Agenda

- Lecture
  - Bottom-up motivation
  - Shared memory primitives
  - Shared memory synchronization
    - Barriers and locks
- Next discussion papers
  - Selecting Locking Primitives for Parallel Programming
  - Selecting Locking Designs for Parallel Programs

Acknowledgments

- Pseudo code from:
  - "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", Mellor-Crummey & Scott, ACM TOCS, Feb 1991
  - http://www.cs.rochester.edu/research/synchronization/pseudocode/ss.html

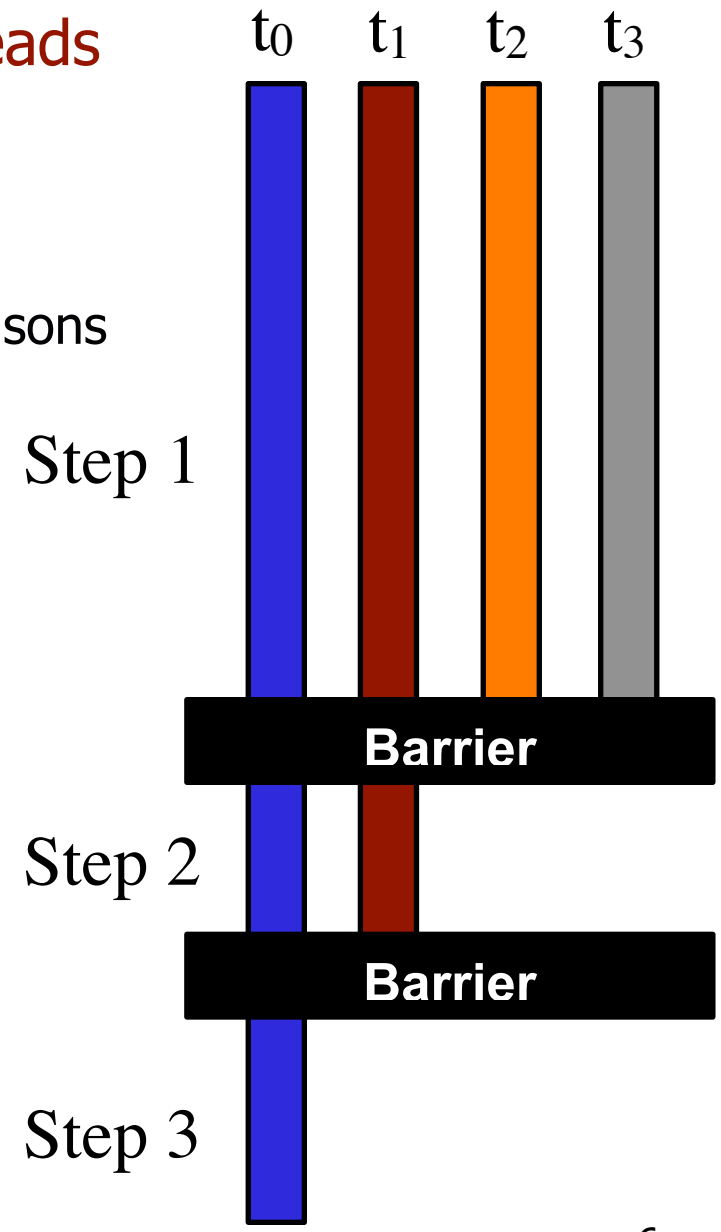
Barriers

Common Parallel Idiom: Barriers

- Physics simulation computation
  - Divide up each timestep computation into N independent pieces
  - Each timestep: compute independently, synchronize
- Example: each thread executes:
  - segment_size = total_particles / **number_of_threads**
  - my_start_particle = thread_id * segment_size
  - my_end_particle = my_start_particle + segment_size - 1
  - for (timestep = 0; timestep += delta; timestep < stop_time):
    - calculate_forces(t, my_start_particle, my_end_particle)
    - **barrier()**
    - update_locations(t, my_start_particle, my_end_particle)
    - **barrier()**
- Barrier? All threads wait until all threads have reached it

Example: Barrier-Based Merge Sort

- Merge-sort 4096 elements with four threads
- Step #1:
  - Sort each 1/4th of array
  - (N/4)*log(N/4) = 1024*10 = 10240 comparisons
- Step #2:
  - Two N/2 merges
  - 2048 comparisons
- Step #3:
  - Final merge
  - 4096 comparisons
- Total: 3x speed up four threads
  - Parallel: 16384 comparisons
  - Sequential: ~50k comparisons



