`

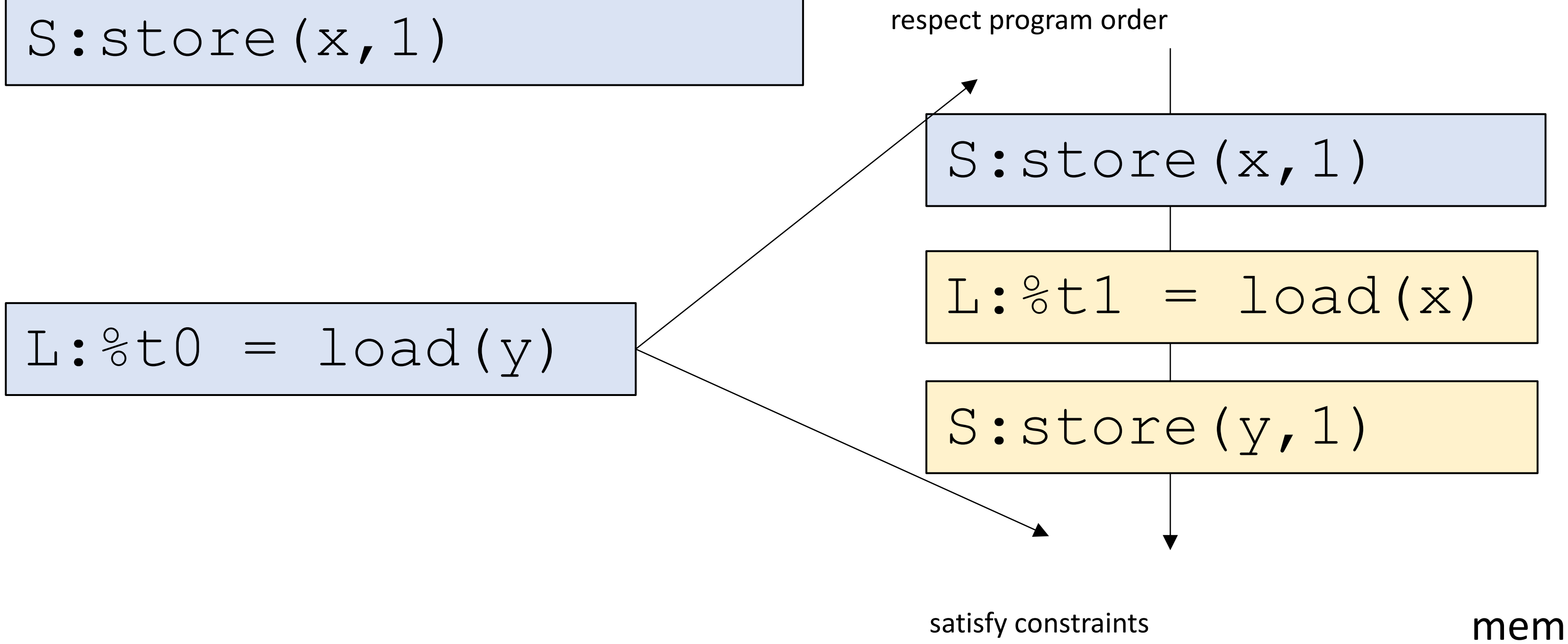
Get out our lego bricks

Thread 0:

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

Thread 1:

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



respect program order

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

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

satisfy constraints

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

What about RMO? YES!

Global variable:

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

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

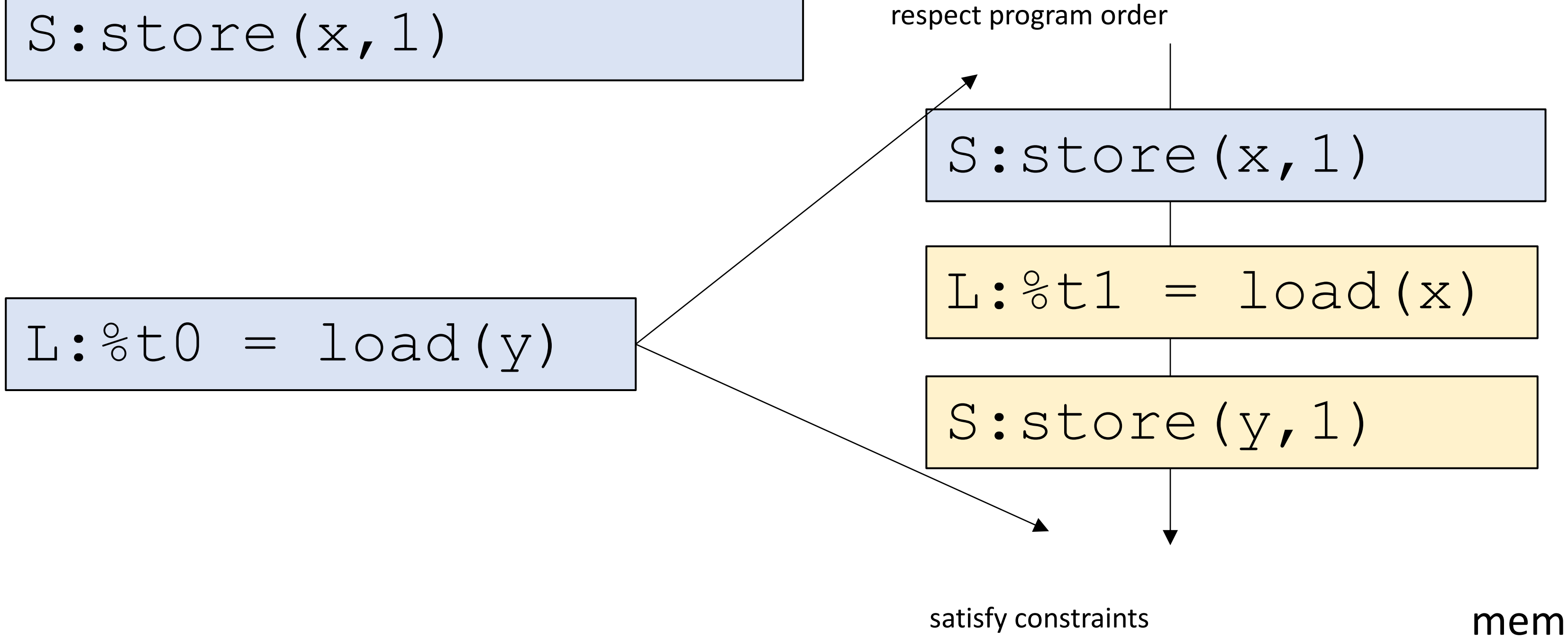
Get out our lego bricks

Thread 0:

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

Thread 1:

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



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

respect program order

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

satisfy constraints

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

How do we disallow the behavior in RMO?

Global variable:

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

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

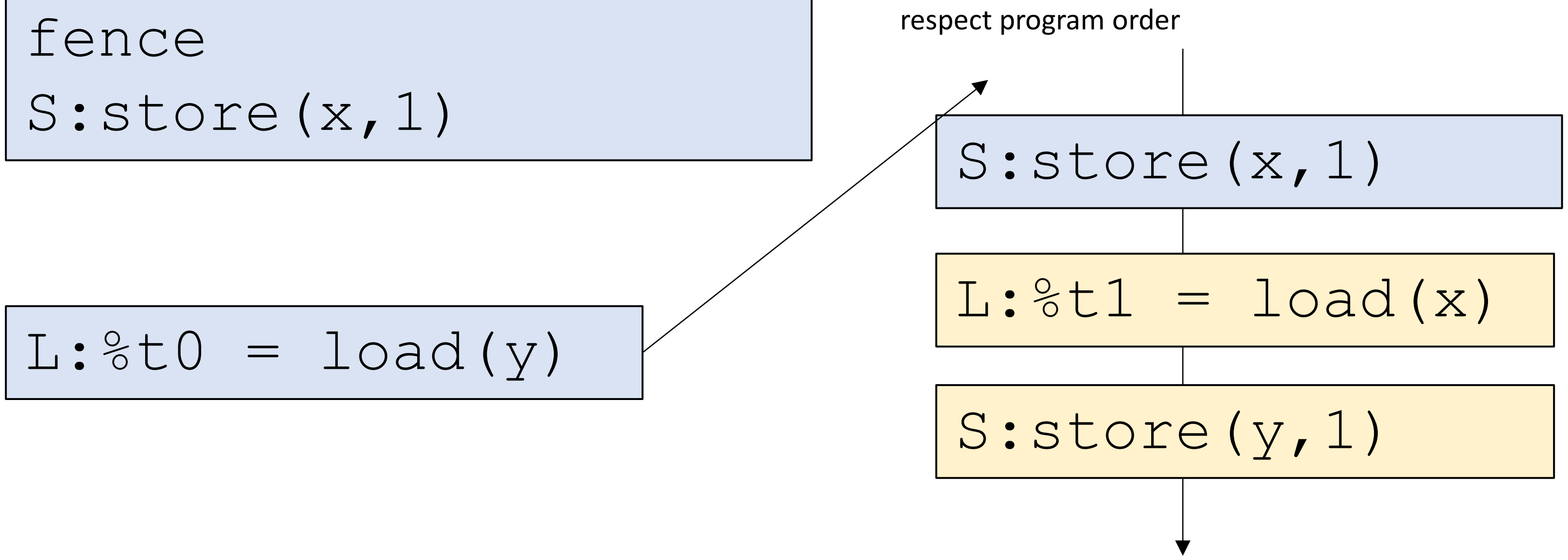
Get out our lego bricks

Thread 0:

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

Thread 1:

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



respect program order

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

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

memory access 0

memory access 1

	L	S
L	YES	Different address
S	🟩 different address	Different address

How do we disallow the behavior in RMO?

Global variable:

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

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

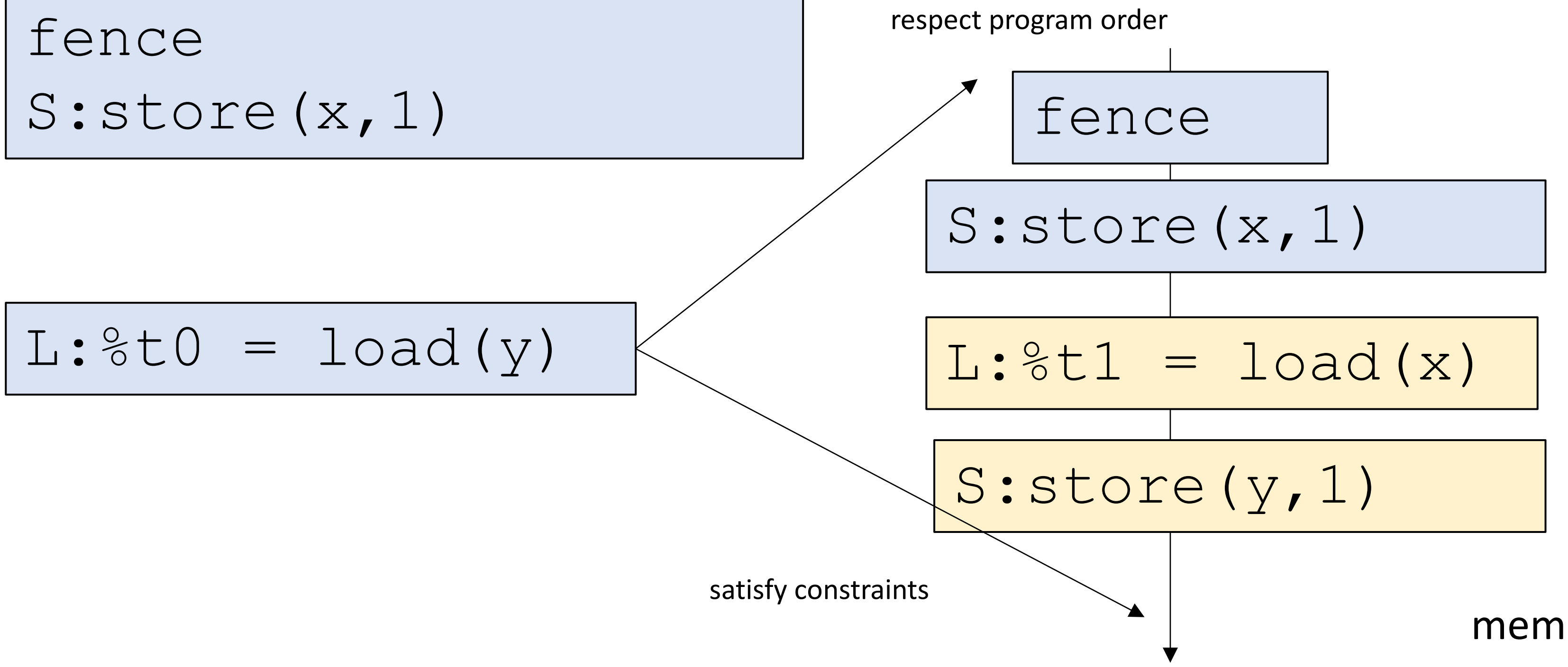
Get out our lego bricks

Thread 0:

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

Thread 1:

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



respect program order

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

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

satisfy constraints

How do we disallow the behavior in RMO?

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

Global variable:

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

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

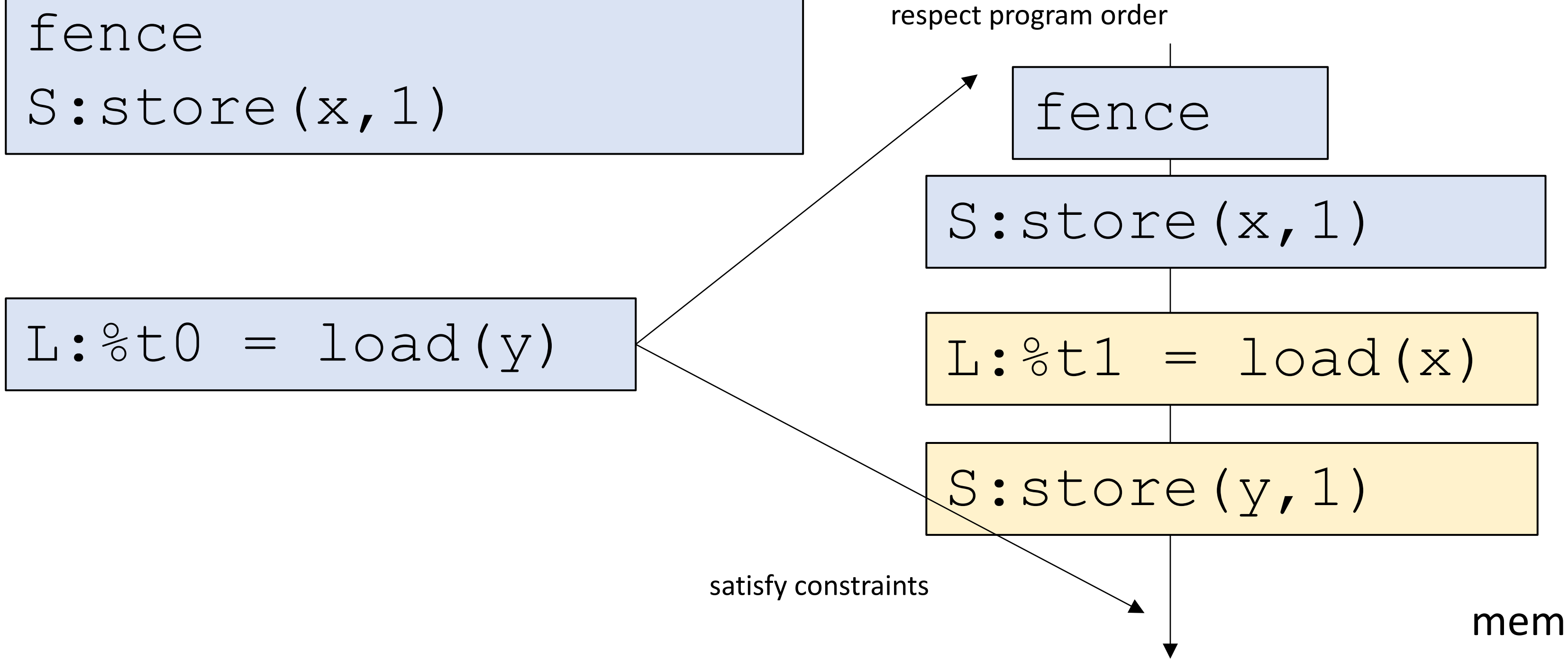
Get out our lego bricks

Thread 0:

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

Thread 1:

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



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

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

		memory access 0	
		L	S
memory access 1	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?`

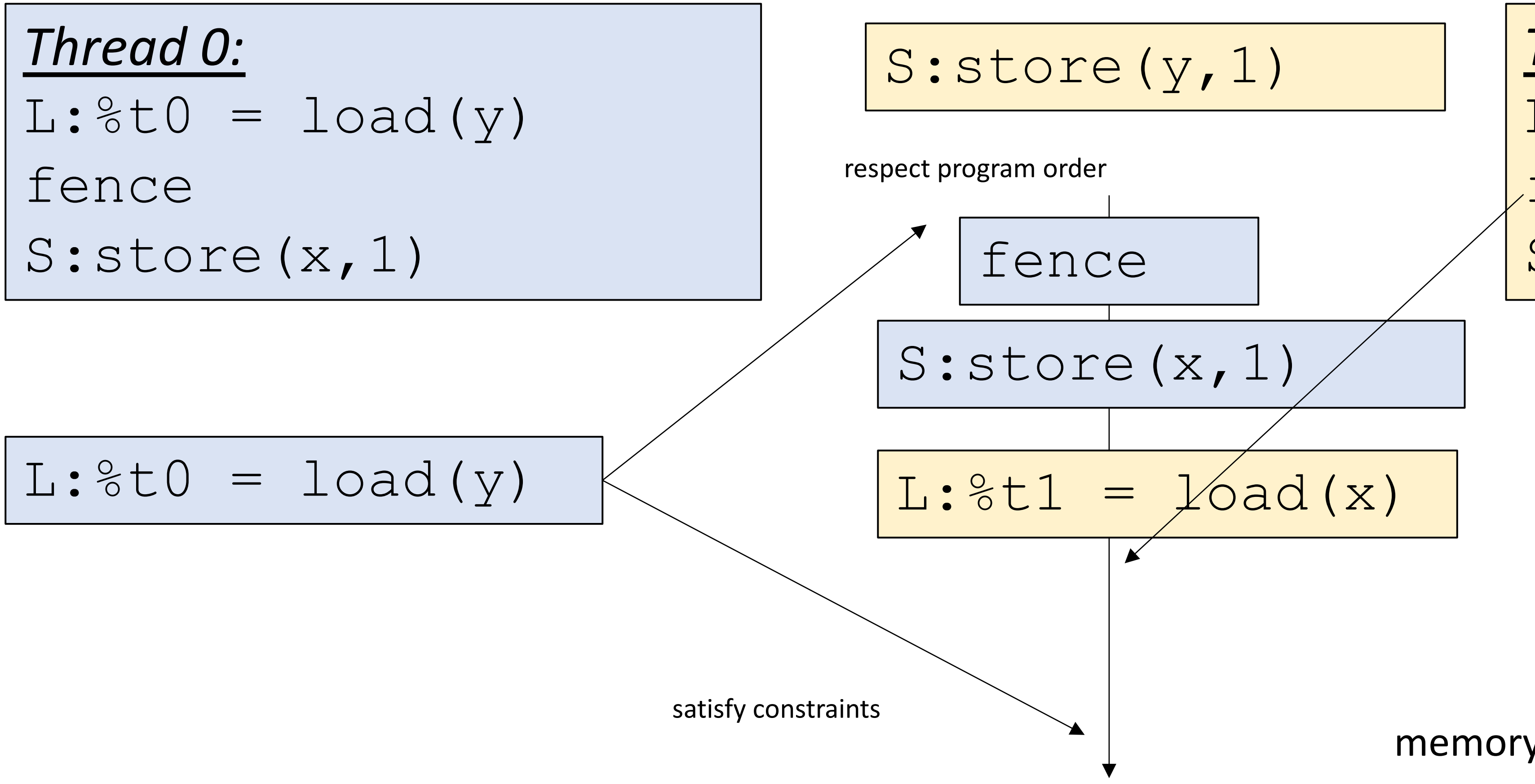
Get out our lego bricks

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?

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

Global variable:

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

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

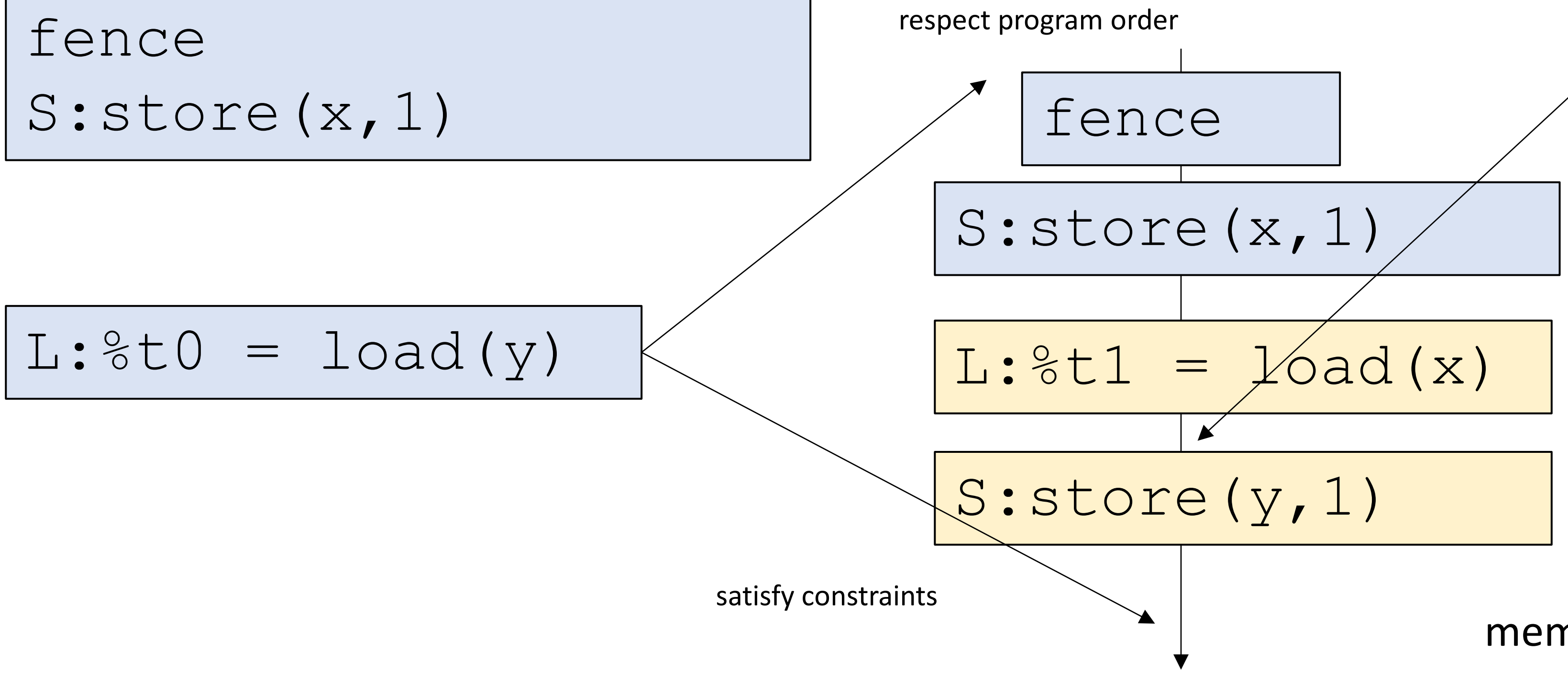
Get out our lego bricks

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

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

One more example

Global variable:

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

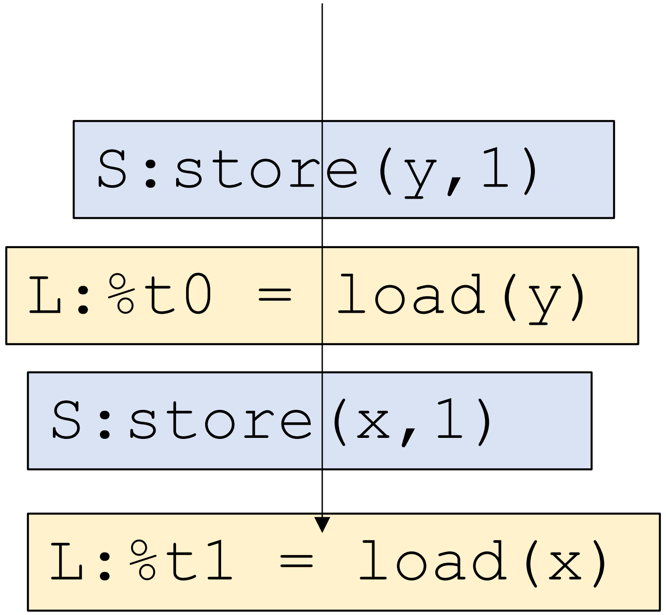
Question: can `t0 == 1 and t1 == 0`?

Thread 0:

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

Thread 1:

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



```
S:store(y,1)
L:%t0 = load(y)
S:store(x,1)
L:%t1 = load(x)
```

Global variable:

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

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

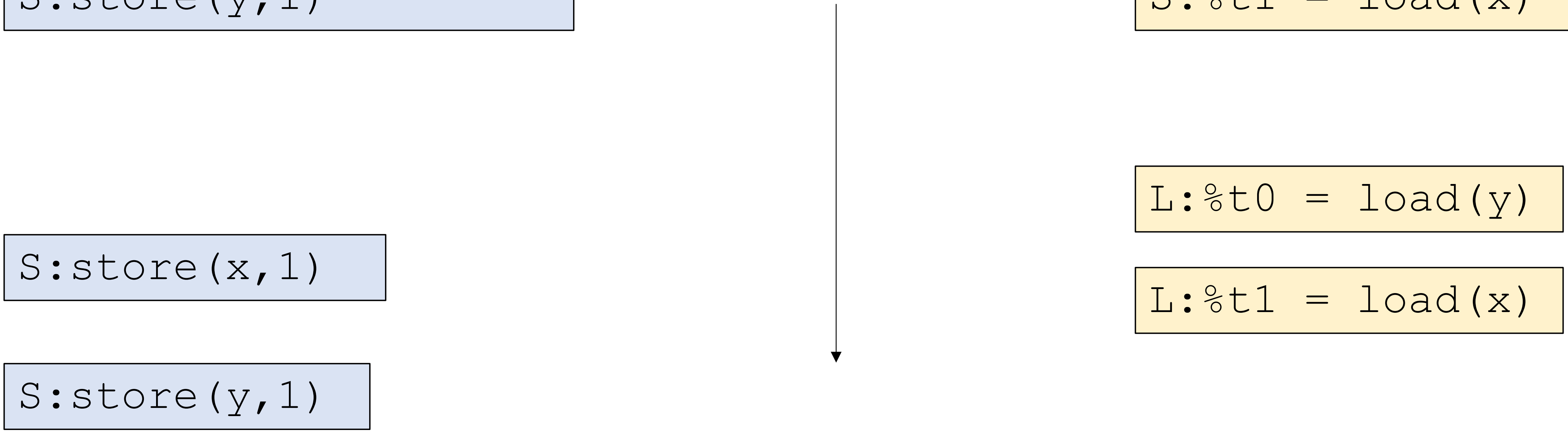
start off thinking about sequential consistency

Thread 0:

```
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)
```

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

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

Global variable:

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

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

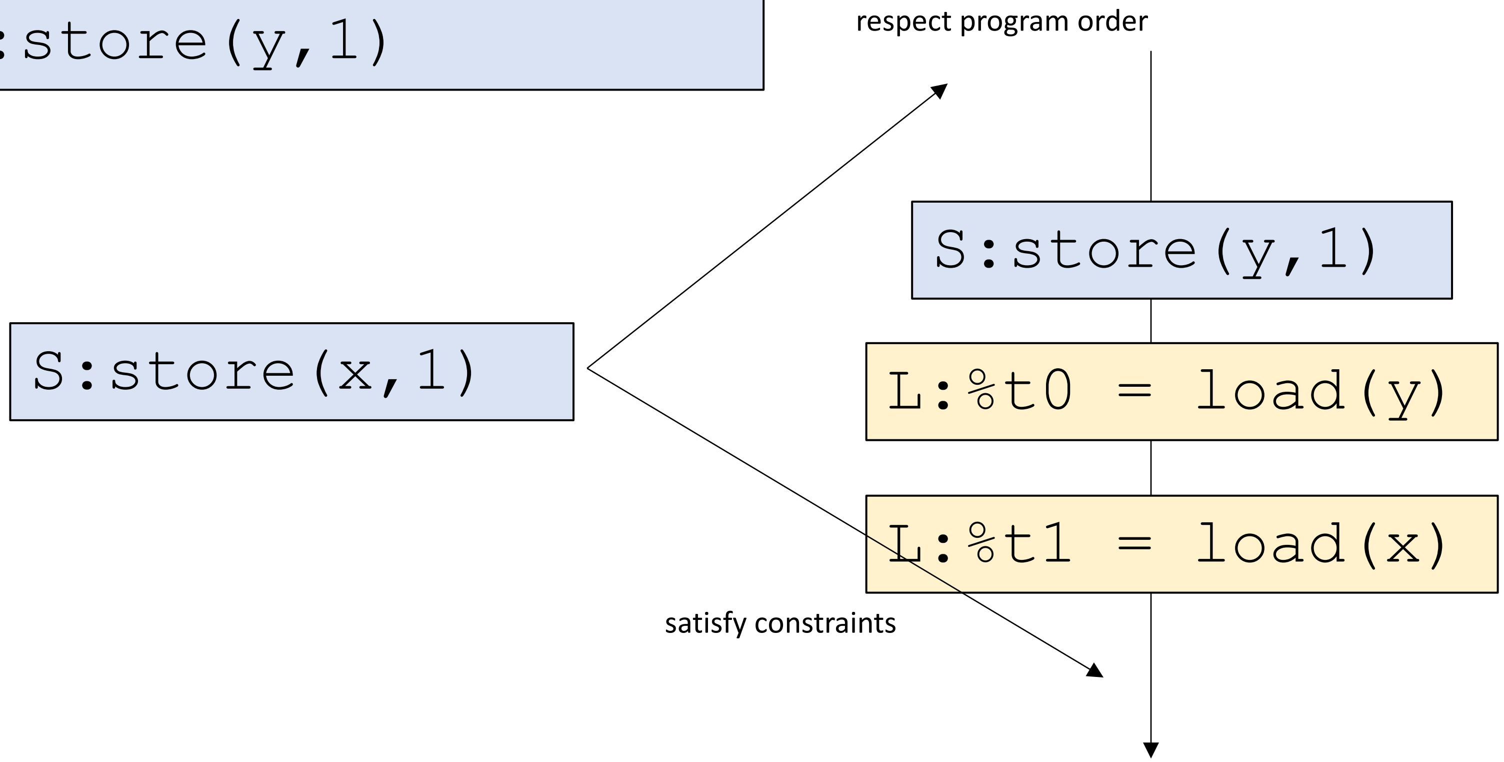
start off thinking about sequential consistency

Thread 0:

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

Thread 1:

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



respect program order

```
S:store(y,1)
L:%t0 = load(y)
L:%t1 = load(x)
```

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

satisfy constraints

Global variable:

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

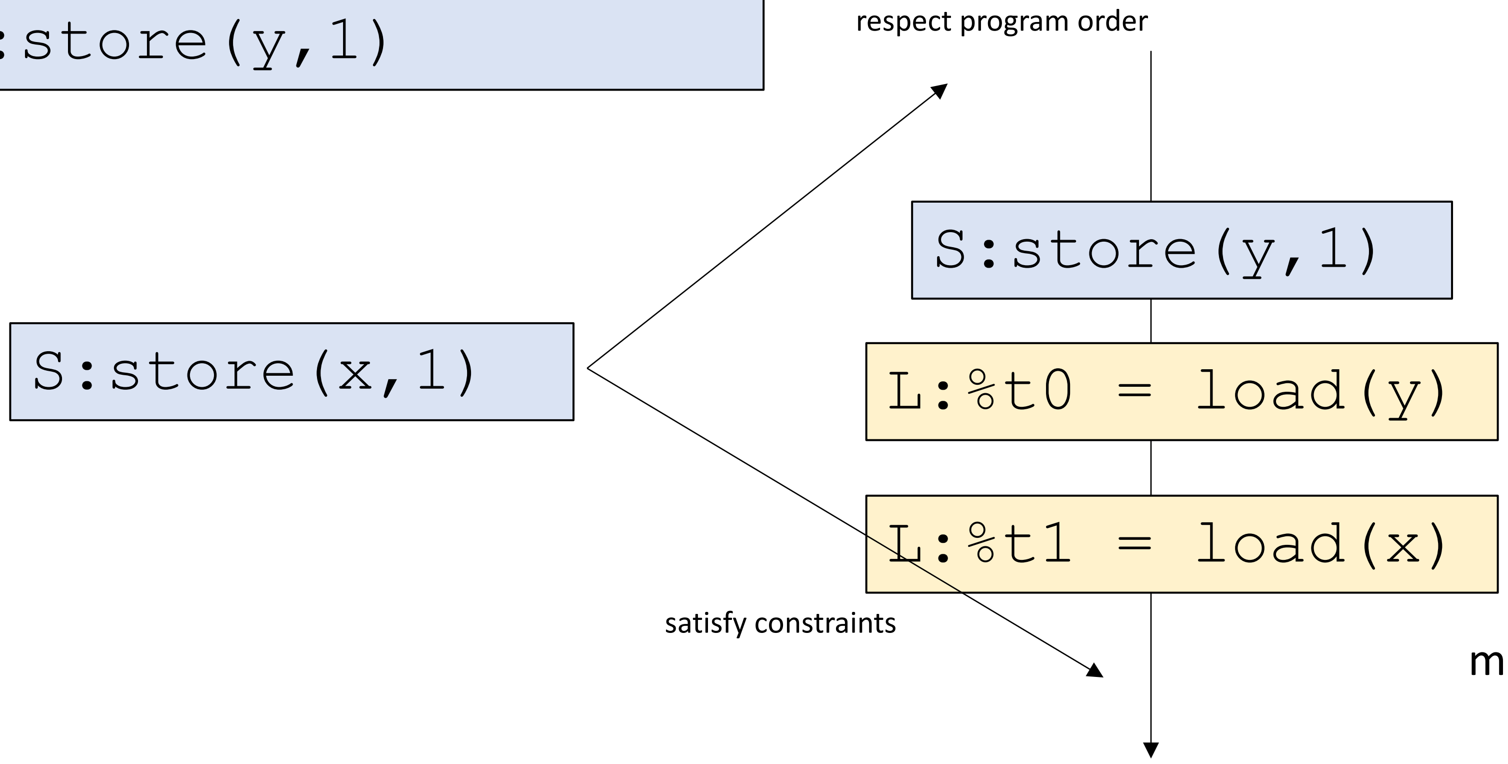
Question: can `t0 == 1 and t1 == 0`?

