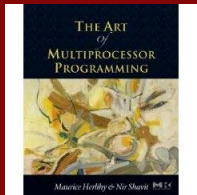
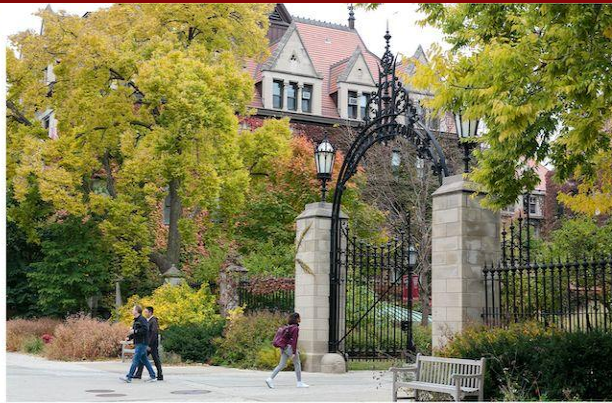


MPCS 52060 - Parallel Programming

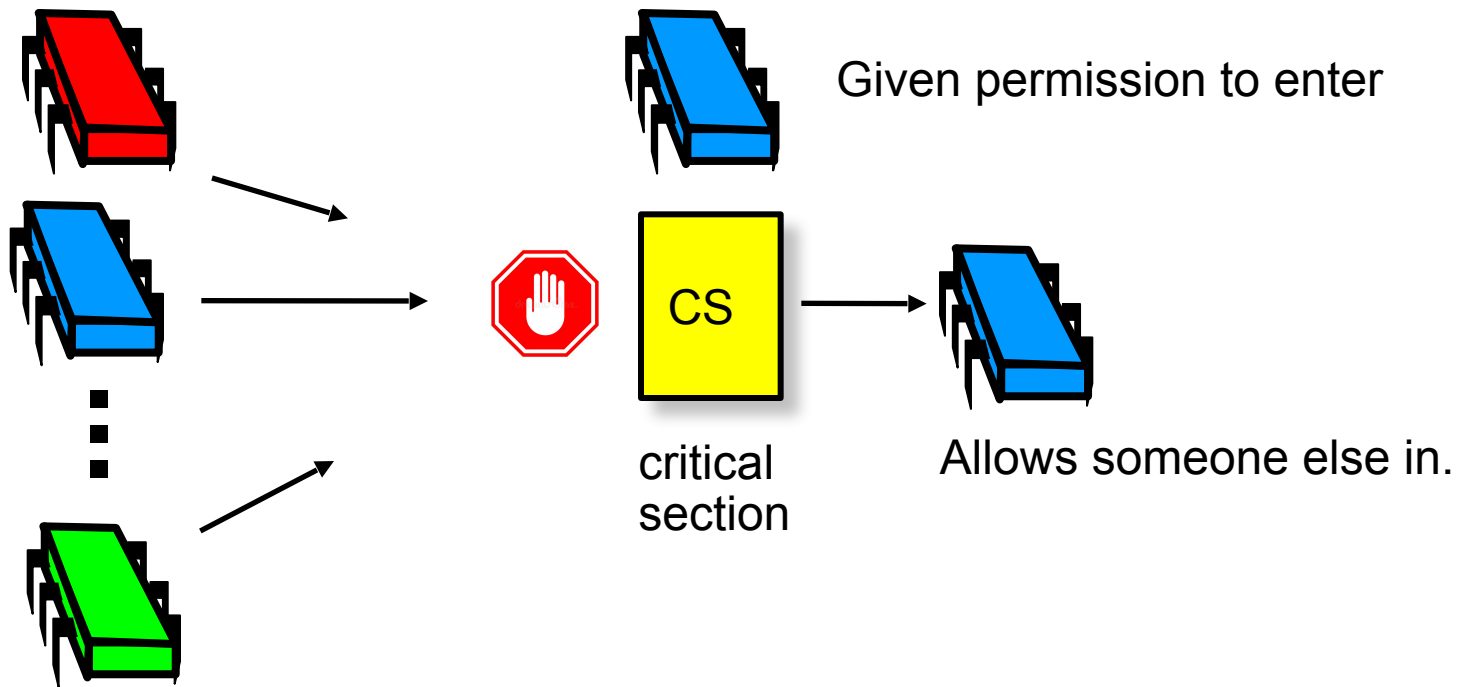
M3: Principles of Mutual Exclusion



Original slides from “The Art of Multiprocessor Programming by Maurice Herlihy & Nir Shavit” With modifications by Lamont Samuels

M3 Objective

- How we are going to ensure determinism when threads want to enter in the critical section?



Principles of Mutual Exclusion

Formalizing Concurrent Computation

Two types of formal properties in asynchronous computation:

- **Safety** Properties:
 - Nothing bad happens ever
 - For example - a traffic light never displays green in all directions, even if power fails.
- **Liveness** Properties:
 - States that a particular “good” thing will happen.
 - For example - a red light will eventually turn green.

Formalizing Critical Sections

- Synchronization primitives need to adhere to the following properties and principles about critical sections in order to be correct:
 - **Mutual Exclusion Property:**
 - Critical sections of different threads do not overlap. Only one thread is executing a critical section at a time
 - Guarantees that a computation's results are correct.
 - This is a safety property.
 - **Deadlock-freedom property:**
 - If multiple threads simultaneously request to enter a critical section, then it must allow one to proceed
 - Threads outside the critical section have no say in which thread can proceed into the critical section, only those currently waiting have influence.
 - It implies the system never “freezes”.
 - This is a liveness property.

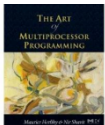
Formalizing Critical Sections

- **Starvation-freedom property:**
 - Every thread that attempts to acquire the lock eventually succeeds.
 - Many mutual exclusive algorithms in practice are not starvation free because its less likely starvation will occur in those algorithms.
 - There is no guarantee on how long thread will wait to acquire the lock.
 - Also known as lockout freedom or bounded-waiting
 - This is a liveness property.
- **Fairness Principle:**
 - A thread who just left the critical section cannot immediately re-enter the critical section if other threads have already requested to enter the critical section.
 - Some algorithms place bounds on how long a thread can wait.

Implementing Locks

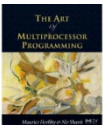
Two-Thread vs n -Thread Solutions

- 2-thread solutions first
 - Illustrate most basic ideas
 - Fits on one slide
- Then n -thread solution
- You will never use these protocols
 - Get over it
- You are advised to understand them
 - The same issues show up everywhere
 - Except hidden and more complex



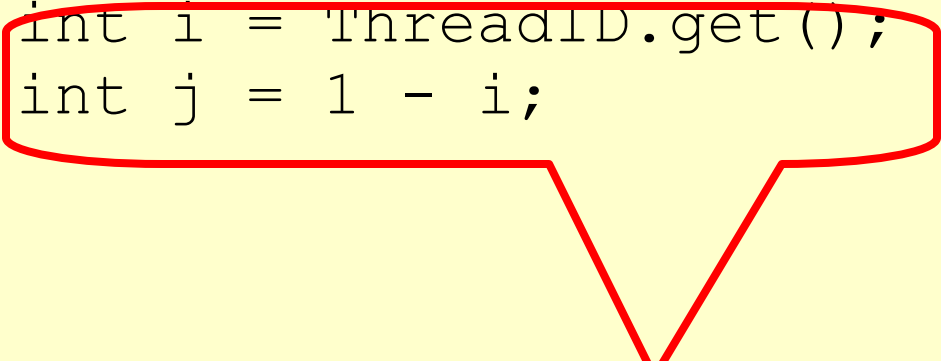
Two-Thread Conventions

```
class ... implements Lock {  
    ...  
    // thread-local index, 0 or 1  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        ...  
    }  
}
```



Two-Thread Conventions

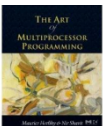
```
class ... implements Lock {  
    ...  
    // thread-local index, 0 or 1  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
    ...  
    }  
}
```



Henceforth: **i** is current
thread, **j** is other thread

LockOne

```
class LockOne implements Lock {  
    private boolean[] flag = new boolean[2];  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
  
    public void unlock() {  
        int i = ThreadID.get();  
        flag[i] = false;  
    }  
}
```



LockOne

```
class LockOne implements Lock {  
    private boolean[] flag = new boolean[2];  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```

Each thread has flag

LockOne

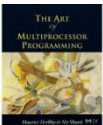
```
class LockOne implements Lock {  
    private boolean[] flag = new boolean[2];  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```

Set my flag

LockOne

```
class LockOne implements Lock {  
    private boolean[] flag = new boolean[2];  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```

Wait for other flag to become
false

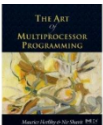


Deadlock Freedom

- LockOne Fails deadlock-freedom
 - Concurrent execution can deadlock

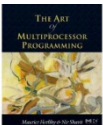
```
flag[i] = true;    flag[j] = true;  
while (flag[j]) {} while (flag[i]) {}
```

- Sequential executions OK



LockTwo

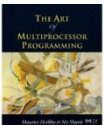
```
public class LockTwo implements Lock {  
    private int victim;  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
  
        victim = i;  
        while (victim == i) {}  
    }  
  
    public void unlock() {}  
}
```



LockTwo

```
public class LockTwo implements Lock {  
    private int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
  
    public void unlock() {}  
}
```

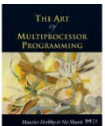
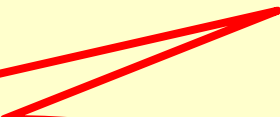
Let other go first



LockTwo

```
public class LockTwo implements Lock {  
    private int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
  
    public void unlock() {}  
}
```

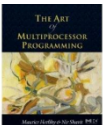
Wait for permission



LockTwo

```
public class Lock2 implements Lock {  
    private int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
    public void unlock() {}  
}
```

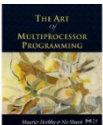
Nothing to do



LockTwo Claims

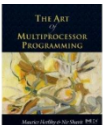
- Satisfies mutual exclusion
 - If thread **i** in critical section
 - Then **victim == j**
 - Cannot be both 0 and 1
- Not deadlock free
 - Sequential execution deadlocks
 - Concurrent execution does not

```
public void LockTwo() {  
    victim = i;  
    while (victim == i) {};  
}
```



Peterson's Algorithm

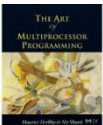
```
public void lock() {  
    int i = ThreadID.get();  
    int j = 1 - i;  
    flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i) {};  
}  
public void unlock() {  
    flag[i] = false;  
}
```



Peterson's Algorithm

Announce I'm
interested

```
public void lock() {  
    flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i) {}  
}  
public void unlock() {  
    flag[i] = false;  
}
```

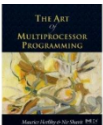


Peterson's Algorithm

```
public void lock() {  
    flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i) {}  
}  
public void unlock() {  
    flag[i] = false;  
}
```

Announce I'm
interested

Defer to other



Peterson's Algorithm

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

```
    flag[i] = true;
```

```
    victim = i;
```

```
    while (flag[j] && victim == i) {};
```

```
}
```

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

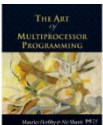
```
    flag[i] = false;
```

```
}
```

Announce I'm
interested

Defer to other

Wait while other
interested & I'm the
victim



Peterson's Algorithm

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

```
    flag[i] = true;
```

```
    victim = i;
```

```
    while (flag[j] && victim == i) {};
```

```
}
```

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

```
    flag[i] = false;
```

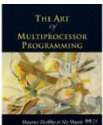
```
}
```

Announce I'm
interested

Defer to other

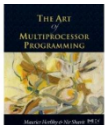
Wait while other
interested & I'm the
victim

No longer
interested



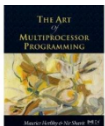
Peterson's Algorithm

- Satisfies mutual exclusion & deadlock freedom properties
 - Uses both lock-one and lock-two strategies
- Downside: only works for two threads



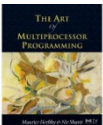
Bakery Algorithm

- N-threaded locking algorithm
- Provides First-Come-First-Served
- How?
 - Take a “number”
 - Wait until lower numbers have been served
- Lexicographic order
 - $(a,i) > (b,j)$
 - If $a > b$, or $a = b$ and $i > j$



Bakery Algorithm

```
class Bakery implements Lock {  
    boolean[] flag;  
    Label[] label;  
    public Bakery (int n) {  
        flag = new boolean[n];  
        label = new Label[n];  
        for (int i = 0; i < n; i++) {  
            flag[i] = false; label[i] = 0;  
        }  
    }  
    ...  
}
```



Bakery Algorithm

```
class Bakery implements Lock {
```

```
    boolean[] flag;
```

```
    Label[] label;
```

```
    public Bakery (int n) {
```

```
        flag = new boolean[n];
```

```
        label = new Label[n];
```

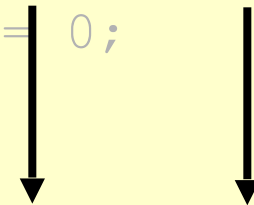
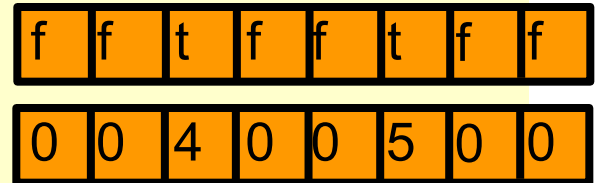
```
        for (int i = 0; i < n; i++) {
```

```
            flag[i] = false; label[i] = 0;
```

```
        }
```

```
    }
```

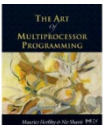
```
    ...
```



CS

Bakery Algorithm

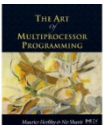
```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1]) + 1;  
        while ( $\exists k$  flag[k]  
                && (label[i], i) > (label[k], k));  
    }
```



Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1]) + 1;  
        while ( $\exists$  k flag[k]  
                && (label[i], i) > (label[k], k));  
    }  
}
```

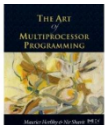
I'm interested



Bakery Algorithm

Take increasing
label (read labels in
some arbitrary
order)

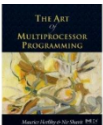
```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1]) + 1;  
        while ( $\exists$  k flag[k]  
                && (label[i], i) > (label[k], k));  
    }
```



Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i], i) > (label[k], k));  
    }
```

Someone is
interested

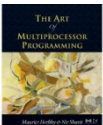


Bakery Algorithm

```
class Bakery implements Lock {  
    boolean flag[n];  
    int label[n];  
  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
            && (label[i],i) > (label[k],k));  
    }  
}
```

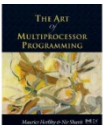
Someone is
interested ...

... whose (label,i) in
lexicographic order is lower



Bakery Algorithm

```
class Bakery implements Lock {  
  
    ...  
  
    public void unlock() {  
        flag[i] = false;  
    }  
}
```



Bakery Algorithm

```
class Bakery implements Lock {
```

```
    ...
```

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

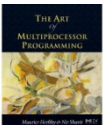
```
        flag[i] = false;
```

