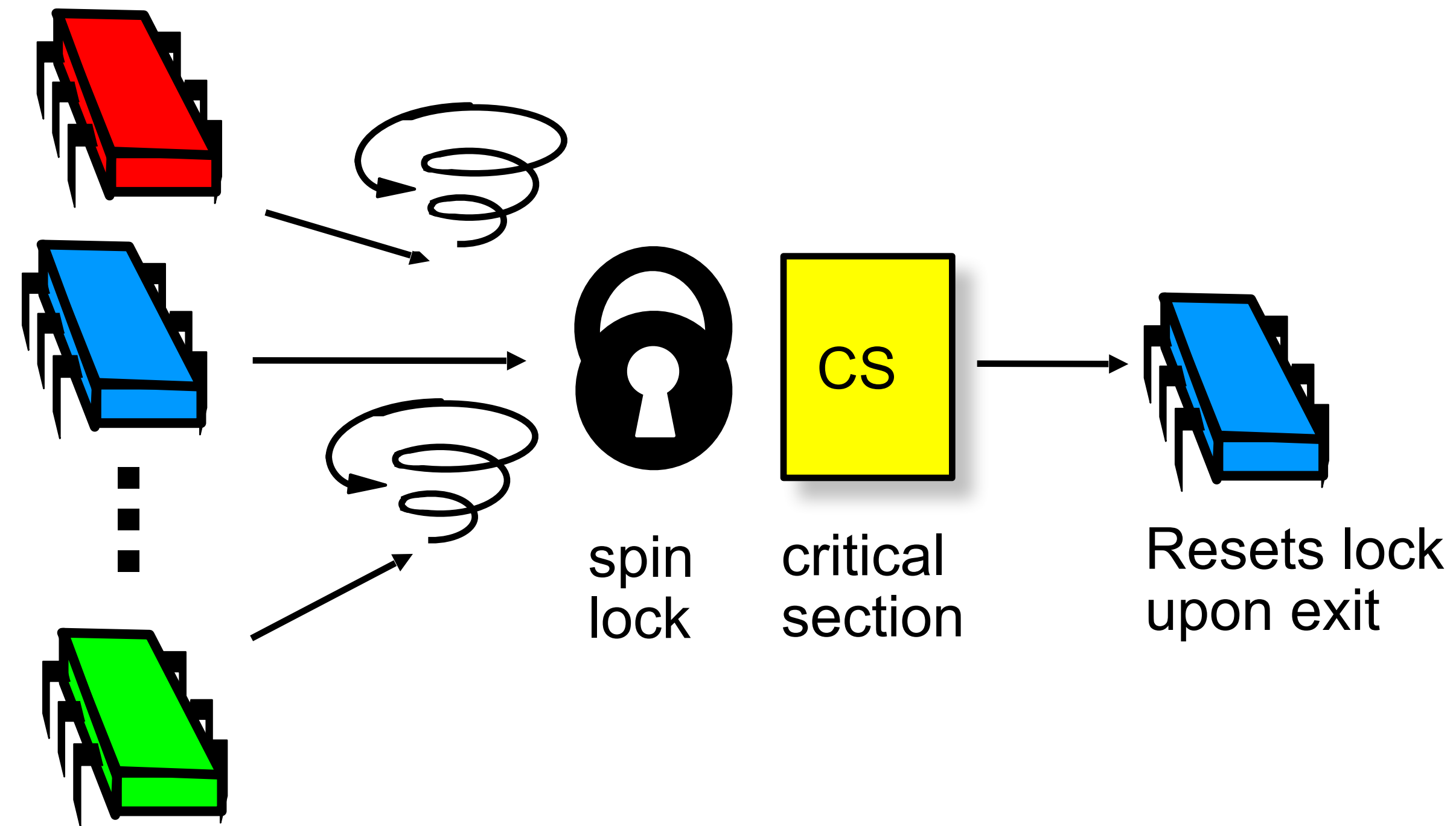


Barrier Synchronization

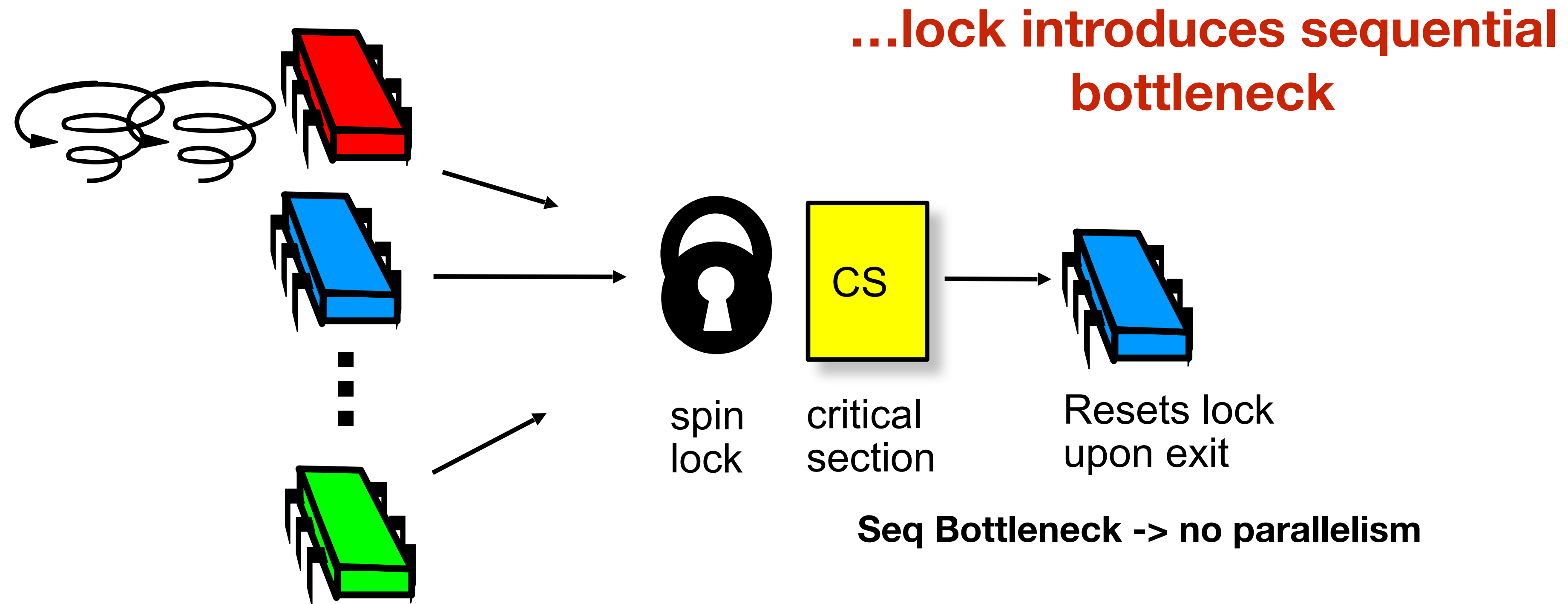
CS 475, Fall 2019

Concurrent & Distributed Systems

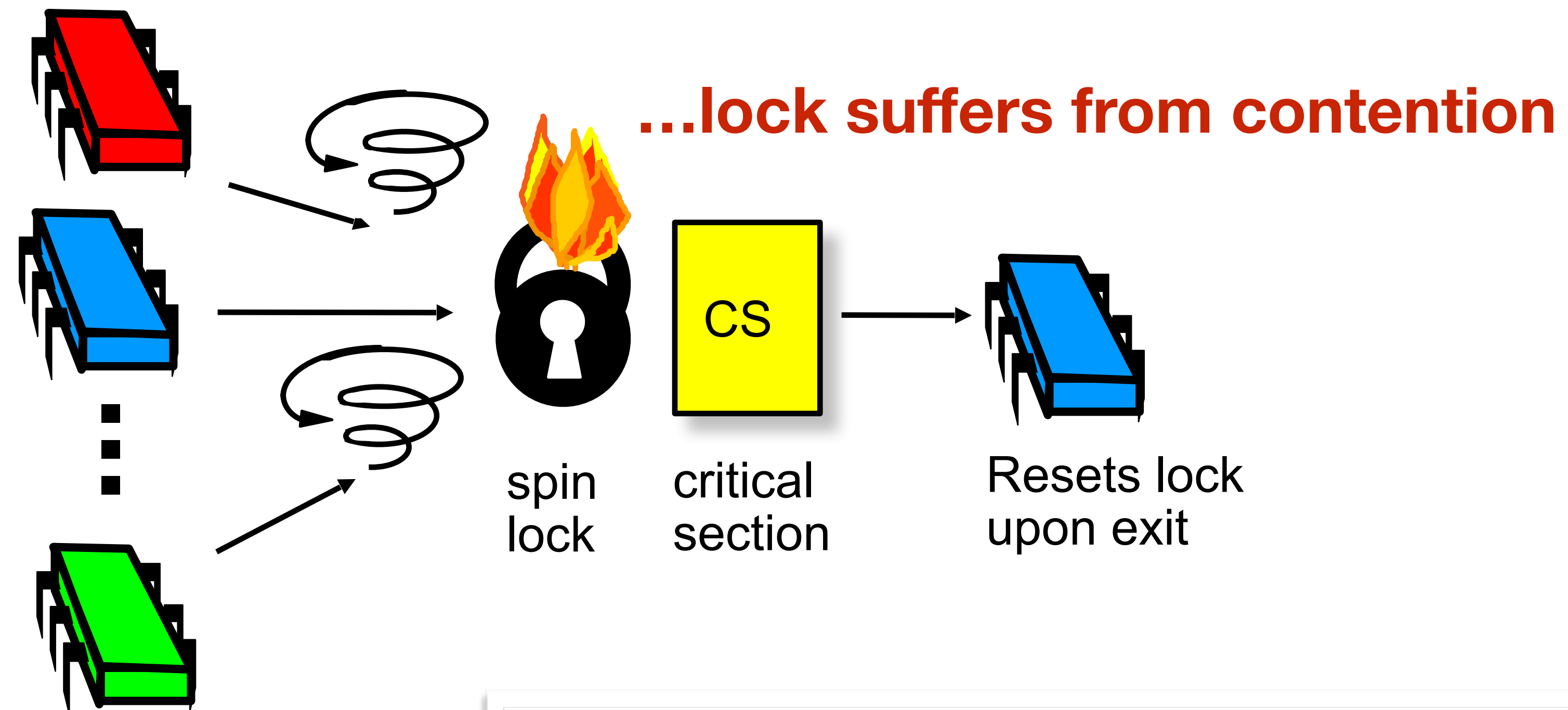
Basic Spin-Lock



Basic Spin-Lock

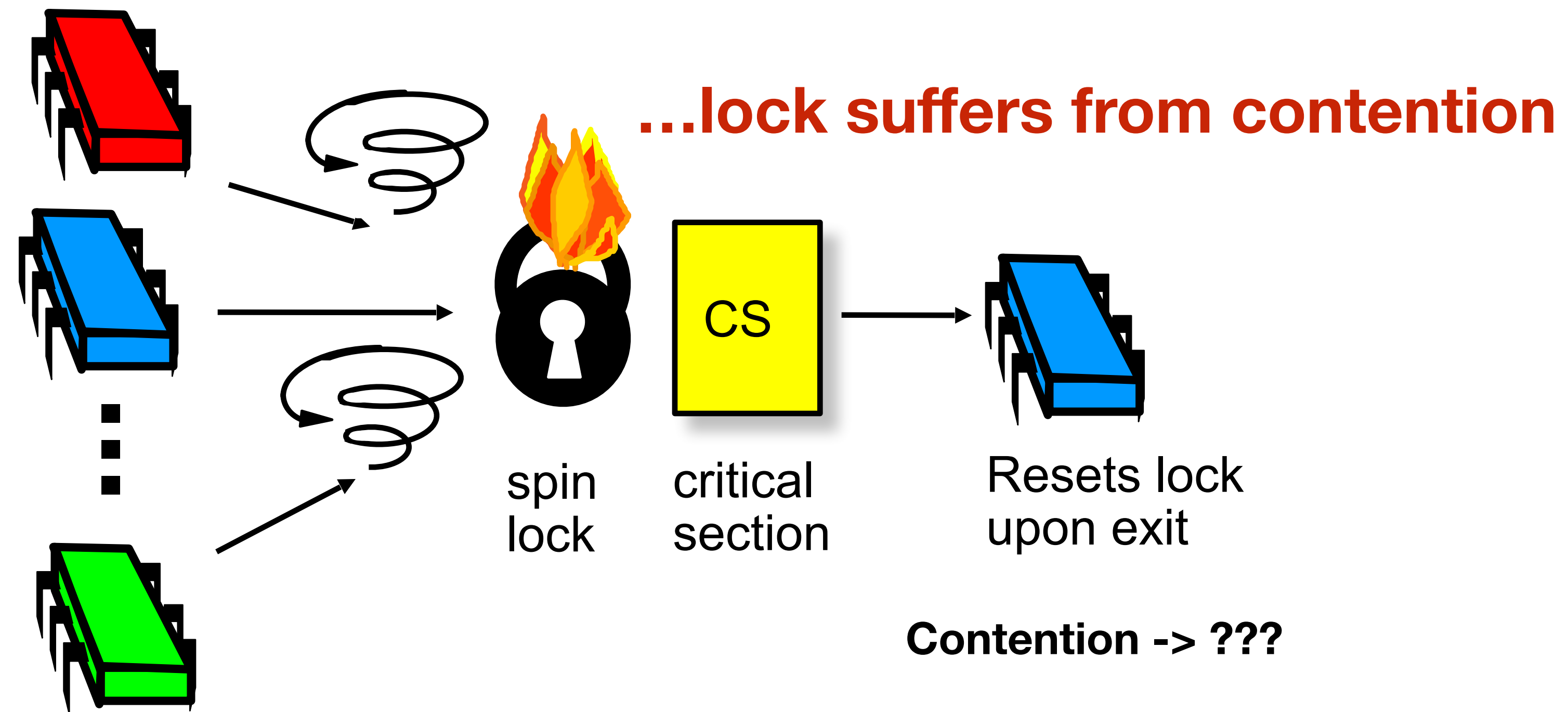


Basic Spin-Lock

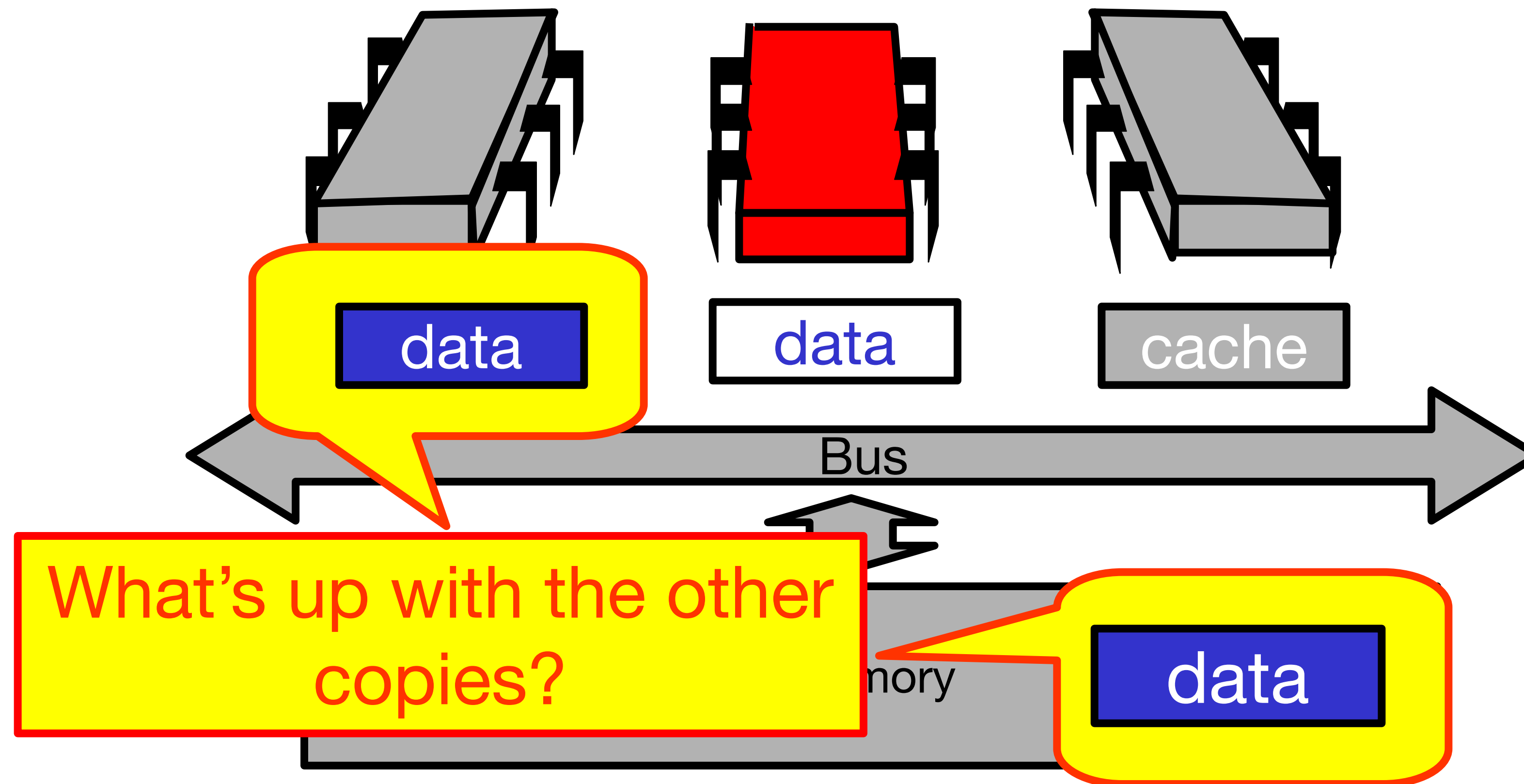


Note: Contention and bottlenecks are separate phenomena

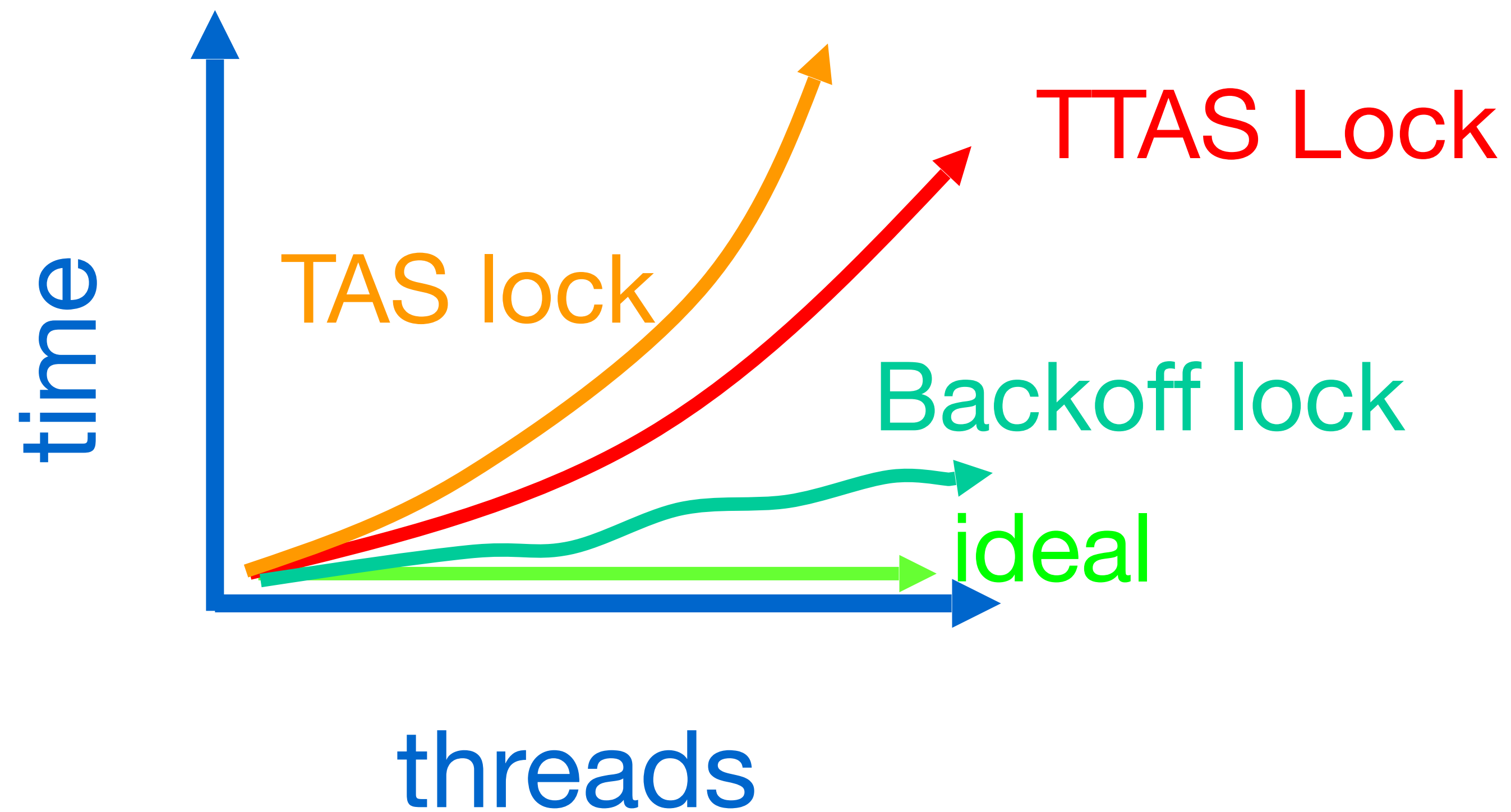
Basic Spin-Lock



Modify Cached Data



Spin-Waiting Overhead



Moral of the story?

- EVEN IF we do pretty good with parallelizing most parts our application, we can still see slowdown from **contention** for locks
- For resources infrequently contended, spin locks can be fast because no context switch is necessary
- But, contention in spin locks can have repercussions due to hardware architectures

Today

- Barrier synchronization - the evil (?) twin of mutual exclusion
- Last synchronization topic!
- Focus very heavily on *implementation*: how much memory is required, and how much contention is there on those resources?

Simple Video Game

- Prepare frame for display
 - By graphics coprocessor
- “soft real-time” application
 - Need at least 35 frames/second
 - OK to mess up rarely

Simple Video Game

```
while (true) {  
    frame.prepare();  
    frame.display();  
}
```

Simple Video Game

- What about overlapping work?
 - 1st thread displays frame