Thread 0:

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

Thread 1:

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



respect program order

```
S:store(y,1)
L:%t0 = load(y)
L:%t1 = load(x)
```

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

satisfy constraints

memory access 0

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

What about TSO?

Global variable:

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

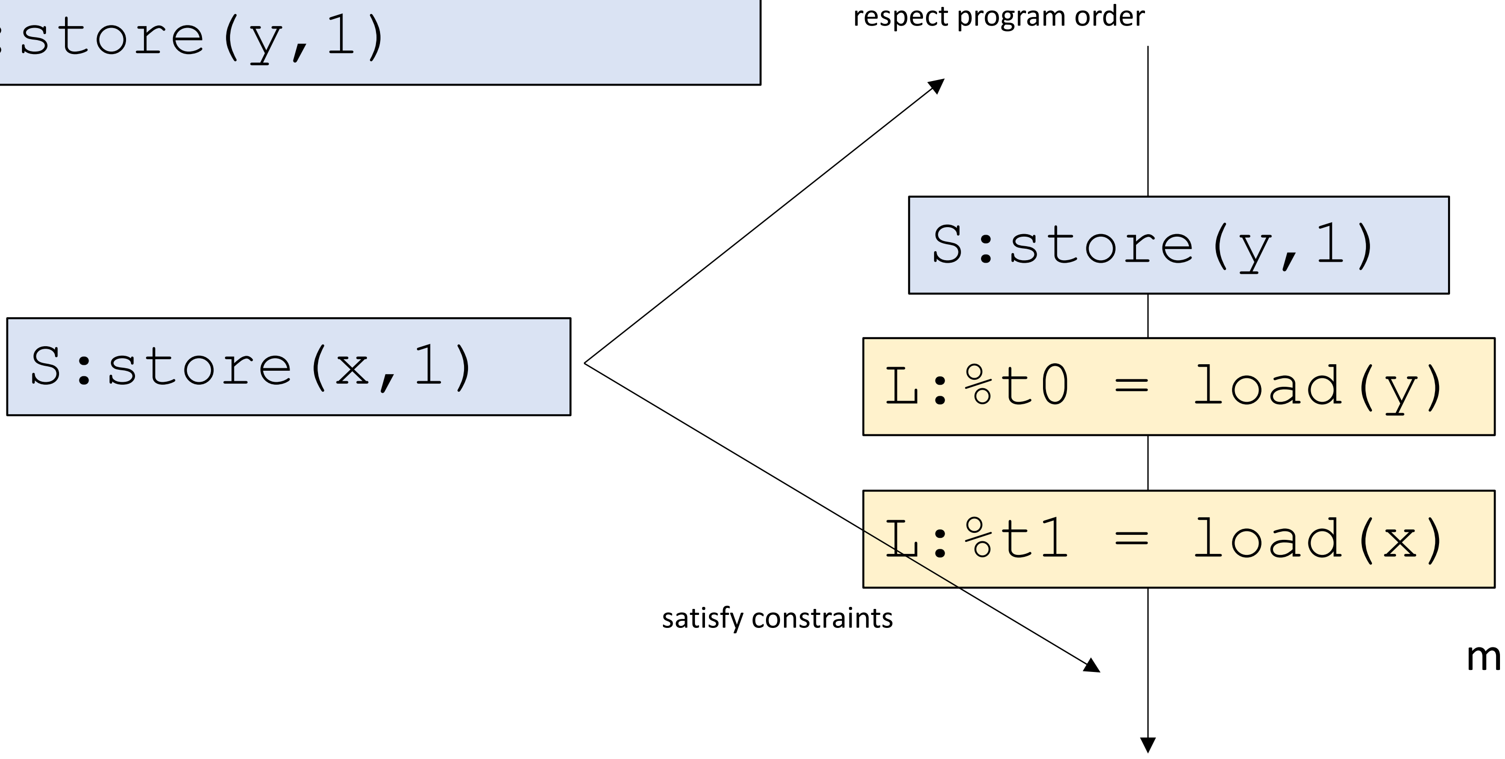
Question: can `t0 == 1 and t1 == 0?`

Thread 0:

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

Thread 1:

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



respect program order

`S:store(y,1)`

`L:%t0 = load(y)`

`L:%t1 = load(x)`

`S:store(x,1)`

satisfy constraints

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

What about TSO? NO

Global variable:

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

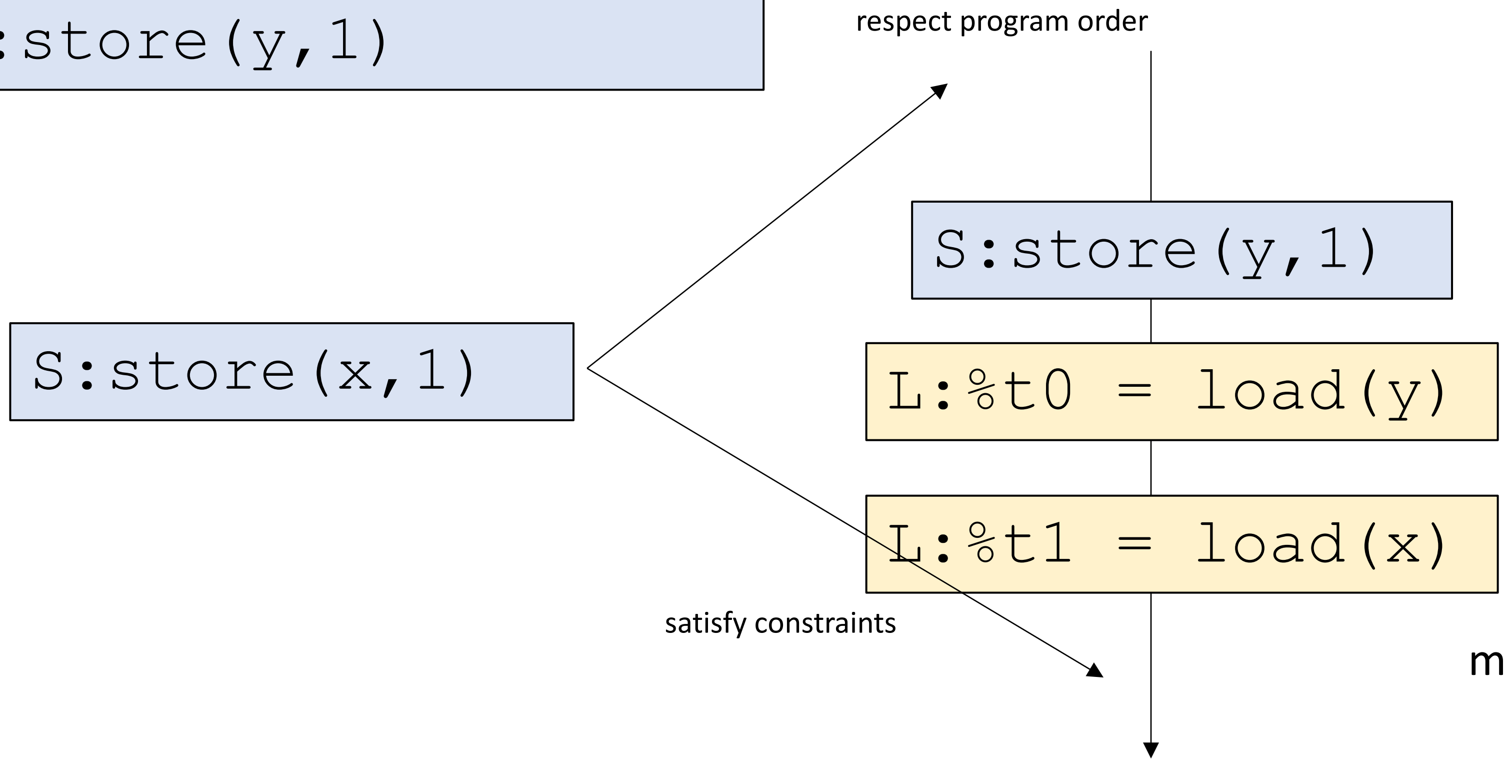
Question: can `t0 == 1 and t1 == 0?`

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
memory access 1	L	NO	Different address
	S	NO	Different address

What about PSO?

Global variable:

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

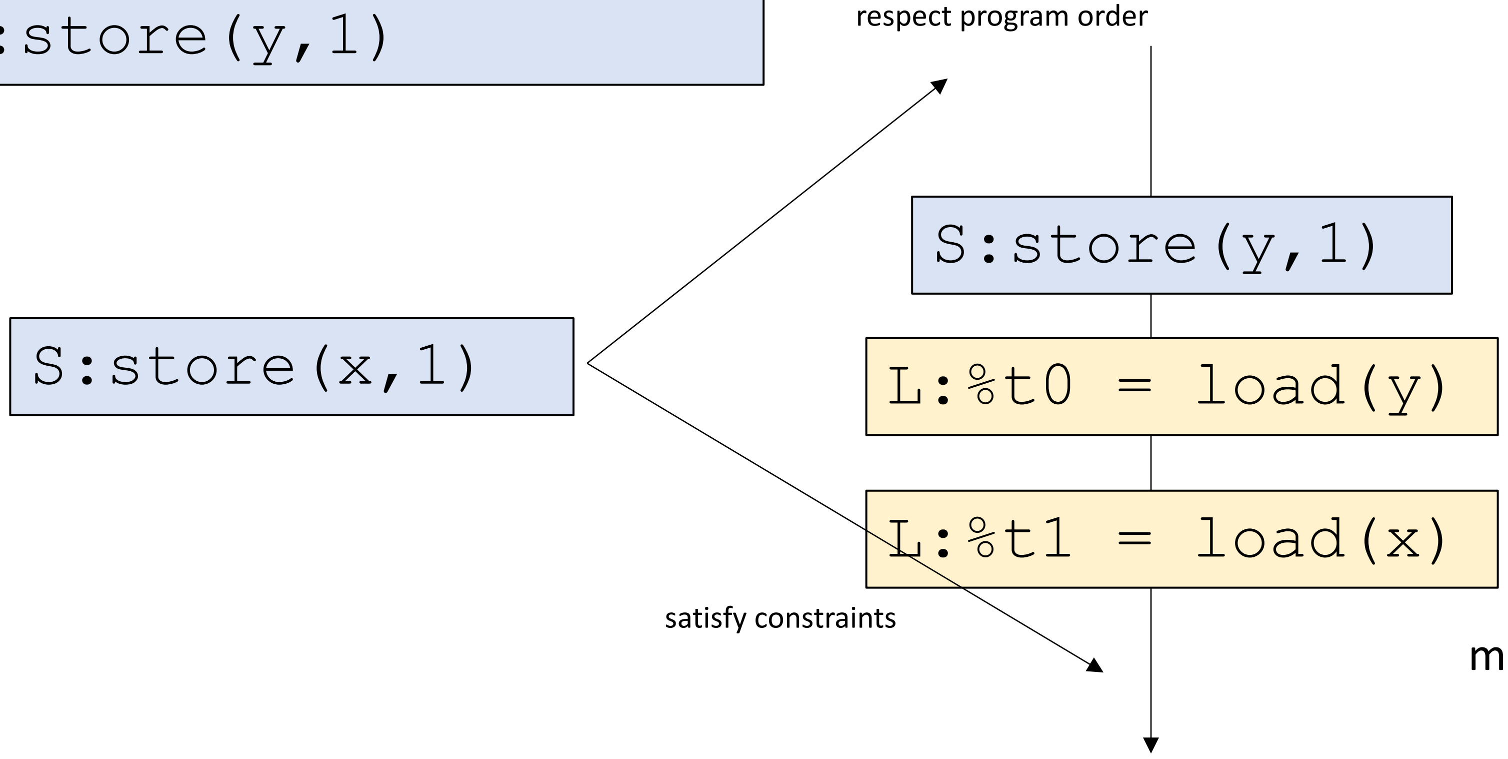
Question: can `t0 == 1 and t1 == 0`?

Thread 0:

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

Thread 1:

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



respect program order

```
S:store(y,1)
L:%t0 = load(y)
L:%t1 = load(x)
```

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

satisfy constraints

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

What about PSO?

Global variable:

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

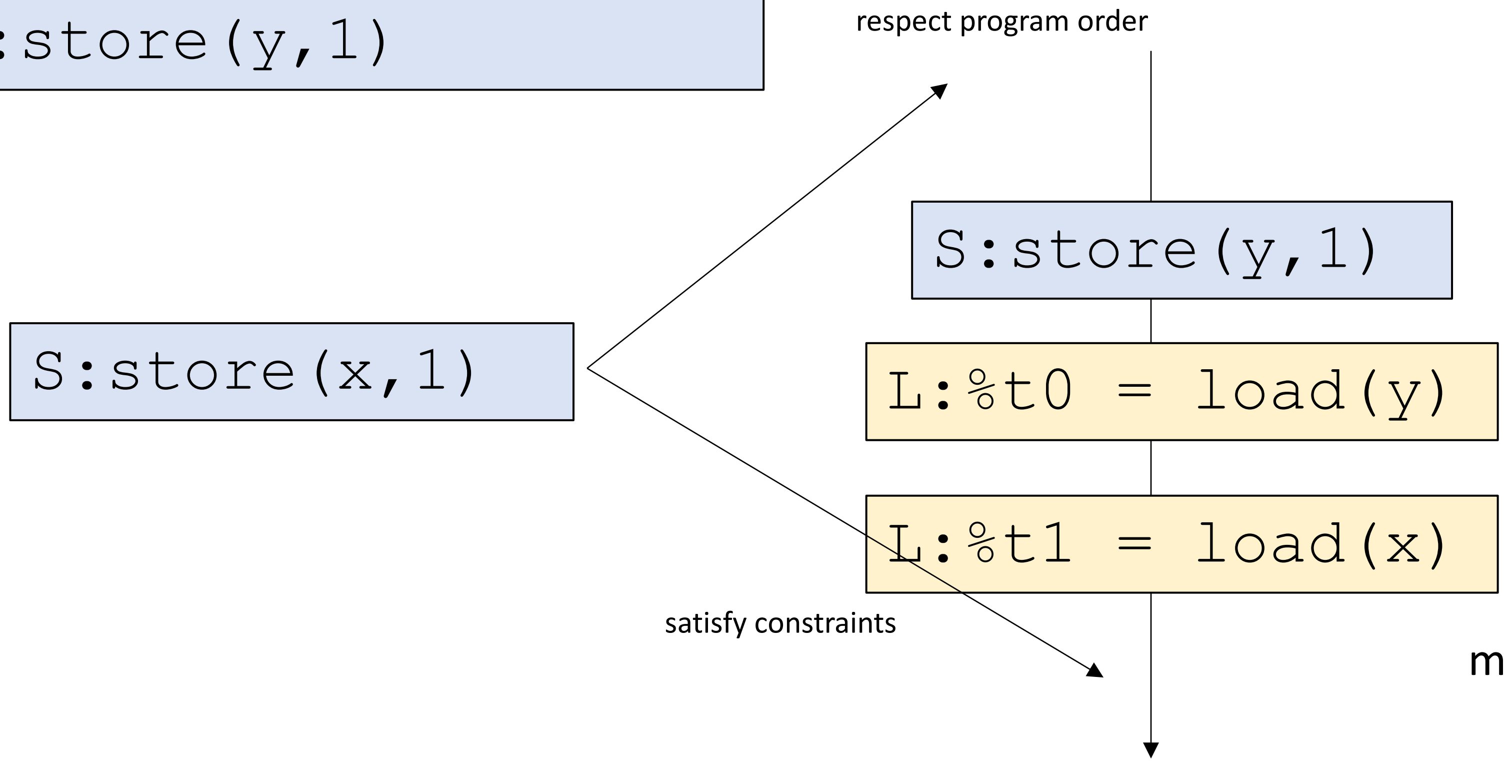
Question: can `t0 == 1 and t1 == 0?`

Thread 0:

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

Thread 1:

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



respect program order

```
S:store(y,1)
L:%t0 = load(y)
L:%t1 = load(x)
```

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

satisfy constraints

memory access 0

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

What about PSO? YES

Global variable:

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

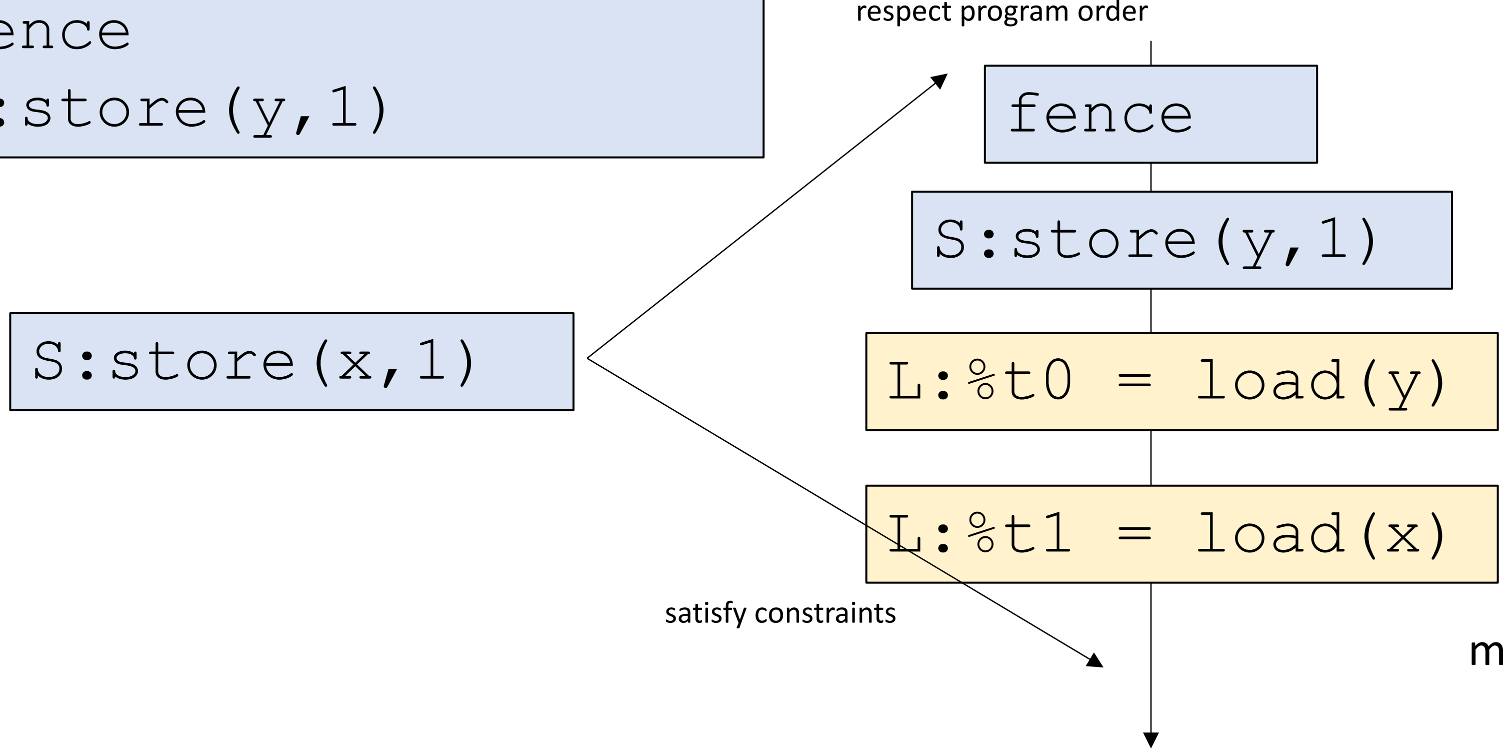
Question: can `t0 == 1 and t1 == 0?`

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
memory access 1	L	NO	Different address
	S	NO	Different address

Now it is disallowed in PSO

Global variable:

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

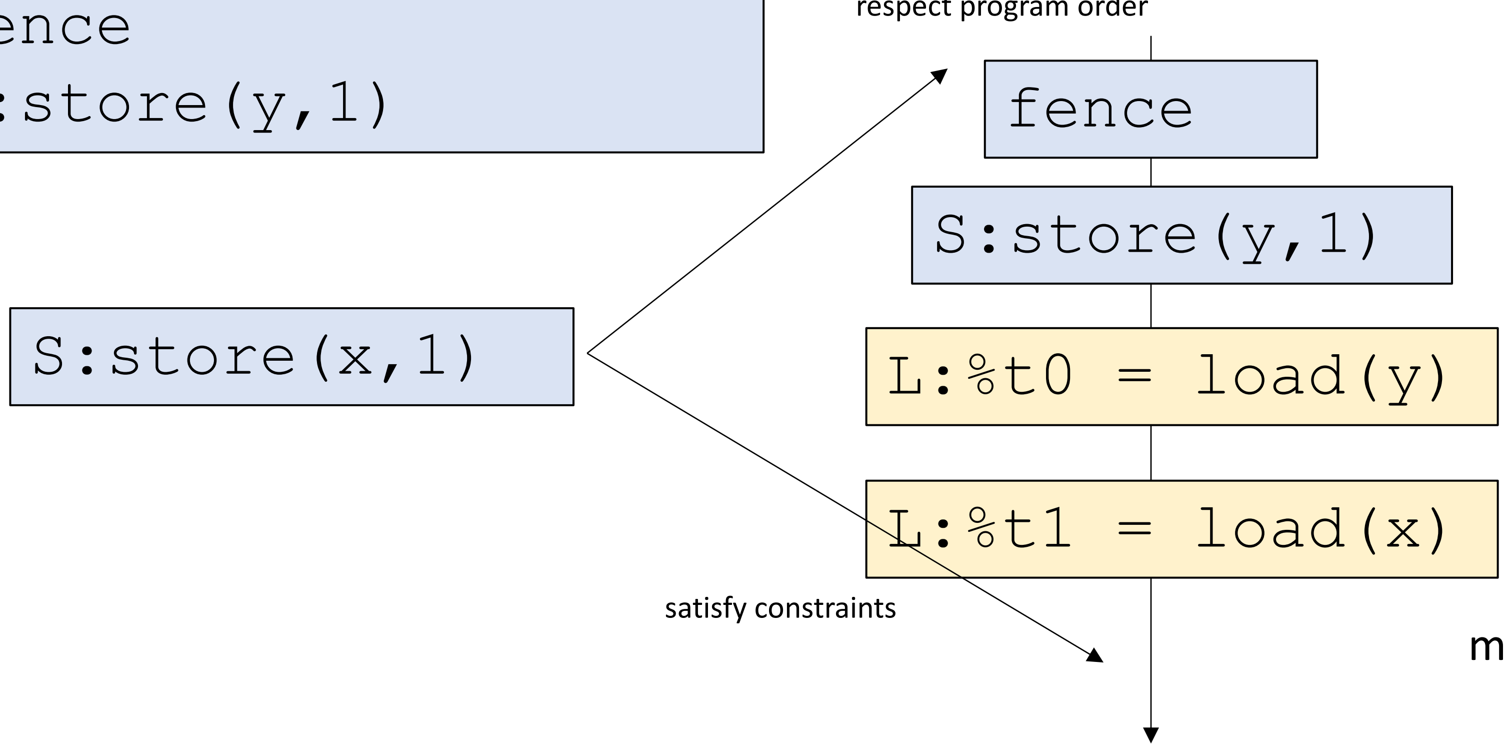
Question: can `t0 == 1 and t1 == 0`?

Thread 0:

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

Thread 1:

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



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

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

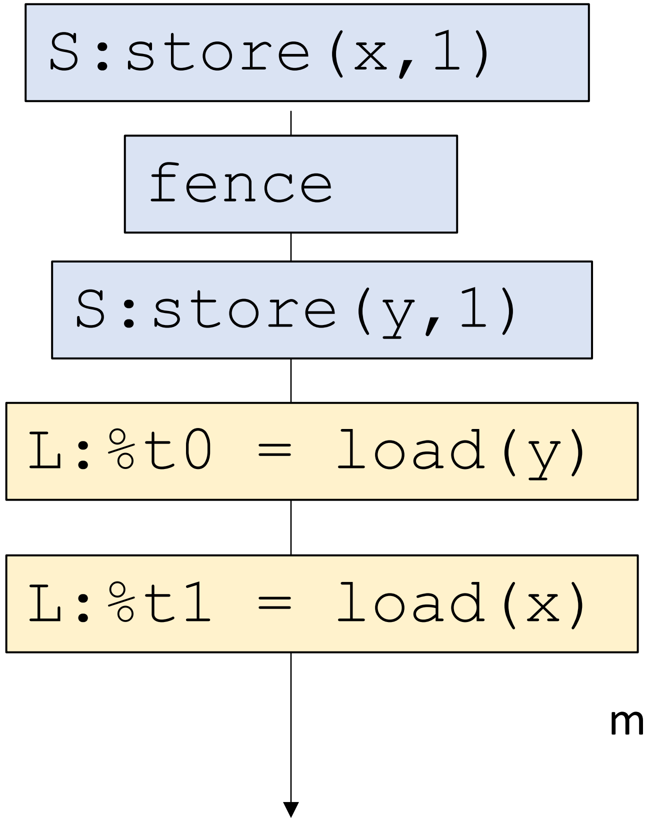
Question: can `t0 == 1 and t1 == 0?`

Thread 0:

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

Thread 1:

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



```
S:store(x,1)
fence
S:store(y,1)
L:%t0 = load(y)
L:%t1 = load(x)
```

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

What about RMO?

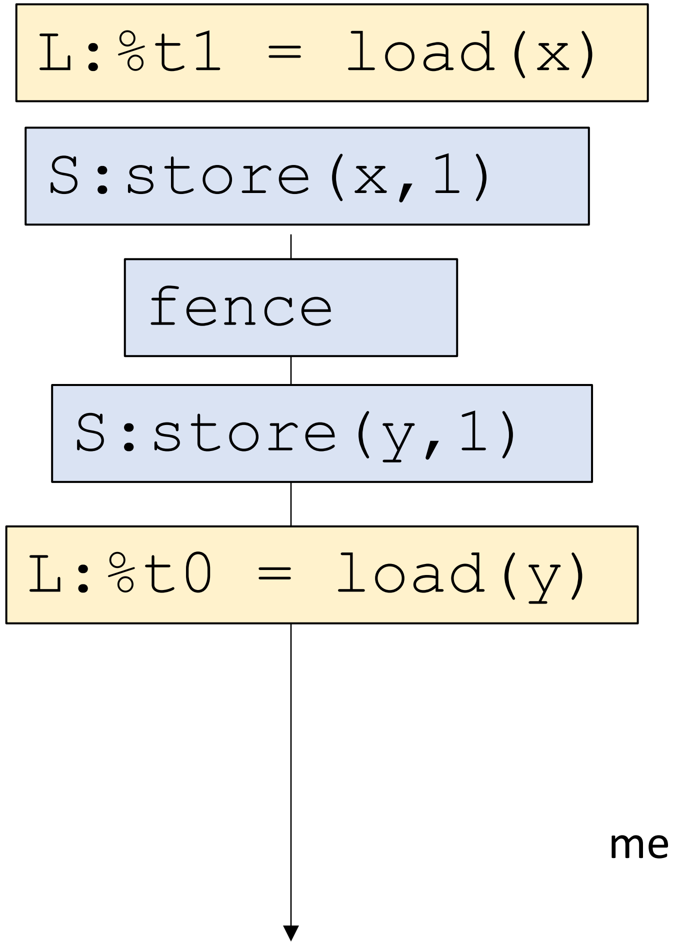
Global variable:

```
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};
```

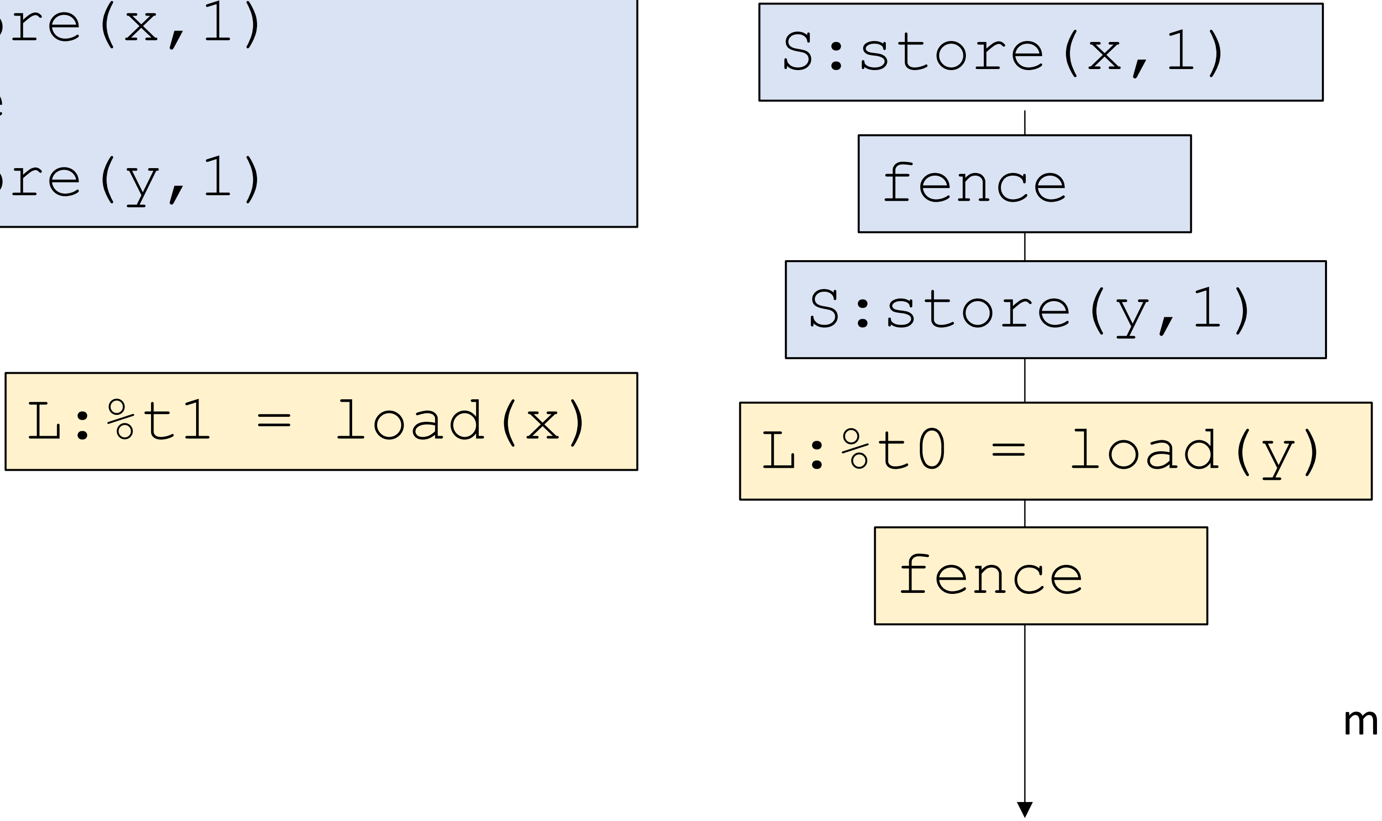
Question: can `t0 == 1 and t1 == 0`?

