# CSE113: Parallel Programming

May 18, 2021

- **Topic**: Memory Consistency and Barriers
  - Compiling for memory consistency
  - Barrier specification
  - Barrier implementation

target machine
TSO (x86)

| | L | S |
|---|---|---|
| L | NO | different address |
| S | NO | No |

# Announcements

- Homework is due this Friday (May 21)
  - I will do the second hour of Wednesday's office hour as an open HW question session
  - New packet uploaded! (fixed a bug in the computation in part 2). Please download new packet and specification
  - I will open office hours ***around*** 2PM on Wednesday (wait for the announcement)
- New HW assigned this Friday by midnight
  - Memory models and Barriers - you should have all the info you need for the assignment after this lecture
- Guest lecture on Thursday!
  - Message passing concurrency and GPU compiler testing

# Announcements

- Midterm Answer Key
  - Trying to get it out by end of this week
  - I am getting 2$^{nd}$ dose of vaccine after class today; I appreciate your patience!
- Aiming to get HW2 grades out in 1 week

# Quiz

# Quiz

- Discuss answers

# Schedule

- More Memory Model Examples
- Compiling Memory Models
- Barrier Specification
- Barrier Implementation

# Schedule

- **More Memory Model Examples**
- Compiling Memory Models
- Barrier Specification
- Barrier Implementation

*Global variable:*

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

Can `t0 == t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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

*Review: are these instructions in C++?*

*Global variable:*

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

Can `t0 == t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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

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

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

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

```
L:%t0 = load(x);
```

*Global variable:*

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

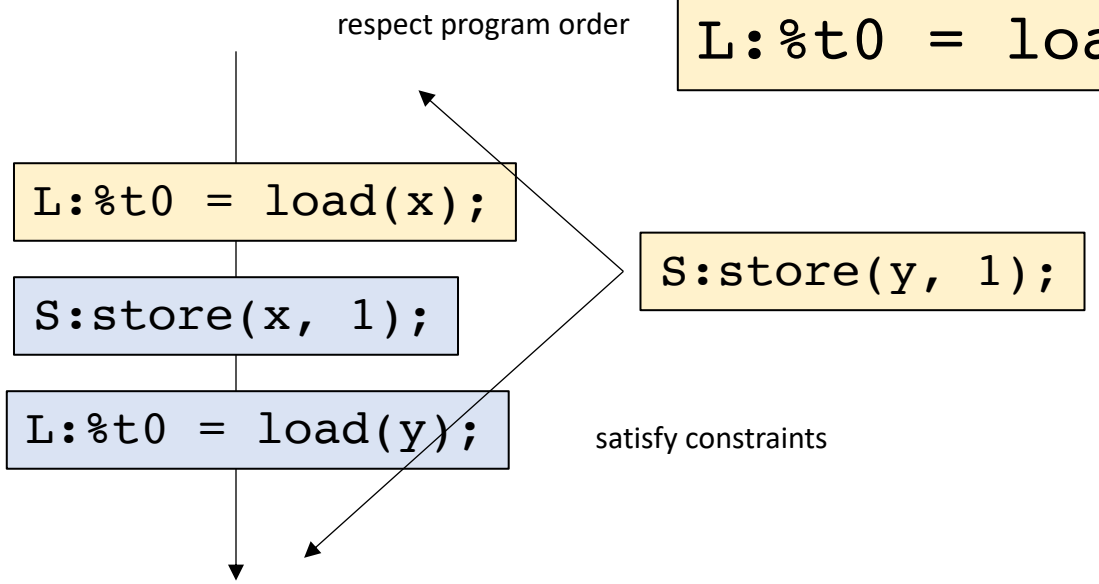
Can `t0 == t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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



*Global variable:*

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

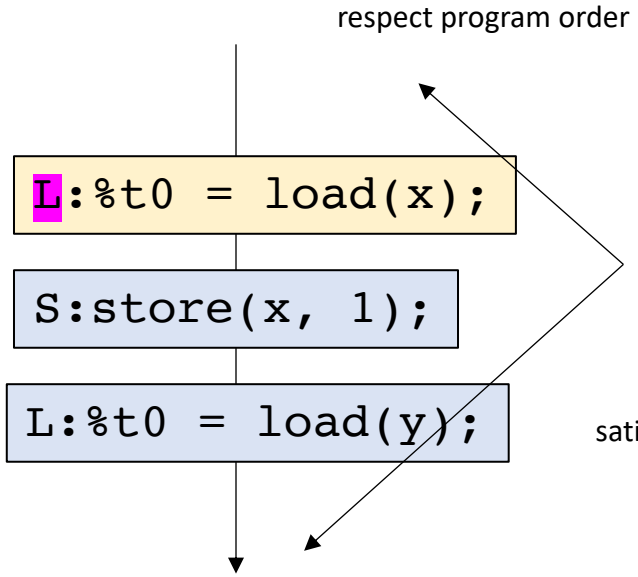
Can `t0 == t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




memory access 0

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

Is this allowed in TSO?

*Global variable:*

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

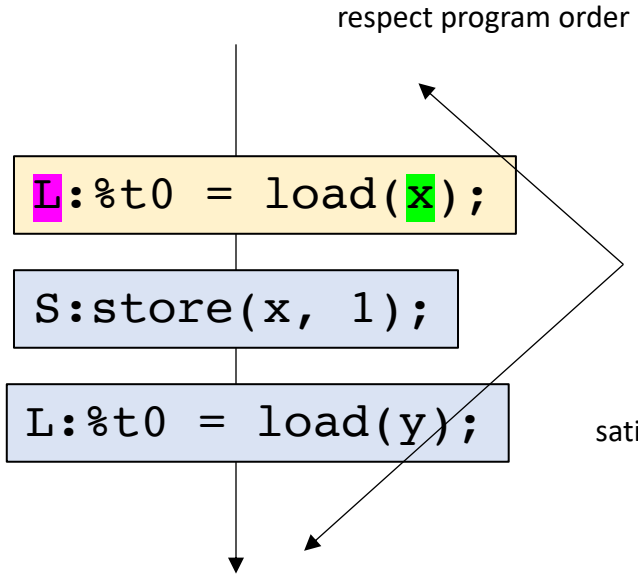
Can `t0 == t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




memory access 0

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

Is this allowed in TSO?

*Global variable:*

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

Can `t0 == t1 == 0`?

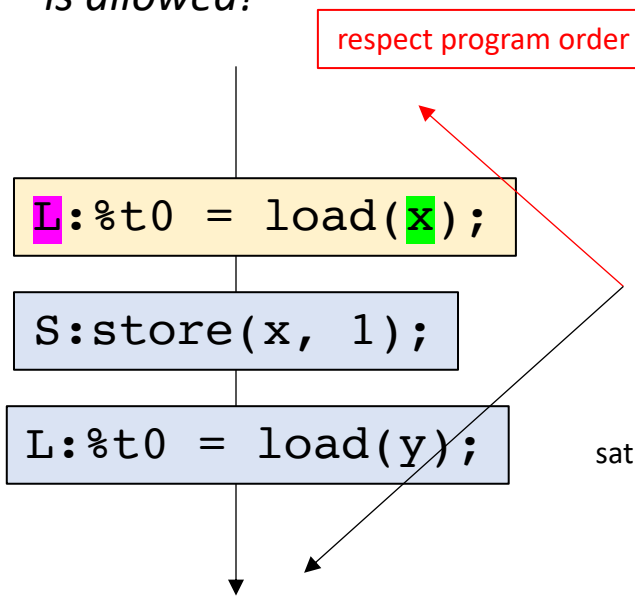
*Thread 0:*

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

*Thread 1:*

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

*we can reorder the store! And the execution is allowed!*




satisfy constraints

memory access 0

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

Is this allowed in TSO?

*Global variable:*

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

Can `t0 == t1 == 0`?

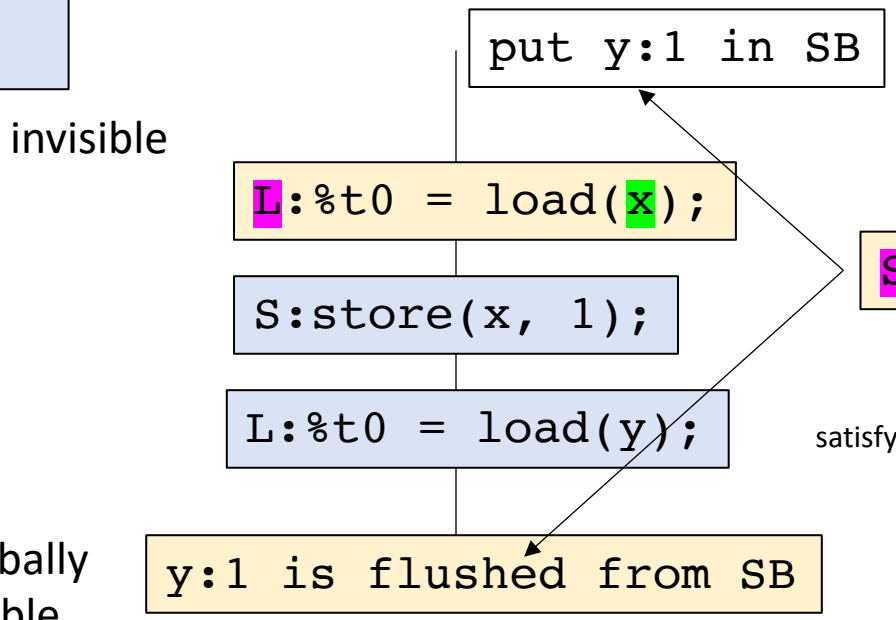
*Thread 0:*

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

*Thread 1:*

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

*what happens behind the scenes?*




satisfy constraints

memory access 0

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

Is this allowed in TSO?

*Global variable:*

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

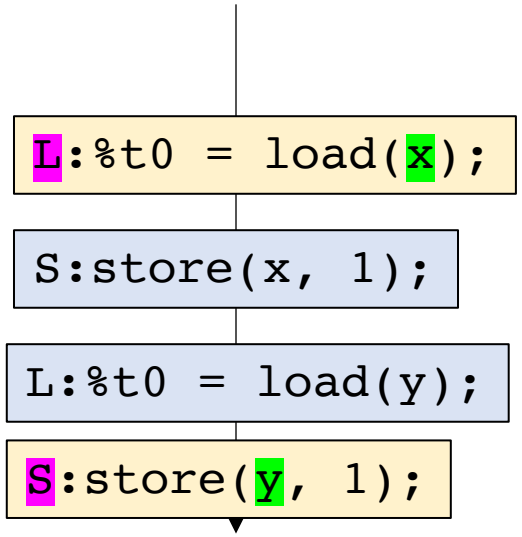
Can `t0 == t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




*in practice we just show the store happening here*

Is this allowed in TSO?

| memory access 1 \ memory access 0 | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | NO |

*Global variable:*

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

Can `t0 == t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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

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

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

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

```
L:%t0 = load(x);
```

How do we disallow the relaxed execution?

*Global variable:*

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

Can `t0 == t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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

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

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

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

```
L:%t0 = load(x);
```

We add fences

*Global variable:*

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

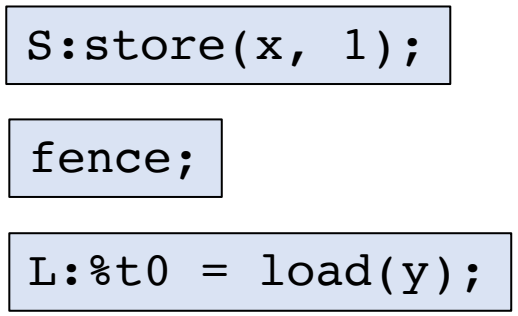
Can `t0 == t1 == 0`?

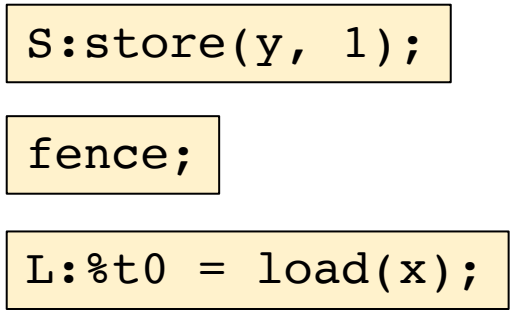
*Thread 0:*

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

*Thread 1:*

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




We add fences

*Global variable:*

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

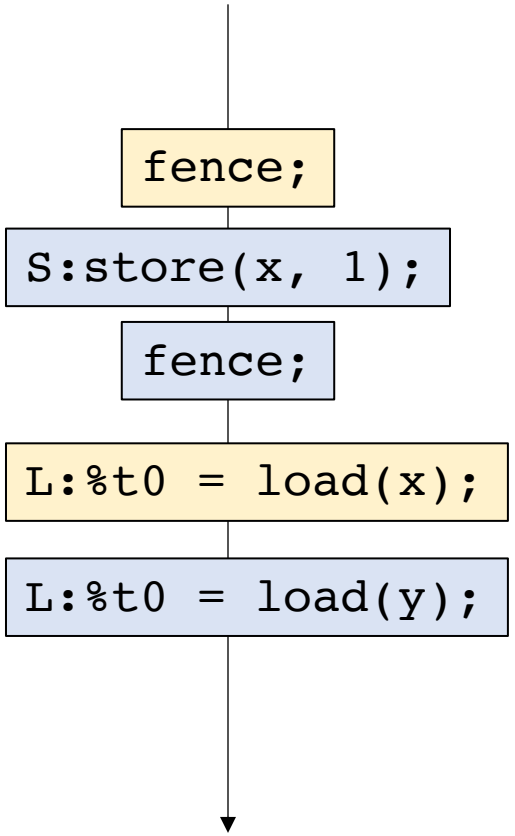
Can `t0 == t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




`S:store(y, 1);`

We add fences

*Global variable:*

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

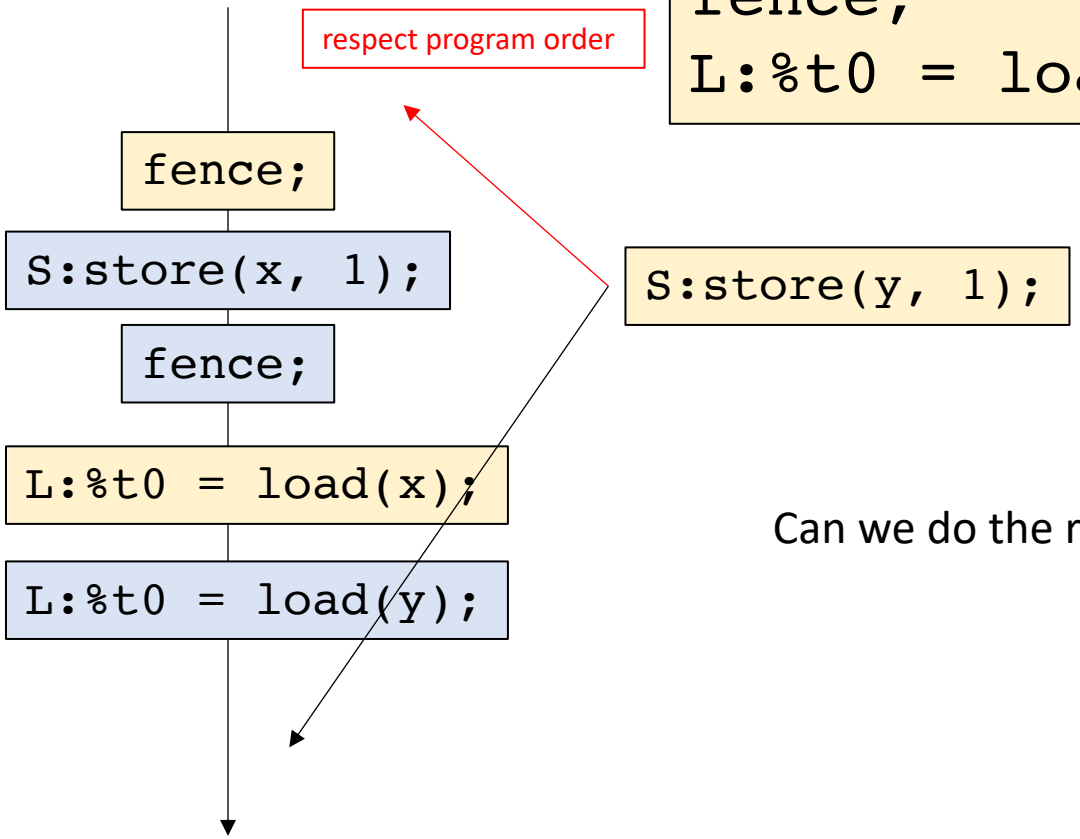
Can `t0 == t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




Can we do the reordering?

| memory access 1 \ memory access 0 | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | NO |

*Global variable:*

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

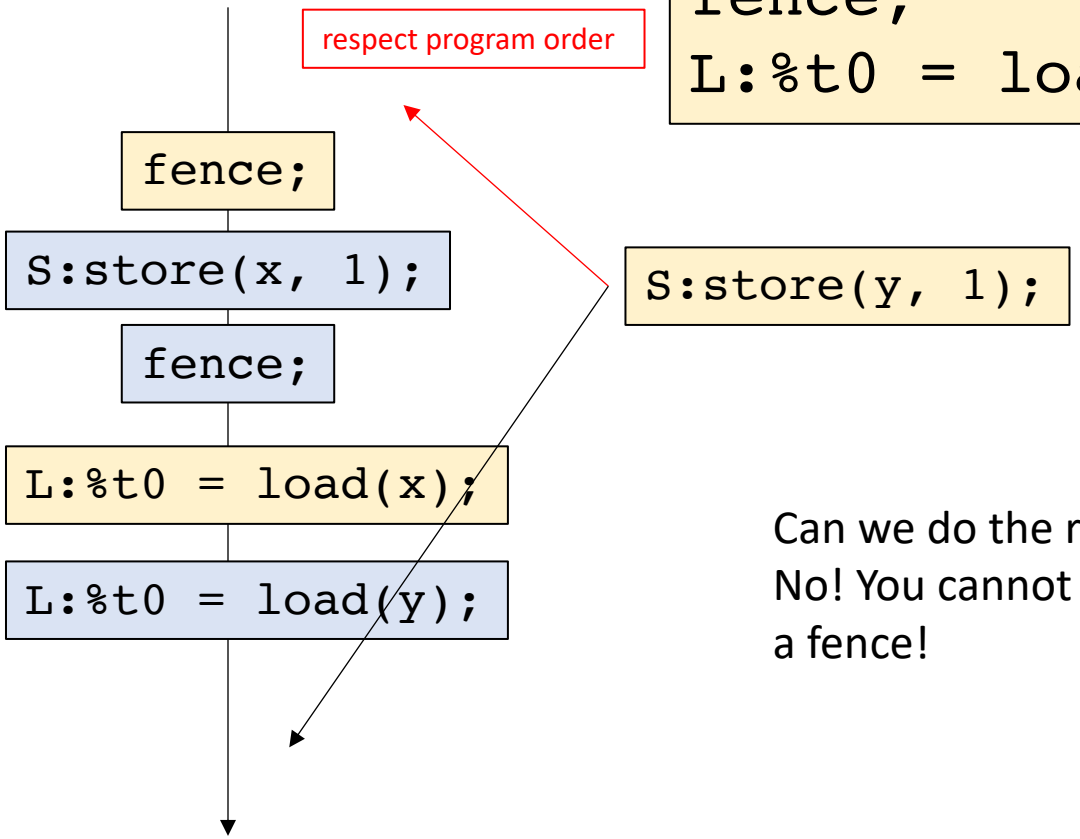
Can `t0 == t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




| memory access 1 \ memory access 0 | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | NO |

Can we do the reordering?
No! You cannot move past a fence!