 - 2nd prepares next frame

```
while (true) {  
    frame.prepare();  
    frame.display();  
}
```

Two-Phase Rendering

```
while (true) {  
  if (phase) {  
    frame[0].display();  
  } else {  
    frame[1].display();  
  }  
  phase = !phase;  
}
```

```
while (true) {  
  if (phase) {  
    frame[1].prepare();  
  } else {  
    frame[0].prepare();  
  }  
  phase = !phase;  
}
```

Two-Phase Rendering

```
while (true) {  
  if (phase) {  
    frame[0].display();  
  } else {  
    frame[1].display();  
  }  
  phase = !phase;  
}
```

```
while (true) {  
  if (phase) {  
    frame[1].prepare();  
  } else {  
    frame[0].prepare();  
  }  
  phase = !phase;  
}
```

Even phases

Two-Phase Rendering

```
while (true) {  
  if (phase) {  
    frame[0].display();  
  } else {  
    frame[1].display();  
  }  
  phase = !phase;  
}
```

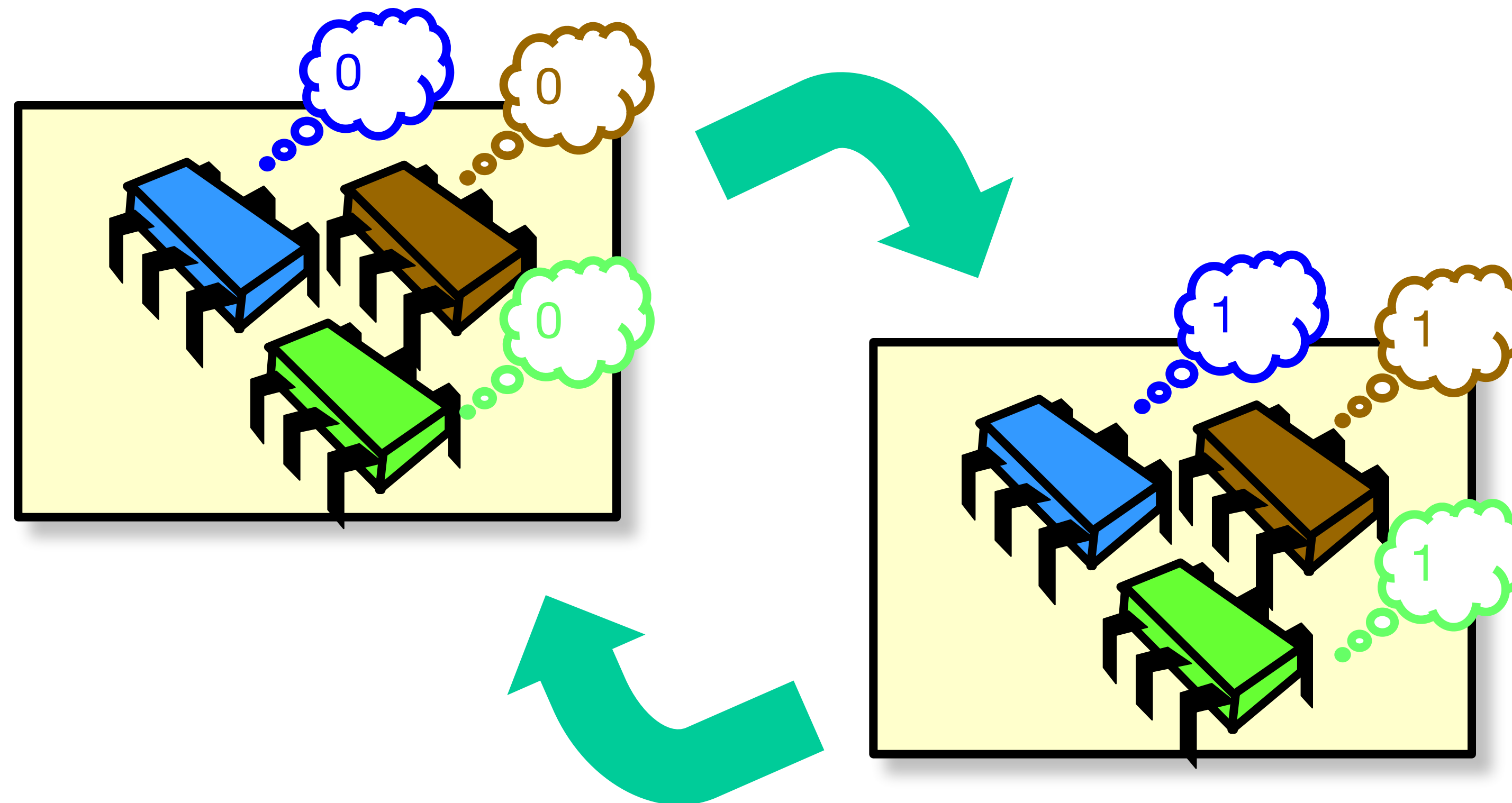
```
while (true) {  
  if (phase) {  
    frame[1].prepare();  
  } else {  
    frame[0].prepare();  
  }  
  phase = !phase;  
}
```

odd phases

Synchronization Problems

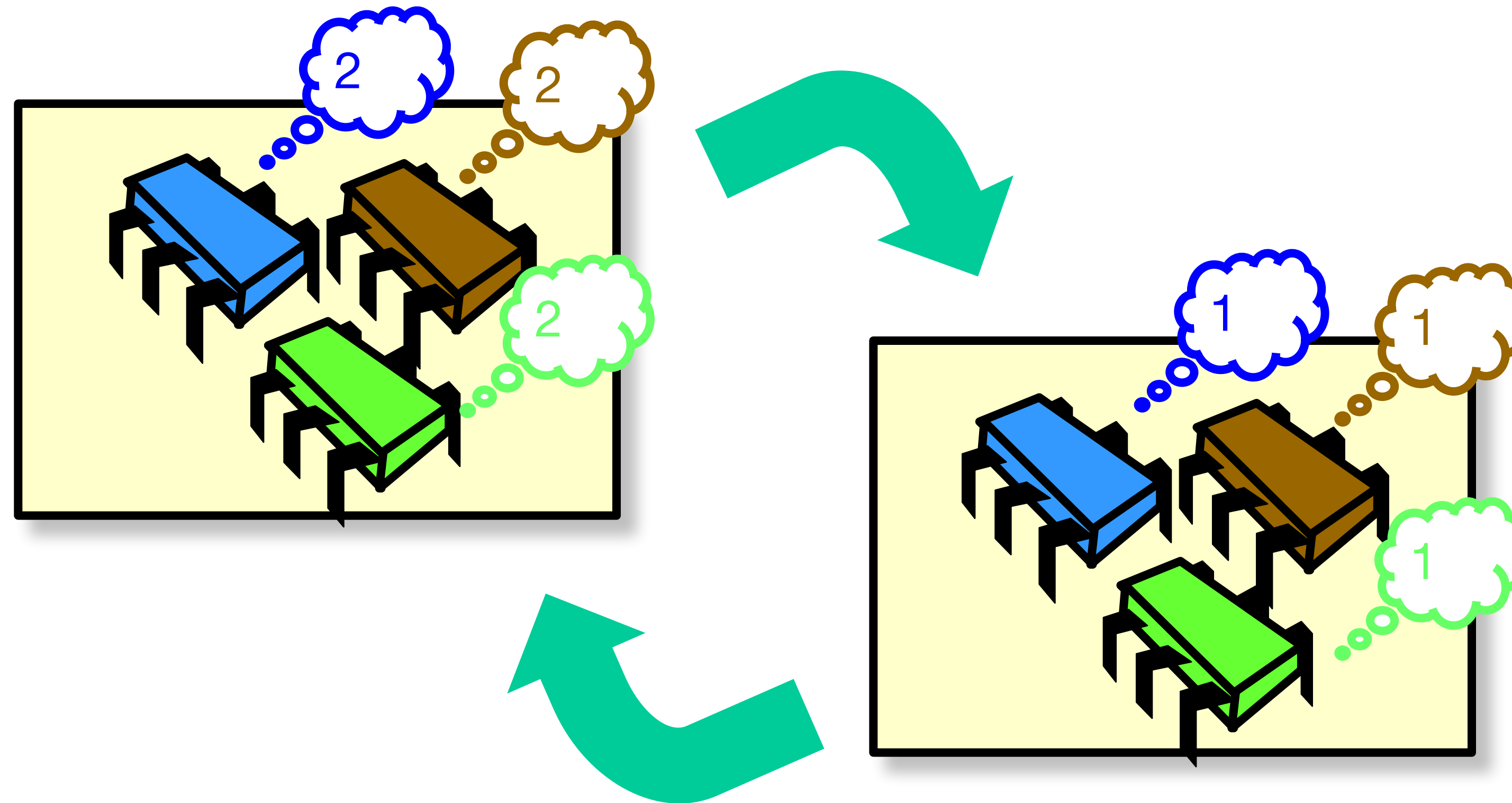
- How do threads stay in phase?
- Too early?
 - “we render no frame before its time”
- Too late?
 - Recycle memory before frame is displayed

Ideal Parallel Computation



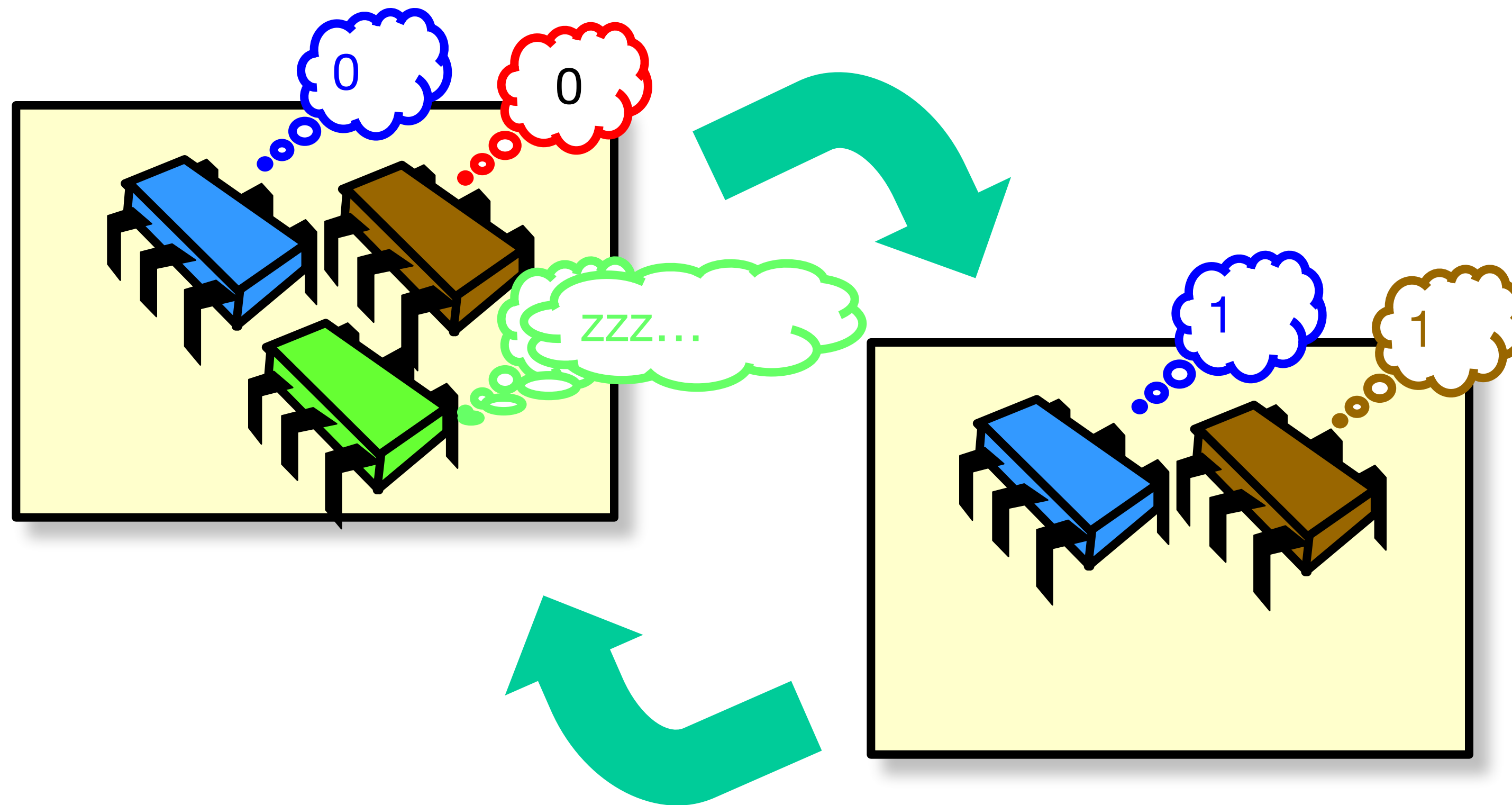
Ideal: All processes/threads move between phases at the same rate - no falling behind/getting ahead