Thread 0:

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

Thread 1:

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



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

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

		memory access 0	
		L	S
memory access 1	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};
```

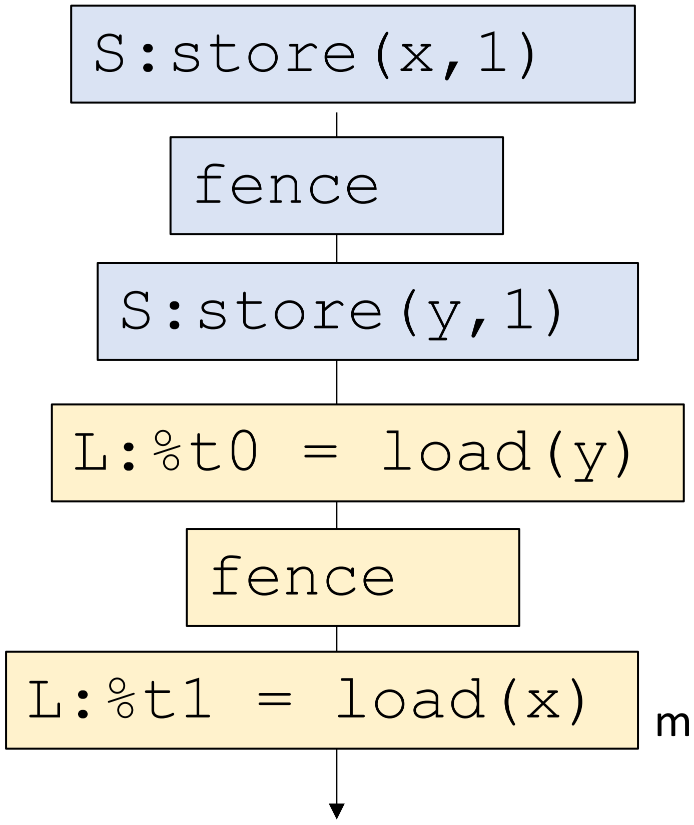
Question: can `t0 == 1 and t1 == 0?`

Thread 0:

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

Thread 1:

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



```
S:store(x,1)
fence
S:store(y,1)
L:%t0 = load(y)
fence
L:%t1 = load(x)
```

memory access 1

memory access 0

	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

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)

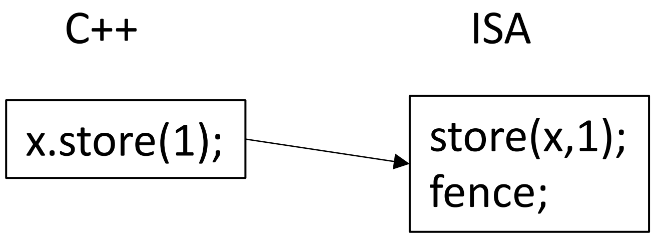
	L	S
L	NO	different address
S	NO	No

C++11 atomic operation compilation

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

language
C++11 (sequential consistency)

	L	S
L	NO	NO
S	NO	NO


C++

x.store(1);

ISA

store(x,1);
fence;

or

target machine
TSO (x86)

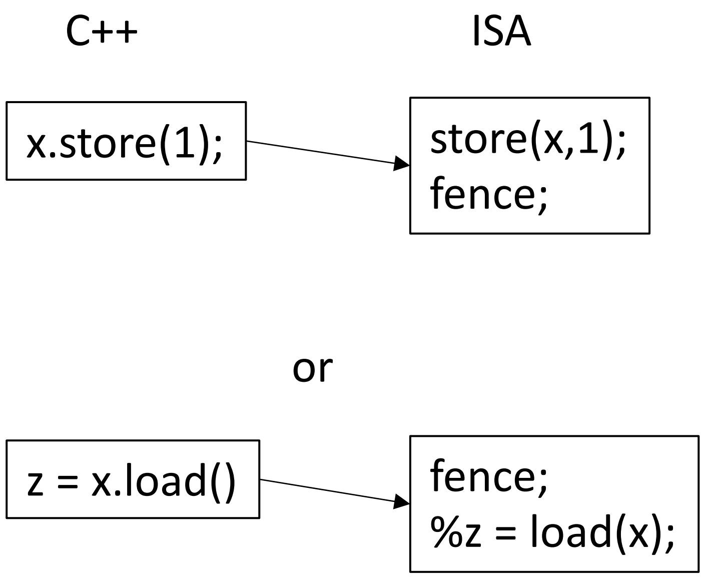
	L	S
L	NO	different address
S	NO	No

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



C++

ISA

```
x.store(1);
```

```
store(x,1);
fence;
```

or

```
z = x.load()
```

```
fence;
%z = load(x);
```

target machine
TSO (x86)

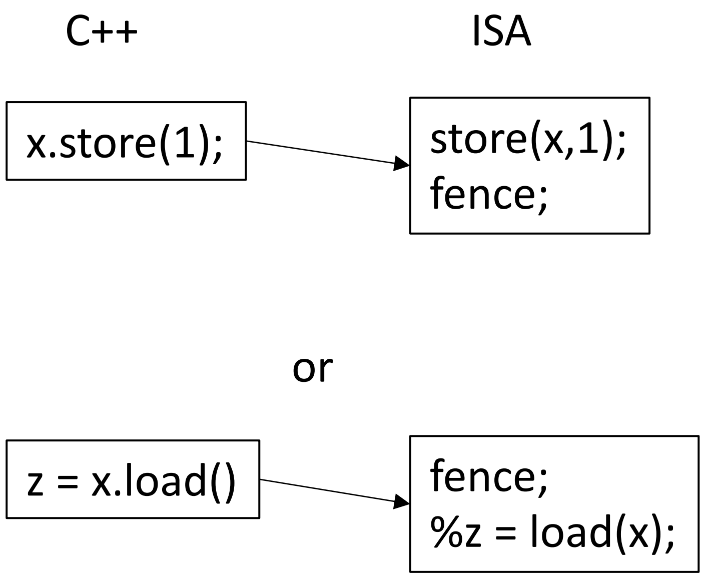
	L	S
L	NO	different address
S	NO	No

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



C++

ISA

x.store(1);

store(x,1);
fence;

or

z = x.load()

fence;
%z = load(x);

target machine
TSO (x86)

	L	S
L	NO	different address
S	NO	No

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

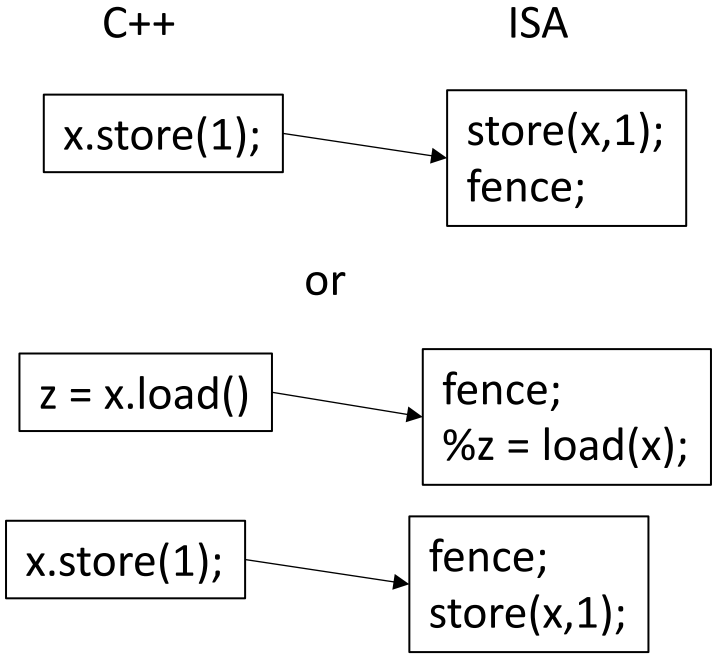
	L	S
L	NO	different address
S	NO	different address

C++11 atomic operation compilation

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

language
C++11 (sequential consistency)

	L	S
L	NO	NO
S	NO	NO



C++ ISA

```
x.store(1);   ->   store(x,1);
                   fence;
```

or

```
z = x.load()  ->   fence;
                   %z = load(x);

x.store(1);   ->   fence;
                   store(x,1);
```

target machine
PSO

	L	S
L	NO	different address
S	NO	different address

Memory orders

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

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)

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

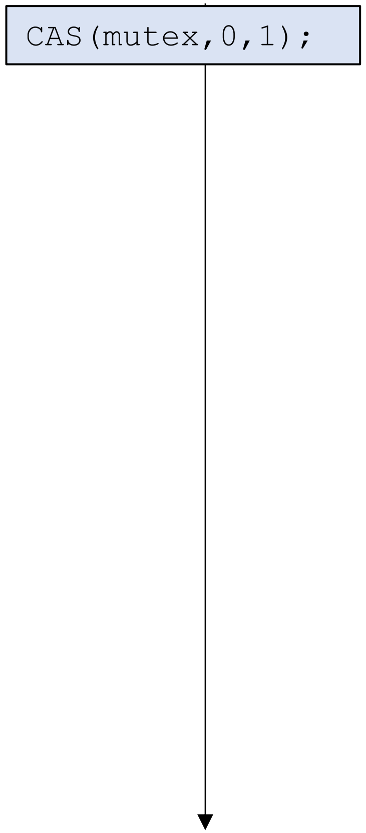
What is an execution?

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);
```

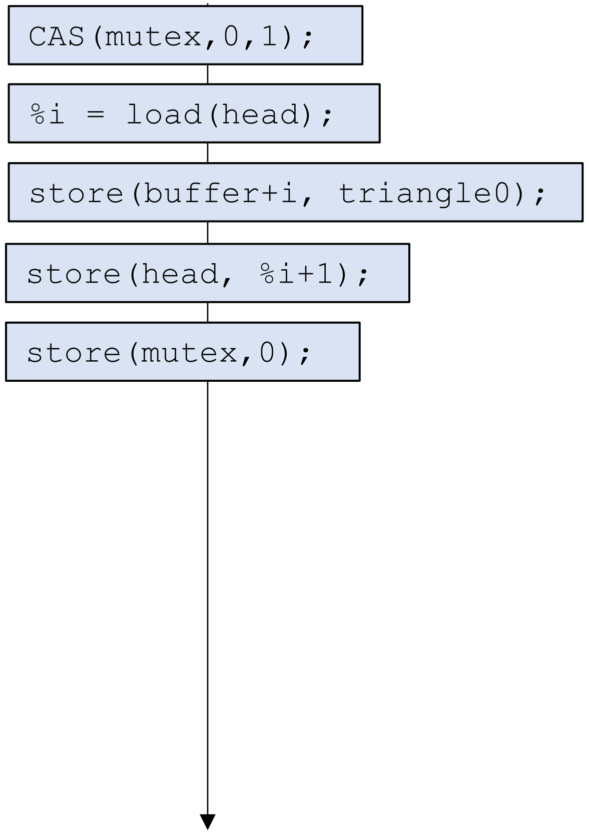
if blue goes first
it gets to complete
its critical section
while thread 1 is spinning

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);
```



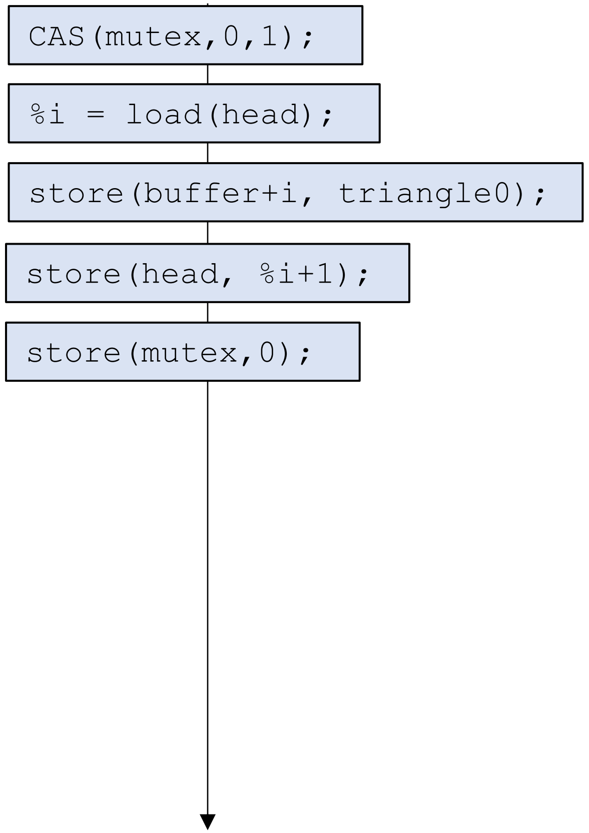
```
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

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);
```

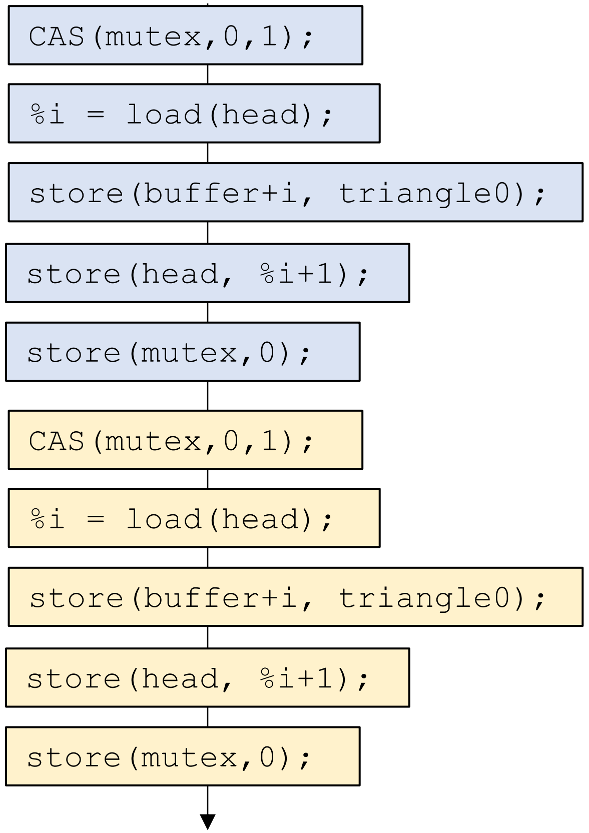
now yellow gets a change to go

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);
```



```
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);
```

now yellow gets a change to go

Thread 0:

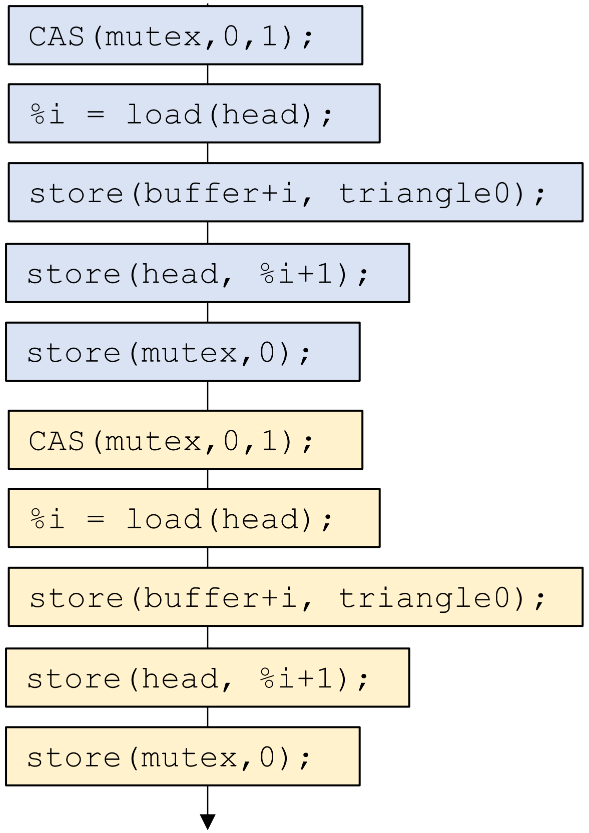
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

Thread 1:

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

what can happen in a PSO memory model?

	L	S
L	NO	Different address
S	NO	Different address



```
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:

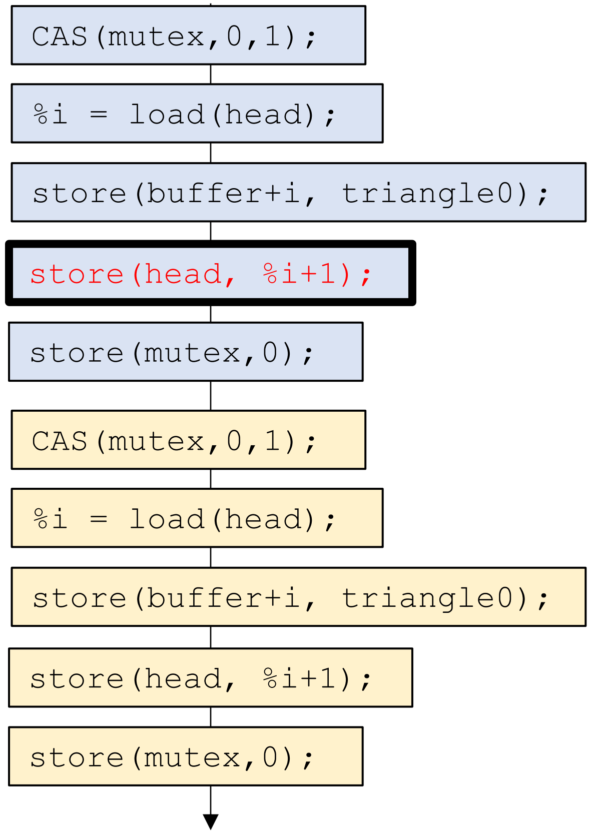
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

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



```
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:

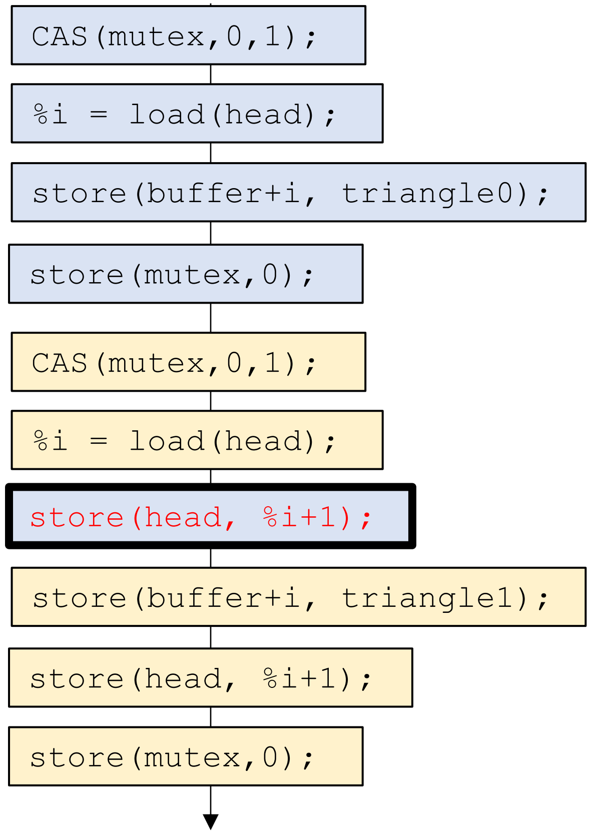
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

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



```
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(mutex,0);
CAS(mutex,0,1);
%i = load(head);
store(head, %i+1);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex,0);
```

What just happened if this store moves?

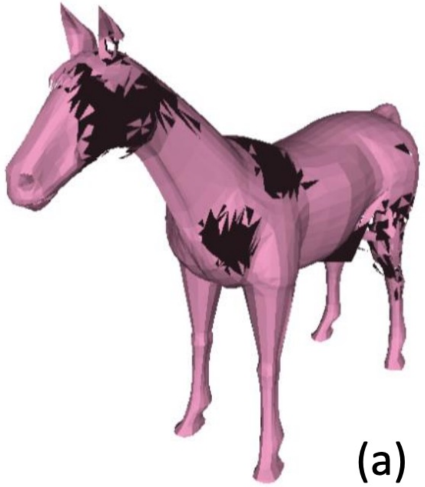
Nvidia in 2015

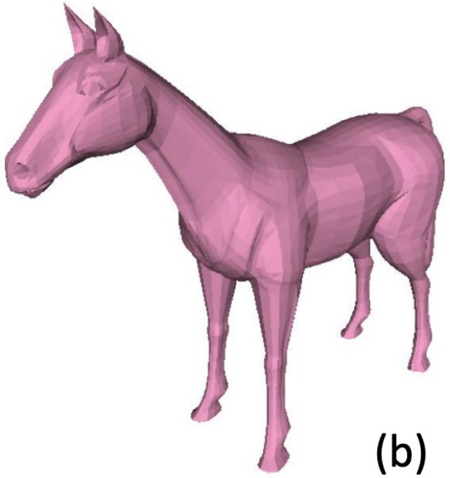
- Nvidia architects implemented a weak memory model

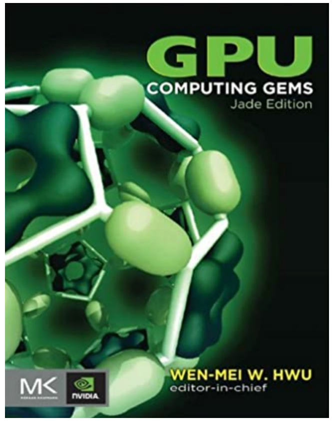
- Nvidia programmers expected a strong memory model

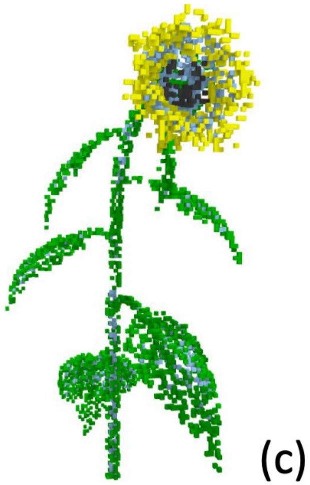
- Mutexes implemented without fences!

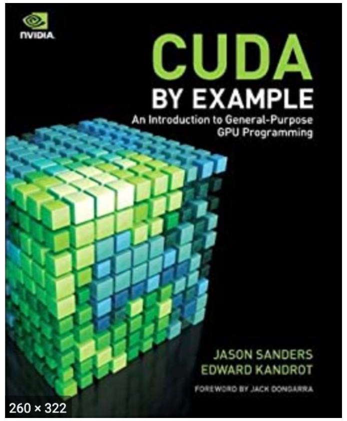
Nvidia in 2015











Thread 0:

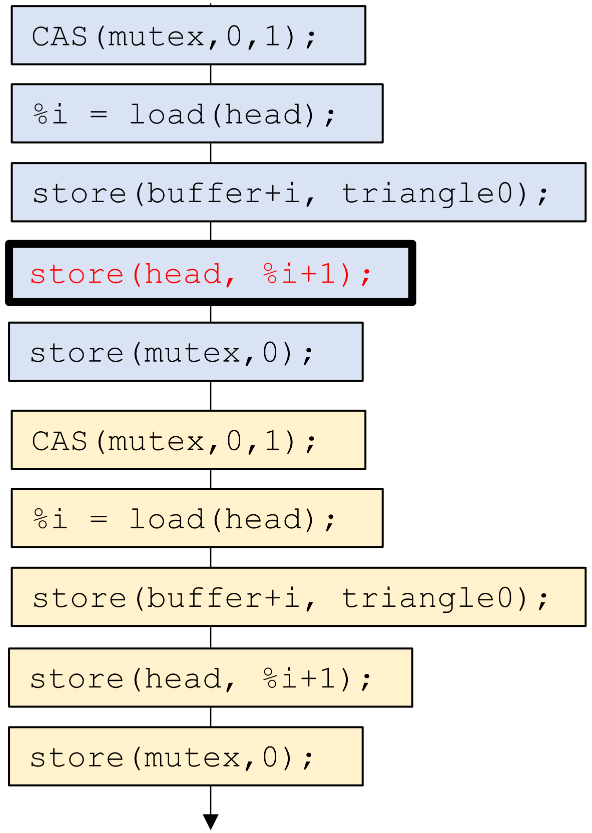
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

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



```
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);
```

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!

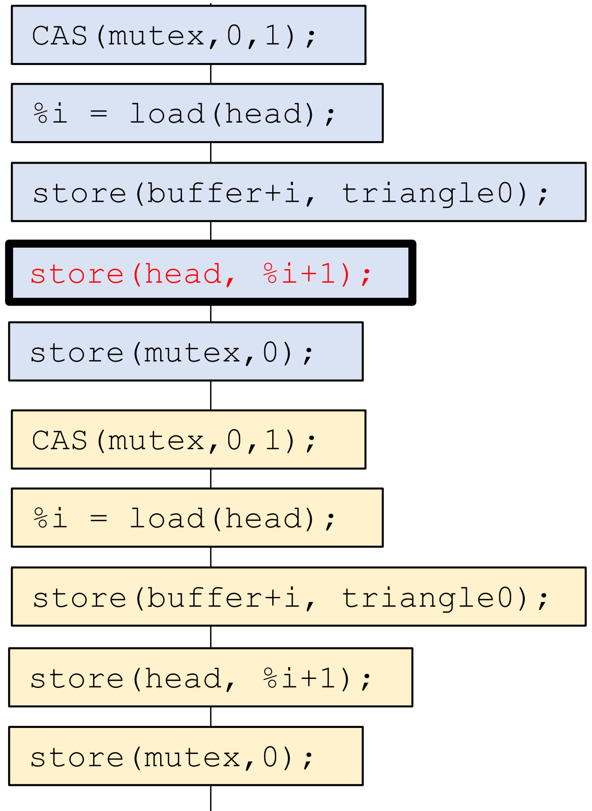
Thread 1:

```
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



```
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);
```

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!

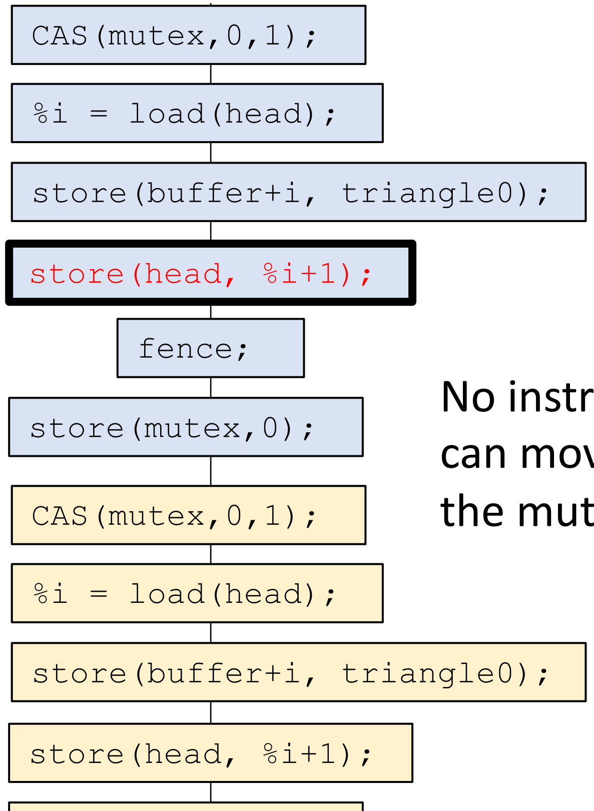
Thread 1:

```
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



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

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

	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

General Concurrent Set

Set Interface

- Unordered collection of items
- No duplicates

- We will implement this as a sorted linked list

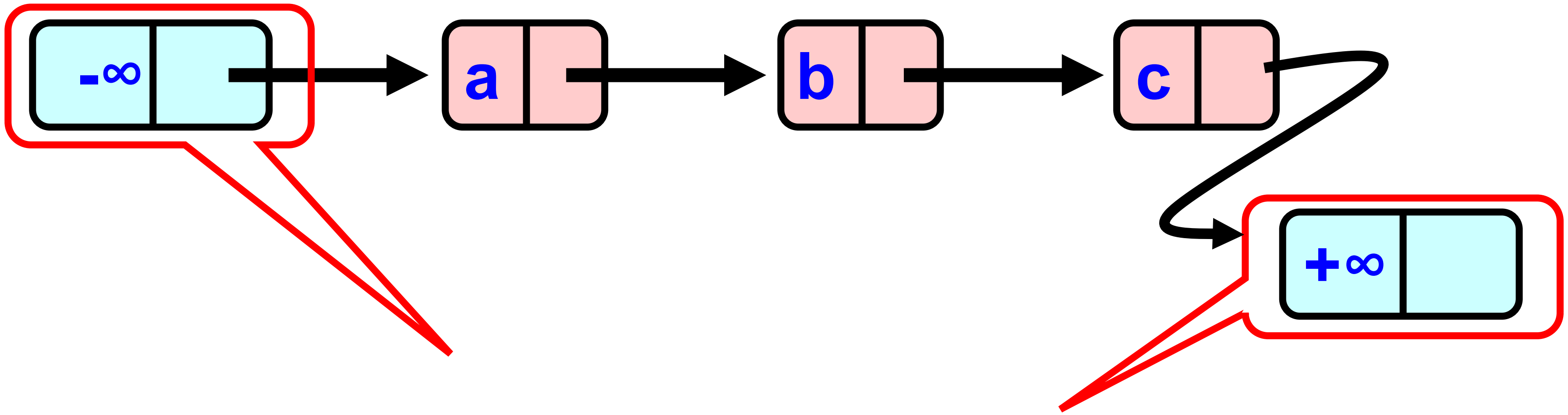
Set Interface

- 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;
}
```

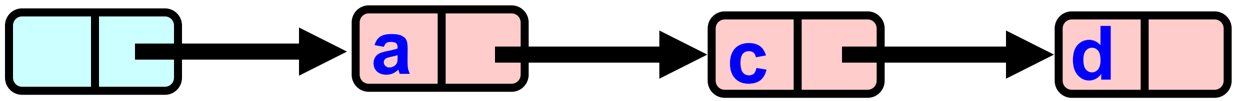
The List-Based Set



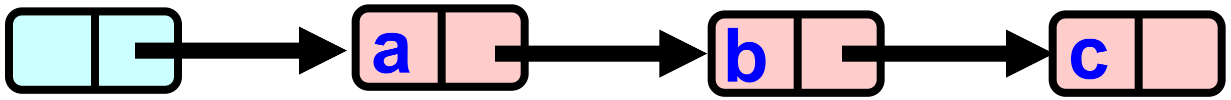
Sorted with Sentinel nodes
(min & max possible keys)

Sequential List Based Set

add(b)

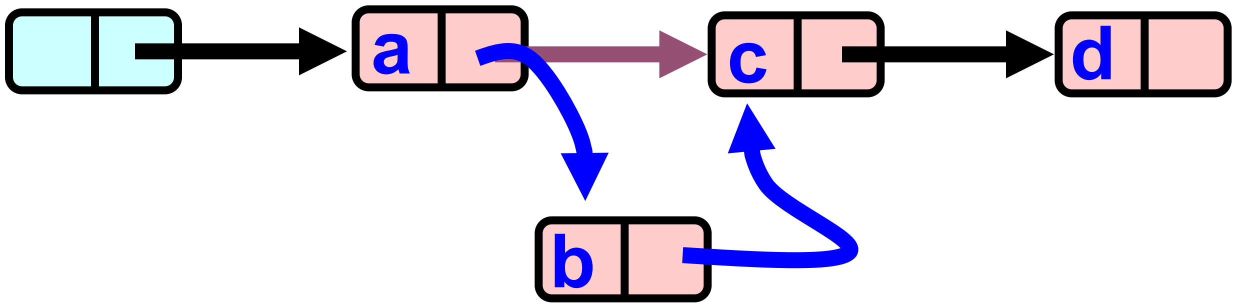


remove(b)

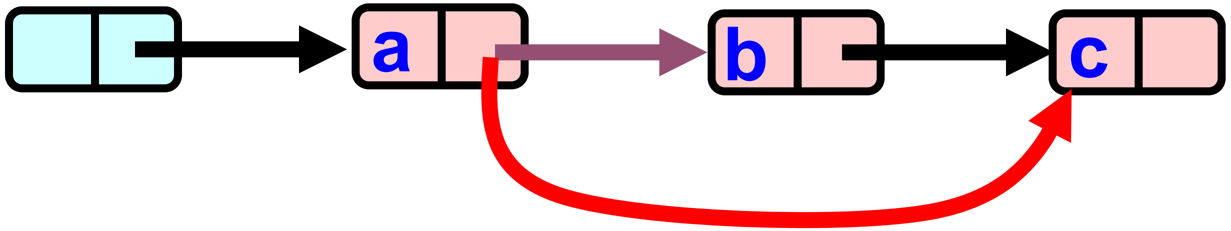


Sequential List Based Set

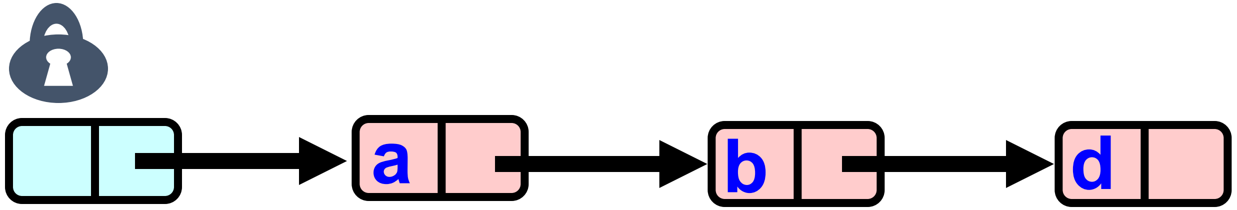
add(b)



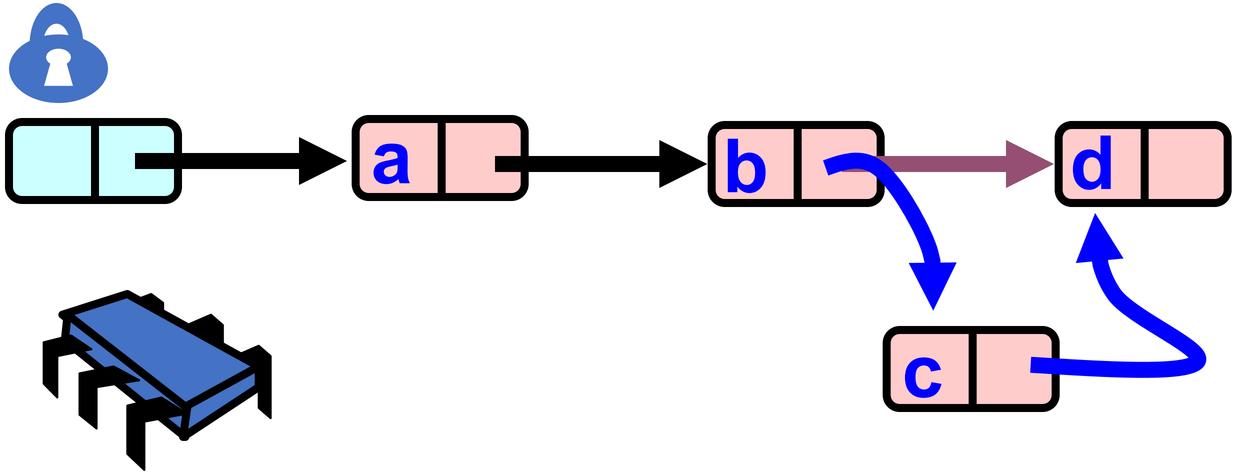