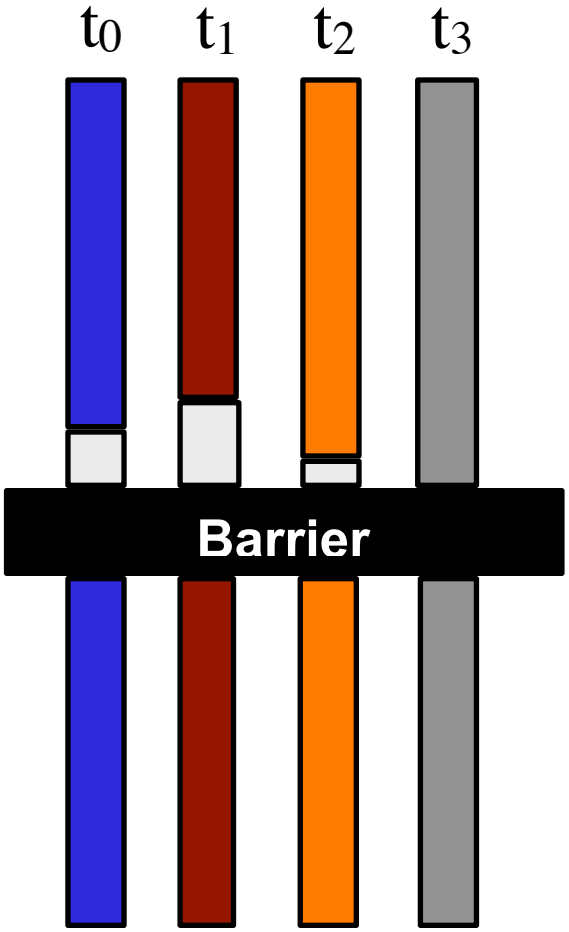
Global Synchronization Barrier

- At a barrier
  - All threads wait until all other threads have reached it
- Strawman implementation (**wrong!**)

```
global (shared) count : integer := P

procedure central_barrier
  if fetch_and_decrement(&count) == 1
    count := P
  else
    repeat until count == P
```

- What is wrong with the above code?



Sense-Reversing Barrier

- Correct barrier implementation:

```
global (shared) count : integer := P
global (shared) sense : Boolean := true
local (private) local_sense : Boolean := true

procedure central_barrier
  // each processor toggles its own sense
  local_sense := !local_sense
  if fetch_and_decrement(&count) == 1
    count := P
    // last processor toggles global sense
    sense := local_sense
  else
    repeat until sense == local_sense
```

- Single counter makes this a "centralized" barrier

Other Barrier Implementations

- Problem with centralized barrier
  - All processors must increment each counter
  - Each read/modify/write is a serialized coherence action
    - Each one is a cache miss
  - O(n) if threads arrive simultaneously, slow for lots of processors
- Combining Tree Barrier
  - Build a $\log_k(n)$ height tree of counters (one per cache block)
  - Each thread coordinates with **k** other threads (by thread id)
  - Last of the **k** processors, coordinates with next higher node in tree
  - As many coordination address are used, misses are not serialized
  - O(log n) in best case
- Static and more dynamic variants
  - Tree-based arrival, tree-based or centralized release

Barrier Performance (from 1991)