Ideal Parallel Computation

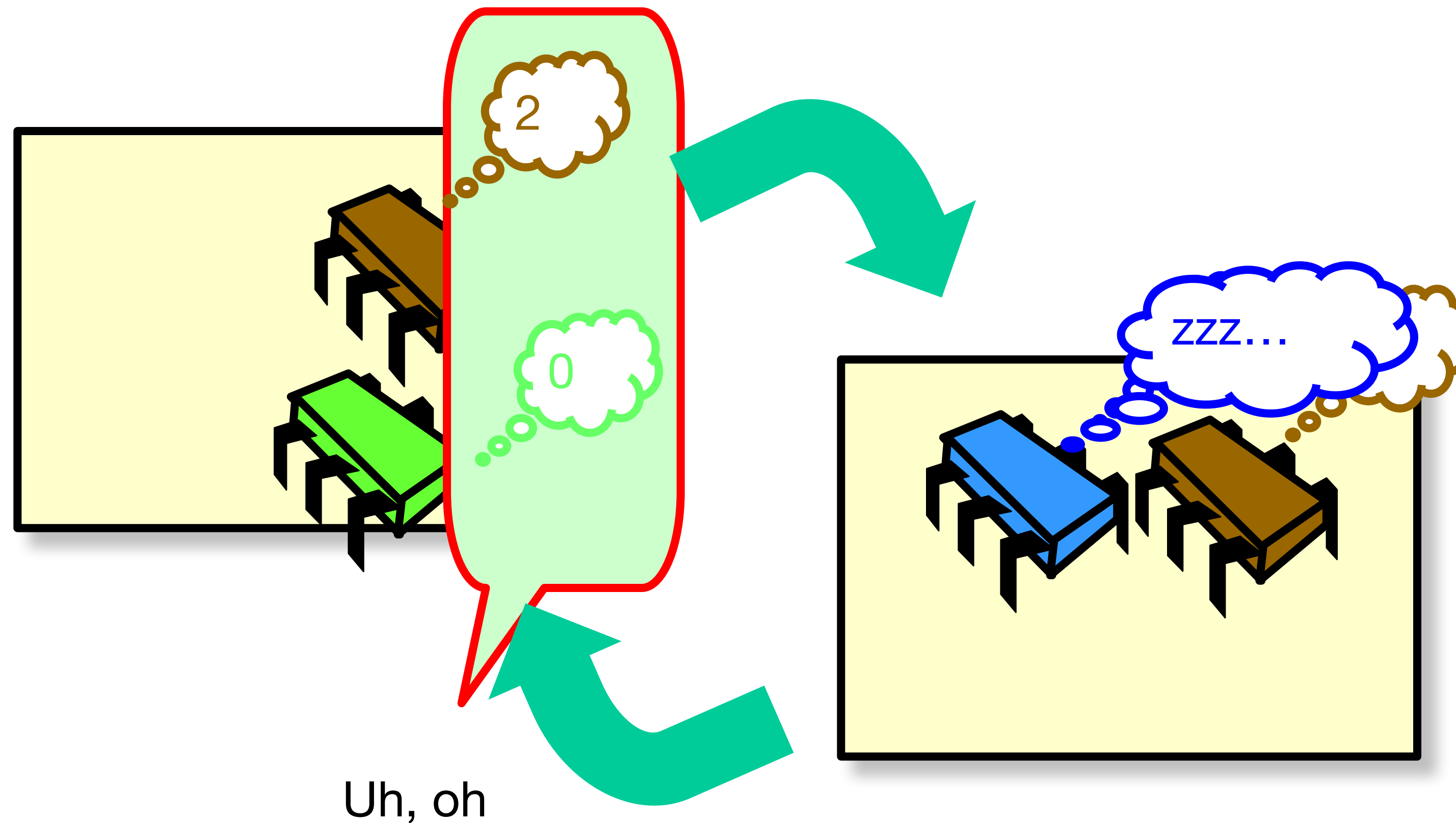


Ideal: All processes/threads move between phases at the same rate - no falling behind/getting ahead

Real-Life Parallel Computation

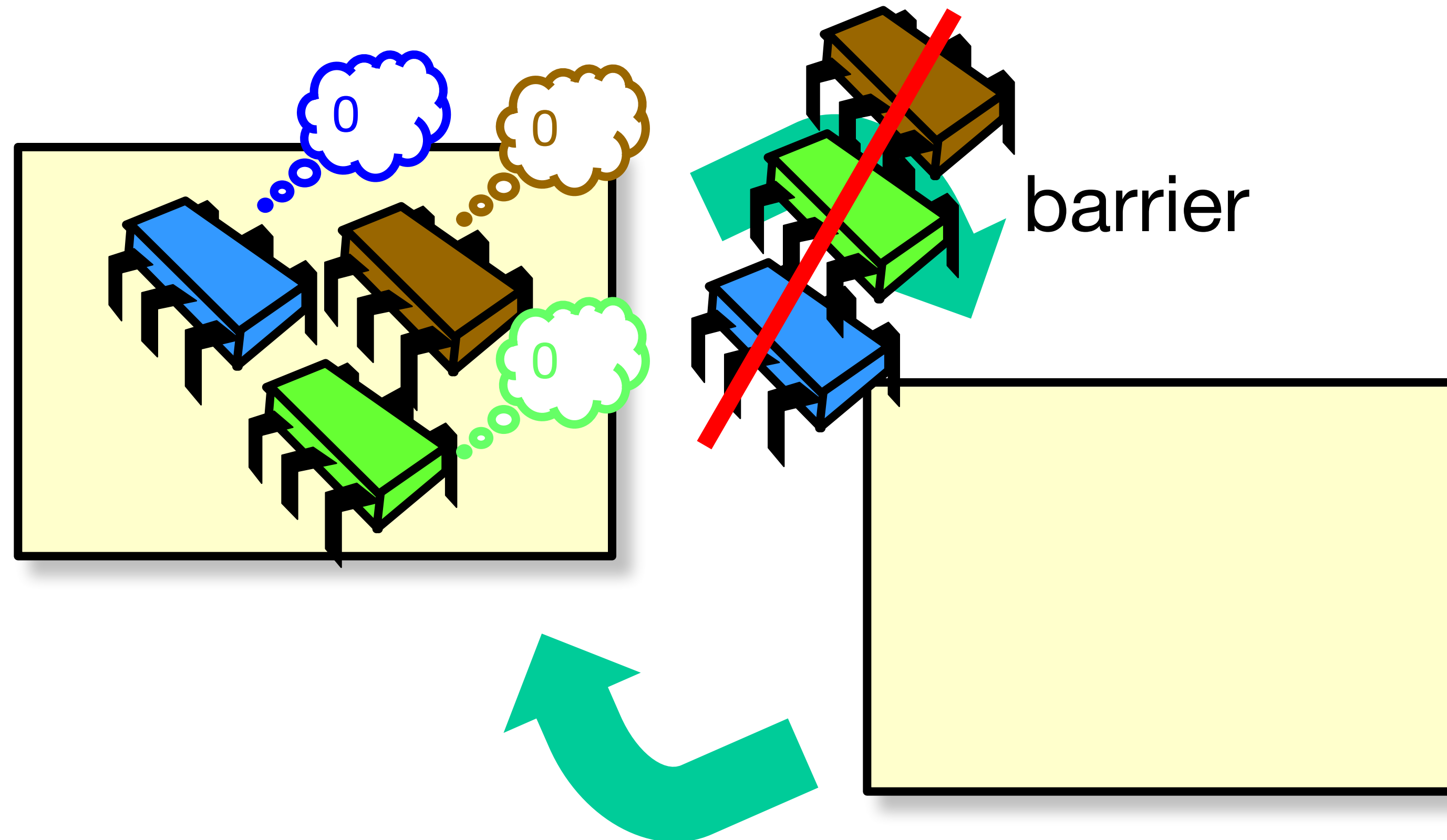


Real-Life Parallel Computation

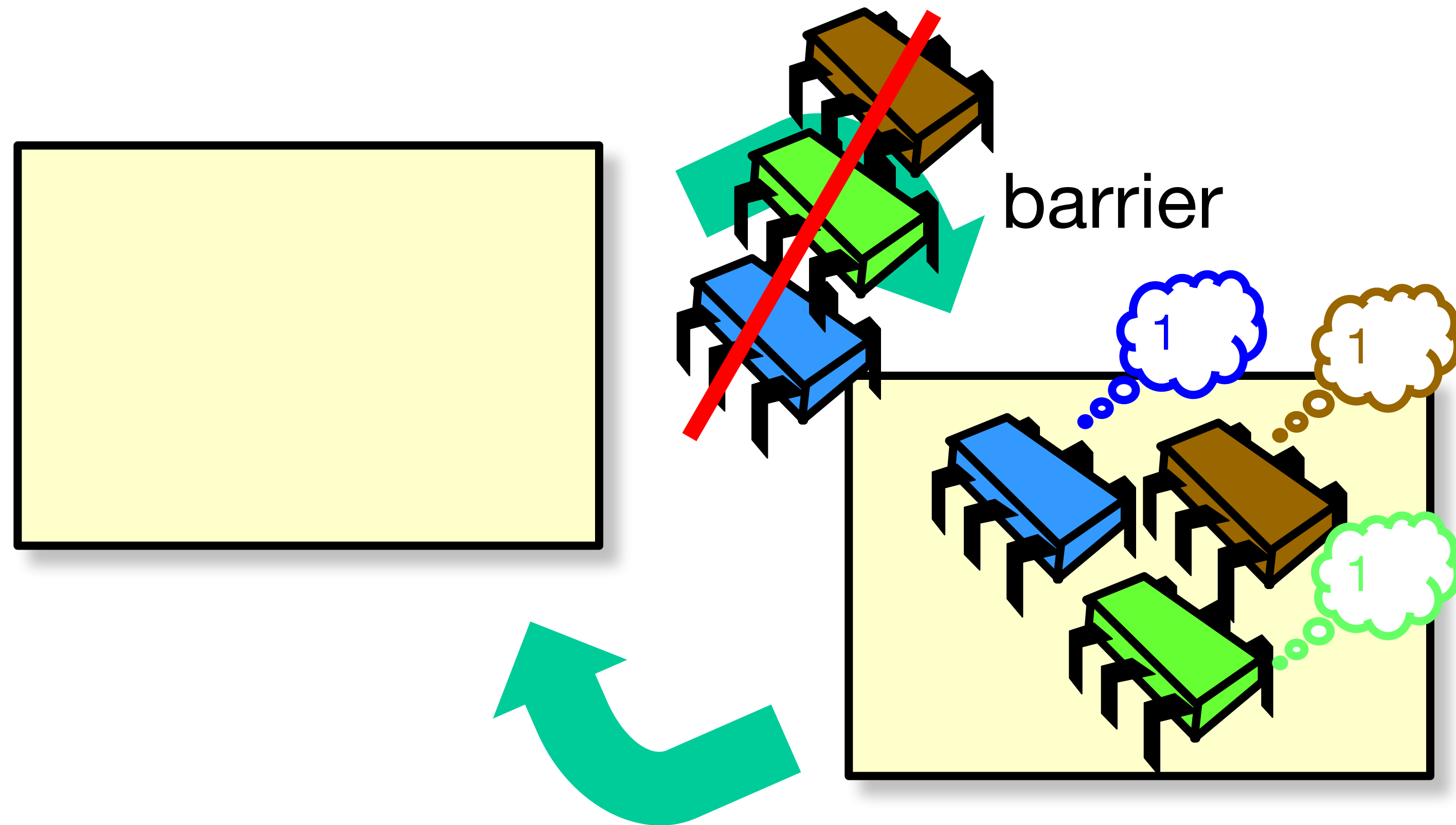


**Actual: Some work is harder than other work, some threads might get less CPU time...
we get out of sync**

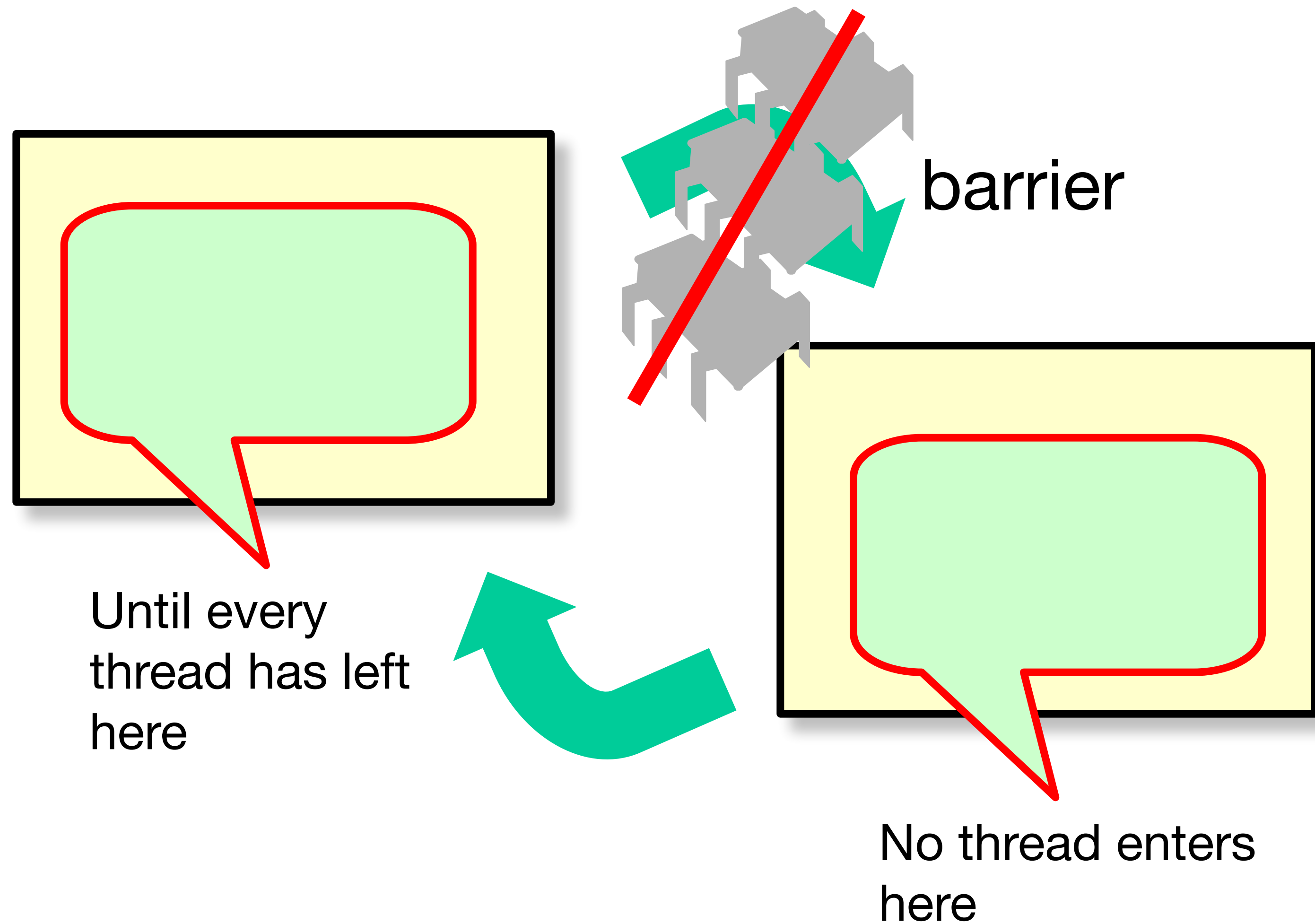
Barrier Synchronization



Barrier Synchronization



Barrier Synchronization



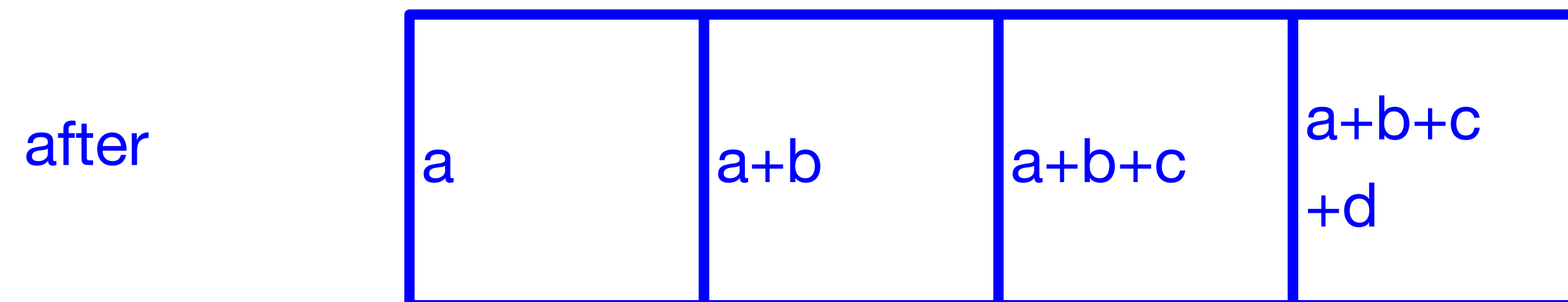
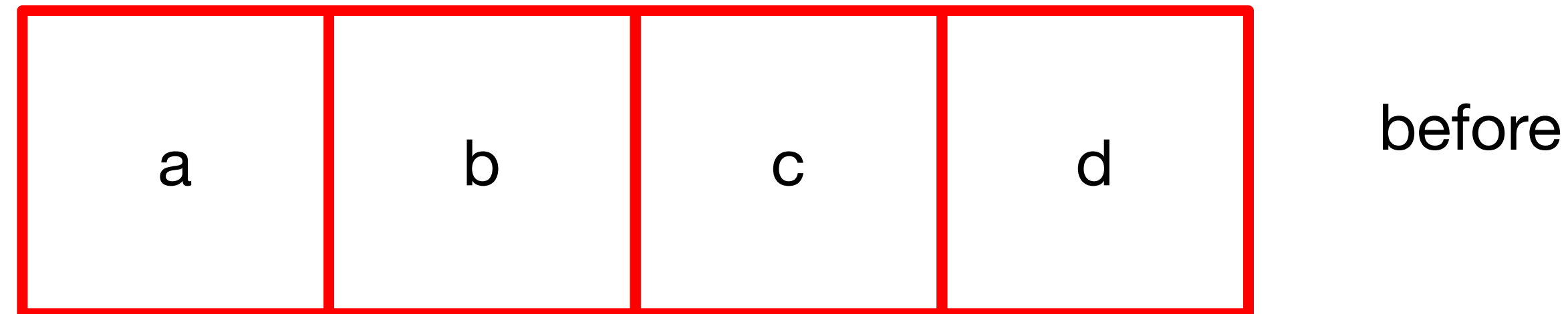
Why Do We Care?