remove(b)



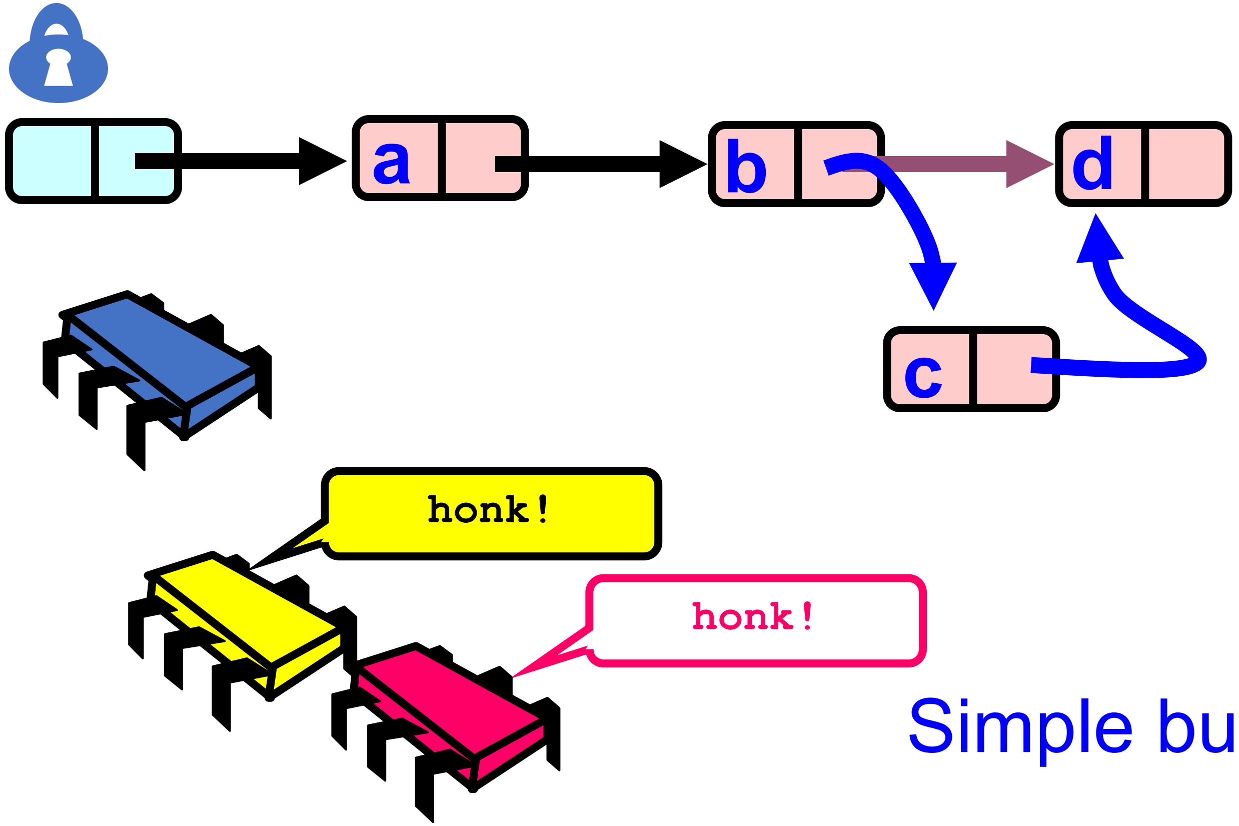
Coarse-Grained Locking



Coarse-Grained Locking



Coarse-Grained Locking



Simple but inefficient!

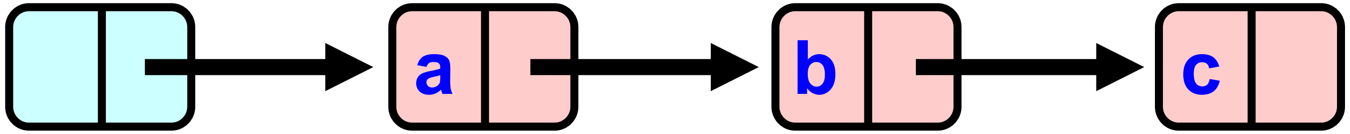
Schedule

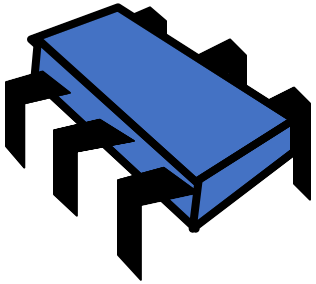
- Concurrent set
  - Coarse-grained lock
  - **fine-grained lock**
  - optimistic locking

Fine-grained Locking

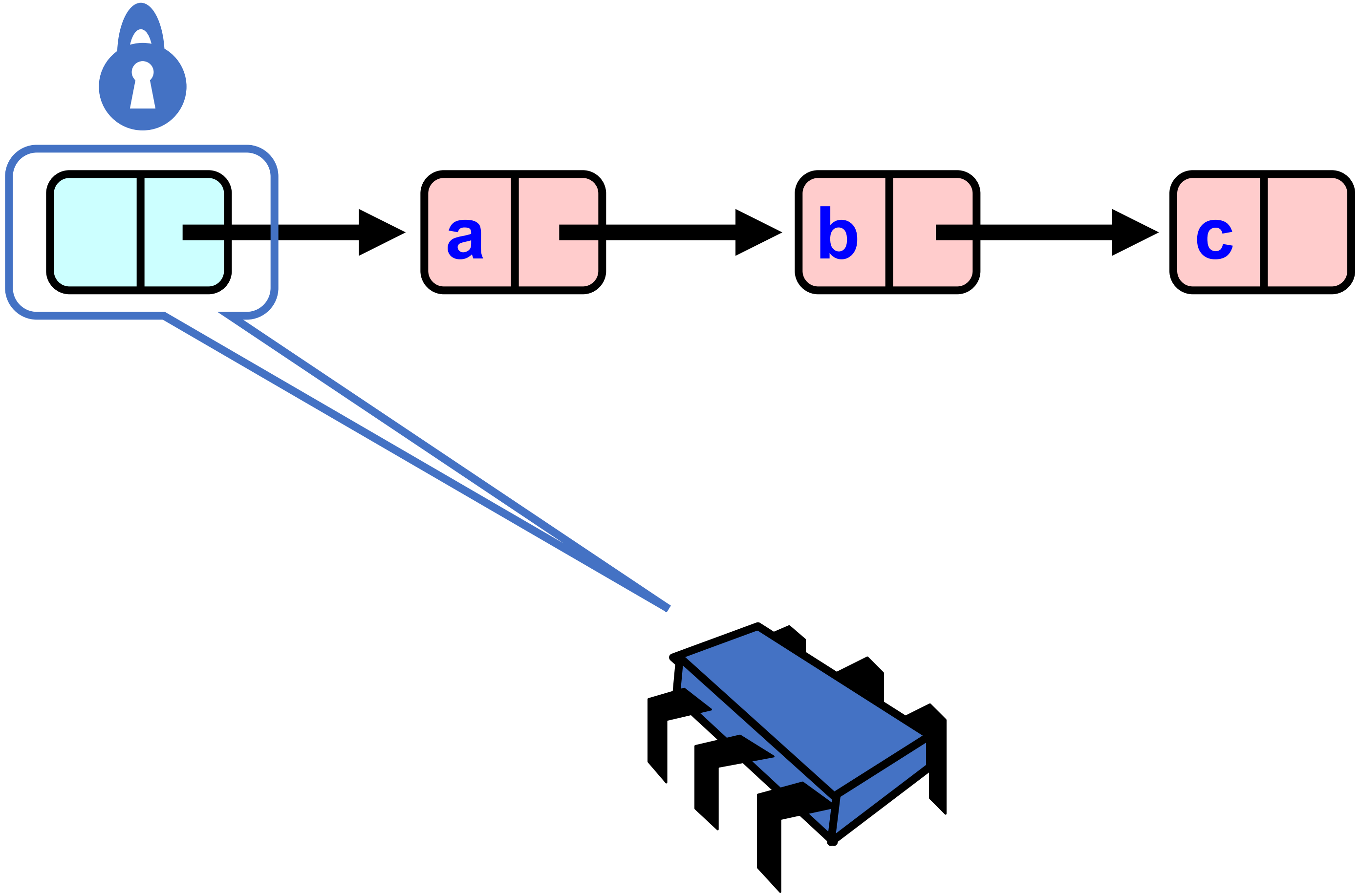
- Requires **careful** thought
- Split object into pieces
    - Each piece has own lock
    - Methods that work on disjoint pieces need not exclude each other

Hand-over-Hand locking

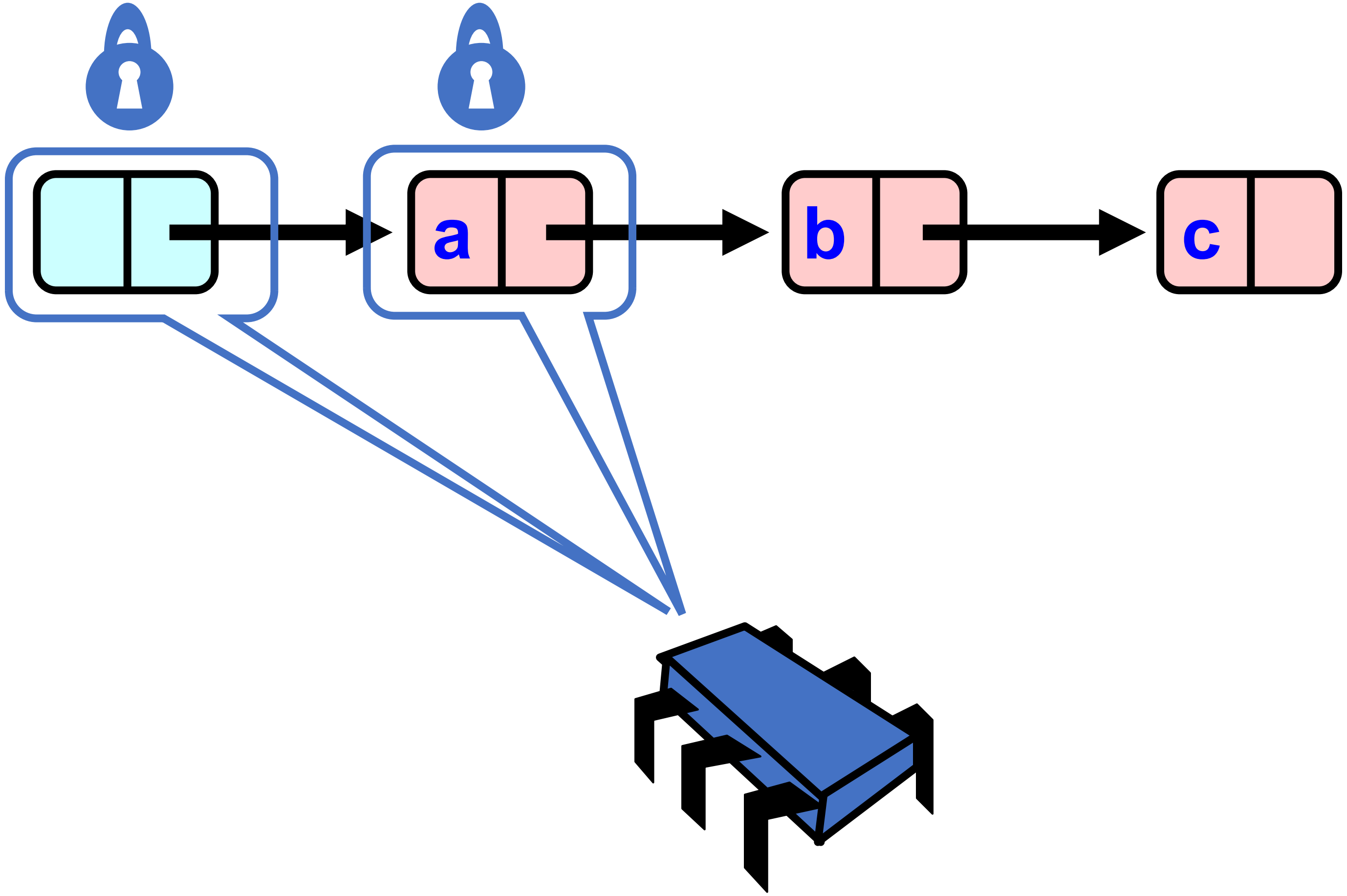




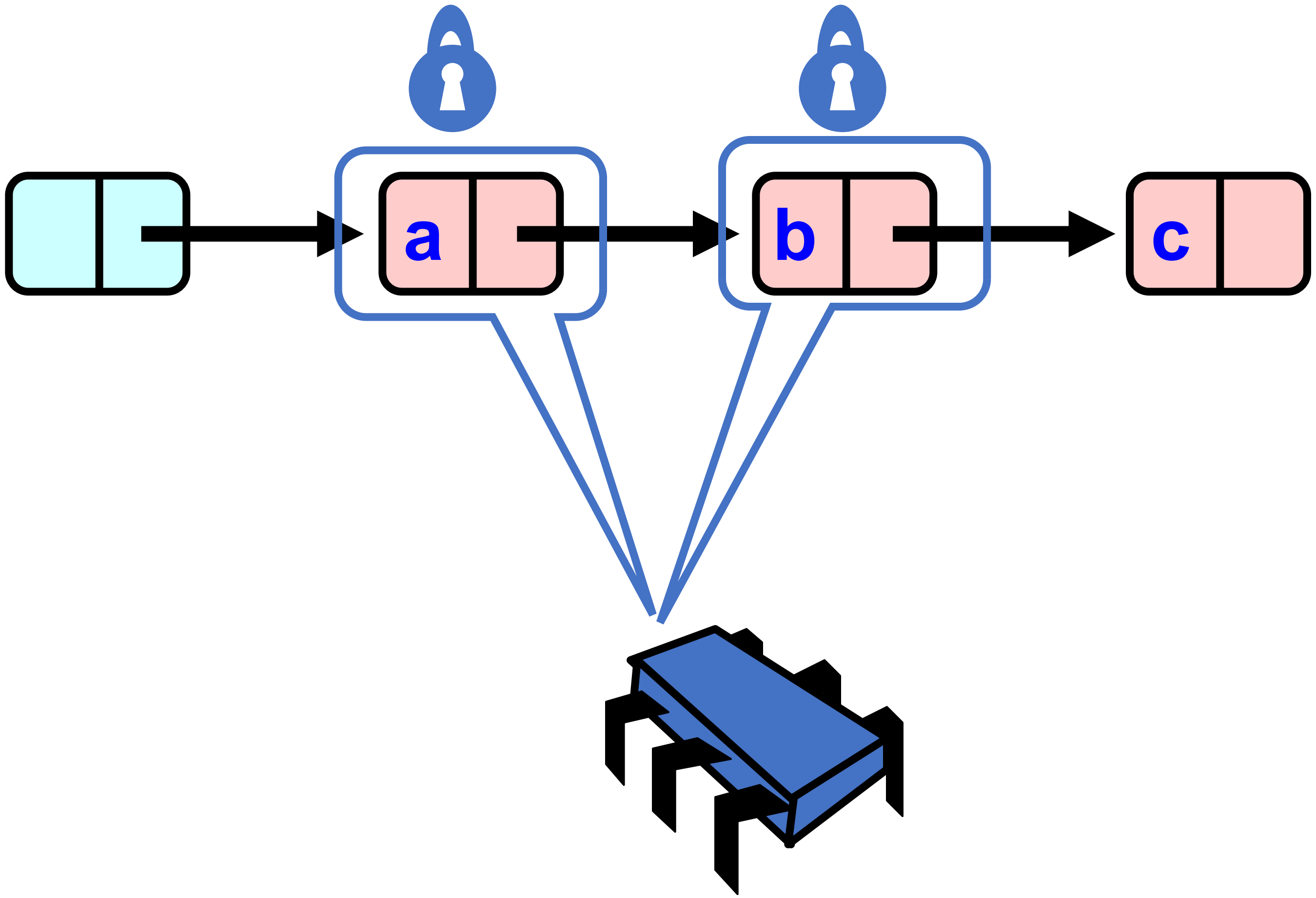
Hand-over-Hand locking



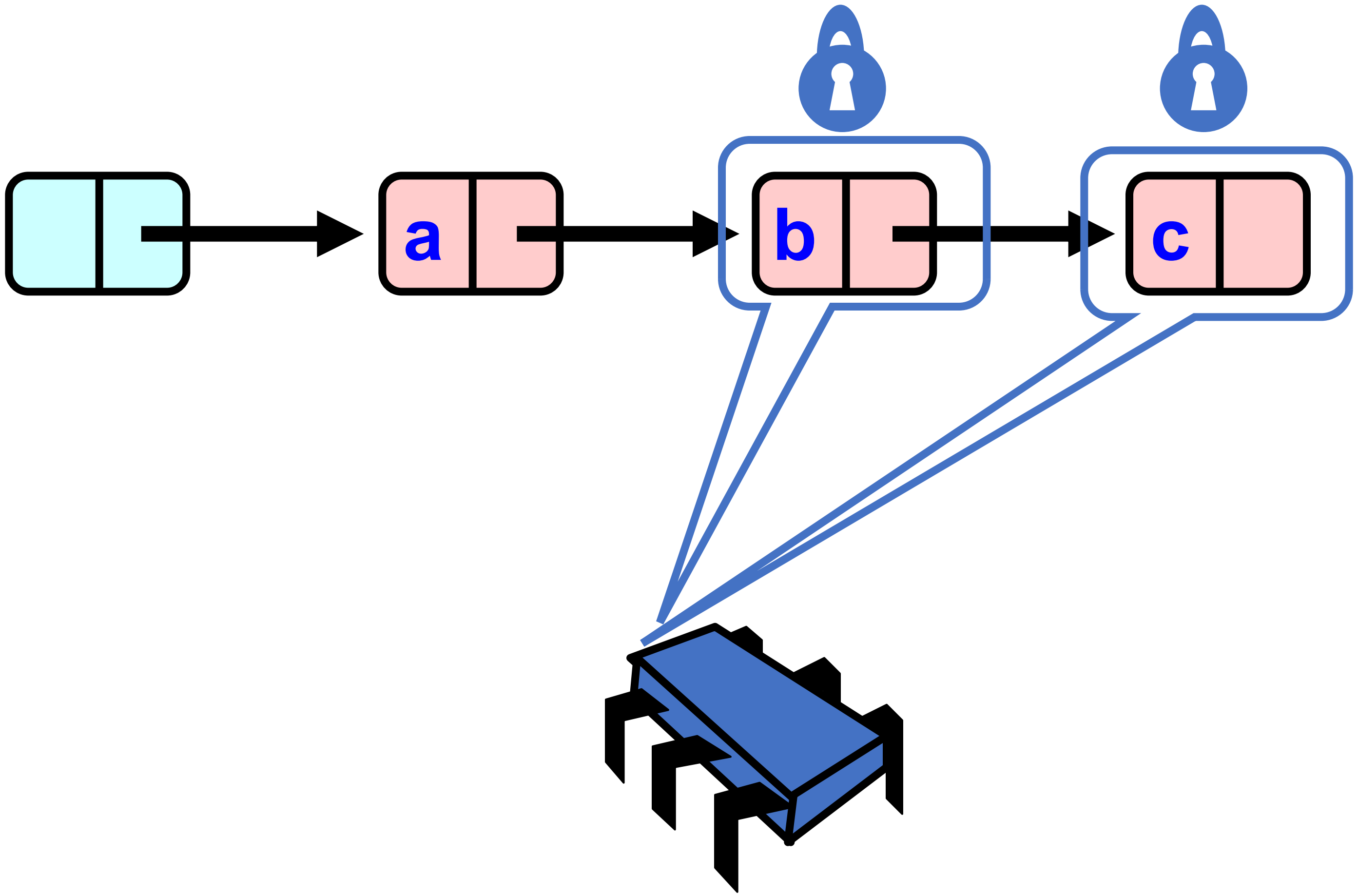
Hand-over-Hand locking



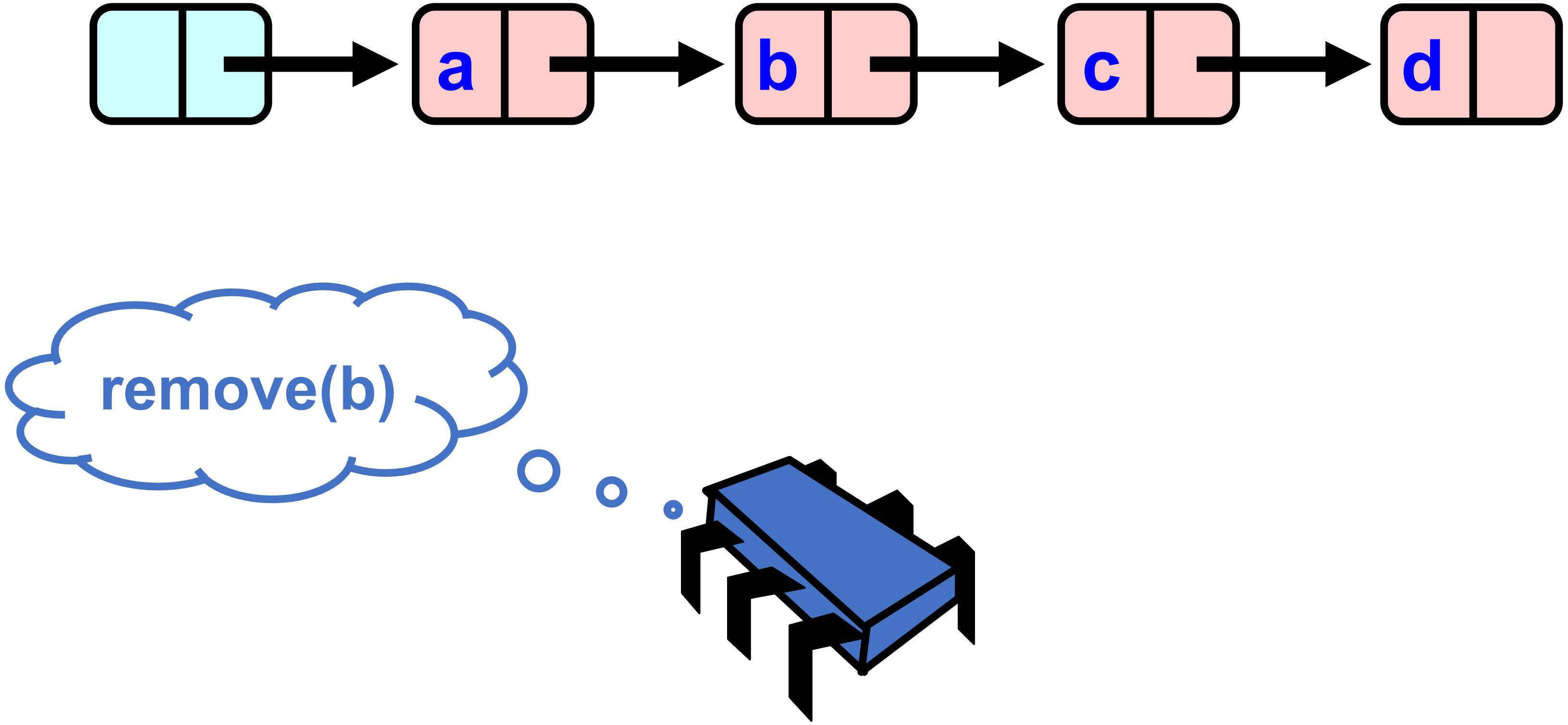
Hand-over-Hand locking



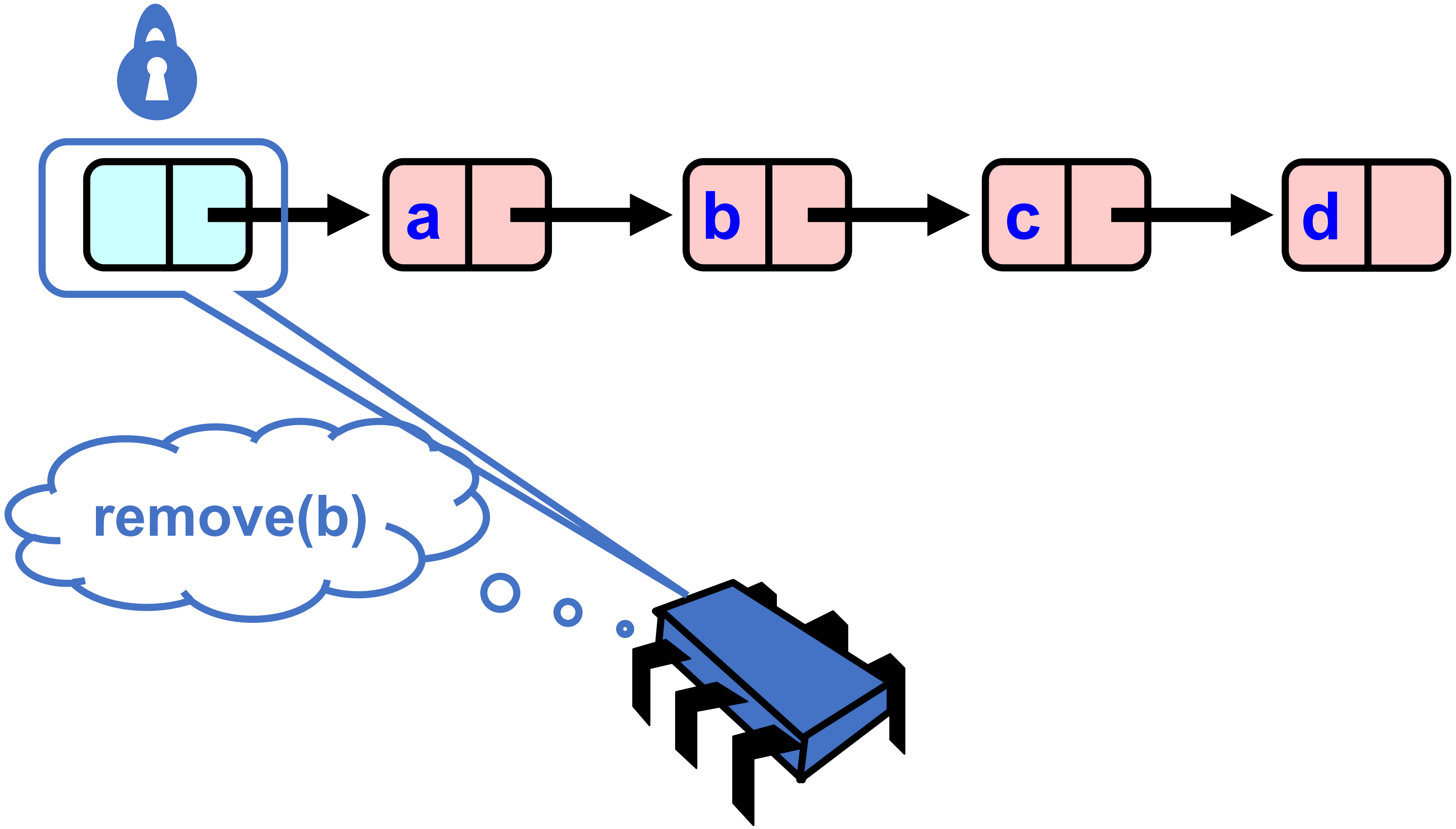