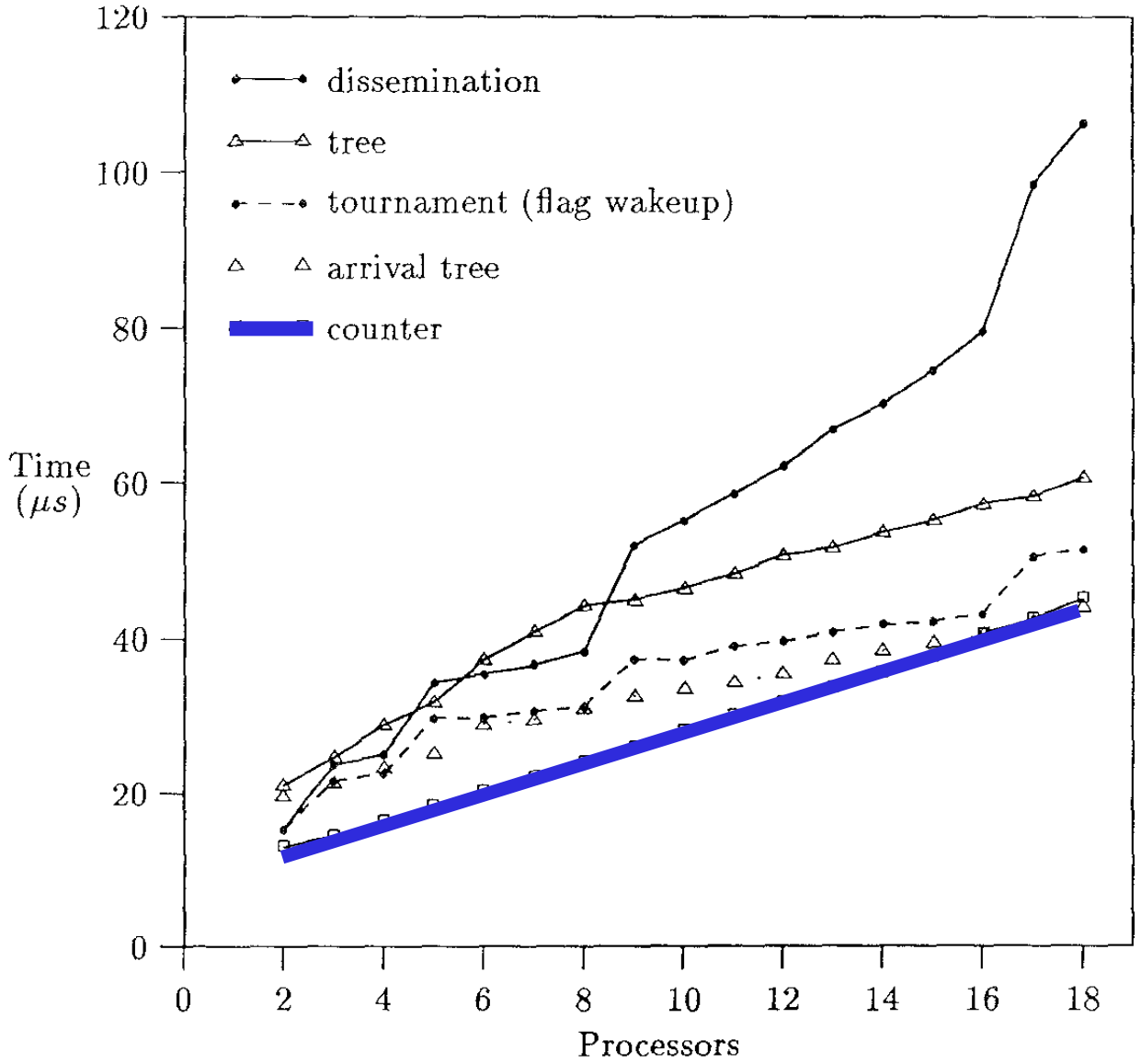


Fig. 21. Performance of barriers on the Symmetry

From Mellor-Crummey & Scott, ACM TOCS, 1991

Locks

Common Parallel Idiom: Locking

- Protecting a shared data structure
- Example: parallel tree walk, apply f() to each node
  - Global "set" object, initialized with pointer to root of tree
  - Each thread, while (true):
    - node* next = set.remove()
    - if next == NULL: return **// terminate thread**
    - func(code->value) **// computationally intense function**
    - if (next->right != NULL):
      - set.insert(next->right)
    - if (next->left != NULL):
      - set.insert(next->left)
- How do we protect the "set" data structure?
  - Also, to perform well, what element should it "remove" each step?

Common Parallel Idiom: Locking

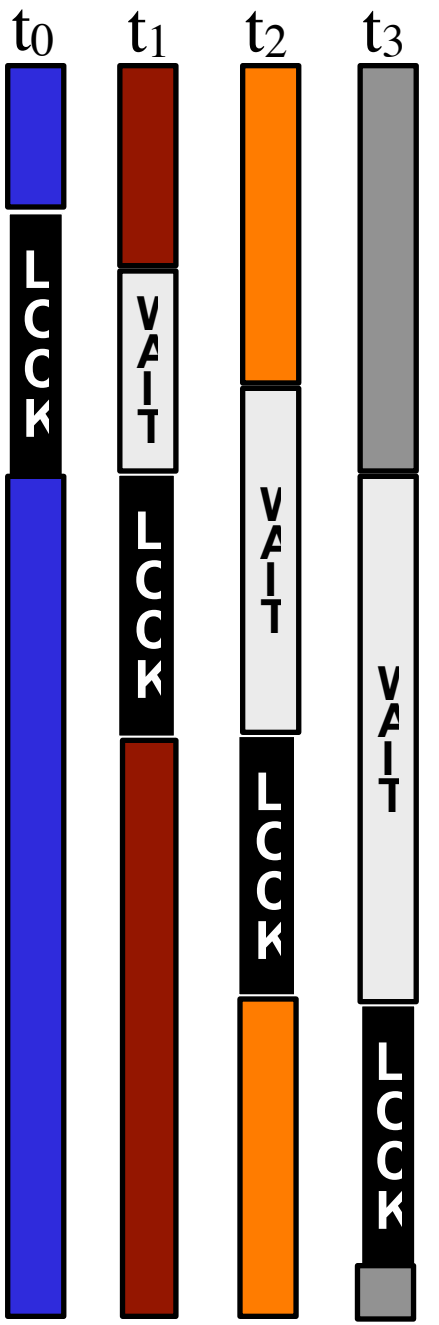
- Parallel tree walk, apply f() to each node
  - Global "set" object, initialized with pointer to root of tree
  - Each thread, while (true):
    - **acquire(set.lock_ptr)**
    - node* next = set.pop()
    - **release(set.lock_ptr)**
    - if next == NULL:
      - return **// terminate thread**
    - func(node->value) **// computationally intense**
    - **acquire(set.lock_ptr)**
    - if (next->right != NULL)
      - set.insert(next->right)
    - if (next->left != NULL)
      - set.insert(next->left)
    - **release(set.lock_ptr)**

Put lock/unlock into **pop**() method?

Put lock/unlock into **insert**() method?

Lock-Based Mutual Exclusion

- Only one thread can hold a "lock" at a time
  - Used a provide serialized access to a data object
- If another threads tries to acquire a held lock
  - Must wait until other thread performs a release
- Performance implications
  - Lock contention limits parallelism
  - Lock acquire/release time adds overheads
- Correctness implications
  - Just one example:
    - Thread #1: Holds lock A, tries to acquire B
    - Thread #2: Holds lock B, tries to acquire A
    - Classic deadlock!



Simple Boolean Spin Locks

- Simplest lock:
  - Single variable, two states: **locked**, **unlocked**
  - When unlocked: atomically transition from unlocked to locked
  - When locked: keep checking (spin) until the lock is unlocked
- Busy waiting versus "blocking"
  - In a multicore, **busy wait** for other thread to release lock
    - Likely to happen soon, assuming critical sections are small
    - Likely nothing "better" for the processor to do anyway
  - In a single processor, if trying to acquire a held lock, **block**
    - The only sensible option is to tell the O.S. to context switch
    - O.S. knows not to reschedule thread until lock is released
  - Blocking has high overhead (O.S. call)
    - IMHO, rarely makes sense in multicore (parallel) programs

Spin Locks and Contention

Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

Focus so far: Correctness

- Models
  - Accurate (we never lied to you)
  - But idealized (so we forgot to mention a few things)
- Protocols
  - Elegant
  - Important
  - But naïve

New Focus: Performance

- Models
  - More complicated (not the same as complex!)
  - Still focus on principles (not soon obsolete)
- Protocols
  - Elegant (in their fashion)
  - Important (why else would we pay attention)
  - And realistic (your mileage may vary)

What Should you do if you can't get a lock?