```
    }
```

```
}
```

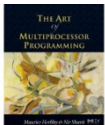
No longer
interested

labels are always increasing



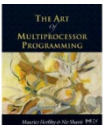
Bakery Algorithm

- Has no deadlock and adheres to mutual exclusion property
- There is always one thread with earliest label
- Is there a problem with the labeling portion of the algorithm?



Deep Philosophical Question

- The Bakery Algorithm is
 - Succinct,
 - Elegant, and
 - Fair.
- Q: So why isn't it practical?
- A: Well, you have to read **N** distinct variables



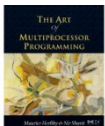
Practical Implementation of Locks

Mutex in Go

- Go has a package called `sync` that provides basic synchronization primitives such as mutual exclusion locks.
 - The `sync` package Go's provides mutual exclusion with `sync.Mutex` and its two methods:
 - `m.Lock()`: locks `m`. If the lock is already in use, the calling goroutine blocks until the mutex is available.
 - `m.Unlock()`: `Unlock` unlocks `m`. It is a run-time error if `m` is not locked on entry to `Unlock`.
- Now let's take a look at how locks are implemented behind the scenes.

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

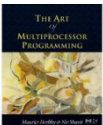
- Keep trying
 - “spin” or “busy-wait”
 - When spinning, a thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.
 - Can be very wasteful of CPU cycles.
 - Can also be unreliable if compiler optimization is turned on.
 - Good if delays are short
- Give up the processor
 - Good if delays are long
 - Always good on uniprocessor



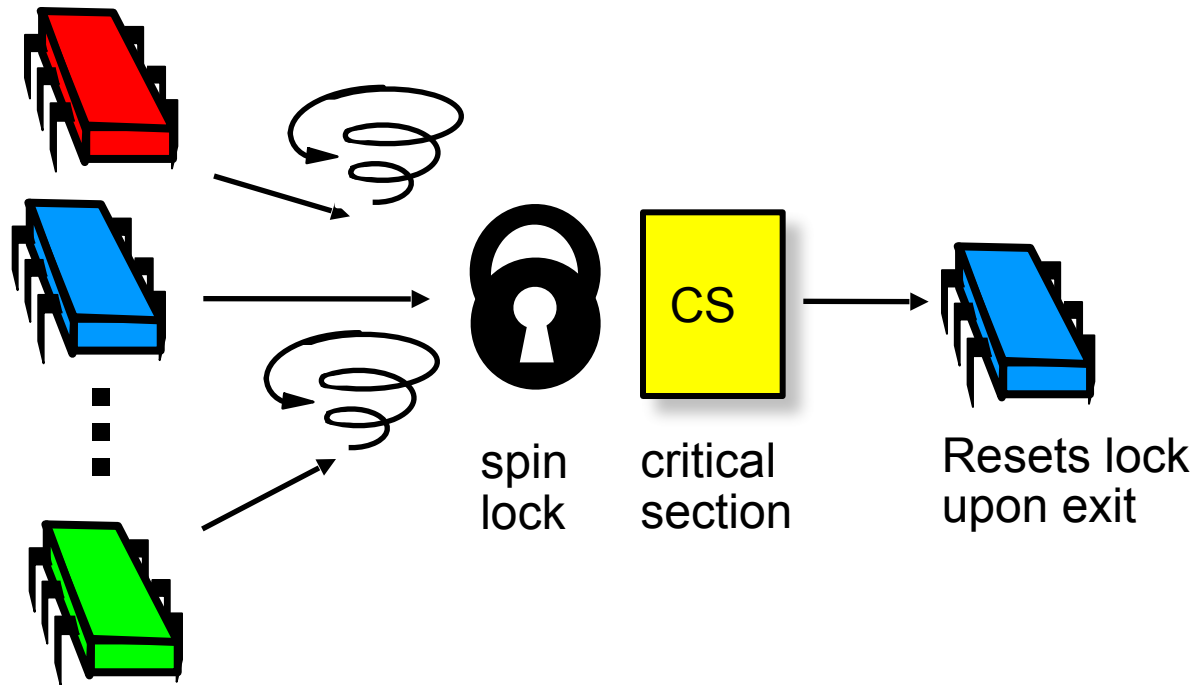
What Should you do if you can't get a lock?

- Keep trying
 - “spin” or “busy-wait”
 - When spinning, a thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.
 - Can be very wasteful of CPU cycles.
 - Can also be unreliable if compiler optimization is turned on.
 - Good if delays are short
- Give up the processor
 - Good if delays are long
 - Always good on uniprocessor

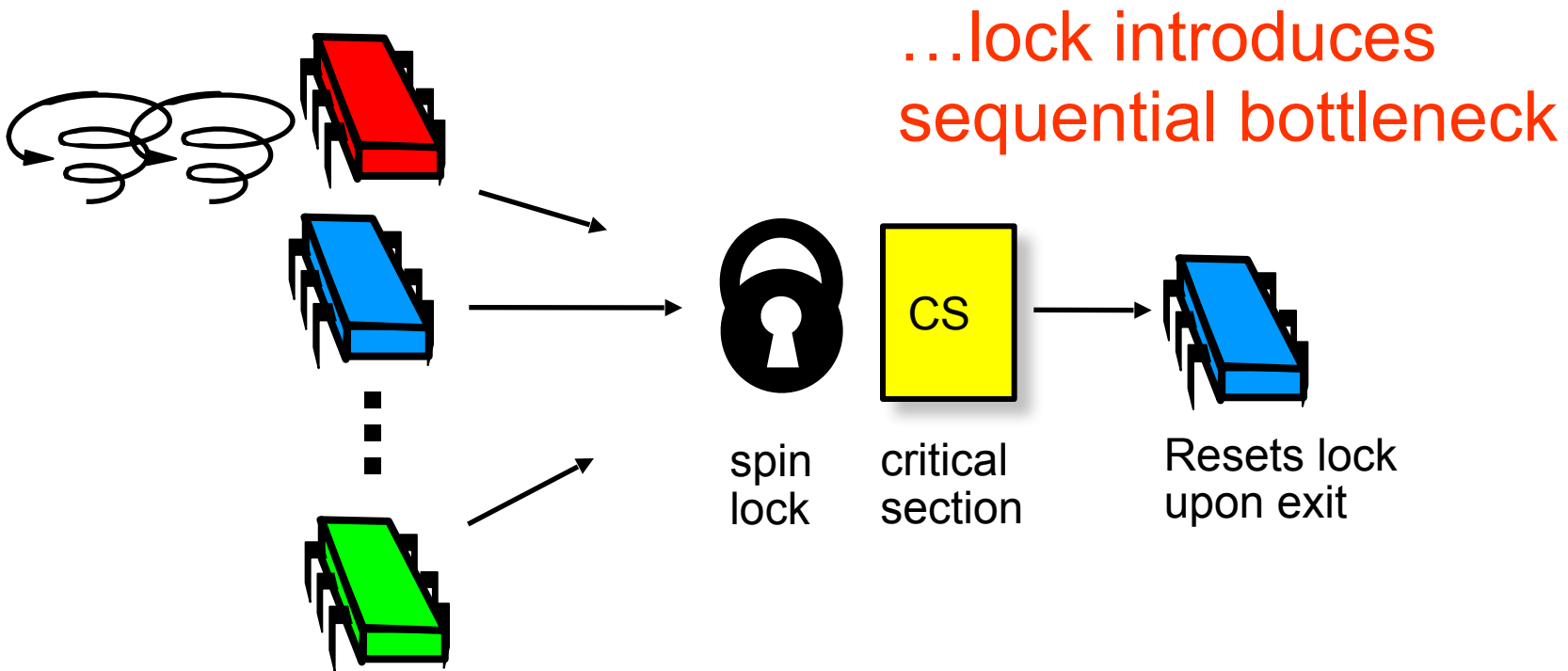
our focus



Basic Spin-Lock

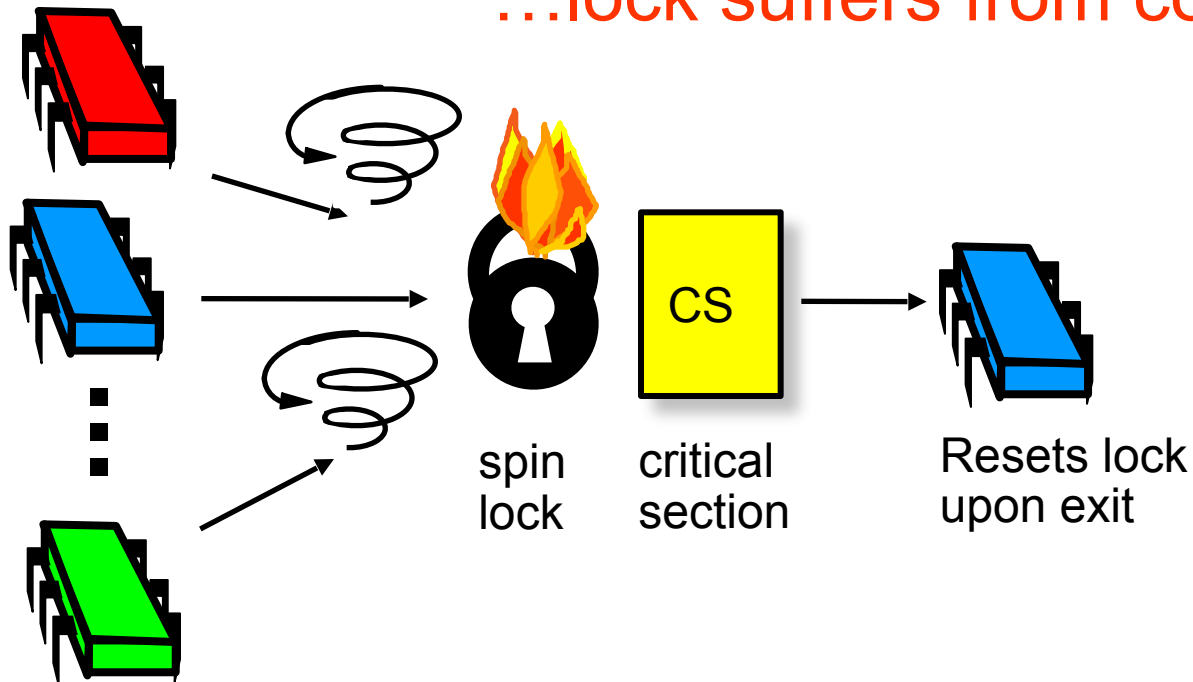


Basic Spin-Lock



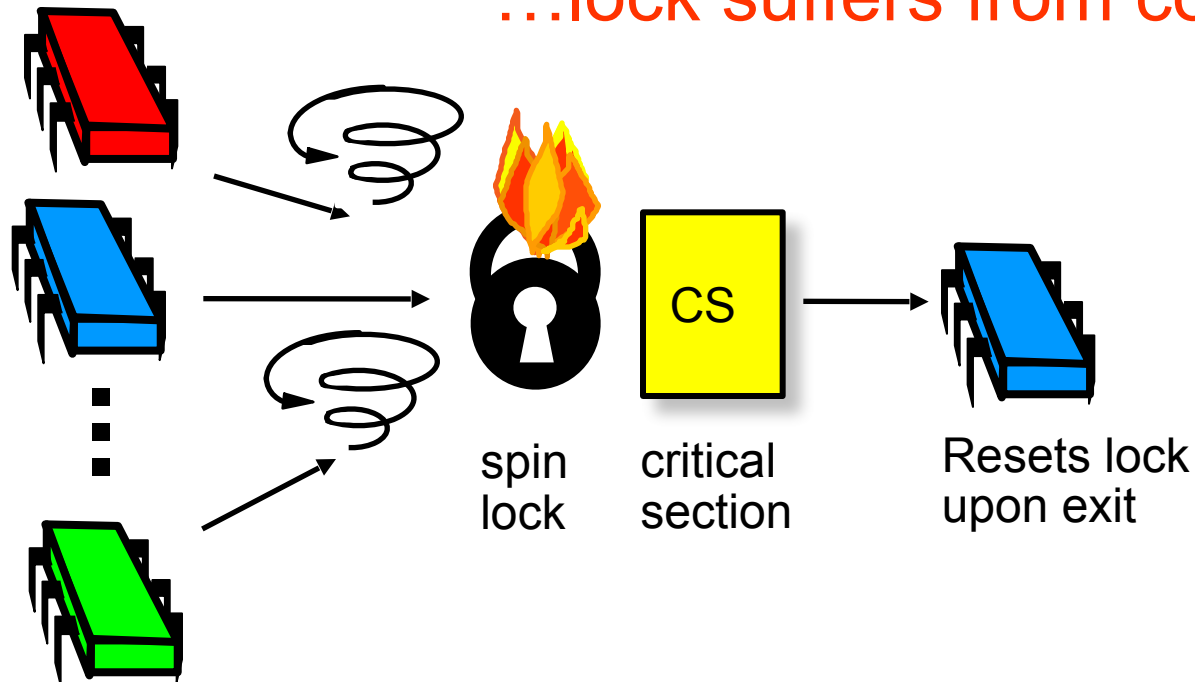
Basic Spin-Lock

...lock suffers from contention



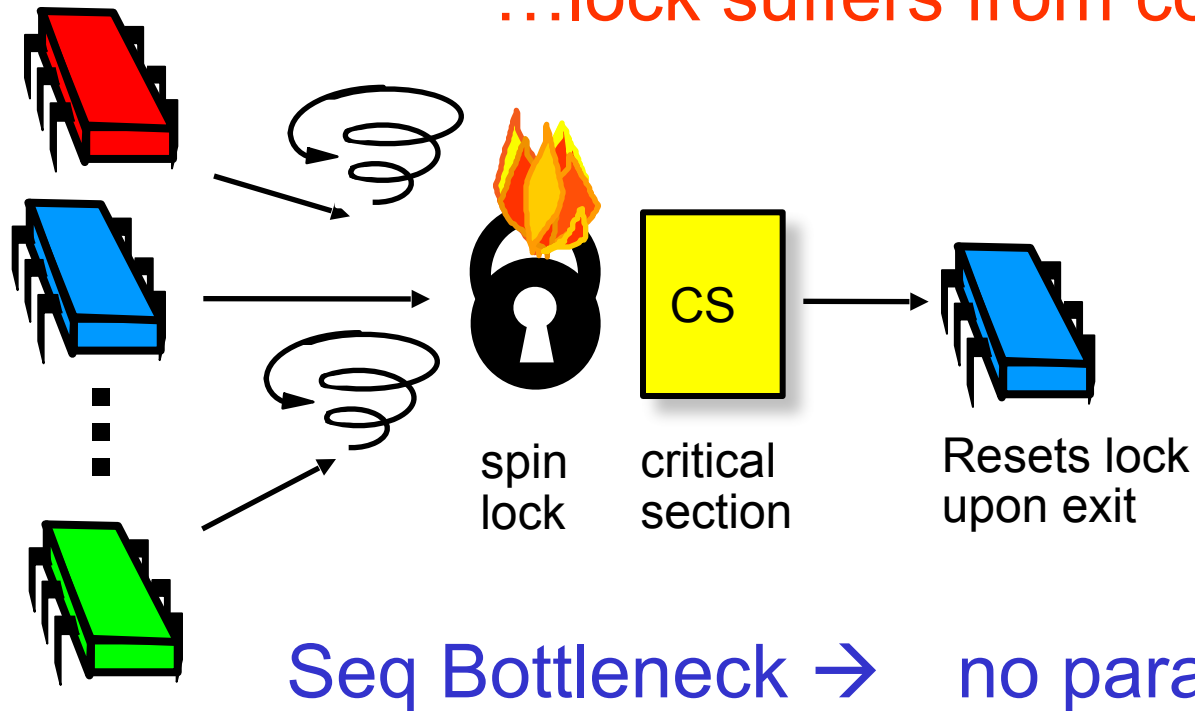
Basic Spin-Lock

...lock suffers from contention



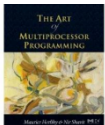
Basic Spin-Lock

...lock suffers from contention



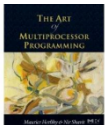
Test-and-Set

- 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”



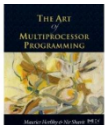
Test-and-Set Locks

- 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

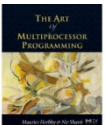
```
public class AtomicBoolean {  
    boolean value;  
  
    public synchronized boolean  
        getAndSet(boolean newValue) {  
        boolean prior = value;  
        value = newValue;  
        return prior;  
    }  
}
```



Test-and-Set

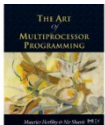
```
public class AtomicBoolean {  
    boolean value;  
  
    public synchronized boolean  
    getAndSet(boolean newValue) {  
        boolean prior = value;  
        value = newValue;  
        return prior;  
    }  
}
```

Swap old and new
values



Test-and-Set

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

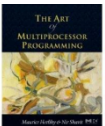


Test-and-Set

```
AtomicBoolean lock  
= new AtomicBoolean(false)
```

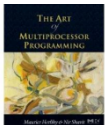
```
boolean prior = lock.getAndSet(true)
```

Swapping in **true** is called “test-and-set” or TAS



Test-and-set Lock

```
class TASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
  
    void unlock() {  
        state.set(false);  
    }  
}
```



Test-and-set Lock

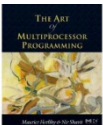
```
class TASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
  
    void unlock() {  
        state.  
    }  
}
```

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



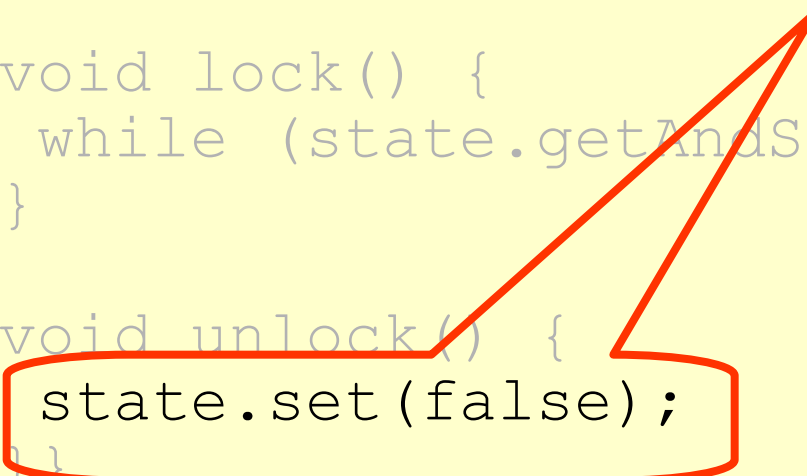
Test-and-set Lock