Hand-over-Hand locking



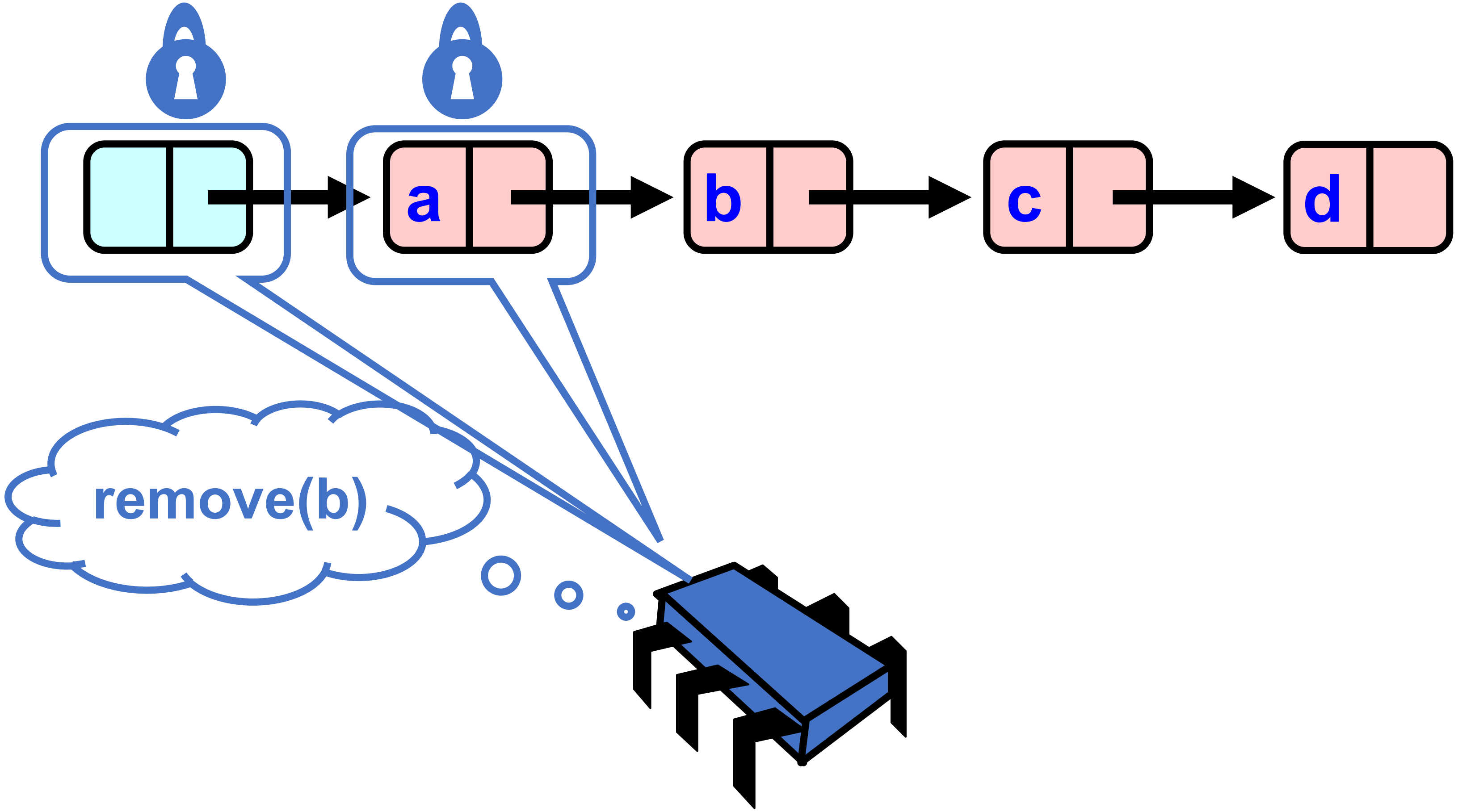
Removing a Node



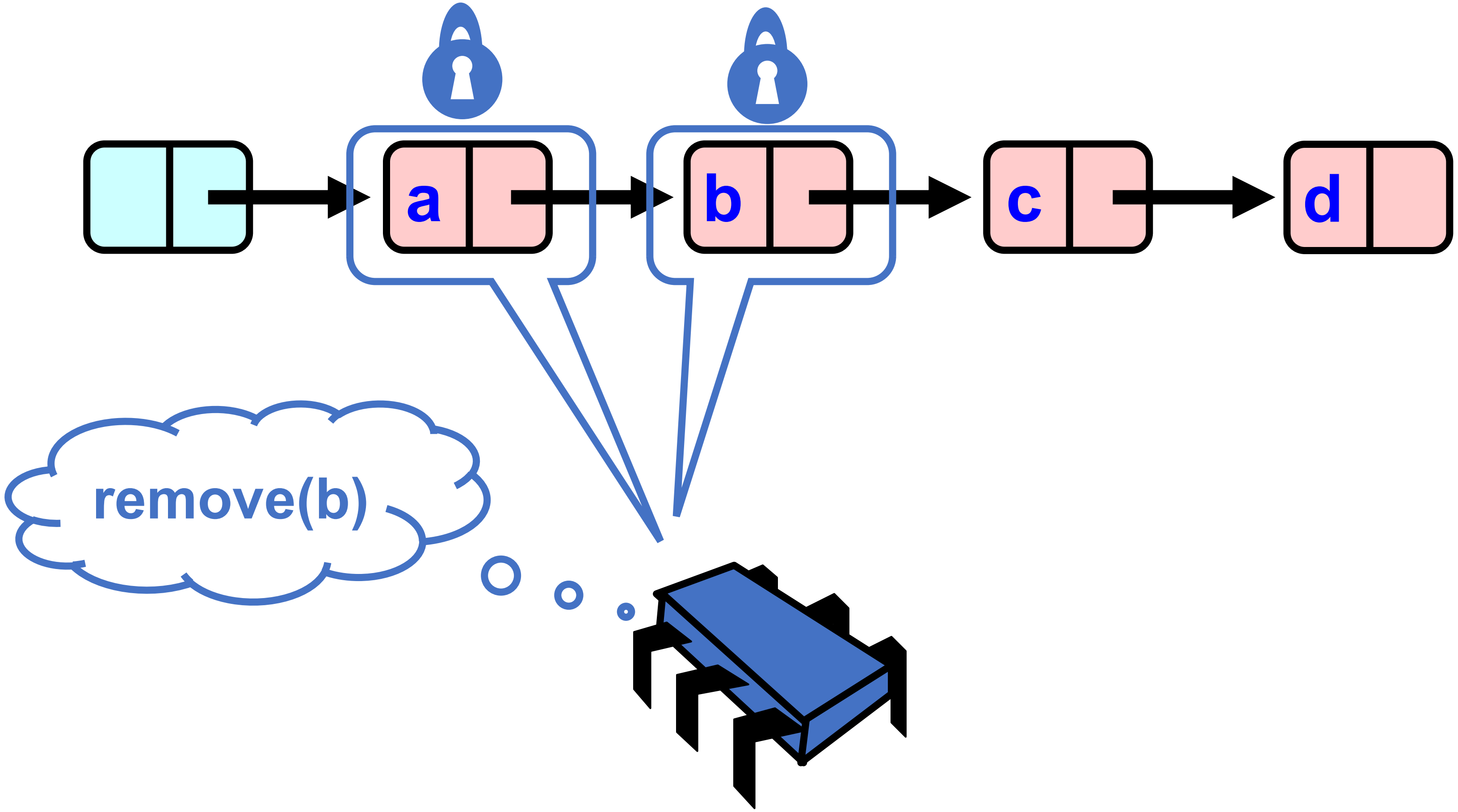
Removing a Node



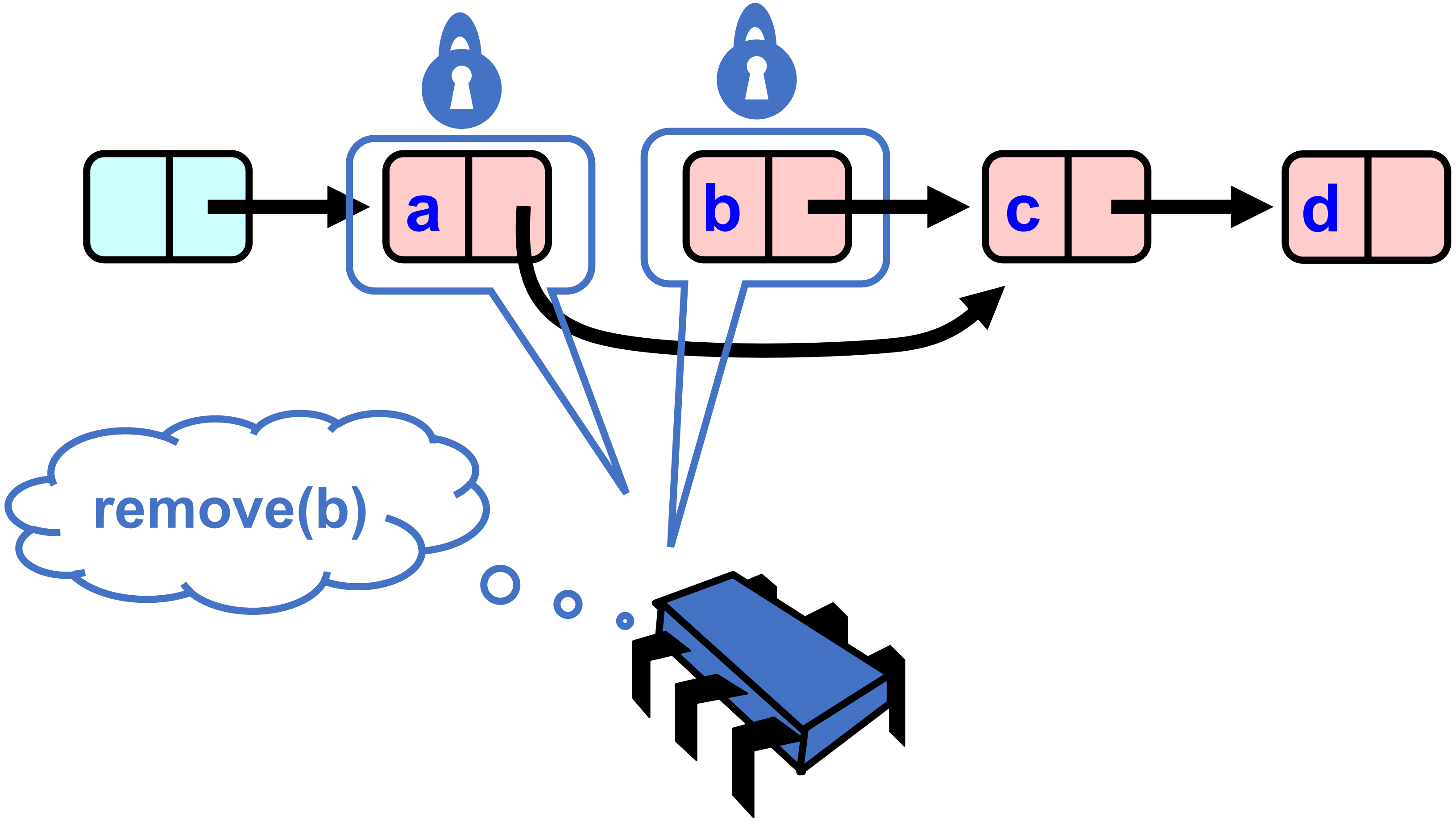
Removing a Node



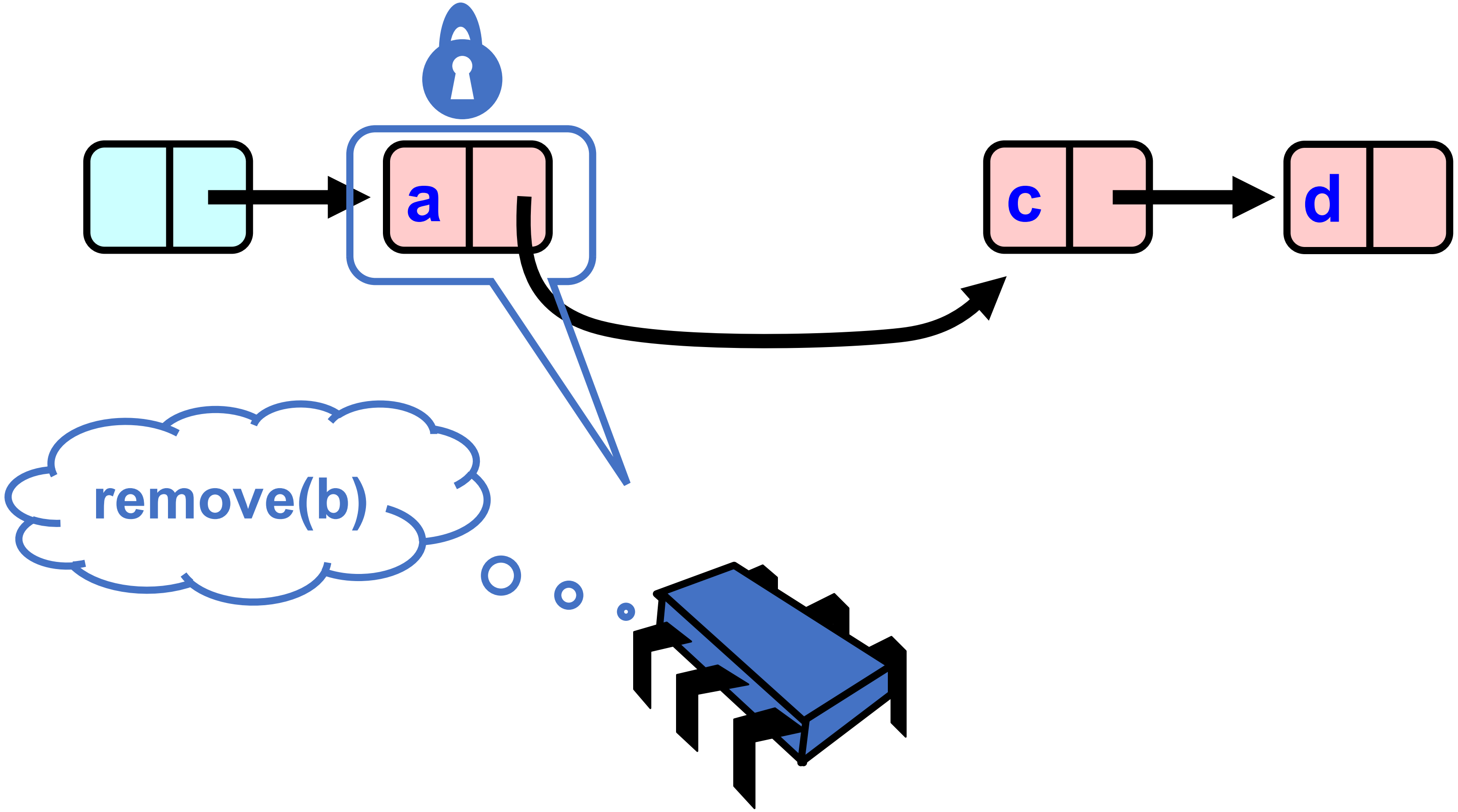
Removing a Node



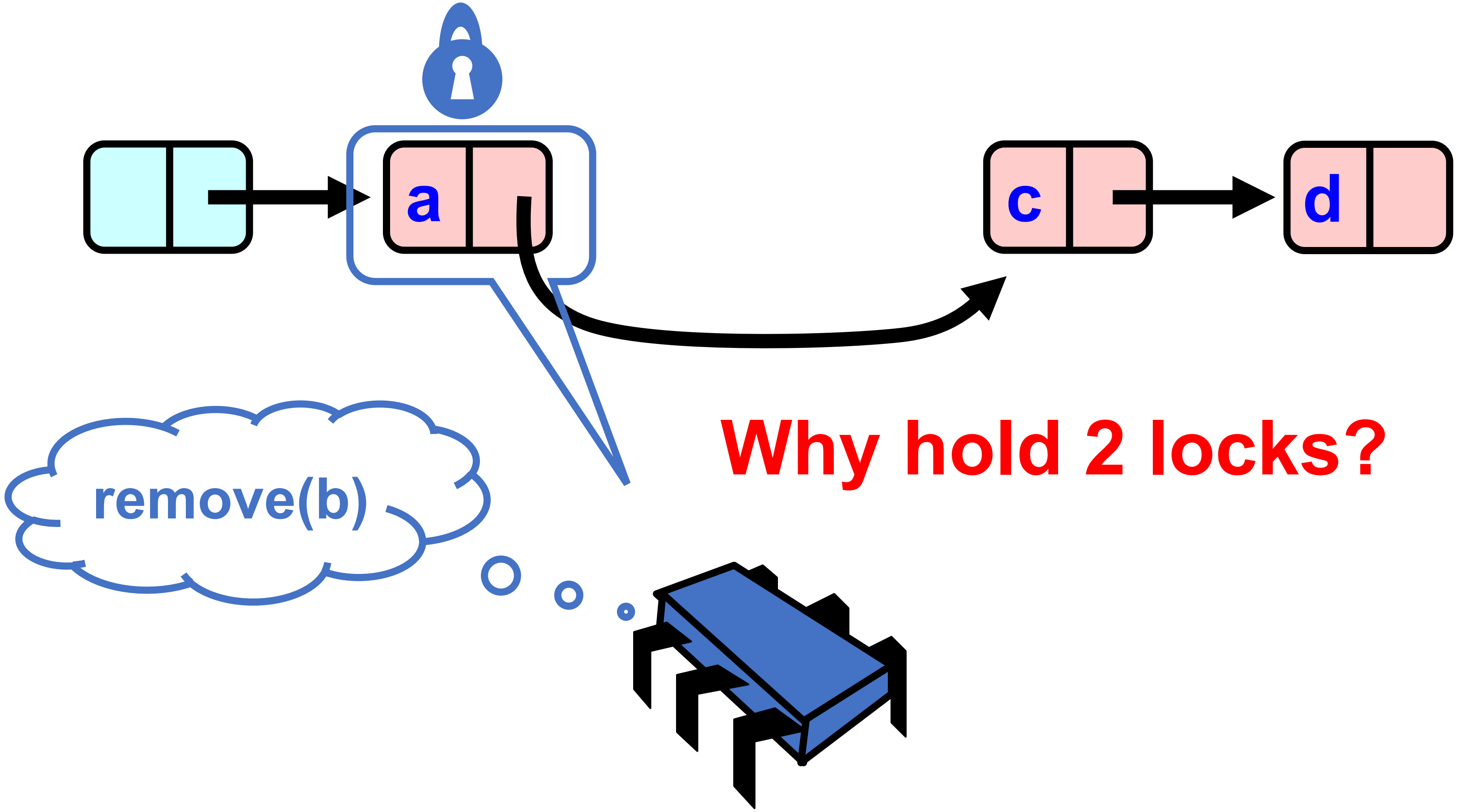
Removing a Node



Removing a Node

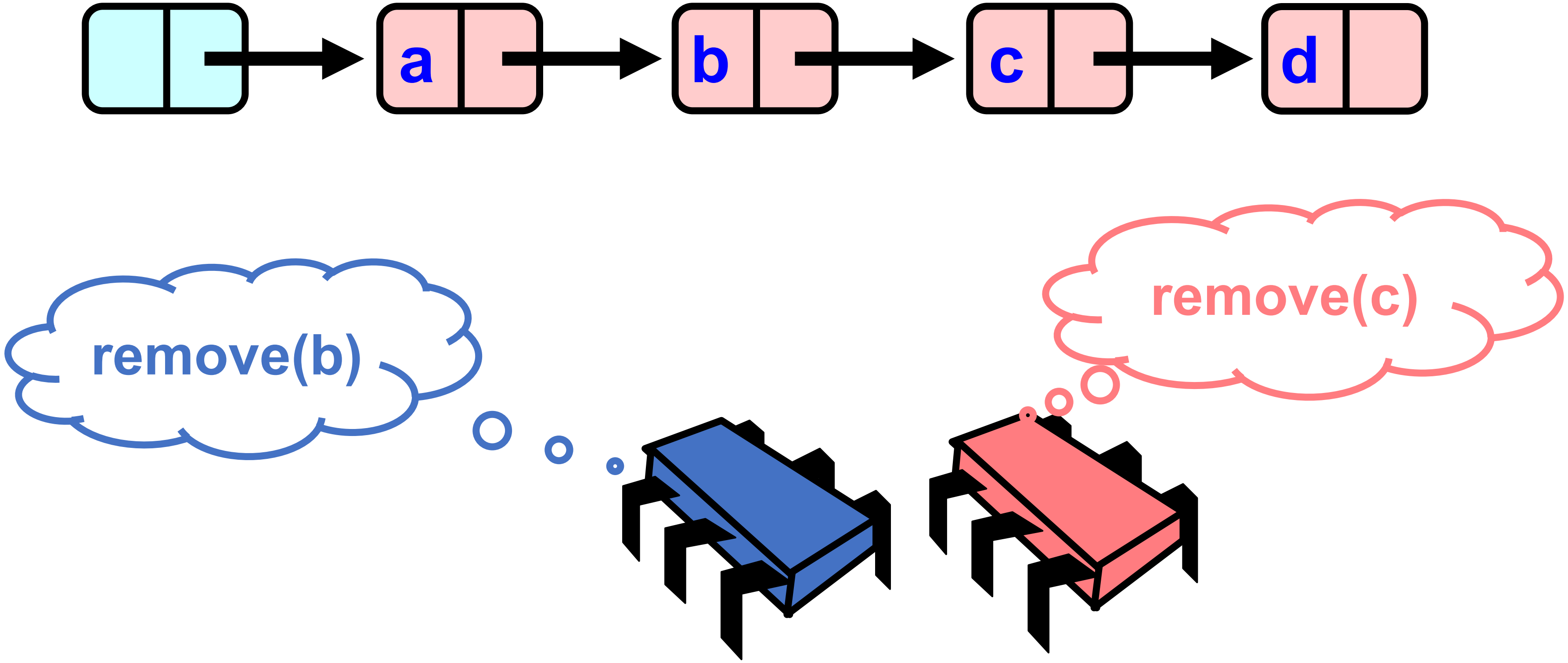


Removing a Node

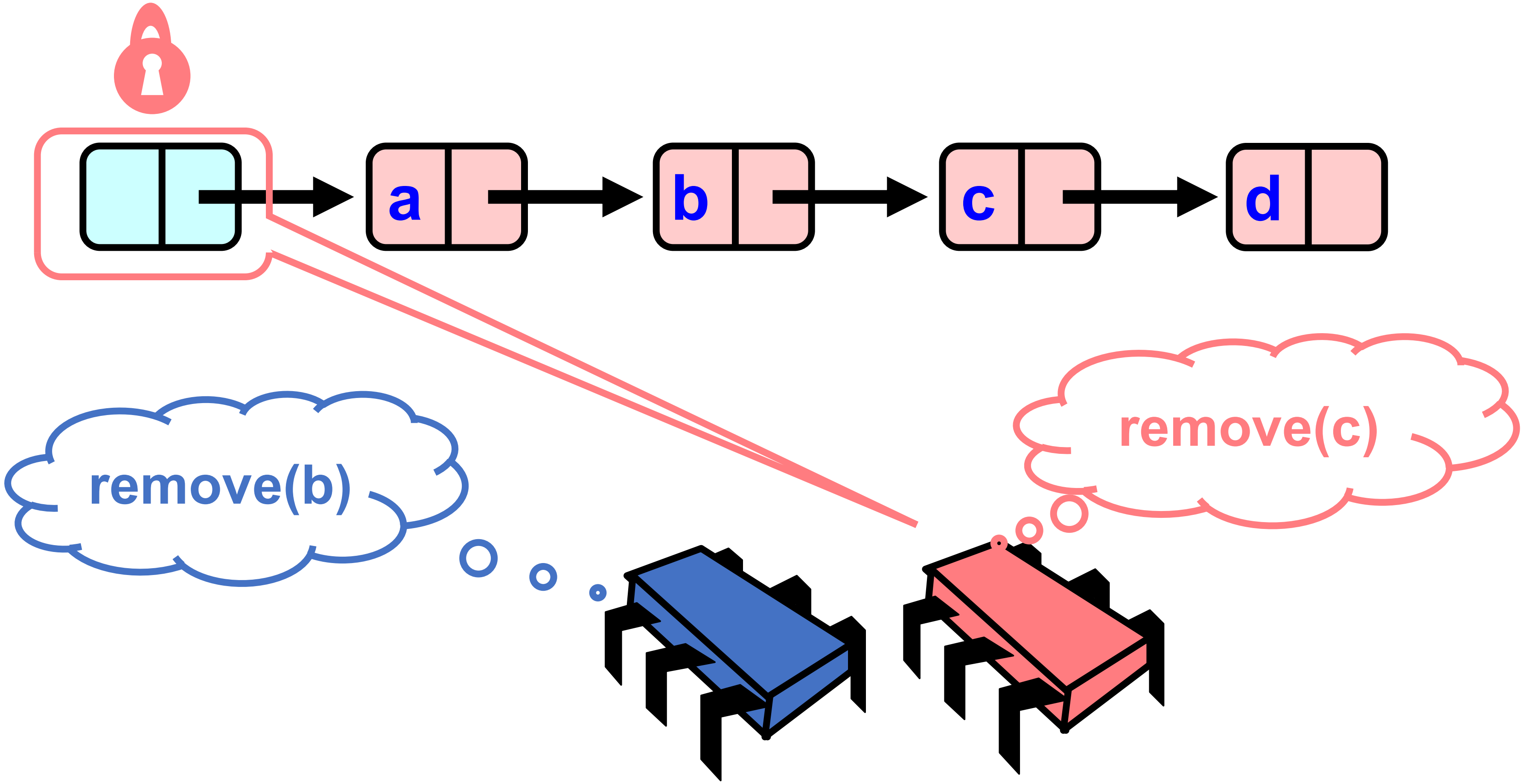


Why hold 2 locks?

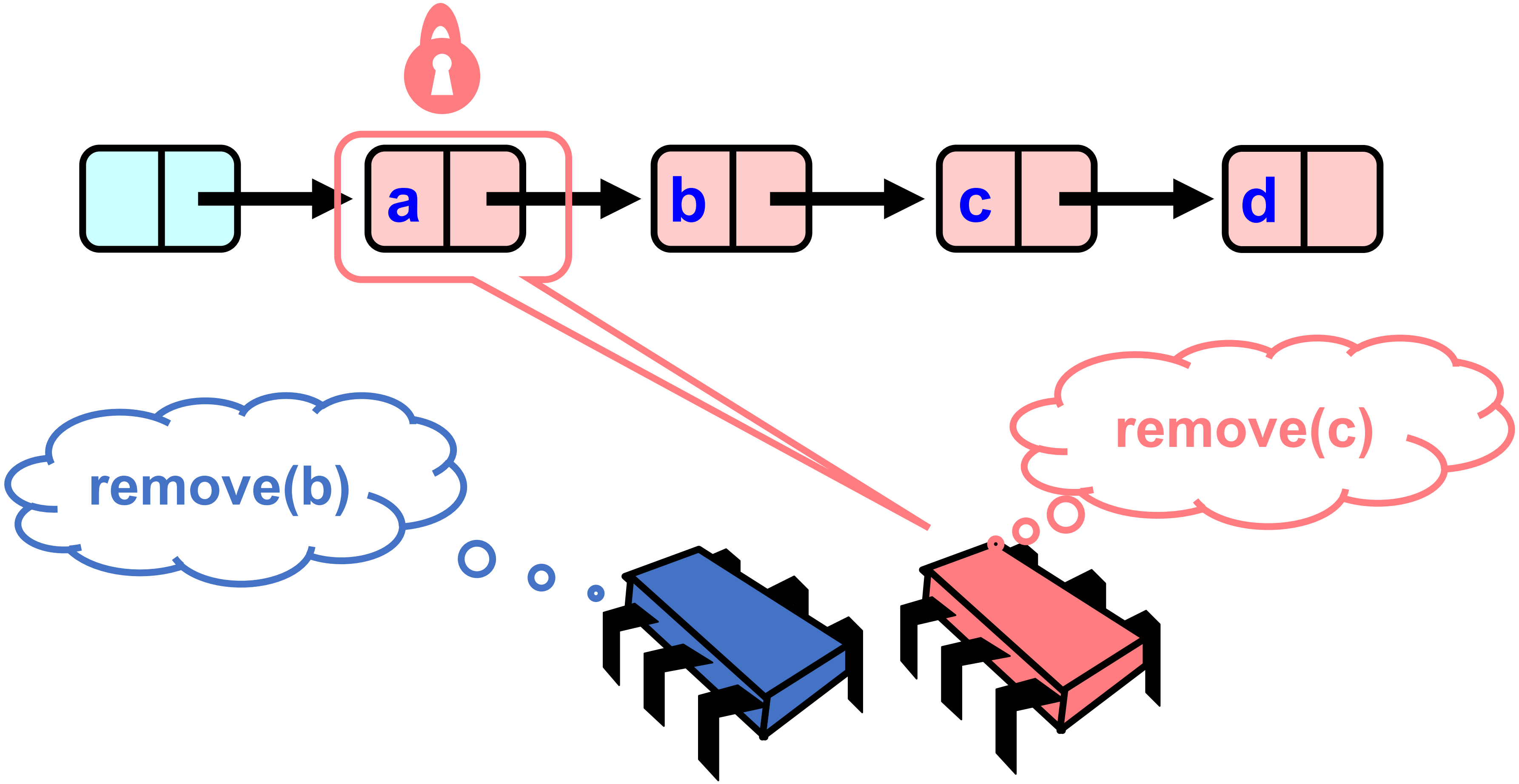
Concurrent Removes



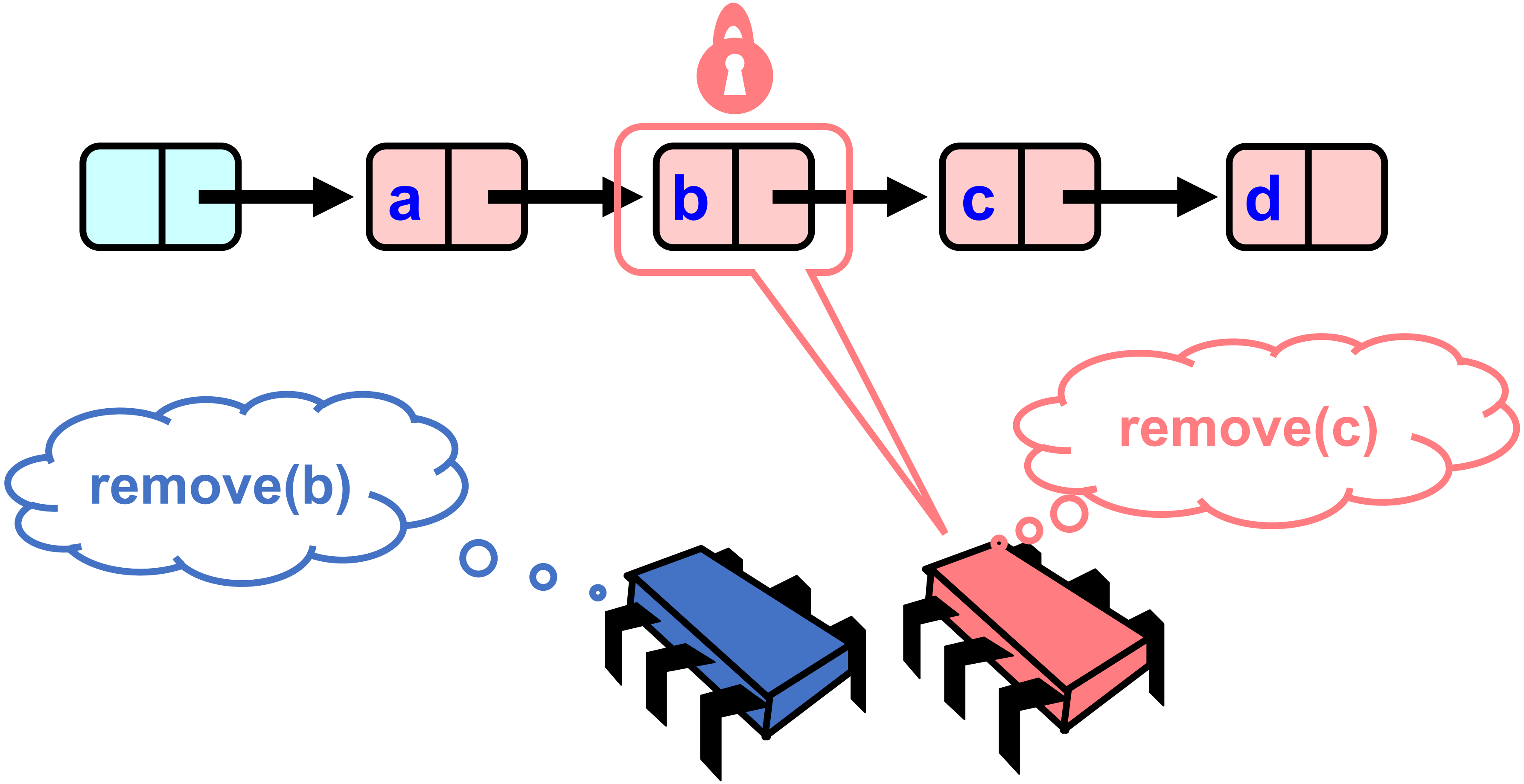
Concurrent Removes



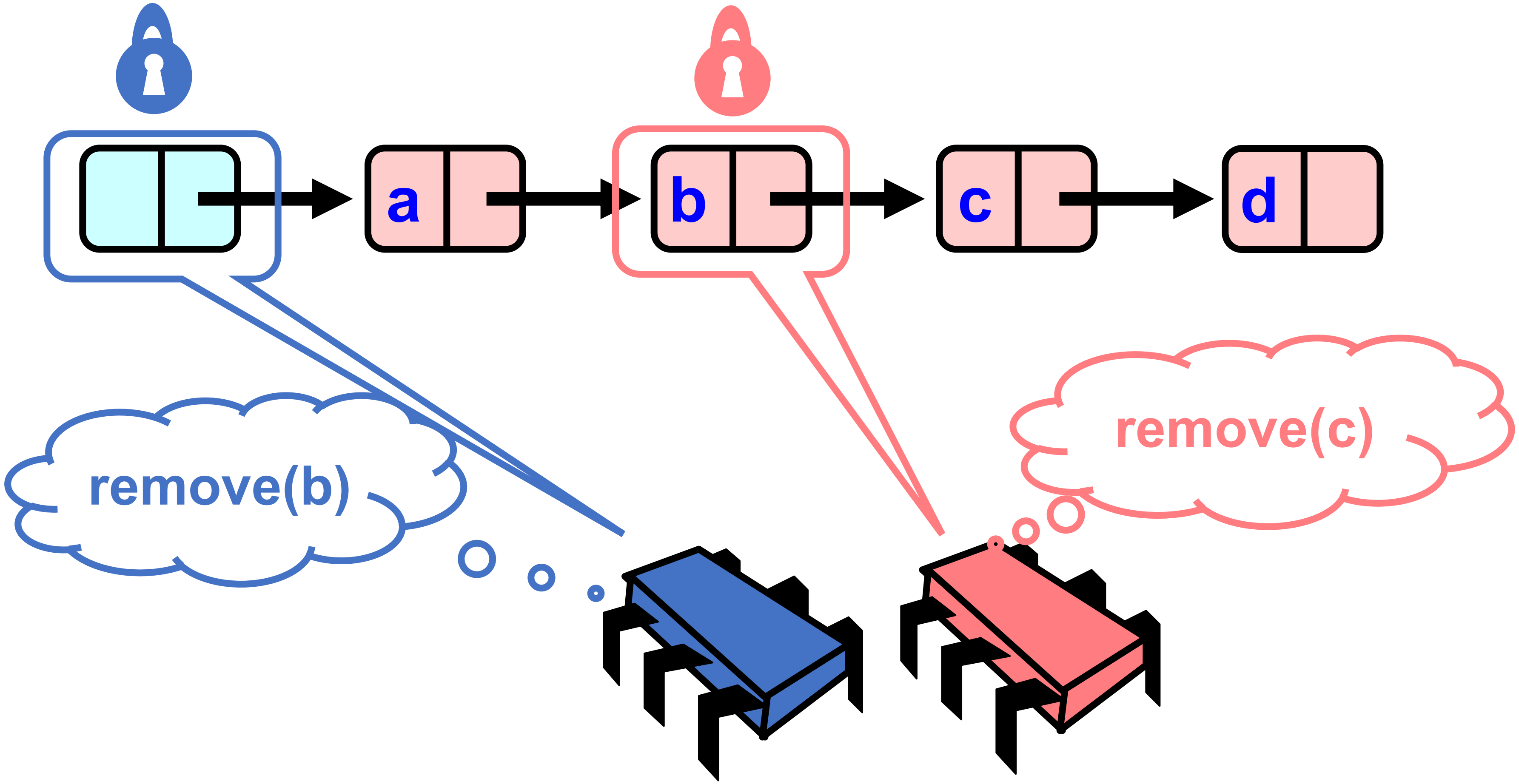
Concurrent Removes



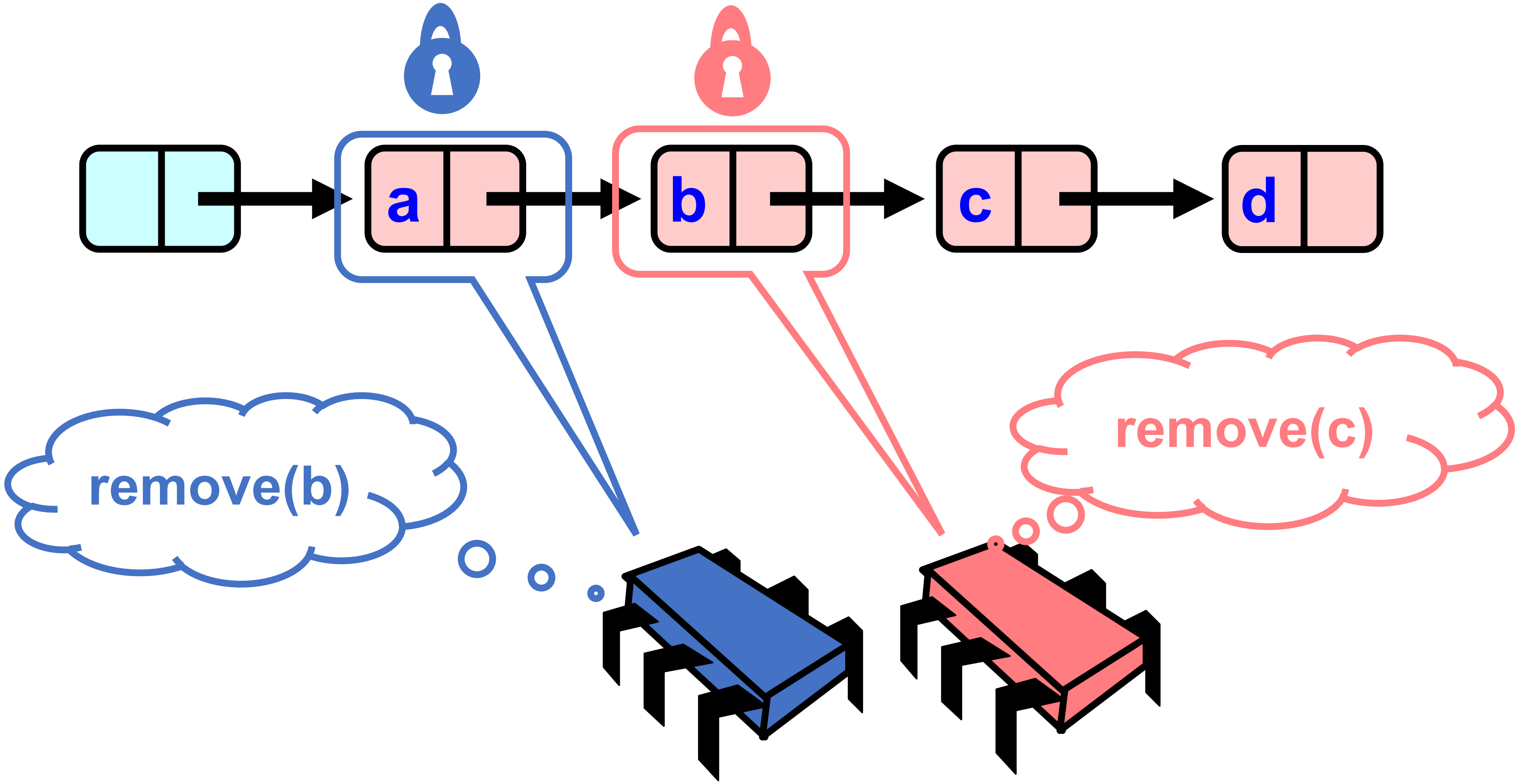
Concurrent Removes



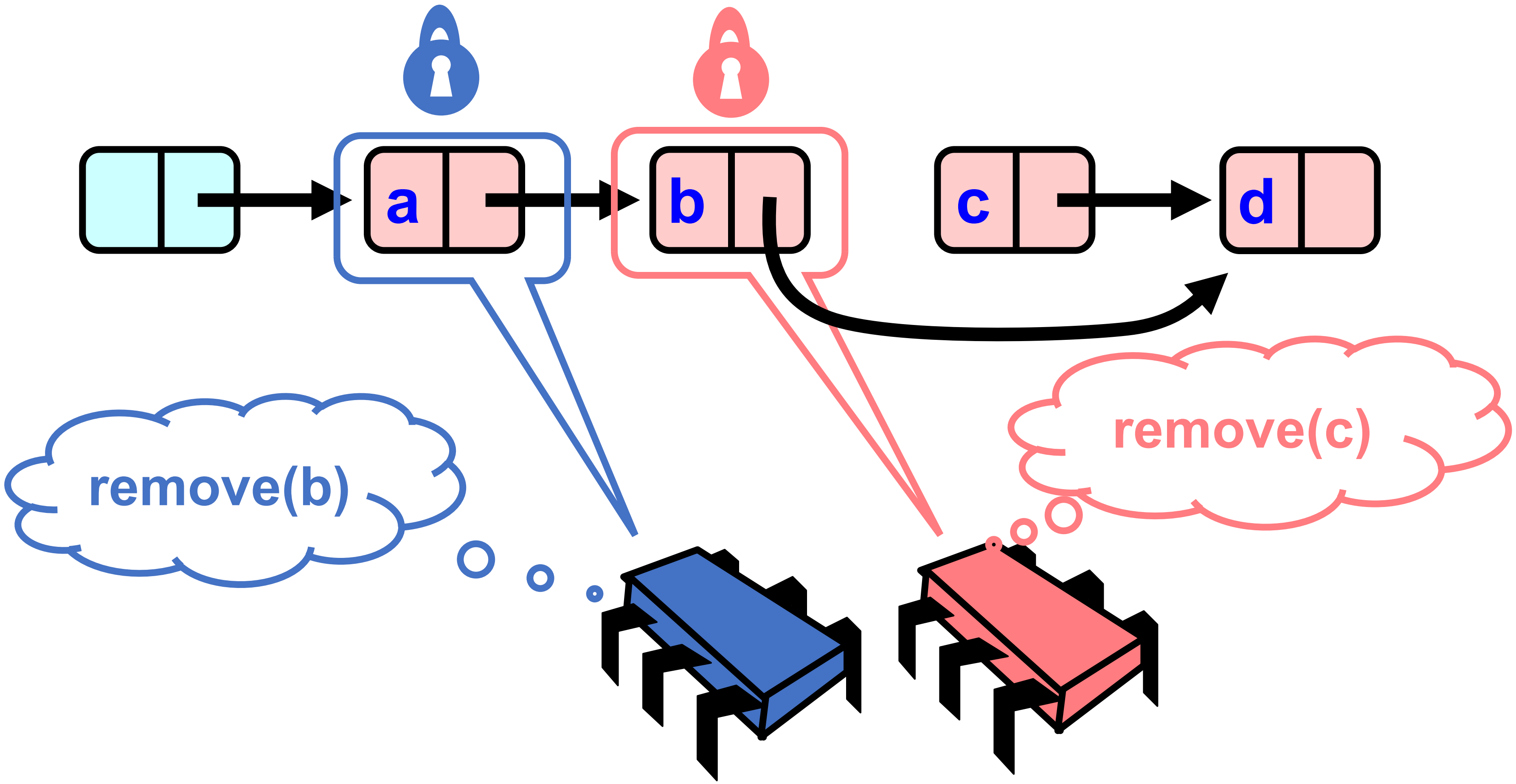
Concurrent Removes



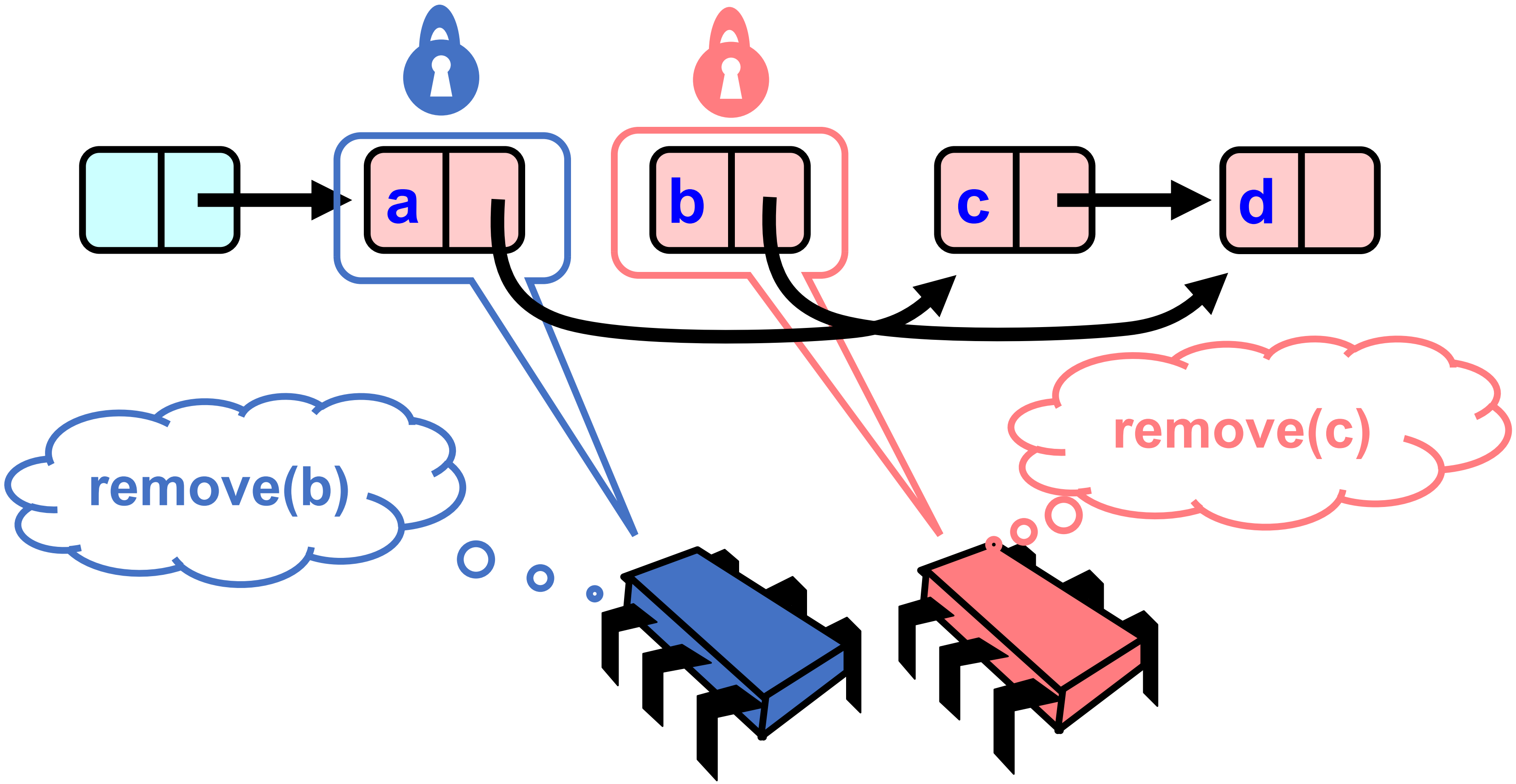
Concurrent Removes



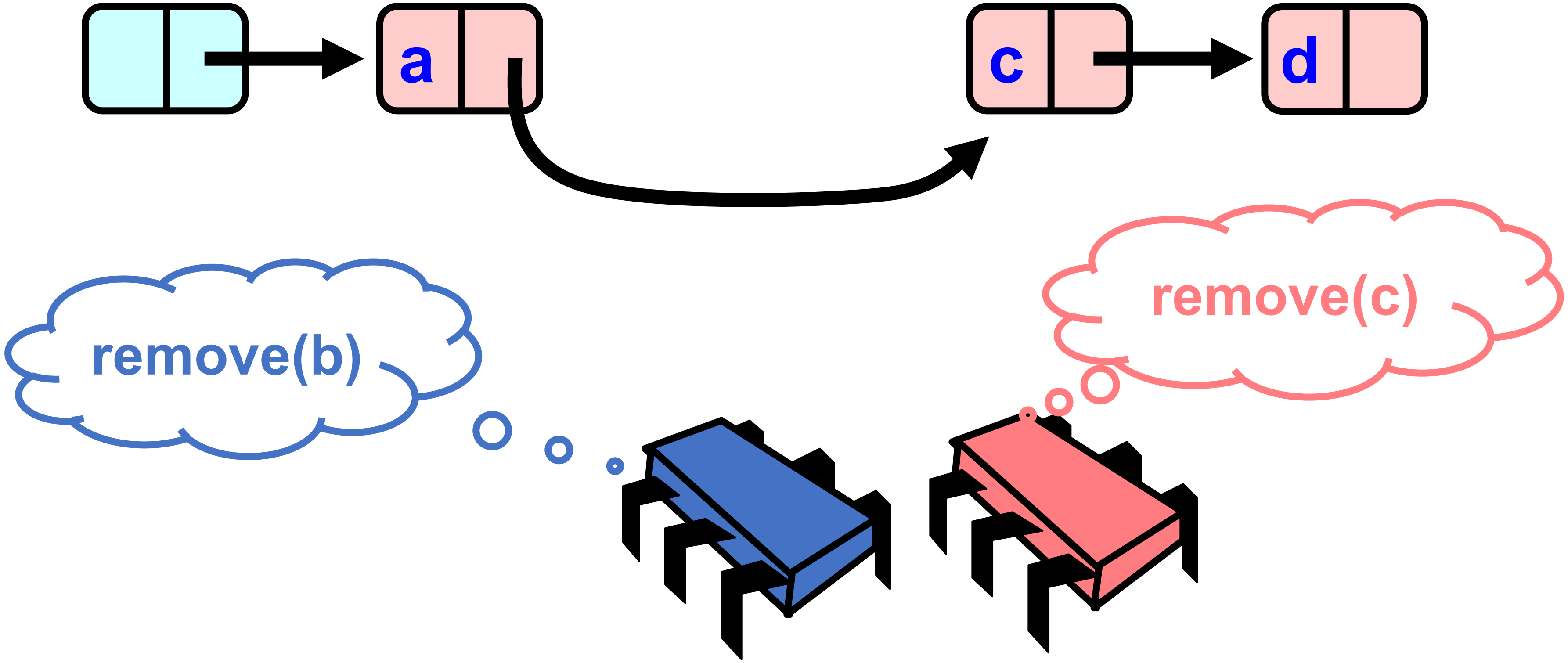