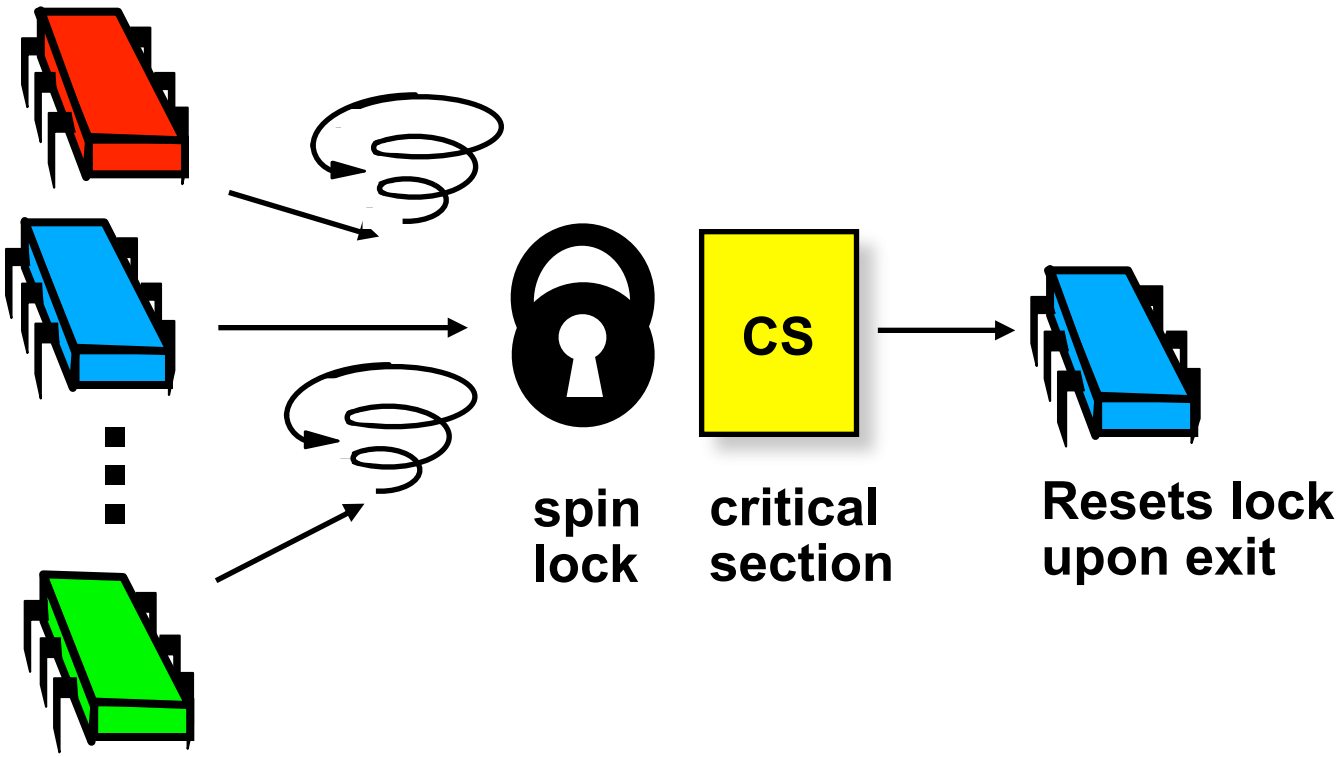
- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

What Should you do if you can't get a lock?

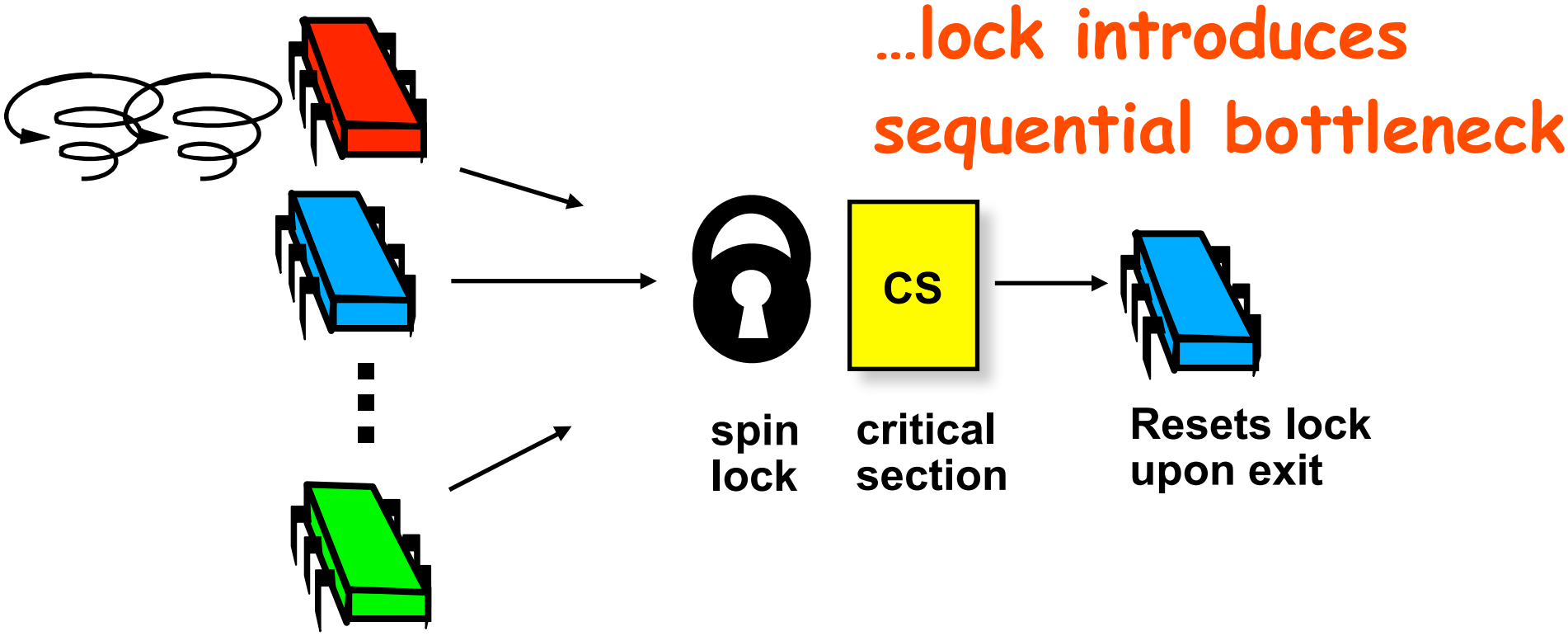
- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