```
class TAIL {
    AtomicBoolean state;
    new AtomicBoolean(state);

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

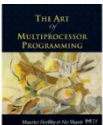
    void unlock() {
        state.set(false);
    }
}
```

Release lock by resetting state to false



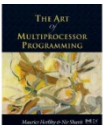
Space Complexity

- TAS spin-lock has small “footprint”
- N thread spin-lock uses $O(1)$ space
- As opposed to $O(n)$ Peterson/Bakery

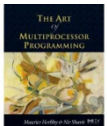
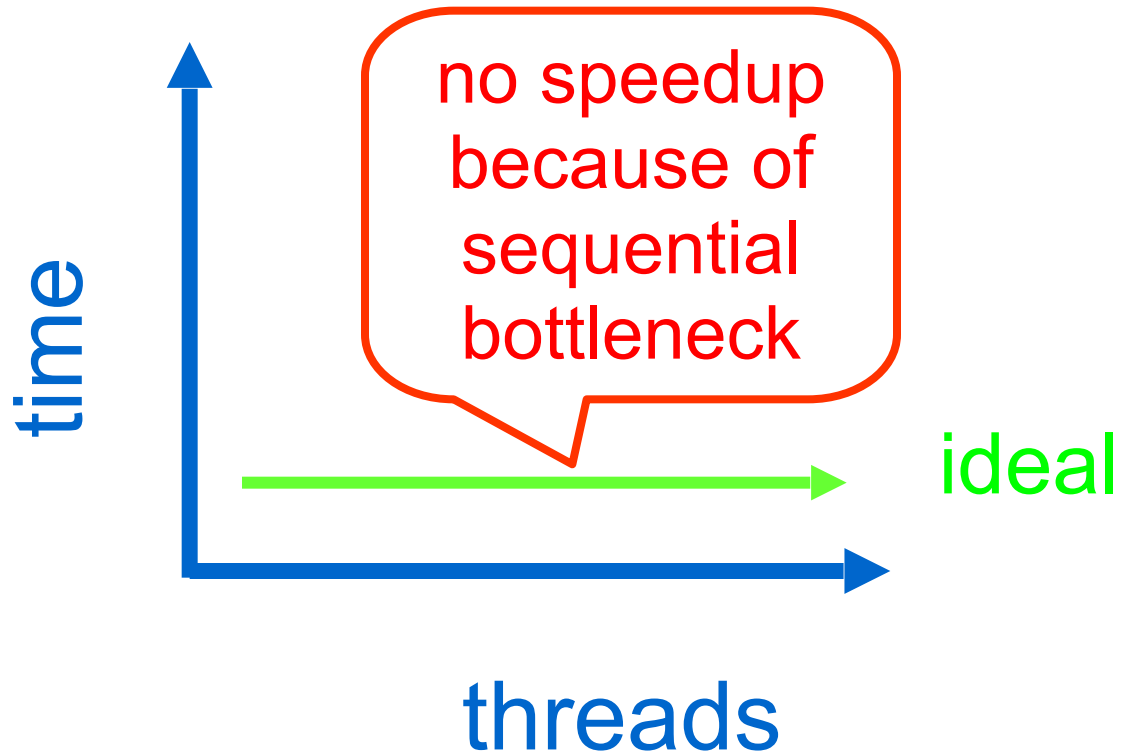


Performance

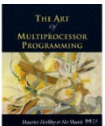
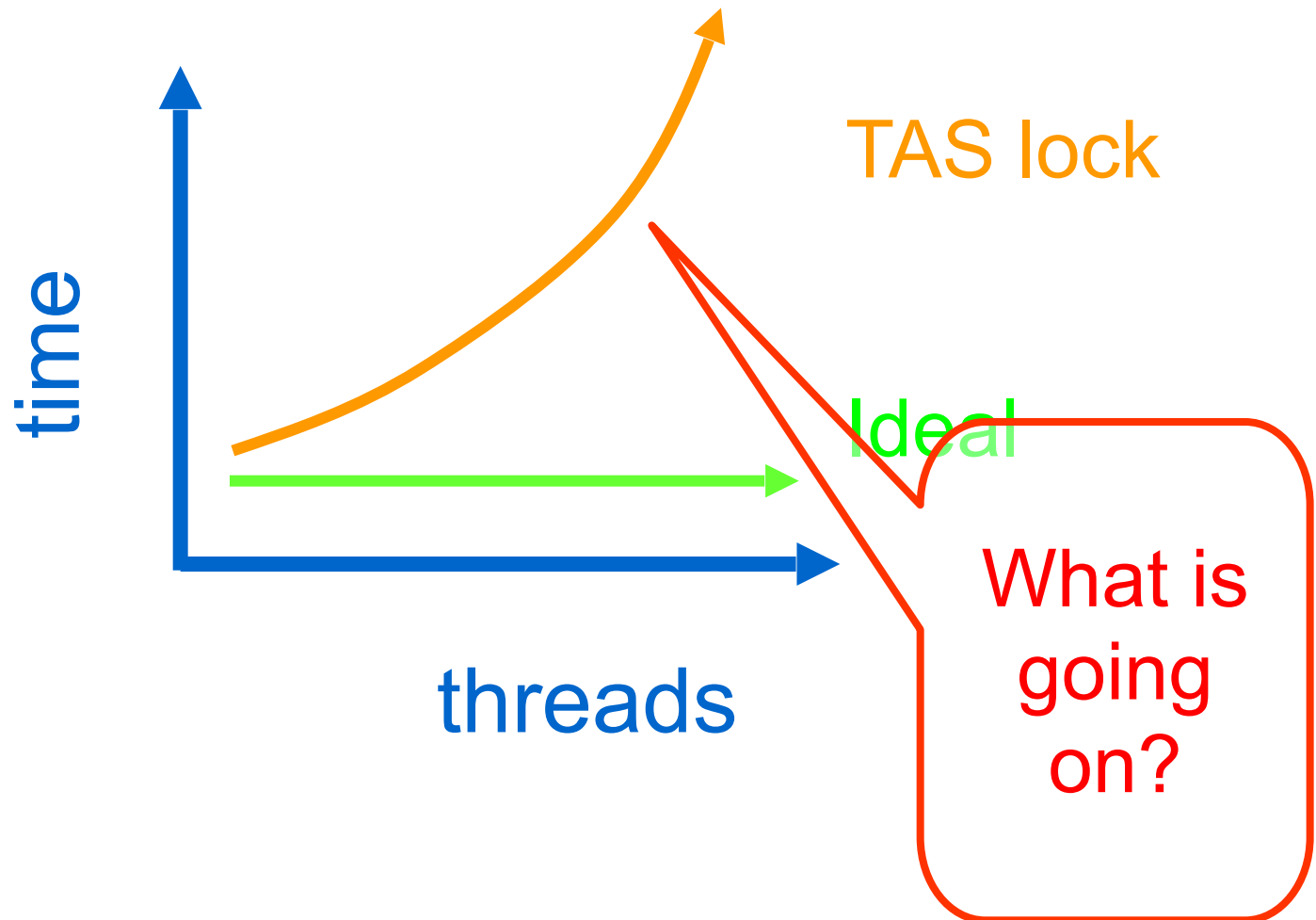
- Experiment
 - n threads
 - Increment shared counter 1 million times
- How long should it take?
- How long does it take?



Graph

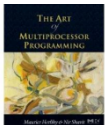


Mystery #1



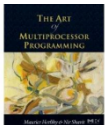
Test-and-Test-and-Set Locks

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



Test-and-test-and-set Lock

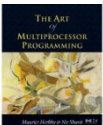
```
class TTASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (true) {  
            while (state.get()) {}  
            if (!state.getAndSet(true))  
                return;  
        }  
    }  
}
```



Test-and-test-and-set Lock

```
class TTASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (true) {  
            while (state.get()) {}  
            if (!state.getAndSet(true))  
                return;  
        }  
    }  
}
```

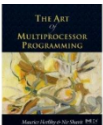
Wait until lock looks free



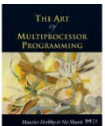
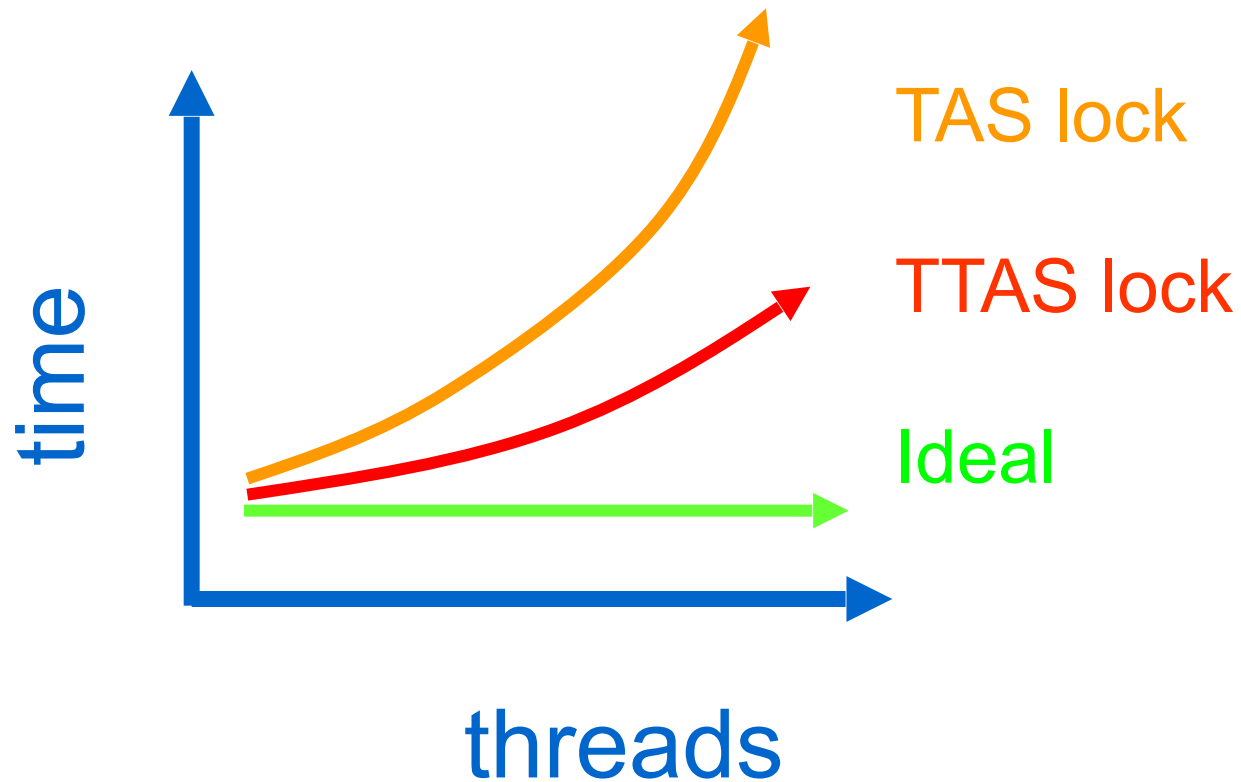
Test-and-test-and-set Lock

