

Locking

Part 2, Chapter 9

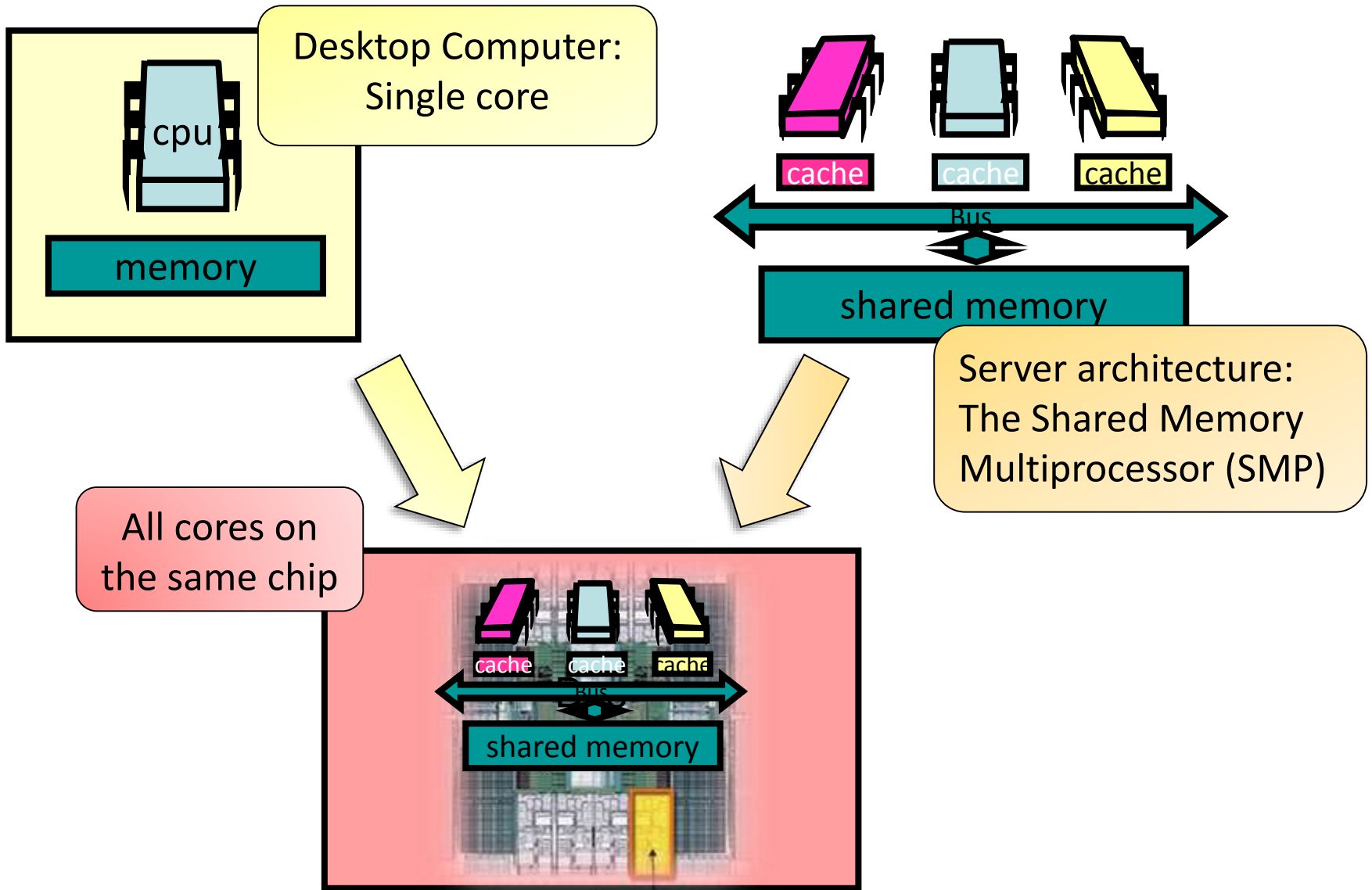


Roger Wattenhofer

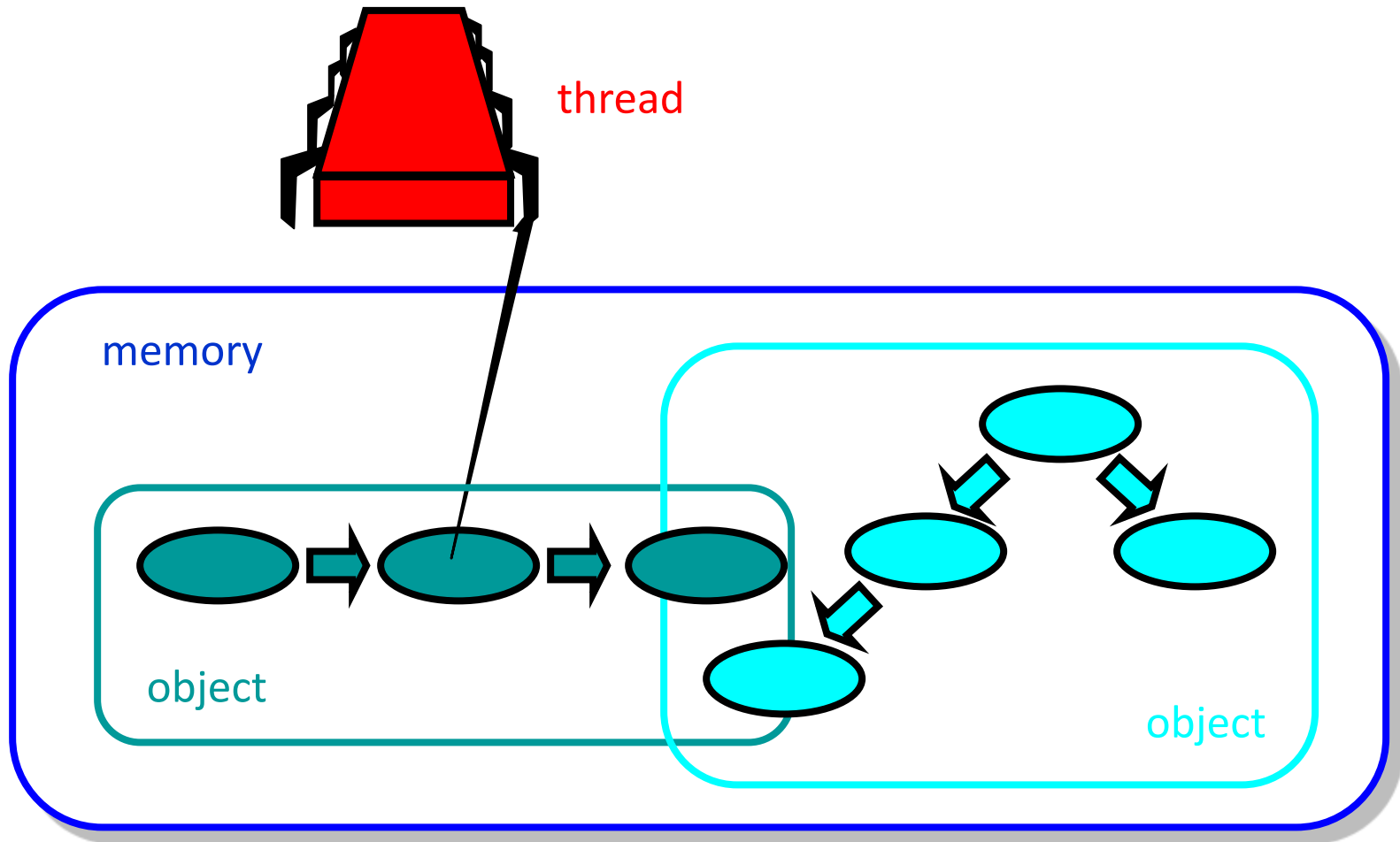
Overview

- Introduction
- Spin Locks
 - Test-and-Set & Test-and-Test-and-Set
 - Backoff lock
 - Queue locks

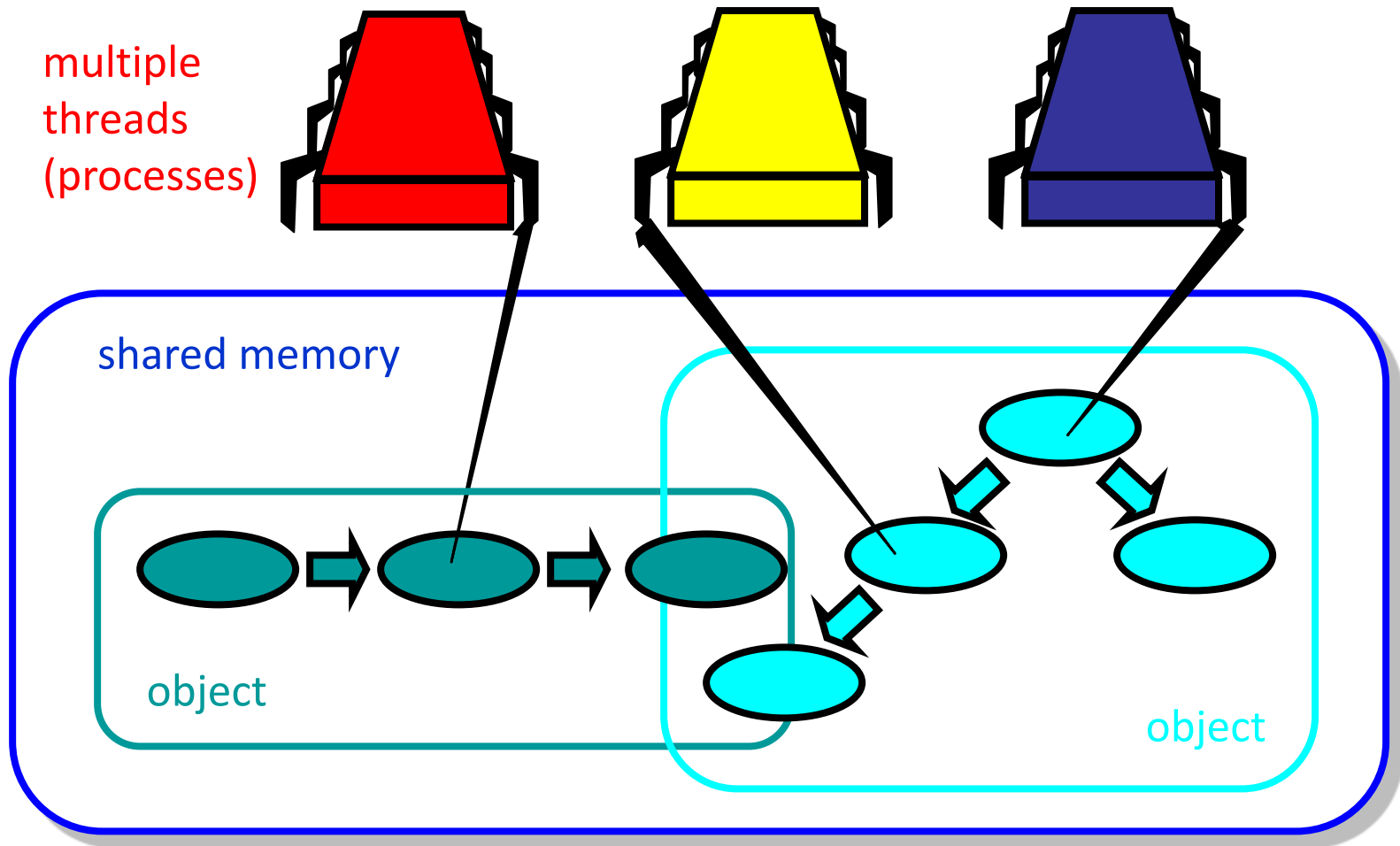
Introduction: From Single-Core to Multicore Computers



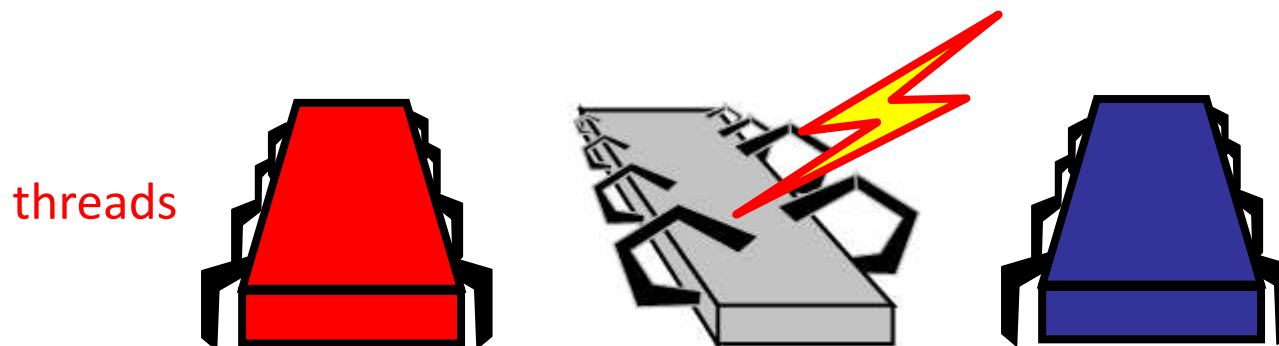
Sequential Computation



Concurrent Computation



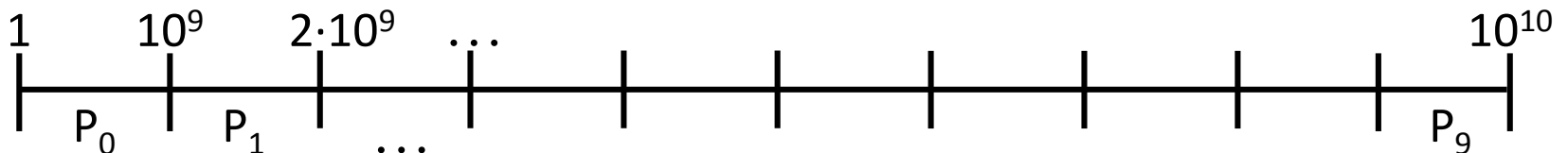
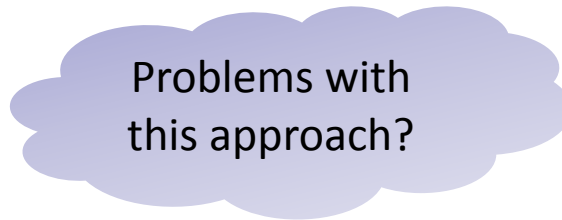
Fault Tolerance & Asynchrony



- Why fault-tolerance?
 - Even if processes do not die, there are “near-death experiences”
- Sudden unpredictable delays:
 - Cache misses (short)
 - Page faults (long)
 - Scheduling quantum used up (really long)

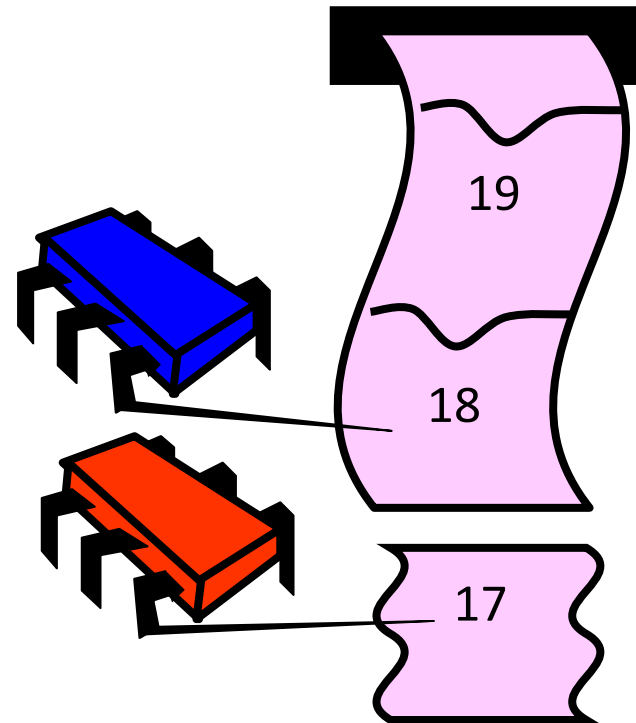
Example: Parallel Primality Testing

- Challenge
 - Print all primes from 1 to 10^{10}
- Given
 - Ten-core multiprocessor
 - One thread per processor
- Goal
 - Get ten-fold speedup (or close)
- Naïve Approach
 - Split the work evenly
 - Each thread tests range of 10^9