our focus

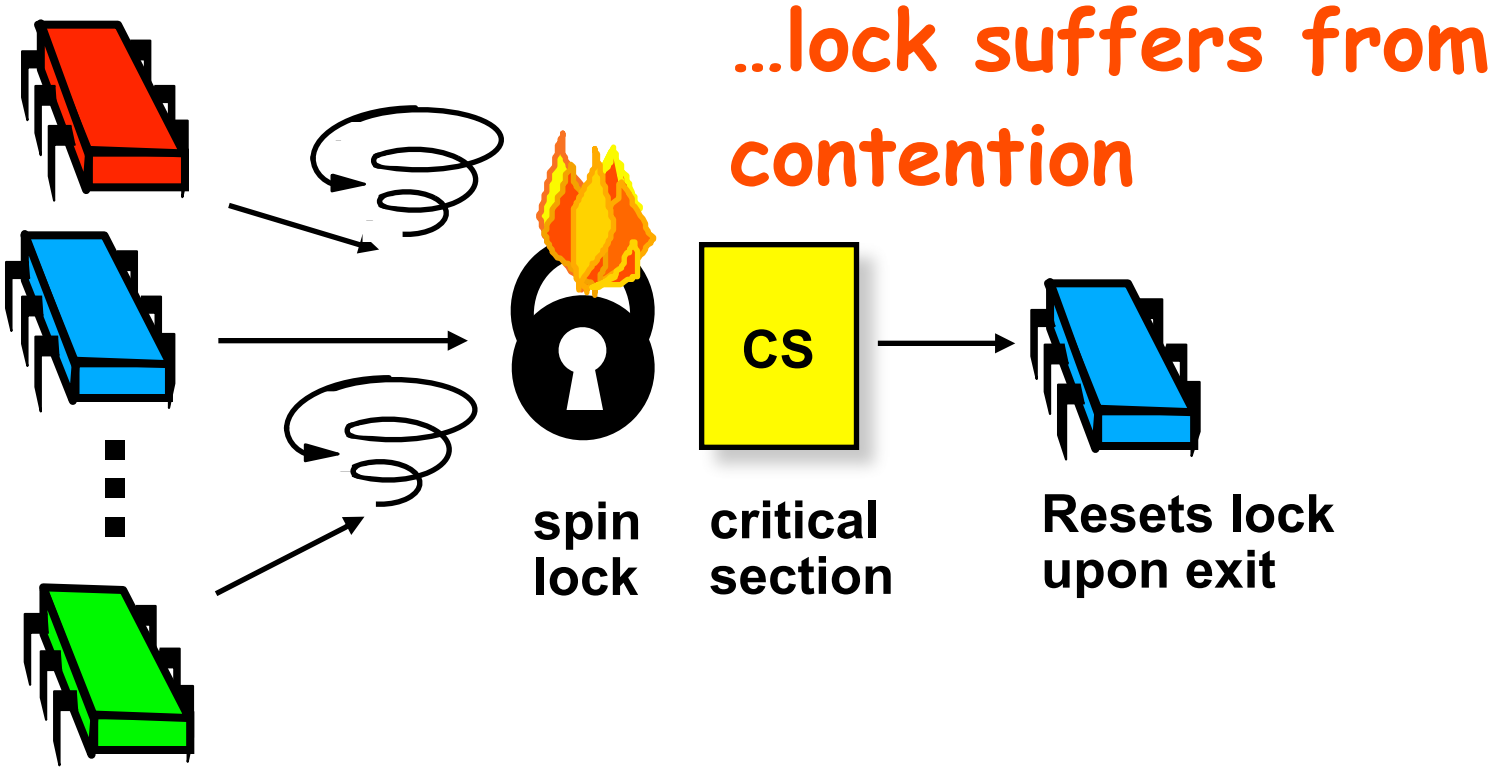
Basic Spin-Lock



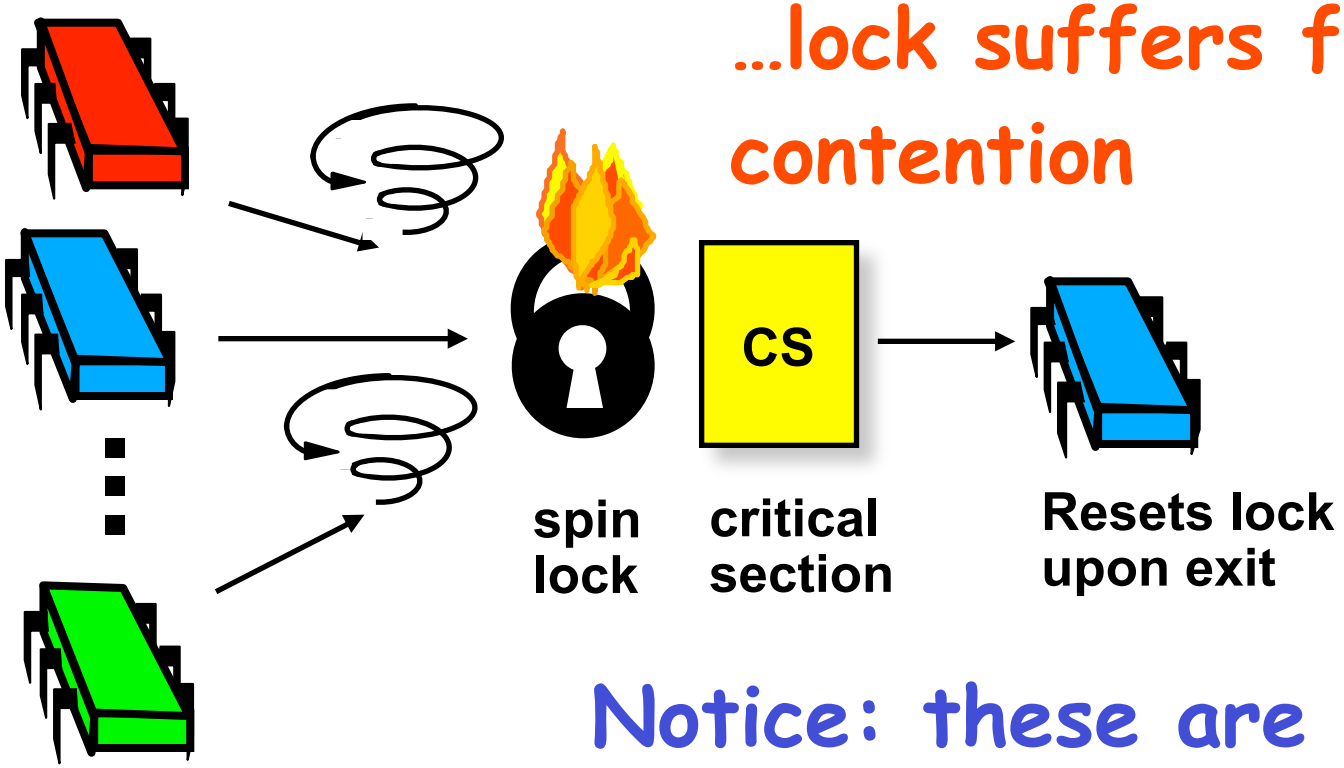
Basic Spin-Lock



Basic Spin-Lock

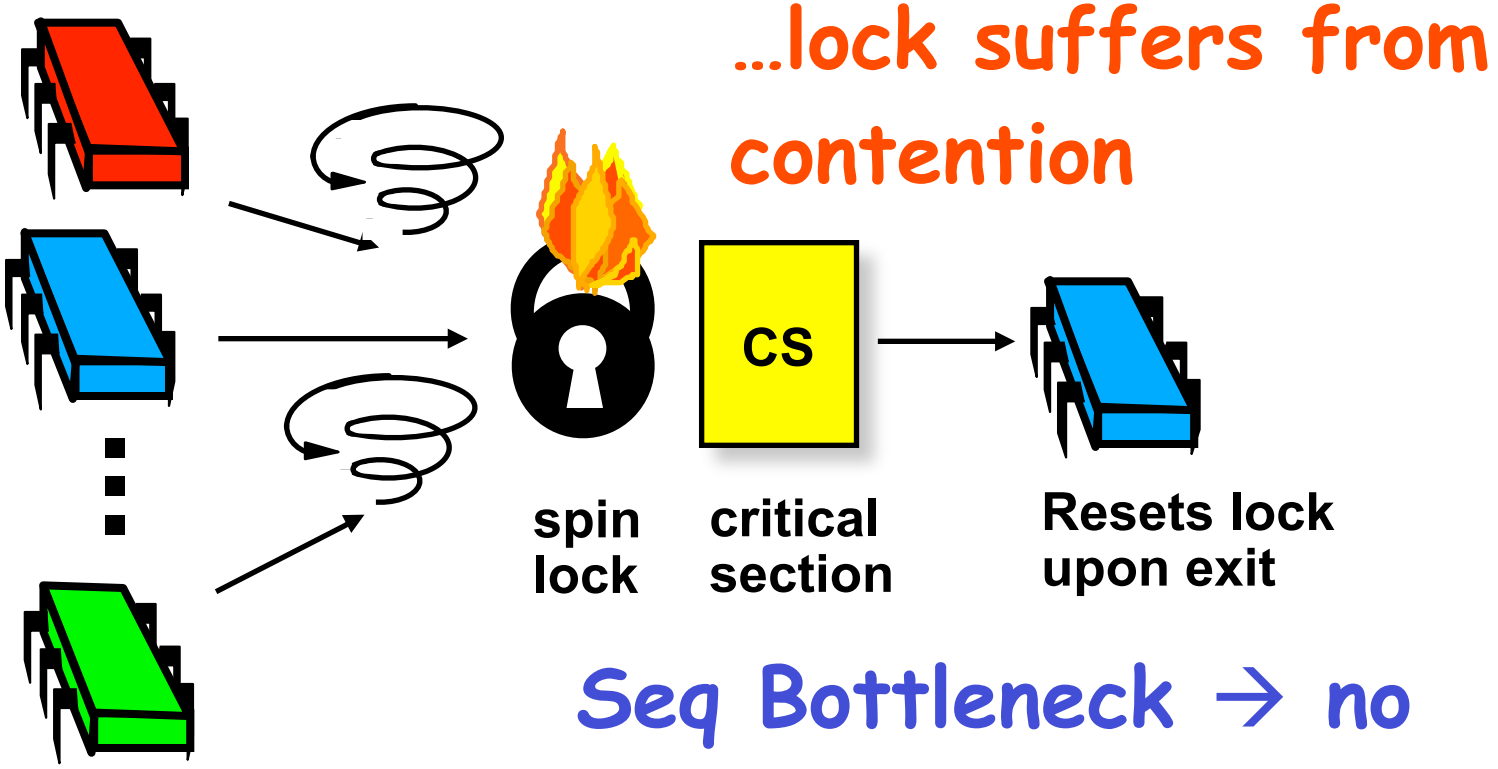


Basic Spin-Lock



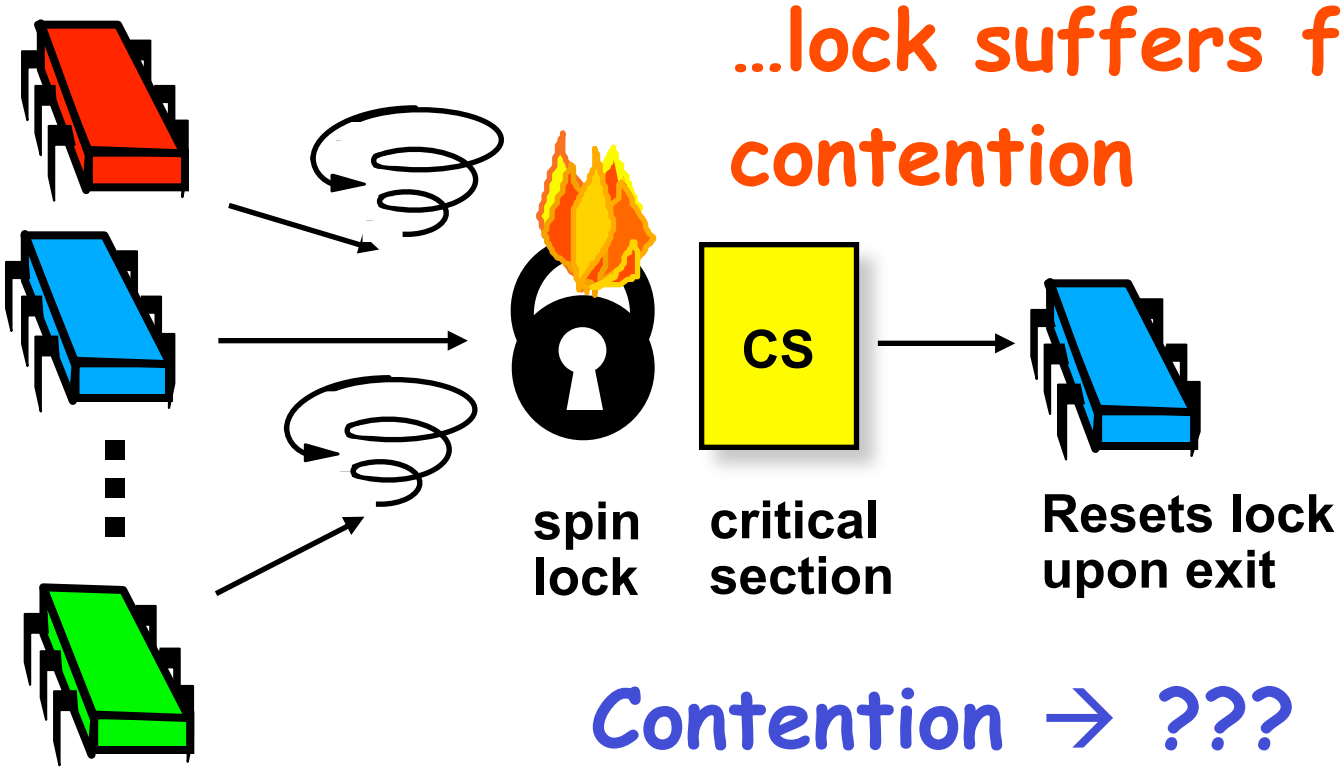
Notice: these are distinct phenomena

Basic Spin-Lock



Seq Bottleneck → no parallelism

Basic Spin-Lock



Review: Test-and-Set

- Boolean value
- Test-and-set (TAS)
  - Swap **true** with current value
  - Return value tells if prior value was **true** or **false**
- Can reset just by writing **false**
- TAS aka "getAndSet"

Review: Test-and-Set

```
public class AtomicBoolean {
 boolean value;

 public synchronized boolean
  getAndSet(boolean newValue) {
    boolean prior = value;
    value = newValue;
    return prior;
  }
}
```

Review: Test-and-Set

```
public class AtomicBoolean {
  boolean value;

  public synchronized boolean
   getAndSet(boolean newValue) {
     boolean prior = value;
     value = newValue;
     return prior;
   }
}
```