*Global variable:*

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

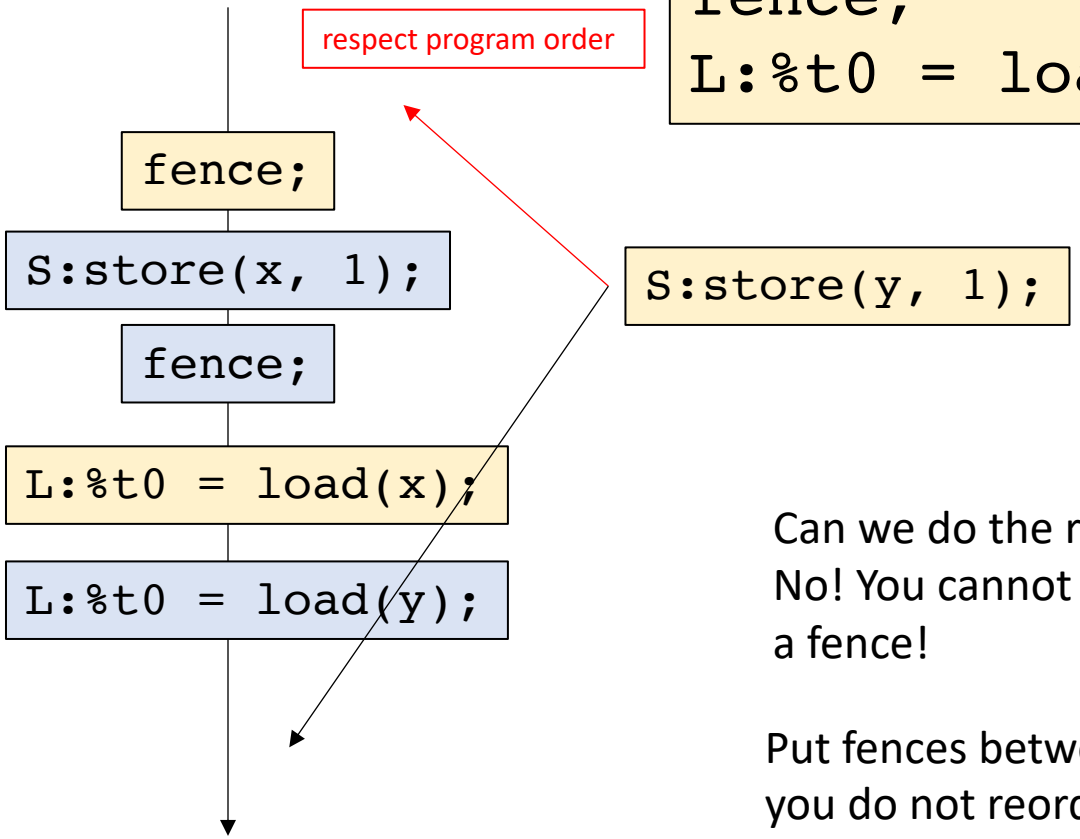
Can `t0 == t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




| memory access 1 \ memory access 0 | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | NO |

Can we do the reordering?
No! You cannot move past a fence!

Put fences between instructions you do not reordered!

# 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(x,1)
```

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

```
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`?

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)
```

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

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

```
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`?

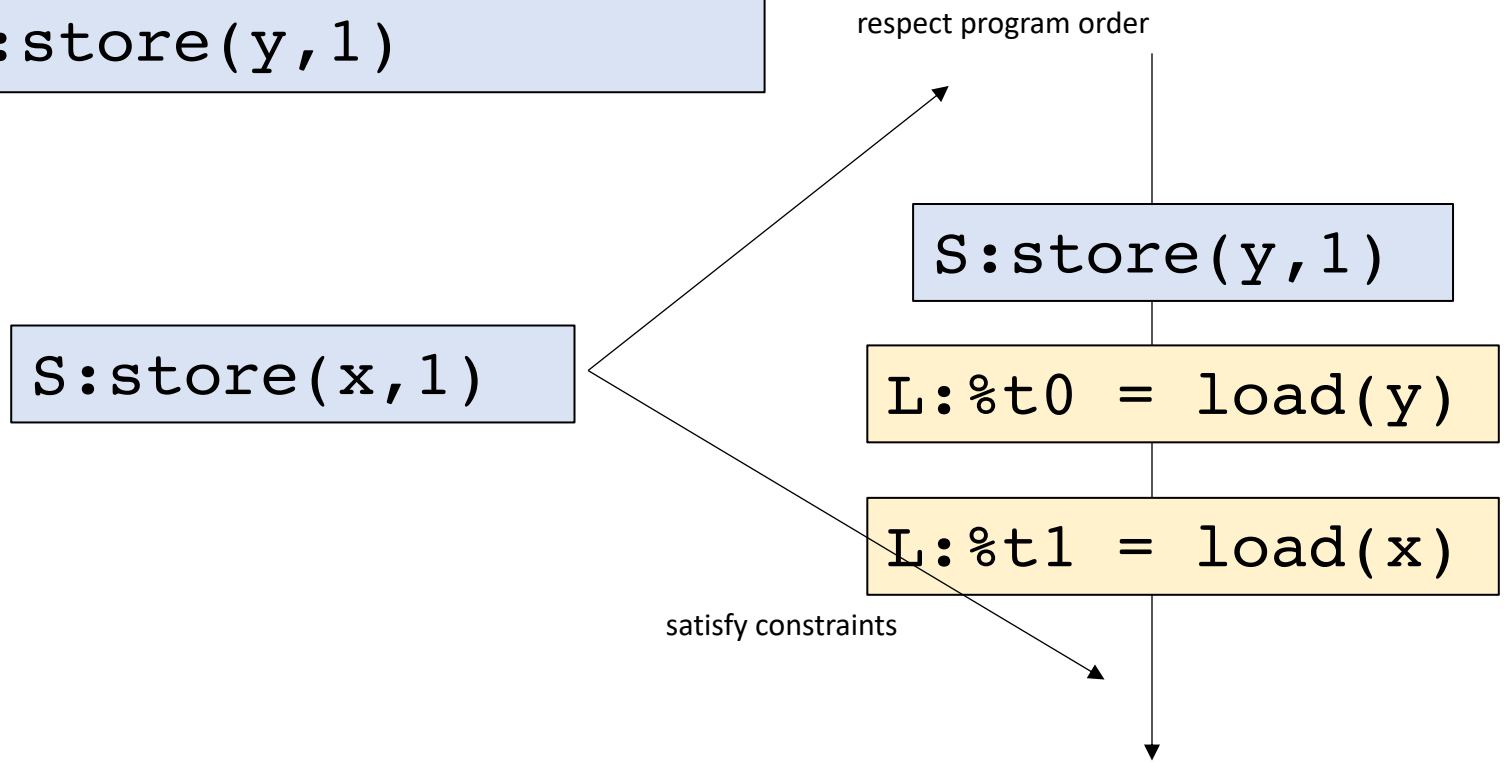
start off thinking about sequential consistency

*Thread 0:*

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

*Thread 1:*

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



*Global variable:*

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

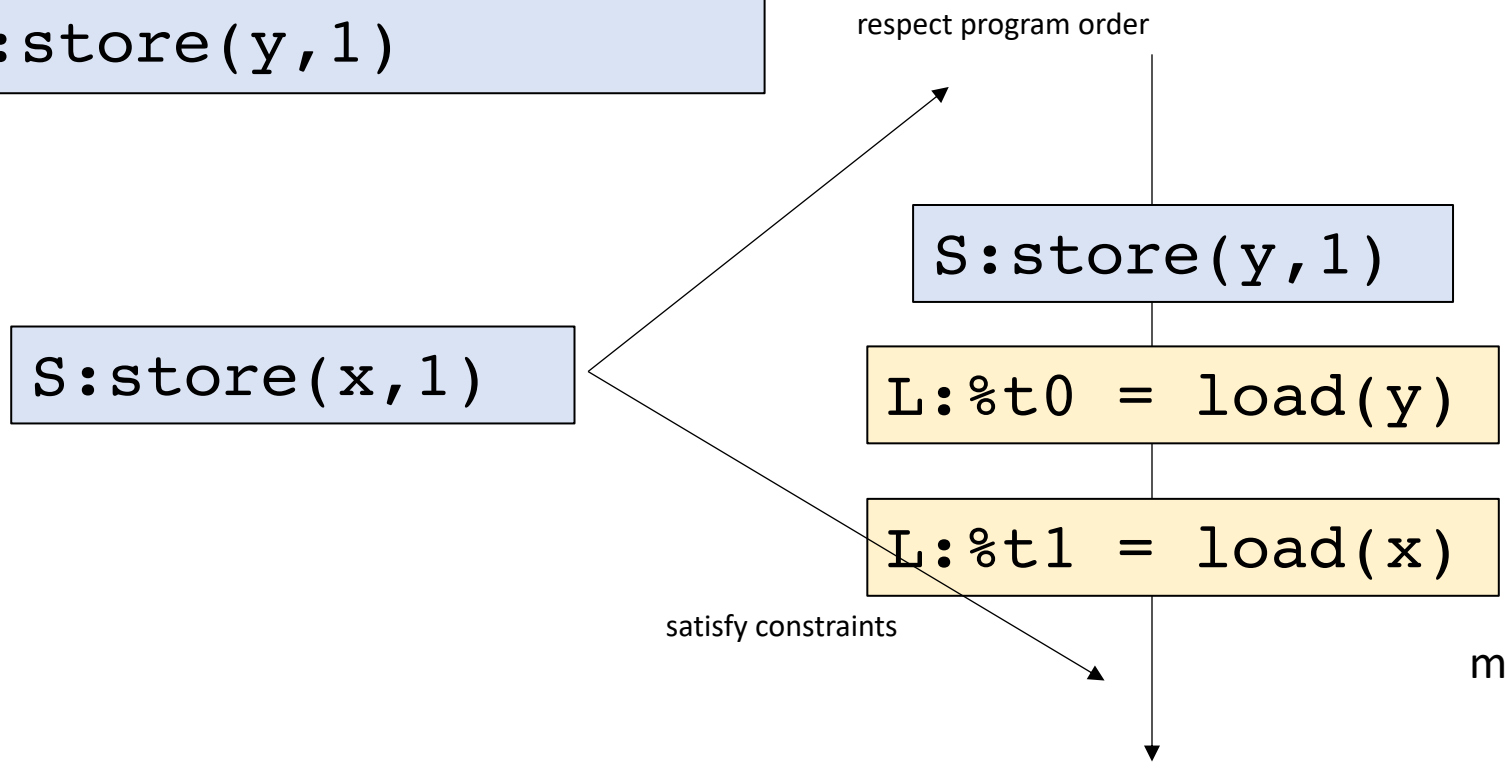
Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




memory access 0

memory access 1

| | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | NO |

What about TSO?

*Global variable:*

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

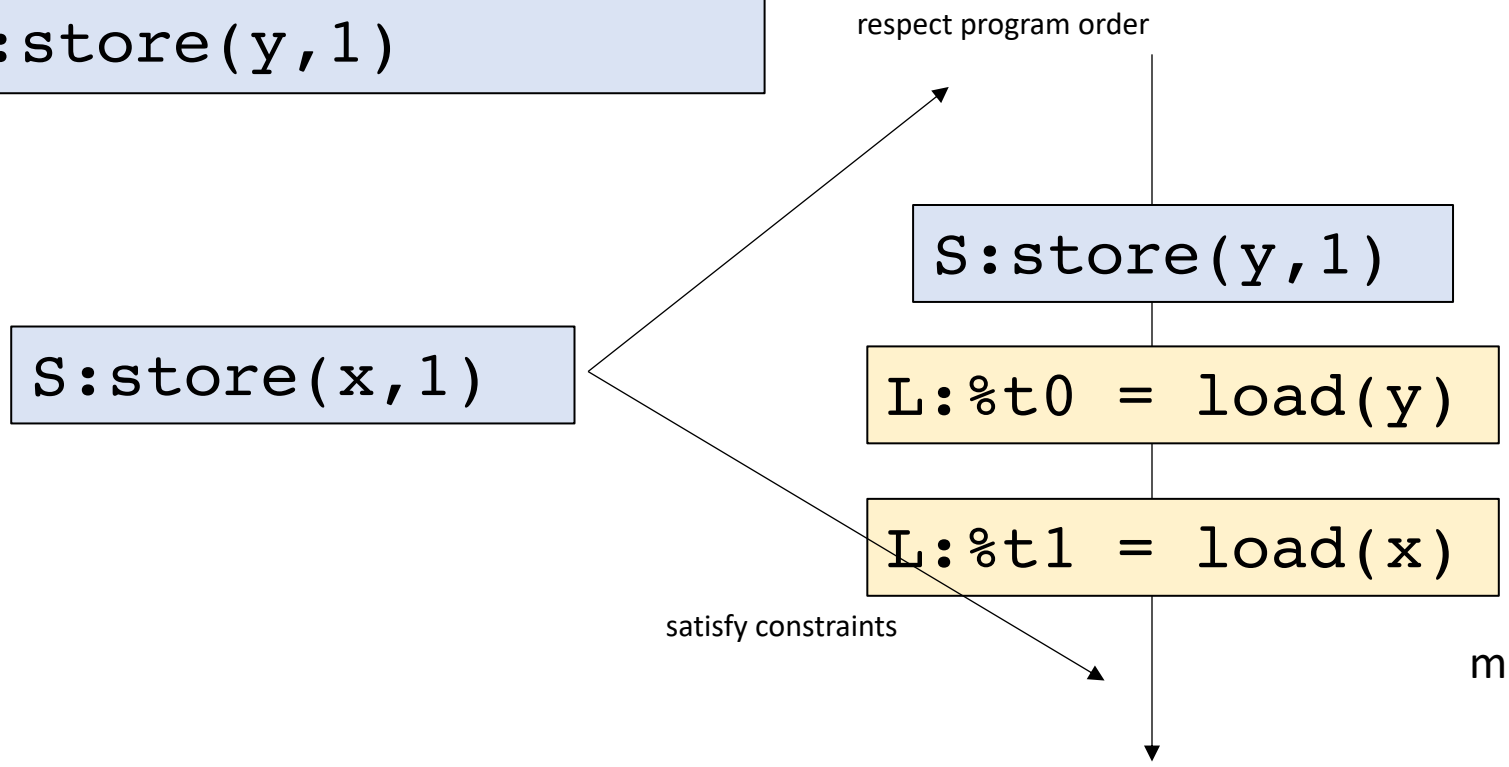
Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