Issues

- Higher ranges have fewer primes
- Yet larger numbers are harder to test
- Thread workloads
 - Uneven
 - Hard to predict
- Need **dynamic** load balancing
- Better approach
 - Shared counter!
 - Each thread takes a number



Procedure Executed at each Thread

```
Counter counter = new Counter(1);
```

```
void primePrint() {  
    long j = 0;  
    while(j < 1010) {  
        j = counter.getAndIncrement();  
        if(isPrime(j))  
            print(j);  
    }  
}
```

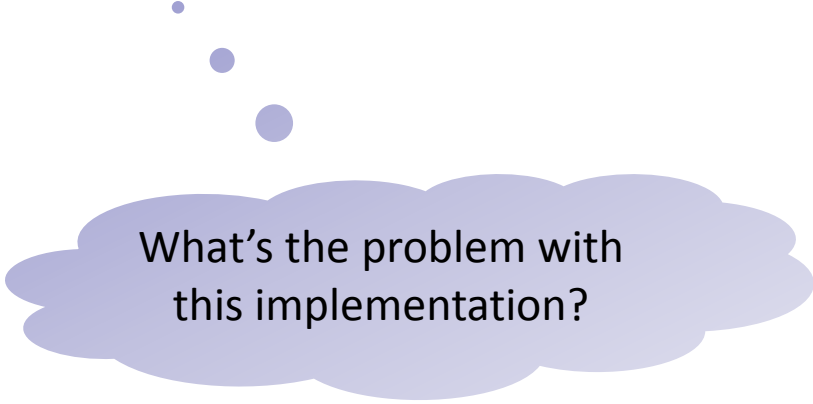
Shared counter object

```
j = counter.getAndIncrement();  
if(isPrime(j))  
    print(j);
```

**Increment counter & test
if return value is prime**

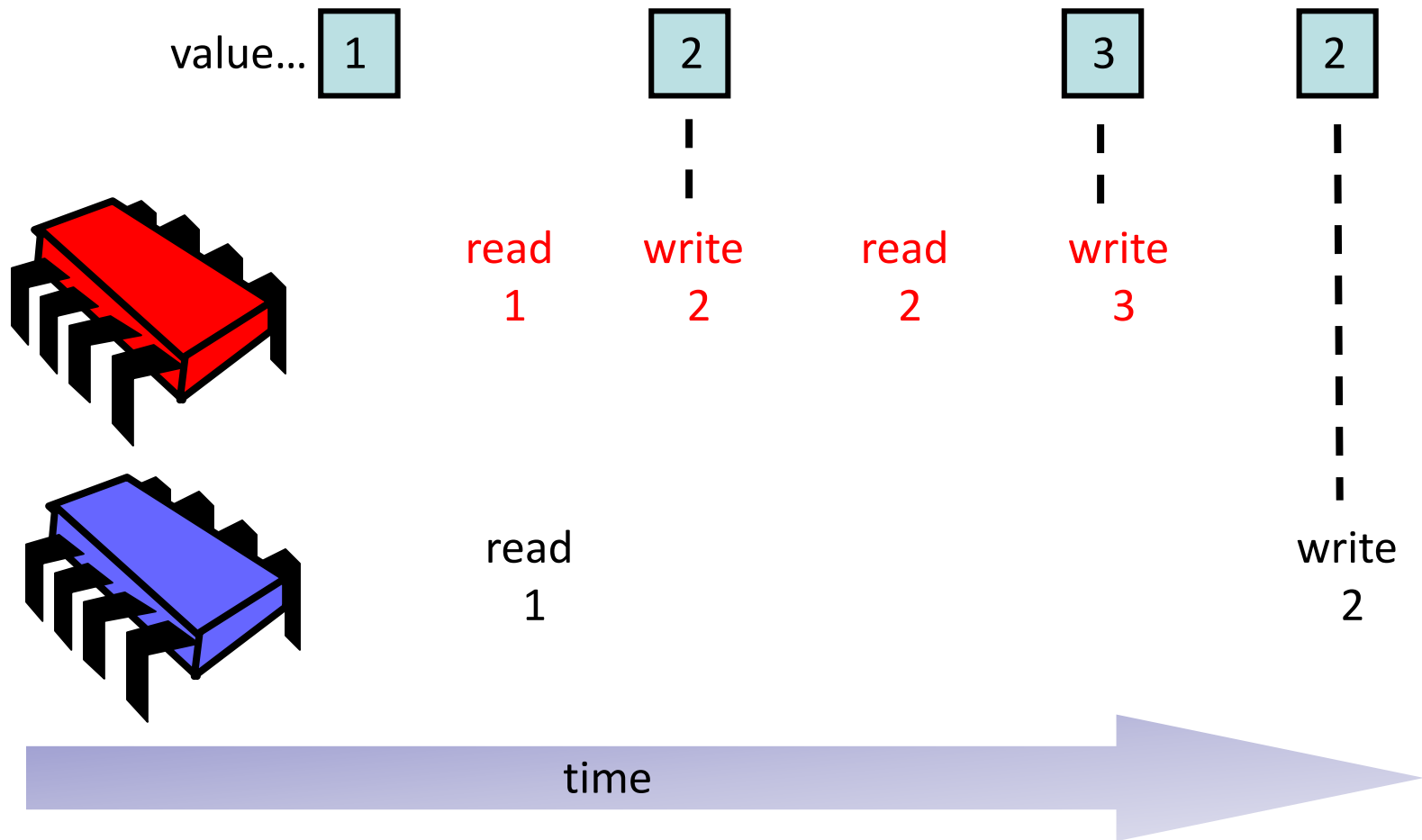
Counter Implementation

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        return value++;  
    }  
}
```



What's the problem with this implementation?

Problem



Counter Implementation

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        temp = value;  
        value = temp + 1;  
        return temp;  
    }  
}
```

**These steps must
be atomic!**

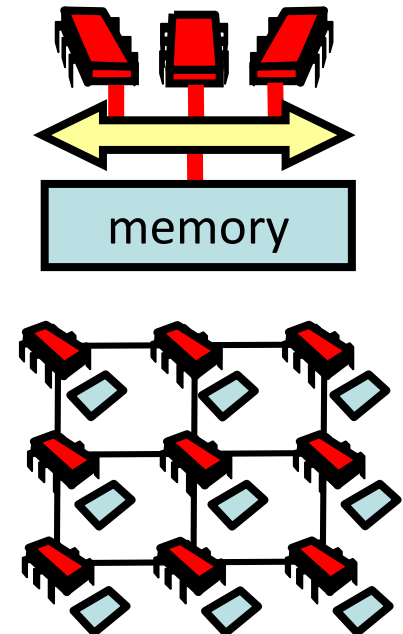
Recall: We can use **Read-Modify-Write (RMW)** instructions!

We have to guarantee
mutual exclusion

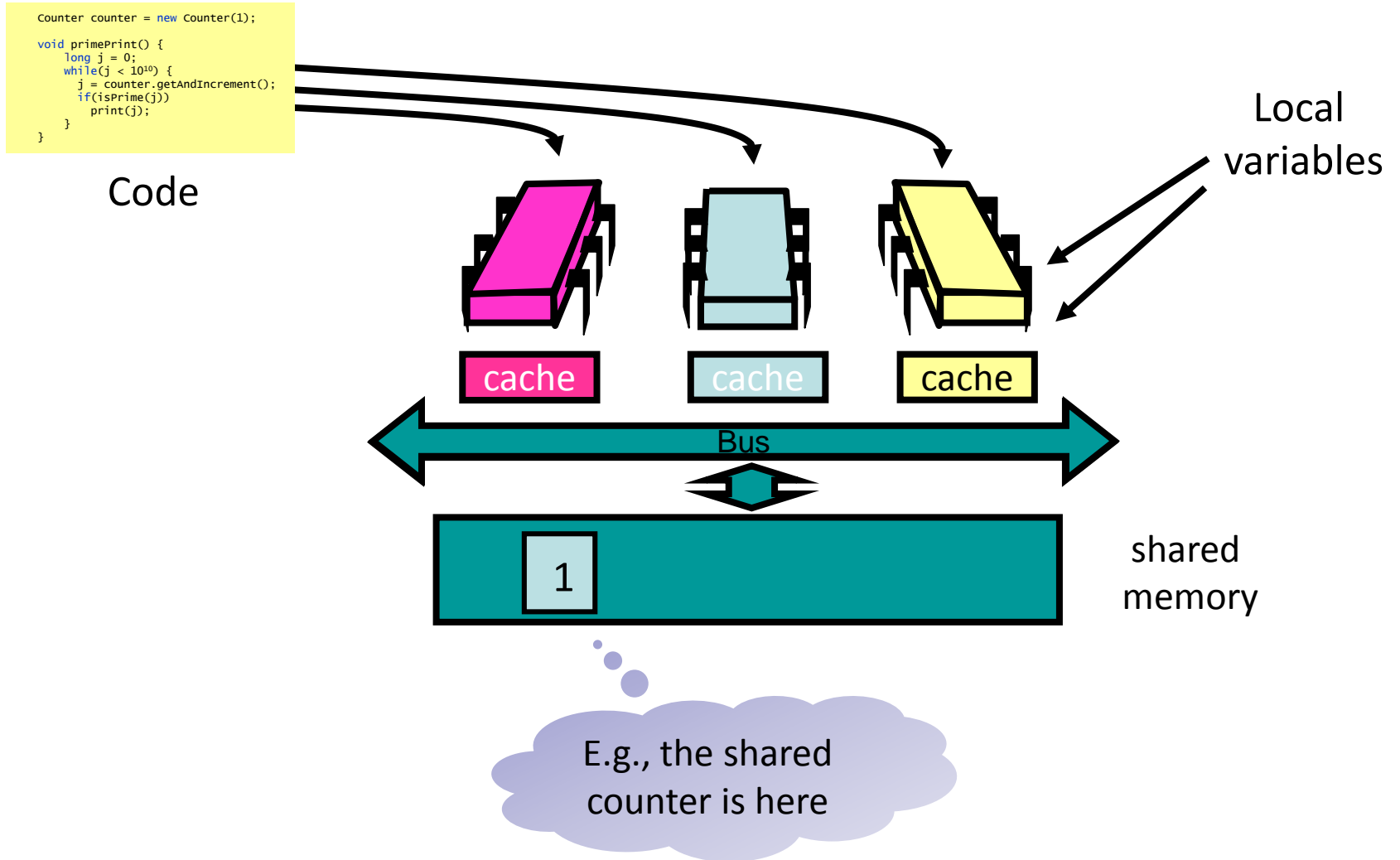
Model

- The model in this part is slightly more complicated
 - However, we still focus on principles
- What remains the **same**?
 - Multiple instruction multiple data (**MIMD**) architecture
 - Each thread/process has its own code and local variables
- What is **new**?
 - There is a **shared memory** that all threads can access
 - Typically, communication runs over a **shared bus** (alternatively, there may be **several channels**)
 - Communication contention
 - Communication latency
 - Each thread has a local **cache**

I.e., multiprocessors



Model: Where Things Reside



Revisiting Mutual Exclusion

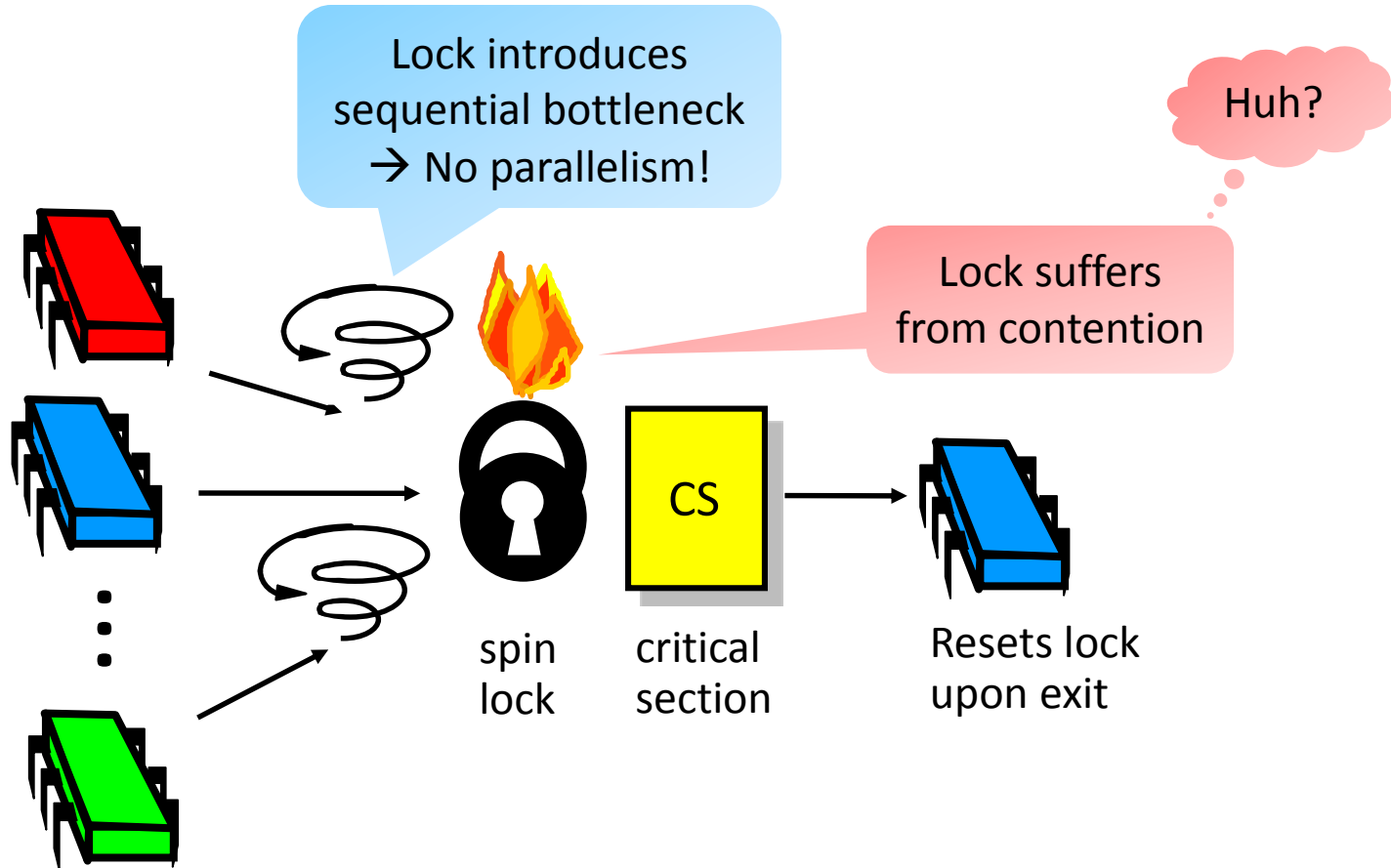
- We need **mutual exclusion** for our counter
- We are now going to study mutual exclusion from a different angle
 - Focus on performance, not just correctness and progress
- We will begin to understand how performance depends on our software properly utilizing the multiprocessor machine's hardware, and get to know a collection of **locking algorithms**!