```
class TTASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (true) {  
            while (state.get()) {}  
            if (!state.getAndSet(true))  
                return;  
        }  
    }  
}
```

Then try to
acquire it

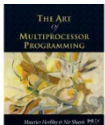


Mystery #2



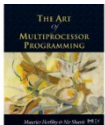
Mystery

- Both
 - TAS and TTAS
 - Do the same thing (in our model)
- Except that
 - TTAS performs much better than TAS
 - Neither approaches ideal

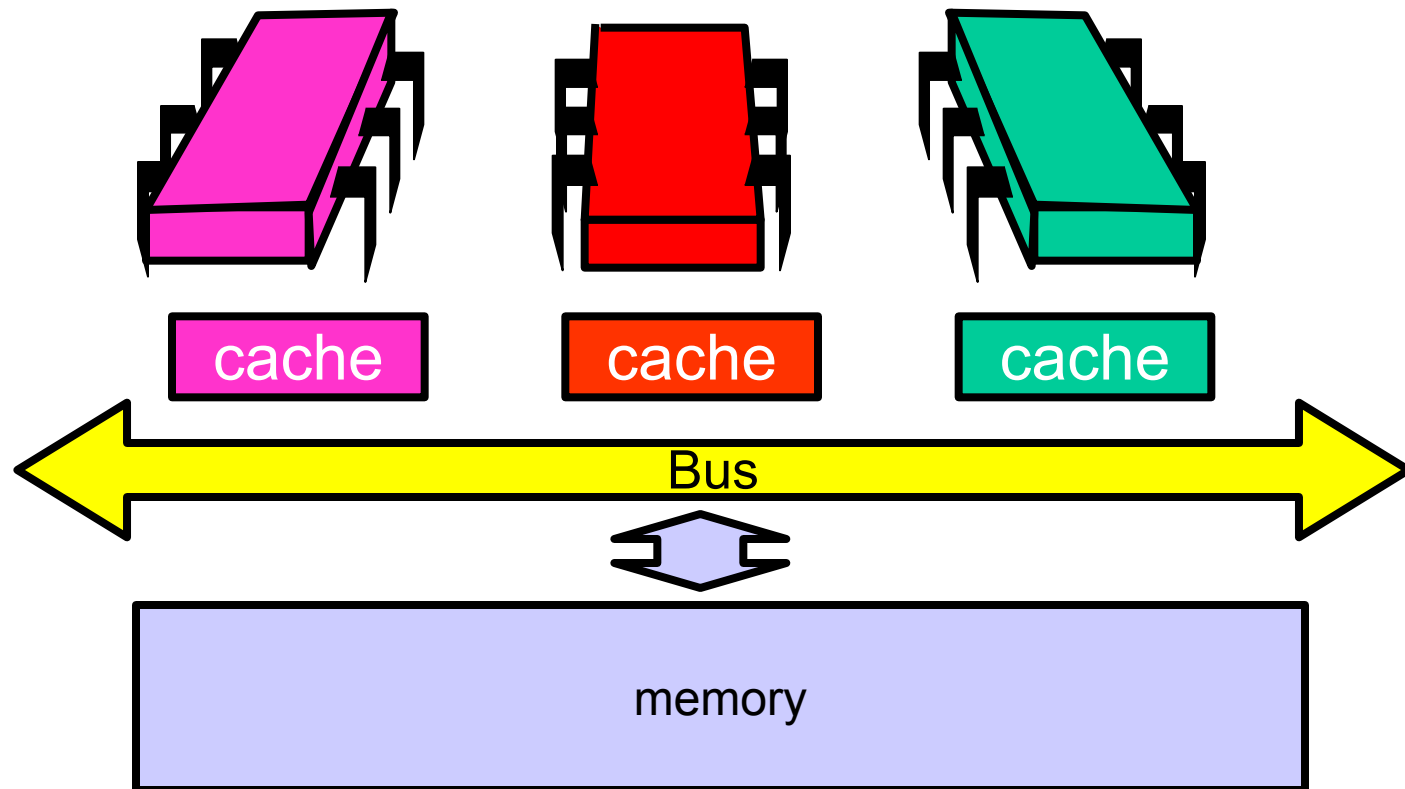


Opinion

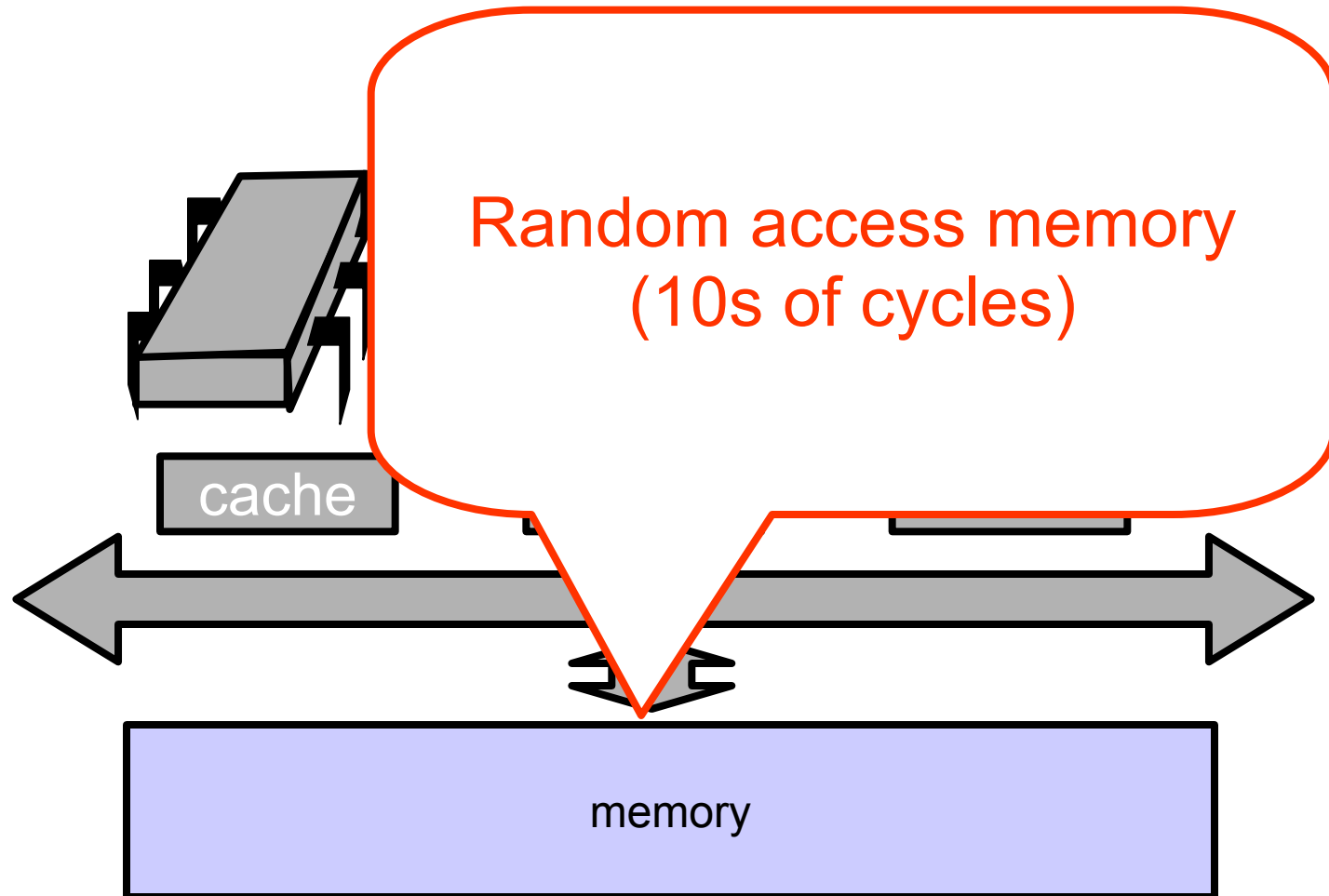
- Our memory abstraction is broken
- TAS & TTAS methods
 - Are provably the same (in our model)
 - Except they aren't (in field tests)
- Need a more detailed model ...



Bus-Based Architectures



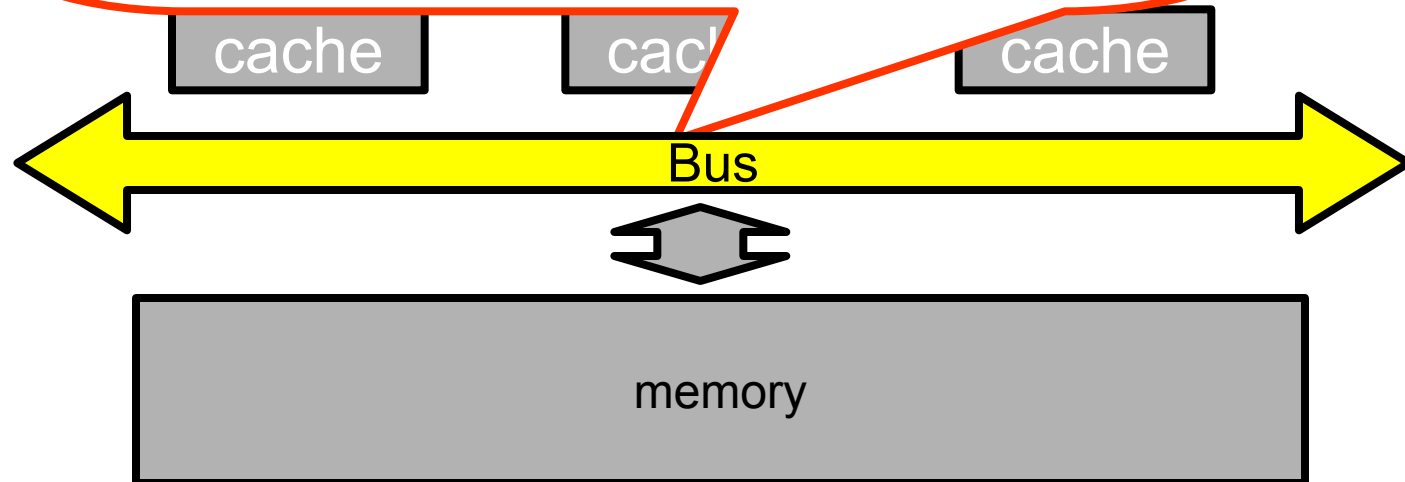
Bus-Based Architectures



Bus-Based Architectures

Shared Bus

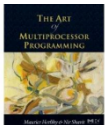
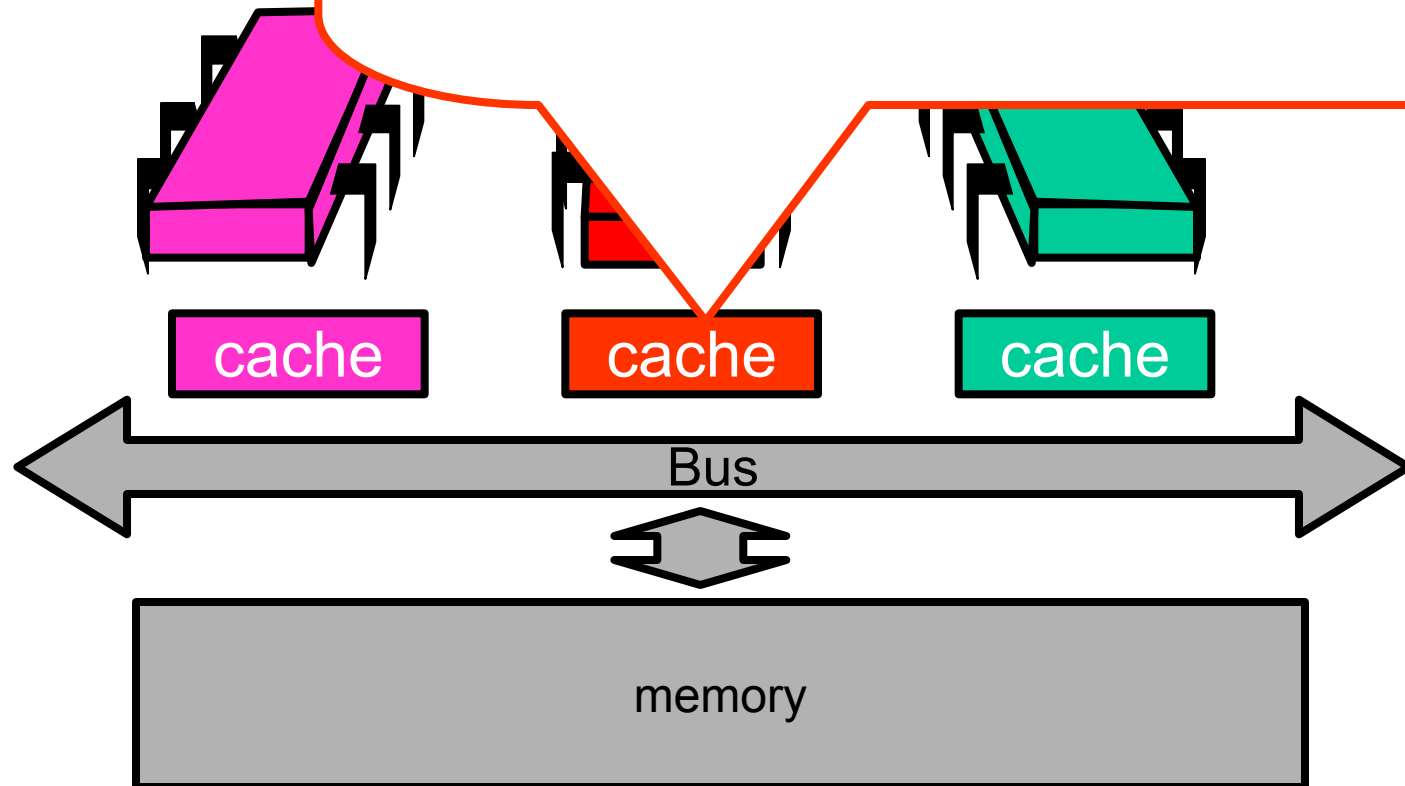
- Broadcast medium
- One broadcaster at a time
- Processors and memory all “snoop”



Bus-

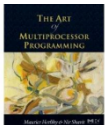
Per-Processor Caches

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



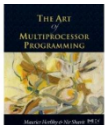
Mutual Exclusion

- What do we want to optimize?
 - Bus bandwidth used by spinning threads
 - Release/Acquire latency
 - Acquire latency for idle lock



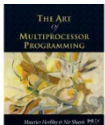
Simple TASLock

- TAS invalidates cache lines
- Spinners
 - Miss in cache
 - Go to bus
- Thread wants to release lock