memory access 0

memory access 1

| | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | NO |

What about TSO? NO

*Global variable:*

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

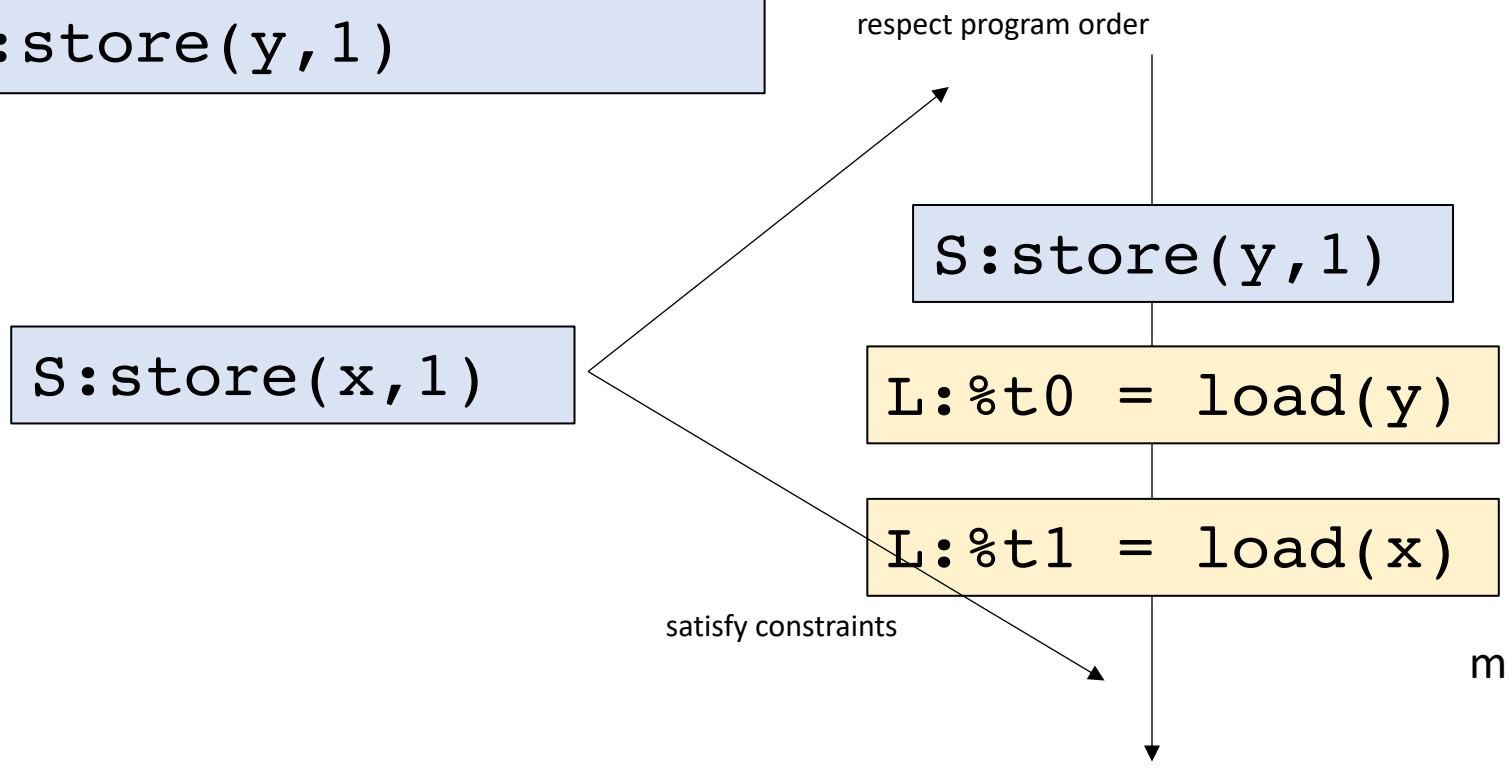
Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




memory access 0

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};
```

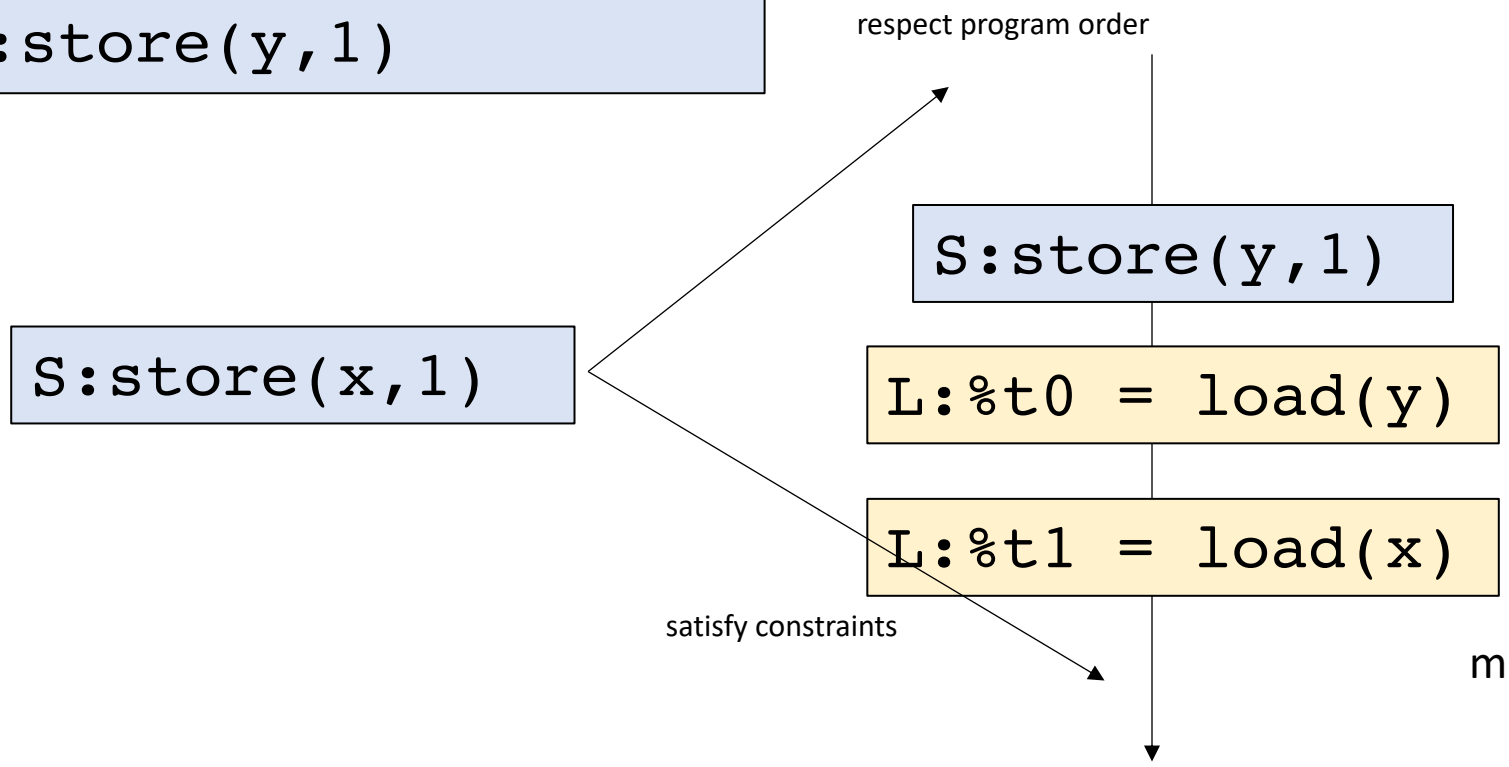
Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




memory access 0

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};
```

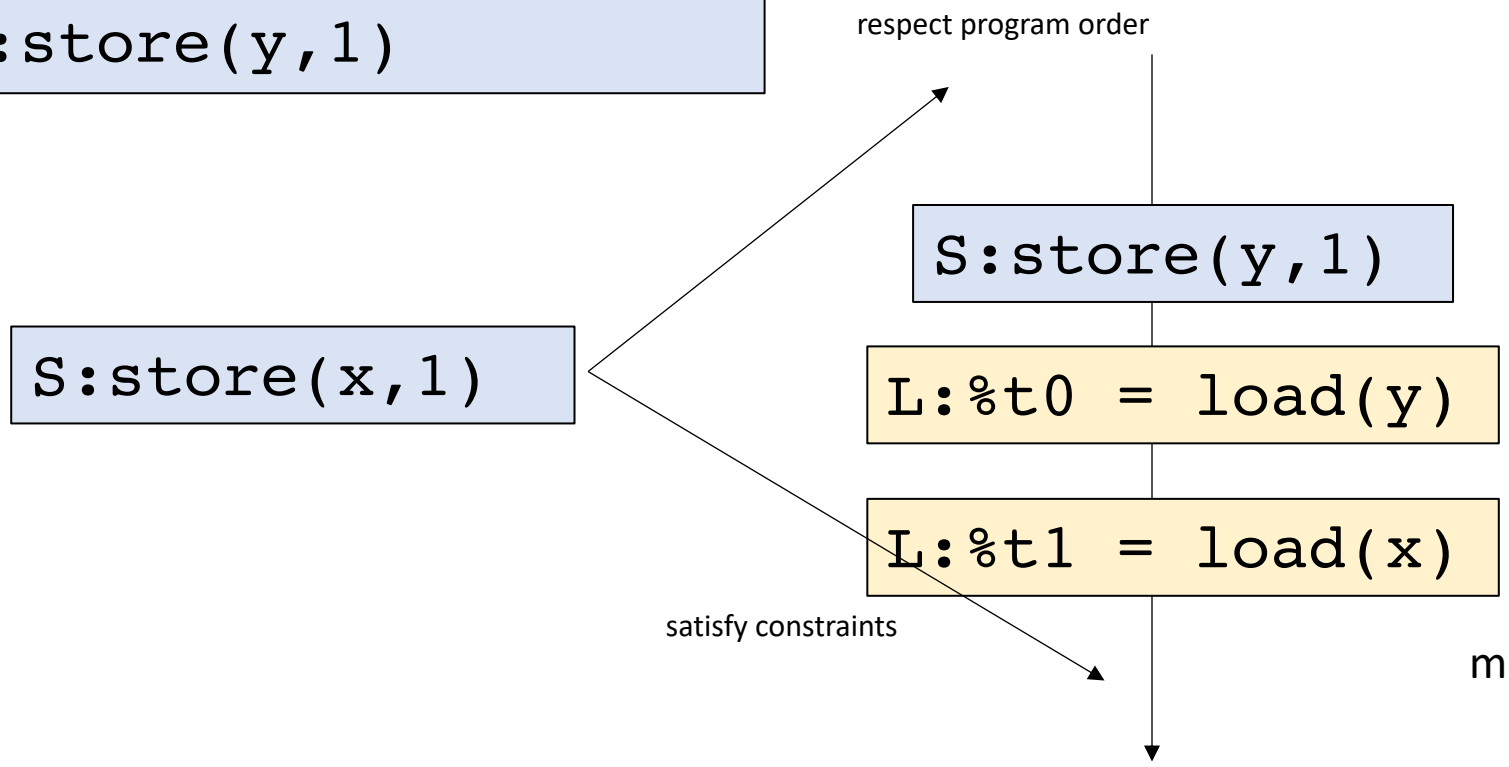
Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




memory access 0

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};
```

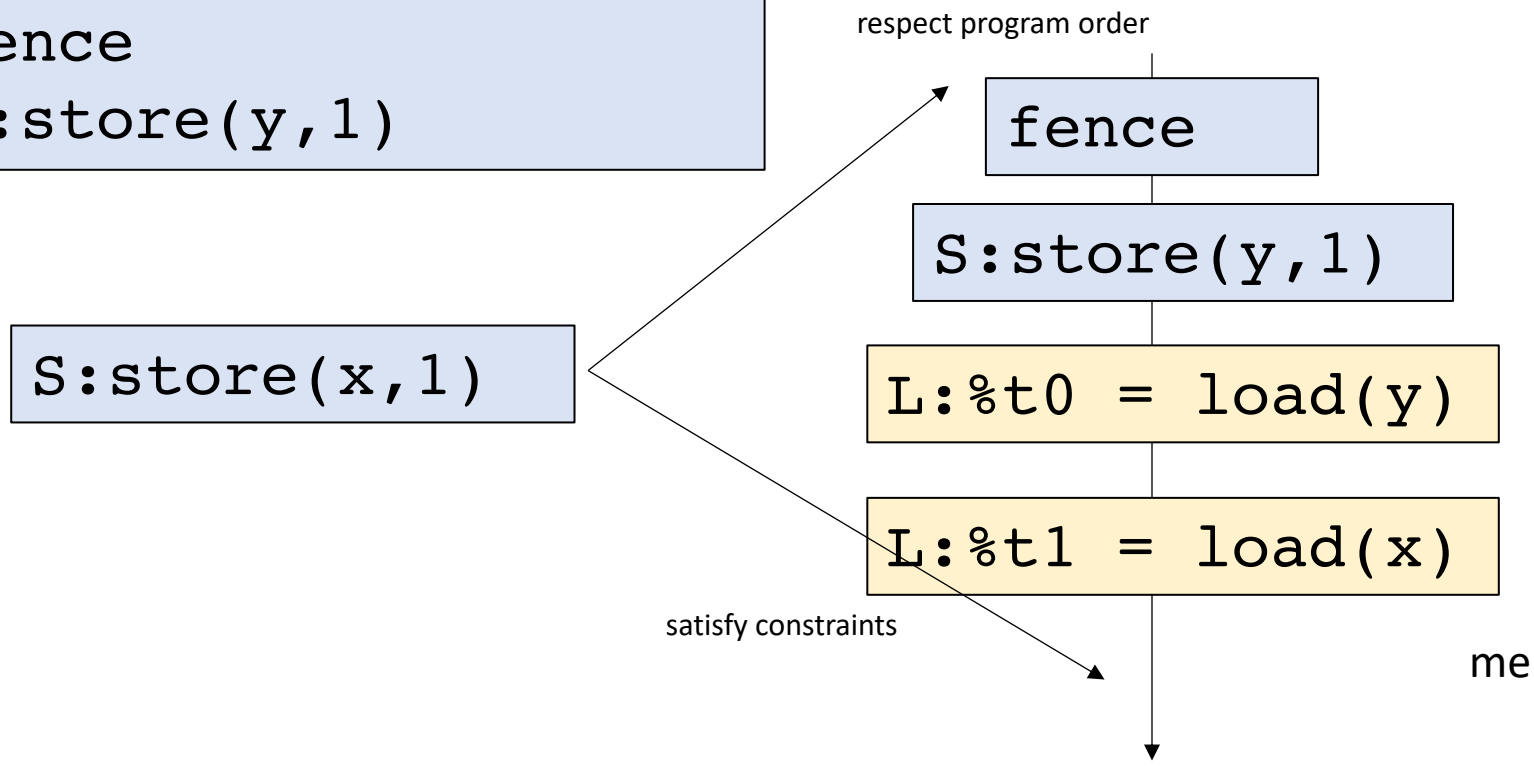
Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




memory access 0

memory access 1

| | 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};
```

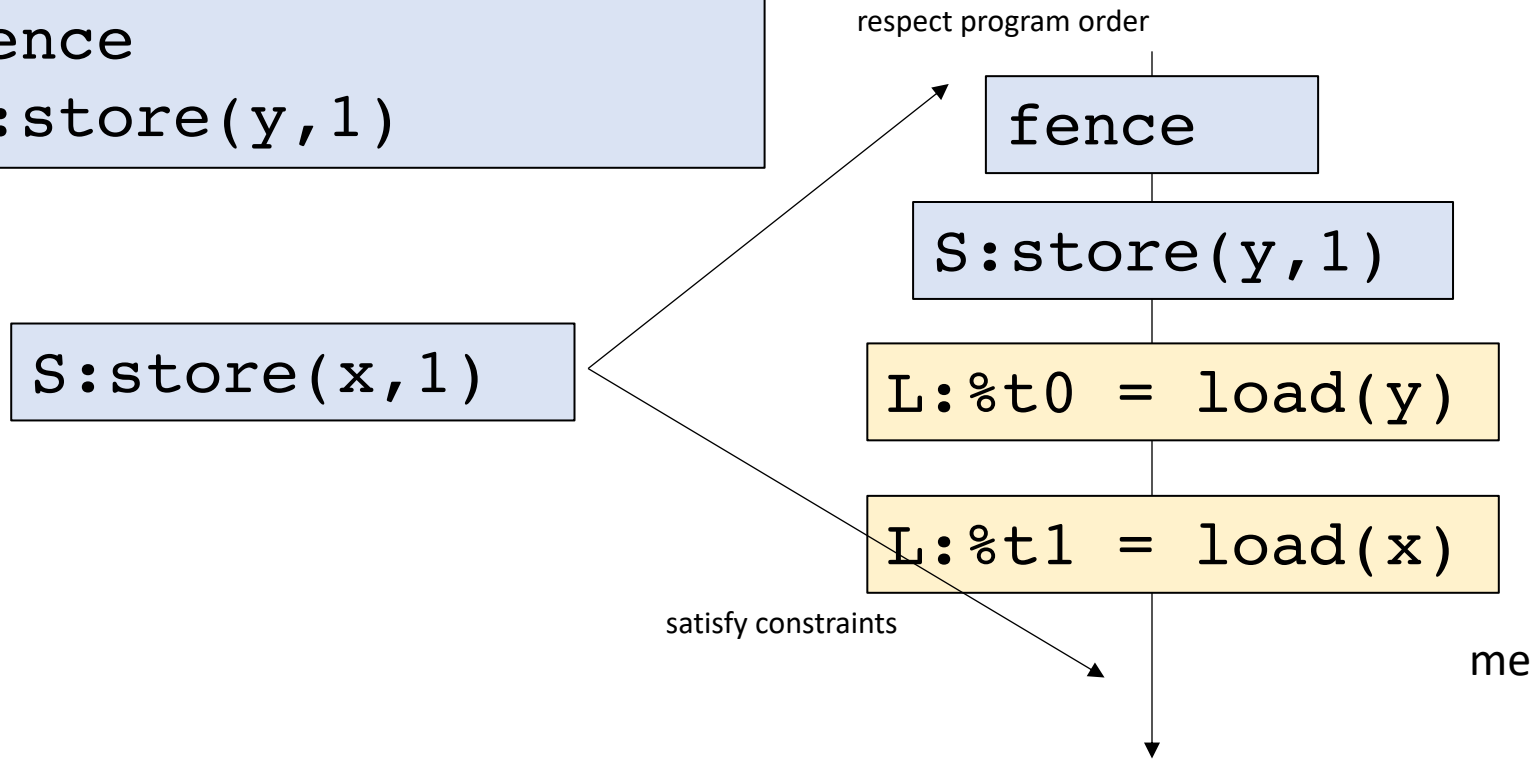
Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




memory access 0

| memory access 1 | L | S |
|---|---|---|
| L | YES | Different address |
| S | Different address | Different address |

What about RMO?