- 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



Reminder: Test&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**
- Also known as “getAndSet”

Reminder: Test&Set

```
public class AtomicBoolean {  
    private boolean value;
```

java.util.concurrent.atomic

```
    public synchronized boolean getAndSet() {
```

```
        boolean prior = this.value;  
        this.value = true;  
        return prior;
```

```
    }
```

```
}
```

**Get current value and set
value to true**

Test&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&Set Lock

```
public class TASLock implements Lock {
```

```
    AtomicBoolean state = new AtomicBoolean(false);
```

```
    public void lock() {
```

```
        while (state.getAndSet()) {}
```

```
    }
```

```
    public void unlock() {
```

```
        state.set(false);
```

```
    }
```

```
}
```

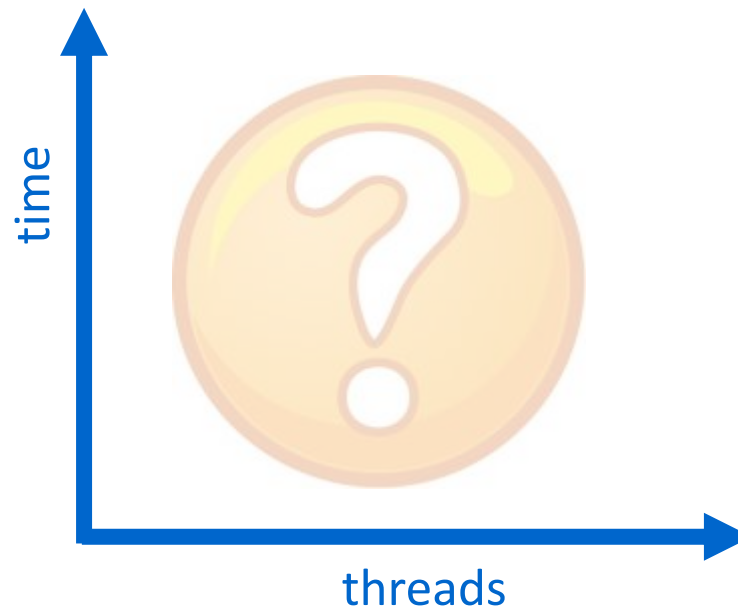
Lock state is AtomicBoolean

**Keep trying until
lock acquired**

Release lock by resetting state to false

Performance

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



Test&Test&Set Locks

- How can we improve TAS?
- A crazy idea: Test before you test and set!

- Lurking stage
 - Wait until lock “looks” free
 - Spin while read returns true (i.e., the lock is taken)
- Pouncing state
 - As soon as lock “looks” available
 - Read returns false (i.e., the lock is free)
 - Call TAS to acquire the lock
 - If TAS loses, go back to lurking

Test&Test&Set Lock

```
public class TTASLock implements Lock {
    AtomicBoolean state = new AtomicBoolean(false);

    public void lock() {
        while (true) {
            while(state.get()) {}
            if(!state.getAndSet())
                return;
        }
    }

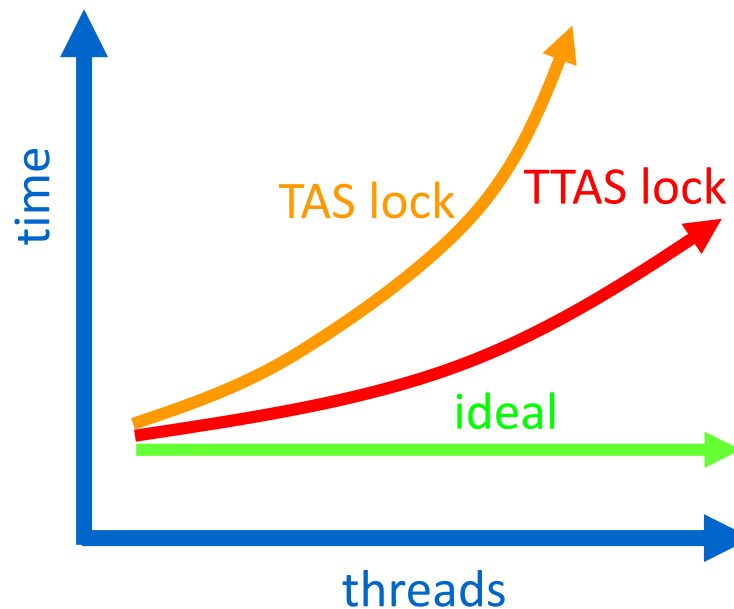
    public void unlock() {
        state.set(false);
    }
}
```

Wait until lock looks free

Then try to acquire it

Performance

- Both TAS and TTAS do the same thing (in our old model)
- So, we would expect basically the same results



- Why is TTAS so much better than TAS? Why are both far from ideal?

Opinion

- TAS & TTAS locks
 - are provably the same (in theory)
 - except they aren't (in reality)
- Obviously, it must have something to do with the model...
- Let's take a closer look at our new model and try to find a reasonable explanation!

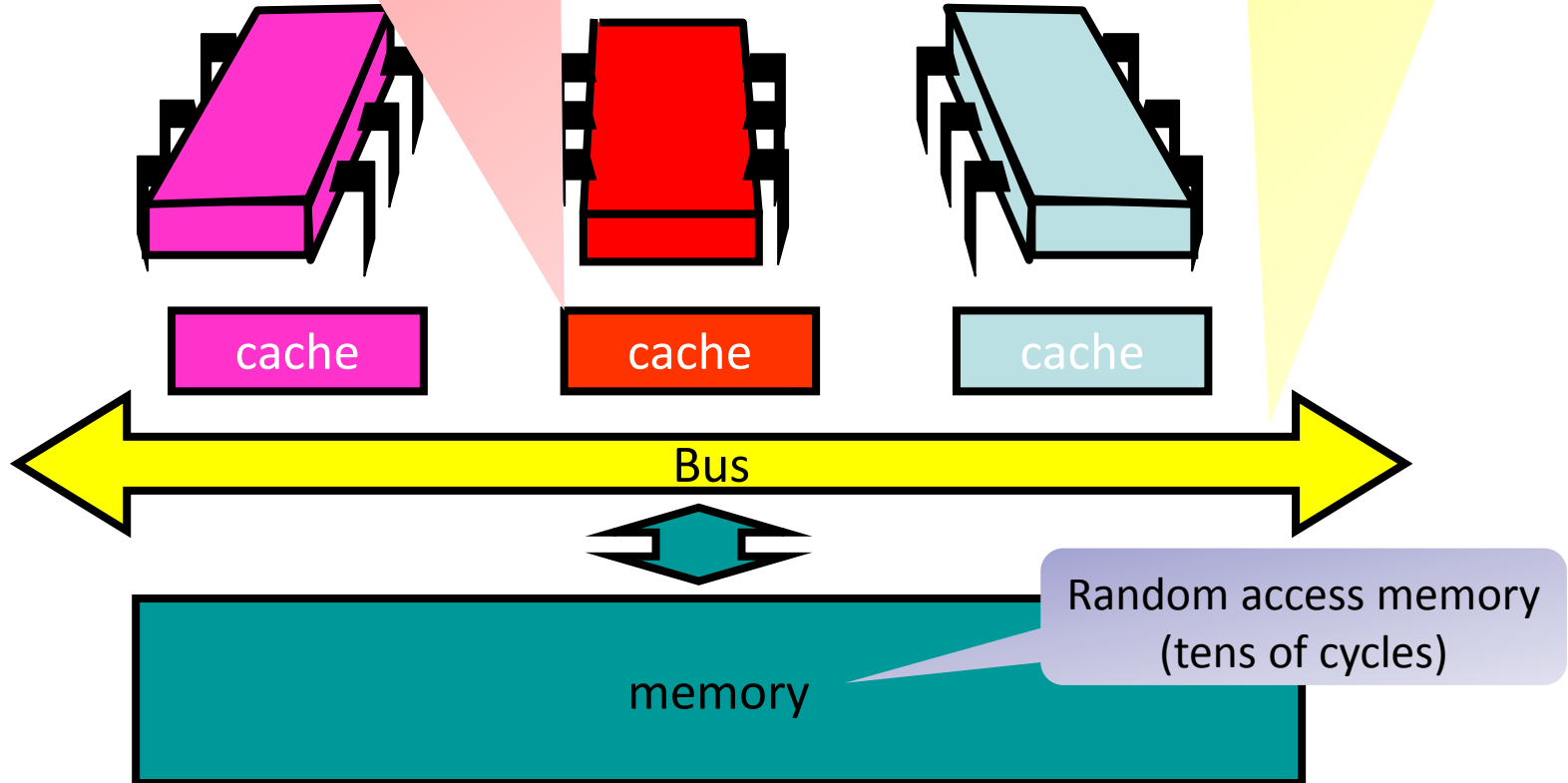
Bus-Based Architectures

Per-processor caches

- Small
- Fast: 1 or 2 cycles
- Address and state information

Shared bus

- Broadcast medium
- One broadcaster at a time
- Processors (and memory) “snoop”

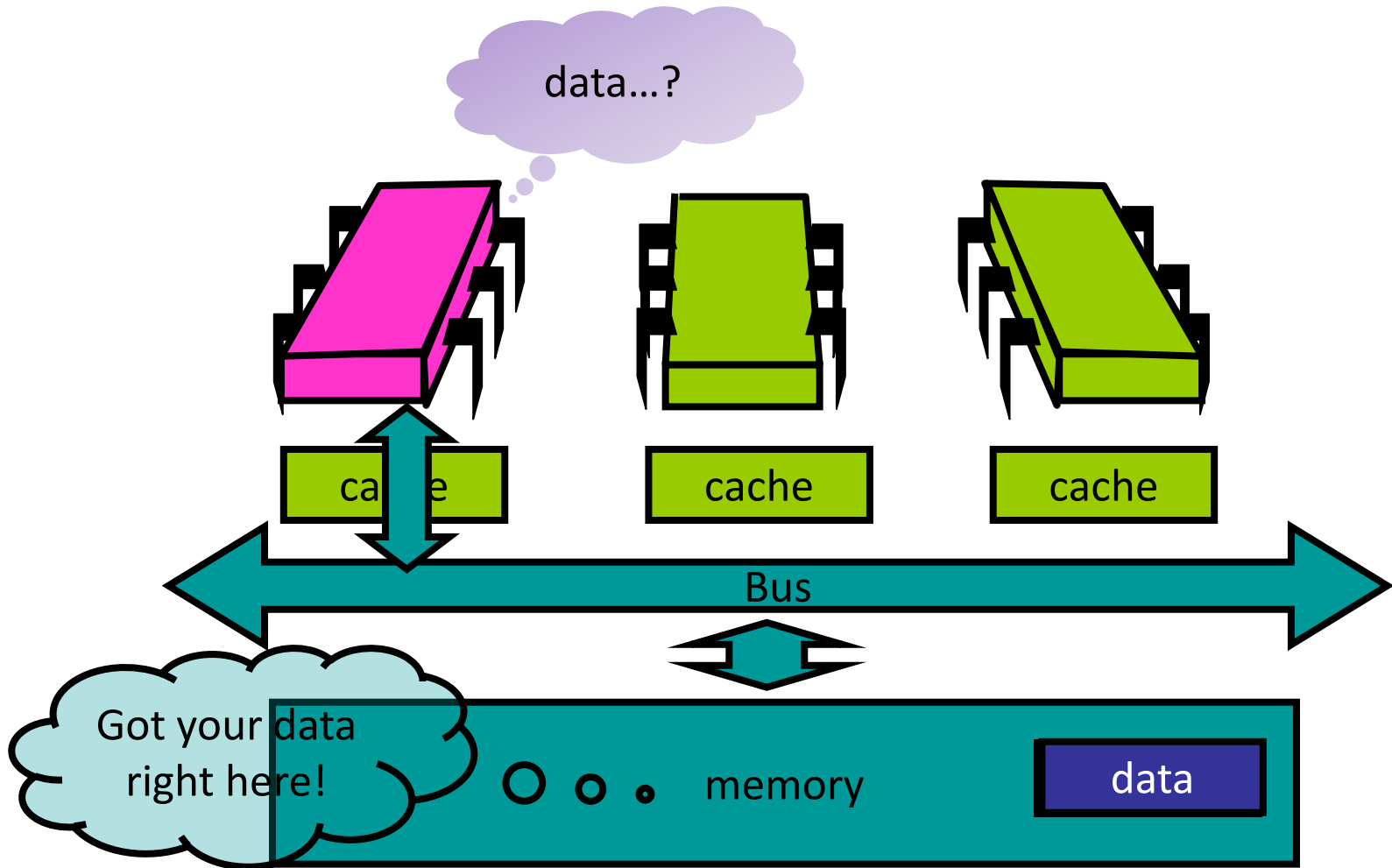


Jargon Watch

- Load request
 - When a thread wants to access data, it issues a load request
- Cache hit
 - The thread found the data in its own cache
- Cache miss
 - The data is not found in the cache
 - The thread has to get the data from memory