 - delayed behind spinners

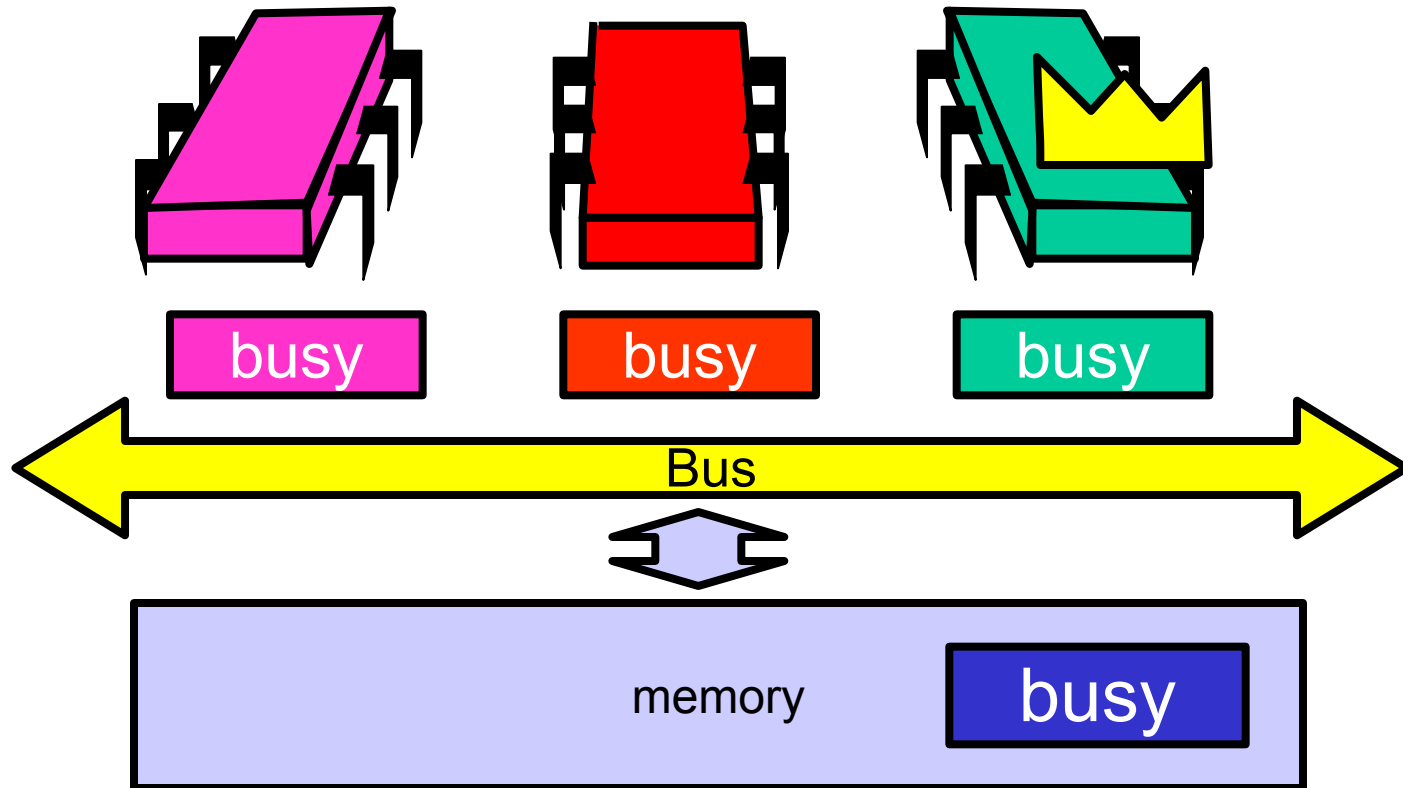


Test-and-test-and-set

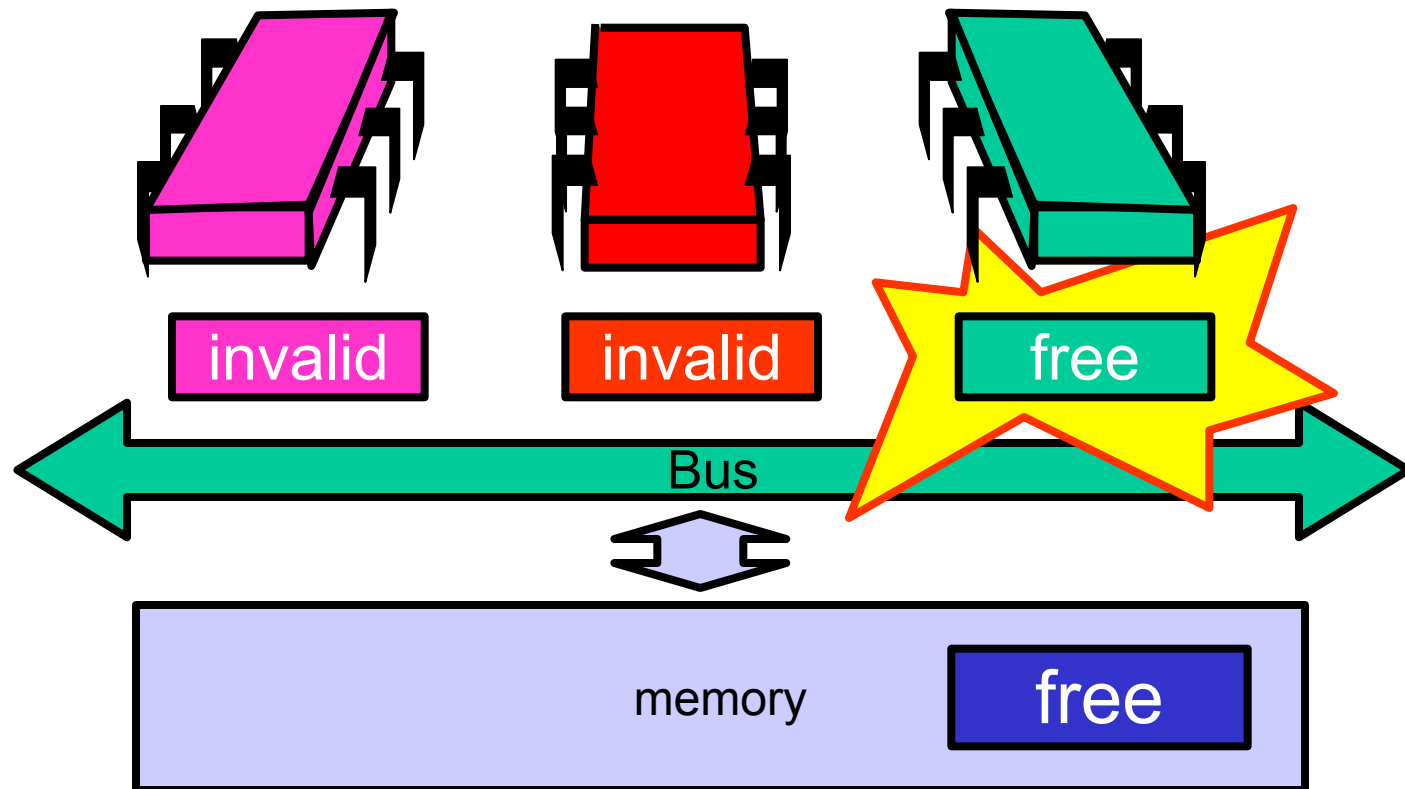
- Wait until lock “looks” free
 - Spin on local cache
 - No bus use while lock busy
- Problem: when lock is released
 - Invalidation storm ...



Local Spinning while Lock is Busy

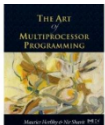
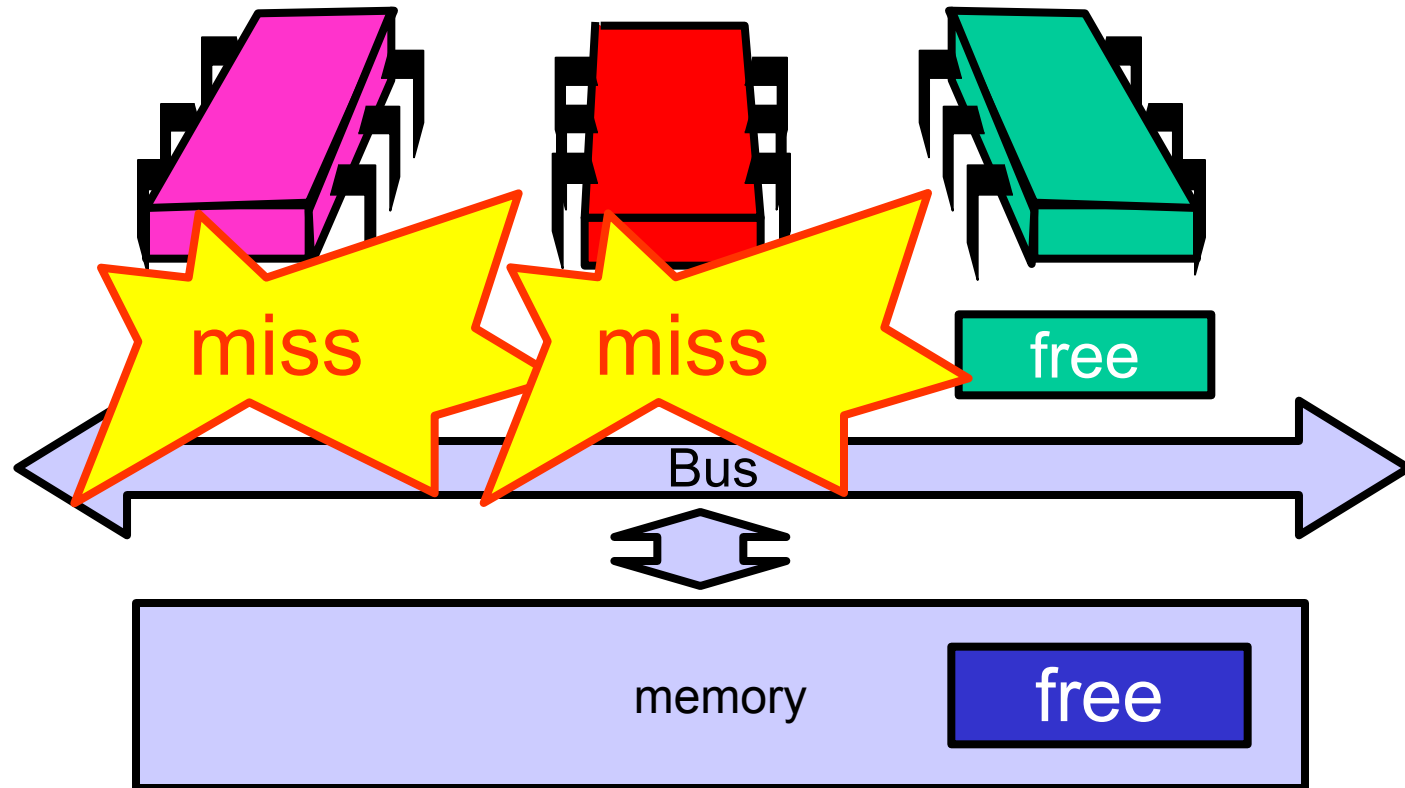


On Release



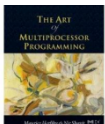
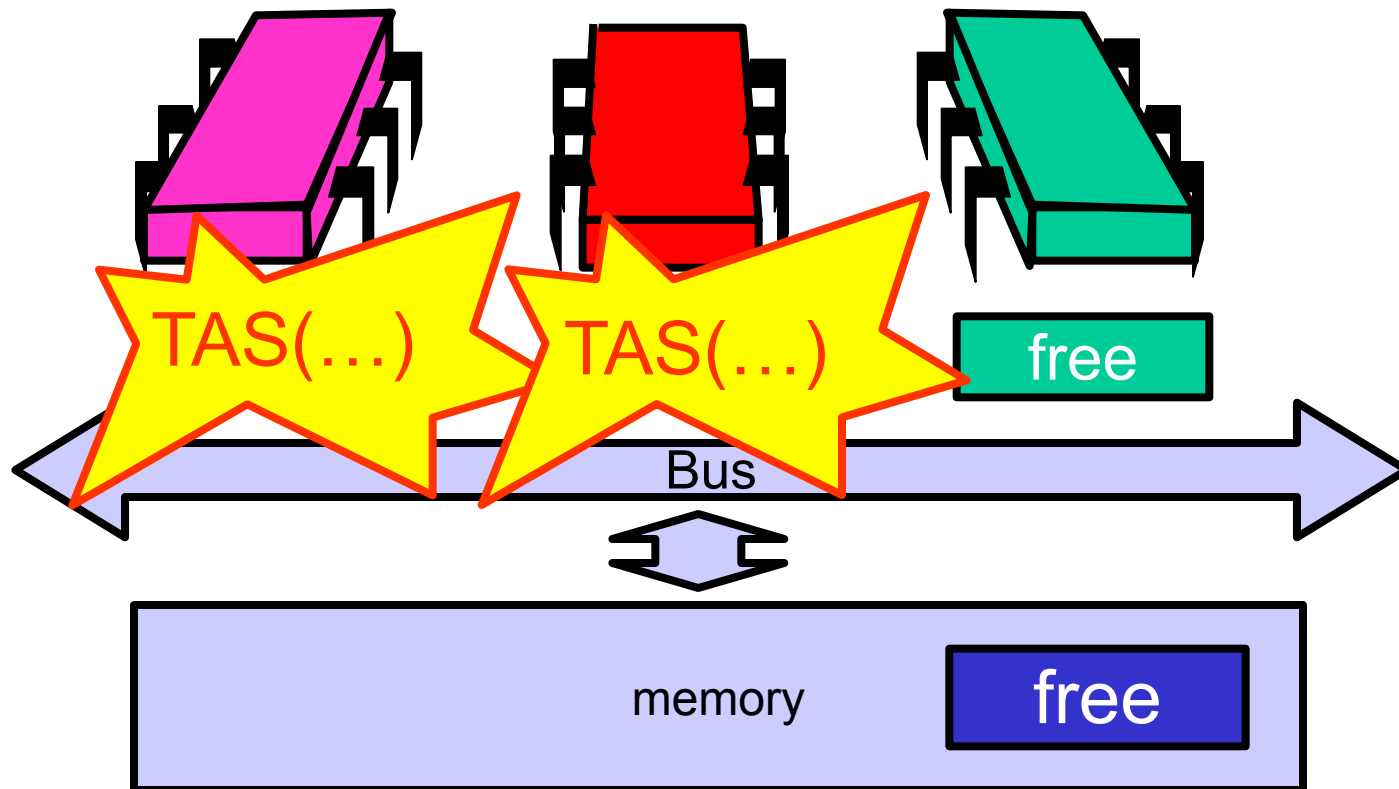
On Release

Everyone misses,
rereads



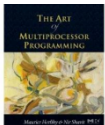
On Release

Everyone tries TAS

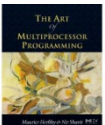
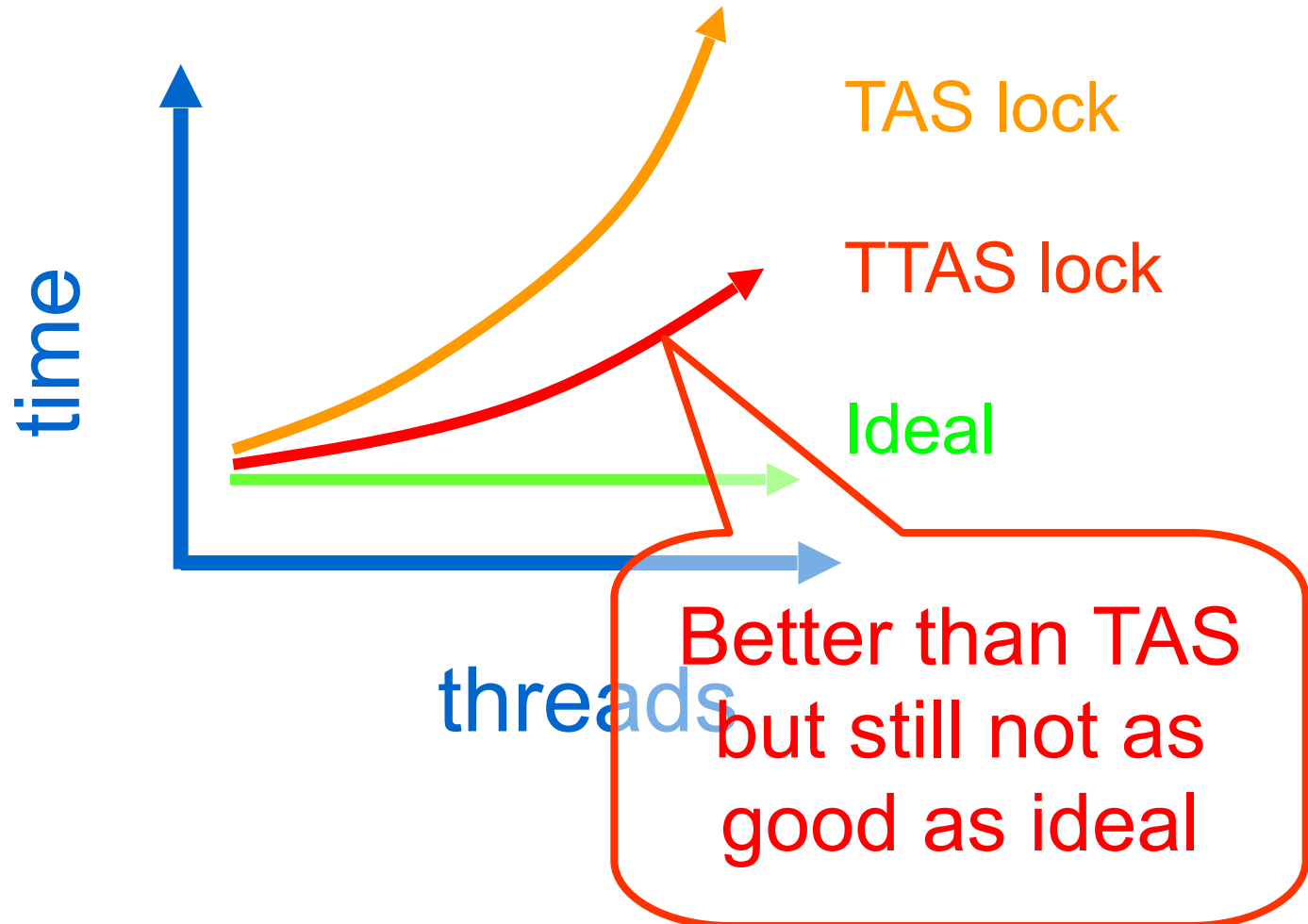


Problems

- Everyone misses
 - Reads satisfied sequentially
- Everyone does TAS
 - Invalidates others' caches

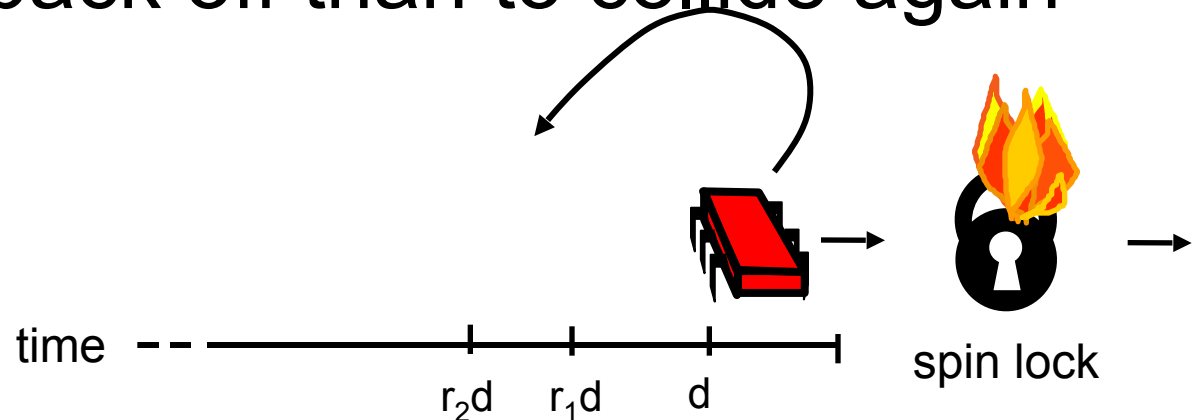


Mystery Explained

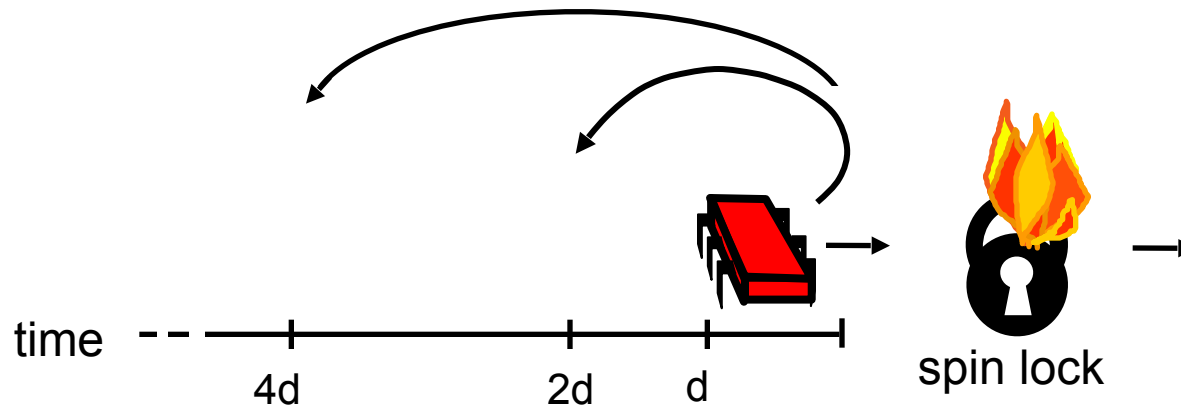


Solution: Introduce Delay

- If the lock looks free
 - But I fail to get it
- There must be contention
 - Better to back off than to collide again



Dynamic Example: Exponential Backoff

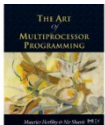


If I fail to get lock

- wait random duration before retry
- Each subsequent failure doubles expected wait

Exponential Backoff Lock

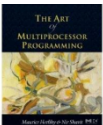
```
public class Backoff implements lock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)  
                delay = 2 * delay;  
        }  
    }  
}
```



Exponential Backoff Lock