- Mostly of interest to
 - Scientific & numeric computation
 - Distributed protocols
- Elsewhere
 - Garbage collection
 - Less common in systems programming
 - Still important topic

Duality

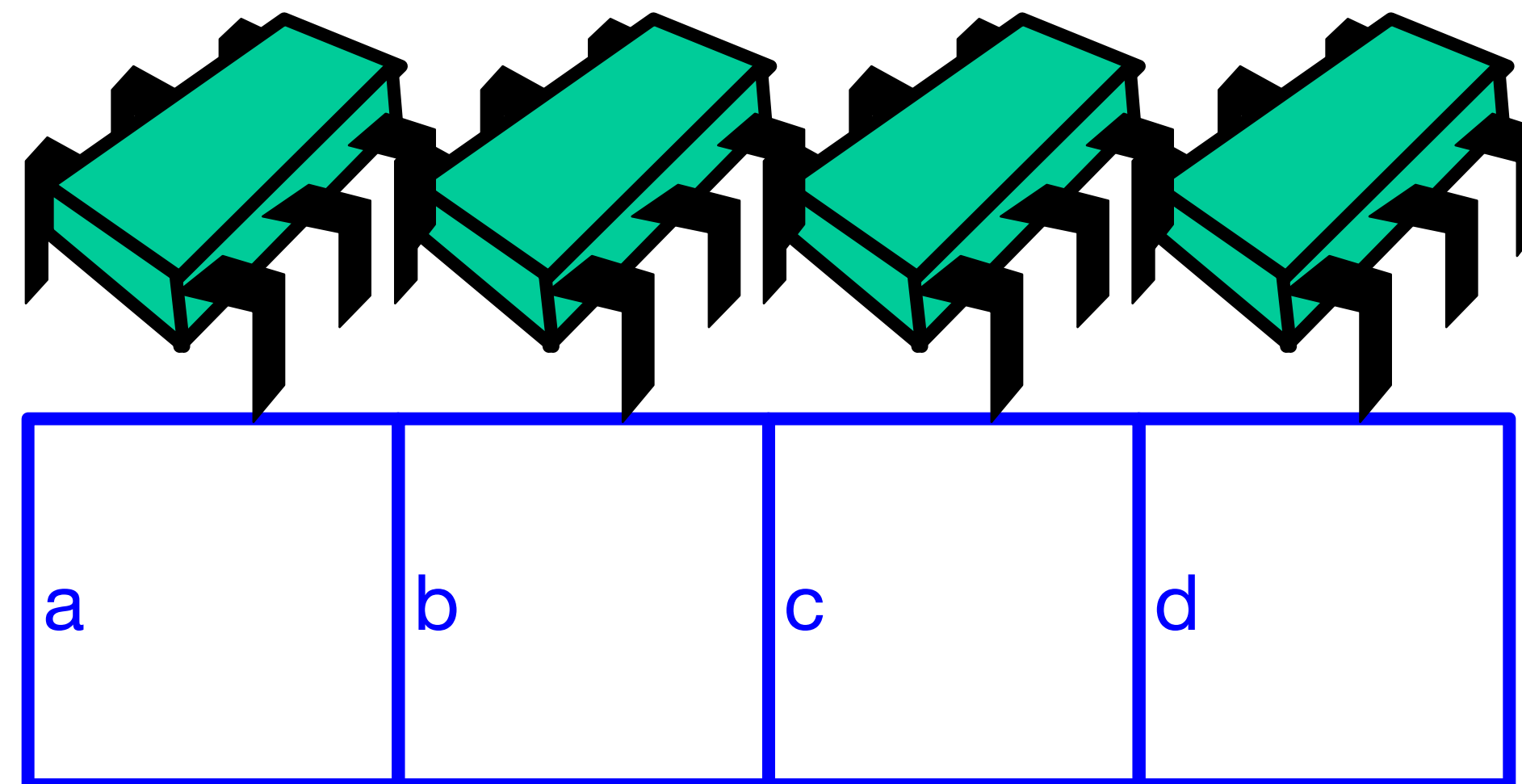
- Dual to mutual exclusion
 - Include others, not exclude them
- Same implementation issues
 - Interaction with caches ...
 - Invalidation?
 - Local spinning?

Example: Parallel Prefix

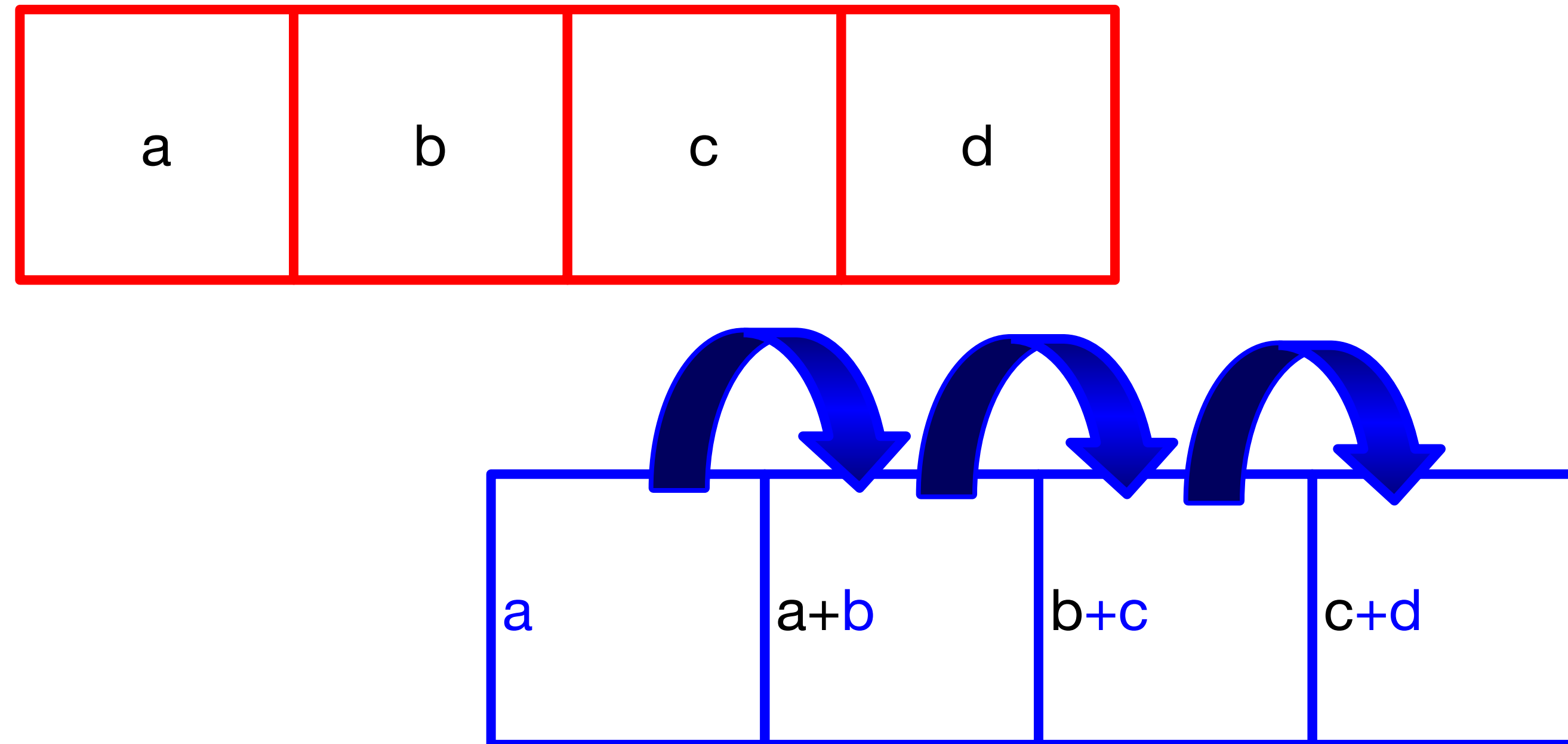


Parallel Prefix

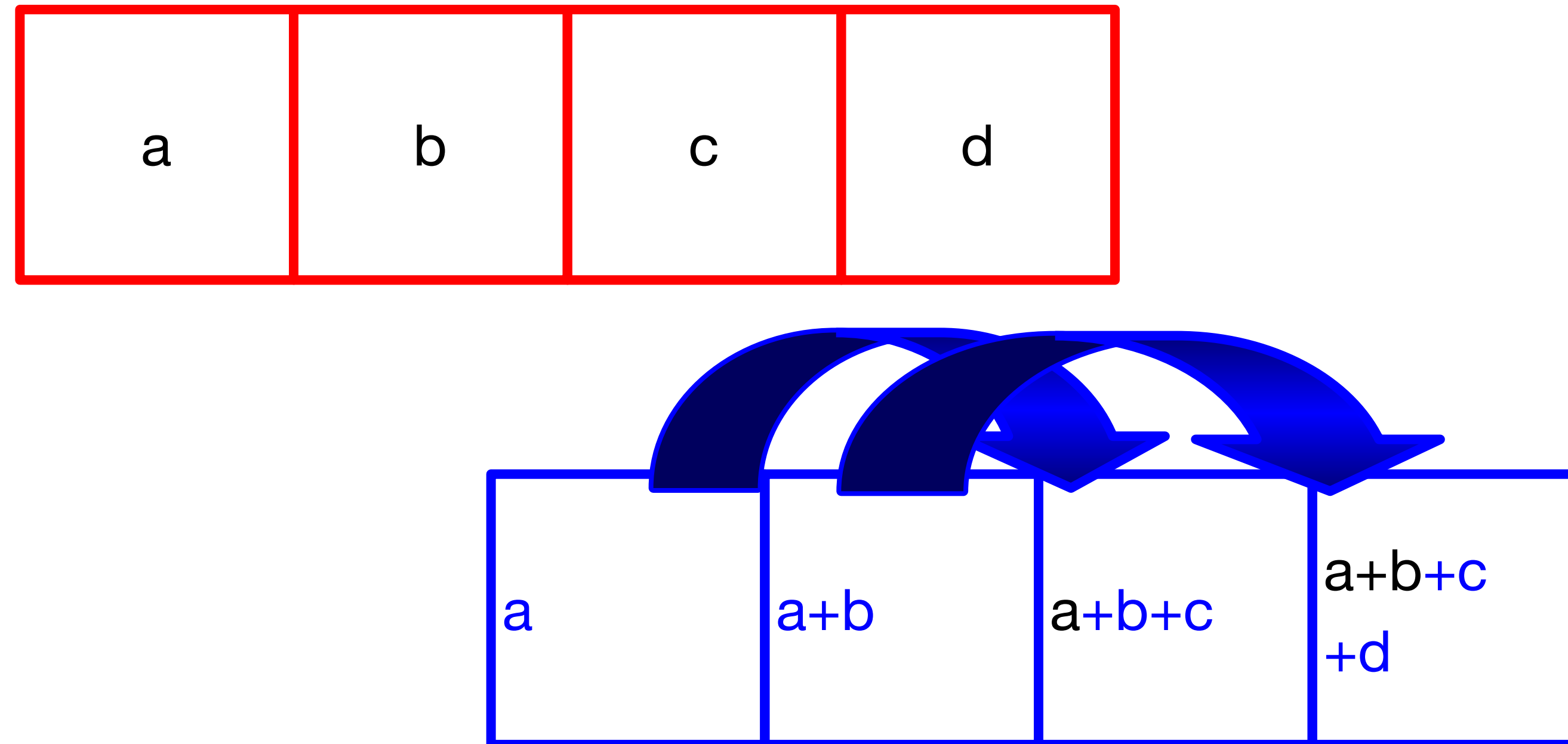
One thread
Per entry



Parallel Prefix: Phase 1



Parallel Prefix: Phase 2



Parallel Prefix

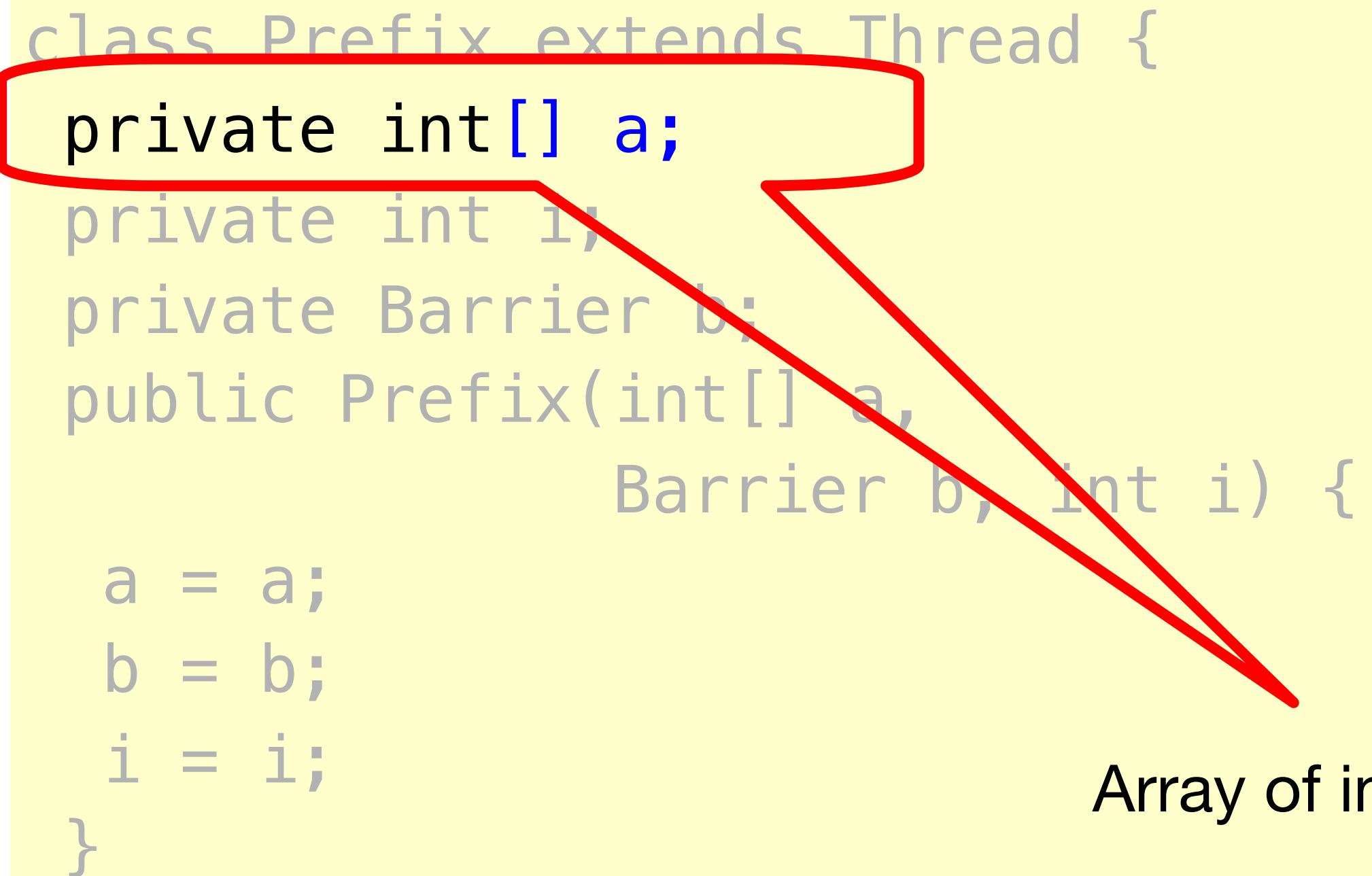
- N threads can compute
 - Parallel prefix
 - Of N entries
 - In $\log_2 N$ rounds
- What if system is asynchronous?
 - Why we need barriers

Prefix

```
class Prefix extends Thread {  
    private int[] a;  
    private int i;  
    private Barrier b;  
    public Prefix(int[] a,  
                 Barrier b, int i) {  
  
        a = a;  
        b = b;  
        i = i;  
    }  
}
```

Prefix

```
class Prefix extends Thread {  
    private int[] a;  
    private int i;  
    private Barrier b;  
    public Prefix(int[] a,  
                 Barrier b, int i) {  
        a = a;  
        b = b;  
        i = i;  
    }  
}
```