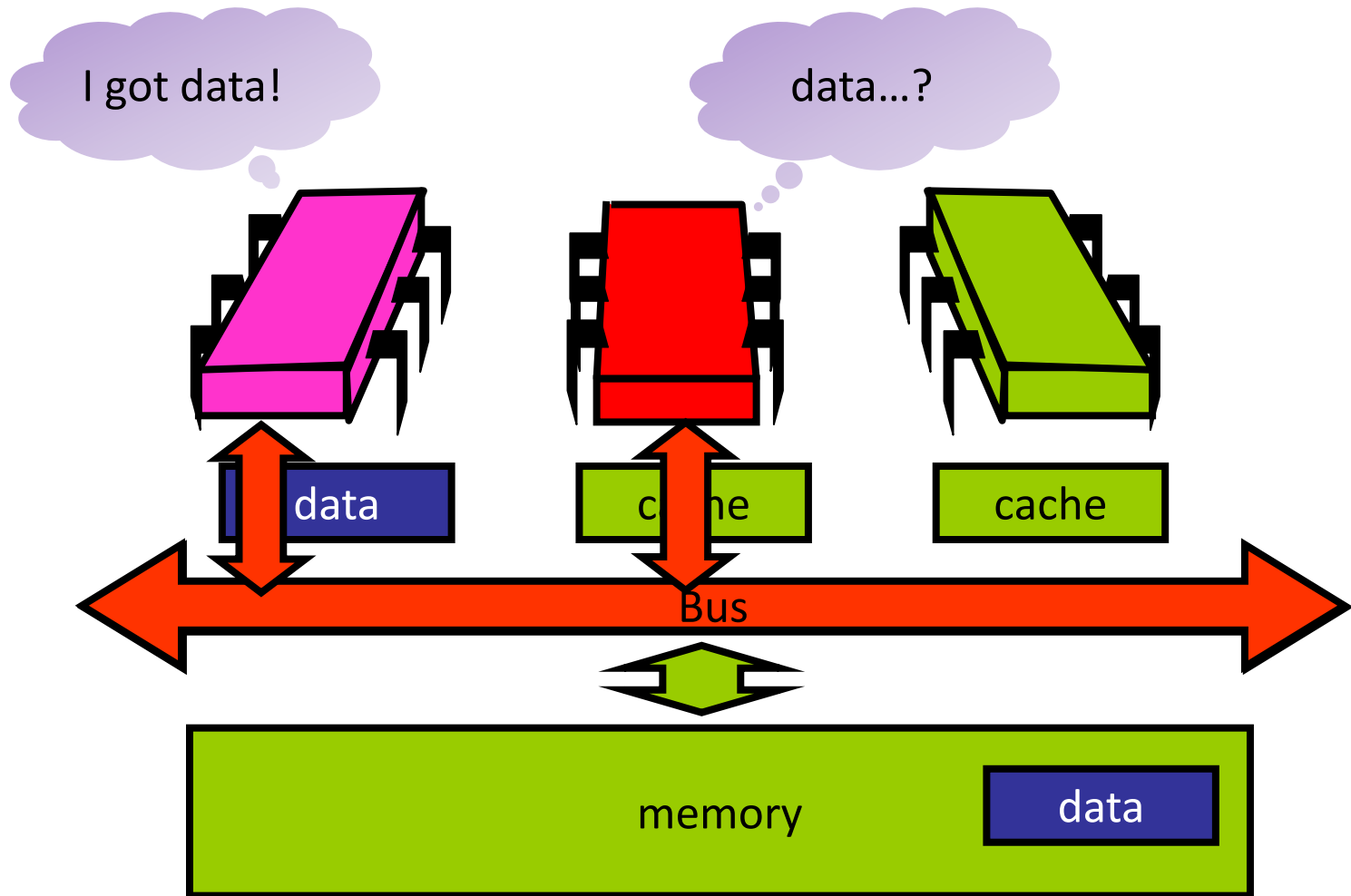
Load Request

- Thread issues load request and memory responds



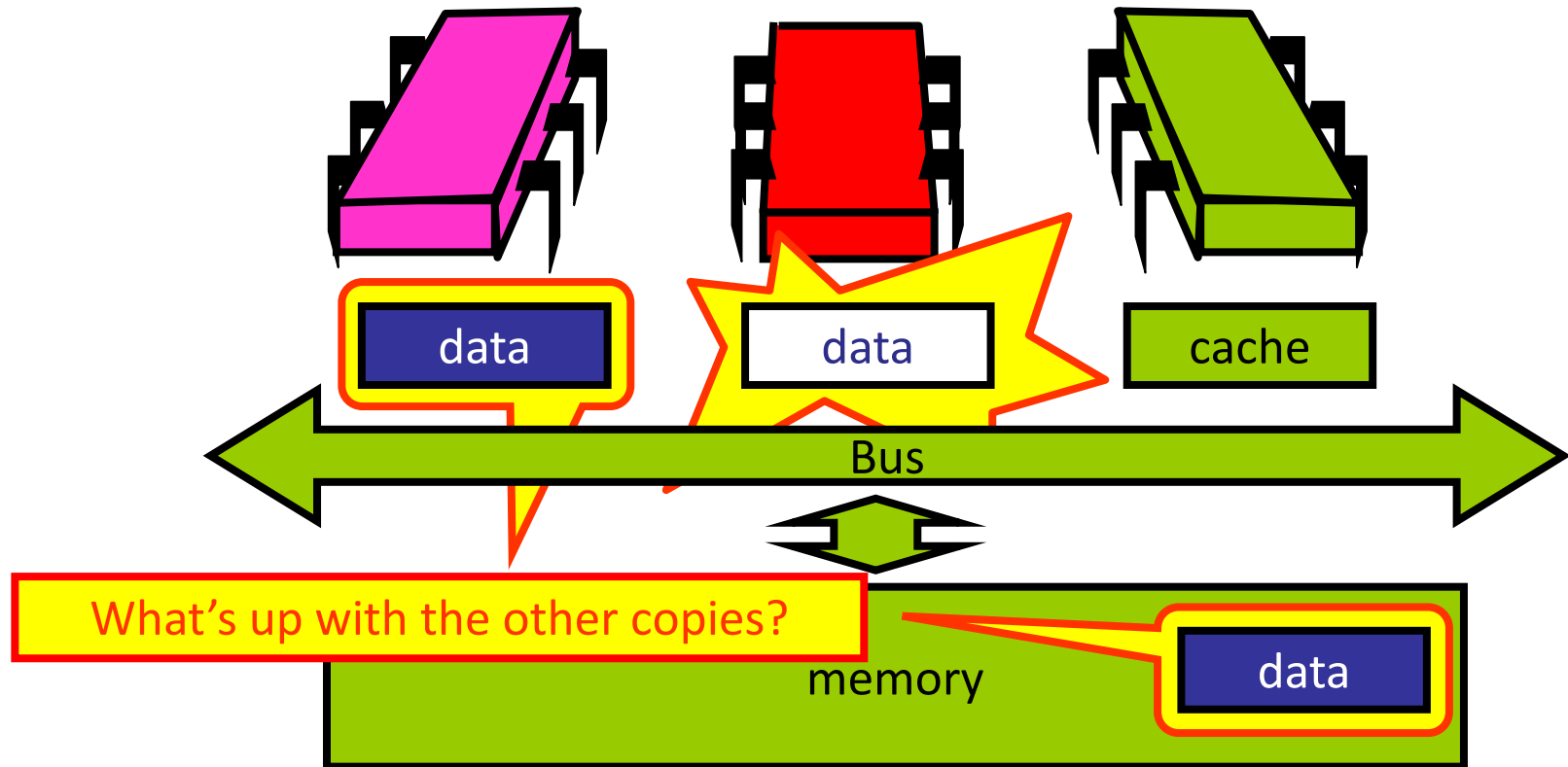
Another Load Request

- Another thread wants to access the same data. Get a copy from the cache!



Modify Cached Data

- Both threads now have the data in their cache
- What happens if the red thread now **modifies** the data...?



Cache Coherence

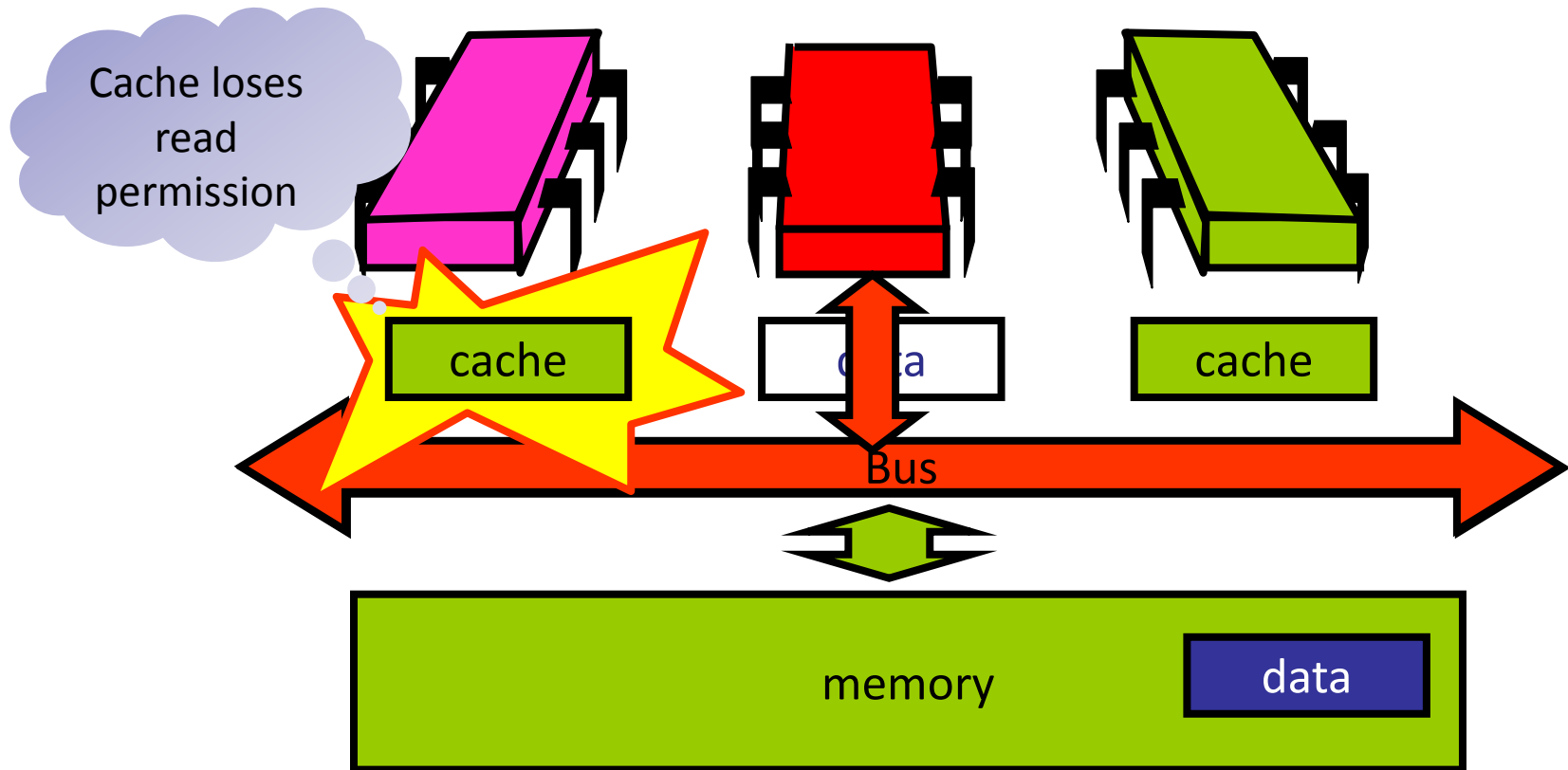
- We have lots of copies of data
 - Original copy in memory
 - Cached copies at processors
- Some processor modifies its own copy
 - What do we do with the others?
 - How to avoid confusion?

Write-Back Caches

- Accumulate changes in cache
- Write back when needed
 - Need the cache for something else
 - Another processor wants it
- On first modification
 - Invalidate other entries
 - Requires non-trivial protocol ...
- Cache entry has three states:
 - Invalid: contains raw bits
 - Valid: I can read but I can't write
 - Dirty: Data has been modified
 - Intercept other load requests
 - Write back to memory before reusing cache

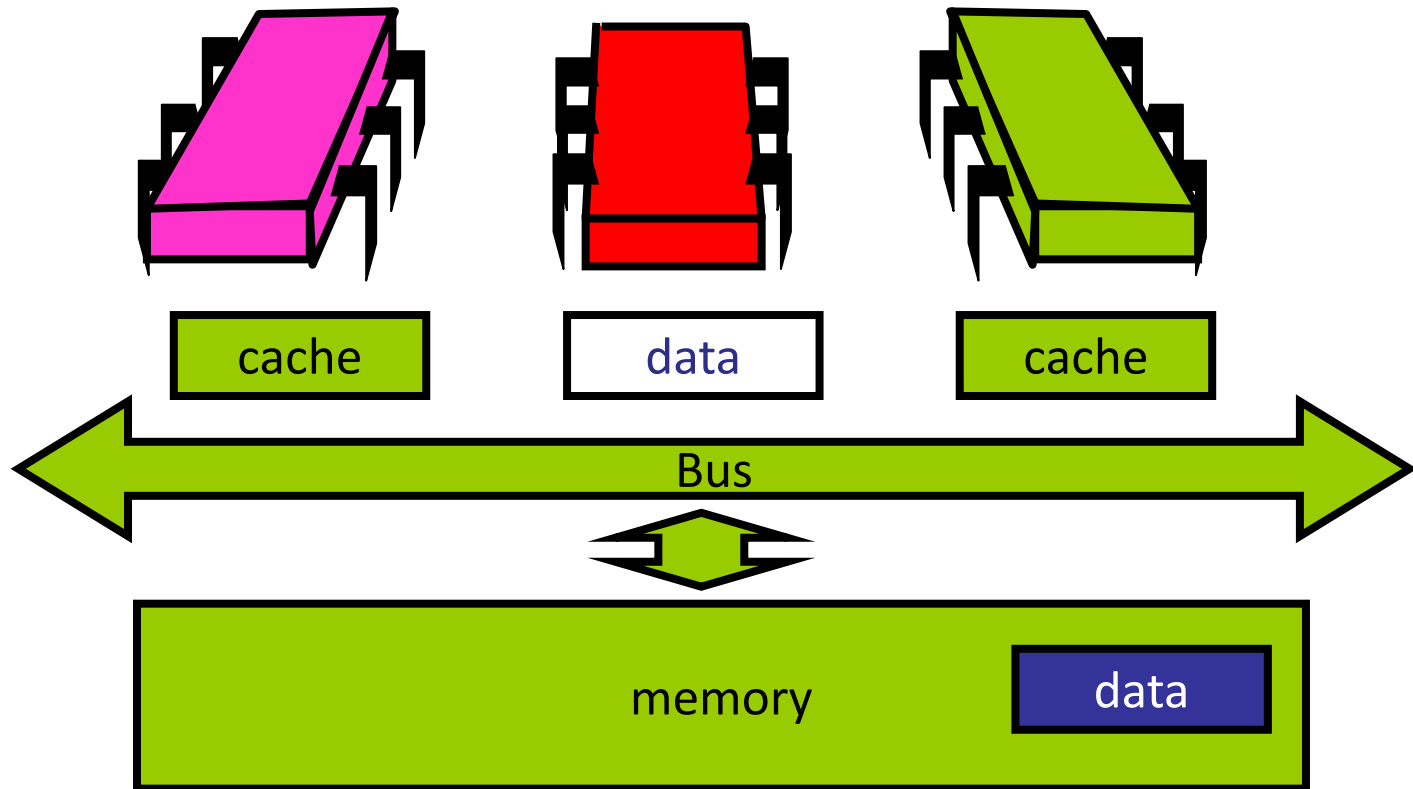
Invalidate

- Let's rewind back to the moment when the red processor updates its cached data
- It broadcasts an **invalidation** message → Other processor invalidates its cache!



Invalidate

- Memory provides data only if not present in any cache, so there is no need to change it now (this is an expensive operation!)
- Reading is not a problem → The threads get the data from the red process



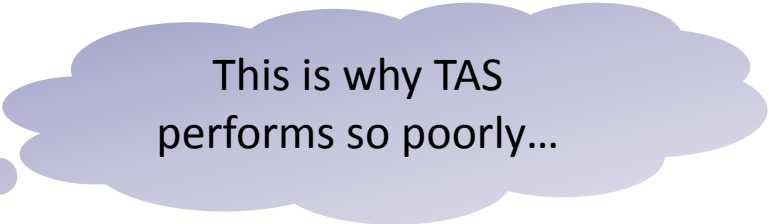
Mutual Exclusion

- What do we want to optimize?
 1. Minimize the bus bandwidth that the spinning threads use
 2. Minimize the lock acquire/release latency
 3. Minimize the latency to acquire the lock if the lock is idle


TAS vs. TTAS

- TAS invalidates cache lines
- Spinners
 - Always go to bus
- Thread wants to release lock
 - delayed behind spinners!!!

- TTAS waits until lock “looks” free
 - Spin on local cache
 - No bus use while lock busy
- Problem: when lock is released
 - Invalidation storm ...



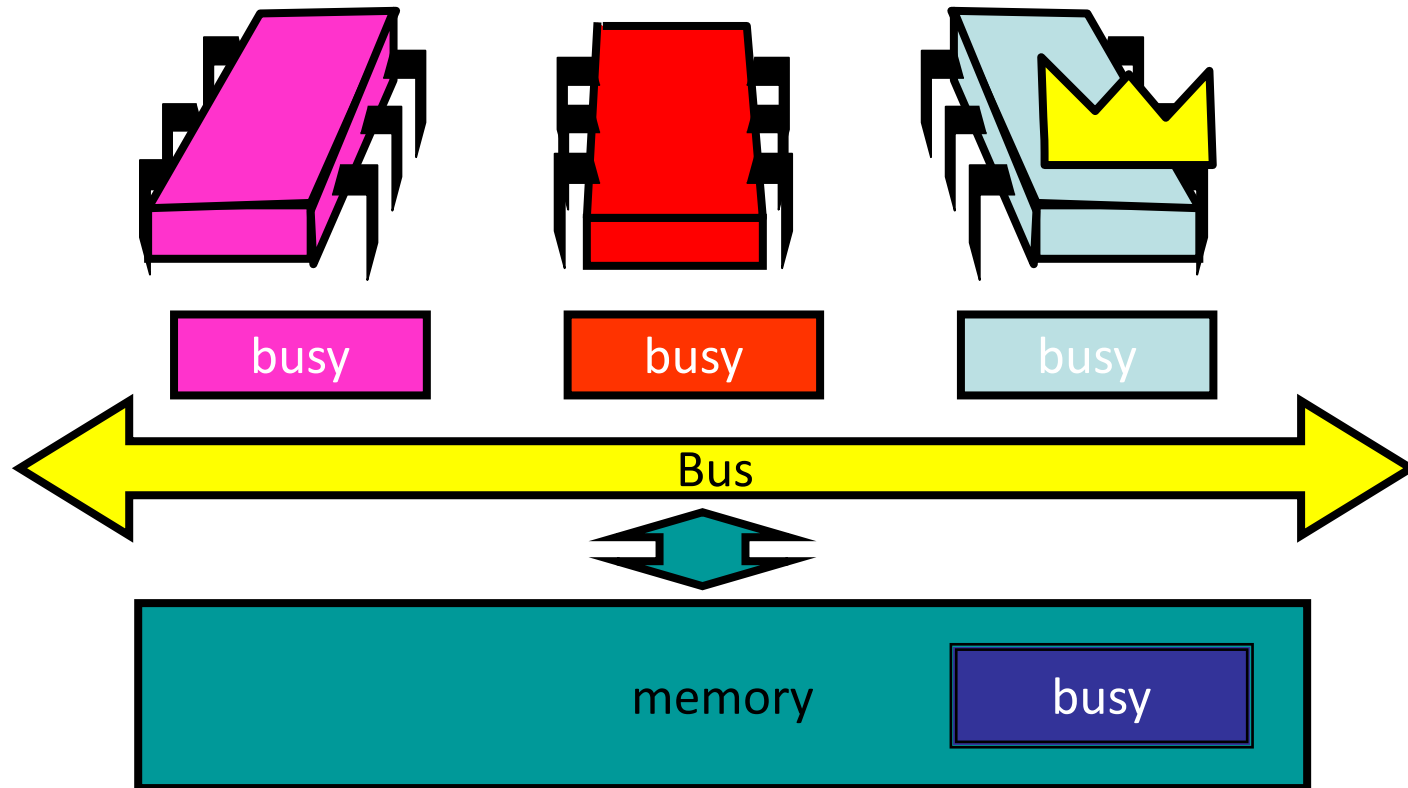
This is why TAS performs so poorly...