```
public class Backoff implements lock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)  
                delay = 2 * delay;  
        }  
    }  
}
```

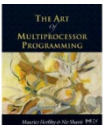
Fix minimum delay



Exponential Backoff Lock

```
public class Backoff implements lock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)  
                delay = 2 * delay;  
        }  
    }  
}
```

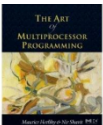
Wait until lock looks free



Exponential Backoff Lock

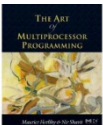
```
public class Backoff implements lock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)  
                delay = 2 * delay;  
        }  
    }  
}
```

If we win, return



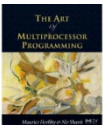
Exponential Backoff Lock

```
public class Backoff implements lock {  
    public Back off for random duration  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)  
                delay = 2 * delay;  
        }  
    }  
}
```

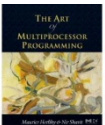
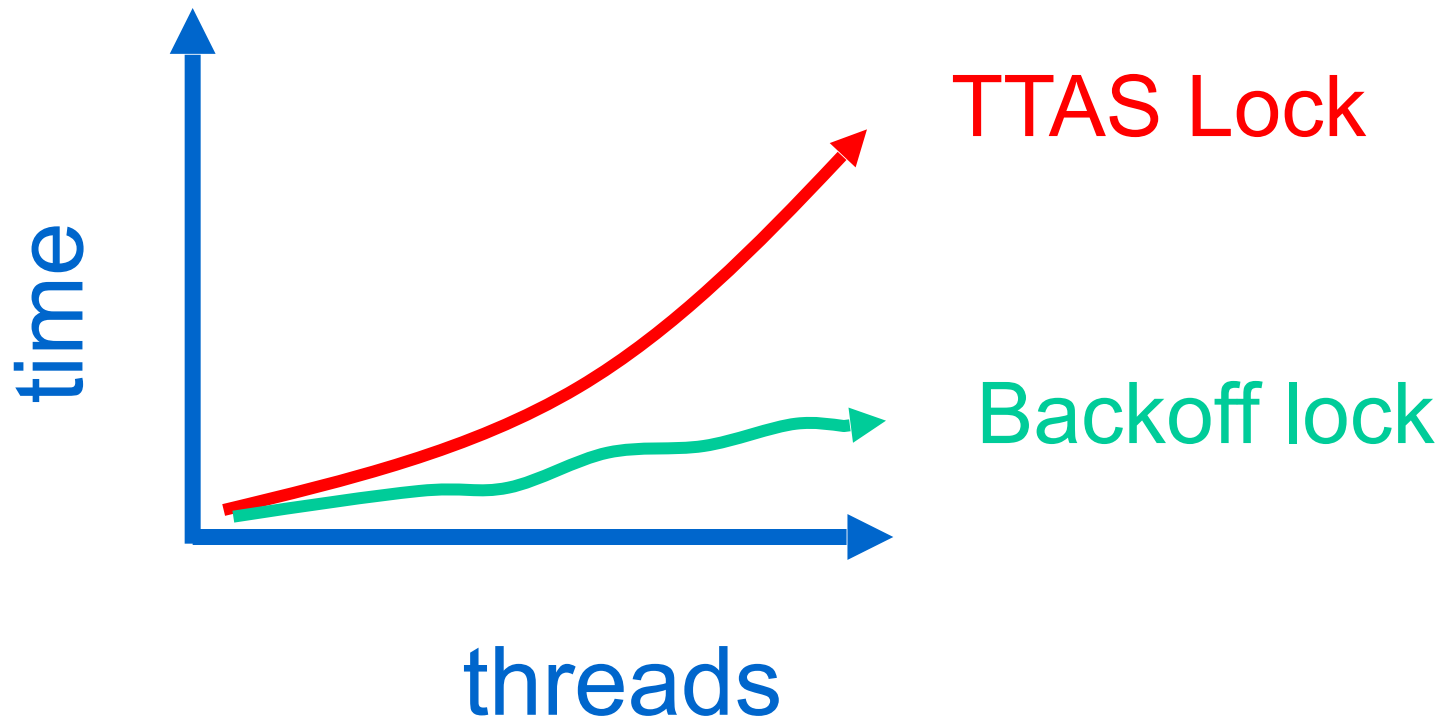


Exponential Backoff Lock