*Global variable:*

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

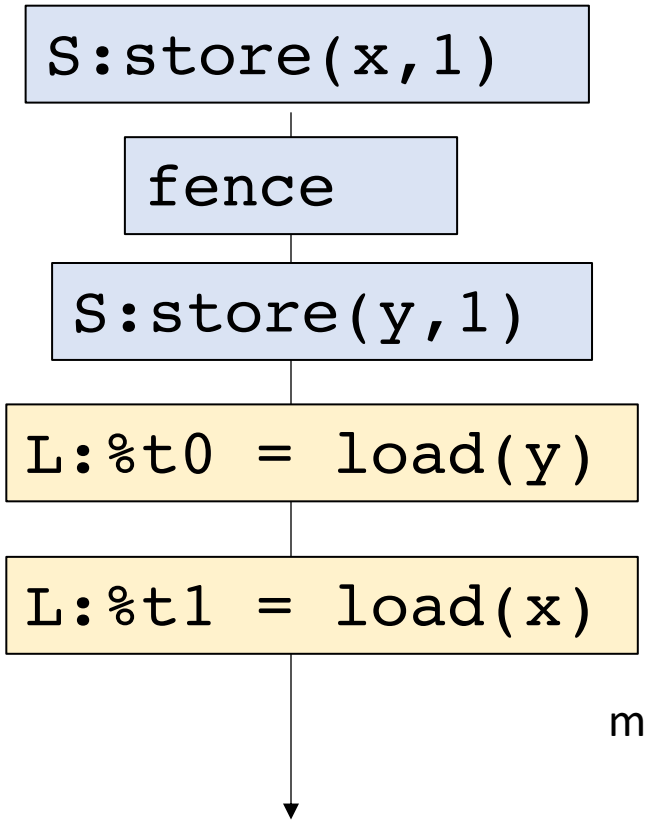
Question: can `t0 == 1 and t1 == 0`?

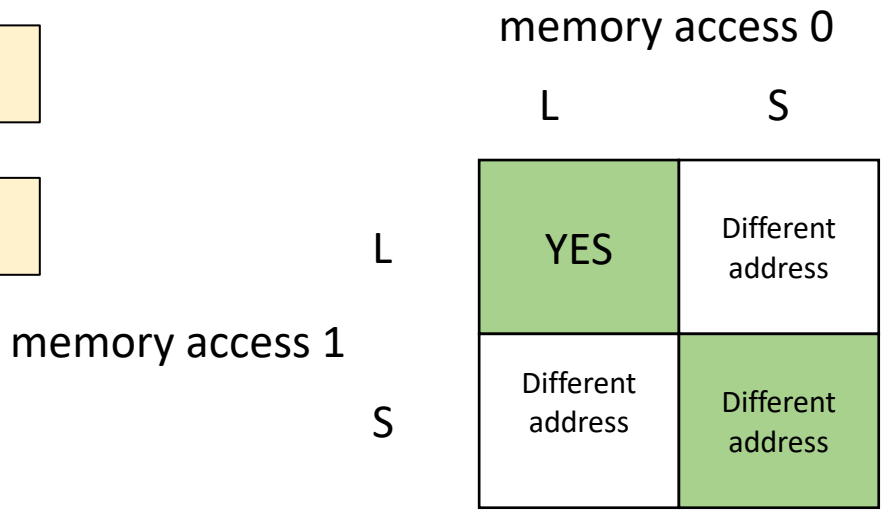
*Thread 0:*

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

*Thread 1:*

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




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 1 \ memory access 0 | L | S |
|---|---|---|
| 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:*

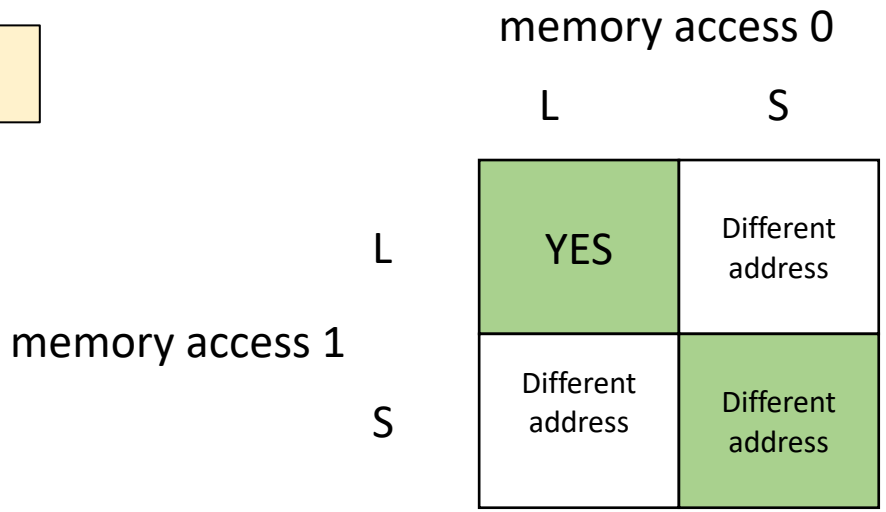
```
S:store(x,1)
fence
S:store(y,1)
```

*Thread 1:*

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

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

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




What about RMO? add a fence

*Global variable:*

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

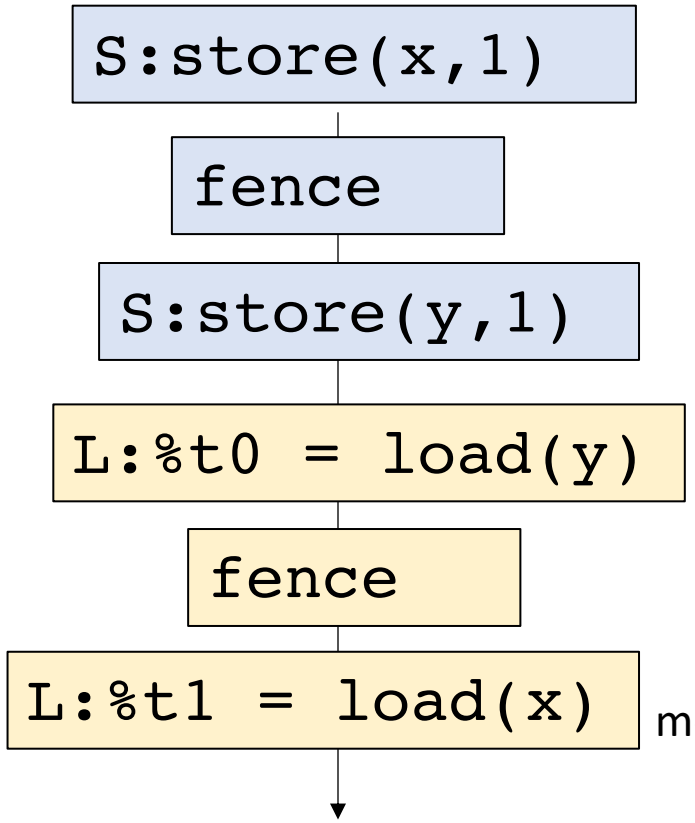
Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

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

*Thread 1:*

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




memory access 0

| memory access 1 | L | S |
|---|---|---|
| L | YES | Different address |
| S | Different address | Different address |

Now the relaxed behavior is disallowed

# This is a mess!

- Luckily, since 2011 we have C++ memory model:
  - Provides sequential consistency

- How does this work?

# Schedule

- More Memory Model Examples
- **Compiling Memory Models**
- Barrier Specification
- Barrier Implementation

# 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++ → ISA

x.store(1);

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 |

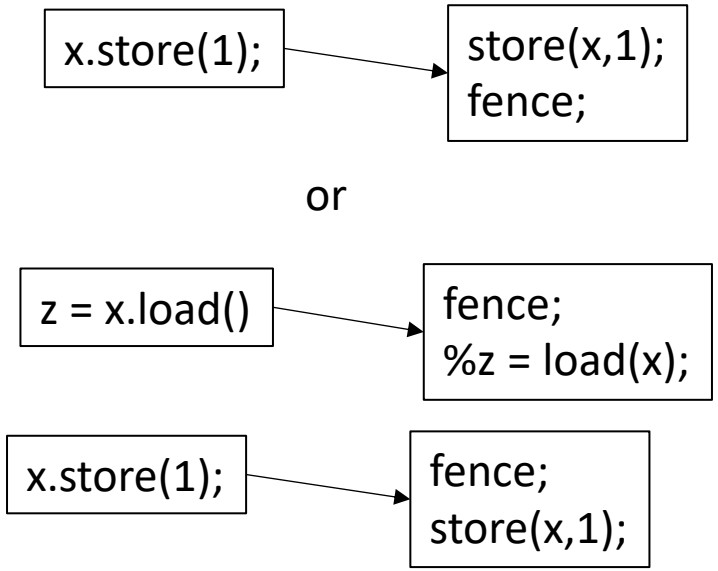
# C++11 atomic operation compilation

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




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

Where have we seen `memory_order_relaxed`?

# Optimizations: relaxed peeking

- What about the load in the loop? Remember the memory fence? Do we need to flush our caches every time we peek?
- We only need to flush when we actually acquire the mutex

```
void lock(int thread_id) {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load(memory_order_relaxed) == true);
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```

# 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 |

TSO

| | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | NO |

PSO

| | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | Different address |

RMO

| | L | S |
|---|---|---|
| L | YES | Different address |
| S | Different address | Different address |

# 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 ommitted)
  - `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

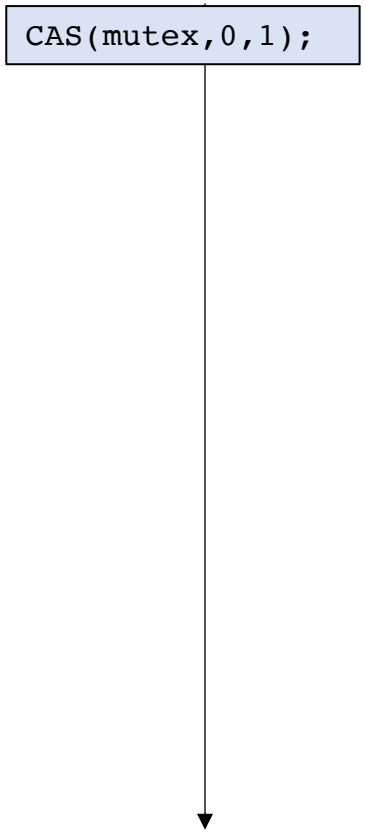
What is an execution?

*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);
```




*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);
```

*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*

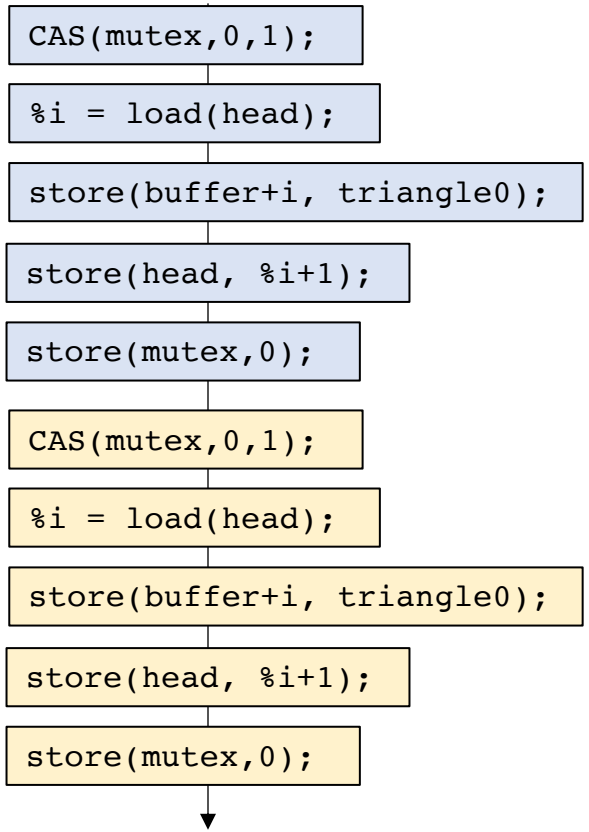
*Thread 0:*

```
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*



*Thread 0:*

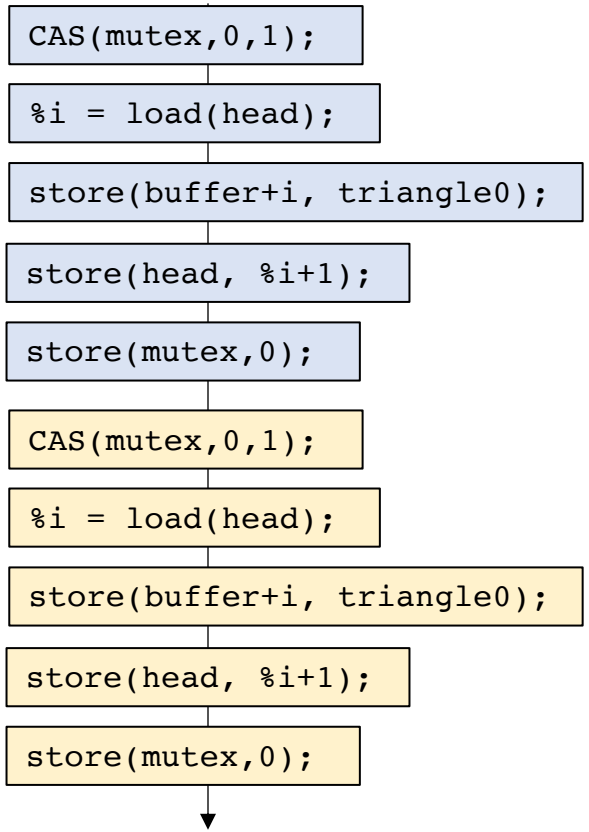
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*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:*

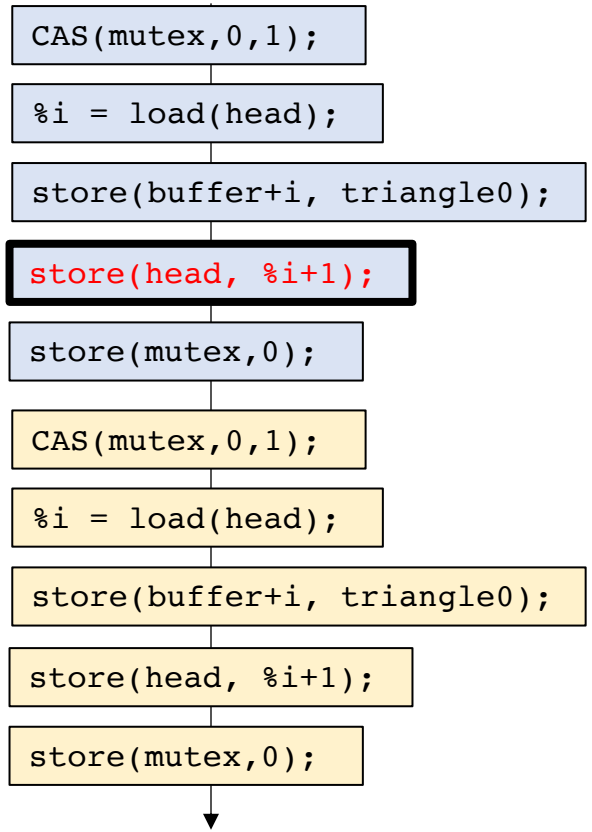
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*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:*

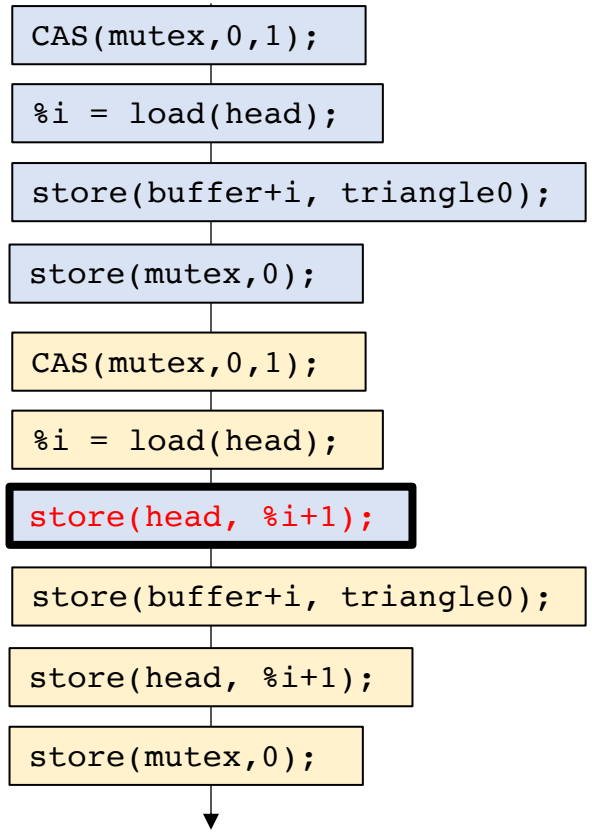
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*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 |

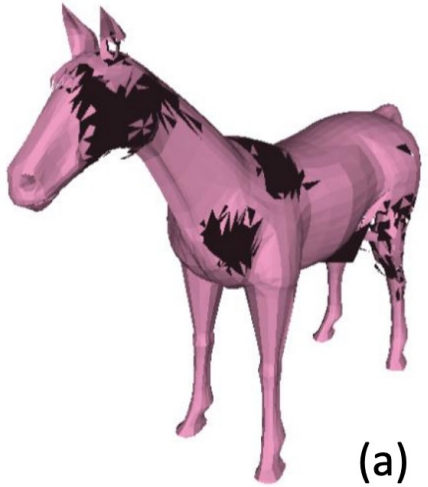



What just happened if this store moves?