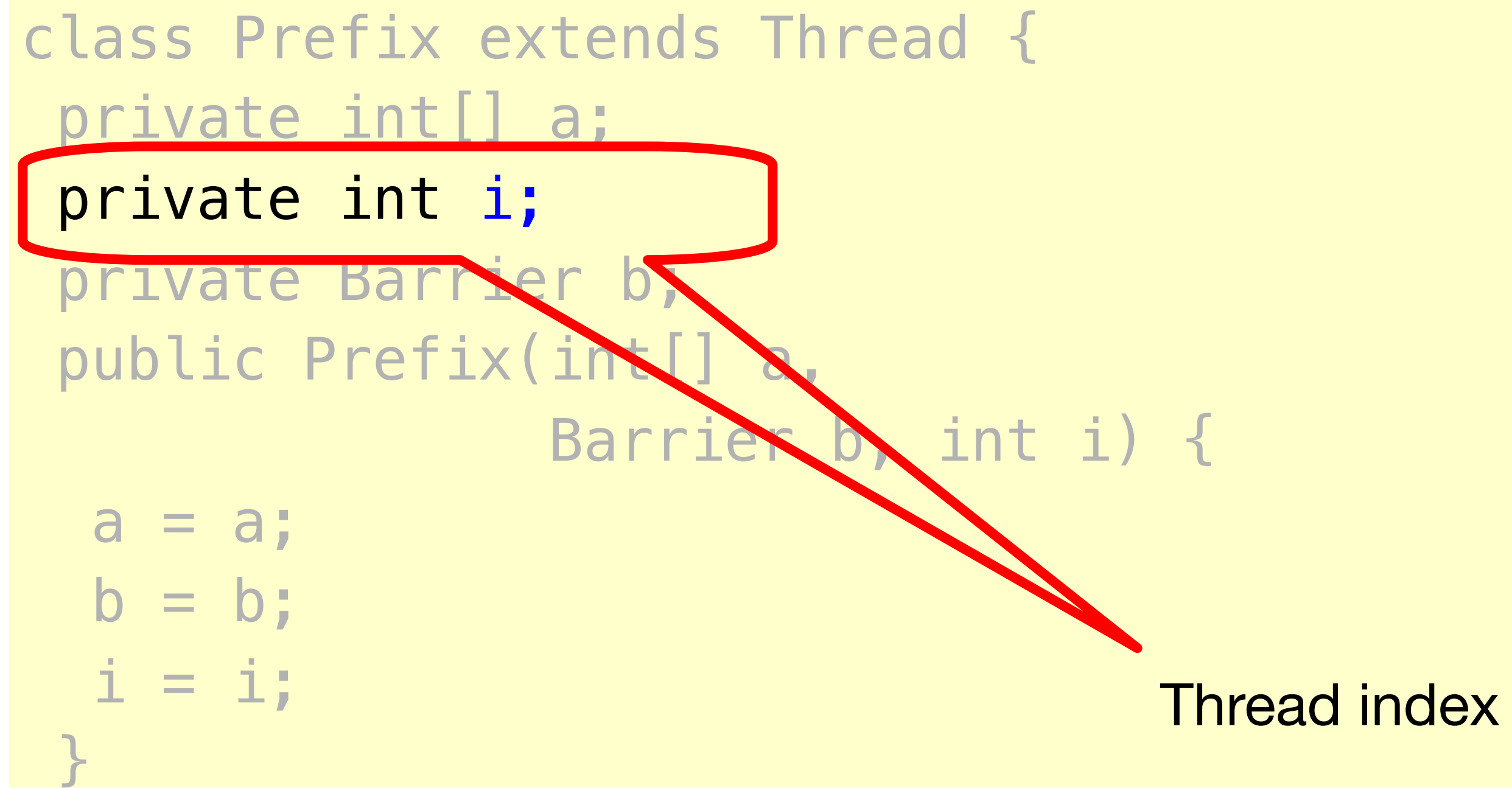


Array of input values

Prefix

```
class Prefix extends Thread {  
    private int[] a;  
    private int i;  
    private Barrier b;  
    public Prefix(int[] a,  
                 Barrier b, int i) {  
  
        a = a;  
        b = b;  
        i = i;  
    }  
}
```

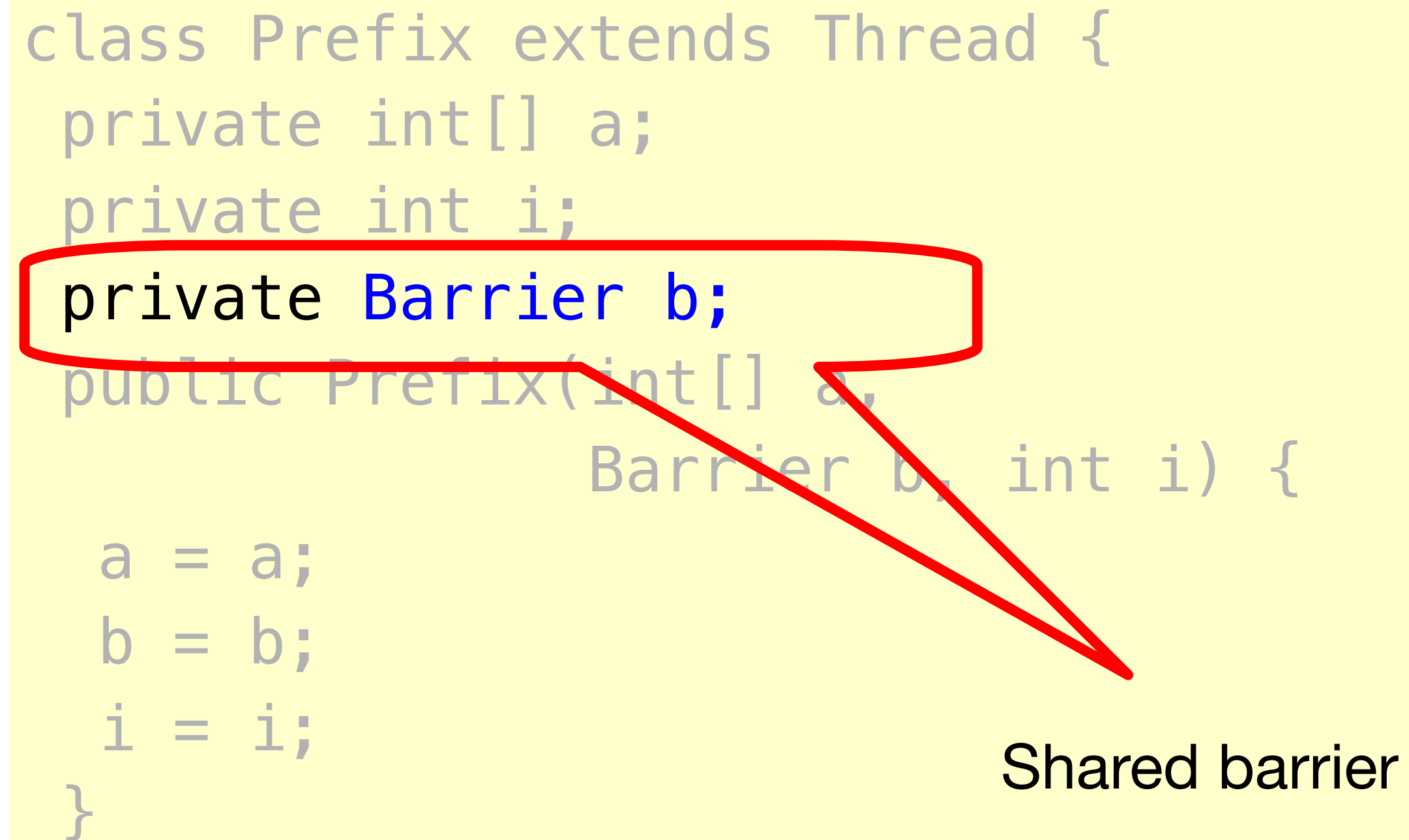
Thread index



Prefix

```
class Prefix extends Thread {  
    private int[] a;  
    private int i;  
    private Barrier b;  
    public Prefix(int[] a,  
                 Barrier b, int i) {  
  
        a = a;  
        b = b;  
        i = i;  
    }  
}
```

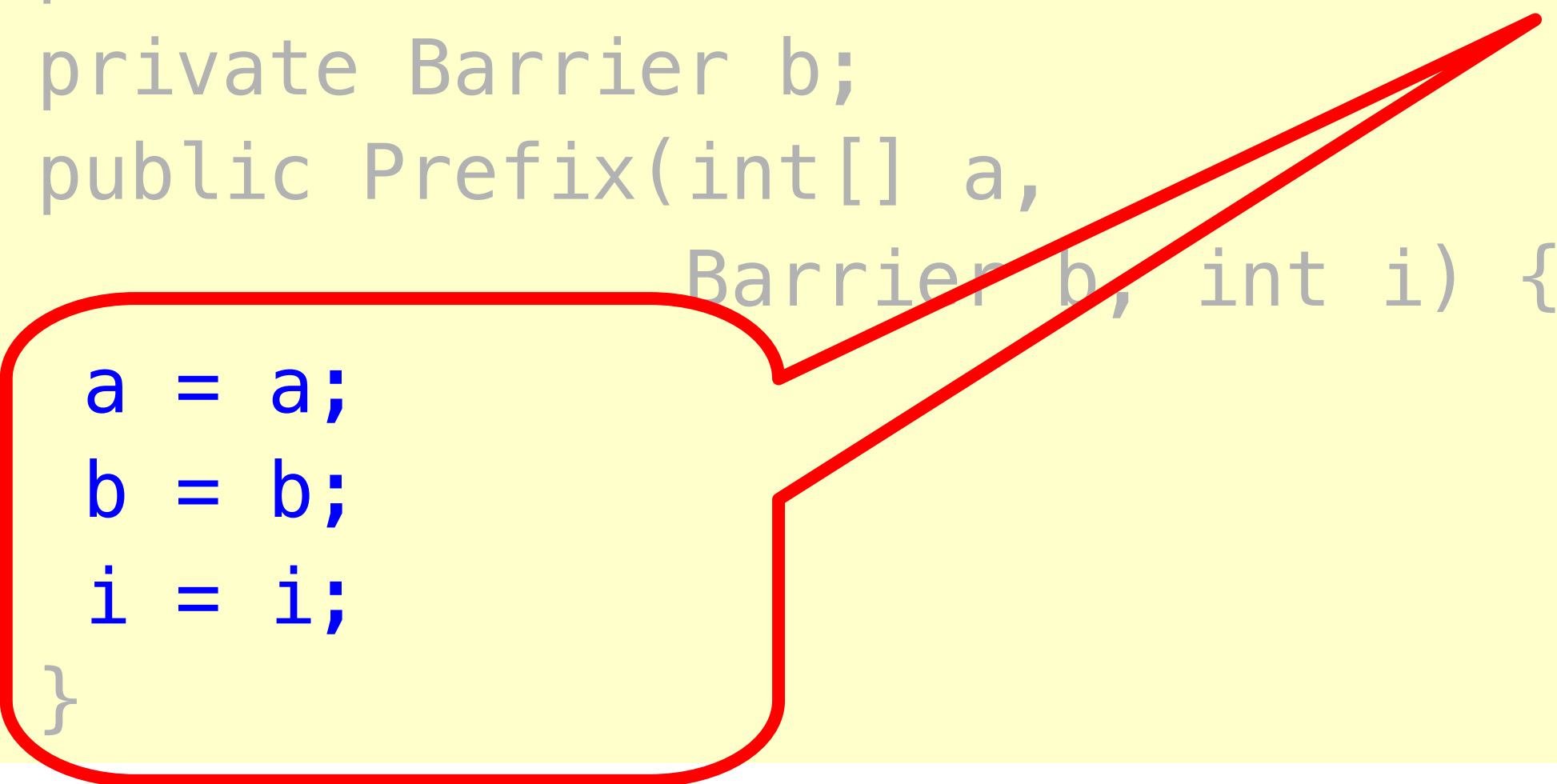
Shared barrier



Prefix

```
class Prefix extends Thread {  
    private int[] a;  
    private int i;  
    private Barrier b;  
    public Prefix(int[] a,  
                 Barrier b, int i) {  
        a = a;  
        b = b;  
        i = i;  
    }  
}
```

Initialize fields



Where Do the Barriers Go?

```
public void run() {  
    int d = 1, sum = 0;  
    while (d < N) {  
        if (i >= d)  
            sum = a[i-d];  
        if (i >= d)  
            a[i] += sum;  
        d = d * 2;  
    }  
}
```

Where Do the Barriers Go?

```
public void run() {
    int d = 1, sum = 0;
    while (d < N) {
        if (i >= d)
            sum = a[i-d];
        b.await();
        if (i >= d)
            a[i] += sum;
        d = d * 2;
    }
}
```

Where Do the Barriers Go?

```
public void run() {  
    int d = 1, sum = 0;  
    while (d < N) {  
        if (i >= d)  
            sum = a[i-d];  
        b.await();  
        if (i >= d)  
            a[i] += sum;  
        d = d * 2;  
    }  
}
```

Make sure everyone reads before anyone writes

Where Do the Barriers Go?

```
public void run() {  
    int d = 1, sum = 0;  
    while (d < N) {  
        if (i >= d)  
            sum = a[i-d];  
        b.await();  
        if (i >= d)  
            a[i] += sum;  
        b.await();  
        d = d * 2;  
    }  
}
```

Make sure everyone reads before anyone writes

Where Do the Barriers Go?

```
public void run() {
    int d = 1, sum = 0;
    while (d < N) {
        if (i >= d)
            sum = a[i-d];
        b.await();
        if (i >= d)
            a[i] += sum;
        b.await();
        d = d * 2;
    }
}
```

Make sure everyone reads before anyone writes

Make sure everyone writes before anyone reads

Barrier Implementations

- Cache coherence
 - Spin on locally-cached locations?
 - Spin on statically-defined locations?
- Latency
 - How many steps?
- Symmetry
 - Do all threads do the same thing?

Barriers

```
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
```

Barriers

```
public class Barrier {  
    AtomicInteger count;  
    int size;  
    public Barrier(int n){  
        count = AtomicInteger(n);  
        size = n;  
    }  
    public void await() {  
        if (count.getAndDecrement()==1) {  
            count.set(size);  
        } else {  
            while (count.get() != 0);  
        }  
    }  
}
```

Number threads not yet arrived

Barriers

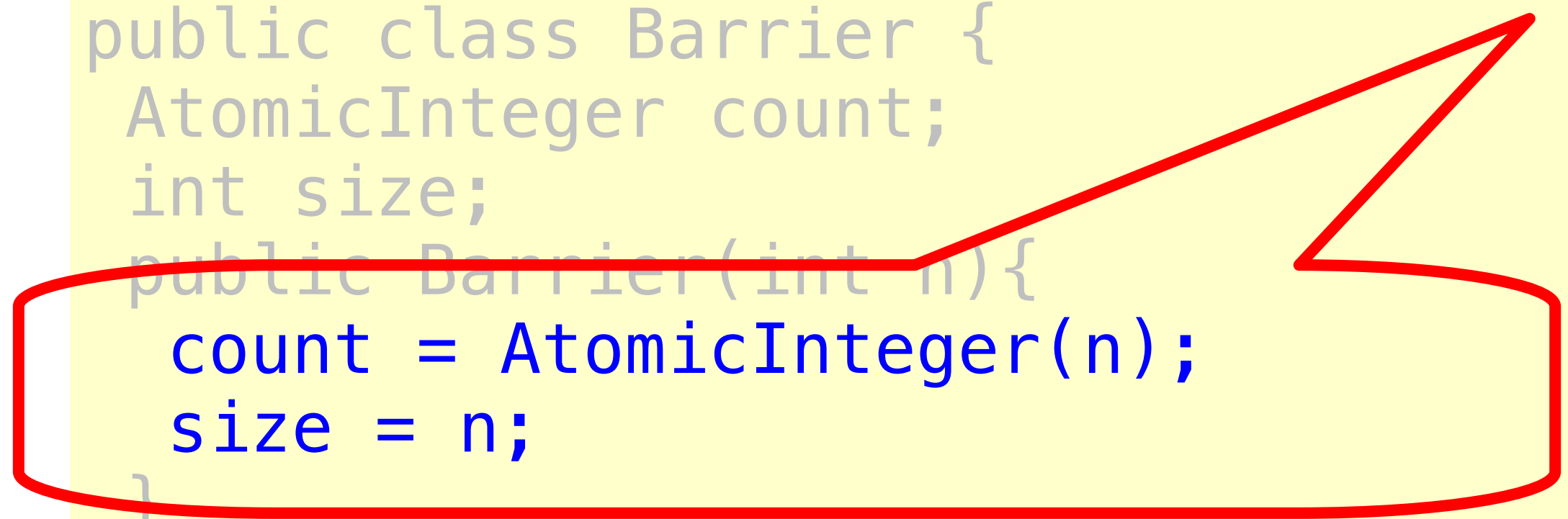
```
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
```

Number threads participating

Barriers

```
public class Barrier {  
    AtomicInteger count;  
    int size;  
    public Barrier(int n) {  
        count = AtomicInteger(n);  
        size = n;  
    }  
    public void await() {  
        if (count.getAndDecrement() == 1) {  
            count.set(size);  
        } else {  
            while (count.get() != 0);  
        }  
    }  
}
```