Concurrent Removes



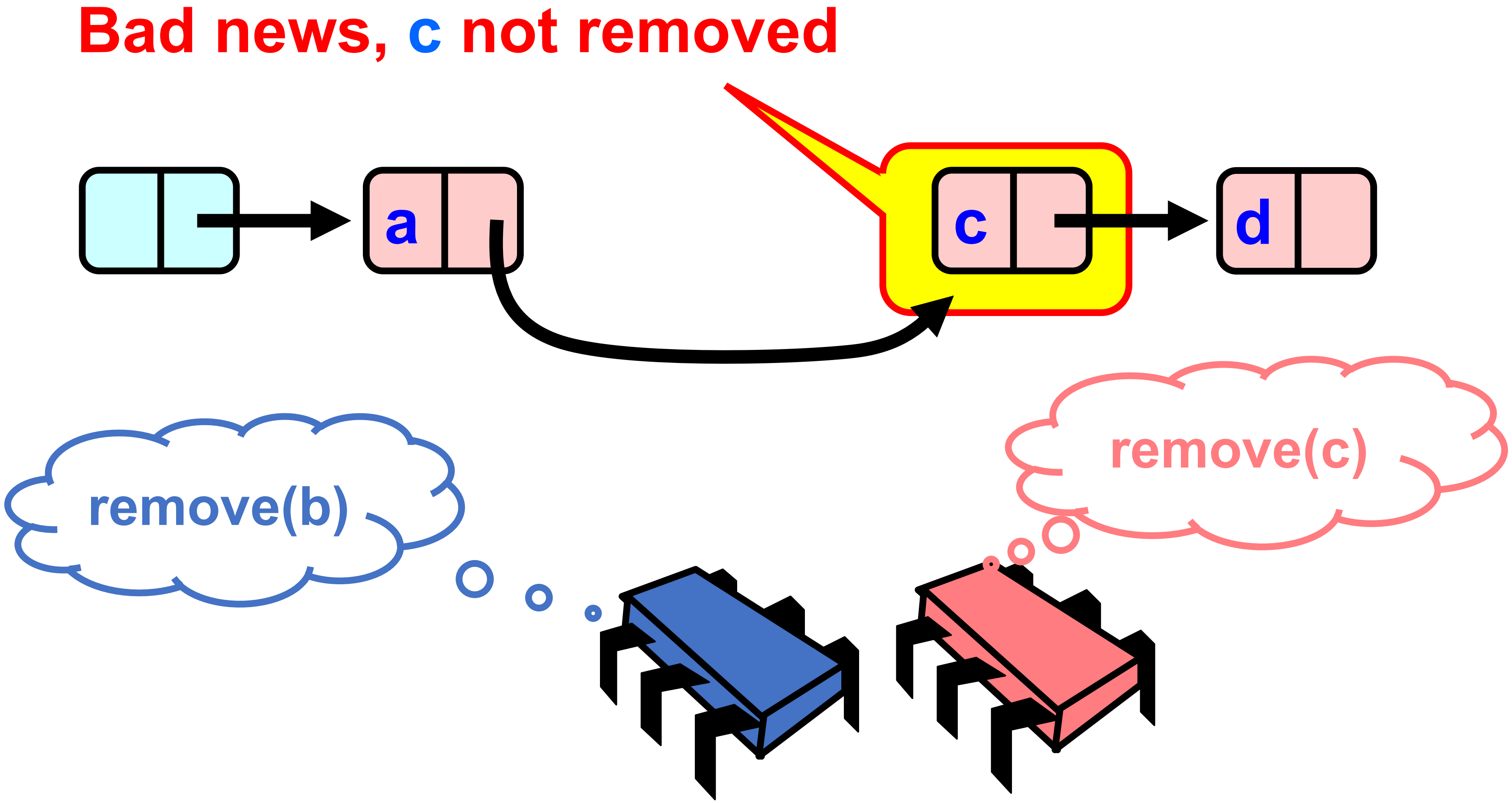
Concurrent Removes



Uh, Oh

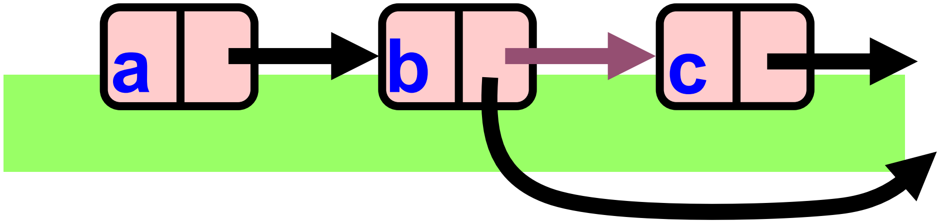


Uh, Oh

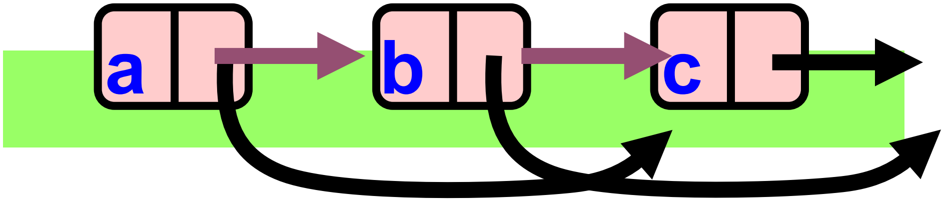


Problem

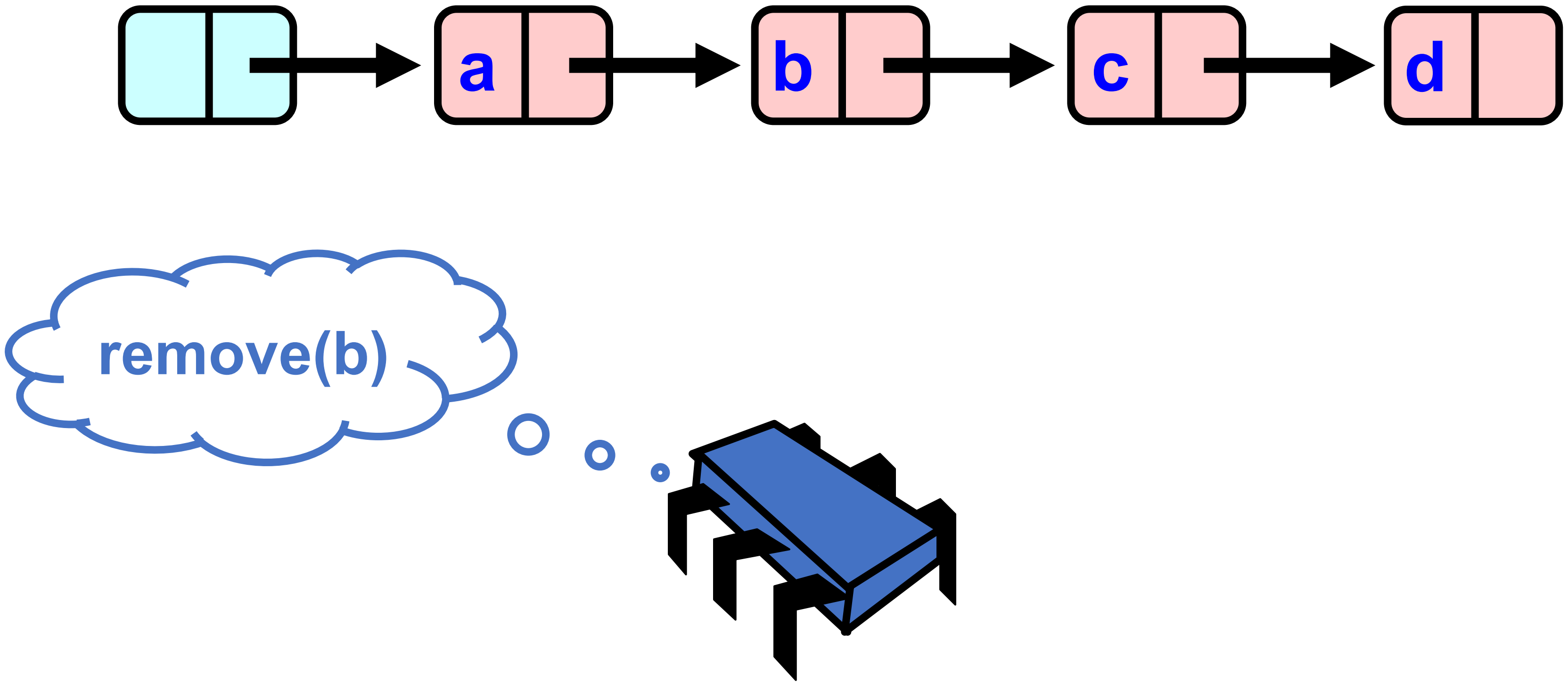
- To delete node c
  - Swing node b's next field to d



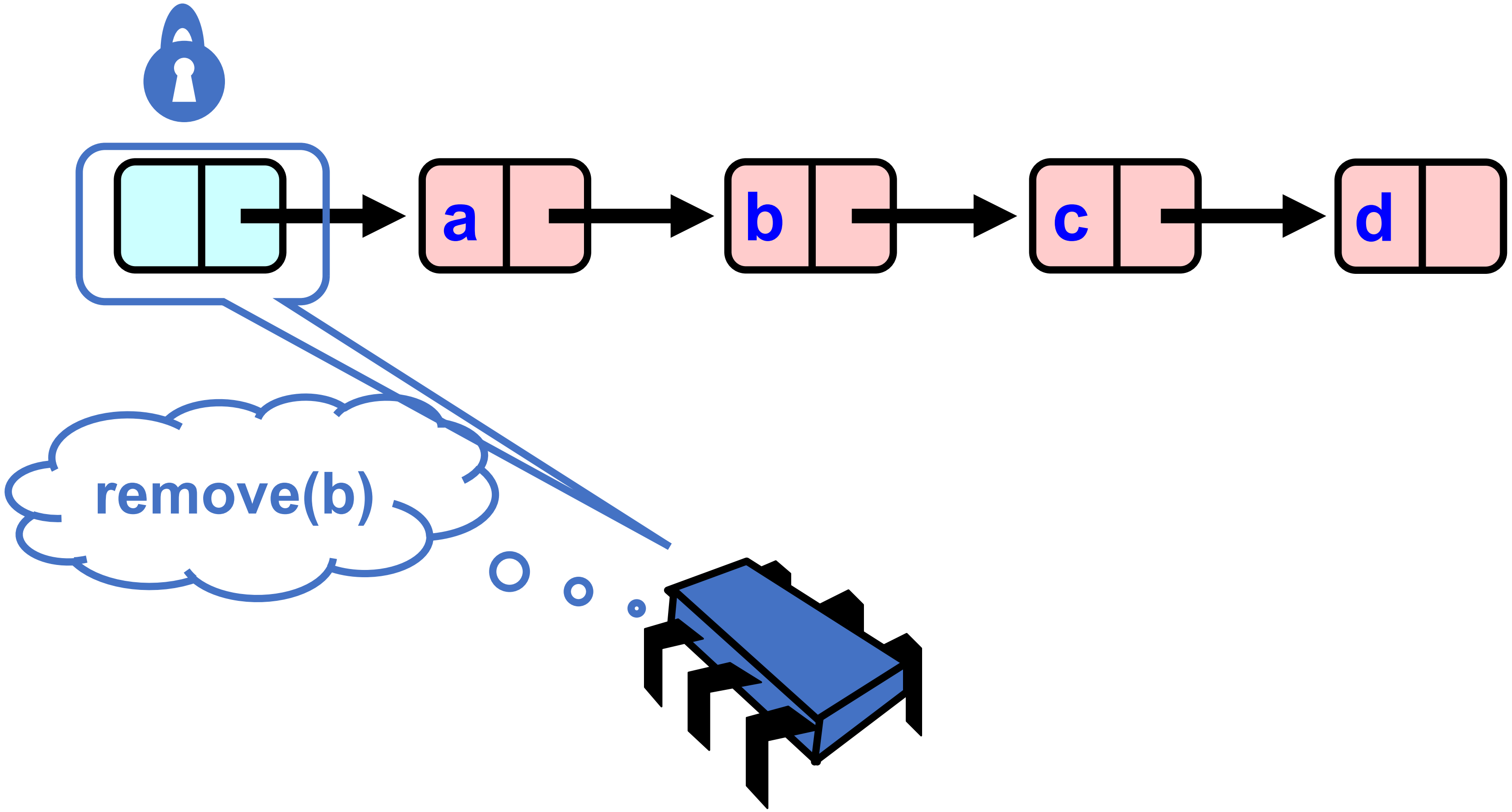
- Problem is,
  - ***Data conflict:***
  - Someone deleting b concurrently could direct a pointer to C



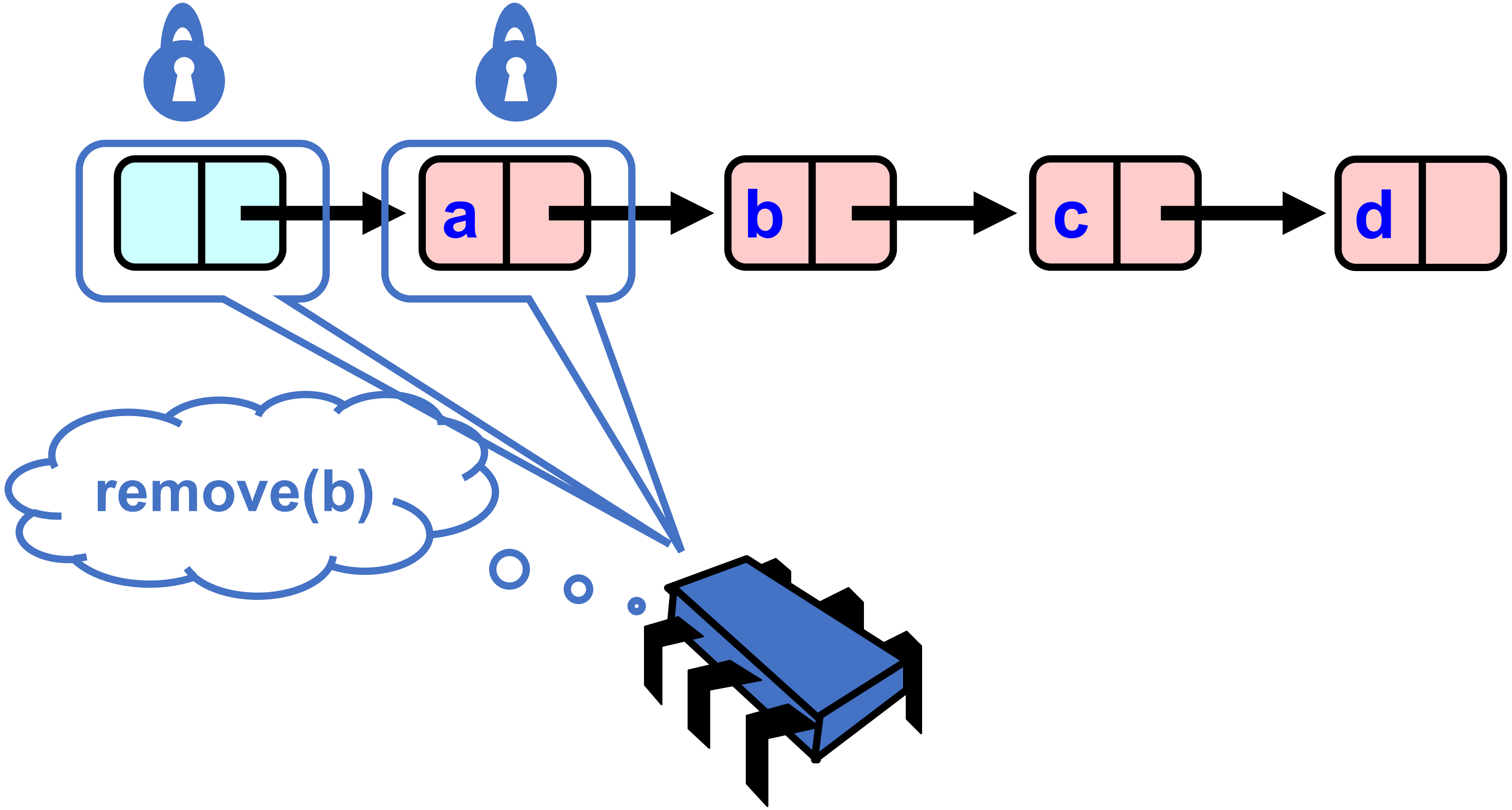
Hand-Over-Hand Again



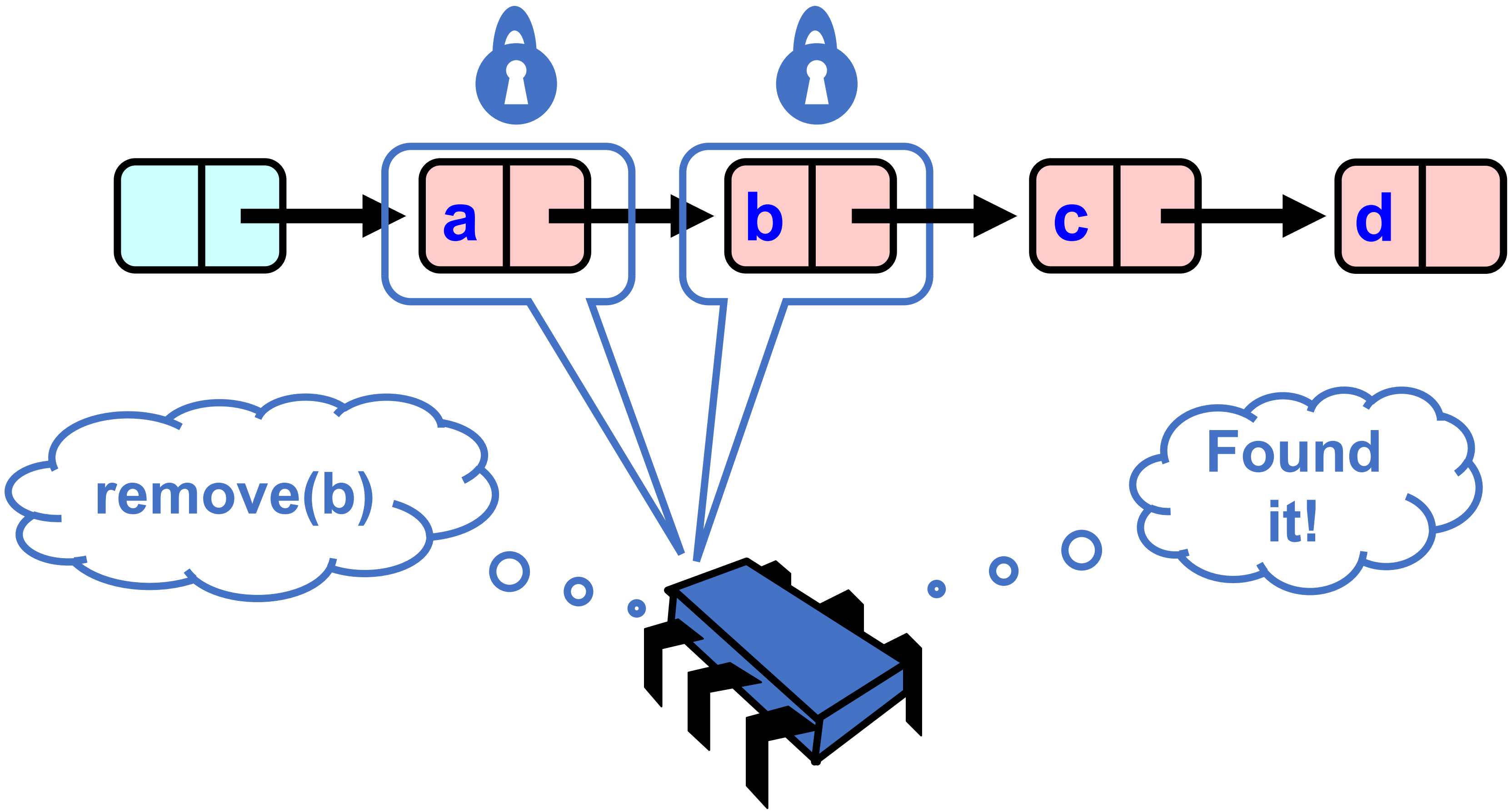
Hand-Over-Hand Again



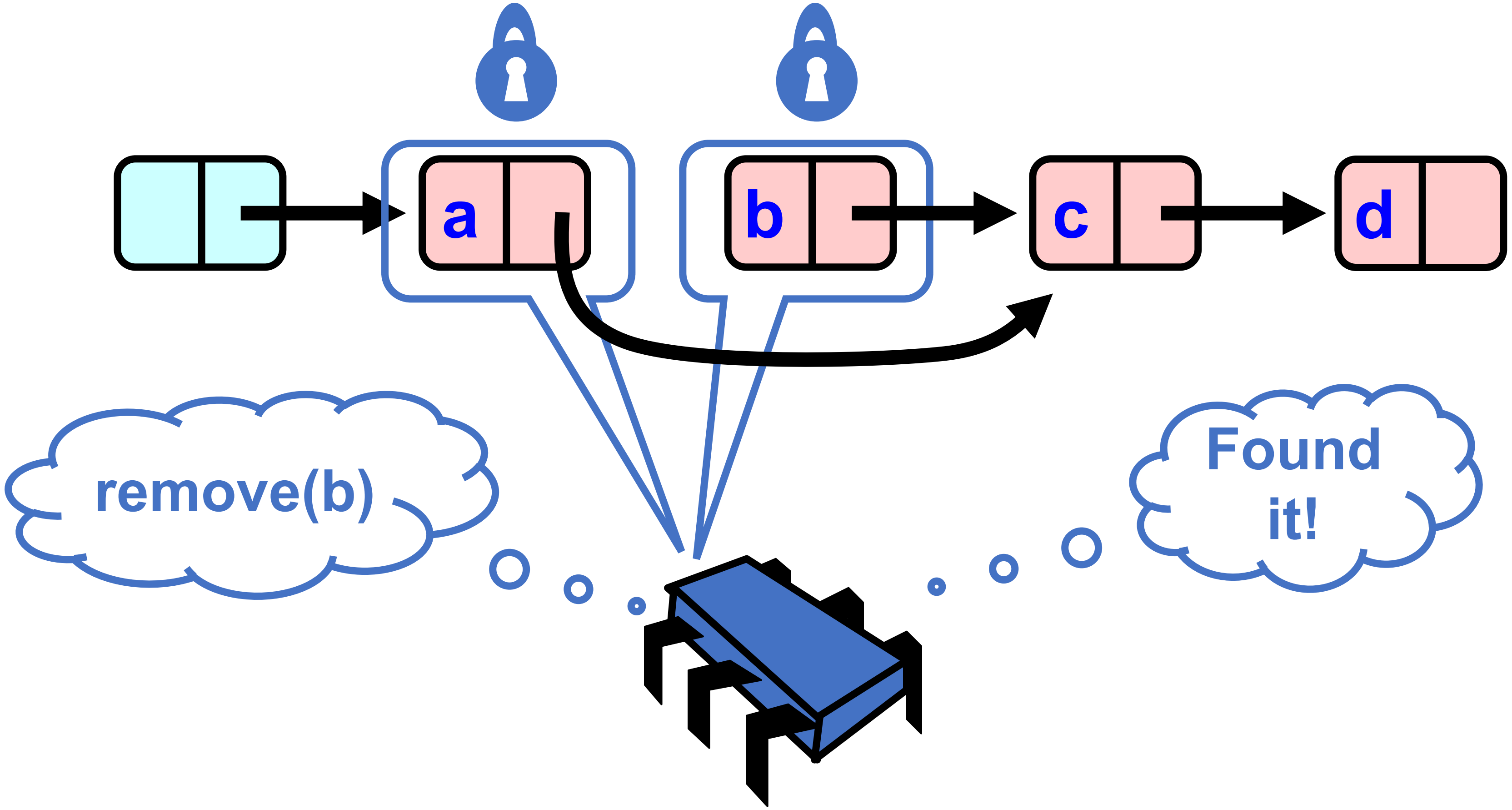
Hand-Over-Hand Again



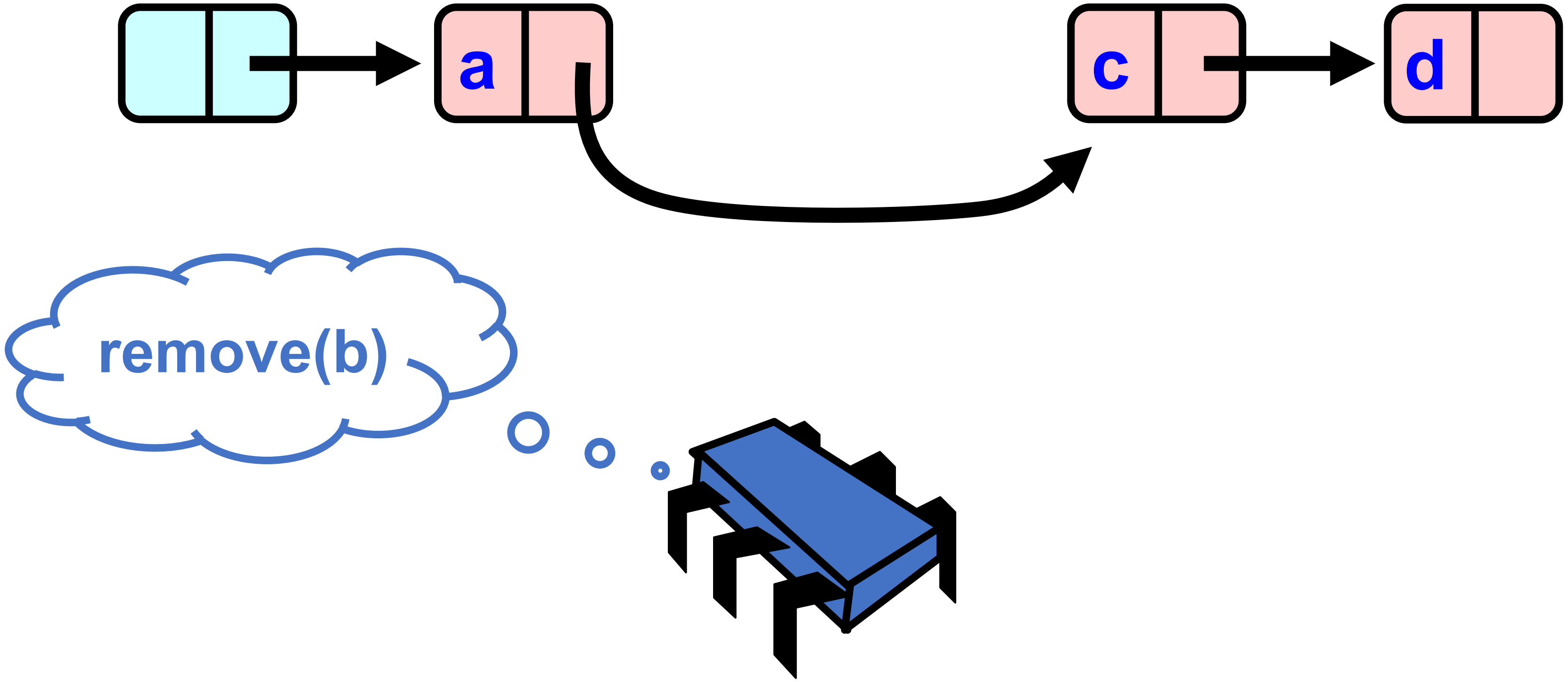
Hand-Over-Hand Again



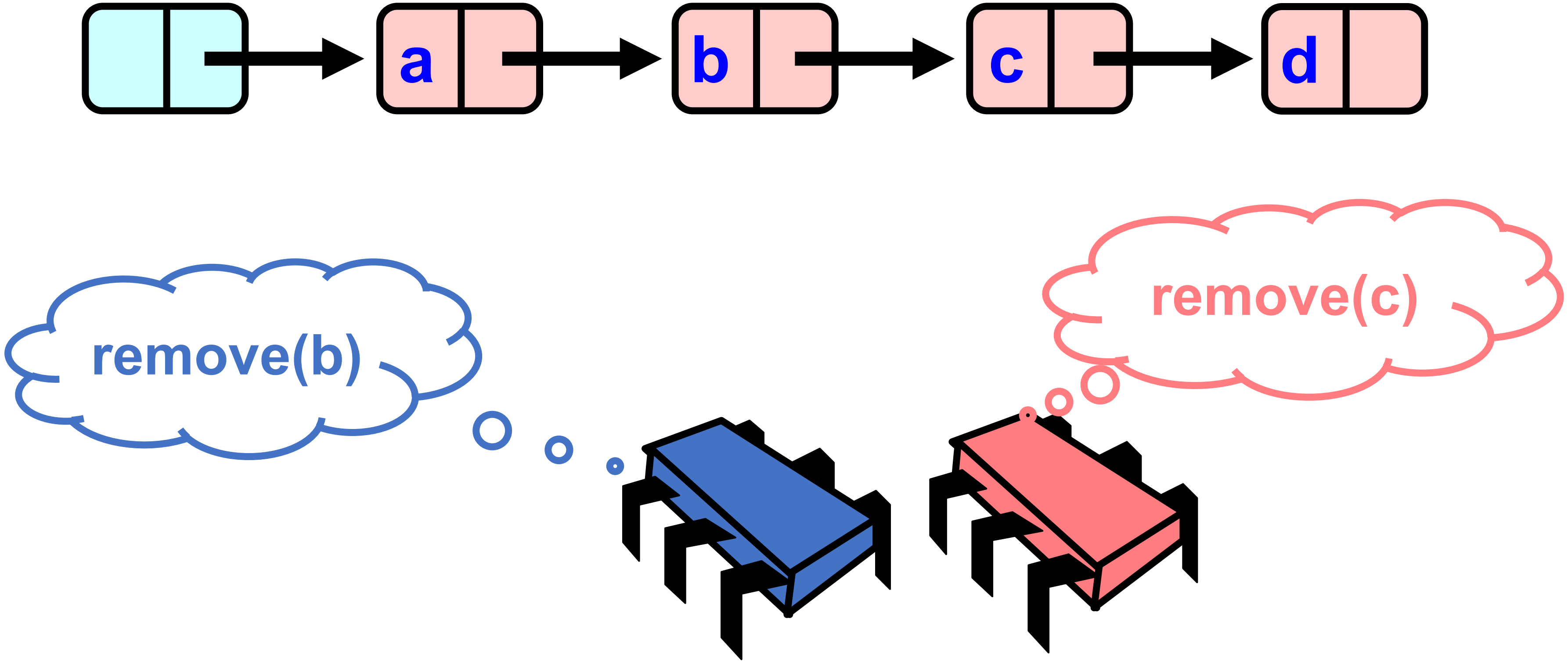
Hand-Over-Hand Again



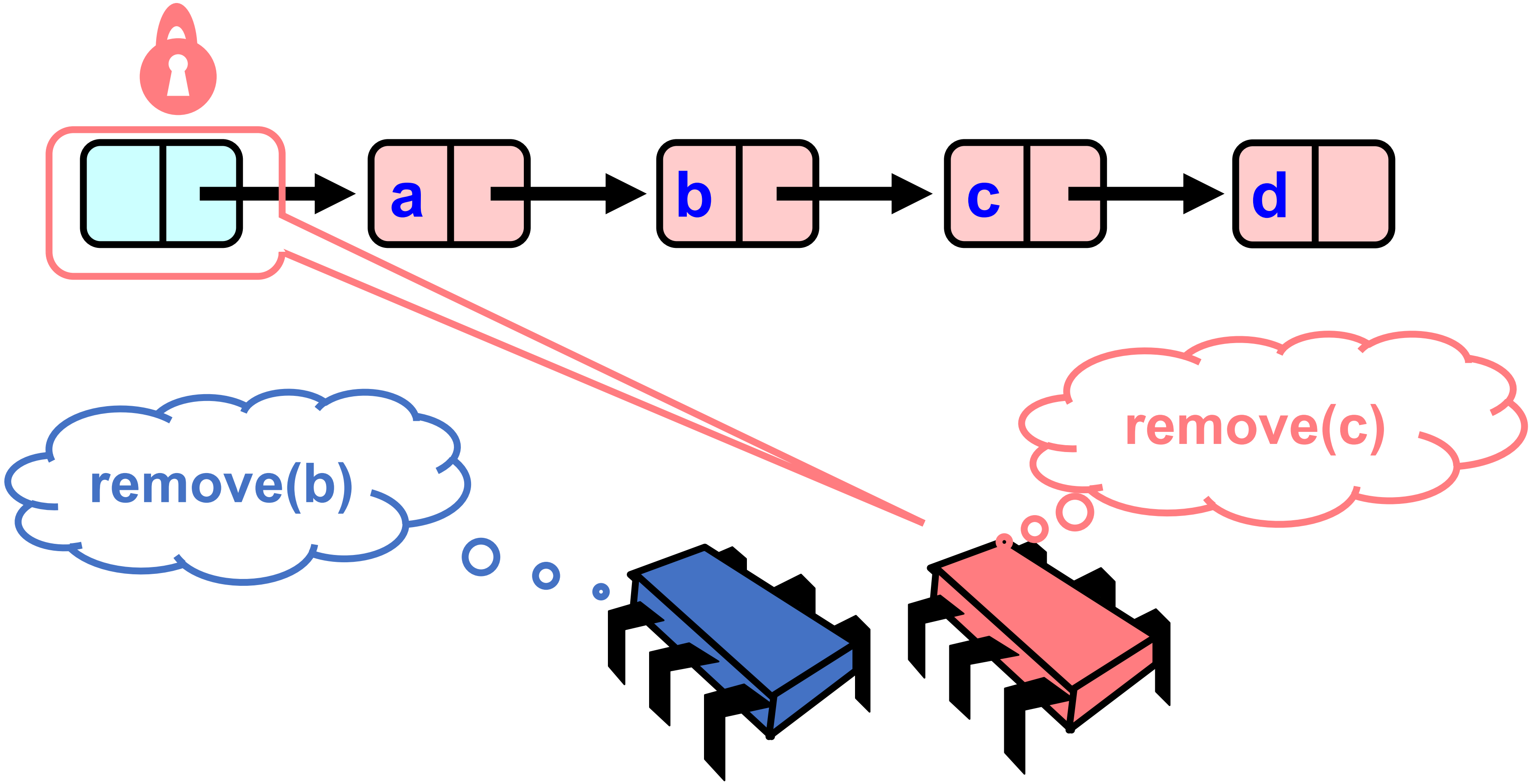
Hand-Over-Hand Again



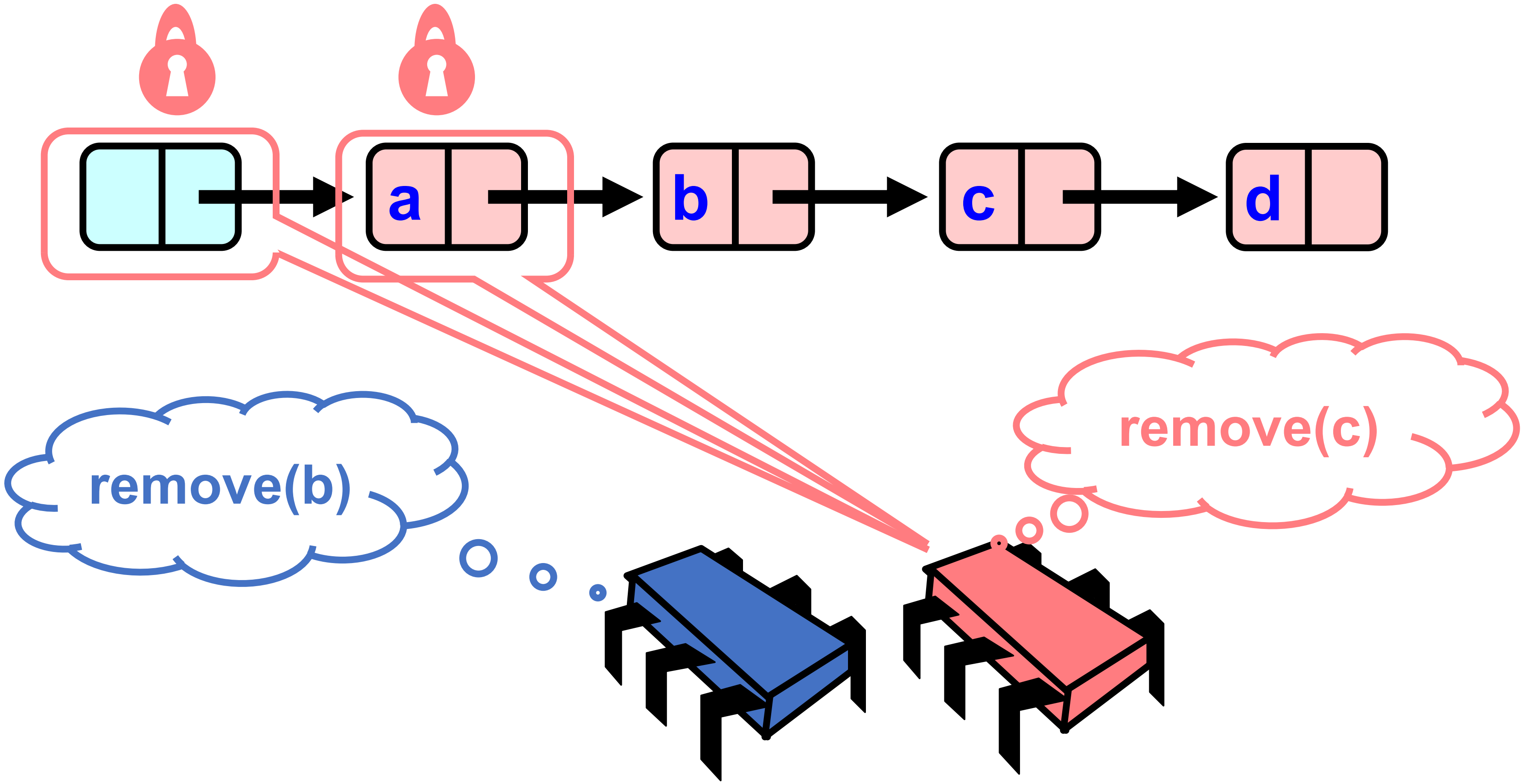
Removing a Node