Package

java.util.concurrent.atomic

Review: Test-and-Set

```
public class AtomicBoolean {
 boolean value;

 public synchronized boolean
  getAndSet(boolean newValue) {
    boolean prior = value;
    value = newValue;
    return prior;
  }
}
```

Swap old and new values

Review: Test-and-Set

```
AtomicBoolean lock
  = new AtomicBoolean(false)

…
boolean prior = lock.getAndSet(true)
```

Review: Test-and-Set

```
AtomicBoolean lock
  = new AtomicBoolean(false)
…
boolean prior = lock.getAndSet(true)
```

Swapping in true is called "test-and-set" or TAS

Test-and-Set Locks

- Locking
  - Lock is free: value is false
  - Lock is taken: value is true
- Acquire lock by calling TAS
  - If result is false, you win
  - If result is true, you lose
- Release lock by writing false

Test-and-set Lock

```
class TASlock {
 AtomicBoolean state =
   new AtomicBoolean(false);

 void lock() {
  while (state.getAndSet(true)) {}
 }

 void unlock() {
  state.set(false);
 }}
```

Test-and-set Lock

```
class TASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (state.getAndSet(true)) {}
 }

 void unlock() {
  state.set(false);
 }}
```

Lock state is AtomicBoolean

Test-and-set Lock

```
class TASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (state.getAndSet(true)) {}
 }

 void unlock() {
  state.set(false);
 }}
```

Keep trying until lock acquired

Test-and-set Lock

```
class TA
 AtomicB
  new At

 void lo
  while (state.getAndSet(true)) {}
 }

 void unlock() {
  state.set(false);
 }}
```

Release lock by resetting state to false

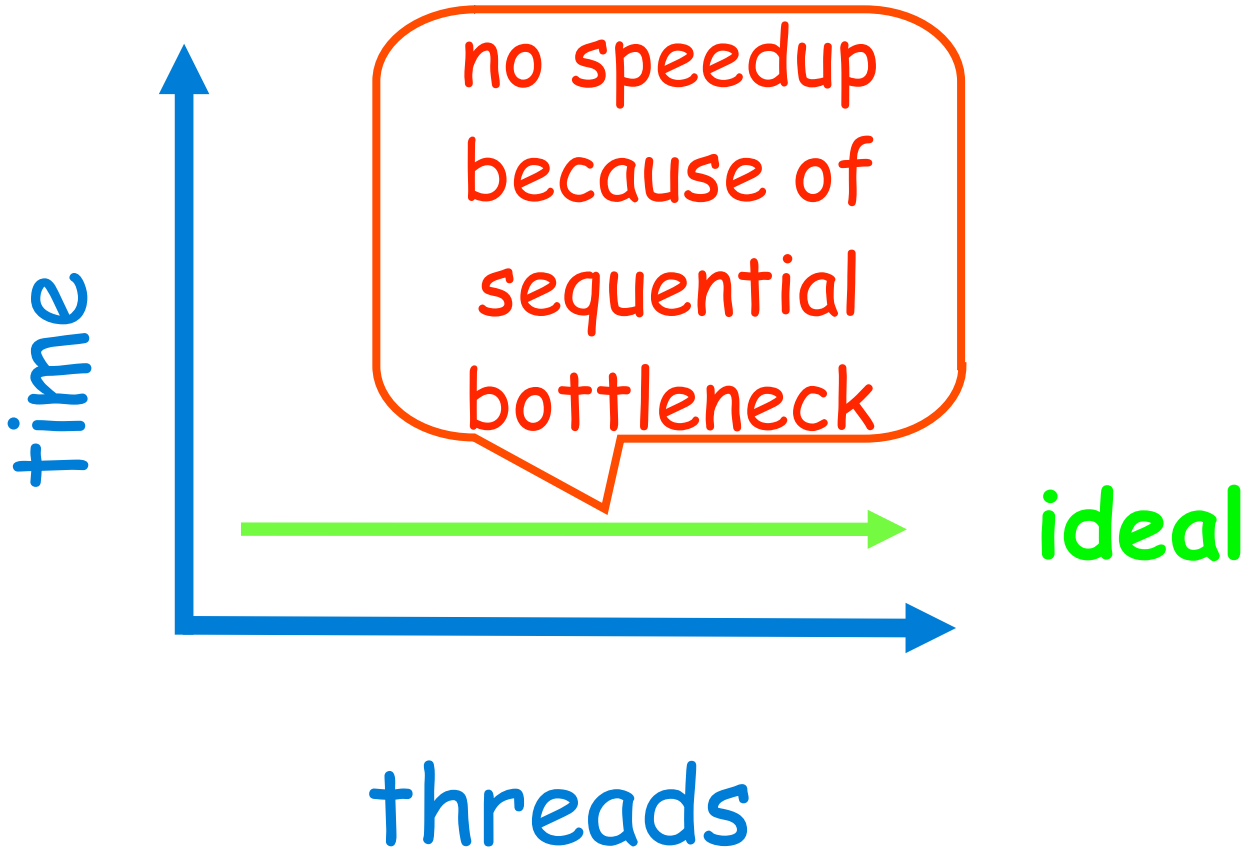
Space Complexity

- TAS spin-lock has small "footprint"
- N thread spin-lock uses $O(1)$ space
- As opposed to $O(n)$ Peterson/Bakery
- How did we overcome the $\Omega(n)$ lower bound?
- We used a RMW operation…

Performance

- Experiment
  - n threads
  - Increment shared counter 1 million times
- How long should it take?
- How long does it take?

Graph


no speedup
because of
sequential
bottleneck
time
ideal
threads

Mystery #1