Huh?

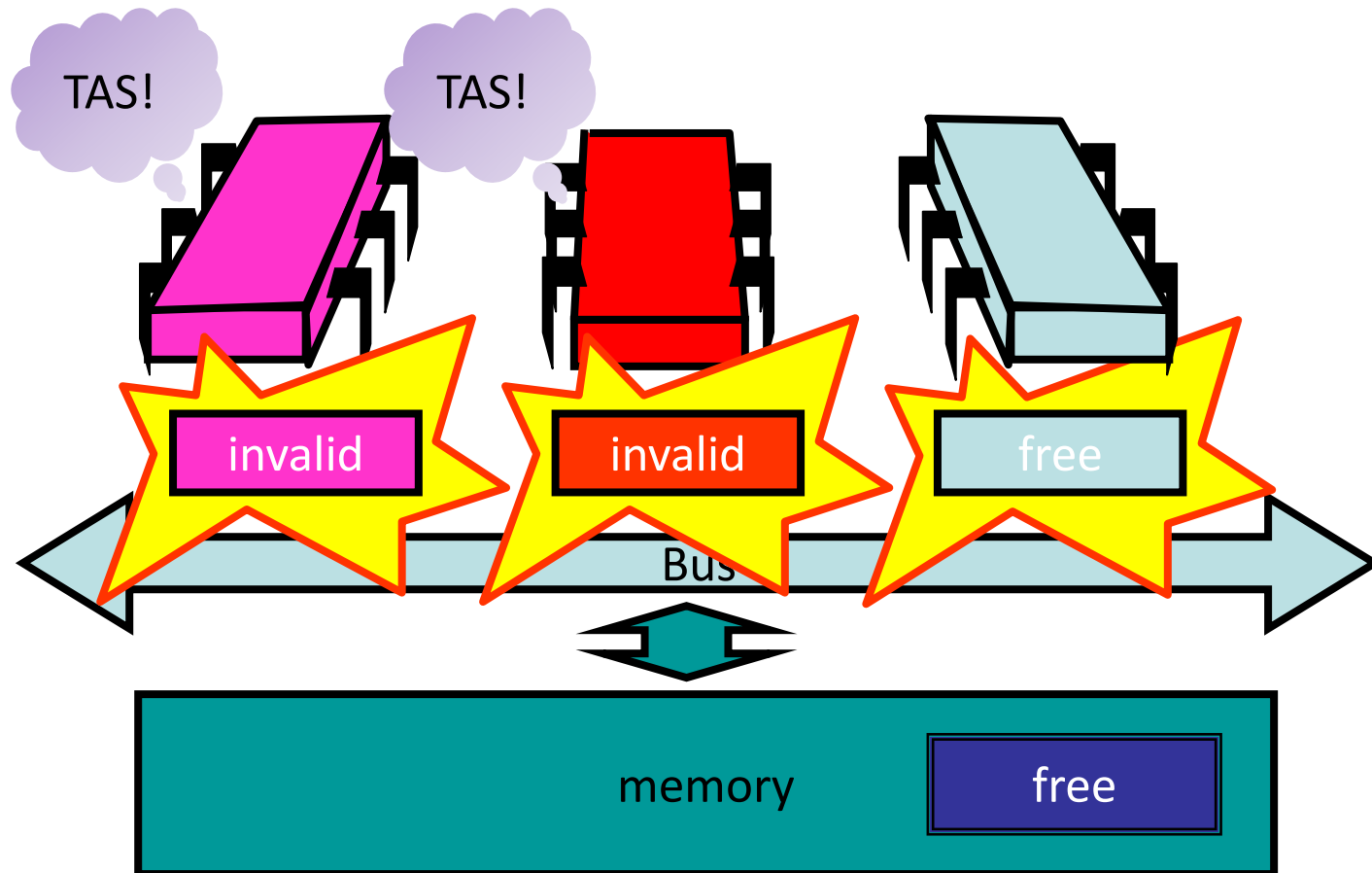
Local Spinning while Lock is Busy

- While the lock is held, all contenders spin in their caches, rereading cached data without causing any bus traffic



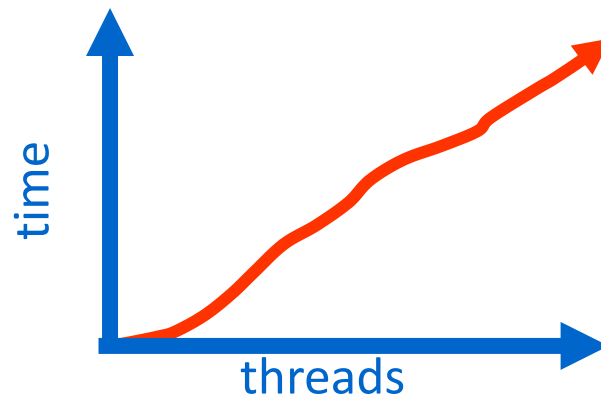
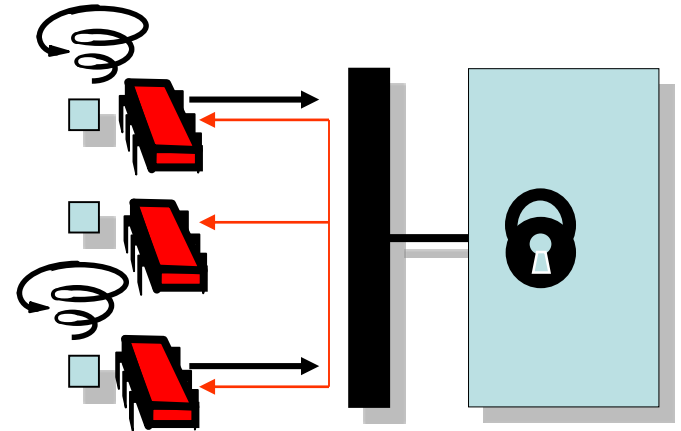
On Release

- The lock is released. All spinners take a cache miss and call Test&Set!



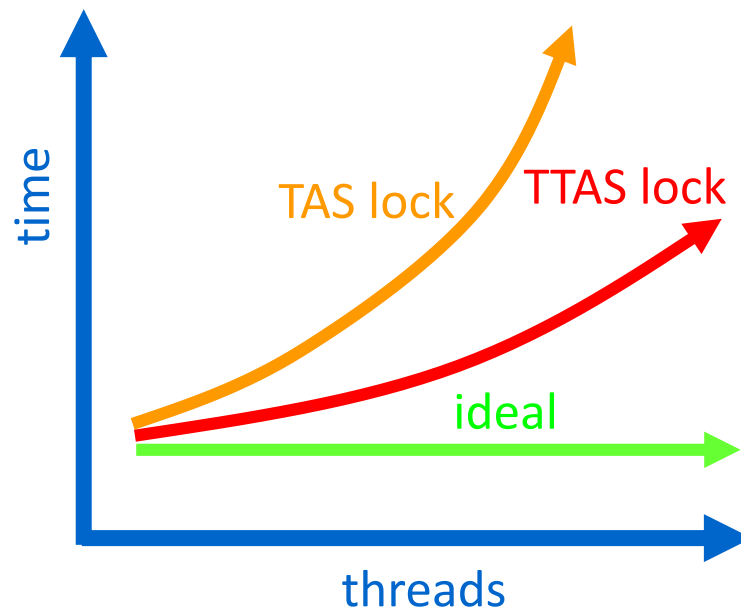
Time to Quiescence

- Every process experiences a cache miss
 - All state.get() satisfied sequentially
- Every process does TAS
 - Caches of other processes are invalidated
- Eventual quiescence (“silence”) after acquiring the lock
- The time to quiescence increases **linearly** with the number of processors for a bus architecture!



Mystery Explained

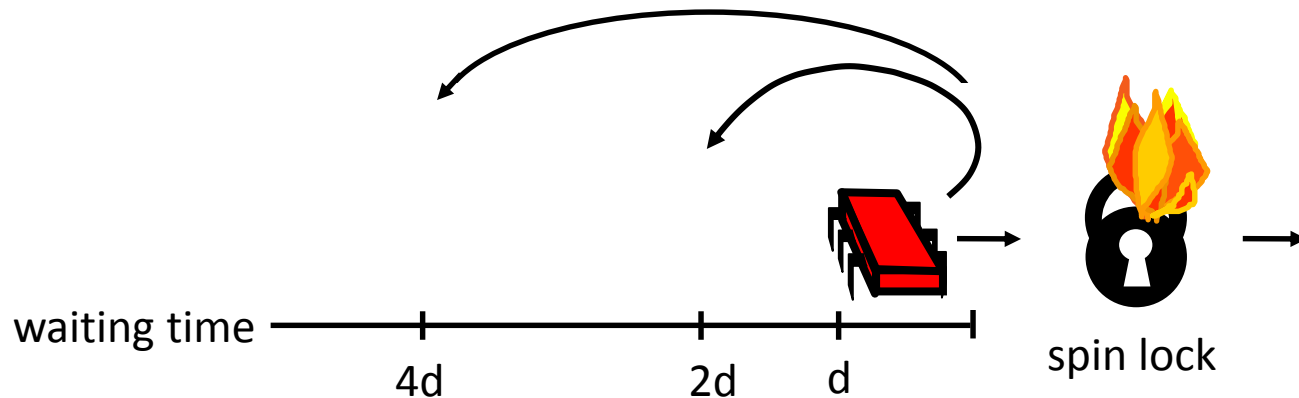
- Now we understand why the TTAS lock performs much better than the TAS lock, but still much worse than an ideal lock!



- How can we do better?

Introduce Delay

- If the lock looks free, but I fail to get it, there must be lots of contention
- It's better to back off than to collide again!
- Example: Exponential Backoff
- Each subsequent failure doubles expected waiting time



Exponential Backoff Lock

```
public class Backoff implements Lock {  
    AtomicBoolean state = new AtomicBoolean(false);
```

```
    public void lock() {
```

```
        int delay = MIN_DELAY;
```

Fix minimum delay

```
        while (true) {
```

```
            while(state.get()) {}
```

```
            if (!state.getAndSet())
```

```
                return;
```

```
            "sleep"(random() % delay);
```

**Back off for
random duration,
but don't swap out**

```
            if (delay < MAX_DELAY)
```

```
                delay = 2 * delay;
```

**Double maximum
delay until an upper
bound is reached**

```
        }
```

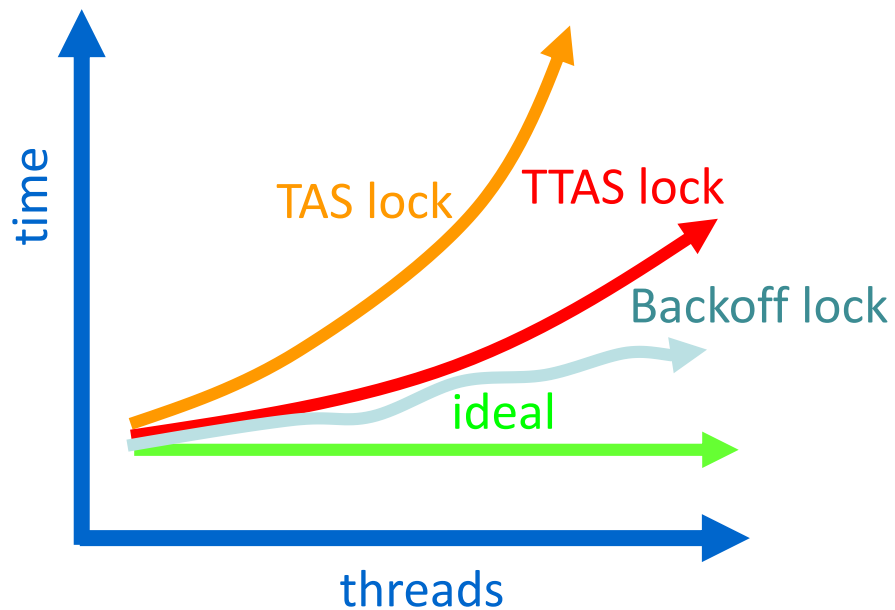
```
    }
```

```
    // unlock() remains the same
```

```
}
```

Backoff Lock: Performance

- The backoff lock outperforms the TTAS lock!
- But it is still not ideal...

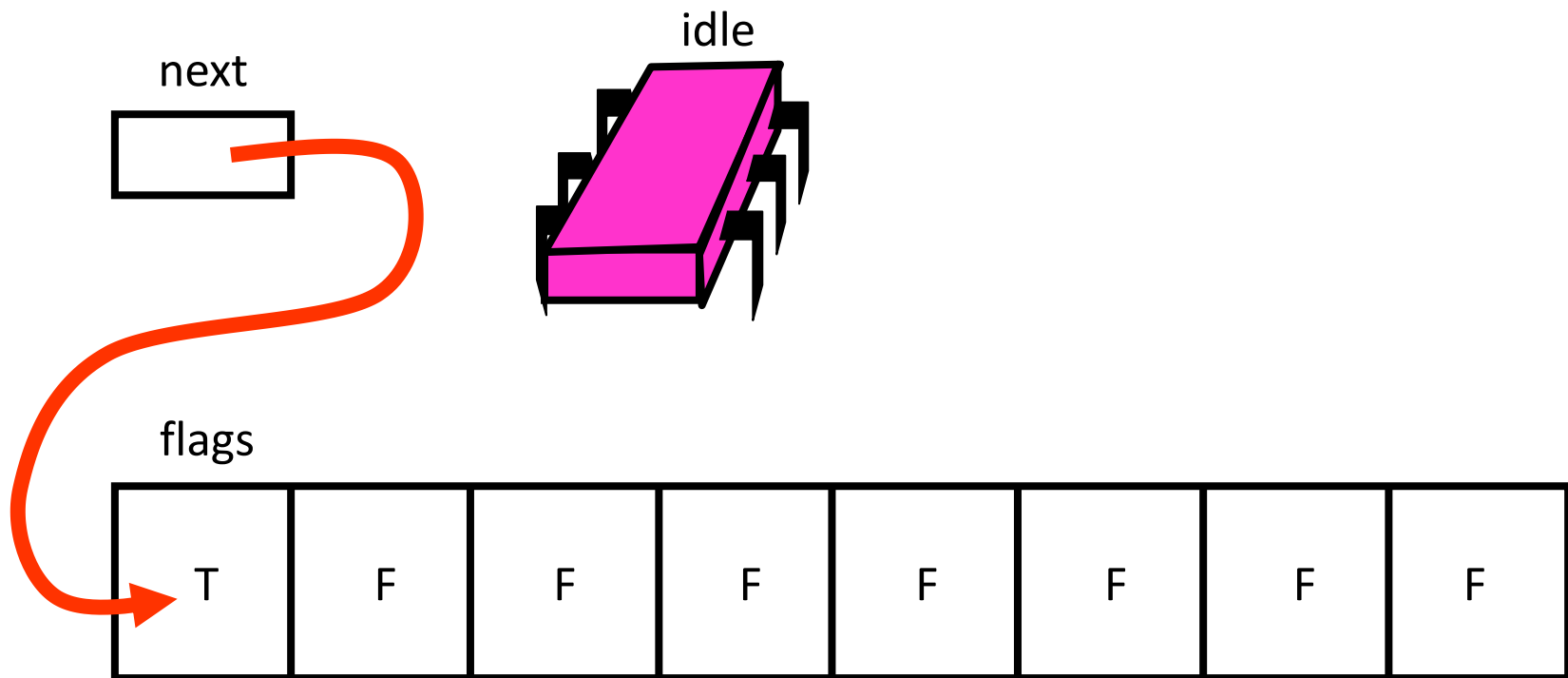


Backoff Lock: Evaluation

- Good
 - Easy to implement
 - Beats TTAS lock
- Bad
 - Must choose parameters carefully
 - Not portable across platforms
- How can we do better?
- Avoid useless invalidations
 - By keeping a queue of threads
- Each thread
 - Notifies next in line
 - Without bothering the others