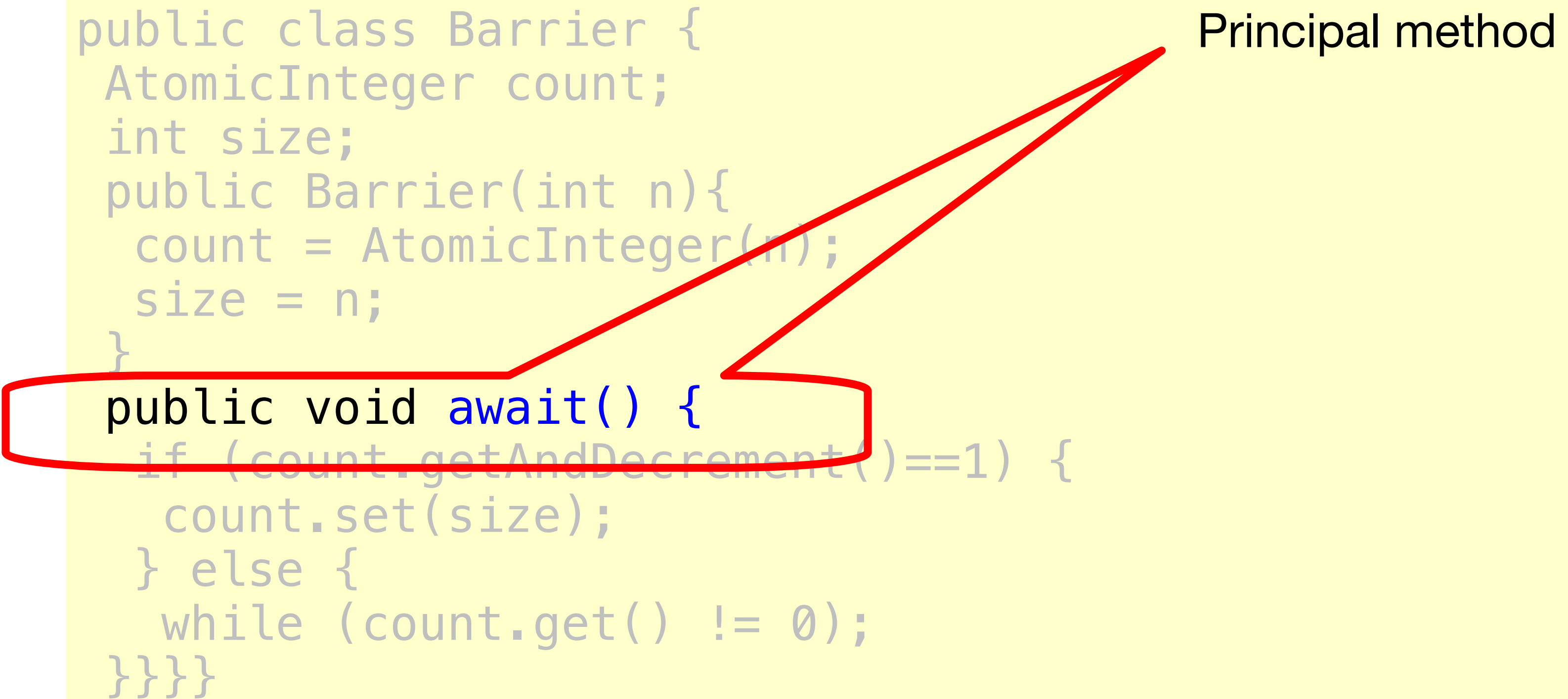
Initialization



Barriers

```
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
```

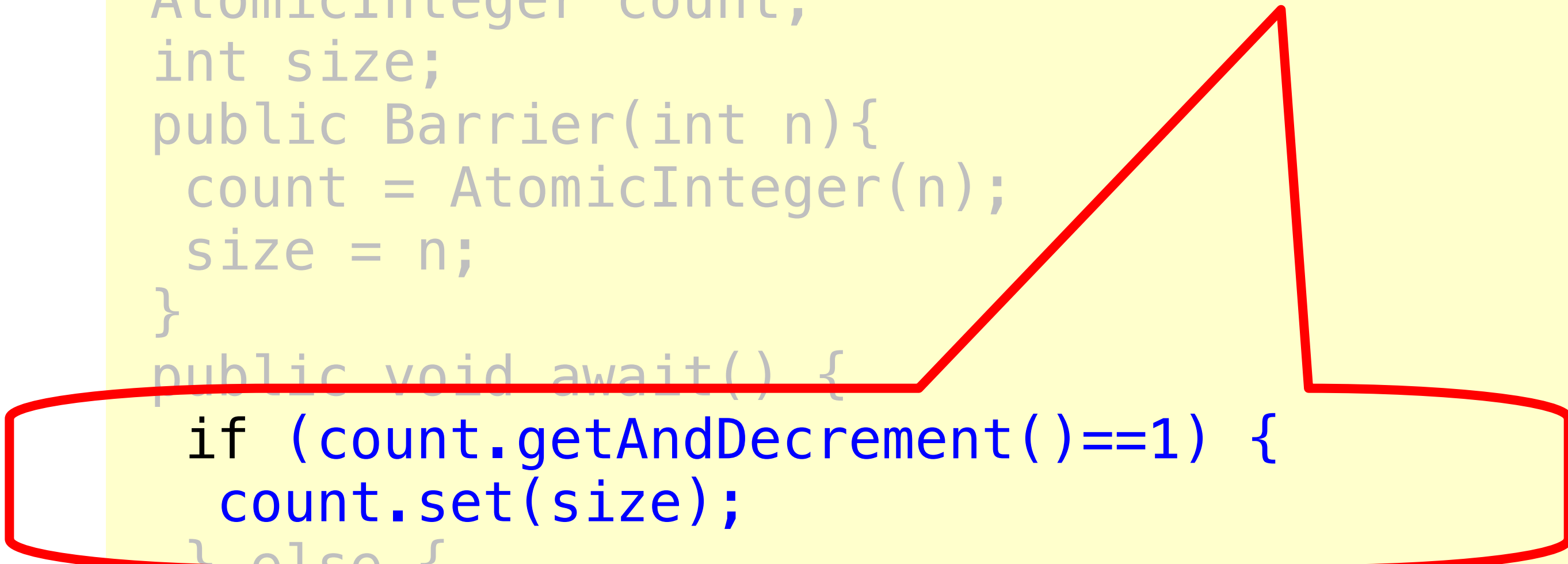
Principal method



Barriers

```
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
```

If I'm last, reset fields for next time



Barriers

Otherwise, wait for everyone else

```
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
```


Barriers

```
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
```

What's wrong with this protocol?

Reuse

```
Barrier b = new Barrier(n);  
while ( mumble() ) {  
    work();  
    b.await();  
}
```

Do work

synchronize

repeat

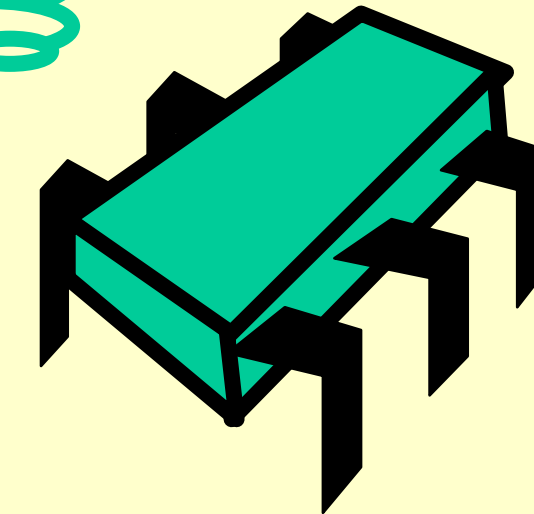
Barriers

```
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
```

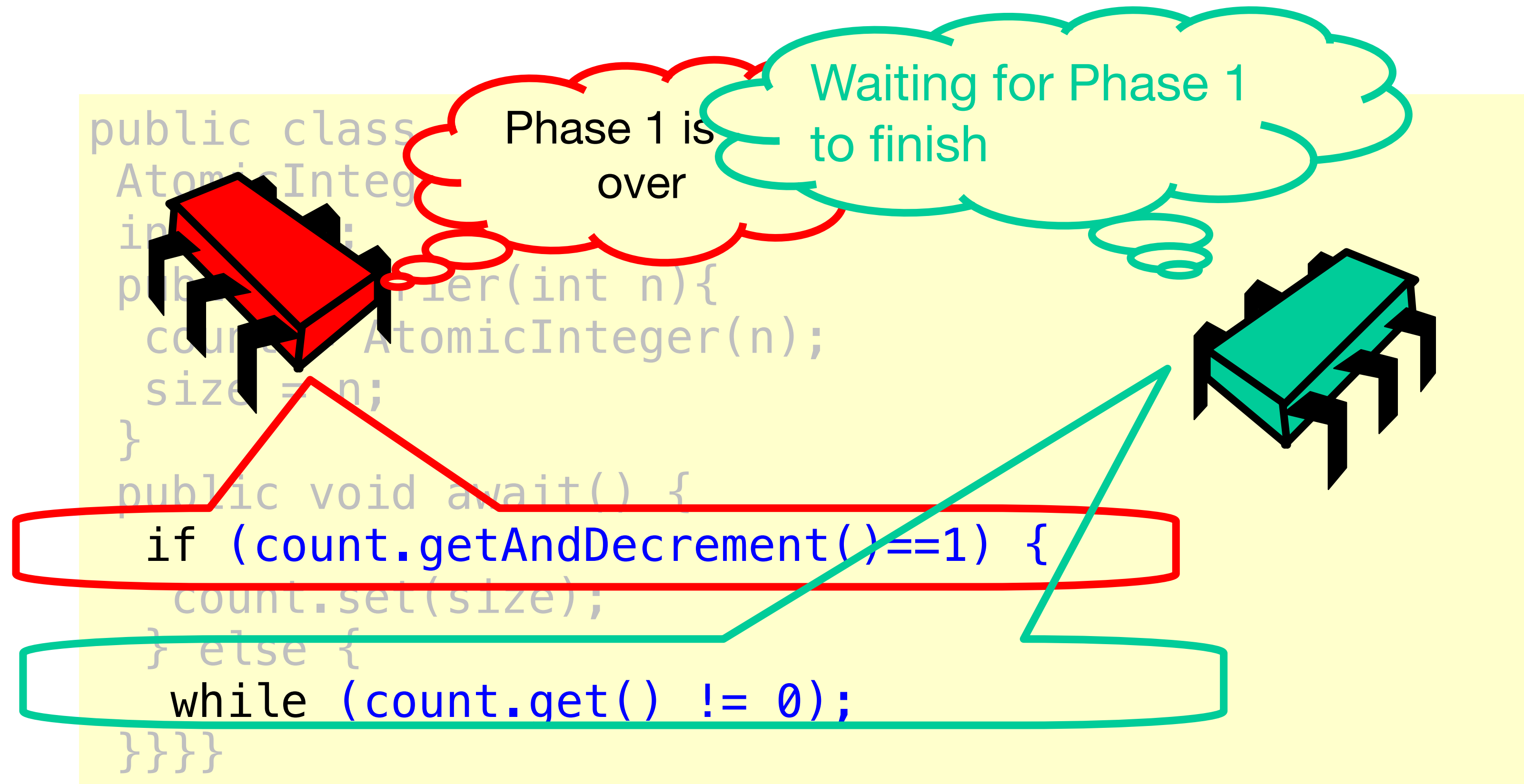
Barriers

```
public class Barrier {  
    AtomicInteger count;  
    int size;  
    public Barrier(int n){  
        count = AtomicInteger(n);  
        size = n;  
    }  
    public void await() {  
        if (count.getAndDecrement()==1) {  
            count.set(size);  
        } else {  
            while (count.get() != 0);  
        }  
    }  
}
```

Waiting for Phase 1
to finish



Barriers

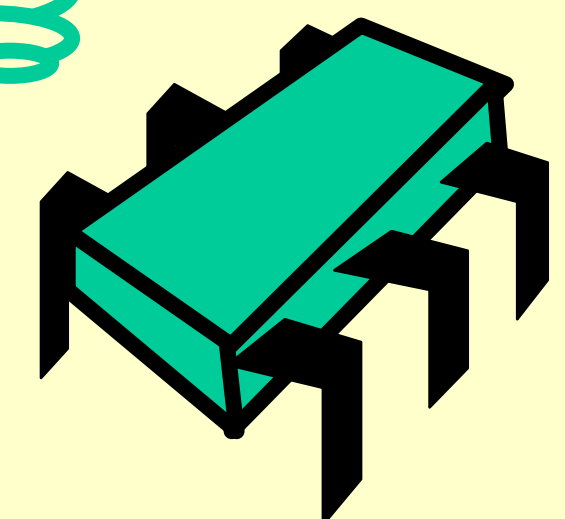
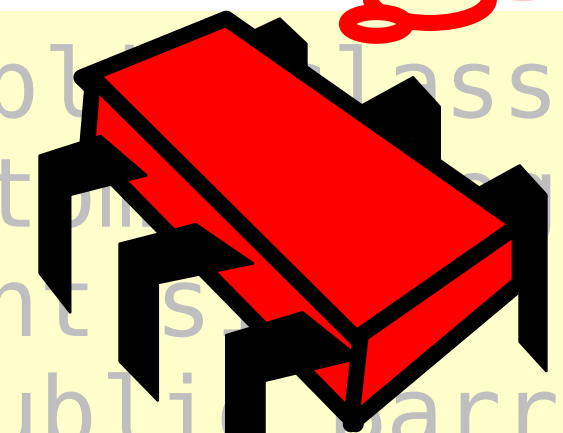


Barriers

Prepare for
phase 2

Waiting for Phase 1
to finish

```
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
```



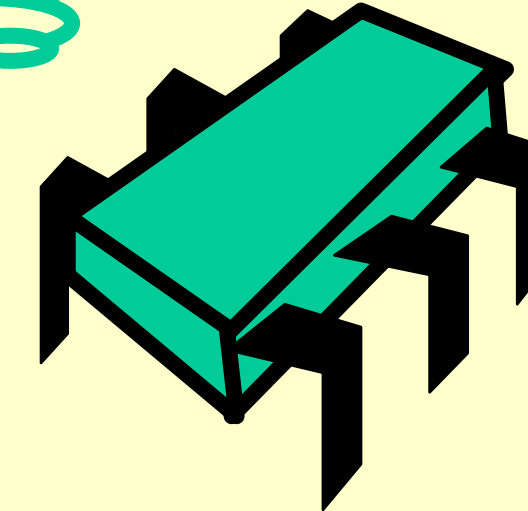
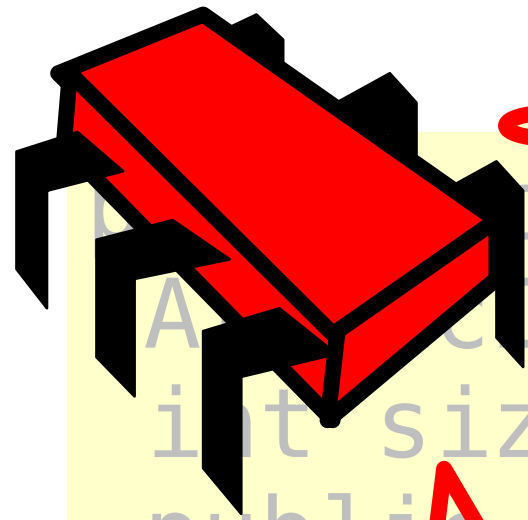
count.set(size);

while (count.get() != 0);

Uh-Oh

Waiting for Phase 2
to finish

Waiting for Phase 1
to finish



```
class Barrier {  
    AtomicInteger count;  
    int size;  
    public Barrier(int n){  
        count = AtomicInteger(n);  
        size = n;  
    }  
    public void await() {  
        if (count.getAndDecrement()==1) {  
            count.set(size);  
        } else {  
            while (count.get() != 0);  
        }  
    }  
}
```

Basic Problem

- One thread “wraps around” to start phase 2
- While another thread is still waiting for phase 1
- One solution:
 - Always use two barriers

Sense-Reversing Barriers

```
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}}}
```

Sense-Reversing Barriers

```
public class Barrier {  
    AtomicInteger count;  
    int size;  
    boolean sense = false;  
    ThreadSense = new ThreadLocal<boolean>...  
  
    public void await {  
        boolean mySense = threadSense.get();  
        if (count.getAndDecrement()==1) {  
            count.set(size); sense = mySense  
        } else {  
            while (sense != mySense) {}  
        }  
        threadSense.set(!mySense)}}}
```

Completed odd or even-numbered phase?

Sense-Reversing Barriers

```
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>...
}

public void await {
    boolean mySense = threadSense.get();
    if (count.getAndDecrement()==1) {
        count.set(size); sense = mySense
    } else {
        while (sense != mySense) {}
    }
    threadSense.set(!mySense)}}}
```

Store sense for next phase,
initialized to true

Sense-Reversing Barriers

```
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>();

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement() == 1) {
            count.set(size); sense = mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}}}
```

Get new sense determined by last phase

Sense-Reversing Barriers

```
public class Barrier {
    AtomicInteger count;      If I'm last, reverse sense for next
    int size;                time
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}}}
```

Sense-Reversing Barriers

```
public class Barrier {
    AtomicInteger count;           Otherwise, wait for sense to flip
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}}}
```