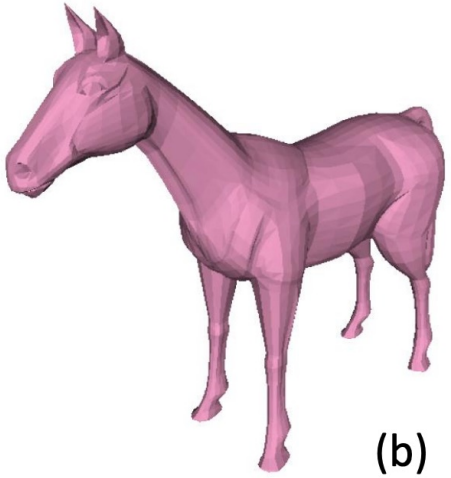
# Nvidia in 2015

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

# Nvidia in 2015



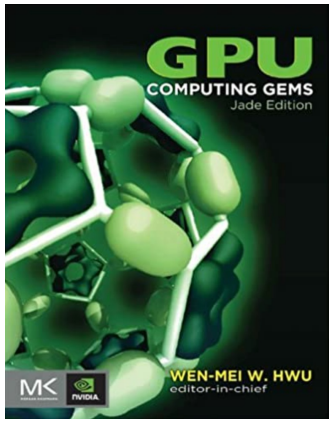
(a)

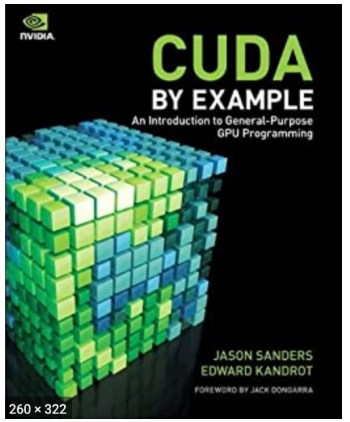


(b)

(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:*

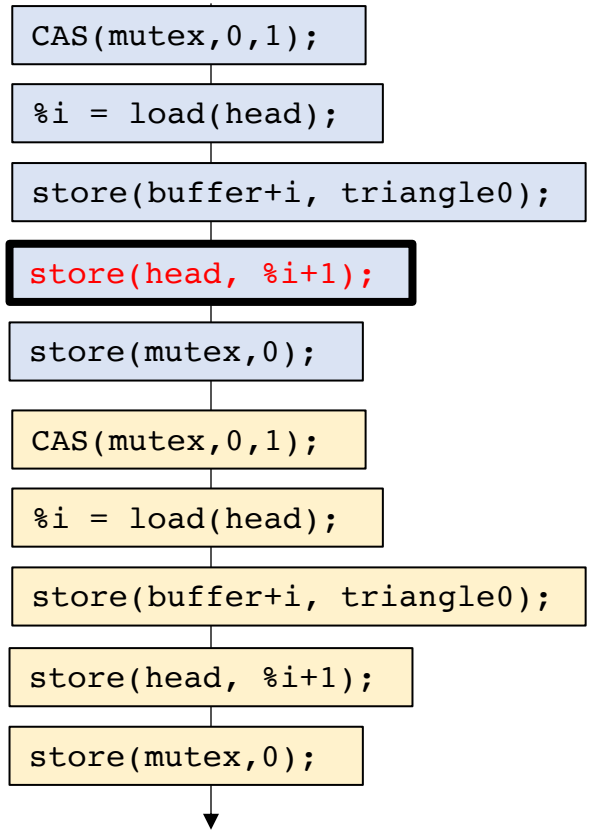
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*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!*

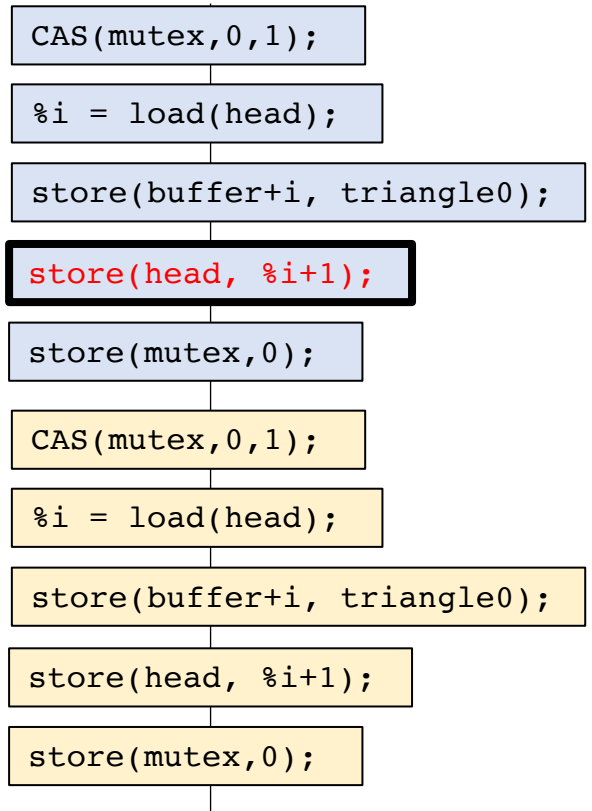
*Thread 1:*

```
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!*

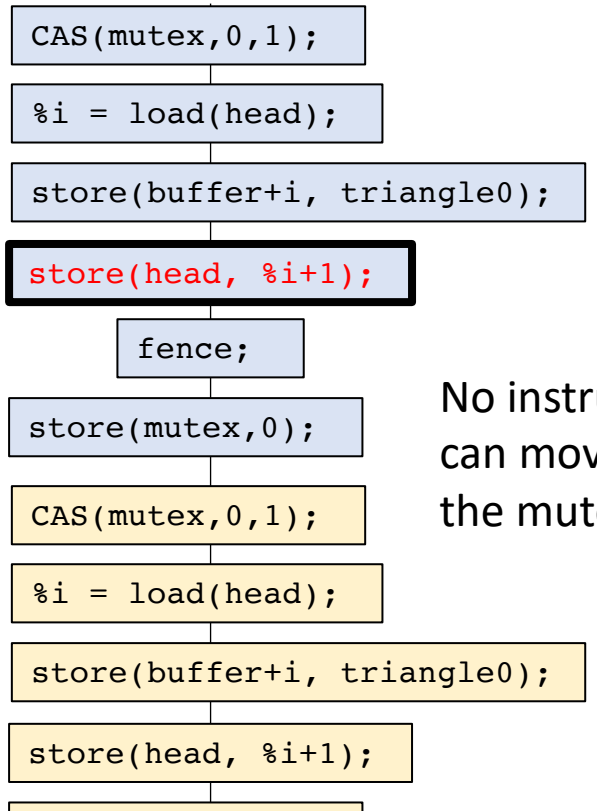
*Thread 1:*

```
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?

TSO

| | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | NO |

PSO

| | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | Different address |

RMO

| | L | S |
|---|---|---|
| L | YES | Different address |
| S | Different address | Different address |

# Memory consistency in the real world

- Historic Chips:
  - X86: TSO
    - Surprising robost
    - 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

    - Very difficult to write correct code under! PPoPP example

# 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

# Memory consistency in the real world

- Modern CPUs:
  - RISC-V : two specs: one similar to TSO, one similar to RMO
  - Apple M1: toggles between TSO and weaker
- GPUs?
  - Metal only provides relaxed atomics
  - Vulkan does not provide any fences that provide S - L ordering
  - We recently showed that Intel/AMD/Nvidia GPUs exhibit RMO behaviors
    - *Does not appear frequently in normal testing, but susceptible to side-channel attacks*

# Finished memory models

- Really interesting area!
  - lots of complicated behaviors
  - new chips/languages are exploring new models
  - constant navigation between flexible hardware and programmability

# Schedule

- More Memory Model Examples
- Compiling Memory Models
- **Barrier Specification**
- Barrier Implementation

# Barriers

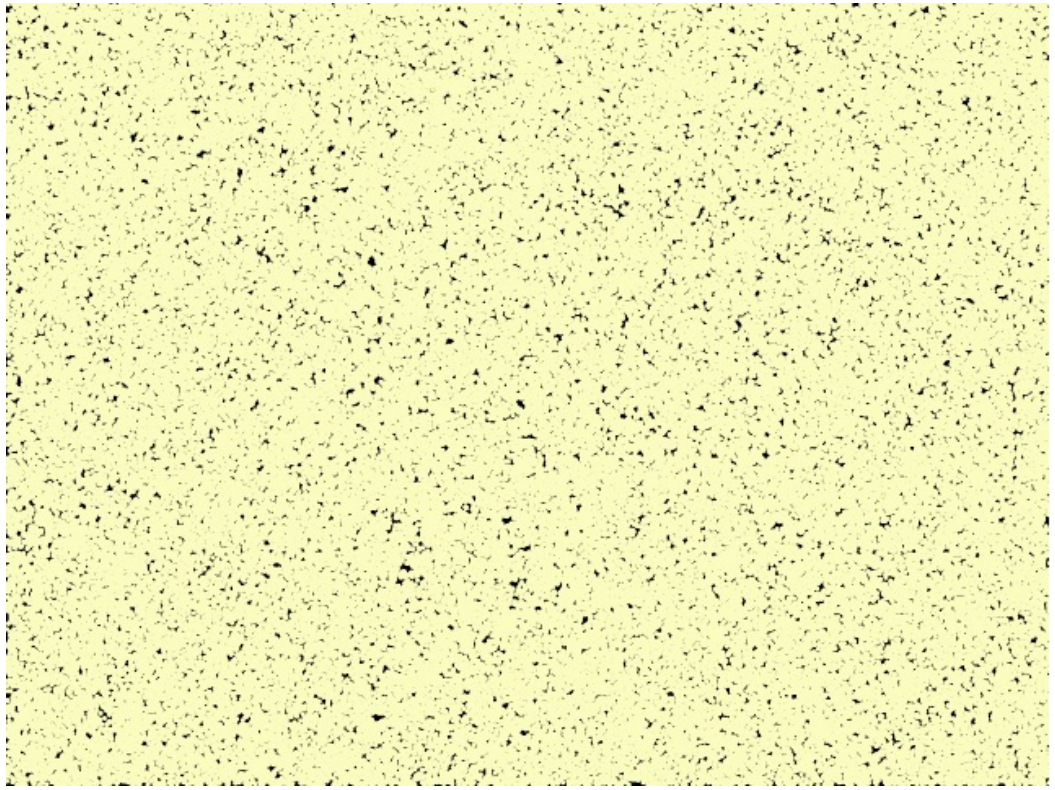
- Why do barriers fit into this module: “Reasoning About Parallel Computing”?
  - Relaxed Memory Models make reasoning about parallel computing HARD
  - Barriers make it EASIER (at the cost of performance potentially)

- A barrier is a concurrent object (like a mutex):
  - Only one method: `barrier` (called `await` in the book)

- Separates computational phases

# Barrier Examples

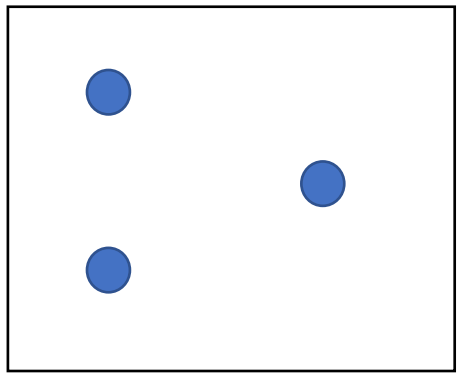
My current favorite: particle simulation



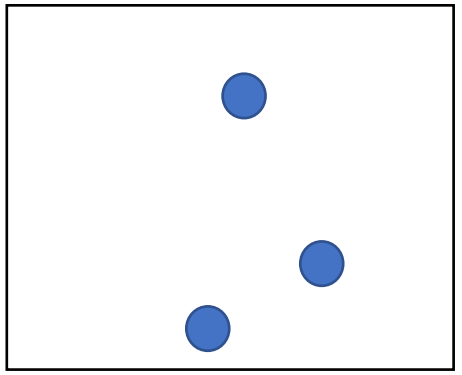
by Yanwen Xu

# Barrier Examples

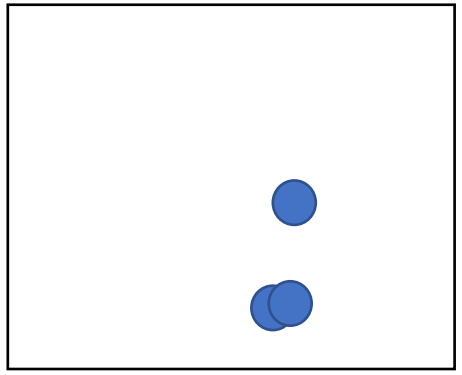
My current favorite: particle simulation



time = 0



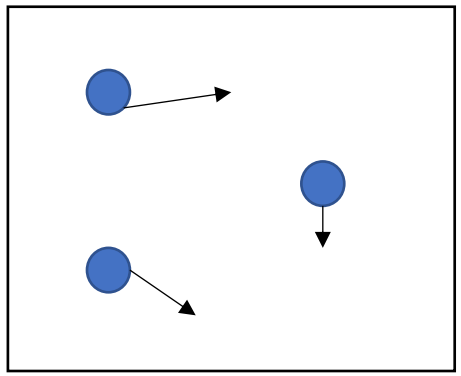
time = 1



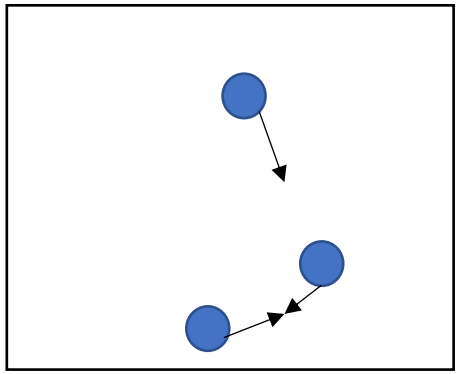
time = 2

# Barrier Examples

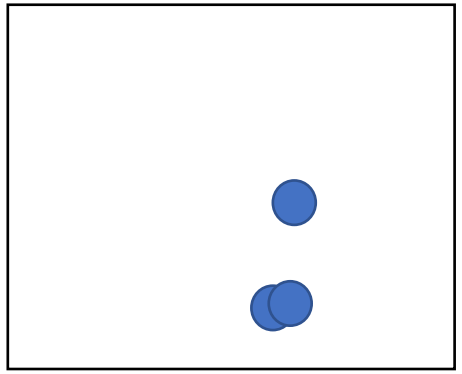
My current favorite: particle simulation



time = 0



time = 1

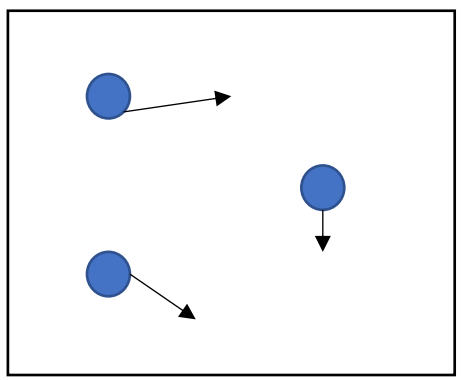


time = 2

at each time, compute
new positions for each particle
(in parallel)