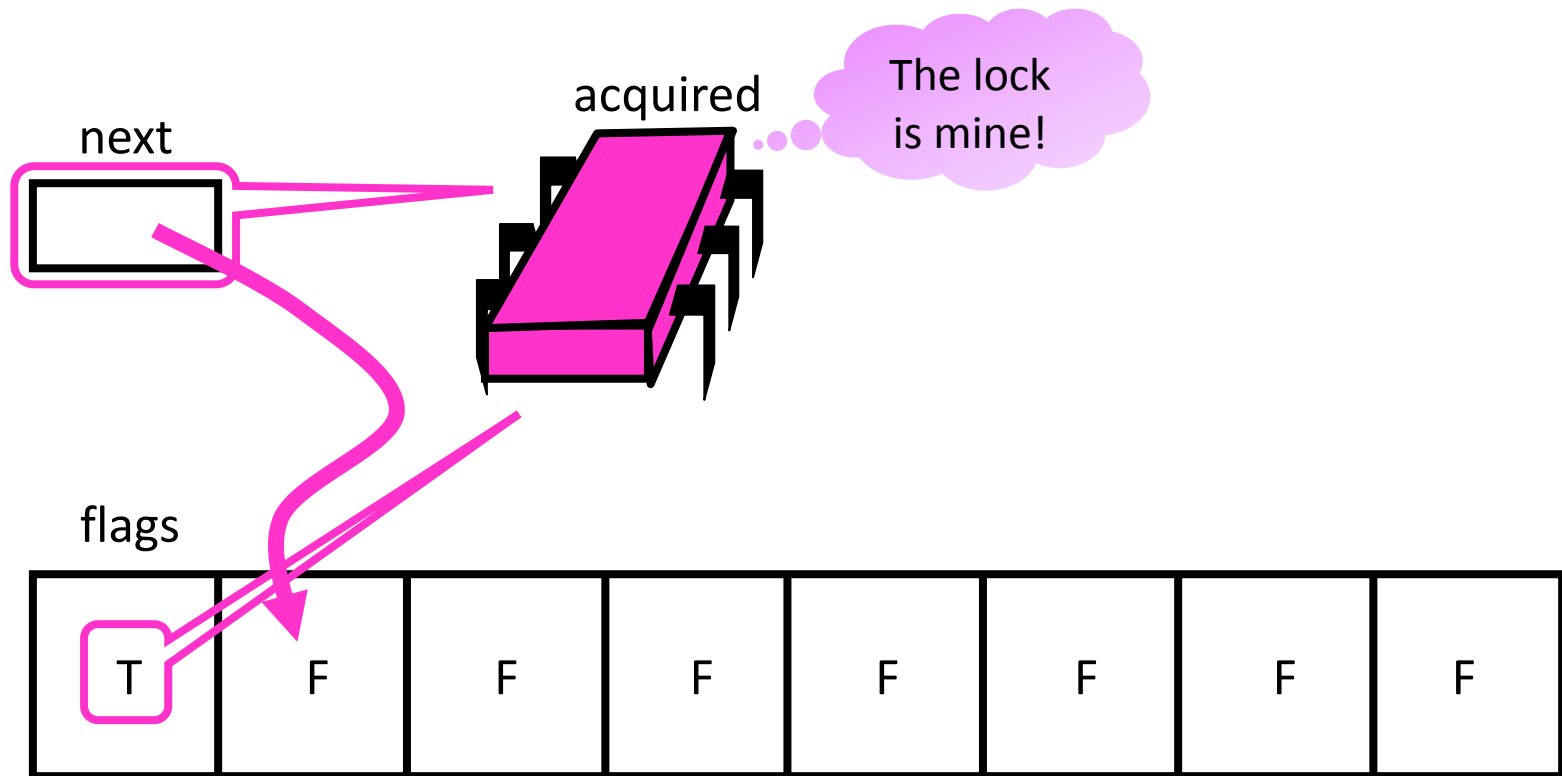
ALock: Initially

- The Anderson queue lock (ALock) is an array-based queue lock
- Threads share an atomic tail field (called next)



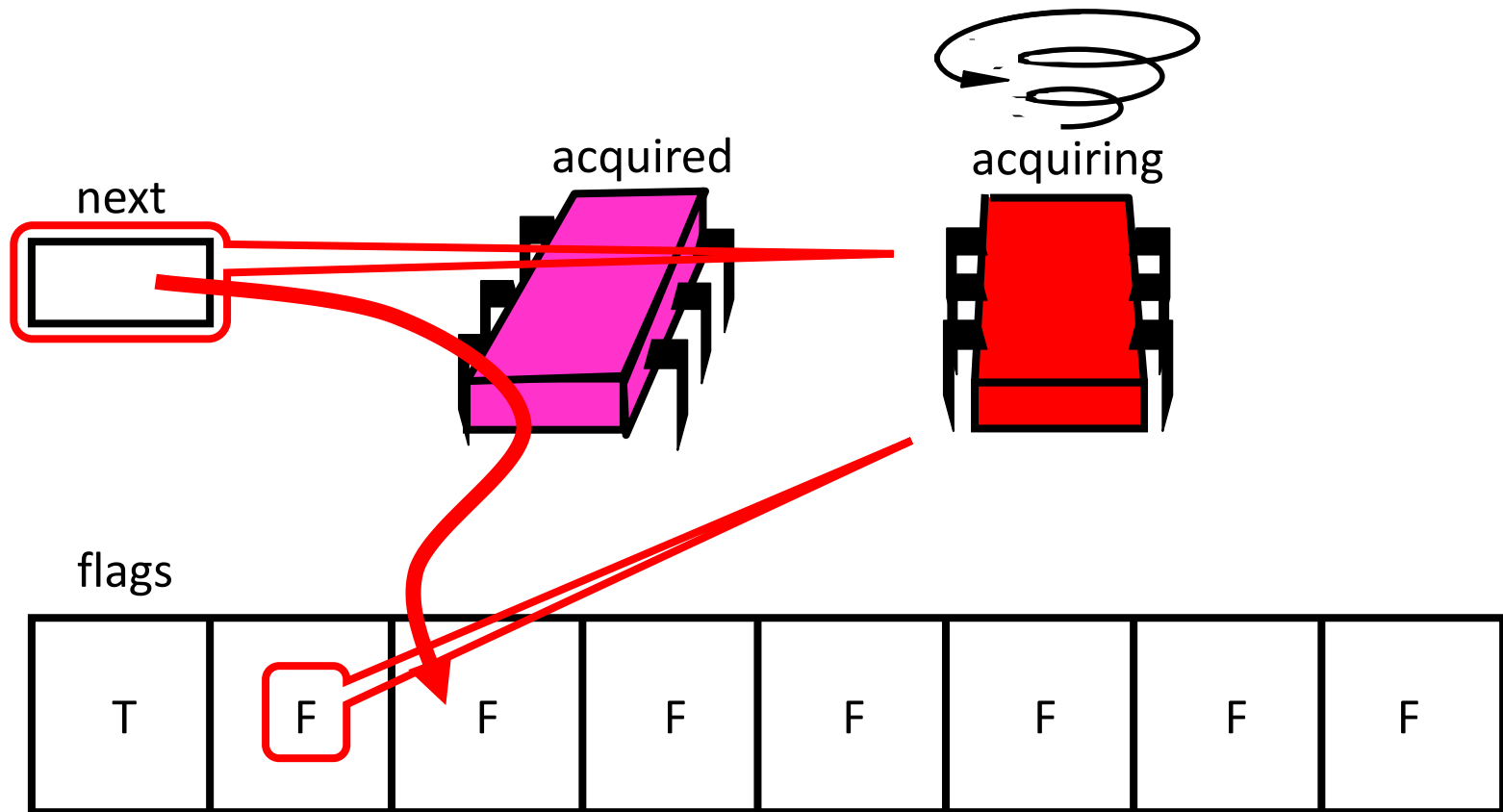
ALock: Acquiring the Lock

- To acquire the lock, each thread atomically increments the tail field
- If the flag is true, the lock is acquired
- Otherwise, spin until the flag is true



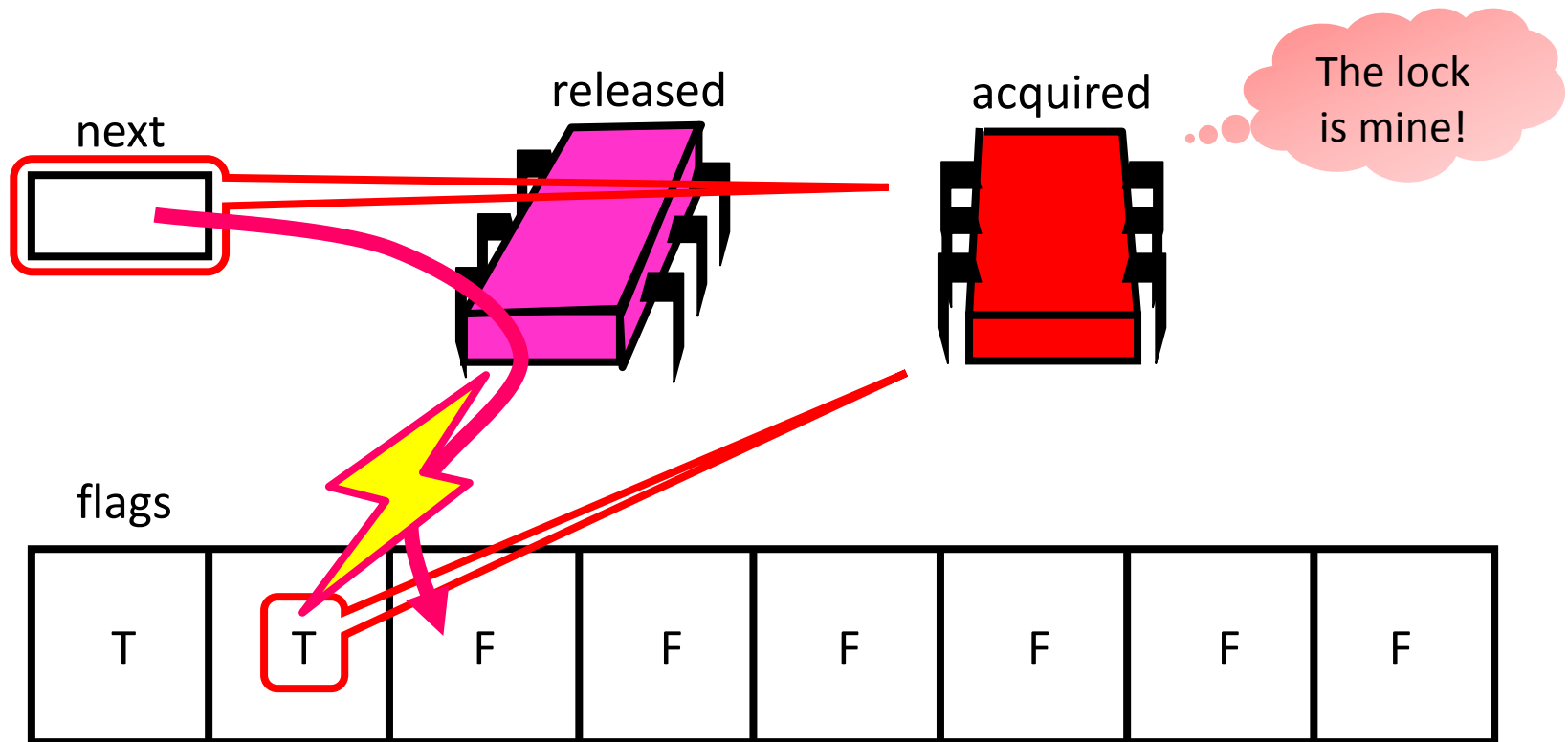
ALock: Contention

- If another thread wants to acquire the lock, it applies get&increment
- The thread spins because the flag is false



ALock: Releasing the Lock

- The first thread releases the lock by setting the next slot to true
- The second thread notices the change and gets the lock



Alock

```
public class Alock implements Lock {  
    boolean[] flags = {true, false, ..., false};  
    AtomicInteger next = new AtomicInteger(0);  
    ThreadLocal<Integer> mySlot;  
  
    public void lock() {  
        mySlot = next.getAndIncrement();  
        while (!flags[mySlot % n]) {}  
        flags[mySlot % n] = false;  
    }  
  
    public void unlock() {  
        flags[(mySlot+1) % n] = true;  
    }  
}
```

One flag per thread

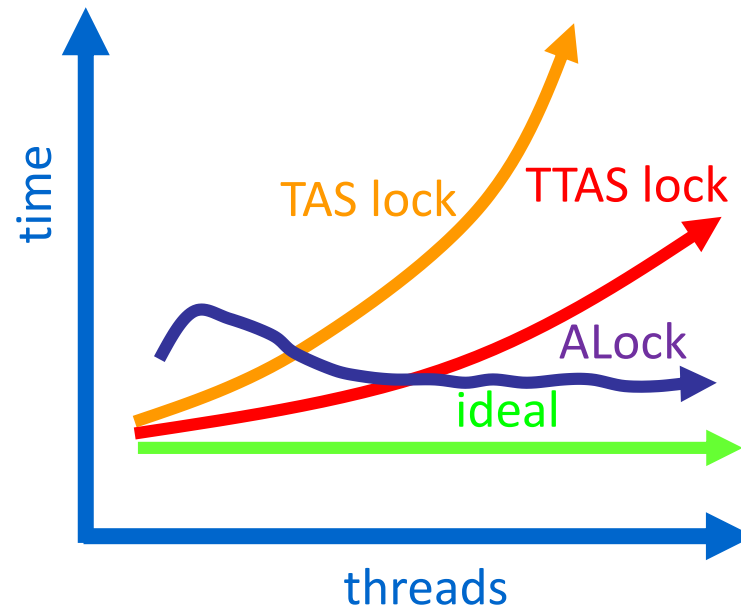
Thread-local variable

Take the next slot

Tell next thread to go

ALock: Performance

- Shorter handover than backoff
- Curve is practically flat
- Scalable performance
- FIFO fairness

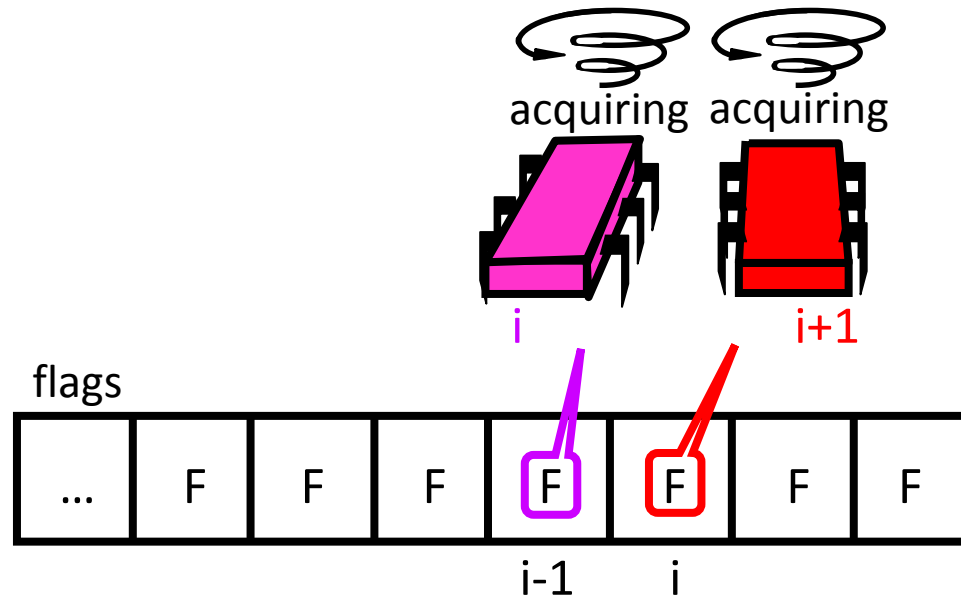


ALock: Evaluation

- Good
 - First truly scalable lock
 - Simple, easy to implement
- Bad
 - One bit per thread
 - Unknown number of threads?