Sense-Reversing Barriers

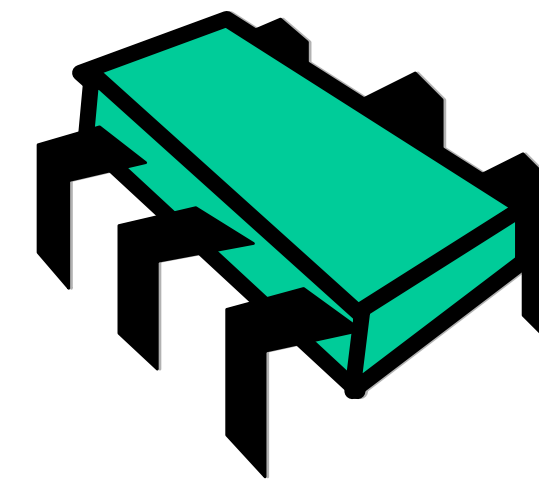
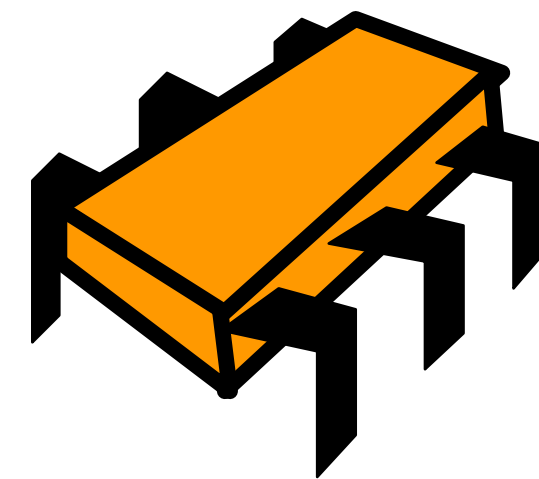
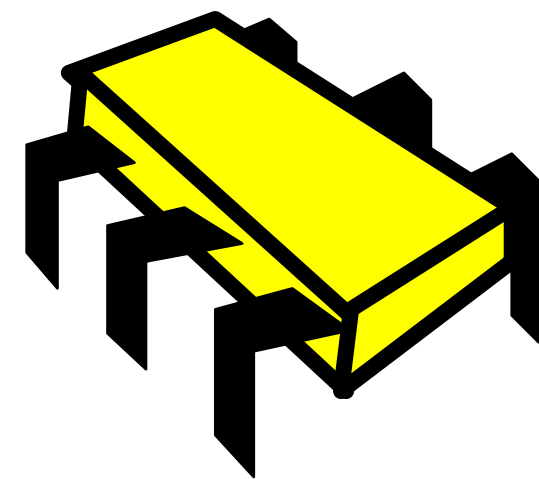
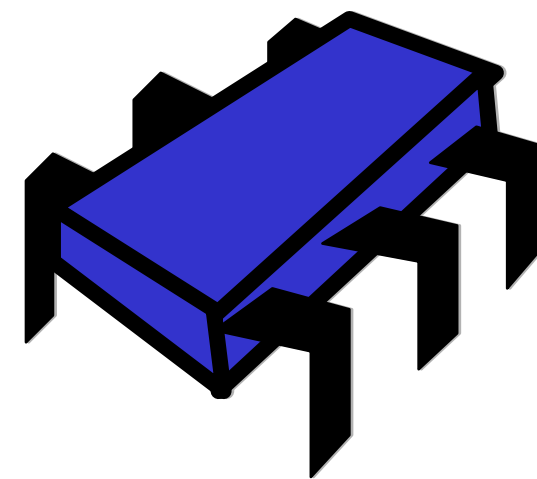
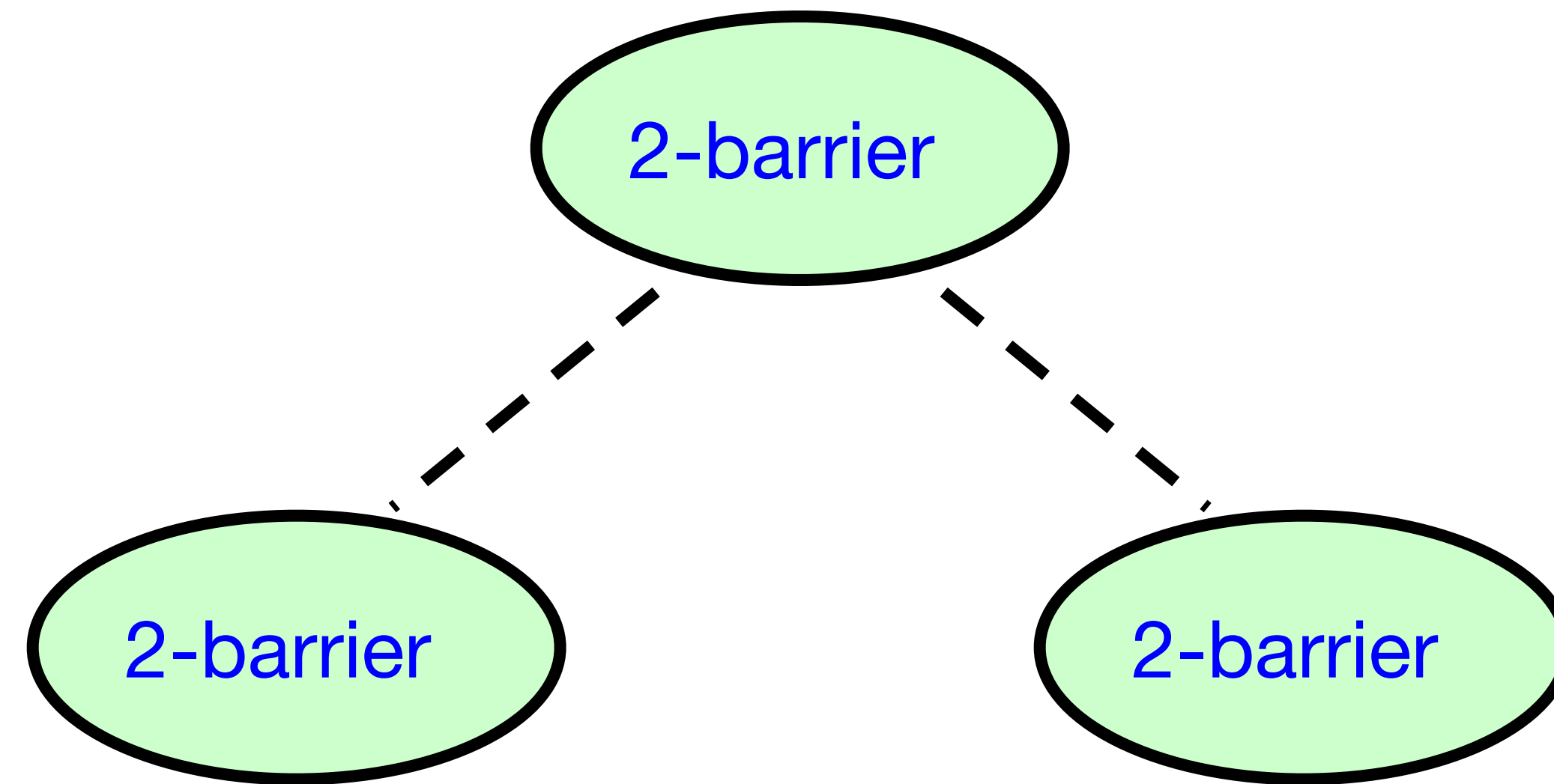
```
public class Barrier {
    AtomicInteger count;        Prepare sense for next phase
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }
    threadSense.set(!mySense)}}}
```

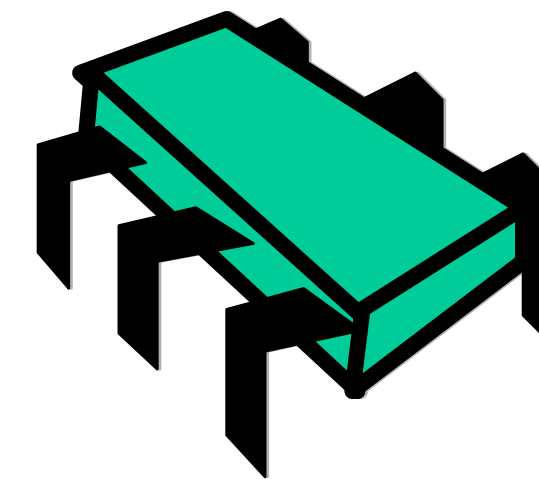
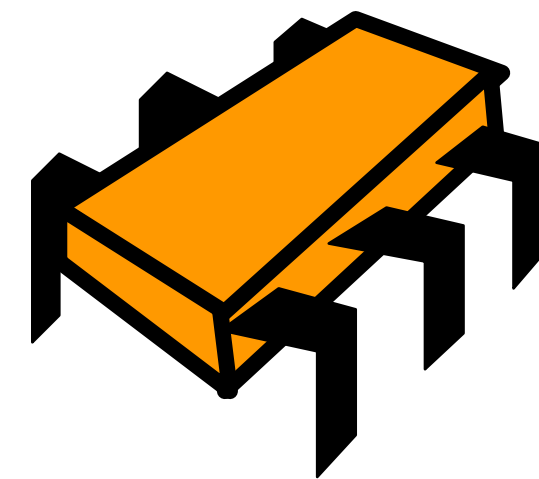
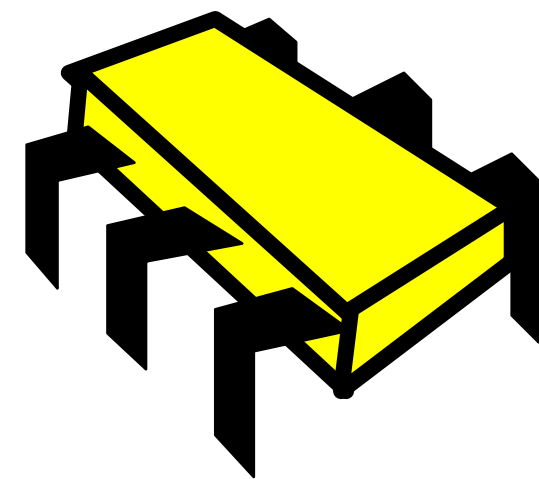
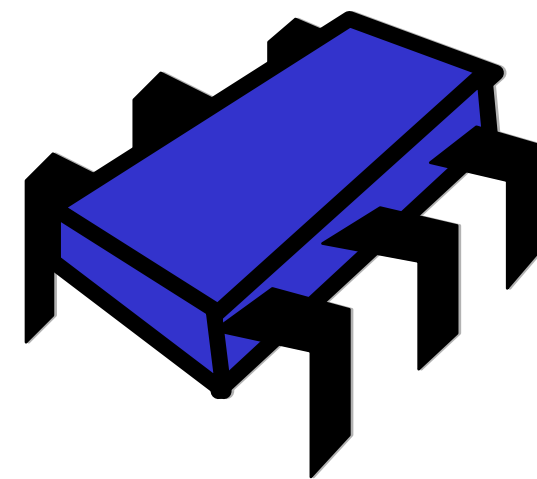
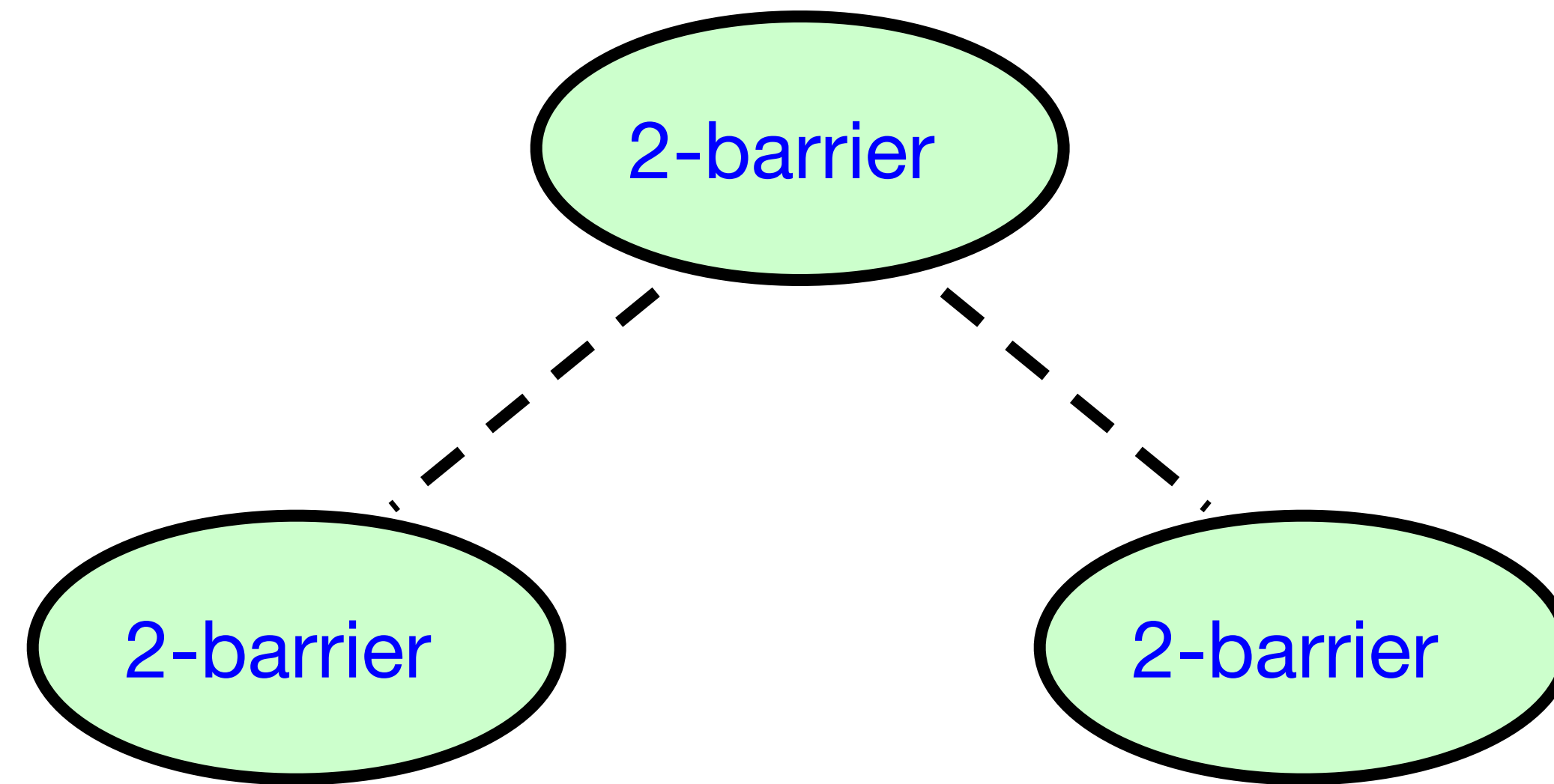
Sense Reversing Barriers

- Good parts:
 - Spinning happens on *sense* which doesn't get modified until all threads are done (no contention from spinning on the count value, which gets modified by each thread)
- Bad parts:
 - Hard to scale to hugely parallel situations — still have all n threads contending for that one *sense* field

Combining Tree Barriers



Combining Tree Barriers



Combining Tree Barrier

```
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await()}
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }...}}}
```

Combining Tree Barrier

```
public class Node{                                     Parent barrier in tree
    AtomicInteger count; int size;
    Node parent; volatile boolean sense;

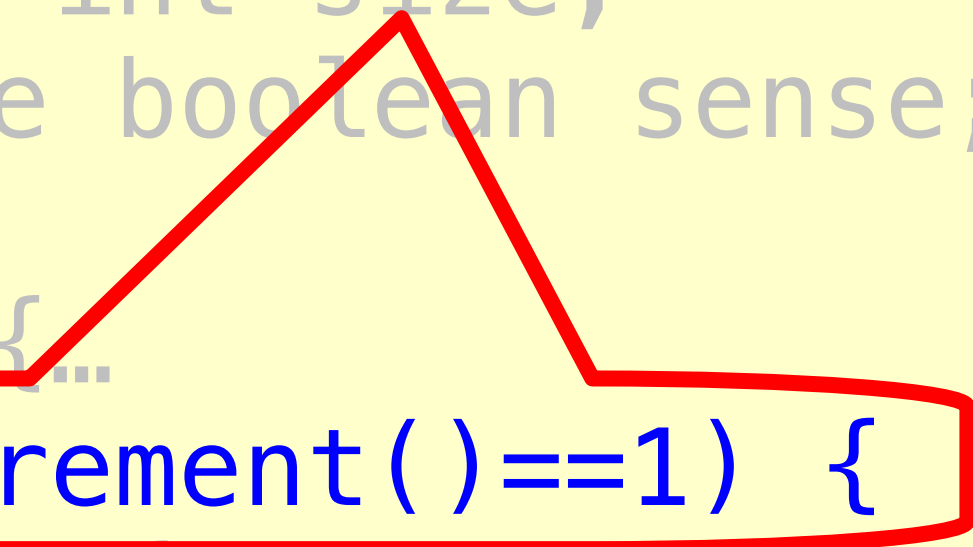
    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await()}
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }...}}}
```

Combining Tree Barrier

```
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await();
            }
            count.set(size);
            sense = mySense;
        } else {
            while (sense != mySense) {}
        }
    }...}}}
```

Am I last?



Combining Tree Barrier

```
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await();}
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }...}}}
```

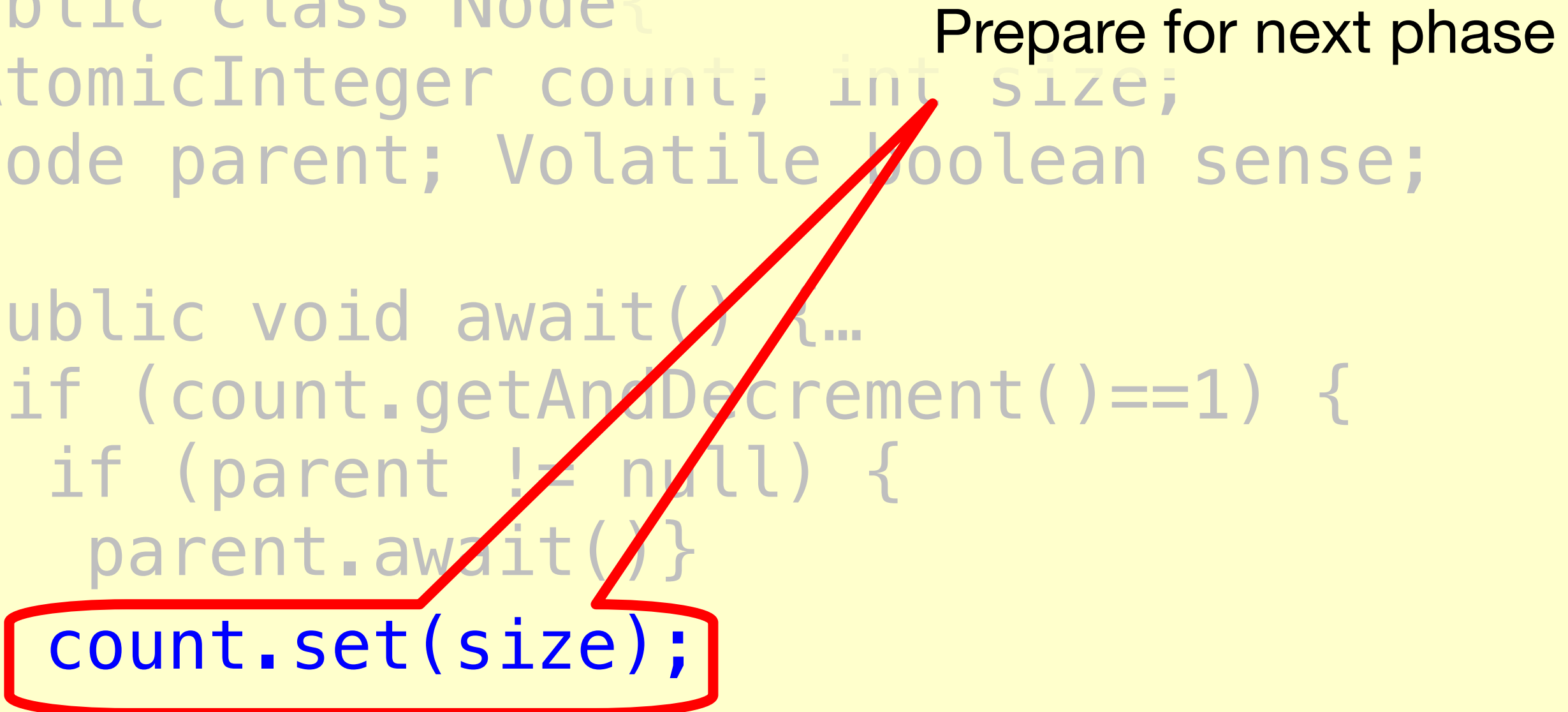
Proceed to parent barrier

Combining Tree Barrier

```
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await();
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }...}}}
```

Prepare for next phase



Combining Tree Barrier

```
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await();
                count.set(size);
                sense = mySense
            } else {
                while (sense != mySense) {}
            }
        }
    }
}
```

Notify others at this node

Combining Tree Barrier

```
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await()}
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }...}}}
```

I'm not last, so wait for notification

Combining Tree Barrier

- No sequential bottleneck
 - Parallel `getAndDecrement()` calls
- Low memory contention
 - Same reason
- Cache behavior
 - Local spinning on bus-based architecture
 - Not so good for non-uniform memory access architectures (NUMA) - common for large multiprocessor systems

Remarks

- Everyone spins on sense field
 - Local spinning on bus-based (good)
 - Network hot-spot on distributed architecture (bad)
- Not really scalable
 - Nodes going up and down the tree are not predictable - which (left or right) goes up? Whichever goes first -> possible contention and cache misses

So...How will we make use of multicores?