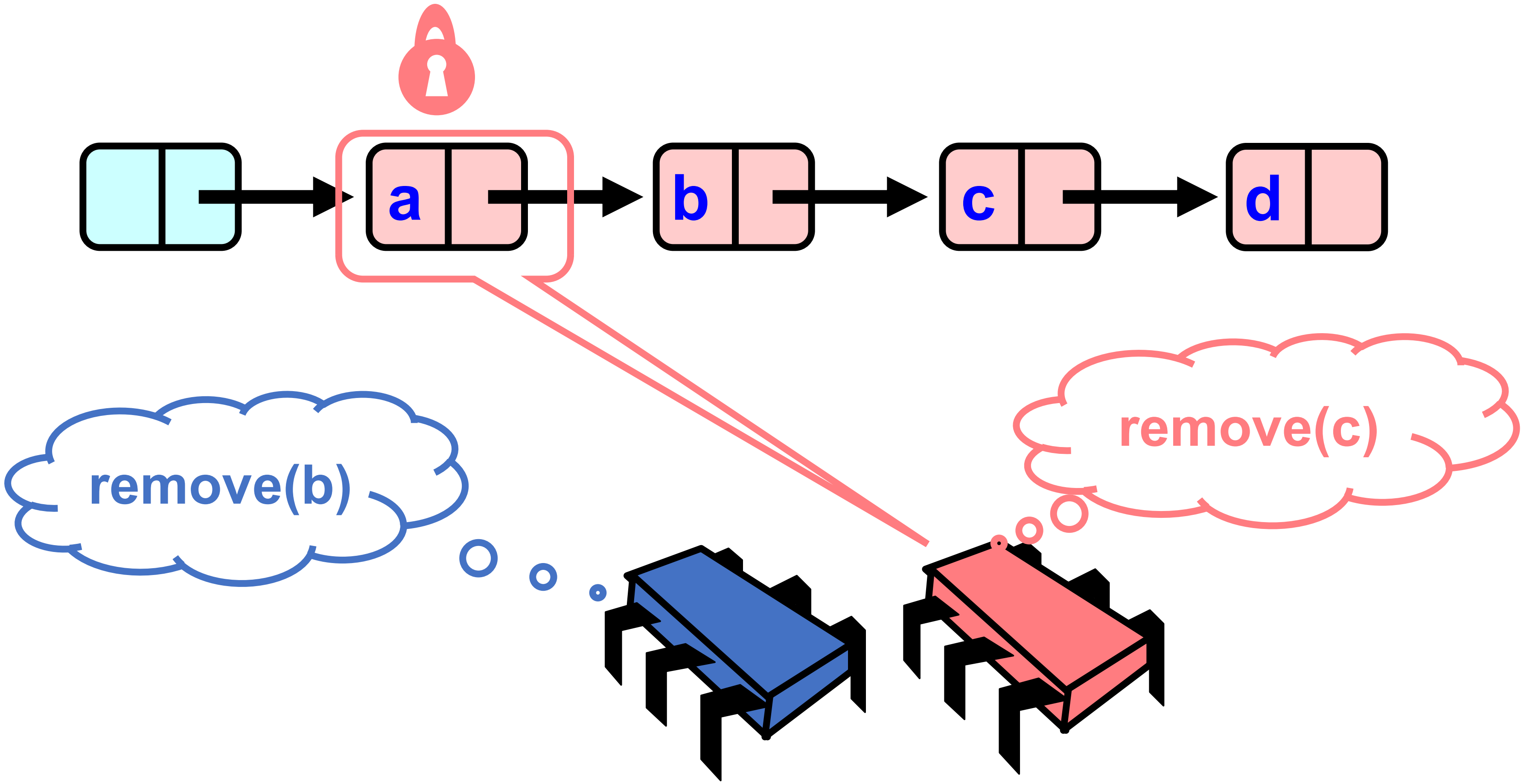
Removing a Node



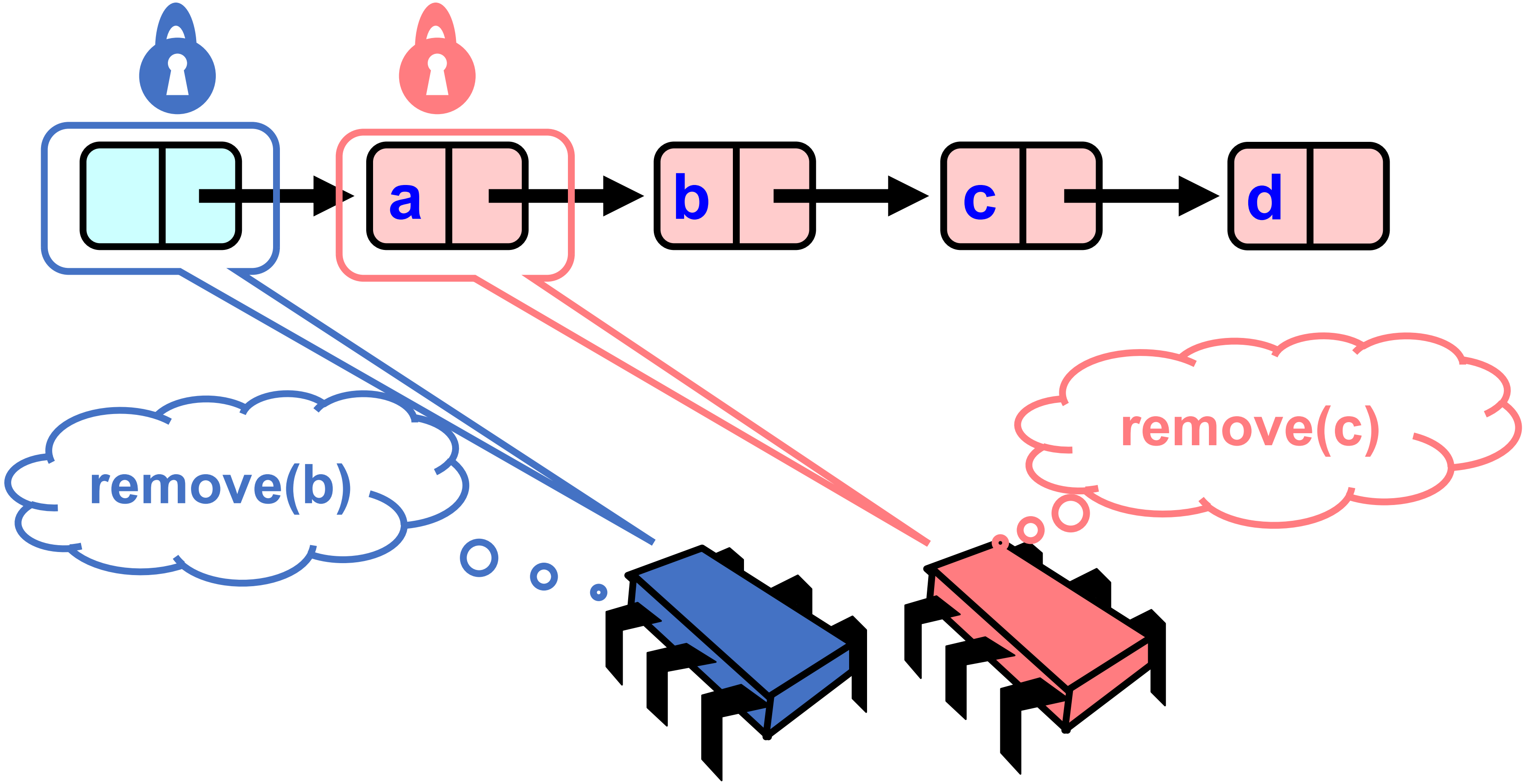
Removing a Node



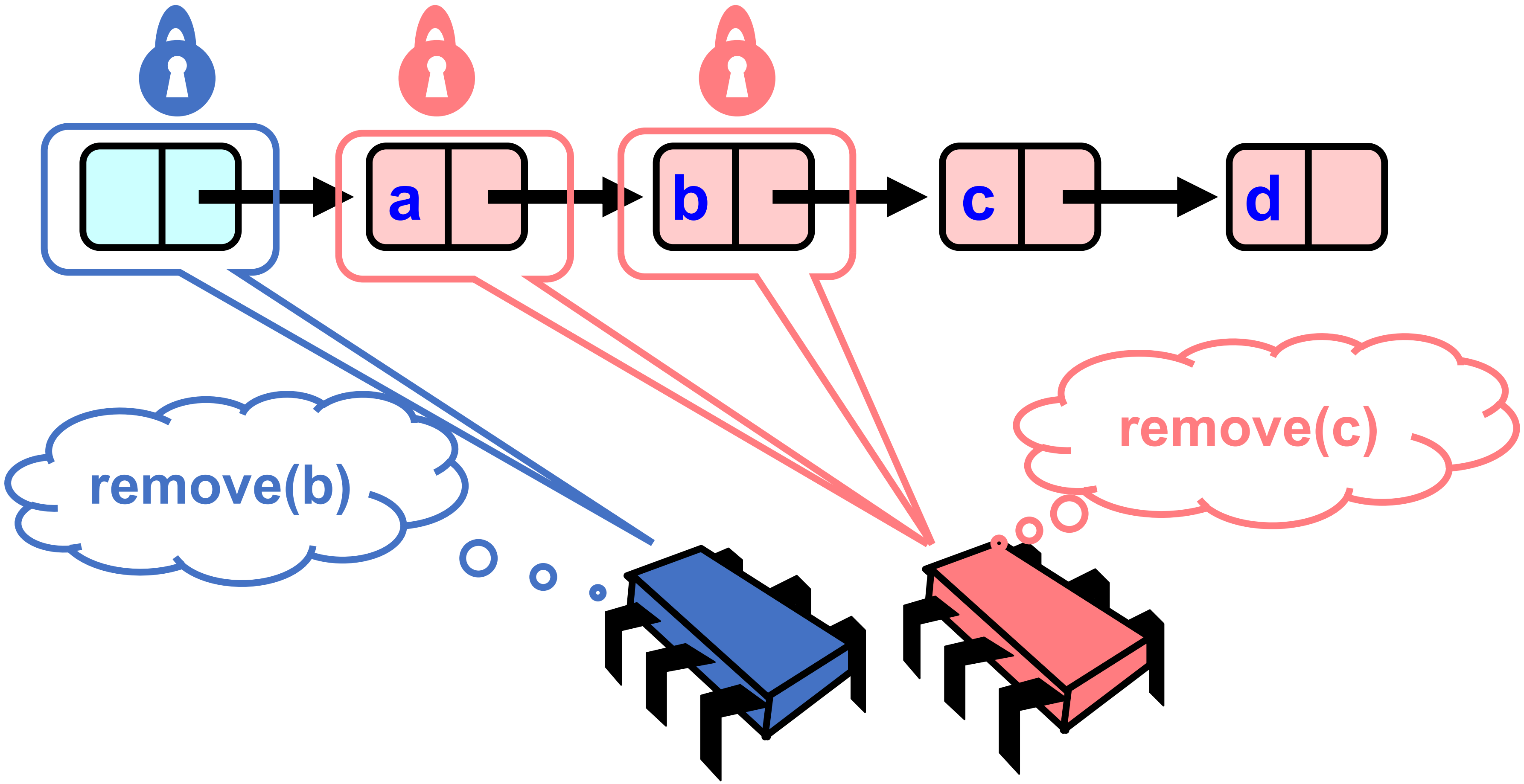
Removing a Node



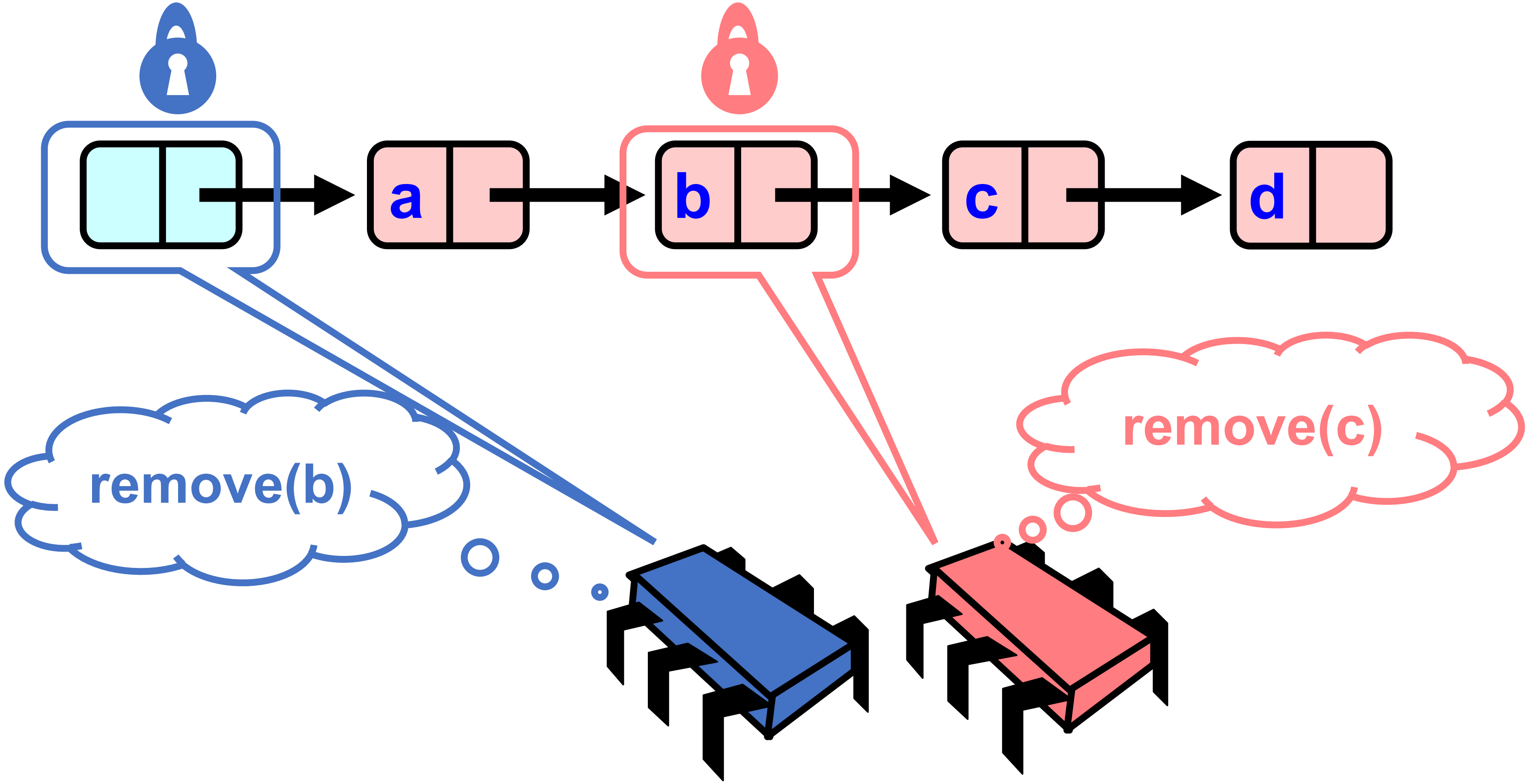
Removing a Node



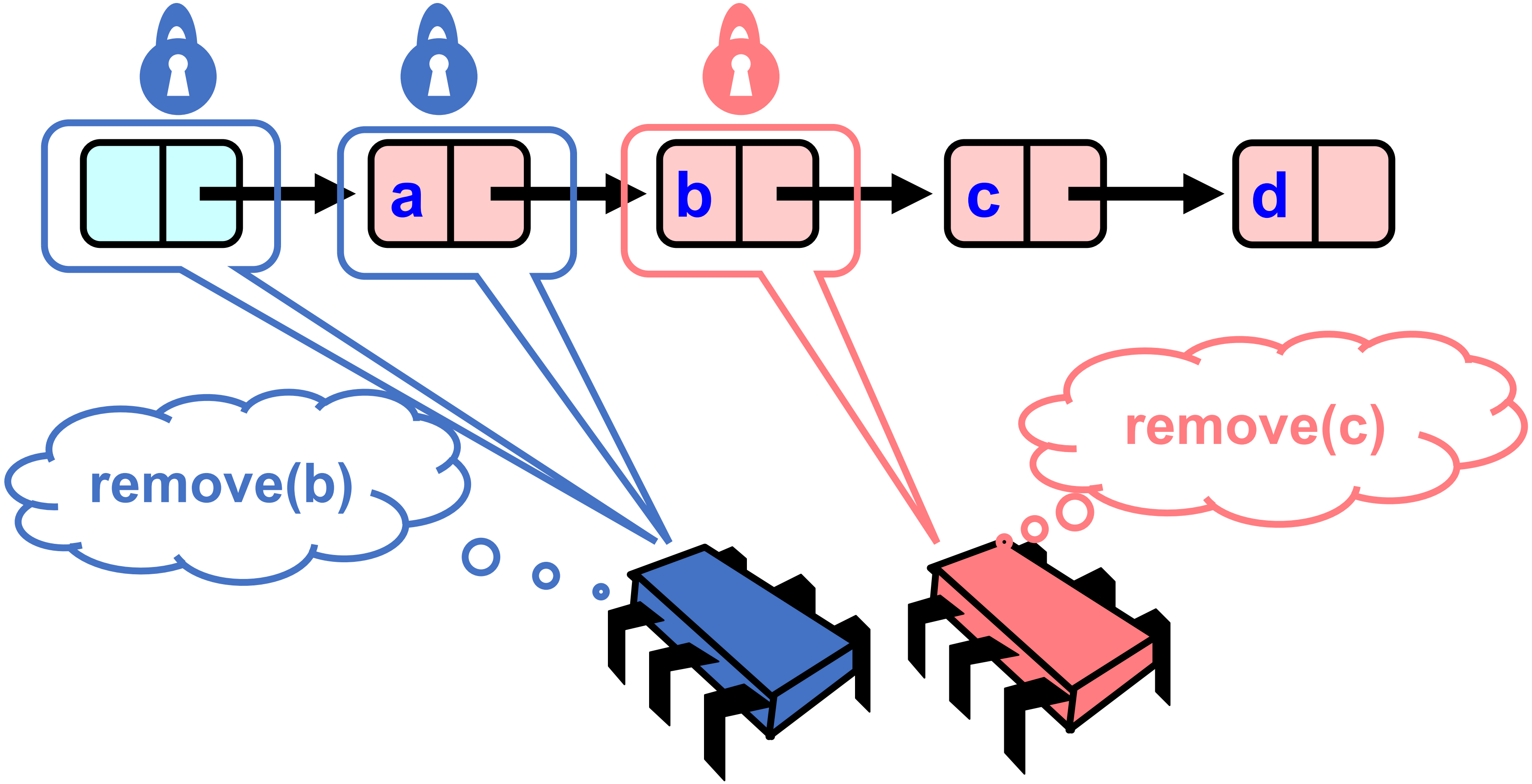
Removing a Node



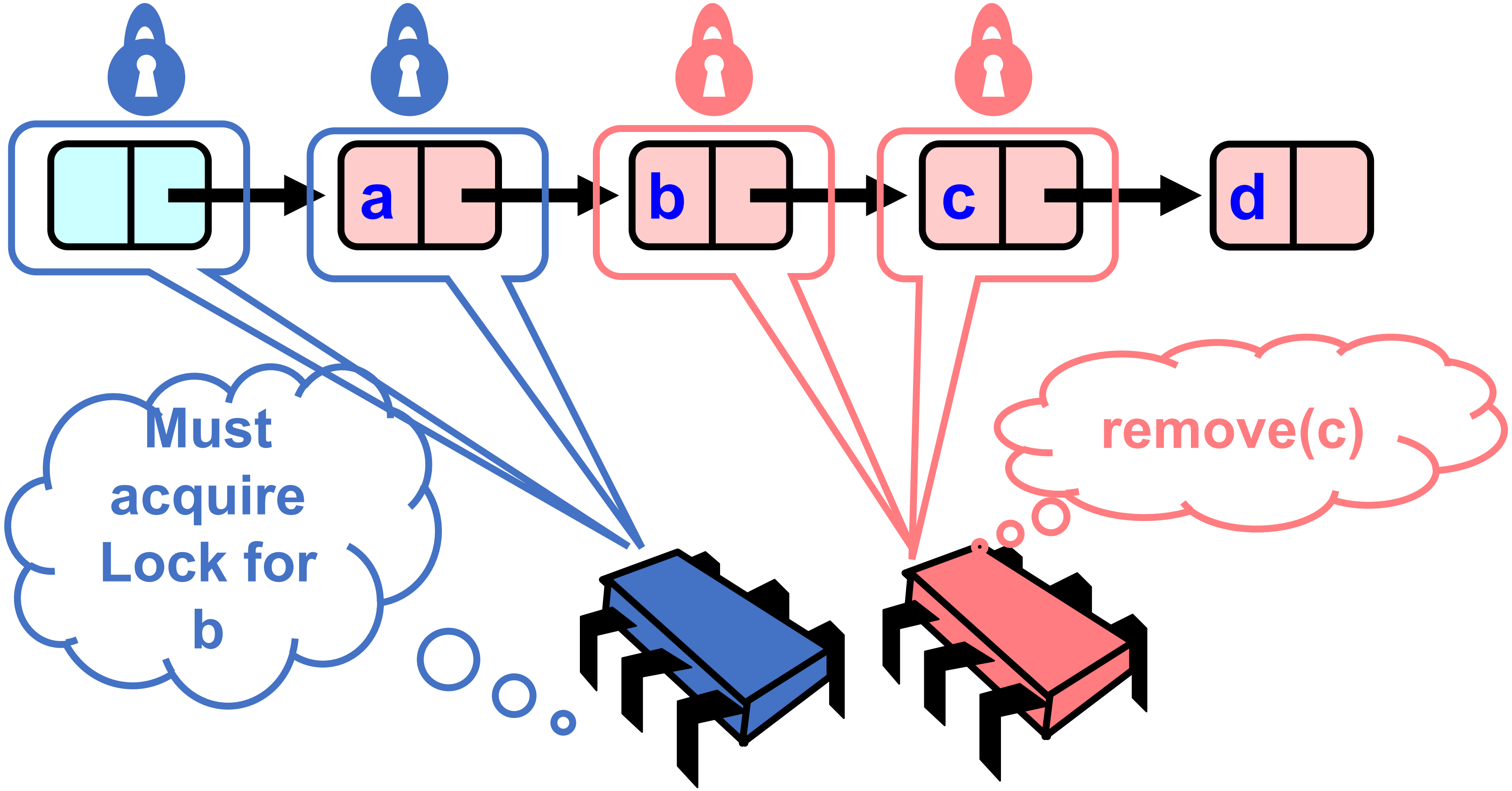
Removing a Node



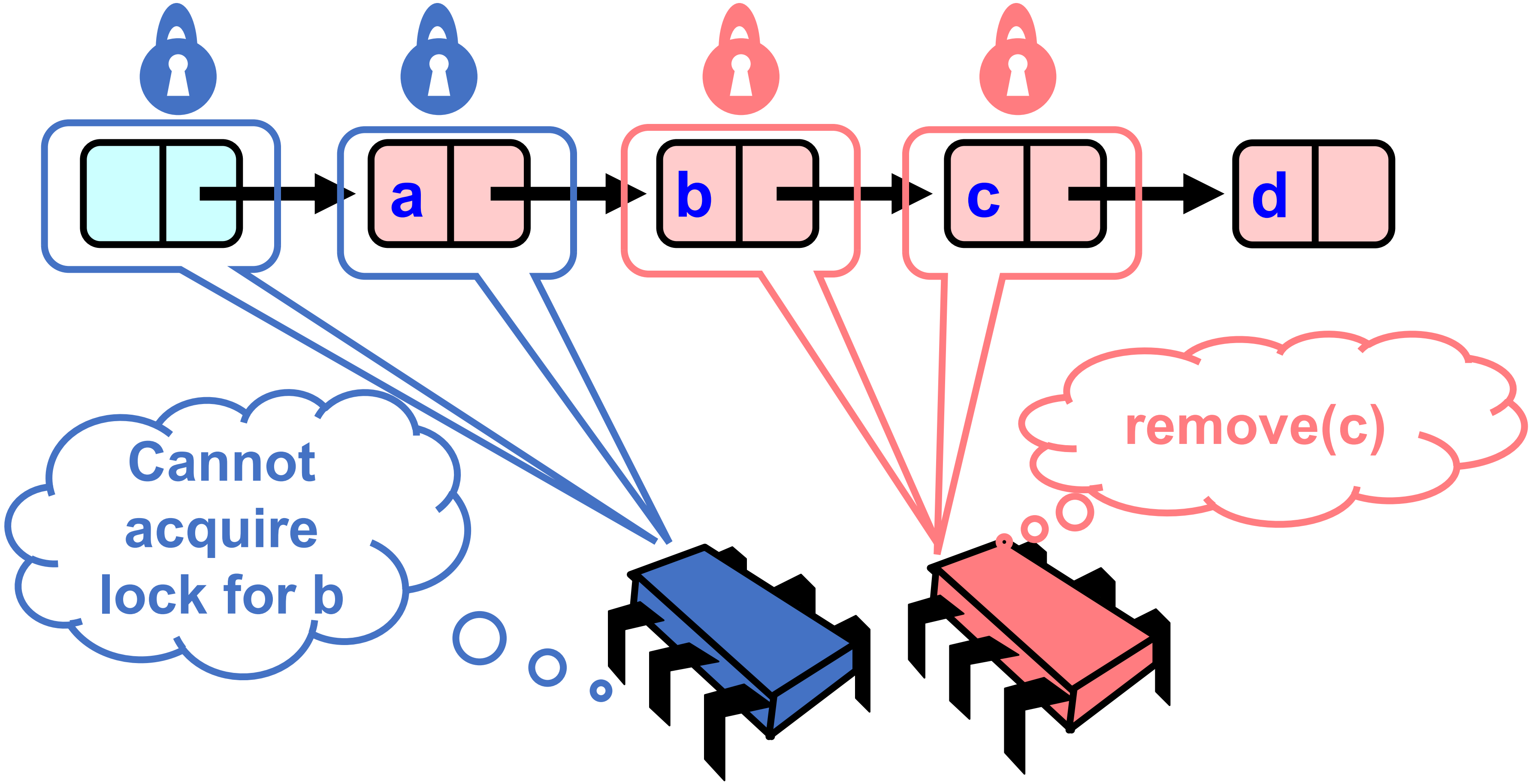
Removing a Node



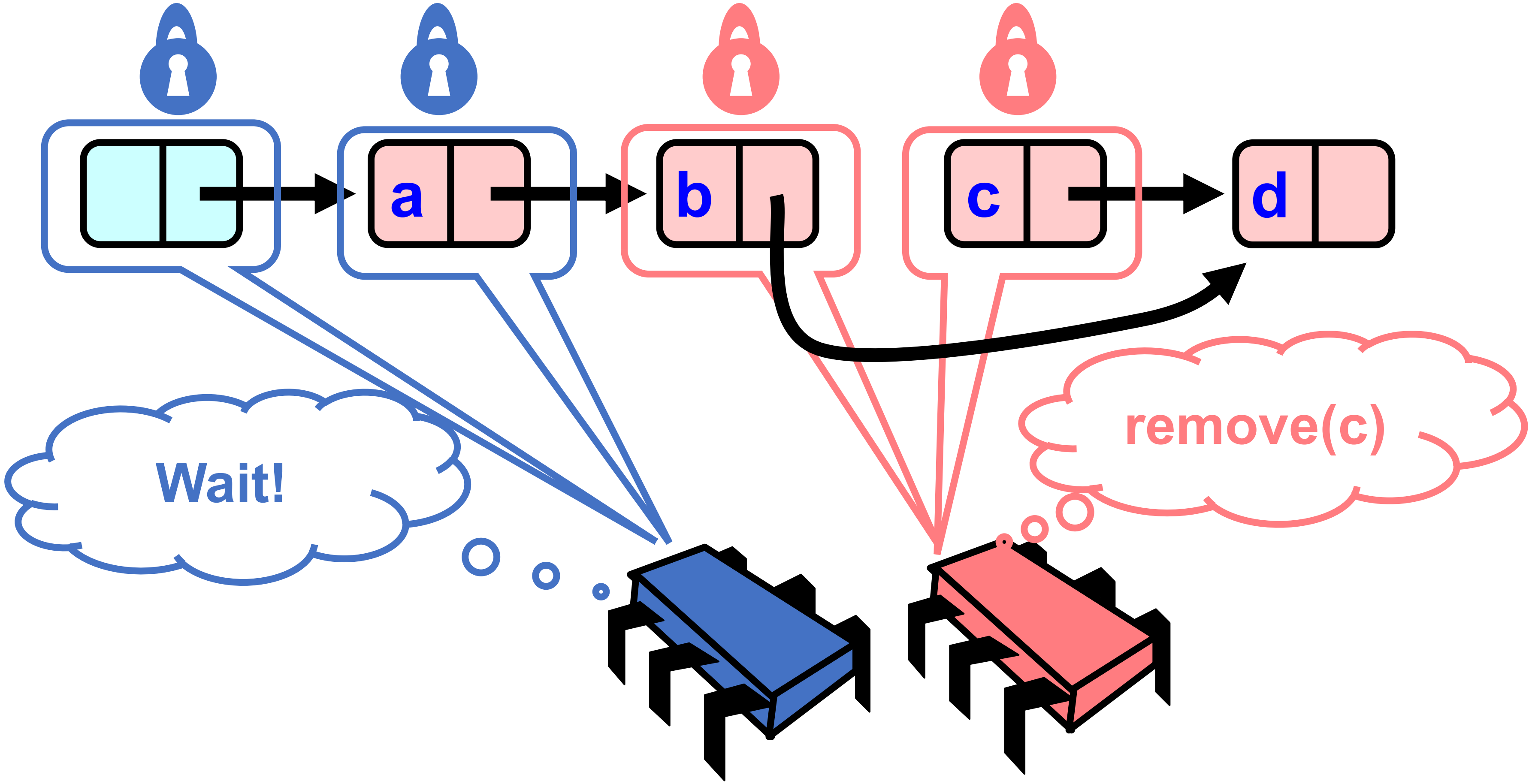
Removing a Node



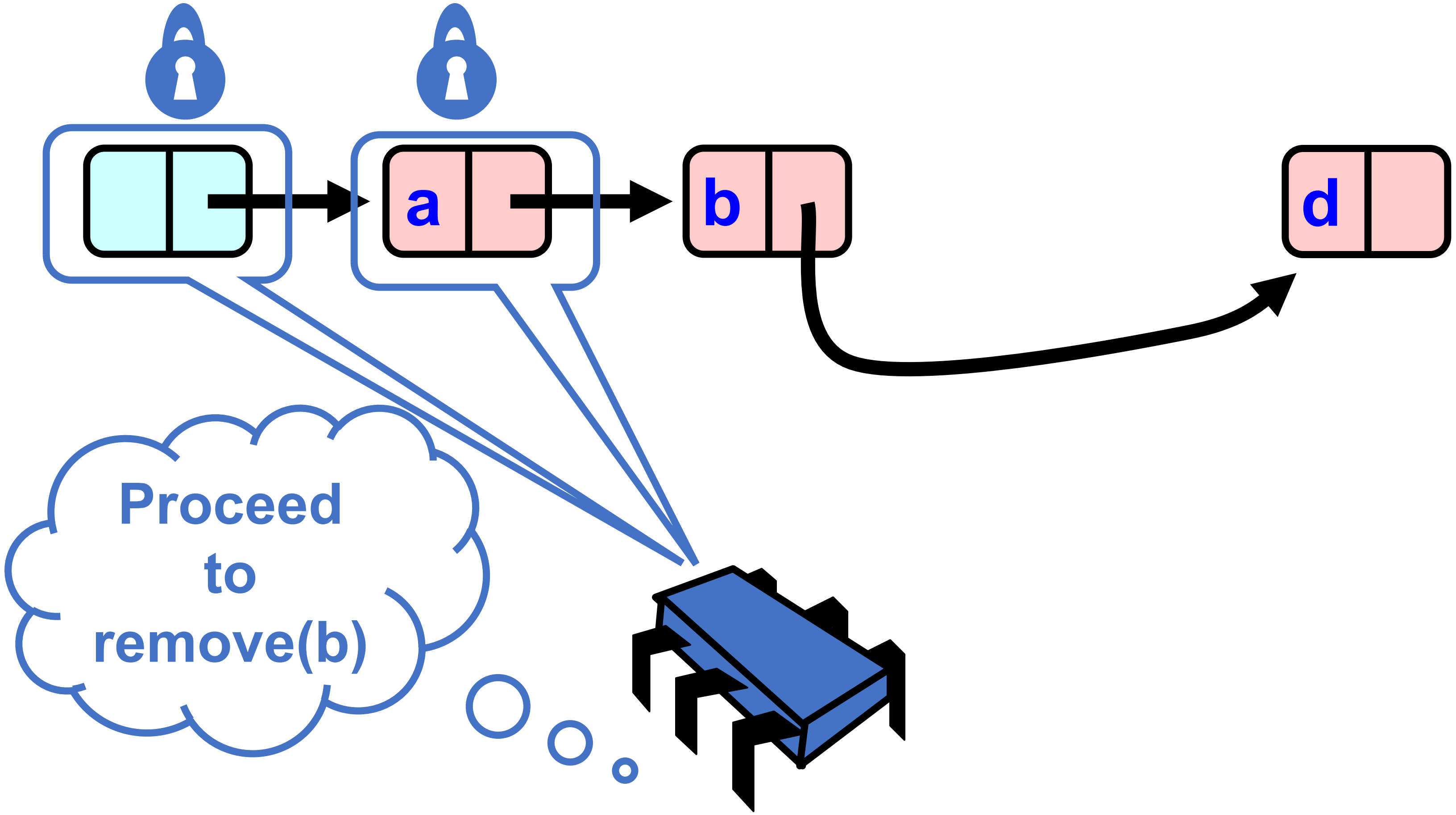
Removing a Node



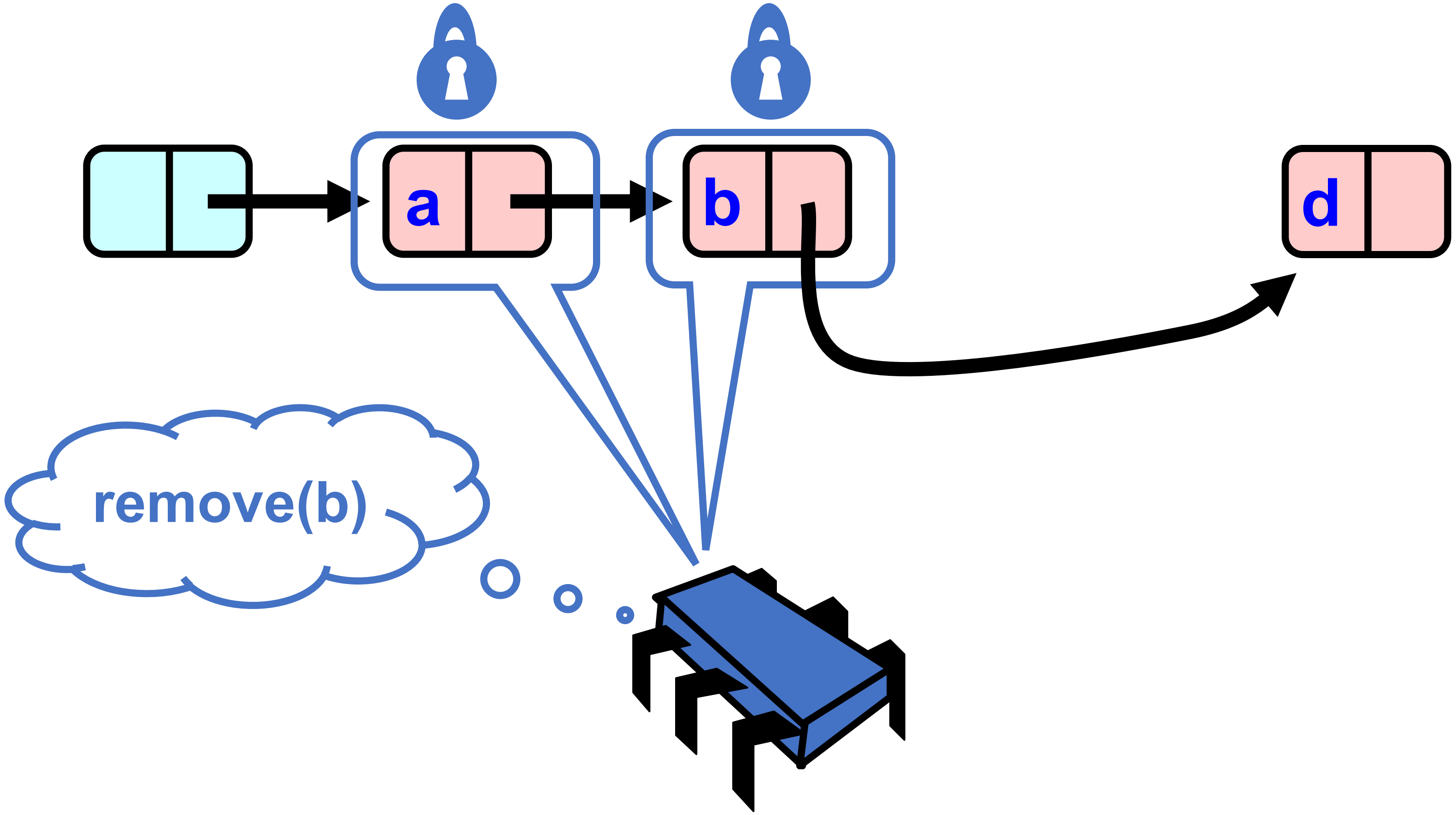
Removing a Node



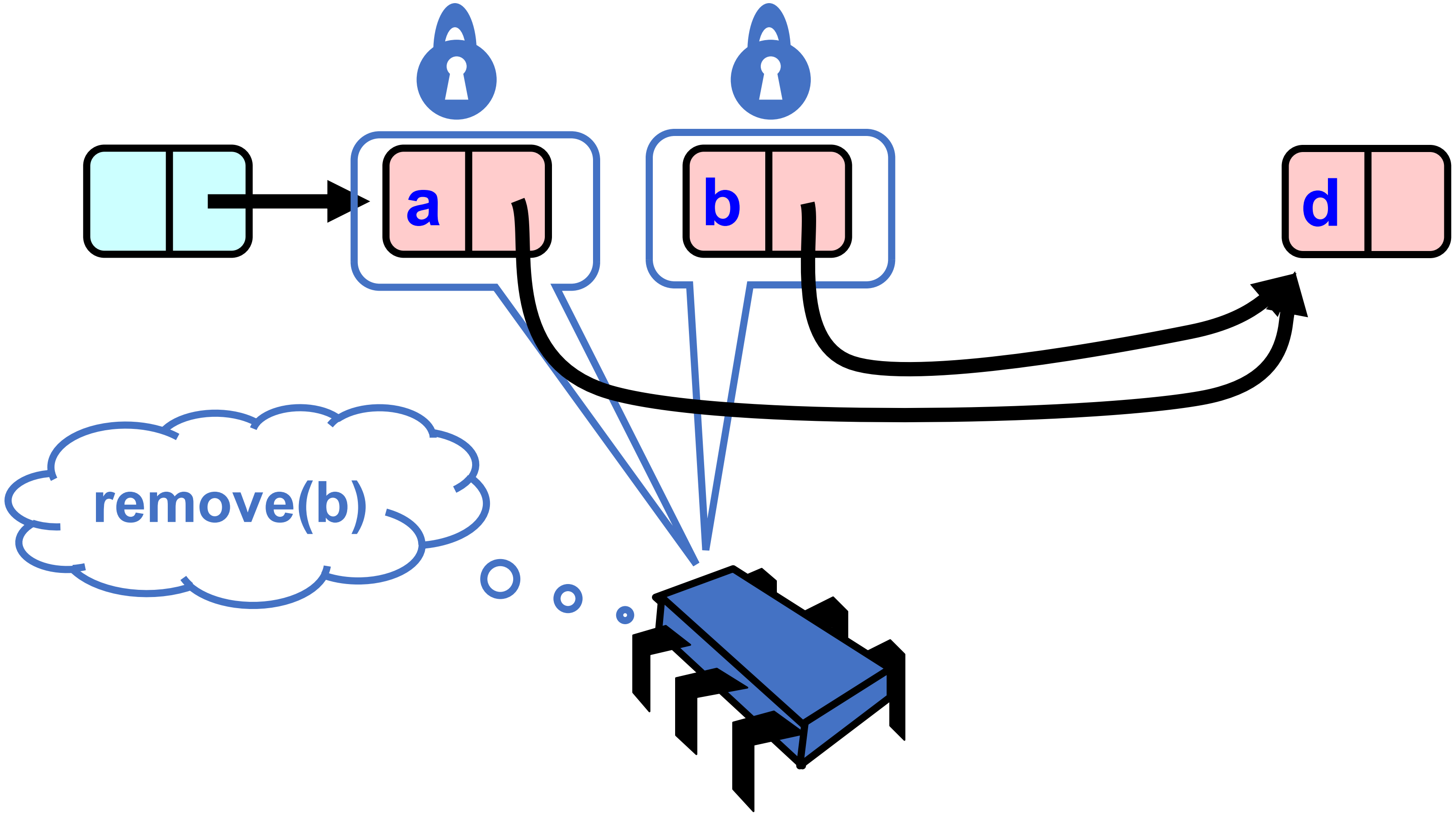
Removing a Node



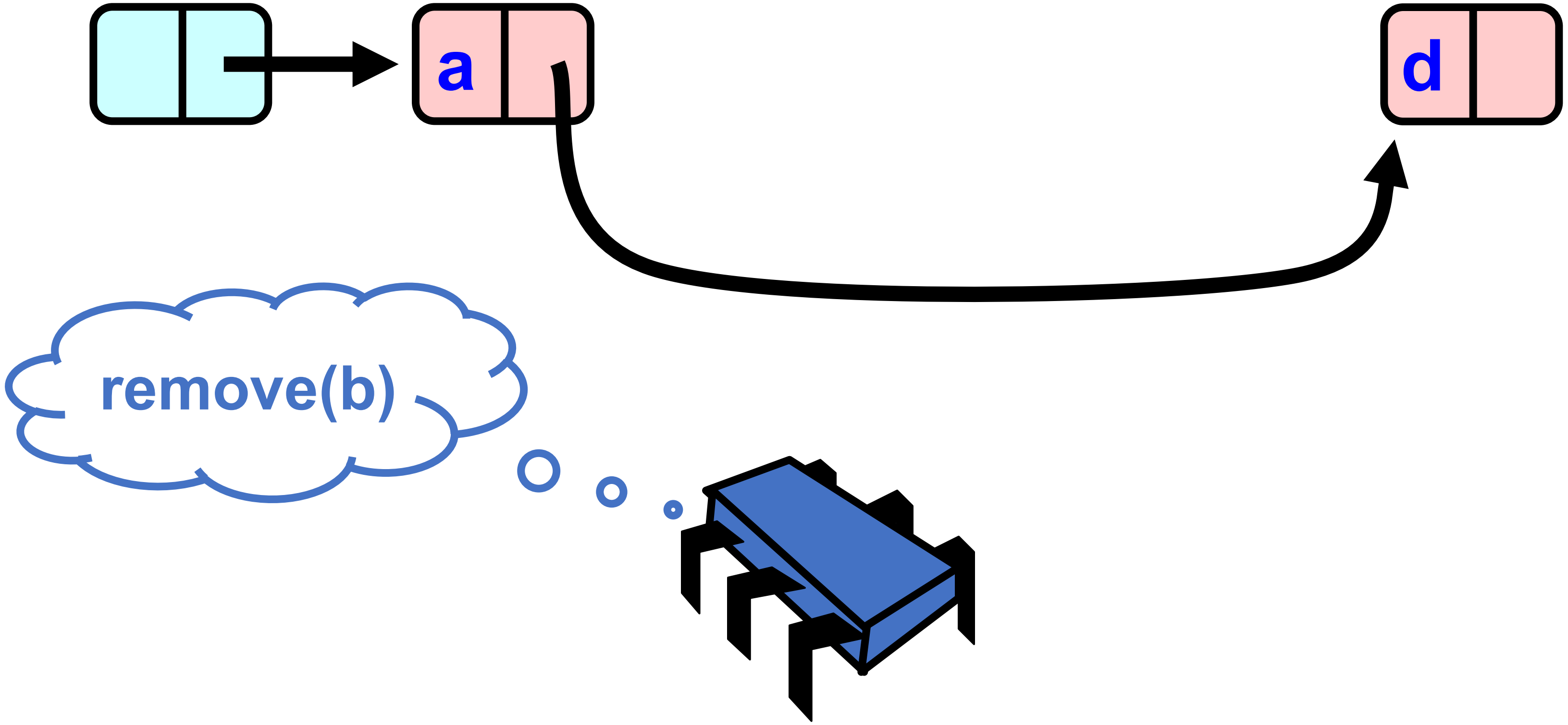
Removing a Node



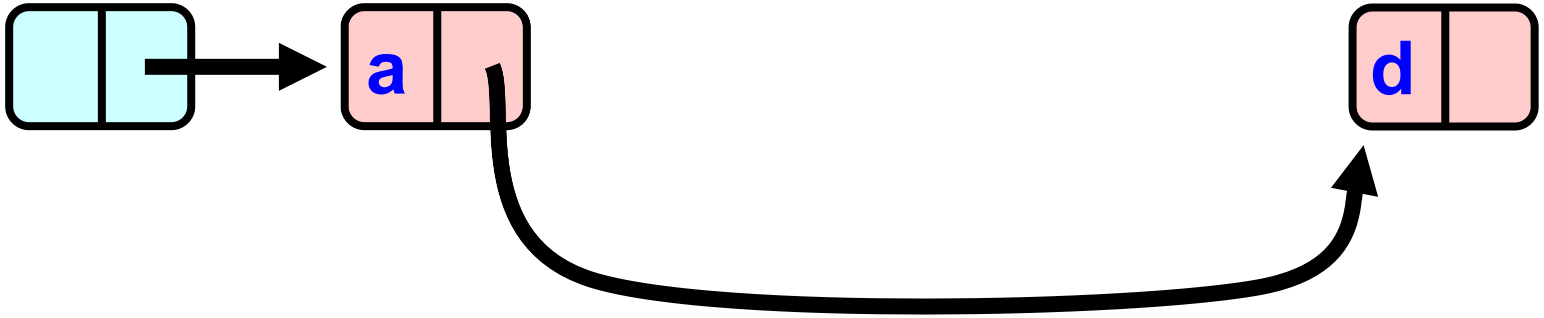
Removing a Node



Removing a Node



Removing a Node



Adding Nodes

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