ALock: Alternative Technique

- The threads could update own flag and spin on their predecessor's flag



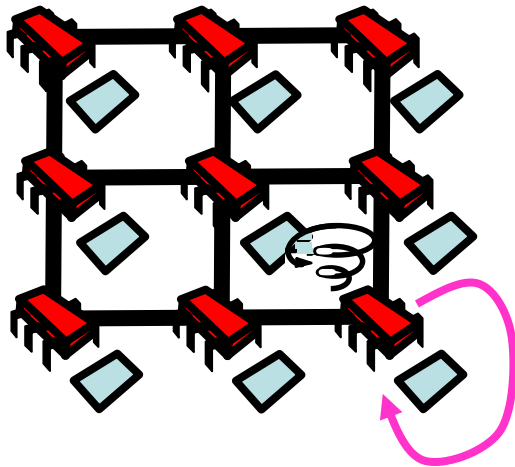
- This is basically what the **CLH lock** does, but using a linked list instead of an array
- Is this a good idea?

Not discussed
in this lecture

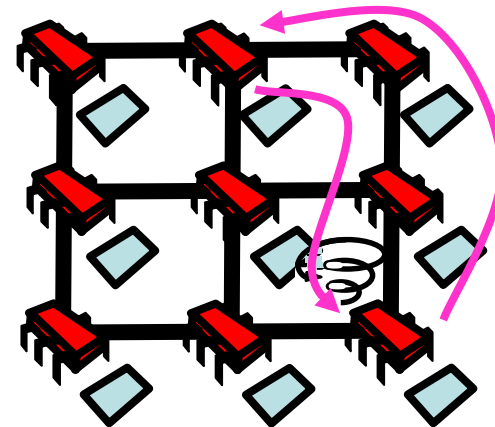
NUMA Architectures

- **Non-Uniform Memory Architecture**
- Illusion
 - Flat shared memory
- Truth
 - No caches (sometimes)
 - Some memory regions faster than others

Spinning on local memory is fast:



Spinning on remote memory is slow:

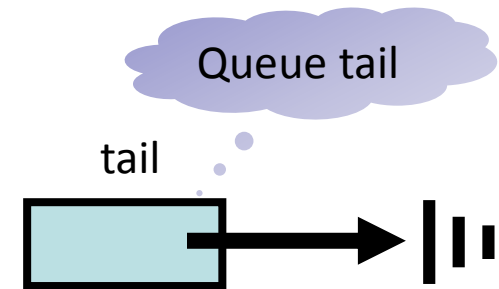
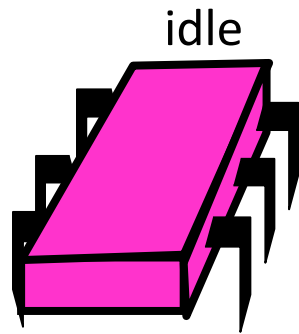


MCS Lock

- Idea
 - Use a linked list instead of an array → small, constant-sized space
 - Spin on own flag, just like the Anderson queue lock
- The space usage
 - L = number of locks
 - N = number of threads
- of the Anderson lock is $O(LN)$
- of the MCS lock is $O(L+N)$

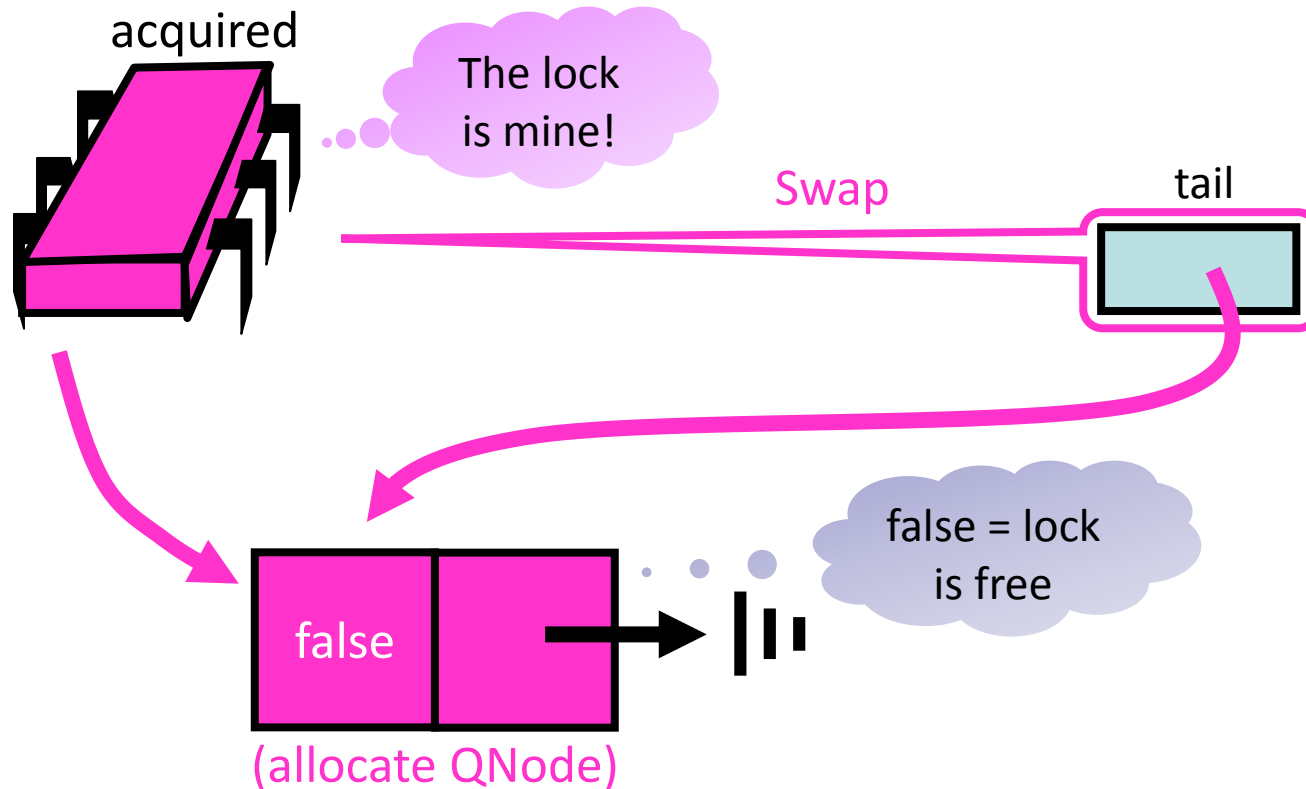
MCS Lock: Initially

- The lock is represented as a linked list of QNodes, one per thread
- The tail of the queue is shared among all threads



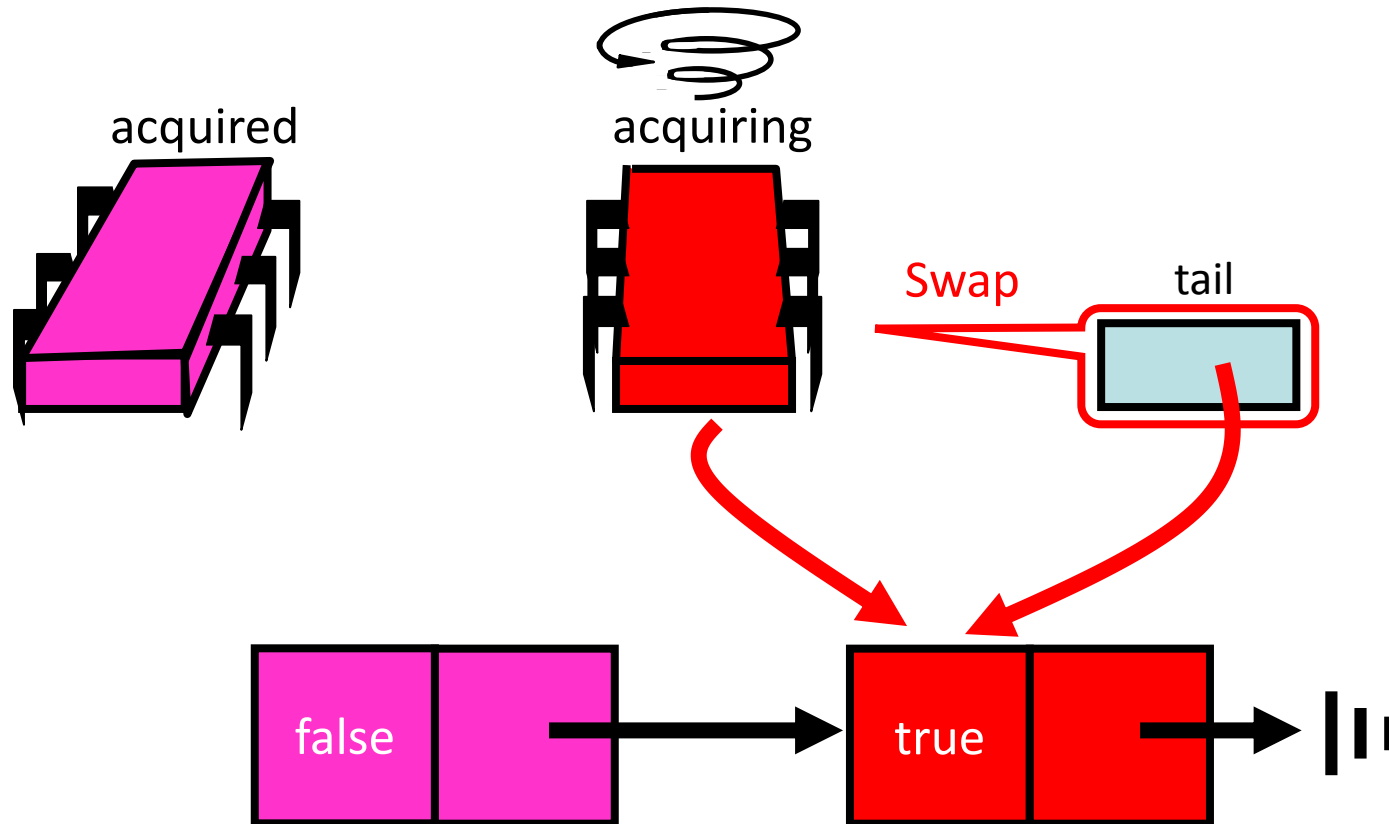
MCS Lock: Acquiring the Lock

- To acquire the lock, the thread places its QNode at the tail of the list by swapping the tail to its QNode
- If there is no predecessor, the thread acquires the lock



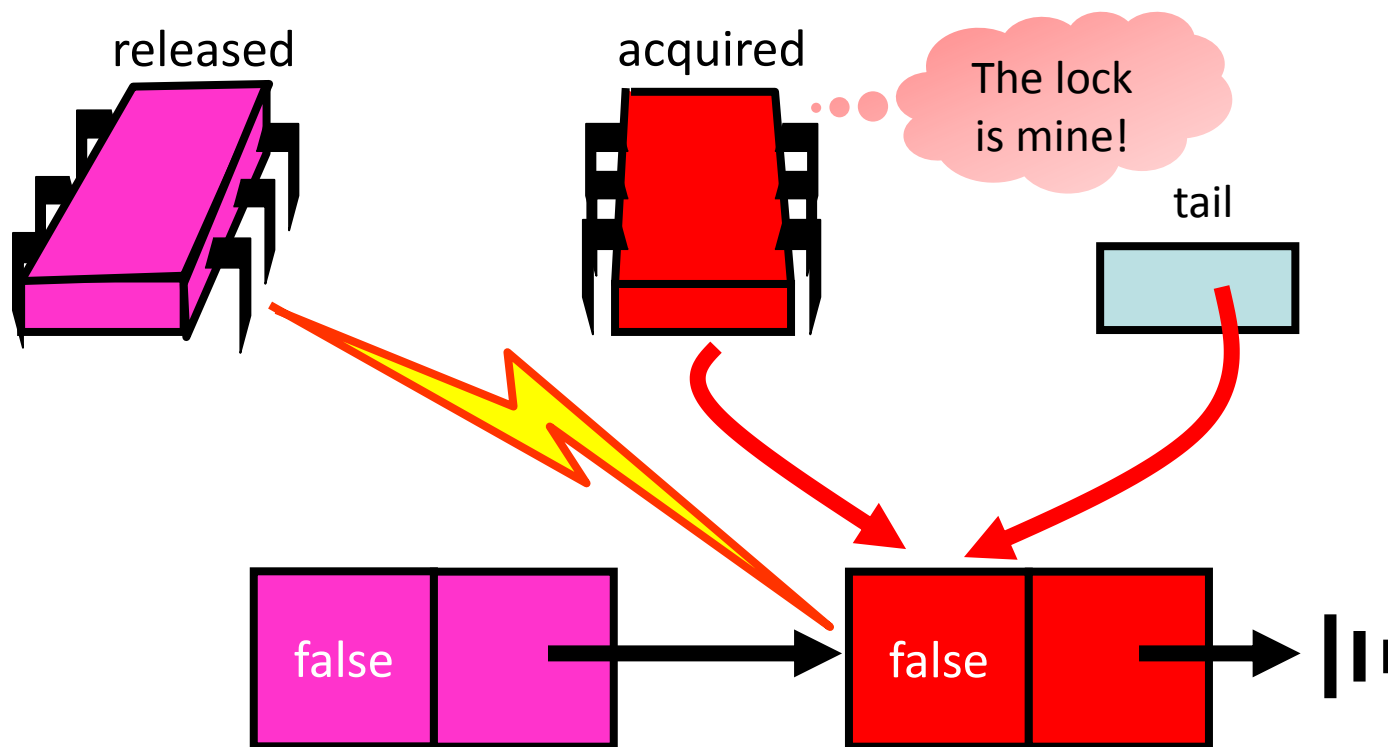
MCS Lock: Contention

- If another thread wants to acquire the lock, it again applies swap
- The thread spins on its own QNode because there is a predecessor



MCS Lock: Releasing the Lock

- The first thread releases the lock by setting its successor's QNode to false



MCS Queue Lock

```
public class QNode {  
    boolean locked = false;  
    QNode next = null;  
}
```

MCS Queue Lock

```
public class MCSLock implements Lock {  
    AtomicReference tail;
```

```
    public void lock() {  
        QNode qnode = new QNode();
```

```
        QNode pred = tail.getAndSet(qnode);
```

```
        if (pred != null) {  
            qnode.locked = true;
```

```
            pred.next = qnode;
```

```
            while (qnode.locked) {}
```

```
        }
```

```
    }
```