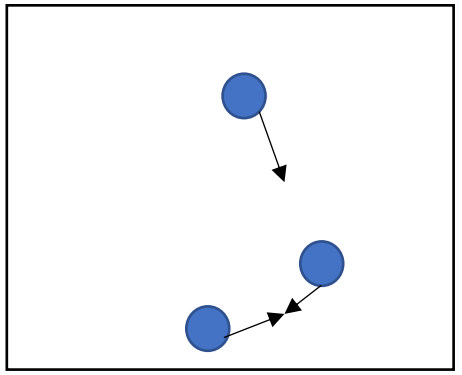
# Barrier Examples

My current favorite: particle simulation



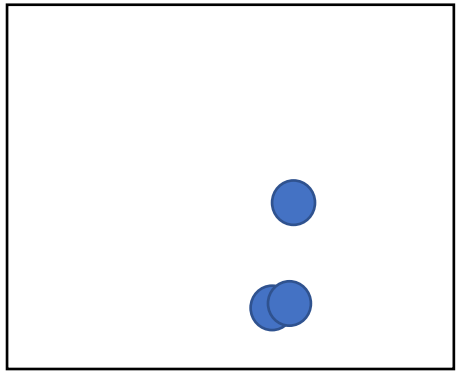
time = 0

```
barrier();
```



time = 1

```
barrier();
```



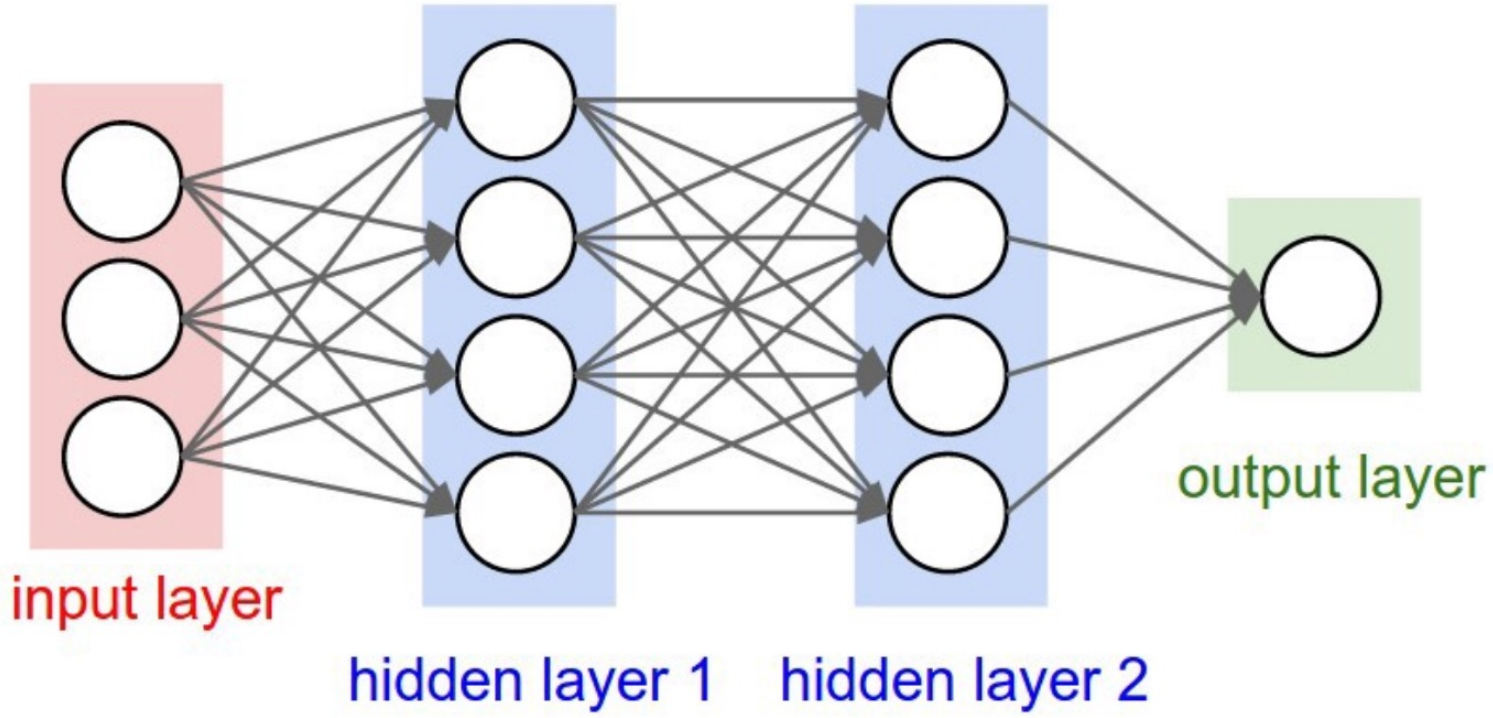
time = 2

at each time, compute
new positions for each particle
(in parallel)

But you need to wait for all particles to be
computed before starting the next time step

# Barrier Examples

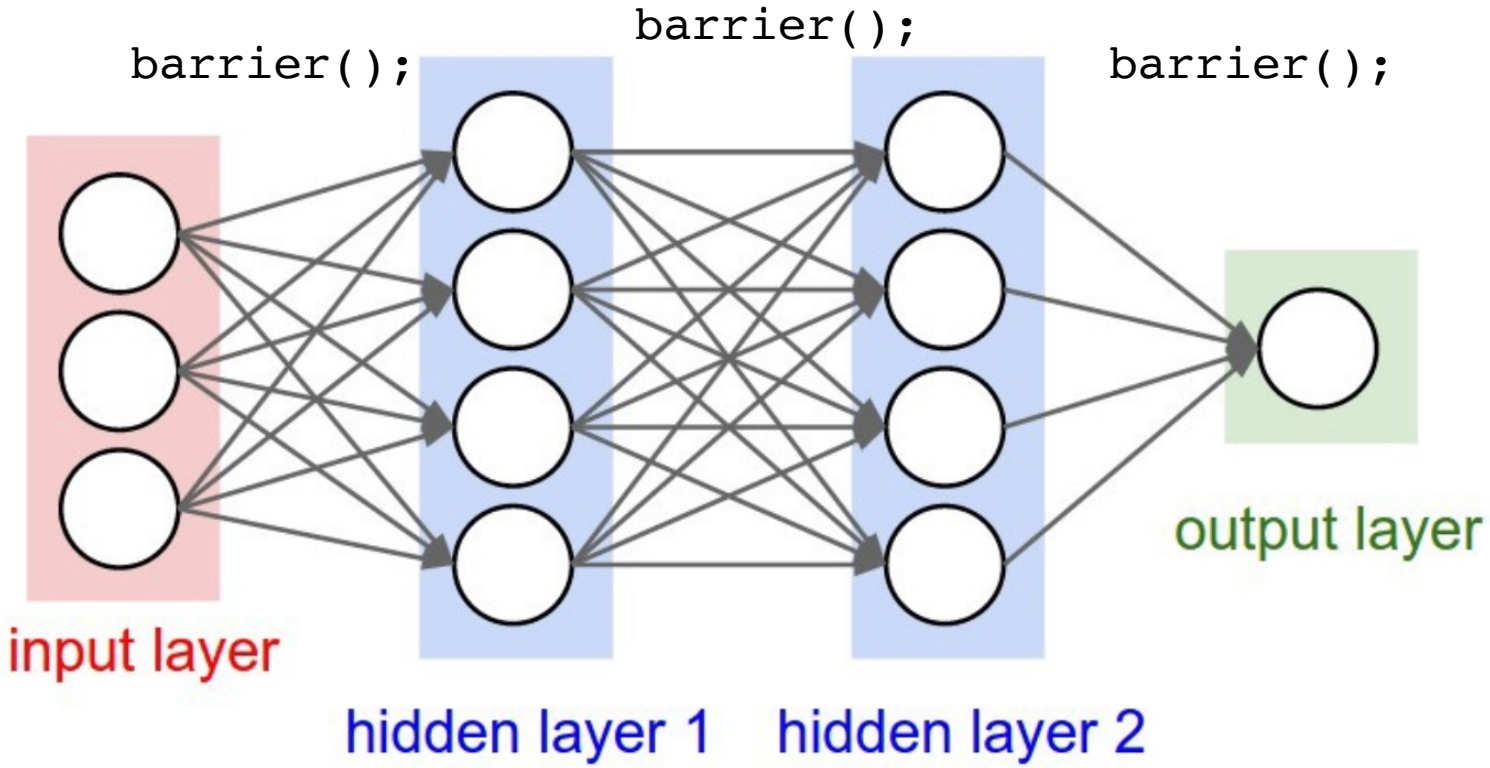
- Deep neural networks




from http://cs231n.stanford.edu/

# Barrier Examples

- Deep neural networks




from http://cs231n.stanford.edu/

# Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

# Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

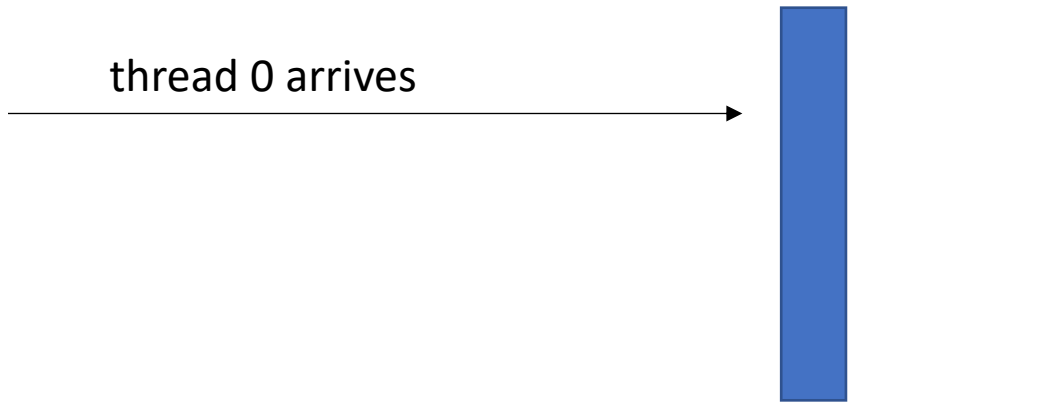
```
barrier();
```

# Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

```
barrier();
```

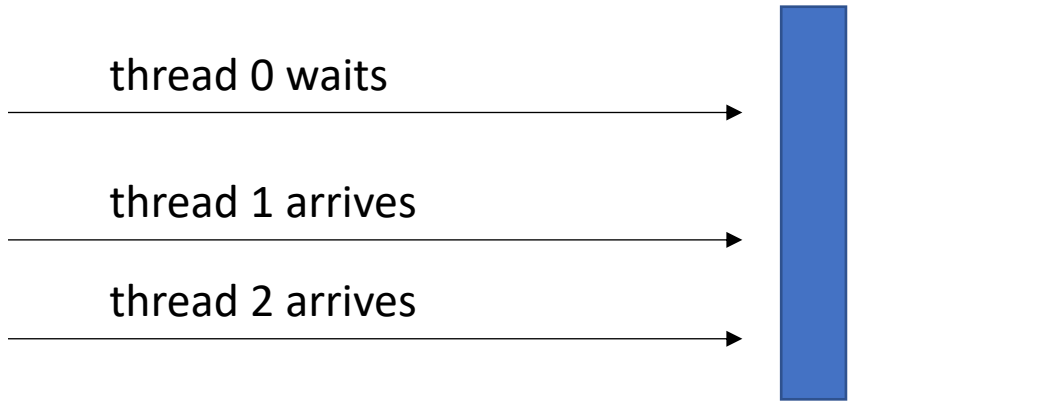


# Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

```
barrier();
```

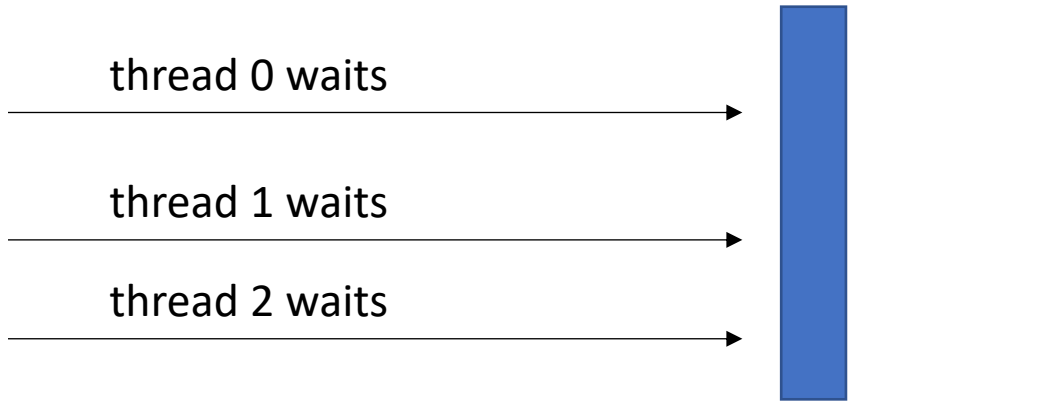


# Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

```
barrier();
```

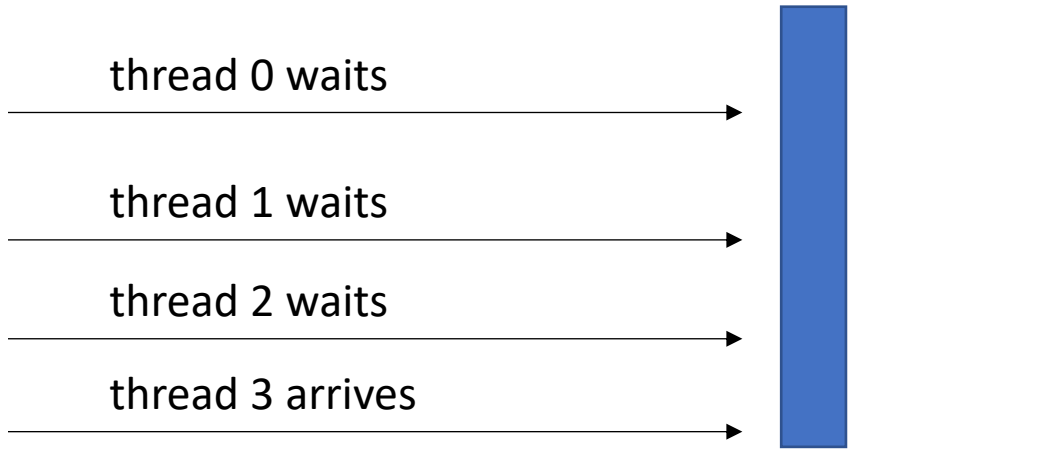


# Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

```
barrier();
```

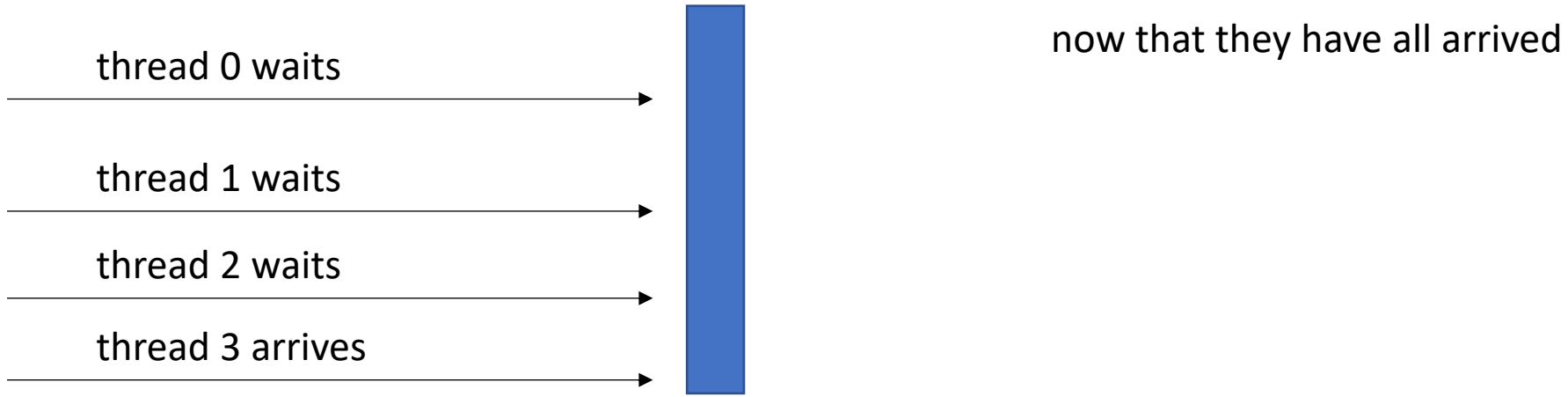


# Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

```
barrier();
```

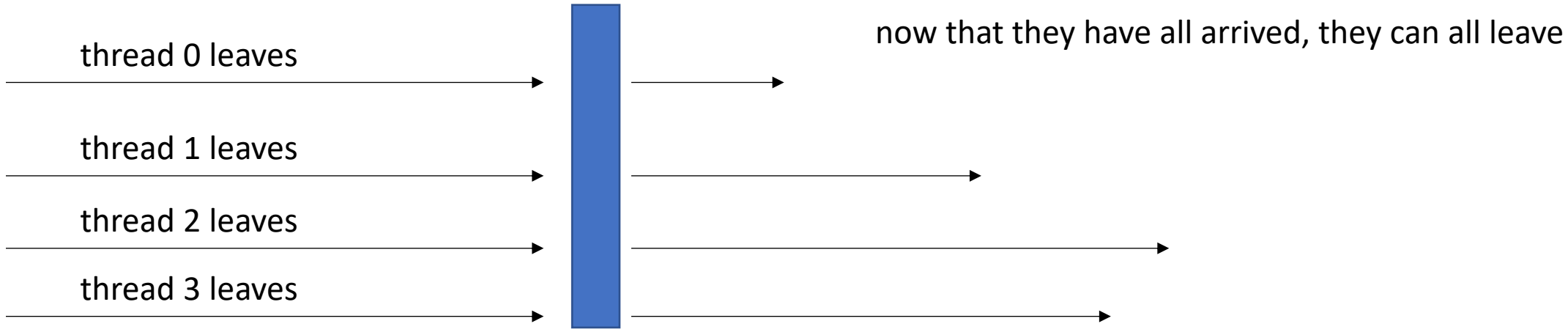


# Barriers

- Intuition: threads stop and wait for each other:
  - Threads ***arrive*** at the barrier
  - Threads ***wait*** at the barrier
  - Threads ***leave*** the barrier once all other threads have arrived

Example: say there are 4 threads:

```
barrier();
```



A more formal specification

Given a global barrier `B`
and a global memory location x where
initially *x = 0;

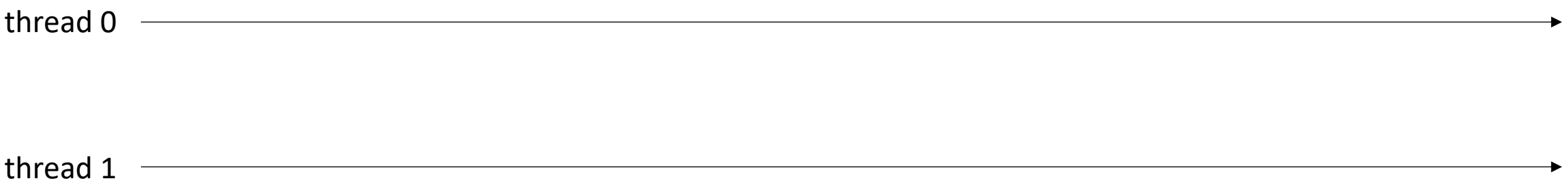
First, what would we expect
var to be after this program?

