Schedule

- Module 4 introduction

- **Barriers**
  - **Specification**
  - Implementation

Barriers

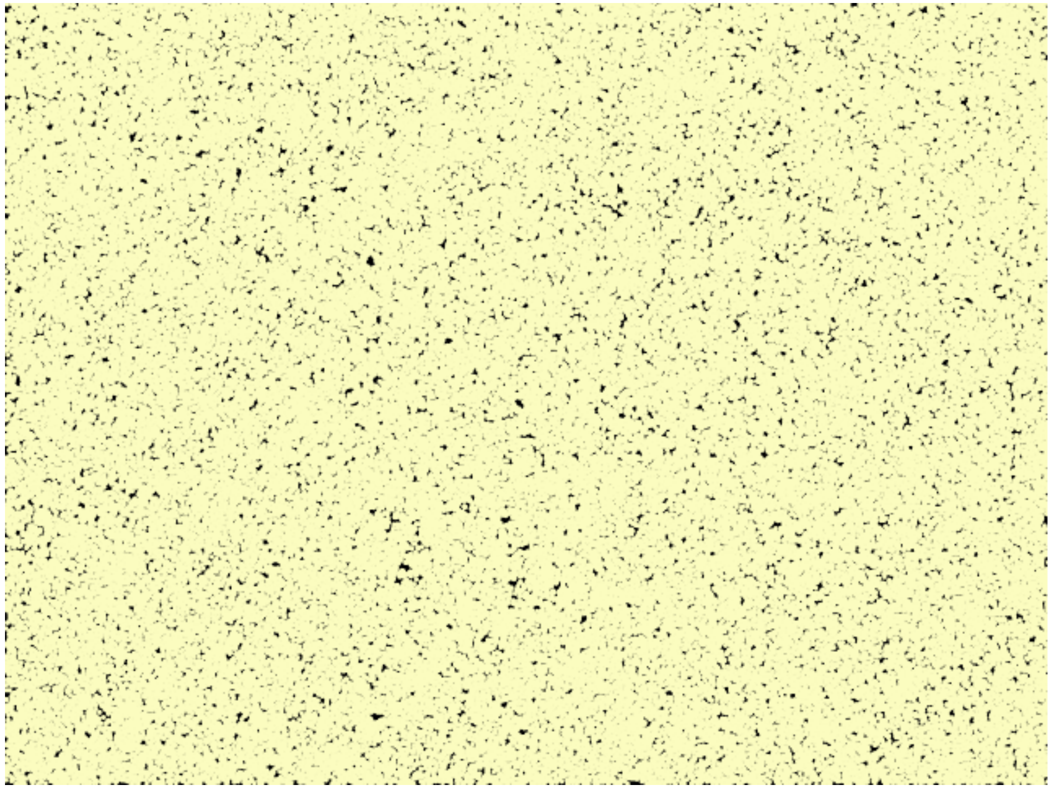
- Why do barriers fit into this module: “Reasoning About Parallel Computing”?
  - Relaxed Memory Models make reasoning about parallel computing HARD
  - Barriers make it EASIER (at the cost of performance potentially)

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

- Separates computational phases

Barrier Examples

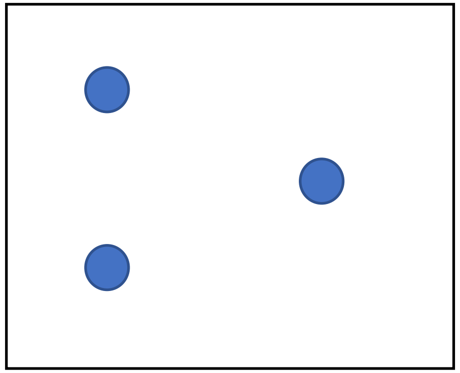
My current favorite: particle simulation



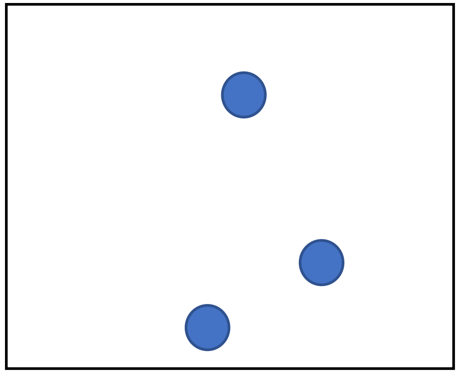
by Yanwen Xu

Barrier Examples

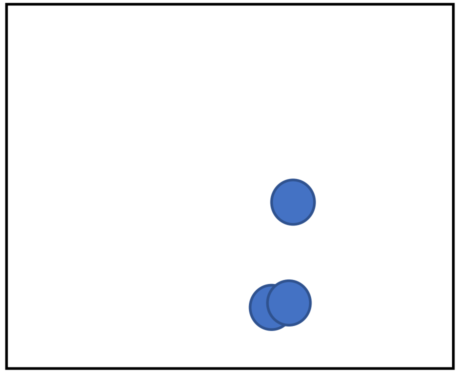
My current favorite: particle simulation