Back to Amdahl's Law:

$$\text{Speedup} = 1/(\text{ParallelPart}/N + \text{SequentialPart})$$

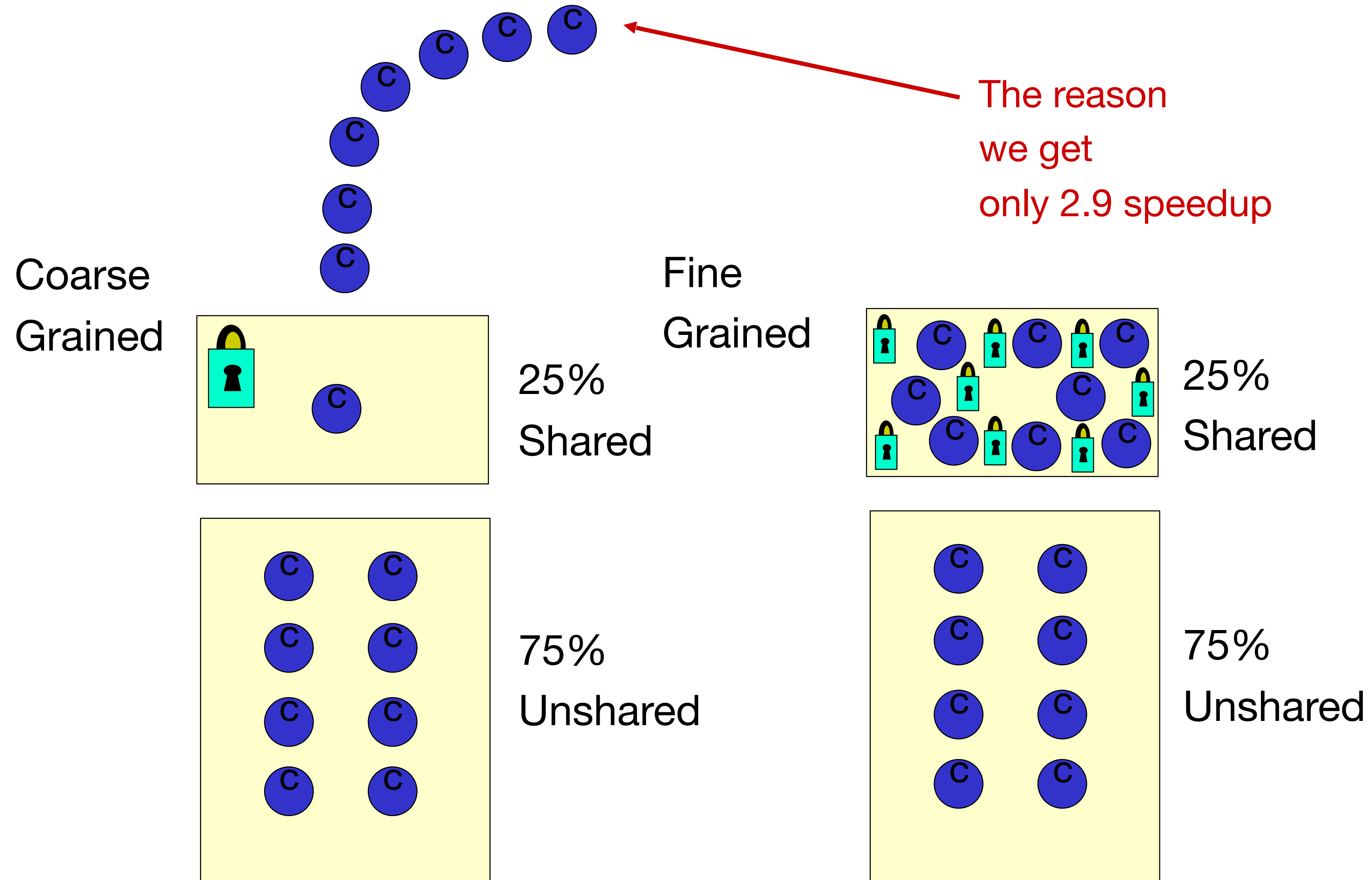
Pay for $N = 8$ cores

$$\text{SequentialPart} = 25\%$$

Speedup = only 2.9 times!

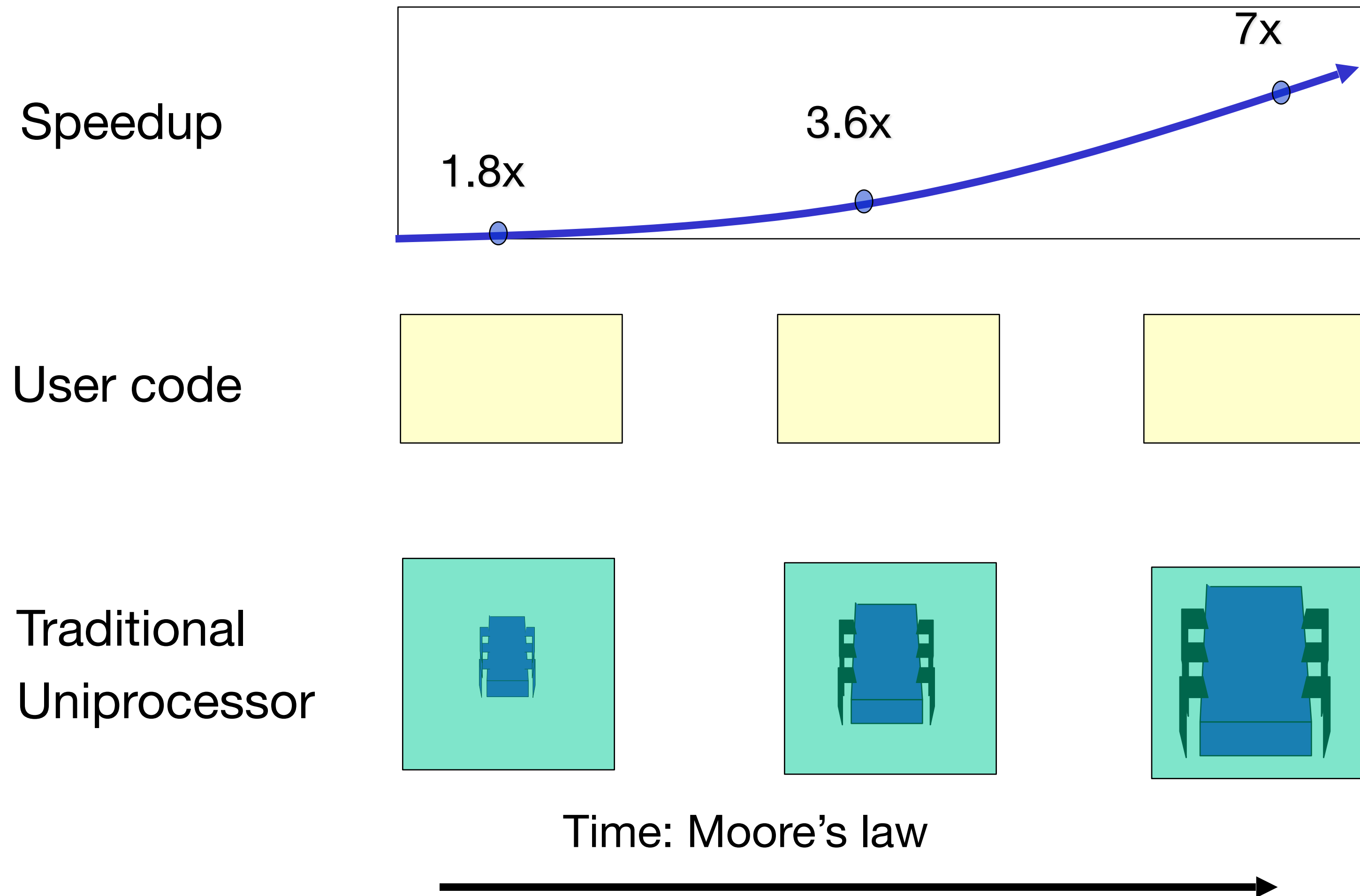
Must parallelize applications on a very fine grain!

Need Fine-Grained Locking

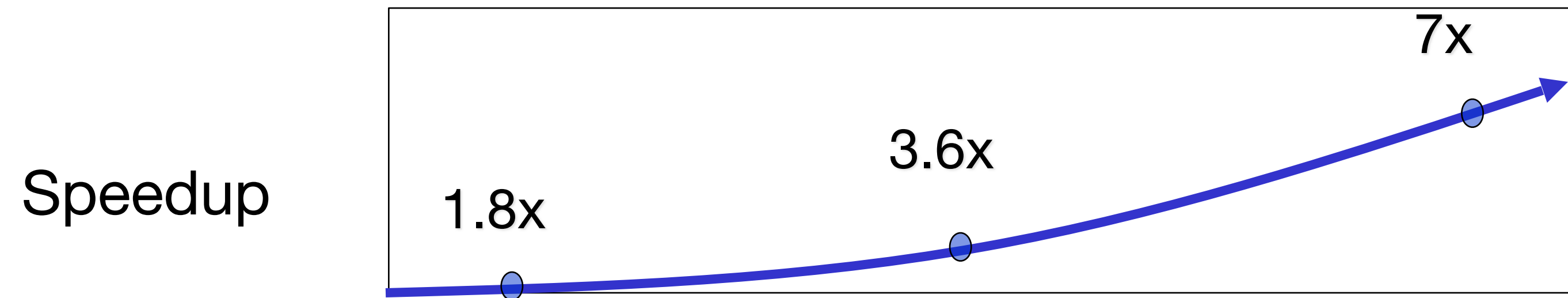


The reason we get only 2.9 speedup

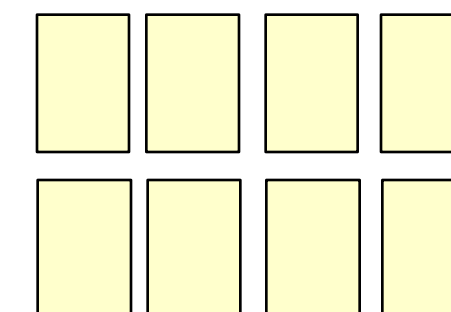
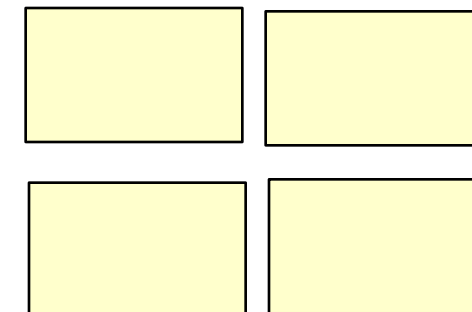
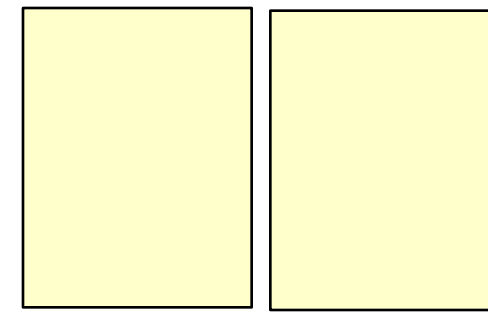
Traditional Scaling Process



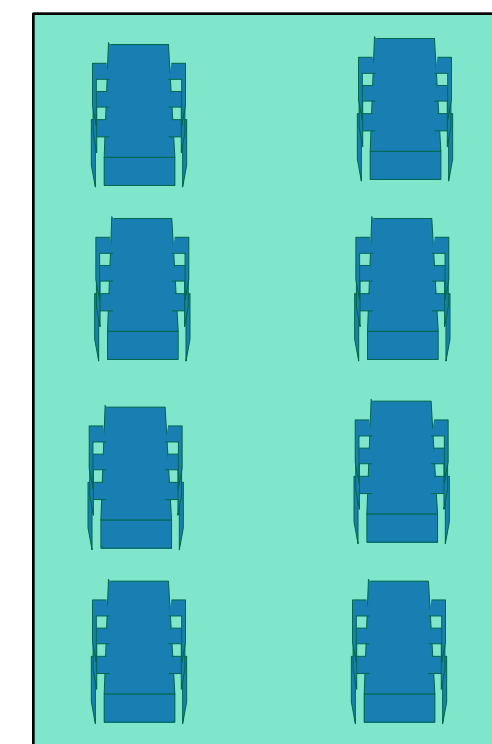
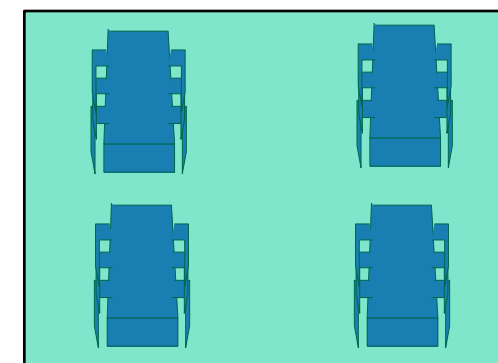
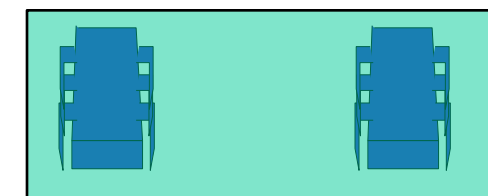
Multicore Scaling Process



User code

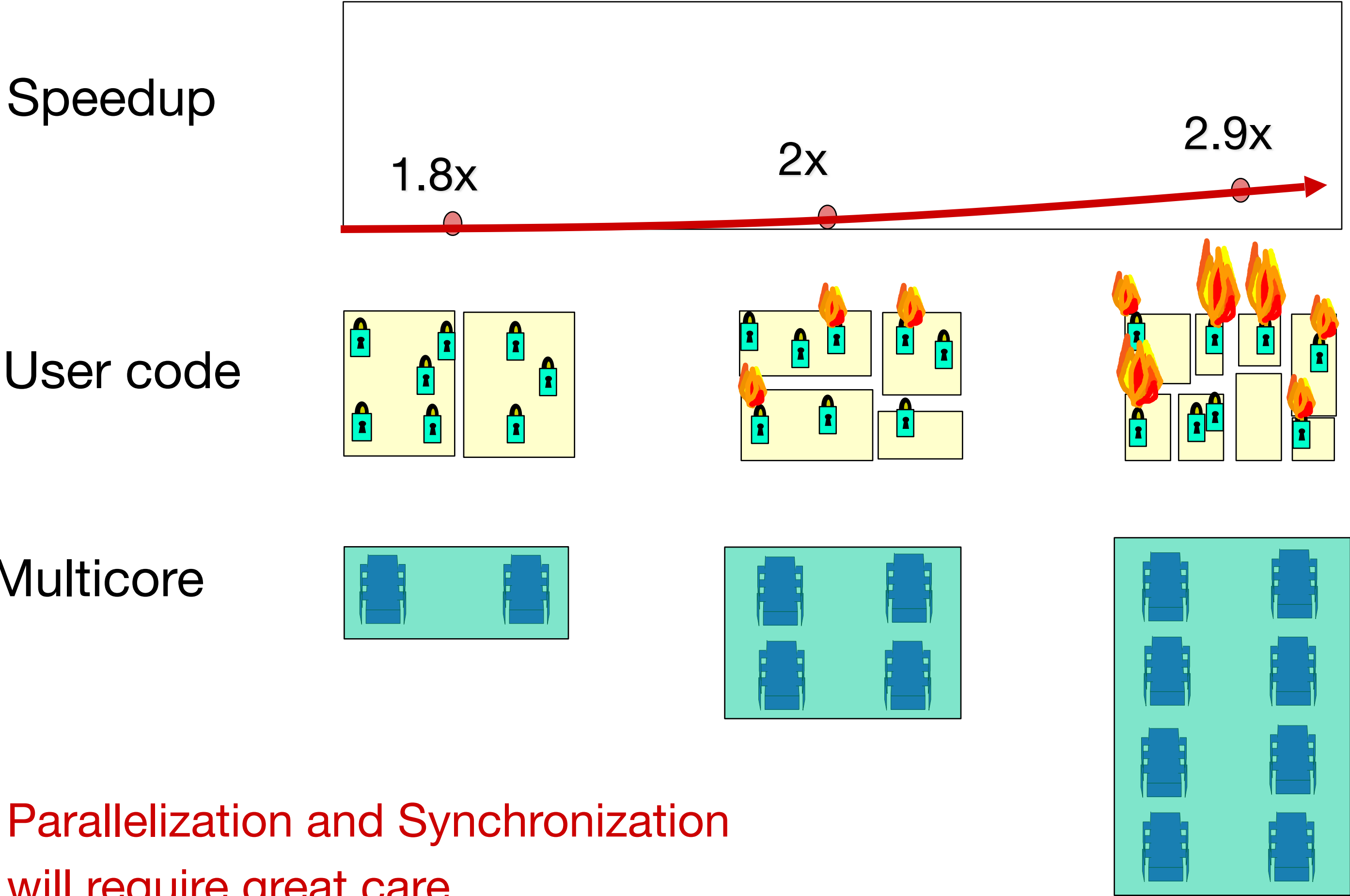


Multicore



As noted, not so simple...

Real-World Scaling Process



Multicore Programming

- Nothing is free - if we want to take advantage of multicore machines, we need to work hard at designing our algorithms and systems to be amenable to parallelization
- We are at the dawn of a new era...
 - Still a lot of research and development to be done
- Moving on now: what happens when each CPU is on a different physical machine, connected by a network?

This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/>
- You are free to:
 - Share — copy and redistribute the material in any medium or format
 - Adapt — remix, transform, and build upon the material
 - for any purpose, even commercially.
- Under the following terms:
 - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
 - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
 - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.