*Thread 0:*

```
*x = 1;
B.barrier();
```

*Thread 1:*

```
B.barrier();
var = *x;
```



A more formal specification

Given a global barrier B
and a global memory location x where
initially *x = 0;

*Thread 0:*

```
*x = 1;
B.barrier();
```

*Thread 1:*

```
B.barrier();
var = *x;
```

gives an event:
barrier arrive

A more formal specification

Given a global barrier `B`
and a global memory location x where
initially *x = 0;

*Thread 0:*

```
*x = 1;
B.barrier();
```

*Thread 1:*

```
B.barrier();
var = *x;
```

gives an event:
barrier arrive

barrier arrive needs to wait for all threads
to arrive (similar to how a mutex request must wait for
another to release)

A more formal specification

Given a global barrier B
and a global memory location x where
initially *x = 0;

*Thread 0:*

```
*x = 1;
B.barrier();
```

*Thread 1:*

```
B.barrier();
var = *x;
```

A more formal specification

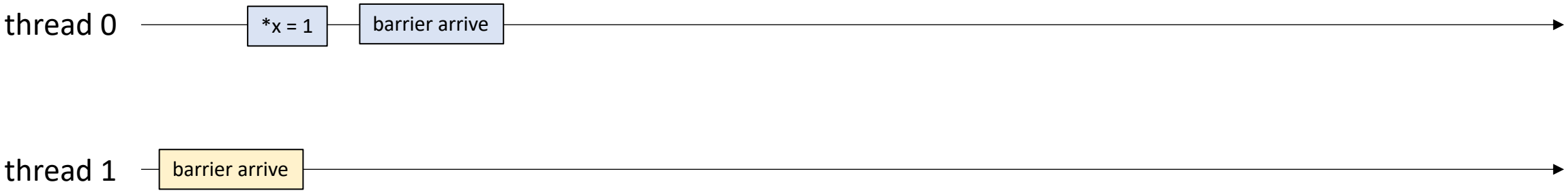
Given a global barrier `B`
and a global memory location x where initially *x = 0;

*Thread 0:*

```
*x = 1;
B.barrier();
```

*Thread 1:*

```
B.barrier();
var = *x;
```



A more formal specification

Given a global barrier `B`
and a global memory location x where
initially *x = 0;

*Thread 0:*

```
*x = 1;
B.barrier();
```

*Thread 1:*

```
B.barrier();
var = *x;
```

now that all threads have arrived:
They can leave (1 event at the same time)

A more formal specification

Given a global barrier `B`
and a global memory location x where initially *x = 0;

*Thread 0:*

```
*x = 1;
B.barrier();
```

*Thread 1:*

```
B.barrier();
var = *x;
```

This finishes the barrier execution

A more formal specification

Given a global barrier `B`
and a global memory location x where initially *x = 0;

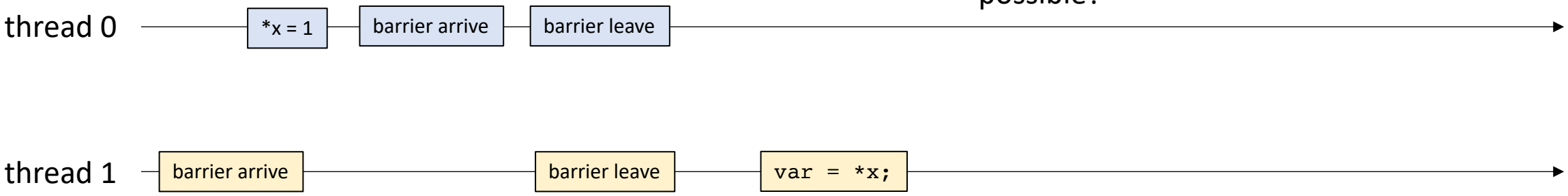
*Thread 0:*

```
*x = 1;
B.barrier();
```

*Thread 1:*

```
B.barrier();
var = *x;
```

what value must this read? Any other value possible?



One more example, assume initially `*x = *y = 0`

*Thread 0:*

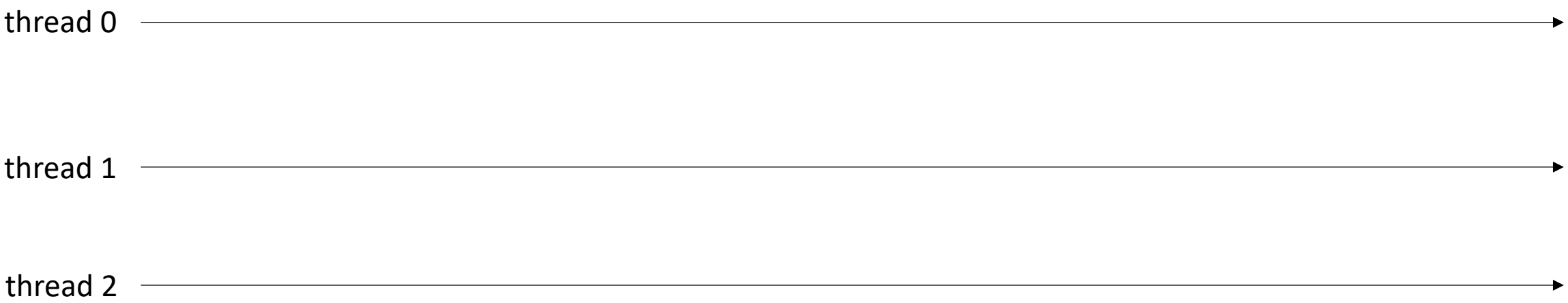
```
*x = 1;
B.barrier();
```

*Thread 1:*

```
*y = 2;
B.barrier();
```

*Thread 2:*

```
B.barrier();
var = *x + *y;
```



One more example, assume initially `*x = *y = 0`

*Thread 0:*

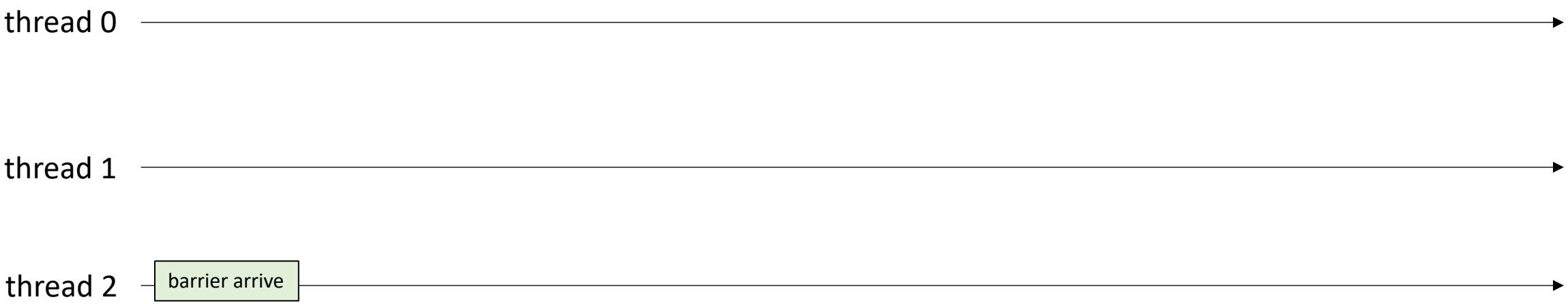
```
*x = 1;
B.barrier();
```

*Thread 1:*

```
*y = 2;
B.barrier();
```

*Thread 2:*

```
B.barrier();
var = *x + *y;
```



One more example, assume initially `*x = *y = 0`

*Thread 0:*

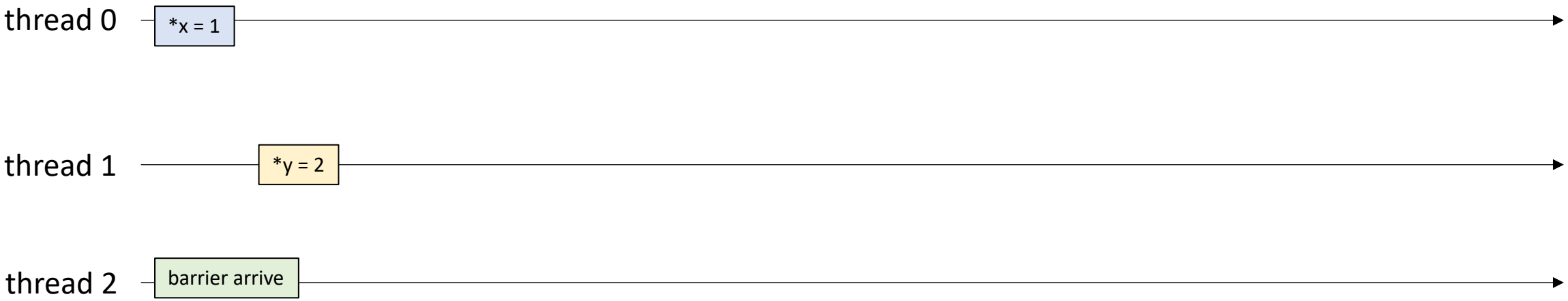
```
*x = 1;
B.barrier();
```

*Thread 1:*

```
*y = 2;
B.barrier();
```

*Thread 2:*

```
B.barrier();
var = *x + *y;
```



One more example, assume initially `*x = *y = 0`

*Thread 0:*

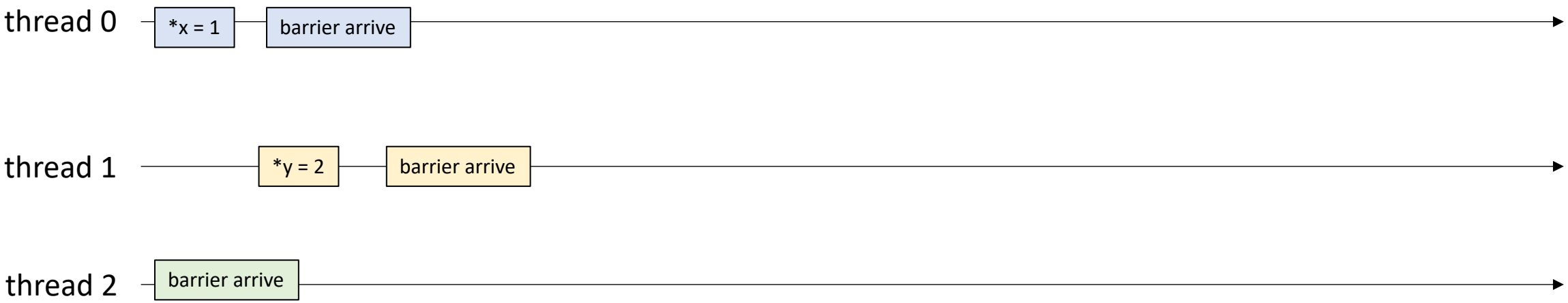
```
*x = 1;
B.barrier();
```

*Thread 1:*

```
*y = 2;
B.barrier();
```

*Thread 2:*

```
B.barrier();
var = *x + *y;
```



One more example, assume initially `*x = *y = 0`

*Thread 0:*

```
*x = 1;
B.barrier();
```

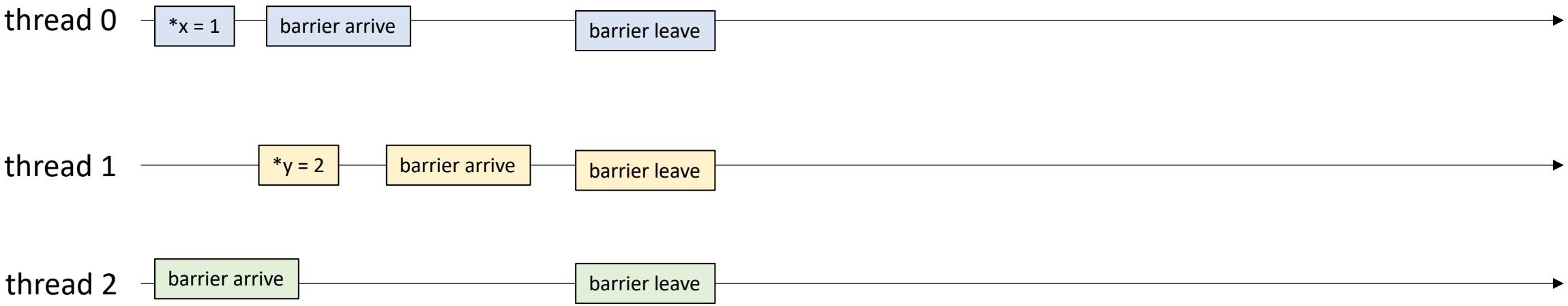
*Thread 1:*

```
*y = 2;
B.barrier();
```

*Thread 2:*

```
B.barrier();
var = *x + *y;
```

They've all arrived



One more example, assume initially `*x = *y = 0`

*Thread 0:*

```
*x = 1;
B.barrier();
```

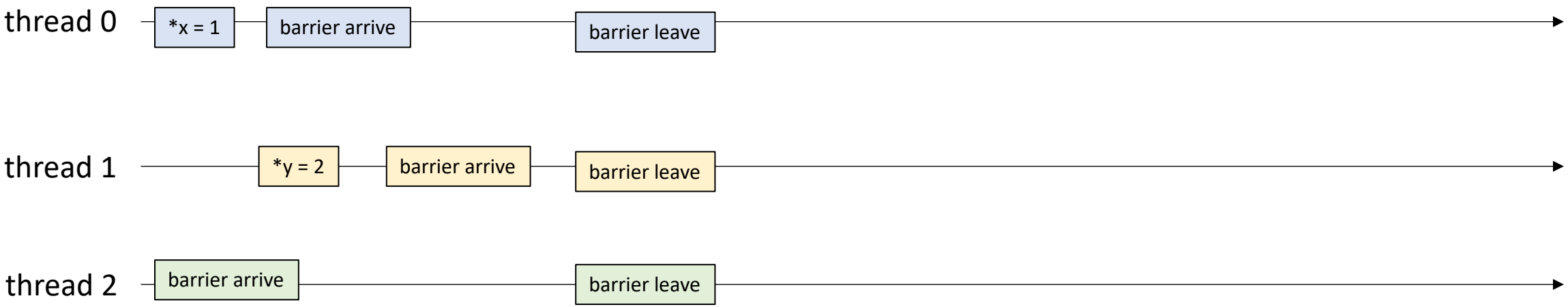
*Thread 1:*

```
*y = 2;
B.barrier();
```

*Thread 2:*

```
B.barrier();
var = *x + *y;
```

They've all arrived



One more example, assume initially `*x = *y = 0`

*Thread 0:*

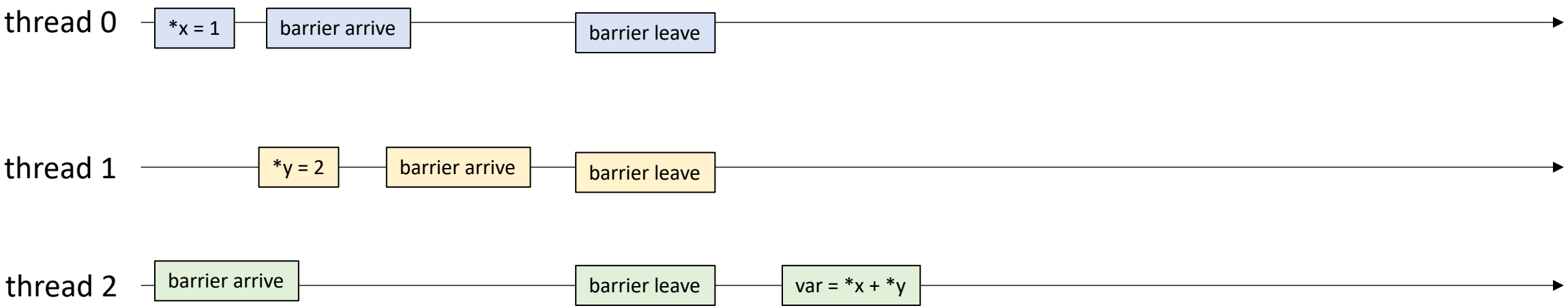
```
*x = 1;
B.barrier();
```

*Thread 1:*

```
*y = 2;
B.barrier();
```

*Thread 2:*

```
B.barrier();
var = *x + *y;
```




What is this guaranteed to be?