time = 0



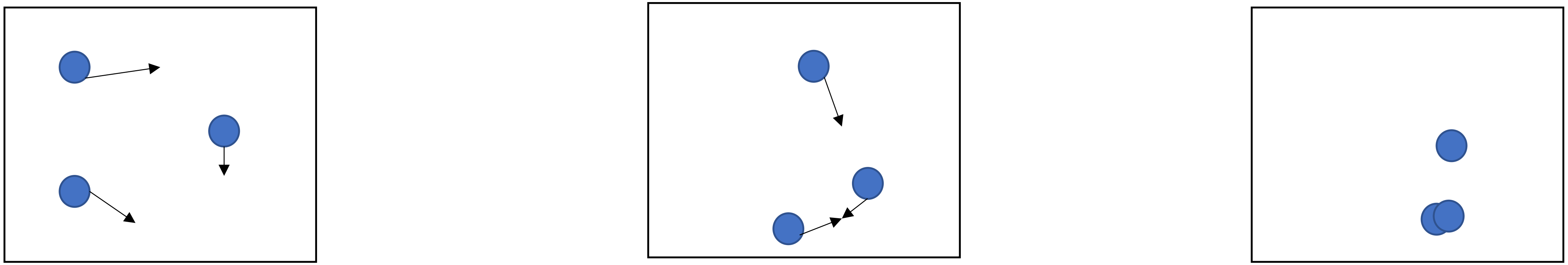
time = 1



time = 2

Barrier Examples

My current favorite: particle simulation



time = 0

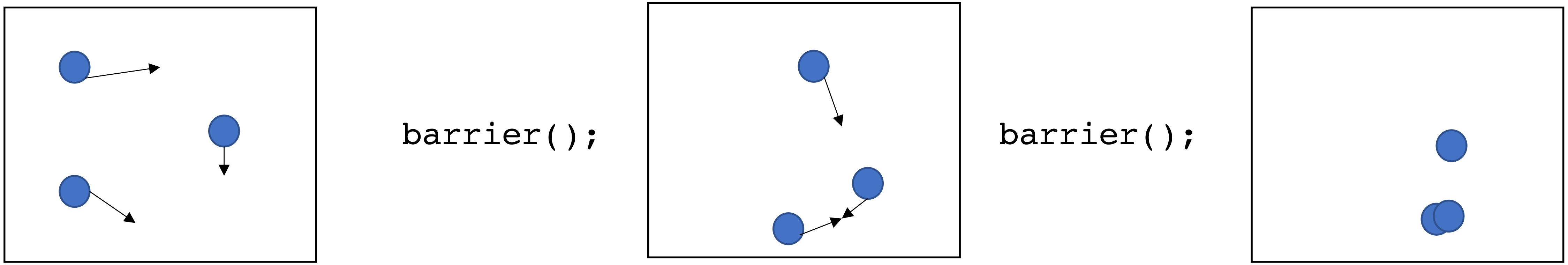
time = 1

time = 2

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

Barrier Examples

My current favorite: particle simulation



`barrier();`

`barrier();`

time = 0

time = 1

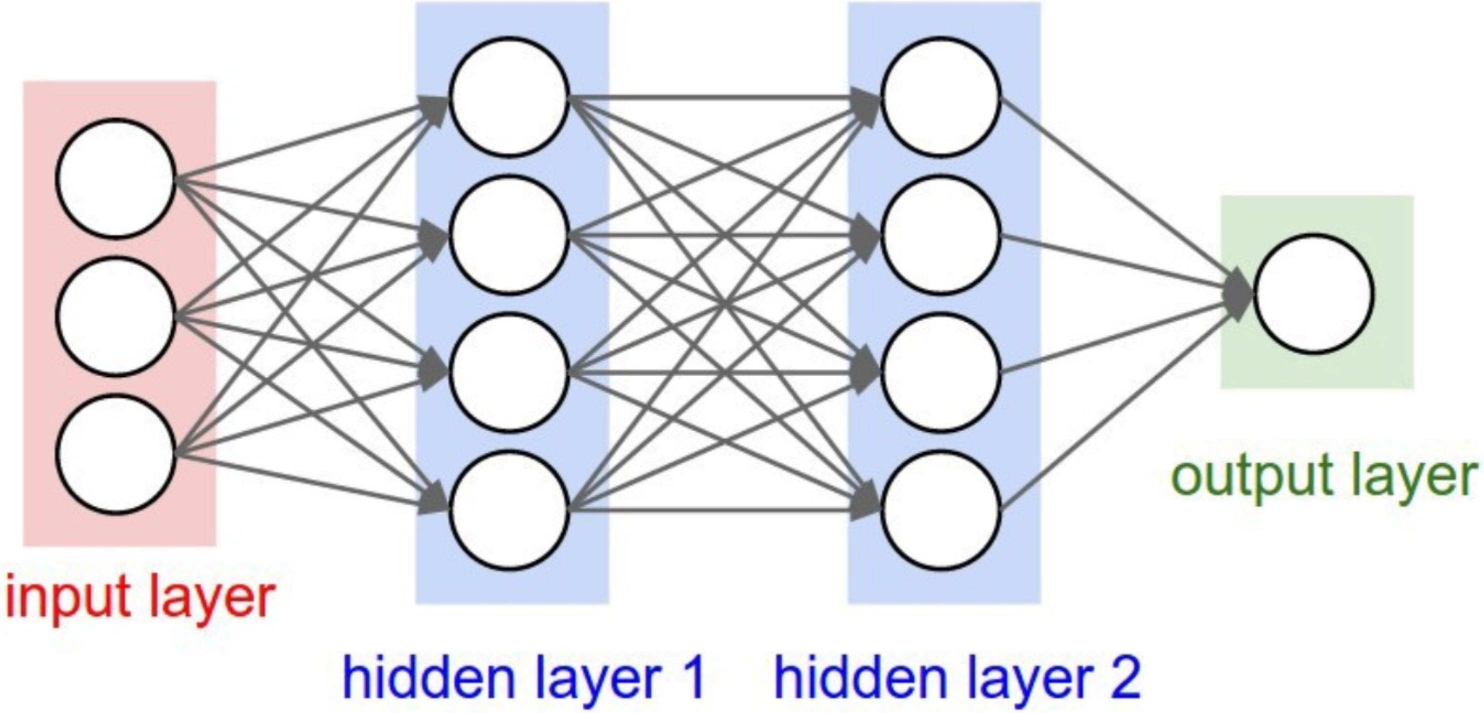
time = 2

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

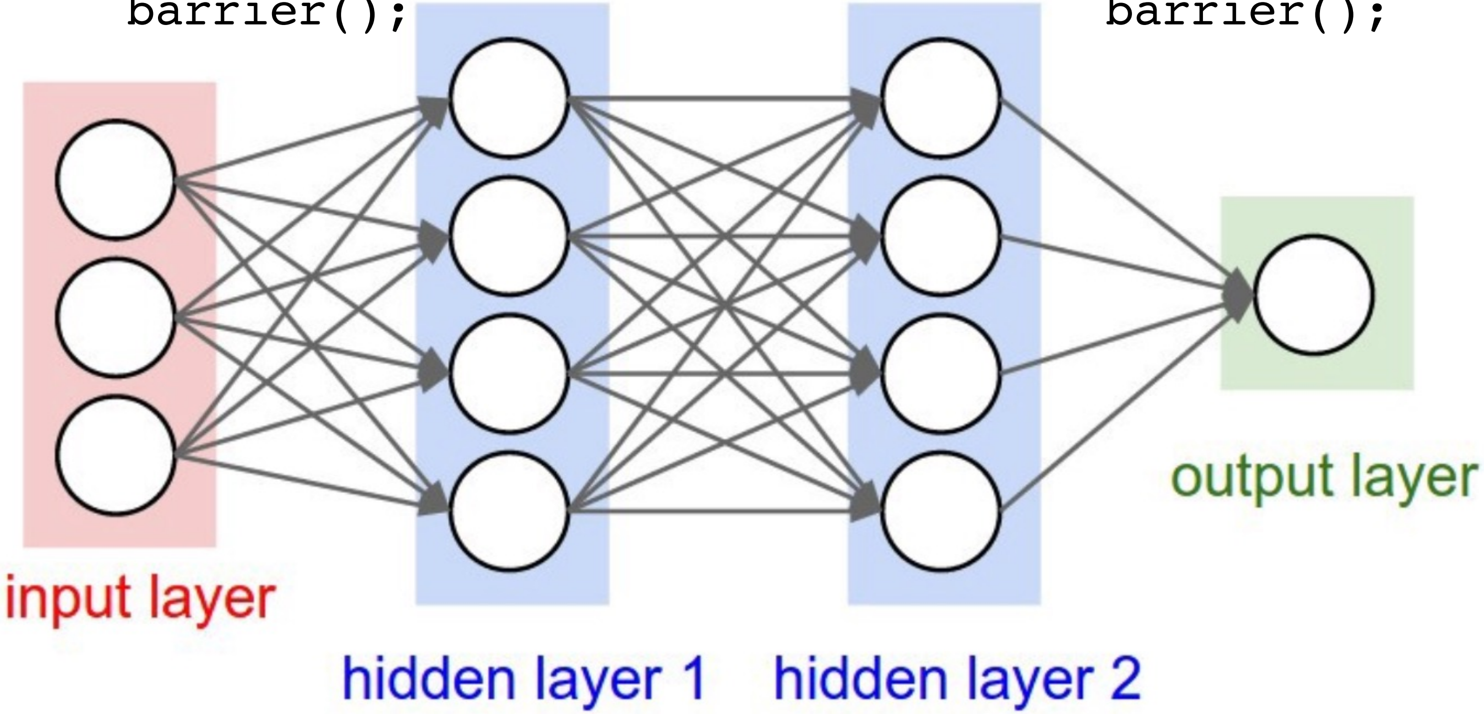


from http://cs231n.stanford.edu/

Barrier Examples

- Deep neural networks

```
barrier();          barrier();          barrier();
```



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:

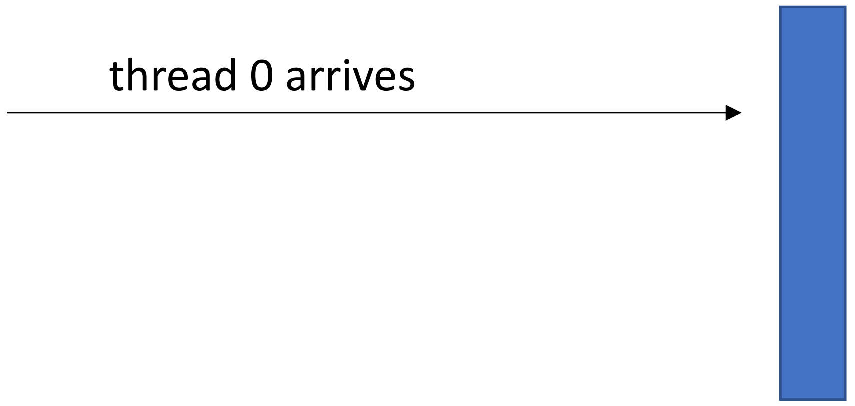
`barrier();`

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:

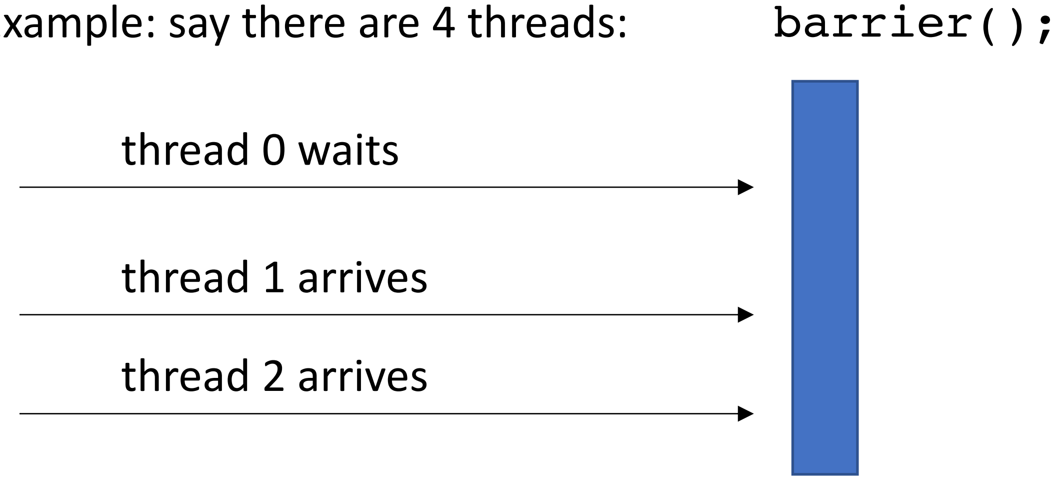
`barrier();`



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();`

thread 0 waits

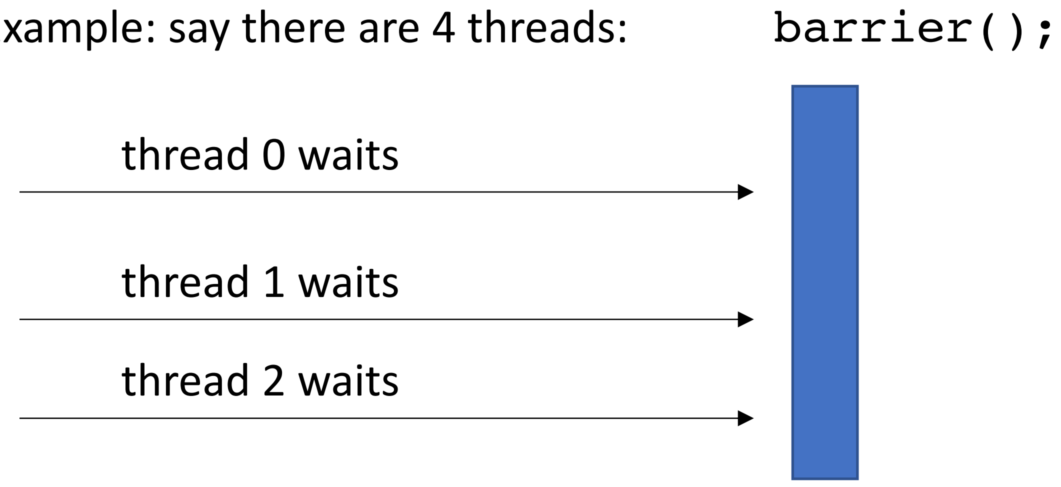
thread 1 arrives

thread 2 arrives

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:

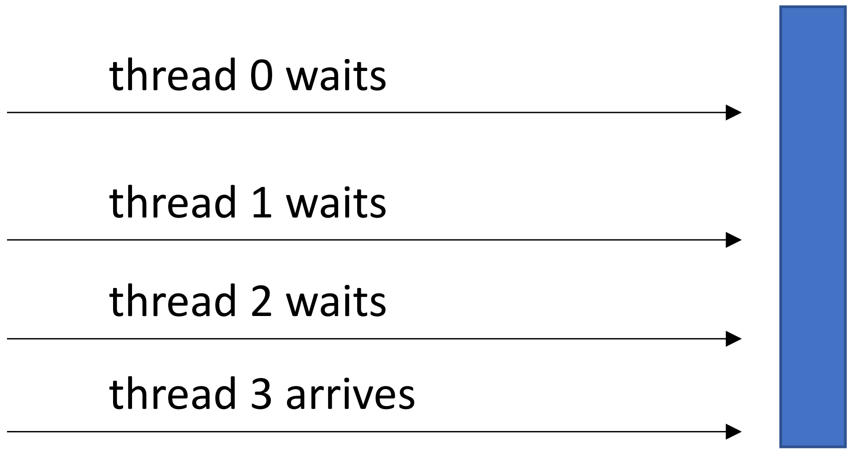


Barriers

- 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();`



thread 0 waits

thread 1 waits

thread 2 waits

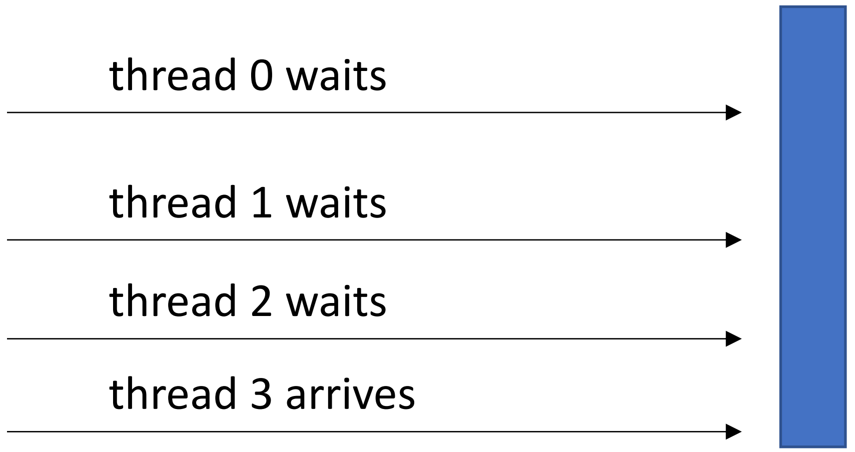
thread 3 arrives

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();
```



thread 0 waits

thread 1 waits

thread 2 waits

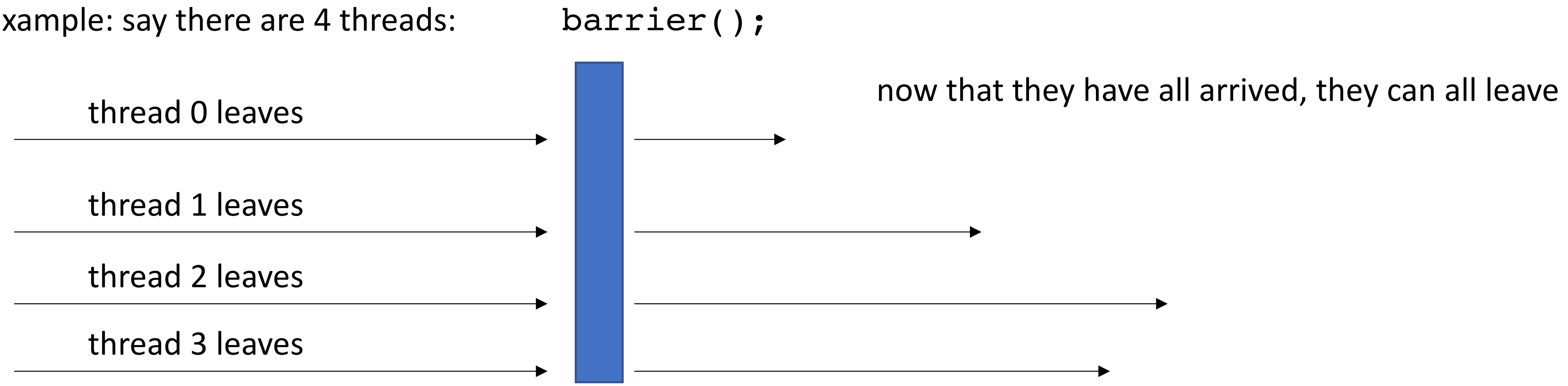
thread 3 arrives

now that they have all 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:



A more formal specification

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

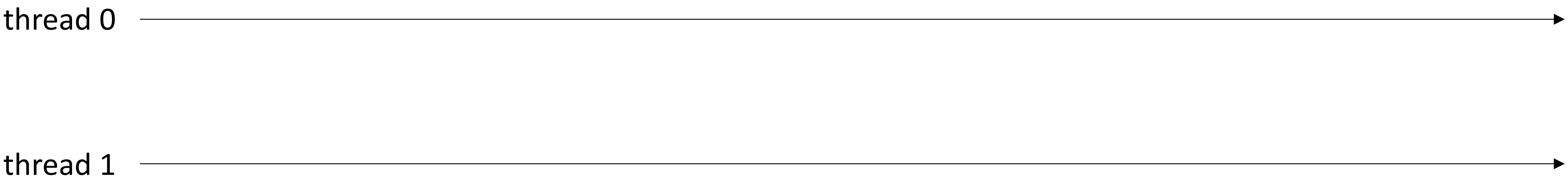
Thread 0:

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

Thread 1:

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

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



A more formal specification

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

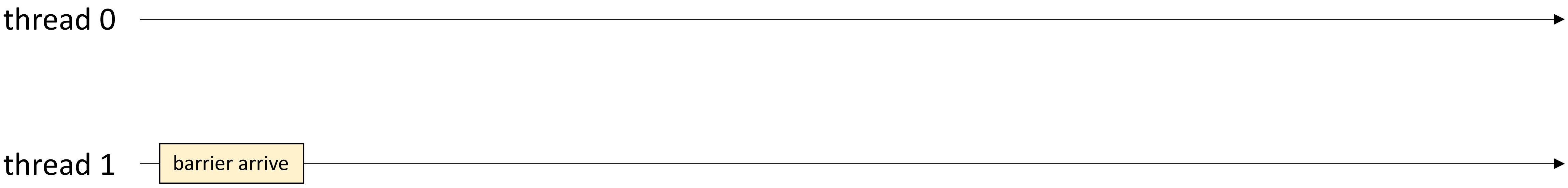
Thread 0:

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

Thread 1:

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

gives an event:
barrier arrive



thread 0

thread 1

barrier arrive

A more formal specification

Thread 0:

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

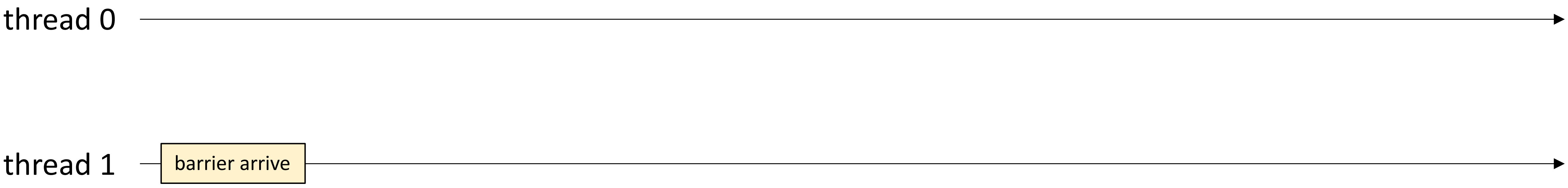
Given a global barrier B
and a global memory location x where
initially *x = 0;

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)



thread 0

thread 1

barrier arrive

A more formal specification

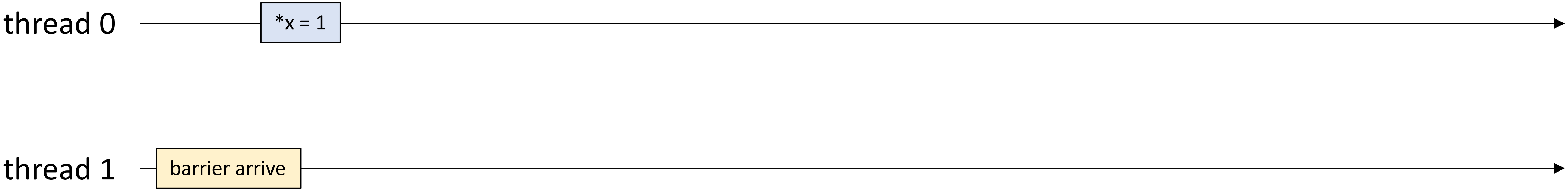
Given a global barrier `B`
and a global memory location x where
initially *x = 0;

Thread 0:

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

Thread 1:

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



A more formal specification

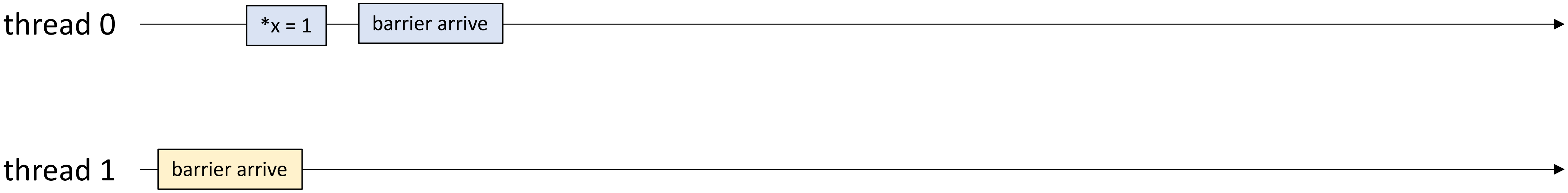
Given a global barrier B
and a global memory location x where
initially *x = 0;

Thread 0:

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

Thread 1:

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



A more formal specification

Thread 0:

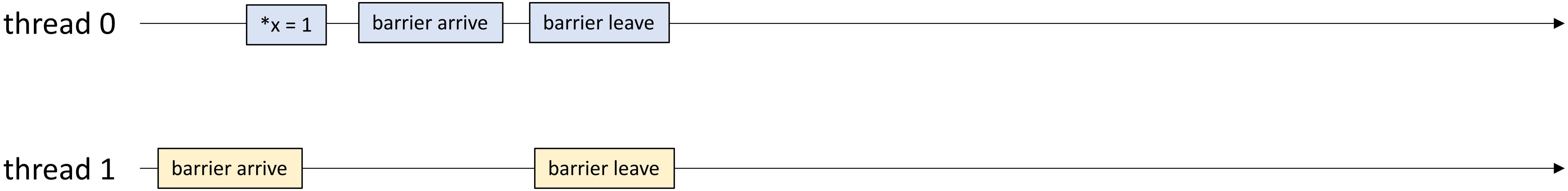
```
*x = 1;
B.barrier();
```

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

Thread 1:

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

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



thread 0

*x = 1

barrier arrive

barrier leave

thread 1

barrier arrive

barrier leave

A more formal specification

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

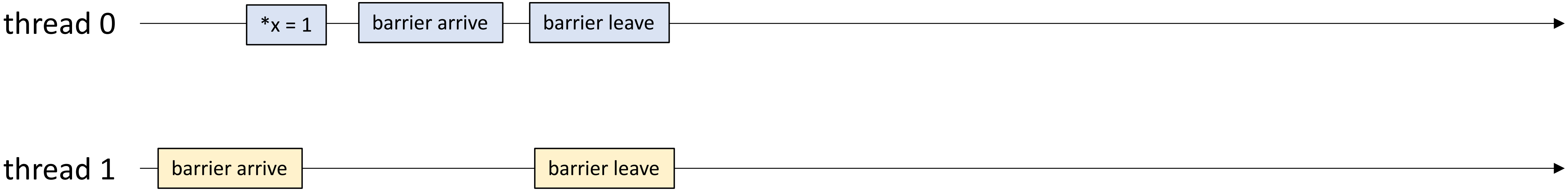
Thread 0:

```
*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;

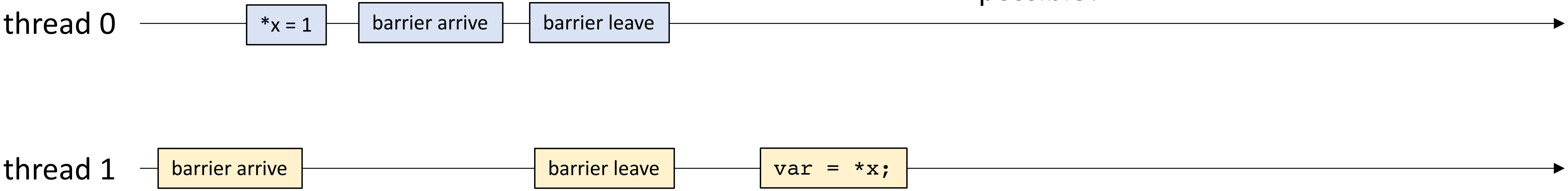
Thread 0:

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

Thread 1:

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

what value must this read? Any other value possible?



thread 0

*x = 1

barrier arrive

barrier leave

thread 1

barrier arrive

barrier leave

`var = *x;`

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

Thread 0:

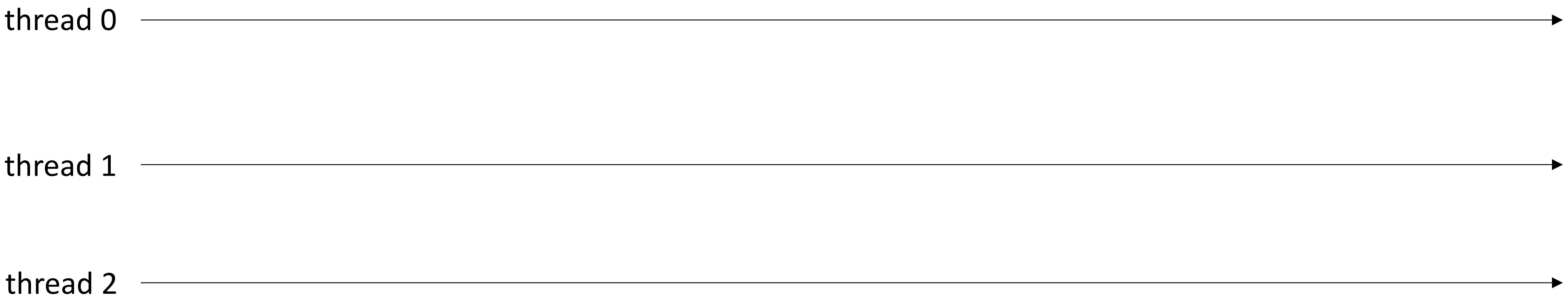
```
*x = 1;
B.barrier();
```

Thread 1:

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

Thread 2:

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



thread 0

thread 1

thread 2

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

Thread 0:

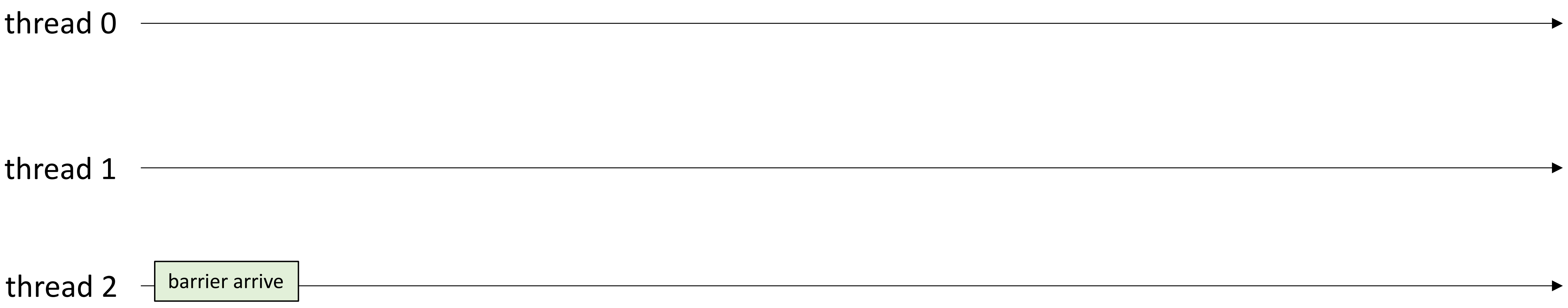
```
*x = 1;
B.barrier();
```

Thread 1:

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

Thread 2:

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



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

Thread 0:

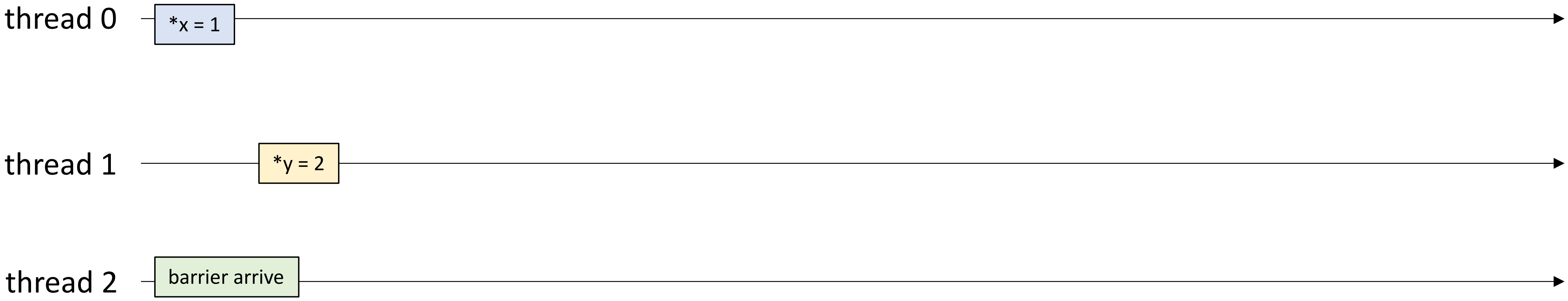
```
*x = 1;
B.barrier();
```

Thread 1:

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

Thread 2:

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



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

Thread 0:

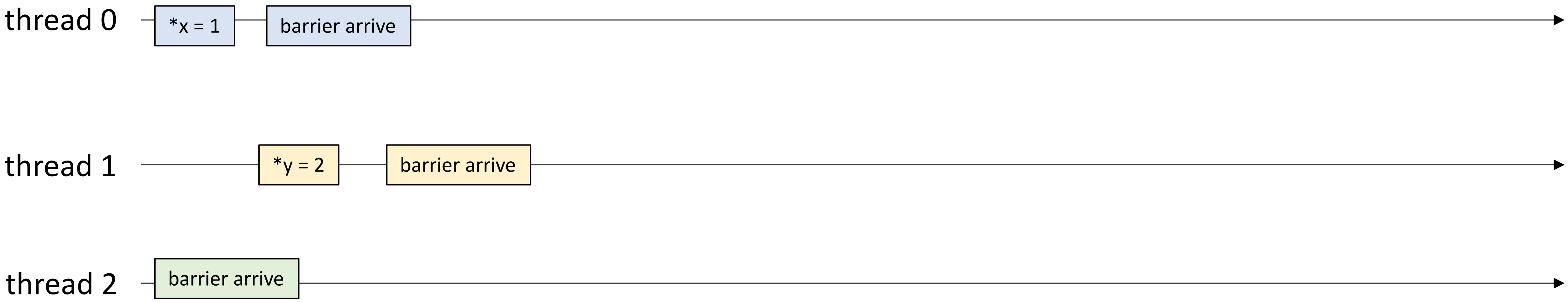
```
*x = 1;
B.barrier();
```

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();
```

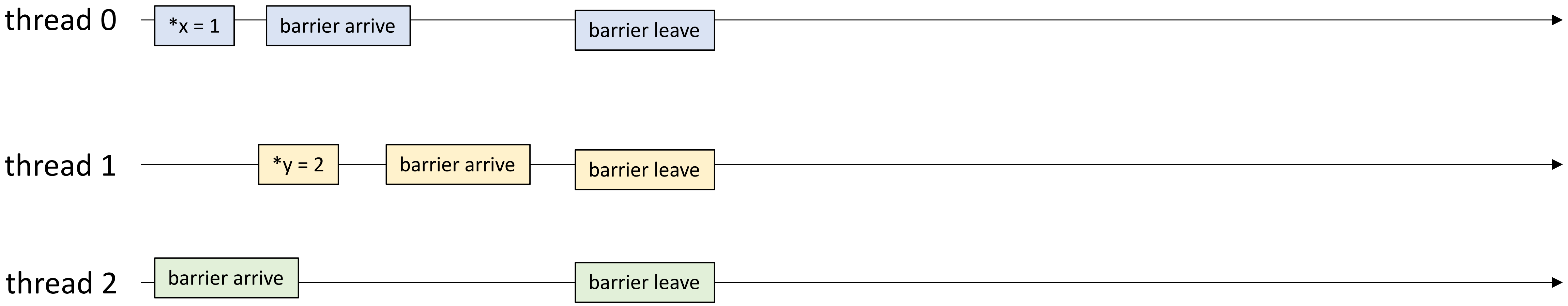
Thread 1:

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

Thread 2:

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

They've all arrived



thread 0

*x = 1

barrier arrive

barrier leave

thread 1

*y = 2

barrier arrive

barrier leave

thread 2

barrier arrive

barrier leave

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

Thread 0:

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

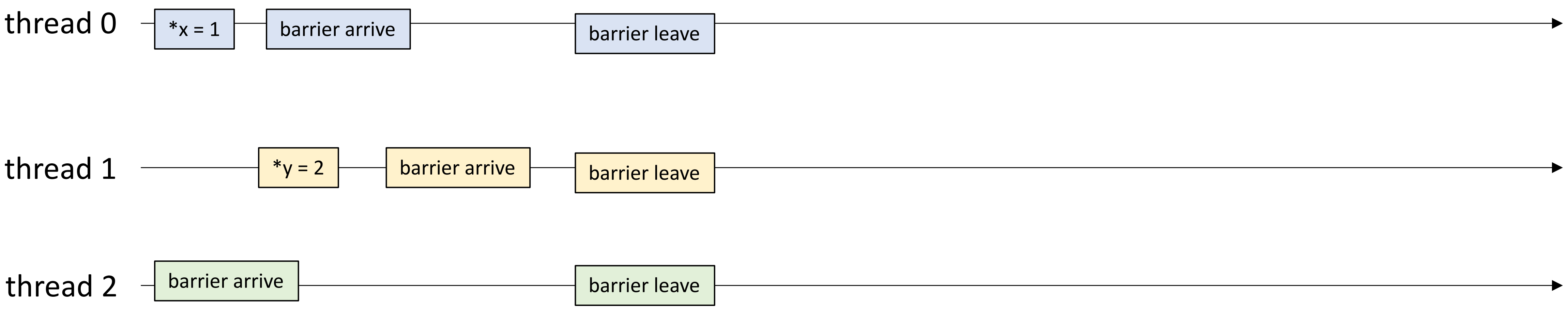
Thread 1:

```
*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:

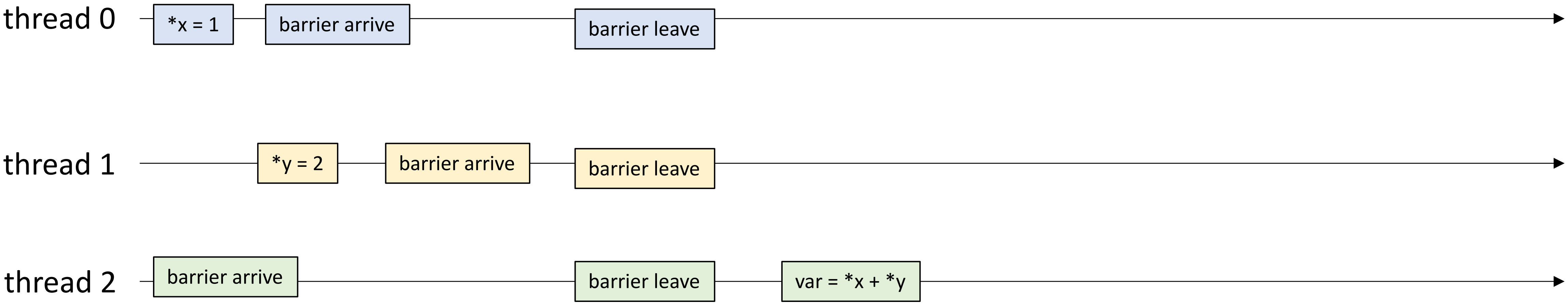
```
*x = 1;
B.barrier();
```

Thread 1:

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

Thread 2:

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



thread 0

*x = 1

barrier arrive

barrier leave

thread 1

*y = 2

barrier arrive

barrier leave

thread 2

barrier arrive

barrier leave

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:

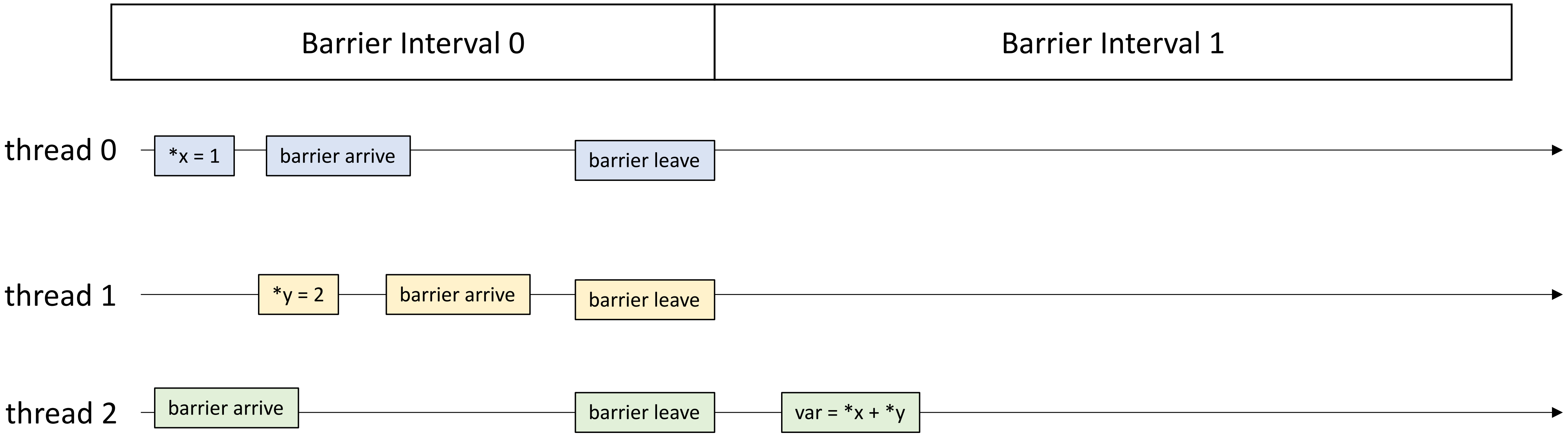
```
*y = 2;
B.barrier();
```

Thread 2:

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

sometimes called a *phase*

extending to the next *barrier leave*



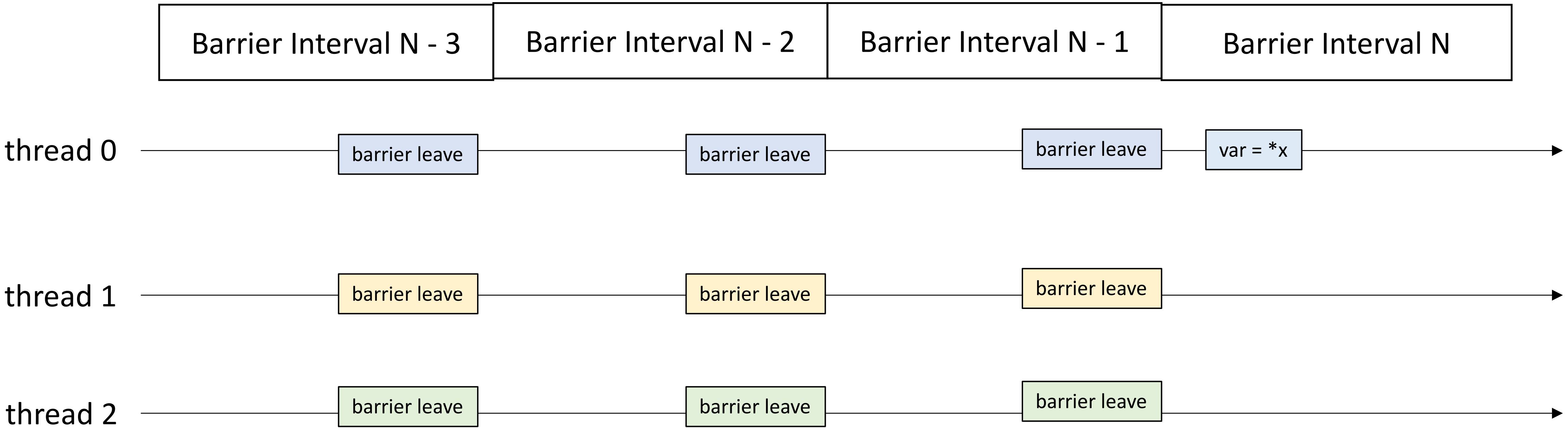
Barriers

- 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

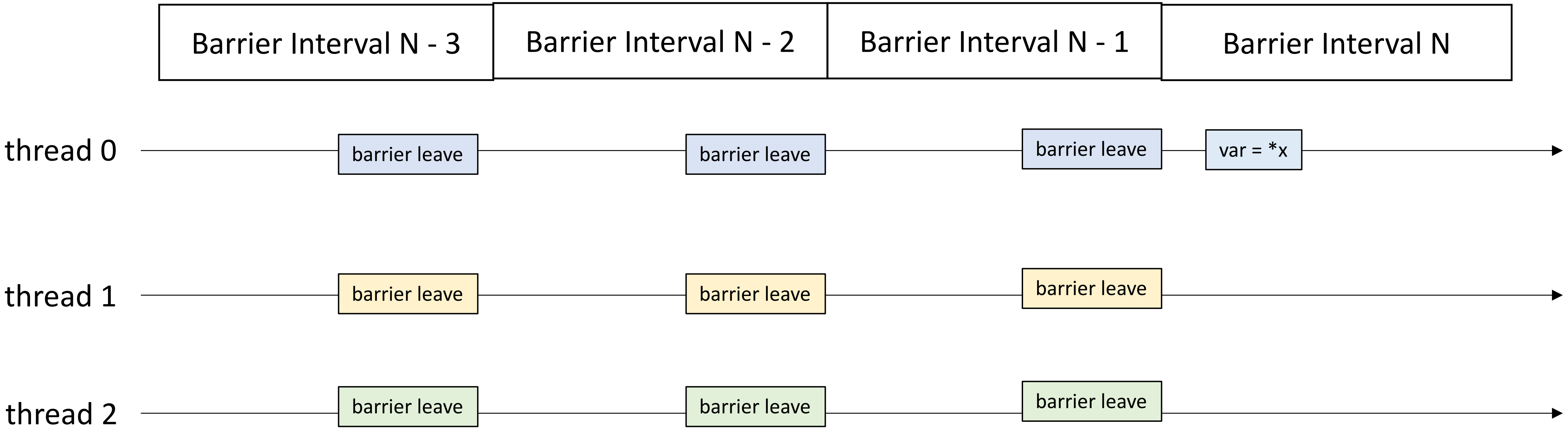
We are only allowed to return one possible value