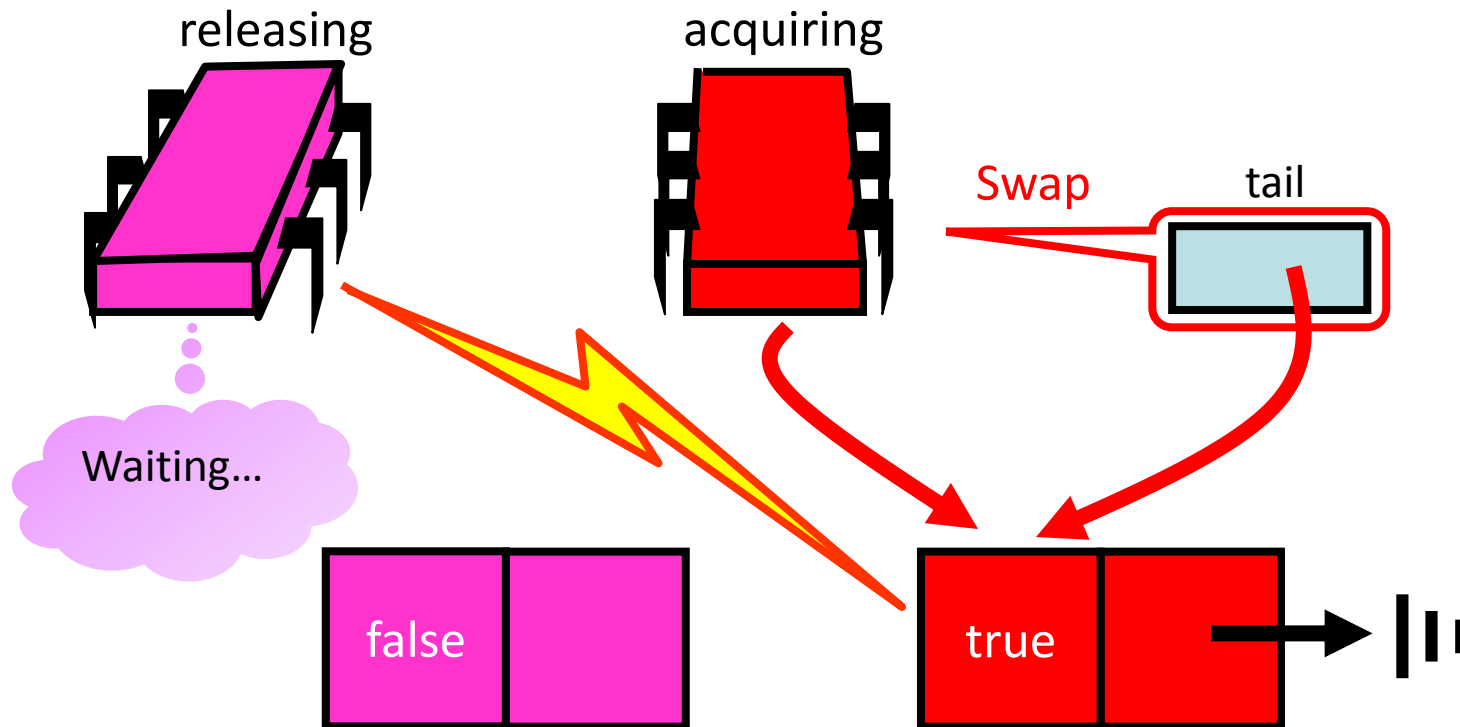
```
    ...
```

Add my node to the tail

Fix if queue was non-empty

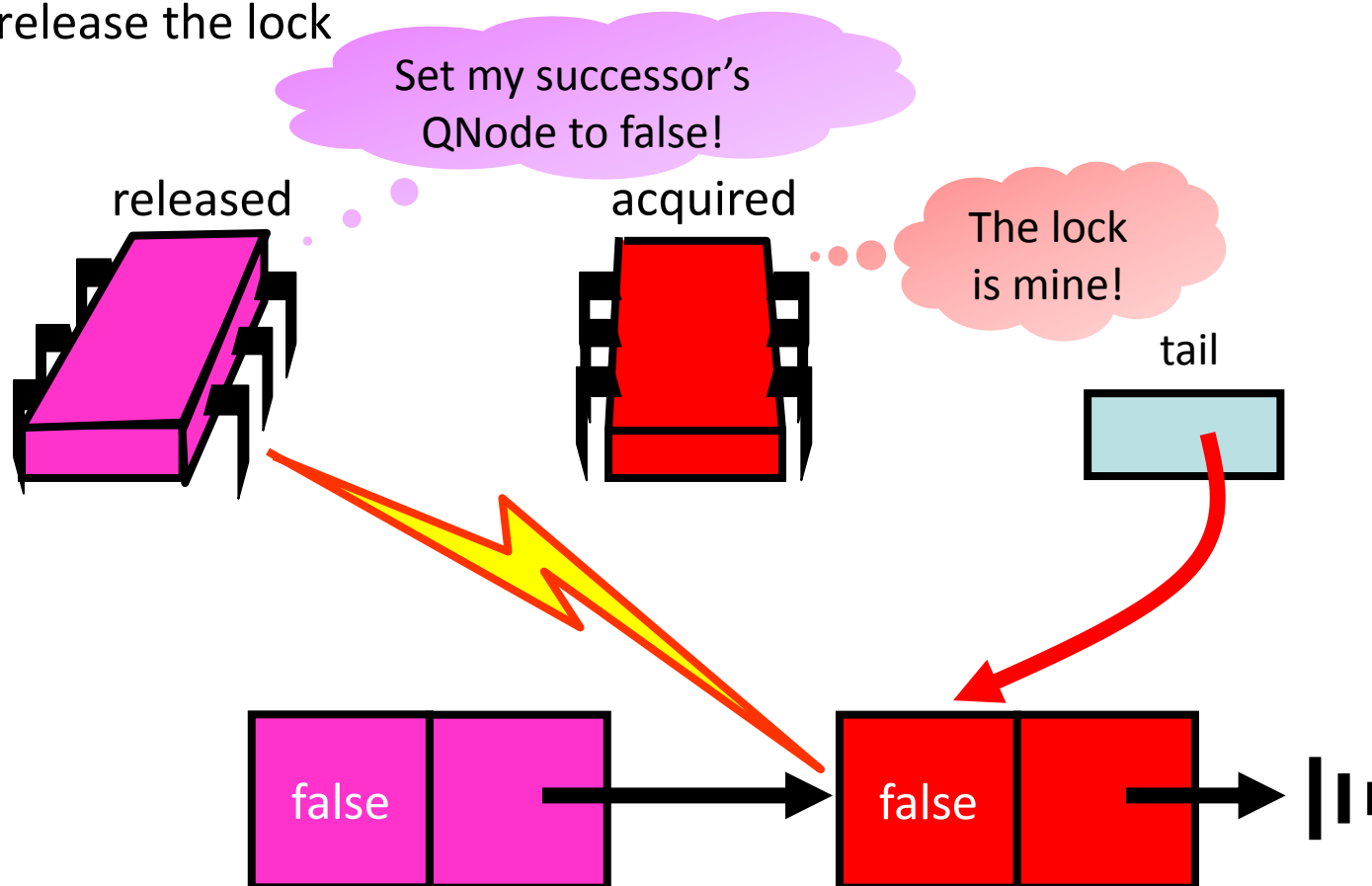
MCS Lock: Unlocking

- If there is a successor, unlock it. But, be cautious!
- Even though a QNode does not have a successor, the purple thread knows that another thread is active because tail does not point to its QNode!



MCS Lock: Unlocking Explained

- As soon as the pointer to the successor is set, the purple thread can release the lock



MCS Queue Lock

...

```
public void unlock() {
```

```
    if (qnode.next == null) {
```

```
        if (tail.CAS(qnode, null))  
            return;
```

```
        while (qnode.next == null) {}
```

```
    }
```

```
    qnode.next.locked = false;
```

```
}
```

```
}
```

Missing successor?

If really no successor,
tail = null

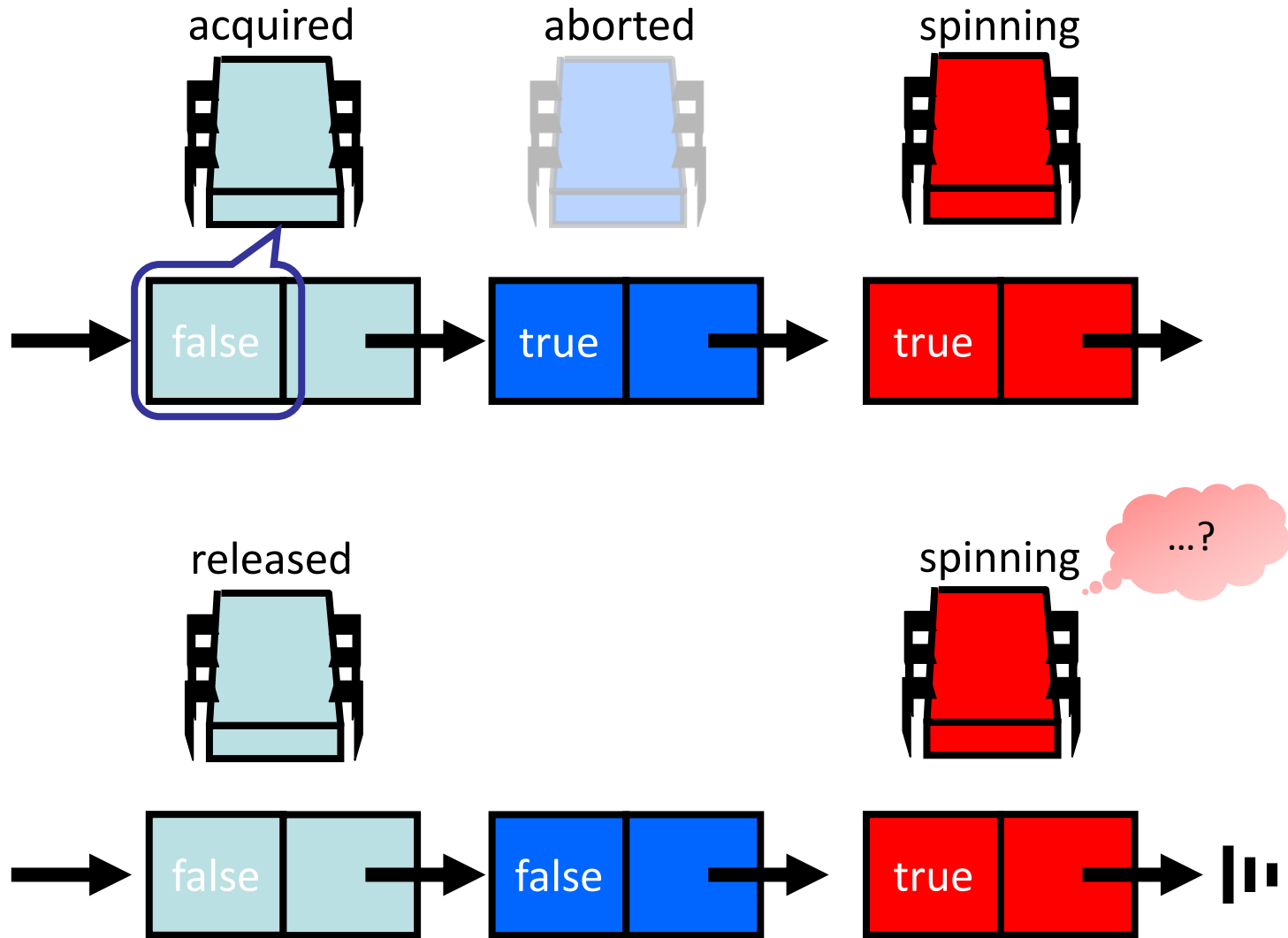
Otherwise, wait for
successor to catch up

Pass lock to successor

Abortable Locks

- What if you want to give up waiting for a lock?
- For example
 - Time-out
 - Database transaction aborted by user
- Back-off Lock
 - Aborting is trivial: Just return from lock() call!
 - Extra benefit: No cleaning up, wait-free, immediate return
- Queue Locks
 - Can't just quit: Thread in line behind will starve
 - Need a graceful way out...

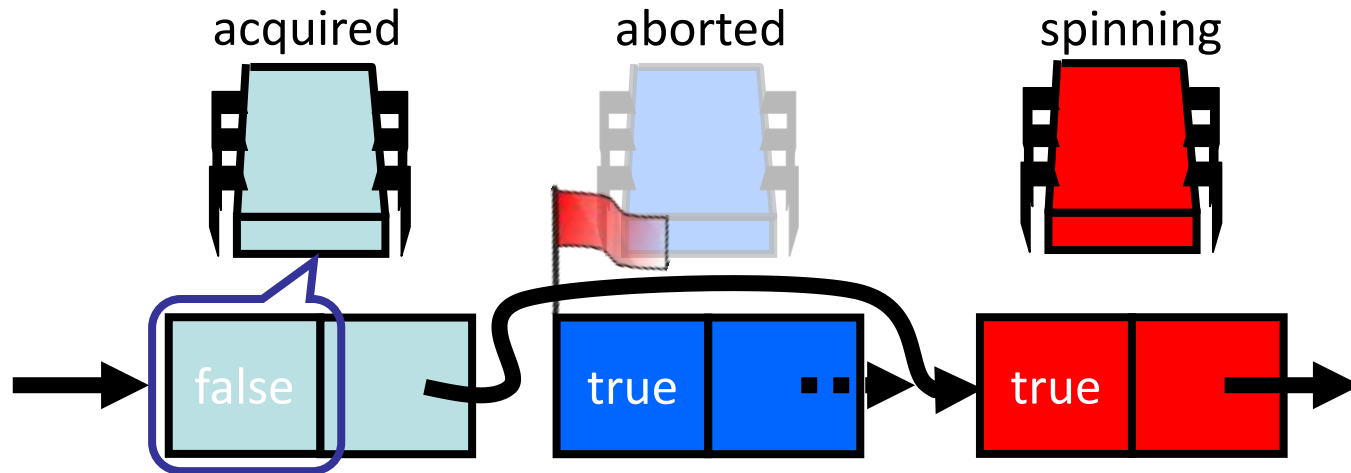
Problem with Queue Locks



Abortable MCS Lock

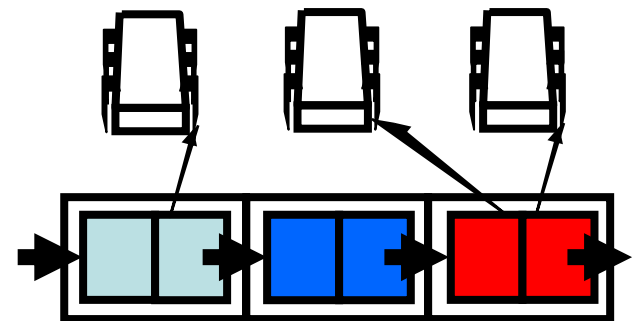
- A mechanism is required to recognize and remove aborted threads
 - A thread can set a flag indicating that it aborted
 - The predecessor can test if the flag is set
 - If the flag is set, its new successor is the successor's successor
 - How can we handle concurrent aborts? This is not discussed in this lecture

Spinning on remote object...?!



Composite Locks

- Queue locks have many advantages
 - FIFO fairness, fast lock release, low contentionbut require non-trivial protocols to handle aborts (and recycling of nodes)
- Backoff locks support trivial time-out protocols but are not scalable and may have slow lock release times
- A **composite lock** combines the best of both approaches!
- Short fixed-sized array of lock nodes
- Threads randomly pick a node and try to acquire it
- Use backoff mechanism to acquire a node
- Nodes build a queue
- Use a queue lock mechanism to acquire the lock



One Lock To Rule Them All?

- TTAS+Backoff, MCS, Abortable MCS...
- Each better than others in some way
- There is not a single best solution
- Lock we pick really depends on
 - the application
 - the hardware
 - which properties are important

Handling Multiple Threads

- Adding threads should not **lower** the throughput
 - Contention effects can mostly be fixed by Queue locks
- Adding threads should **increase** throughput
 - Not possible if the code is inherently sequential
 - Surprising things are parallelizable!
- How can we guarantee **consistency** if there are many threads?

Coarse-Grained Synchronization

- Each method locks the object
 - Avoid contention using queue locks
 - Mostly easy to reason about
 - This is the standard Java model (**synchronized** blocks and methods)
- Problem: Sequential bottleneck
 - Threads “stand in line”
 - Adding more threads does not improve throughput
 - We even struggle to keep it from getting worse...
- So why do we even use a multiprocessor?
 - Well, some applications are inherently parallel...
 - We focus on exploiting non-trivial parallelism

Credits

- The TTAS lock is due to Kruskal, Rudolph, and Snir, 1988.
- Tom Anderson invented the ALock, 1990.
- The MCS lock is due to Mellor-Crummey and Scott, 1991.

That's all!

Questions & Comments?



Roger Wattenhofer