One more example, assume initially `*x = *y = 0`

*Thread 0:*

```
*x = 1;
B.barrier();
```

*Thread 1:*

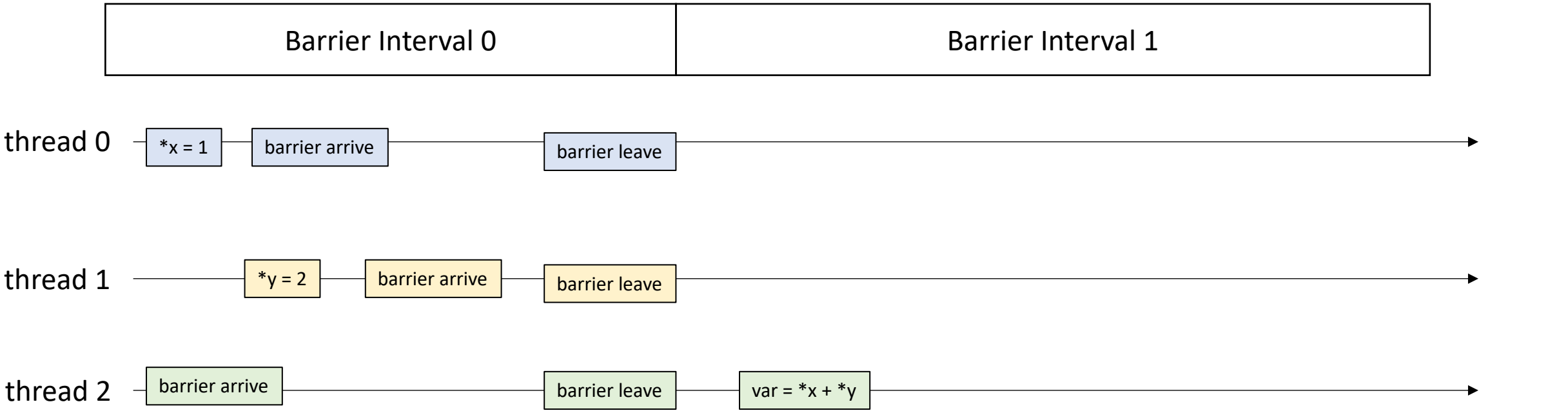
```
*y = 2;
B.barrier();
```

*Thread 2:*

```
B.barrier();
var = *x + *y;
```

sometimes called a *phase*

extending to the next *barrier leave*



# Barriers

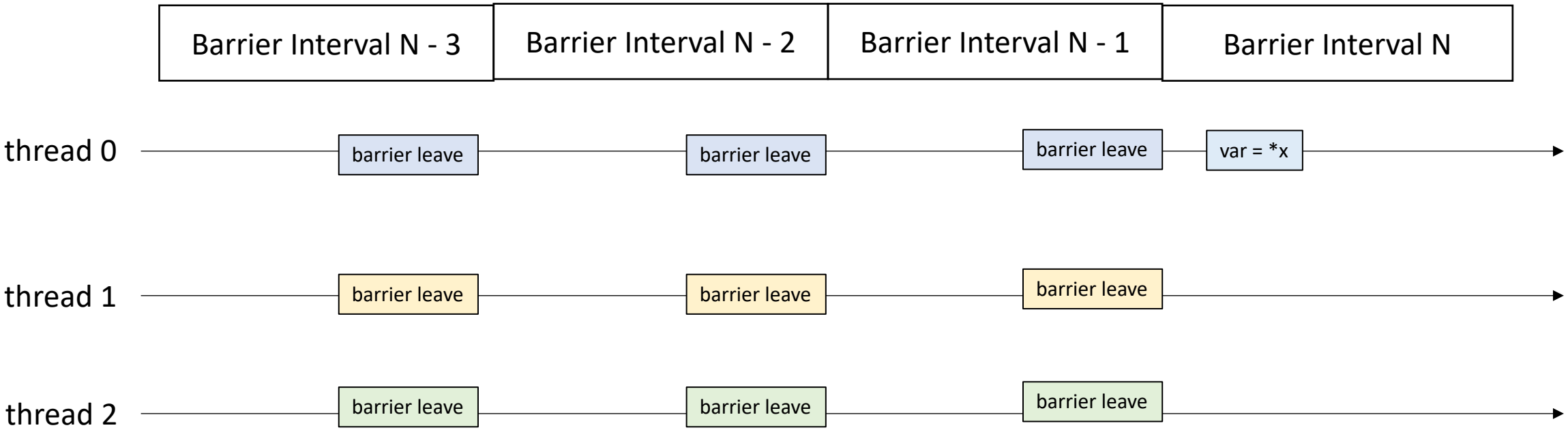
- Barrier Property:
  - If the only concurrent object you use in your program is a barrier (no mutexes, concurrent data-structures, atomic accesses)
  - If every barrier interval contains no data conflicts, then

    ***your program will be deterministic (only 1 outcome allowed)***

  - much easier to reason about ☺

Assume we are reading from x

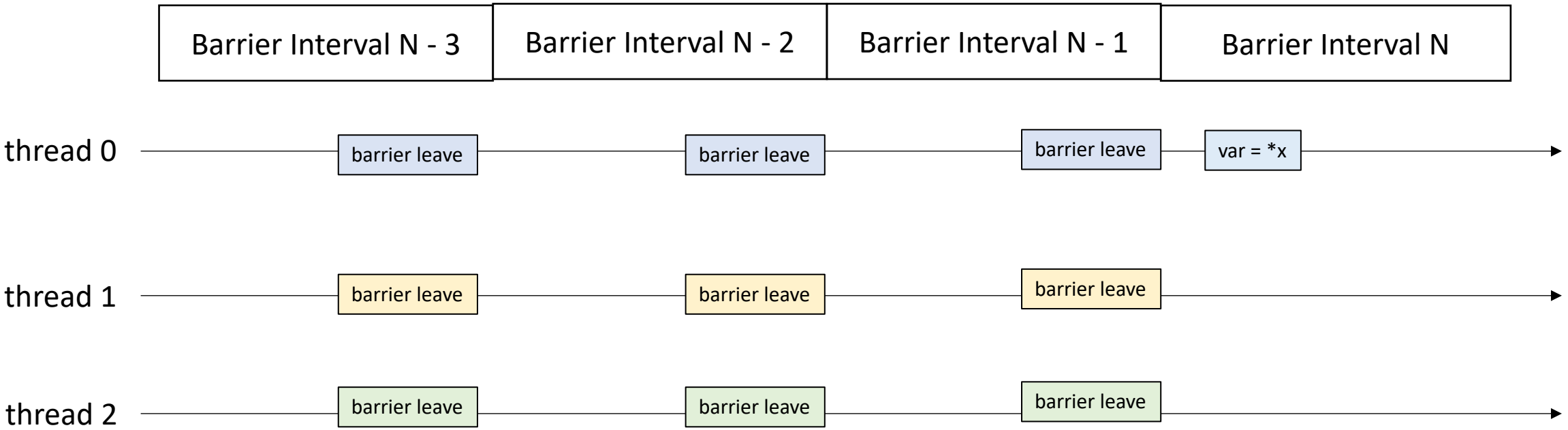
We are only allowed to return one possible value



no data conflicts means that x is written to at most once per barrier interval

Assume we are reading from x

We are only allowed to return one possible value

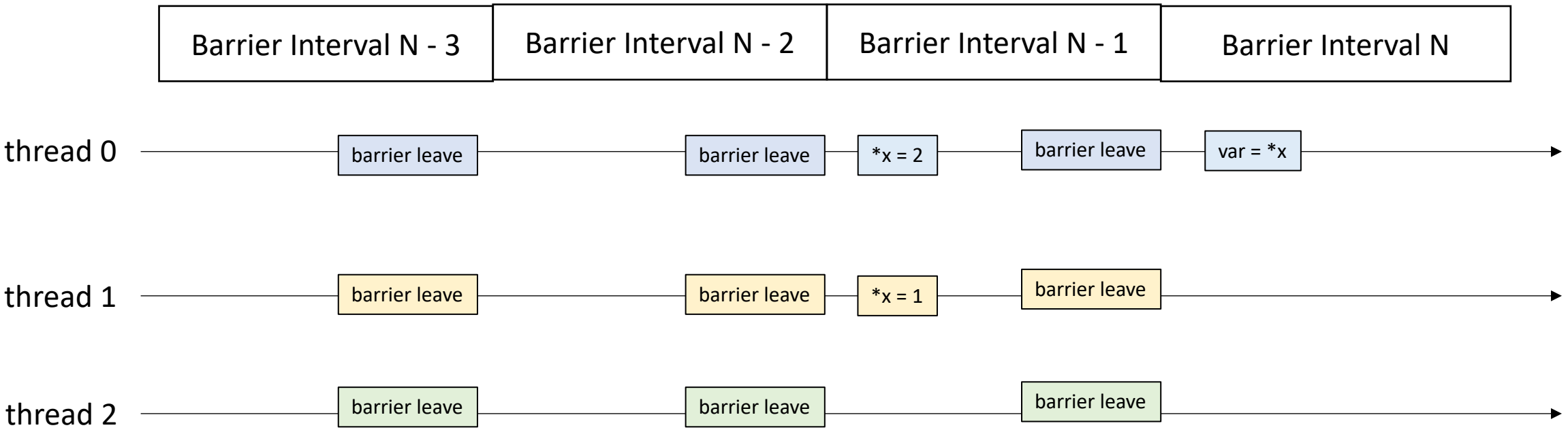


no data conflicts means that x is written to at most once per barrier interval

Assume we are reading from x

We are only allowed to return one possible value

not allowed

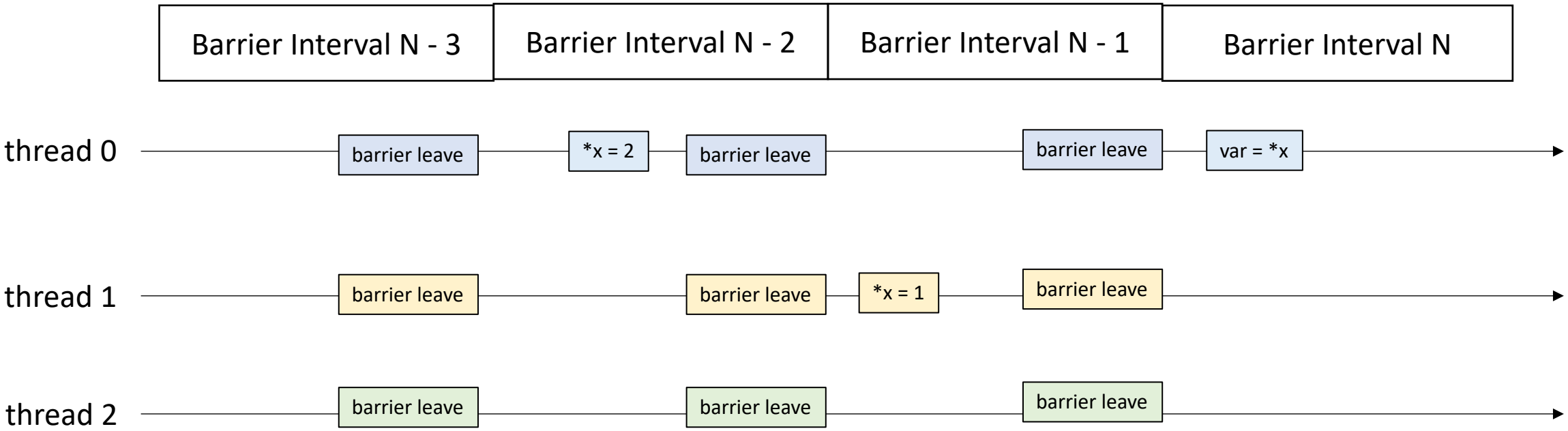


no data conflicts means that x is written to at most once per barrier interval

we will read from the write from the most recent barrier interval

Assume we are reading from x

We are only allowed to return one possible value



# Schedule

- More Memory Model Examples
- Compiling Memory Models
- Barrier Specification
- **Barrier Implementation**

# Barrier Implementation

- First attempt at implementation

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        // ??
     }

}
```

# Barrier Implementation

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        // What next?
     }

}
```

# Barrier Implementation

First handle the case where the thread is the last thread to arrive

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
        }
        // What next?
     }

}
```

# Barrier Implementation

Spin while there is a thread waiting at the barrier

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
     }

}
```

# Barrier Implementation

Spin while there is a thread waiting at the barrier

Does this work?

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
     }

}
```

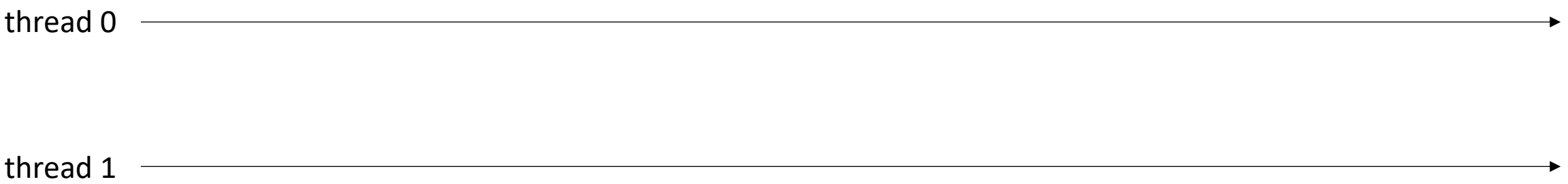
*Thread 0:*

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

*Thread 1:*

```
B.barrier();
B.barrier();
```



num_threads == 2

*Thread 0:*

```
B.barrier();
B.barrier();
```

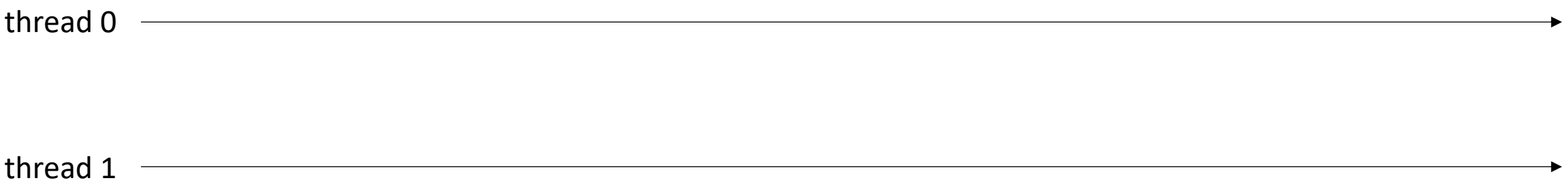
```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

*Thread 1:*

```
B.barrier();
B.barrier();
```

arrival_num = 2

arrival_num = 1



num_threads == 2
counter == 2

*Thread 0:*

```
B.barrier();
B.barrier();
```

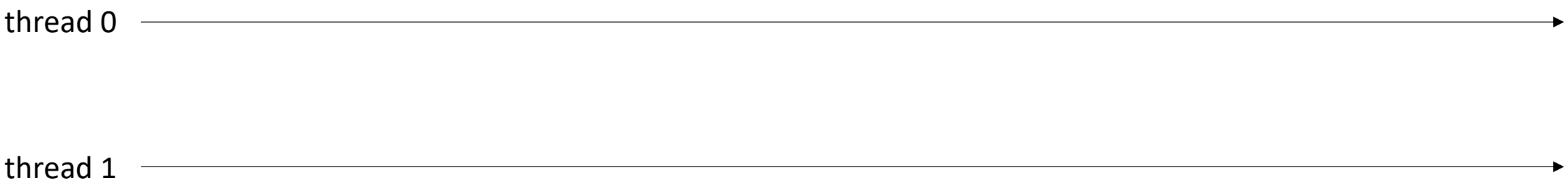
```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

*Thread 1:*

```
B.barrier();
B.barrier();
```

arrival_num = 2

arrival_num = 1



num_threads == 2
counter == 0

*Thread 0:*

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

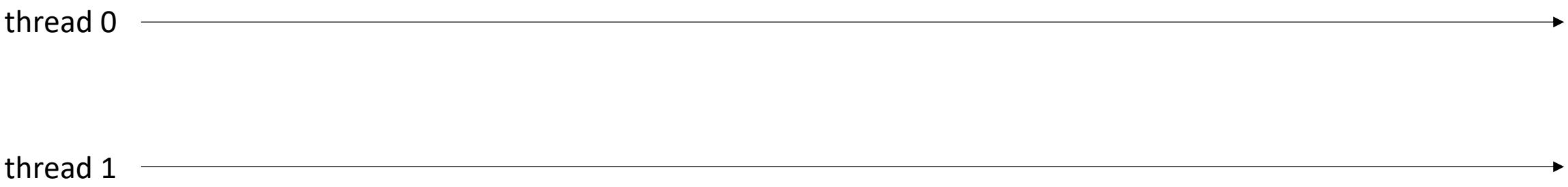
*Thread 1:*

```
B.barrier();
B.barrier();
```

Leaves barrier

arrival_num = 1

in a perfect world,
thread 1 executes now and leaves the barrier



```
num_threads == 2
counter == 0
```

*Thread 0:*

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

*Thread 1:*

```
B.barrier();
B.barrier();
```

Leaves barrier

arrival_num = 1

in a perfect world,
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it was asleep?

num_threads == 2
counter == 0

*Thread 0:*

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

*Thread 1:*

```
B.barrier();
B.barrier();
```

enters next barrier

arrival_num = 1

in a perfect world,
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it was asleep?

num_threads == 2
counter == 1

*Thread 0:*

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

*Thread 1:*

```
B.barrier();
B.barrier();
```

arrival_num == 1

arrival_num = 1

in a perfect world,
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it was asleep?

num_threads == 2
counter == 1

*Thread 0:*

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

*Thread 1:*

```
B.barrier();
B.barrier();
```

*Thread 1 wakes up! Doesn't think its missed anything*

arrival_num == 1

arrival_num = 1

in a perfect world,
thread 1 executes now and leaves the barrier

num_threads == 2
counter == 1

*Thread 0:*

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

*Thread 1:*

```
B.barrier();
B.barrier();
```

*Thread 1 wakes up! Doesn't think its missed anything*

arrival_num == 1

arrival_num = 1

in a perfect world,
thread 1 executes now and leaves the barrier

Both threads get stuck here!

*Thread 0:*

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

*Thread 1:*

```
B.barrier();
B.barrier();
```

Ideas for fixing?

Two different barriers that alternate?

*Thread 0:*

```
B0.barrier();
B1.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

*Thread 1:*

```
B0.barrier();
B1.barrier();
```

Ideas for fixing?

Two different barriers that alternate?

Pros: simple to implement

Cons: user has to alternate barriers

*Thread 0:*

```
B0.barrier();
B1.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

*Thread 1:*

```
B0.barrier();
B1.barrier();
```

Ideas for fixing?

Two different barriers that alternate?

Pros: simple to implement

Cons: user has to alternate barriers

```
B.barrier();
if (...) {
  B.barrier();
}
B.barrier();
```

How to alternate these calls?

# Sense Reversing Barrier

- Book Chapter 17

# Next week

- Guest lecture; don’t miss it!

- Office hours:
  - First hour will be by sign-up sheet
  - Second hour will be open to discuss homework

- HW3 is due on Friday