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

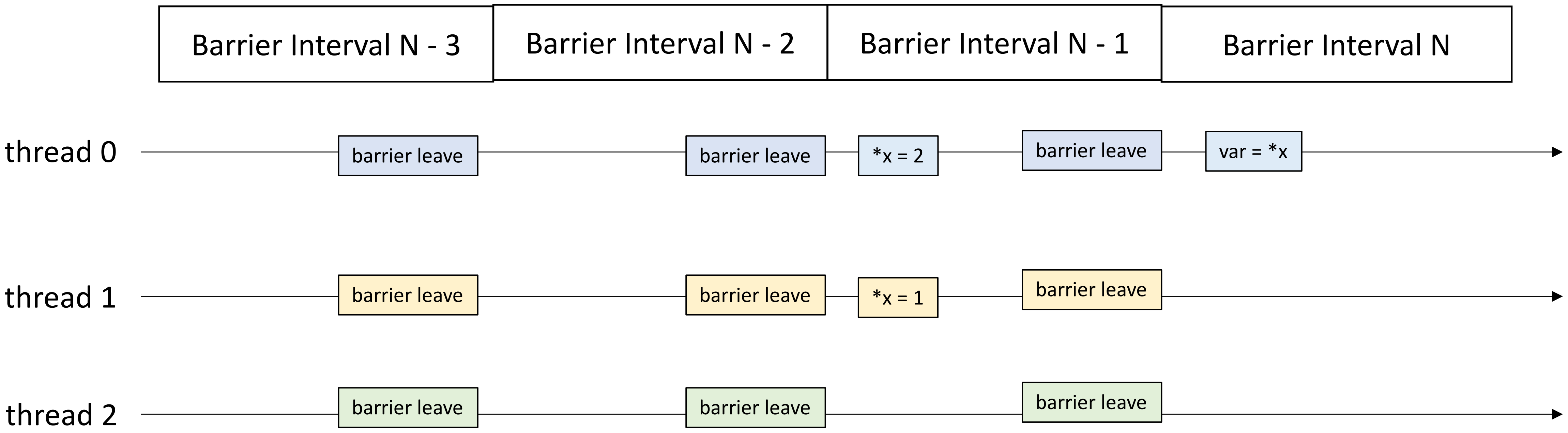


no data conflicts means that x is written to at most once per barrier interval

Assume we are reading from x

We are only allowed to return one possible value

not allowed

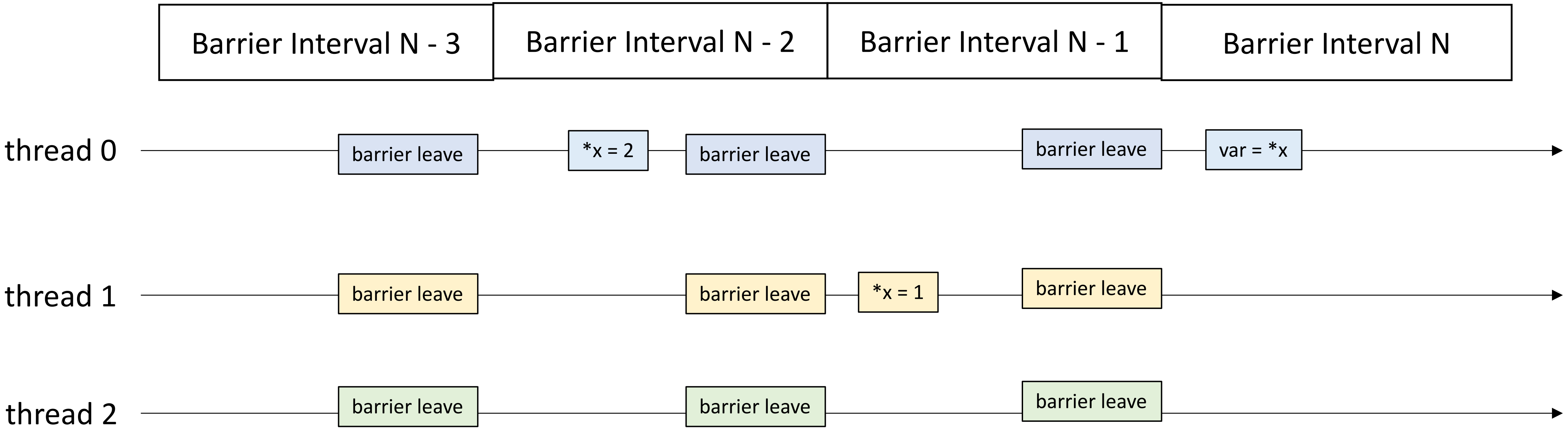


no data conflicts means that x is written to at most once per barrier interval

Assume we are reading from x

We are only allowed to return one possible value

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



Schedule

- Module 4 introduction

- **Barriers**
  - Specification
  - **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 - 1) {
      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 - 1) {
     counter.store(0);
    }
    else {
     while (counter.load() != 0);
    }
  }

}
```

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);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
     }

}
```

Spin while there
is a thread waiting
at the barrier

Does this work?

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();
```

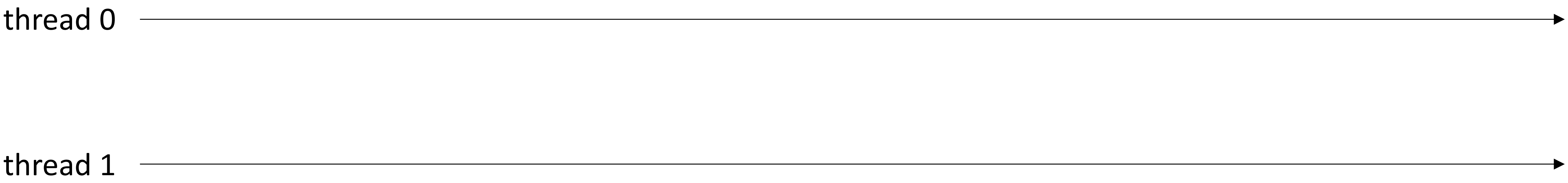
```
void barrier() {
    int arrival_num = atomic_fetch_add(&counter, 1);
    if (arrival_num == num_threads - 1) {
      counter.store(0);
    }
    else {
     while (counter.load() != 0);
    }
  }
```

Thread 1:

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

arrival_num = 1

arrival_num = 0



```
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 - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

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

arrival_num = 1

arrival_num = 0

```
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 - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

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

arrival_num = 1

arrival_num = 0

```
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 - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

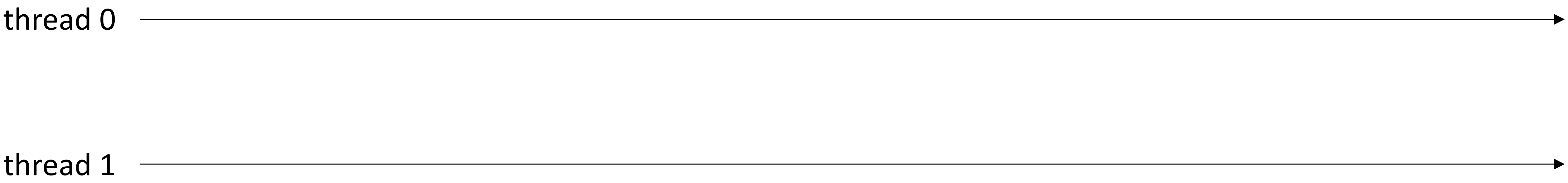
Thread 1:

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

Leaves barrier

arrival_num = 0

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 - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

😴

Thread 1:

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

Leaves barrier

arrival_num = 0

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 - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

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

enters next barrier

arrival_num = 0

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 - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

😴

Thread 1:

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

arrival_num == 0

arrival_num = 0

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 - 1) {
          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 == 0

arrival_num = 0

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 - 1) {
          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 == 0

arrival_num = 0

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 - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

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

Ideas for fixing?

Thread 0:

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

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          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 - 1) {
          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 - 1) {
          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
- Alternating ”sense” dynamically

Thread 0:

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

sync on sense = false

Thread 1:

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

Sense Reversing Barrier

- Book Chapter 17
- Alternating ”sense” dynamically

Thread 0:

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

sync on sense = true

Thread 1:

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

```
class SenseBarrier {
  private:
    atomic_int counter;
    int num_threads;
    atomic_bool sense;
    bool thread_sense[num_threads];
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
      sense = false;
      thread_sense = {true, ...};
    }

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

thread_sense = true

Thread 0:

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

```
num_threads == 2
counter == 0
sense = false
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = true

Thread 1:

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

thread_sense = true

arrival_num = 1

Thread 0:

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

```
num_threads == 2
counter == 2
sense = false
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = true

arrival_num = 0

Thread 1:

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

thread_sense = true

arrival_num = 1

Thread 0:

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

```
num_threads == 2
counter == 2
sense = false
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = true

arrival_num = 0

Thread 1:

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

thread_sense = false

arrival_num = 1

Thread 0:

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

```
num_threads == 2
counter == 0
sense = true
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = true

arrival_num = 0

Thread 1:

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

thread_sense = false

arrival_num = ?

Thread 0:

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

```
num_threads == 2
counter == 0
sense = true
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = true

arrival_num = 0

Thread 1:

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

Remember the issue! Thread 1 went to sleep around this time and thread 0 went into the barrier again!

```
num_threads == 2
counter == 1
sense = true
```

thread_sense = false

arrival_num = 0

Thread 0:

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

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = true

arrival_num = 0

Thread 1:

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

```
num_threads == 2
counter == 1
sense = true
```

thread_sense = false

arrival_num = 0

Thread 0:

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

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = true

arrival_num = 0

Thread 1:

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

both are waiting!,
but thread 1 can leave

thread_sense = false

arrival_num = 0

Thread 0:

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

```
num_threads == 2
counter == 1
sense = true
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = false

arrival_num = 0

Thread 1:

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

both are waiting!,
but thread 1 can leave

```
num_threads == 2
counter == 1
sense = true
```

thread_sense = false

arrival_num = 0

Thread 0:

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

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = false

arrival_num = ?

Thread 1:

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

Thread 1 finishes the barrier

```
num_threads == 2
counter == 1
sense = true
```

thread_sense = false

arrival_num = 0

Thread 0:

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

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = false

arrival_num = ?

Thread 1:

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

Goes into the second barrier

thread_sense = false

arrival_num = 0

__Thread 0:__

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

```
num_threads == 2
counter == 2
sense = true
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = false

arrival_num = 1

__Thread 1:__

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

Goes into the second barrier

thread_sense = false

arrival_num = 0

__Thread 0:__

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

```
num_threads == 2
counter == 2
sense = true
```

thread_sense = false

arrival_num = 1

__Thread 1:__

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

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

Goes into the second barrier

thread_sense = false

arrival_num = 0

```
num_threads == 2
counter == 0
sense = false
```

thread_sense = false

arrival_num = 1

Thread 0:

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

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

Thread 1:

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

Goes into the second barrier

thread_sense = false

arrival_num = 0

Thread 0:

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

```
num_threads == 2
counter == 0
sense = false
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

thread_sense = false

arrival_num = 1

Thread 1:

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

thread 0 can leave, thread 1 can leave and the barrier works as expected!

See you on Wednesday!

- Starting on module 4
- Work on HW 3