```
public class Backoff implements Lock {  
    Double max delay, within reason  
    int delay = MIN_DELAY;  
    while (true) {  
        while (state.get()) {}  
        if (!lock.getAndSet(true))  
            return;  
        sleep(random() % delay);  
        if (delay < MAX_DELAY)  
            delay = 2 * delay;  
    }  
}
```

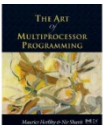


Spin-Waiting Overhead



Backoff: Other Issues

- Good
 - Easy to implement
 - Beats TTAS lock
- Bad
 - Must choose parameters carefully
 - Not portable across platforms



Parallel Performance

Performance Theory

In this course, the main goal for parallelization is to improve performance. But what does “performance” mean? Usually it’s one of the following

- Reducing the **latency**, which is the total time for a program/task to compute a single result.
- Increasing **throughput**, which is the rate at which results are computed.
- Reducing the power consumption of a computation
- Also the distinct of reducing costs or adding resources to meet a deadline.

All of these are valid interpretations of “performance”.

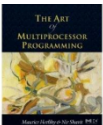
Speedup

- One important metric related to performance and parallelism is **speedup**
 - A ratio between the latency of solving a task with one processing unit versus solving the same problem with multiple processing units in parallel.
 - **Linear speedup** - a algorithm runs P times faster on P processors
 - Very rare in practice due to the extra work disturbing tasks to processors and coordinating them. (Amdahl's Law)

Amdahl's Law

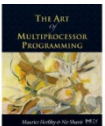
- Limit on speedup - Amdahl's law sometimes called strong scalability which considers speedup as n -threads vary the problem size stays the same.

$$\text{Speedup} = \frac{\text{1-thread execution time}}{\text{\textit{n}-thread execution time}}$$



Amdahl's Law

$$\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}$$

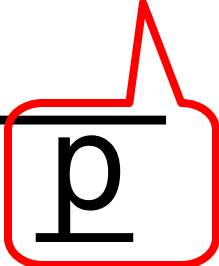


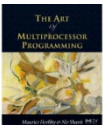
Amdahl's Law

Speedup=

$$\frac{1}{1 - p + \frac{p}{n}}$$

Parallel fraction





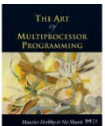
Amdahl's Law

Sequential fraction

Speedup=

Parallel fraction

$$\frac{1}{1 - p + \frac{p}{n}}$$



Amdahl's Law

Sequential fraction

Speedup=

Parallel fraction

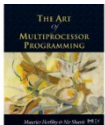
$$\frac{1}{1 - p + \frac{p}{n}}$$

Number of threads

The diagram illustrates Amdahl's Law with the formula $\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}$. Annotations include: 'Sequential fraction' pointing to the '1' in the numerator and the '1 - p' term in the denominator; 'Parallel fraction' pointing to the $\frac{p}{n}$ term in the denominator; and 'Number of threads' pointing to the 'n' in the denominator. Red boxes highlight the '1 - p' and 'p/n' terms.

Example

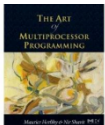
- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?



Example

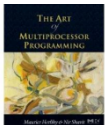
- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 2.17 = \frac{1}{1 - 0.6 + \frac{0.6}{10}}$$



Example

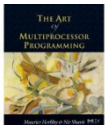
- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?



Example

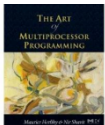
- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 3.57 = \frac{1}{1 - 0.8 + \frac{0.8}{10}}$$



Example

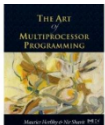
- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?



Example

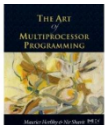
- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 5.26 = \frac{1}{1 - 0.9 + \frac{0.9}{10}}$$



Example

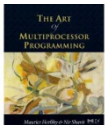
- Ten processors
- 99% concurrent, 1% sequential
- How close to 10-fold speedup?



Example

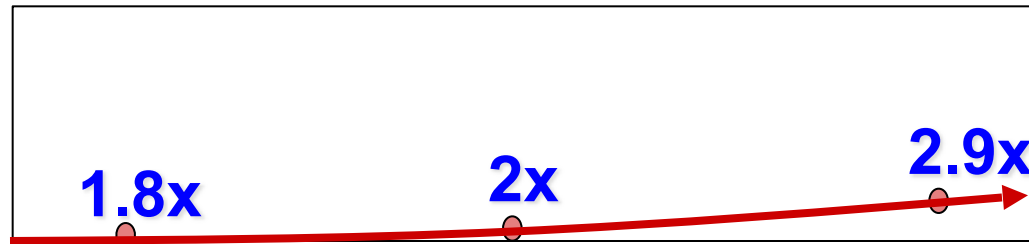
- Ten processors
- 99% concurrent, 1% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}}$$

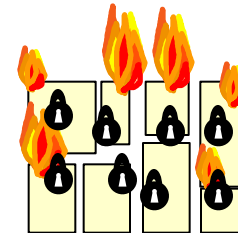
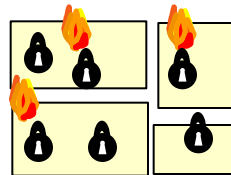
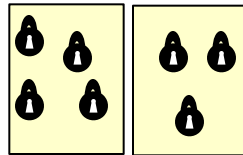


Back to Real-World Multicore Scaling

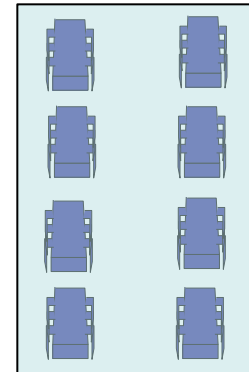
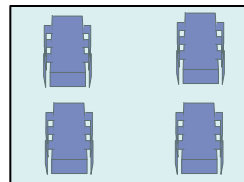
Speedup



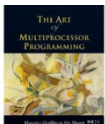
User code



Multicore

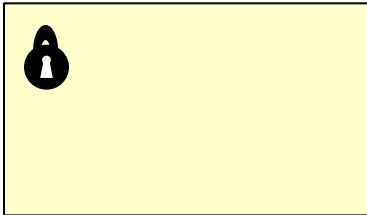


**Not reducing sequential
% of code**

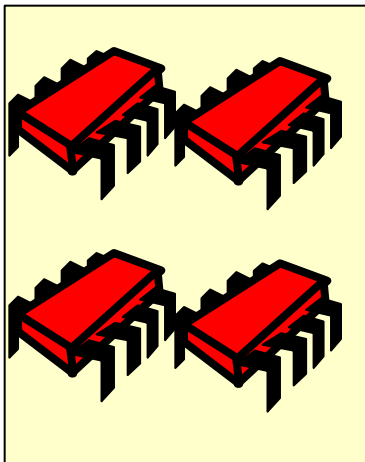


Shared Data Structures

Coarse
Grained

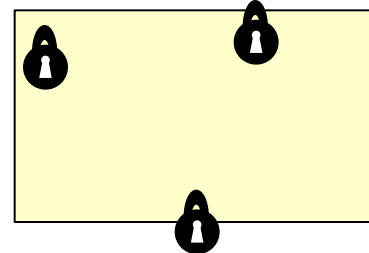


25%
Shared

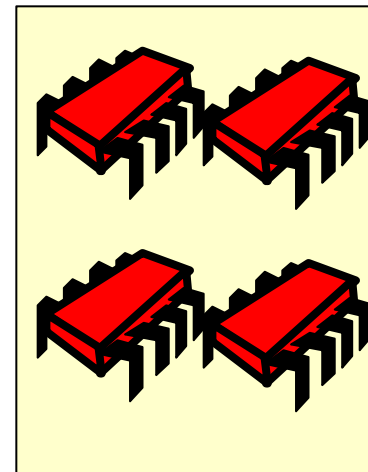


75%
Unshared

Fine
Grained

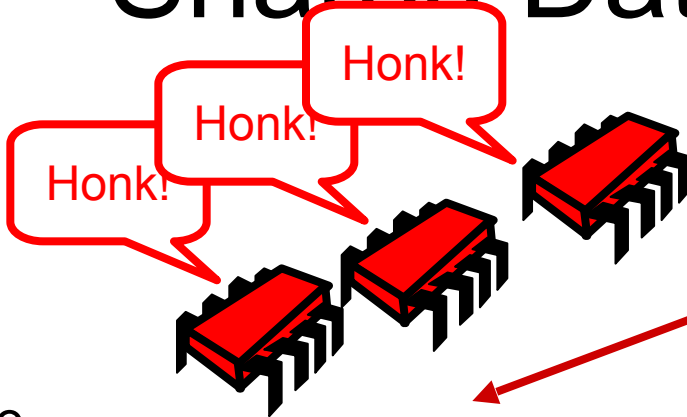


25%
Shared



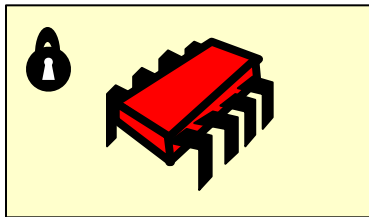
75%
Unshared

Shared Data Structures

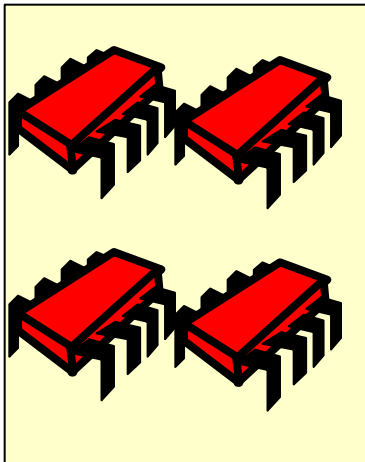


Why only 2.9 speedup

Coarse
Grained

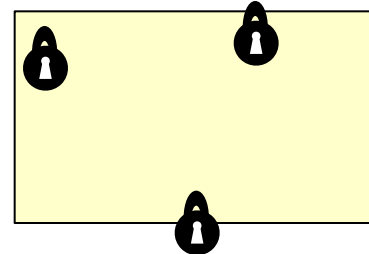


25%
Shared

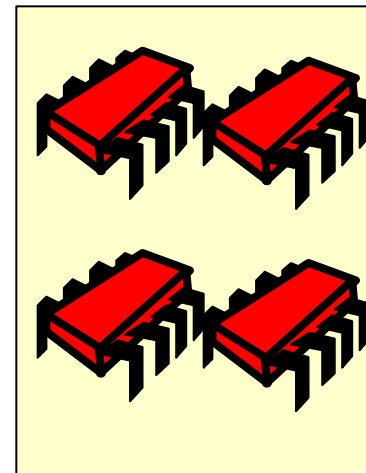


75%
Unshared

Fine
Grained

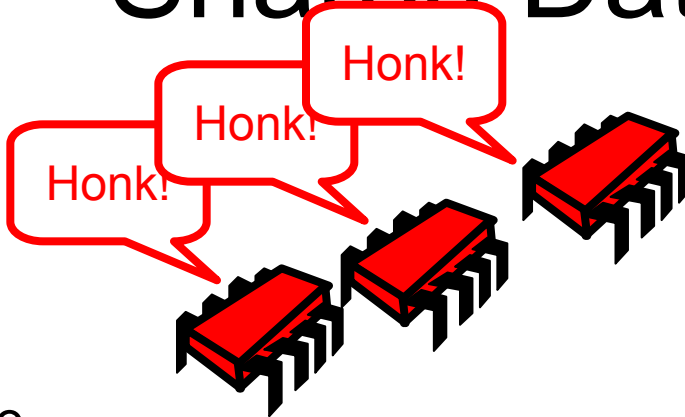


25%
Shared



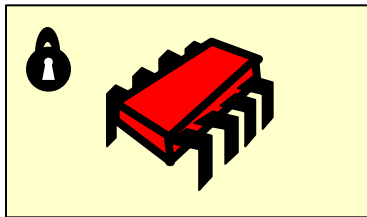
75%
Unshared

Shared Data Structures

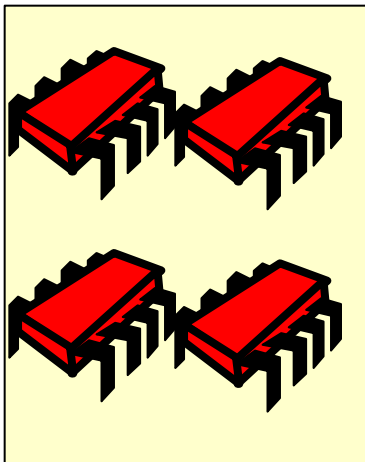


Why fine-grained parallelism matters

Coarse Grained

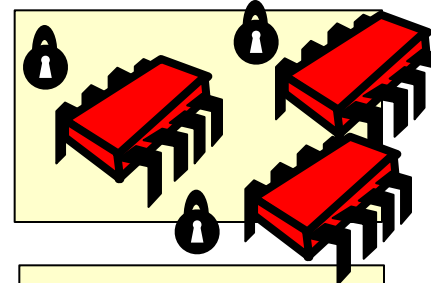


25%
Shared

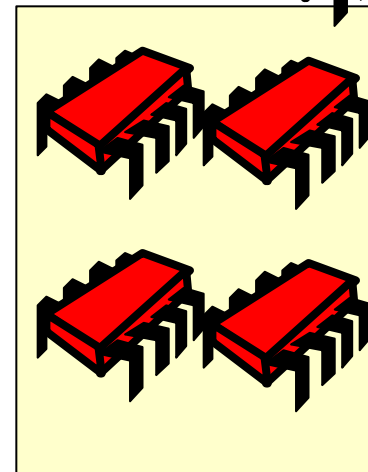


75%
Unshared

Fine Grained



25%
Shared

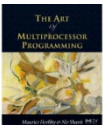


75%
Unshared

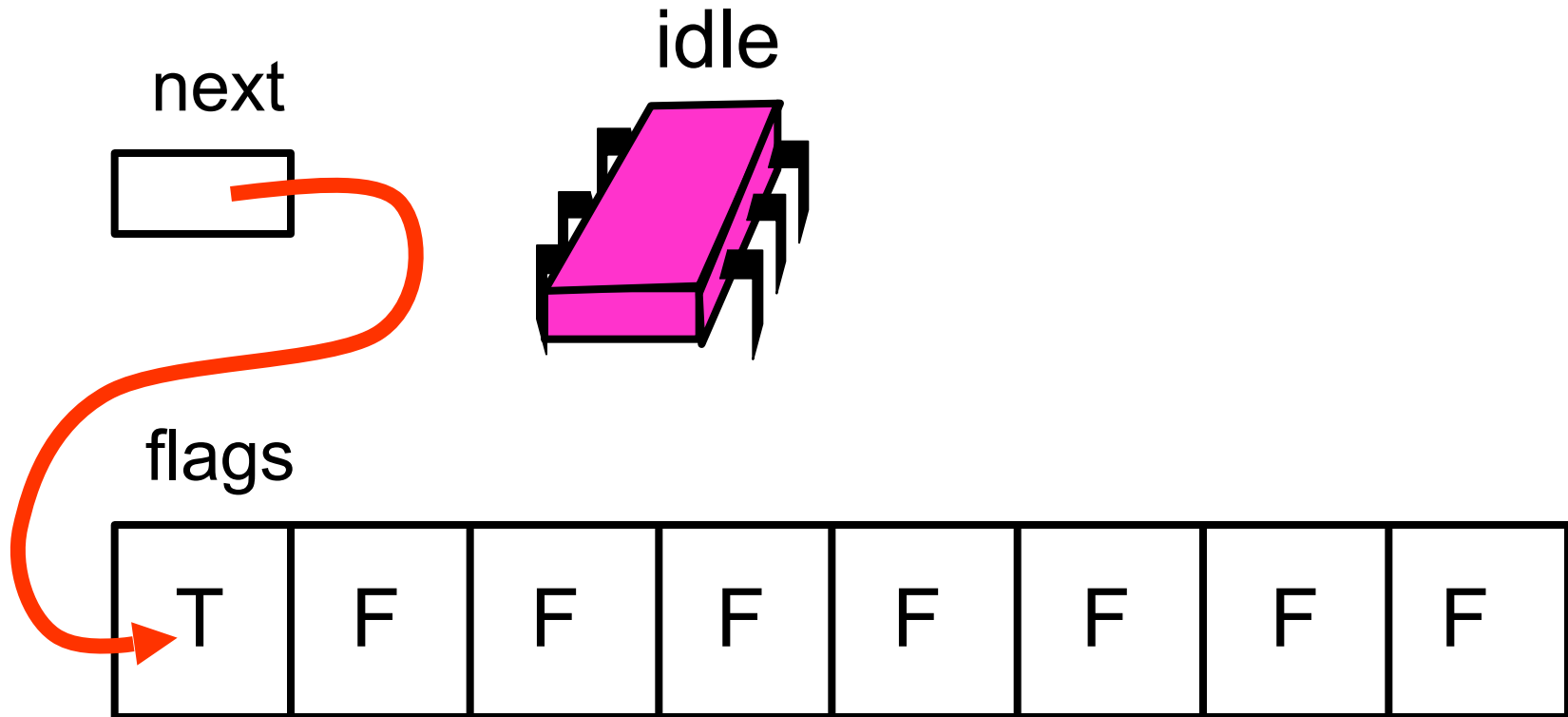
Fairness and Efficient Locks

Idea

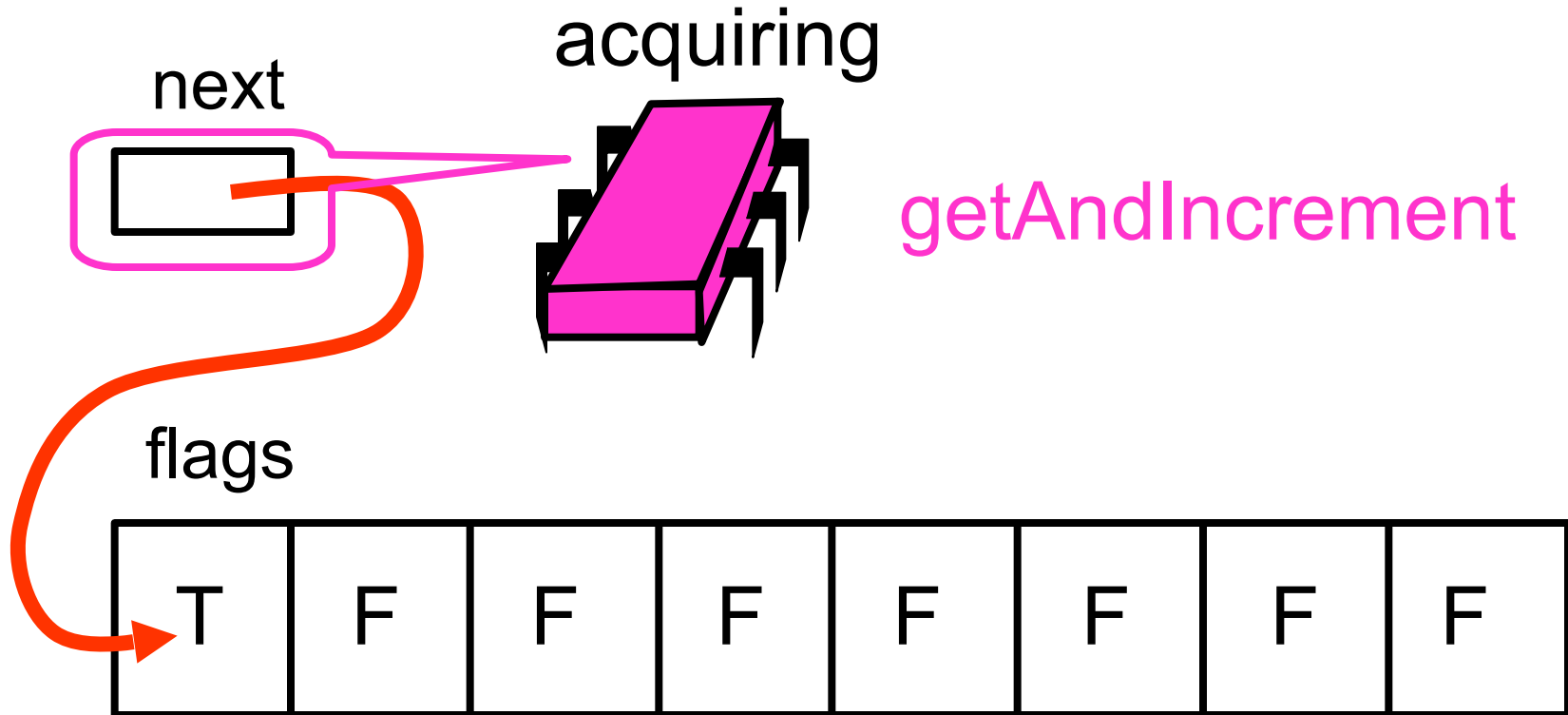
- Avoid useless invalidations
 - By keeping a queue of threads
- Each thread
 - Notifies next in line
 - Without bothering the others
 - Fairness when it comes to who should get lock next based on who arrived first.



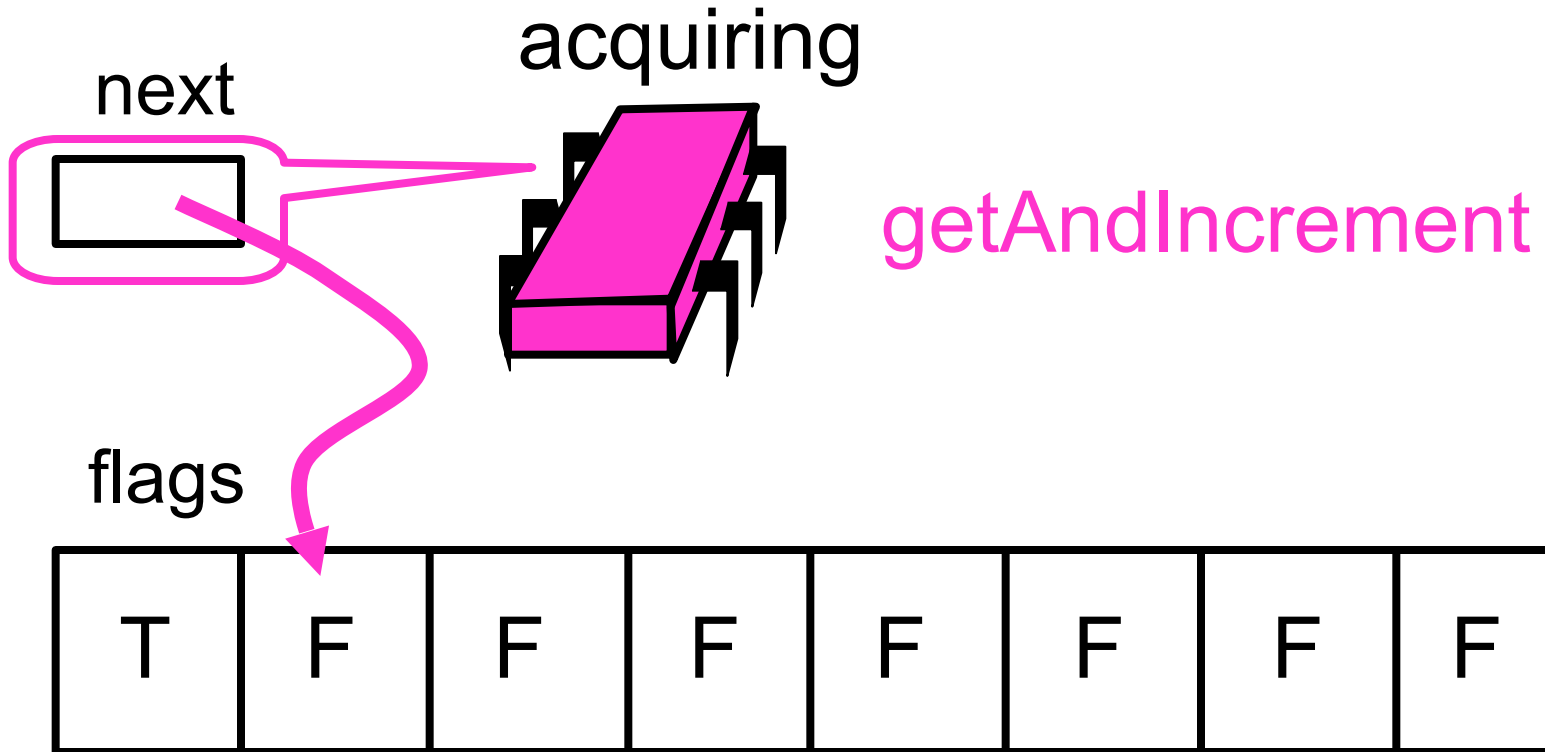
Anderson Queue Lock



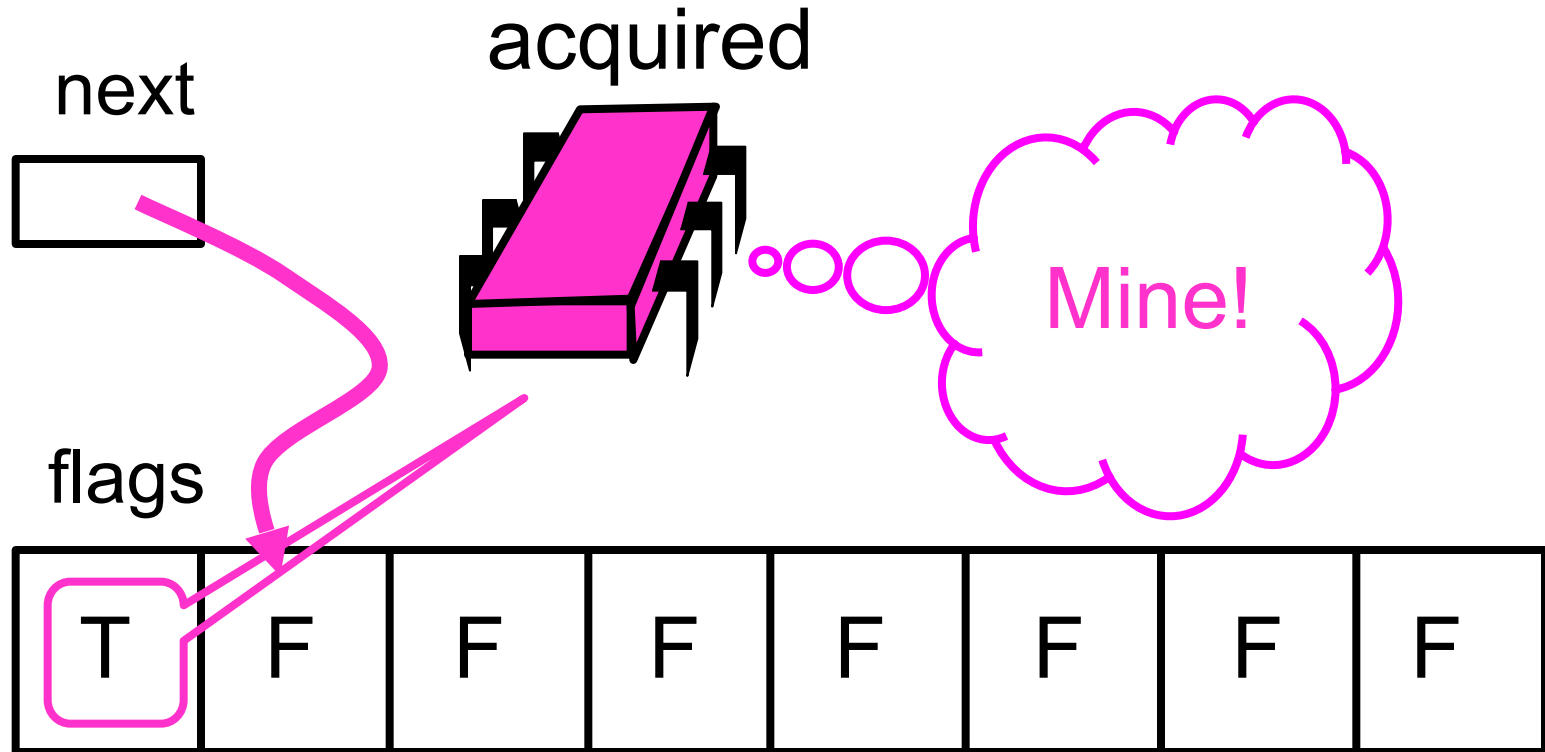
Anderson Queue Lock



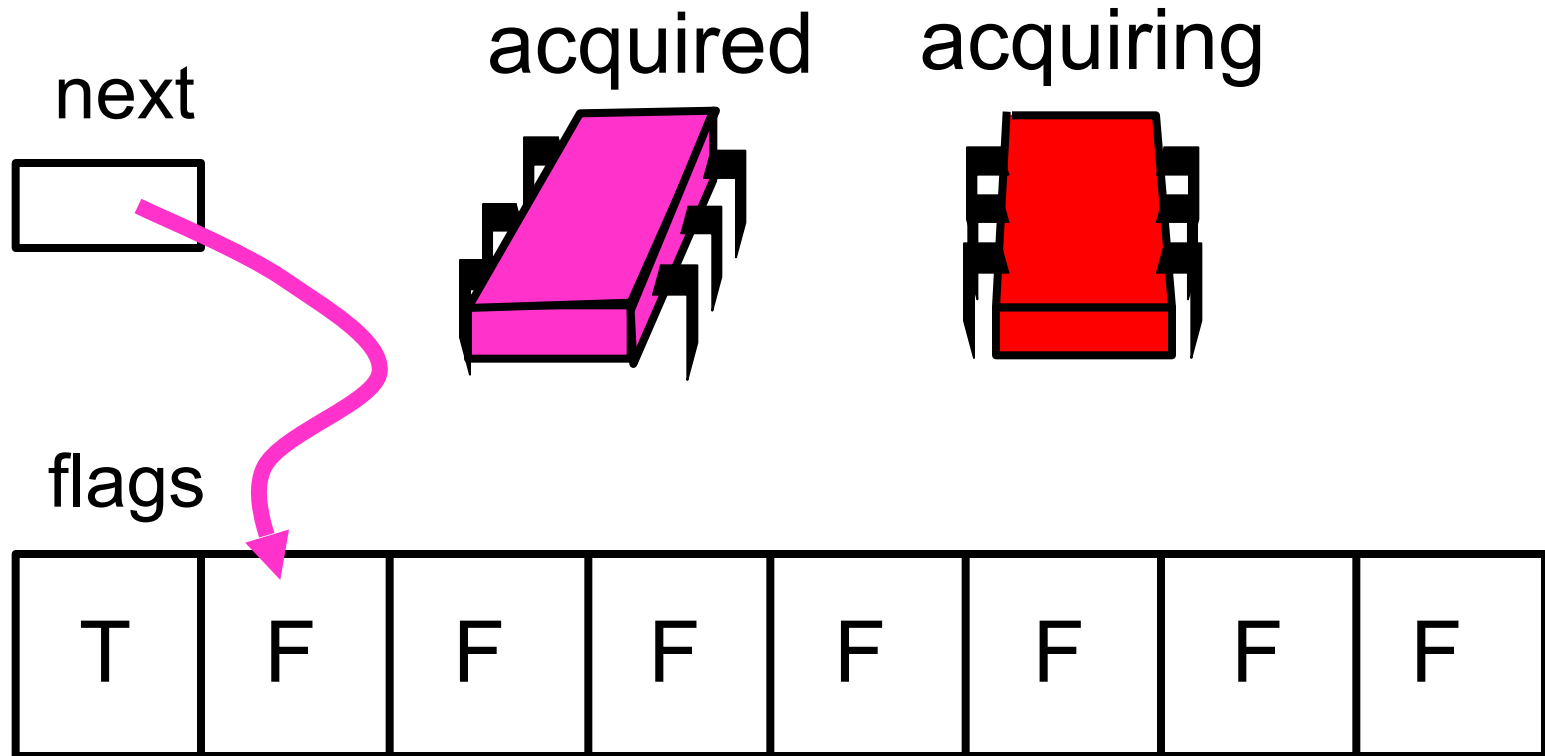
Anderson Queue Lock



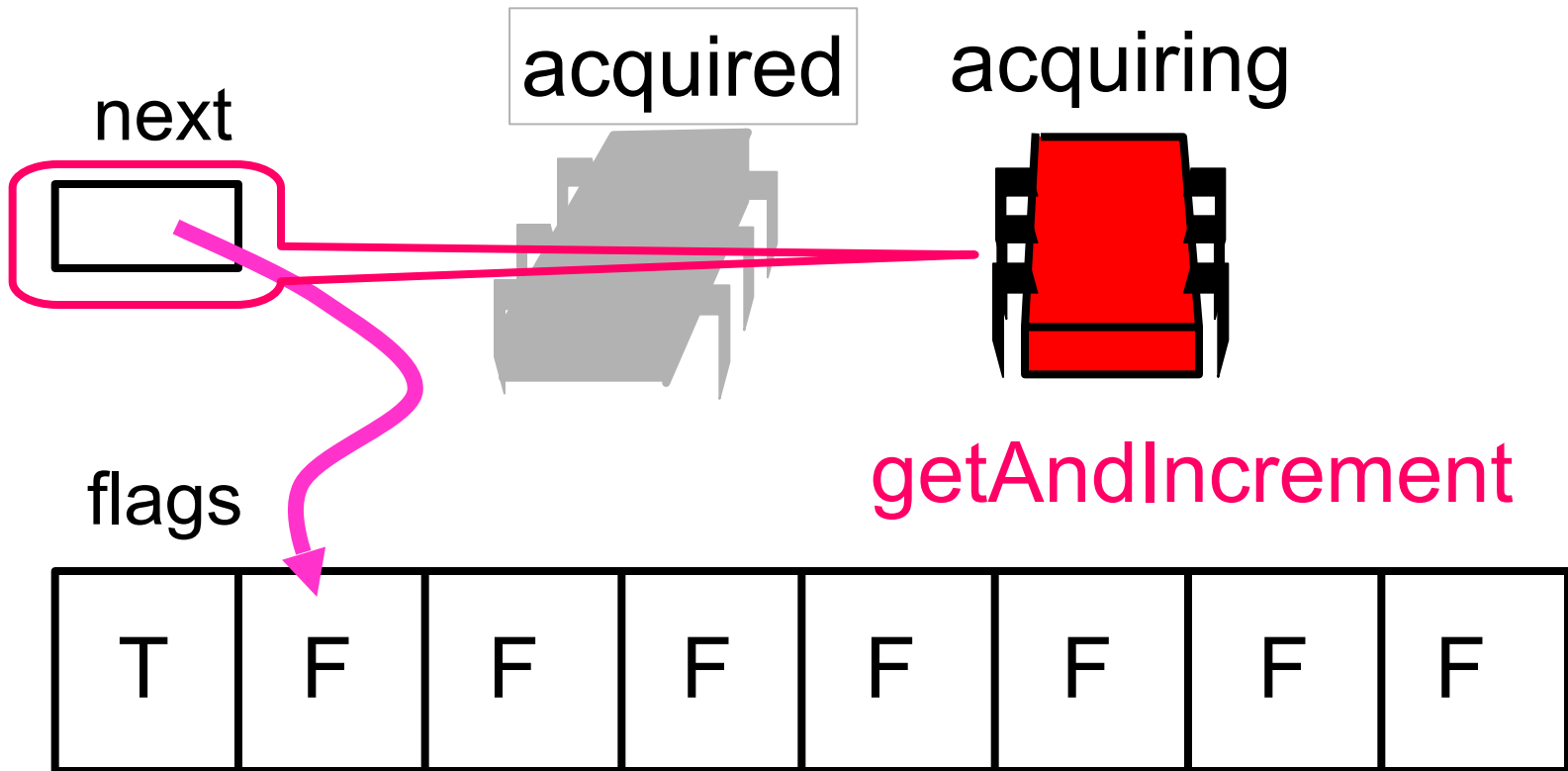
Anderson Queue Lock



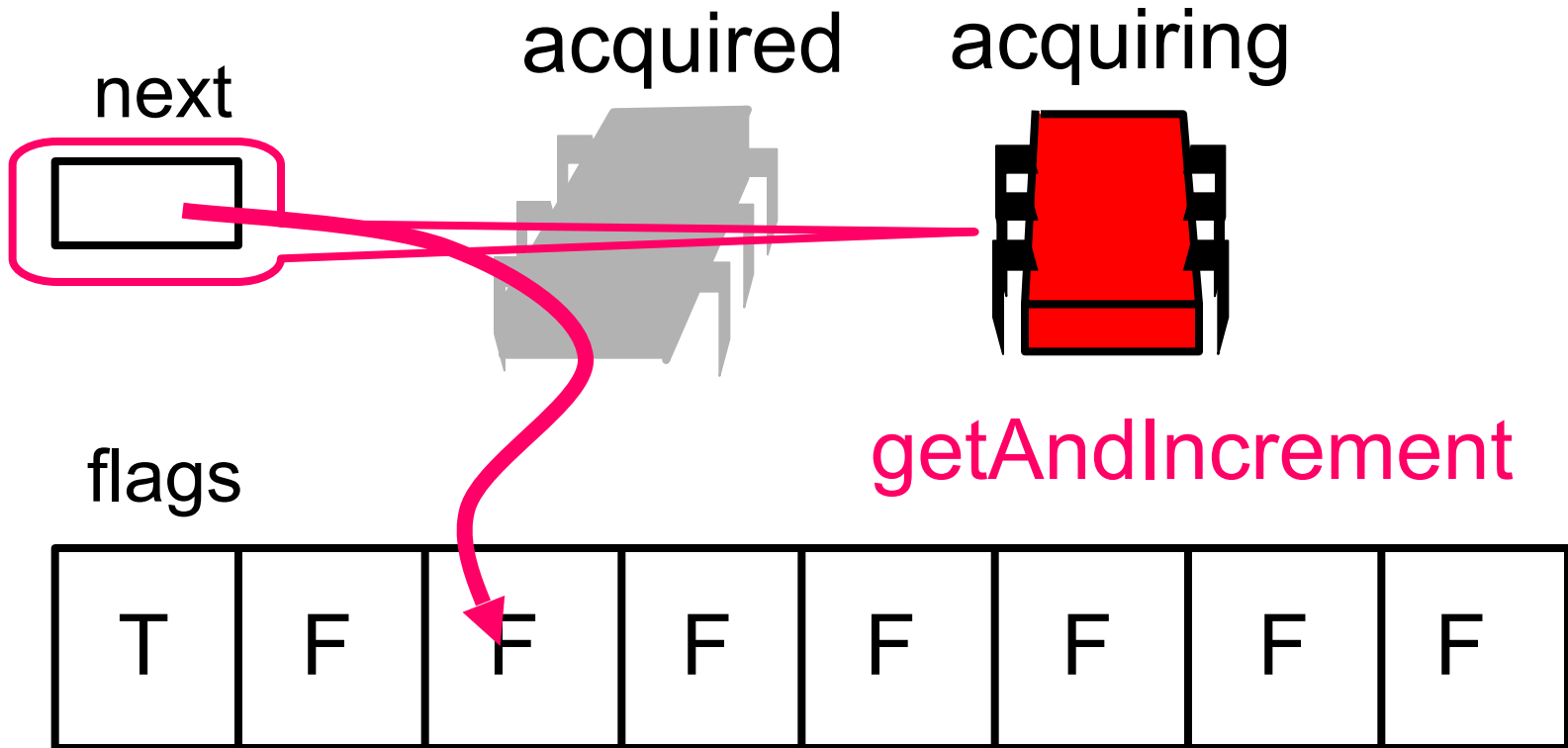
Anderson Queue Lock



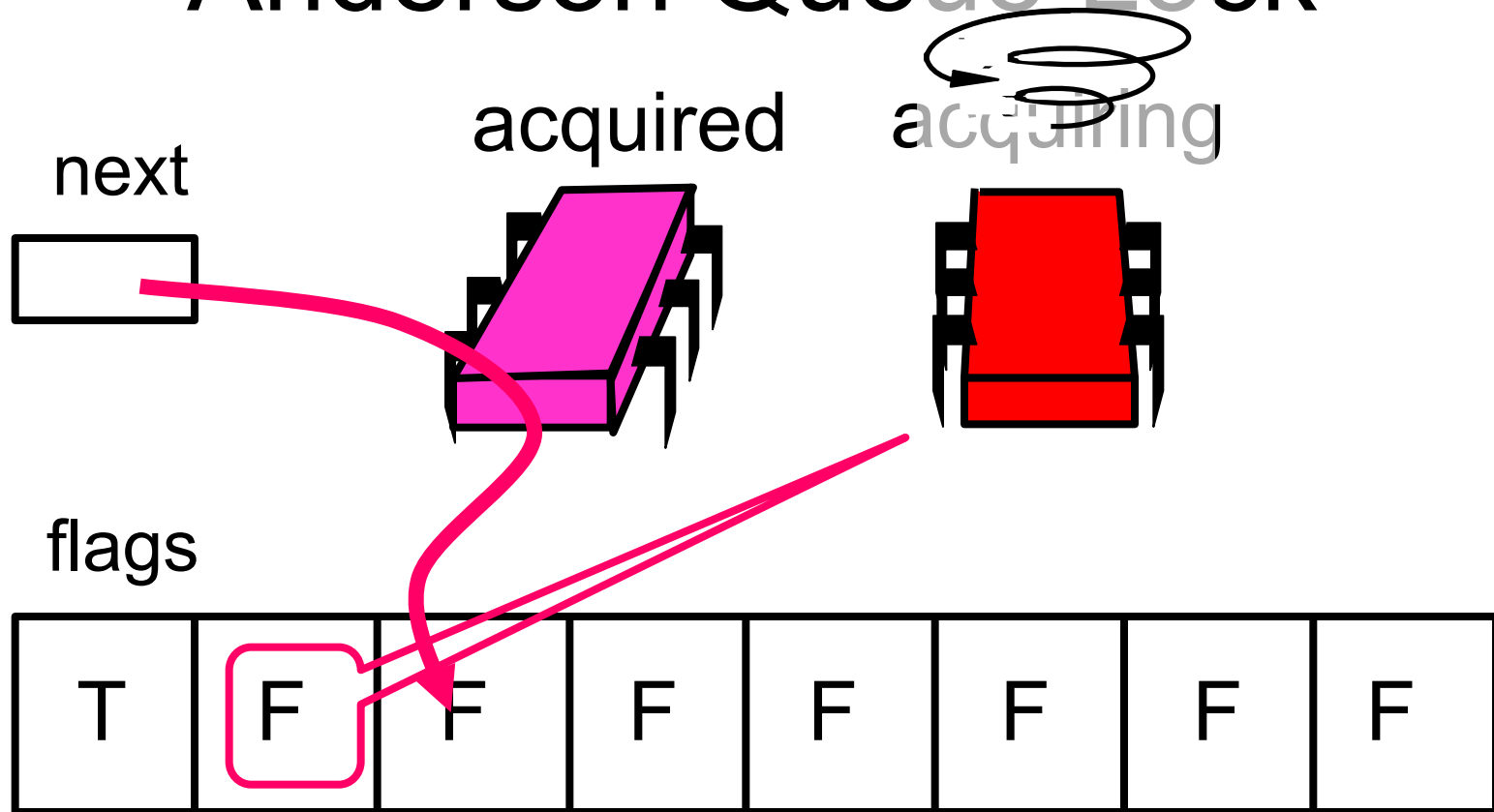
Anderson Queue Lock



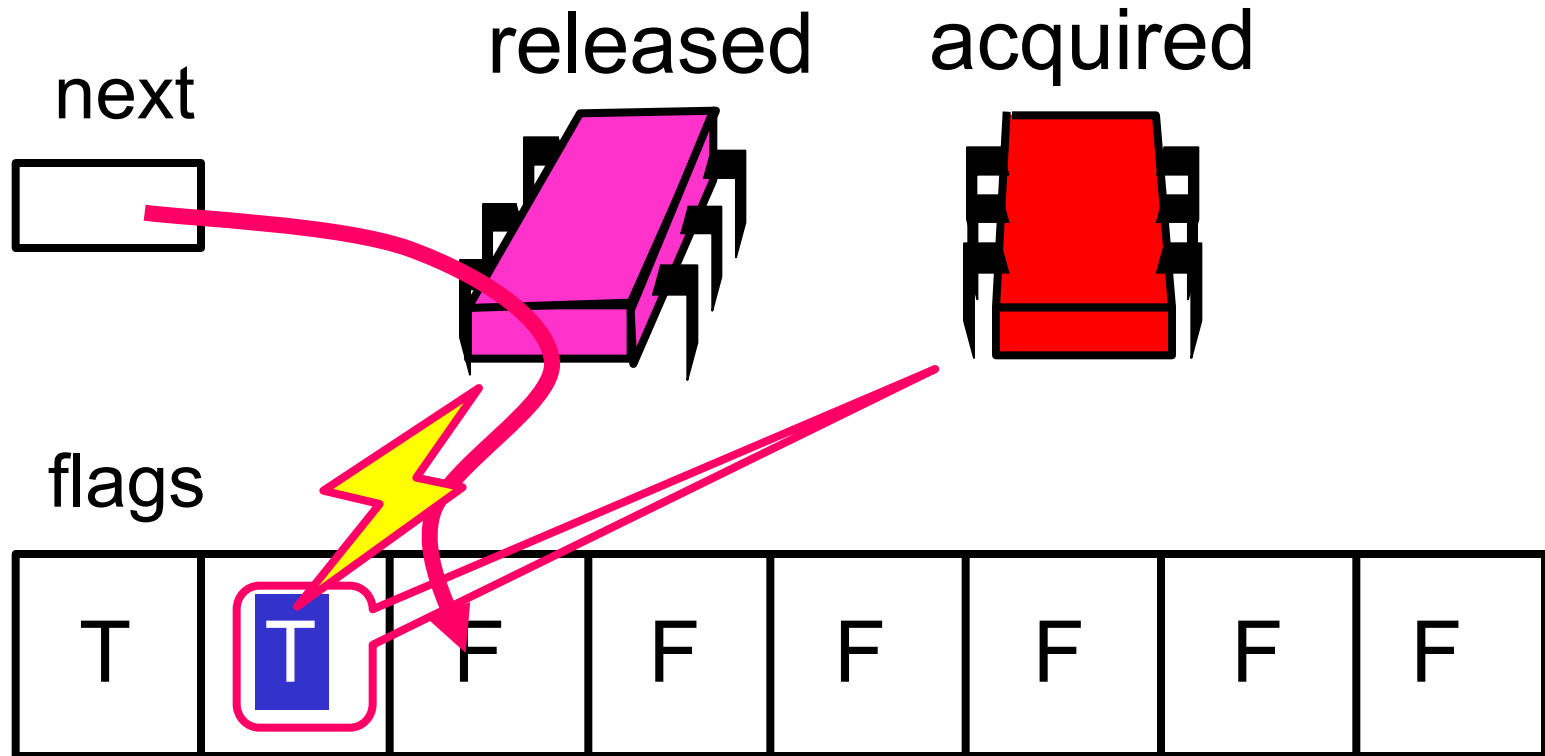
Anderson Queue Lock



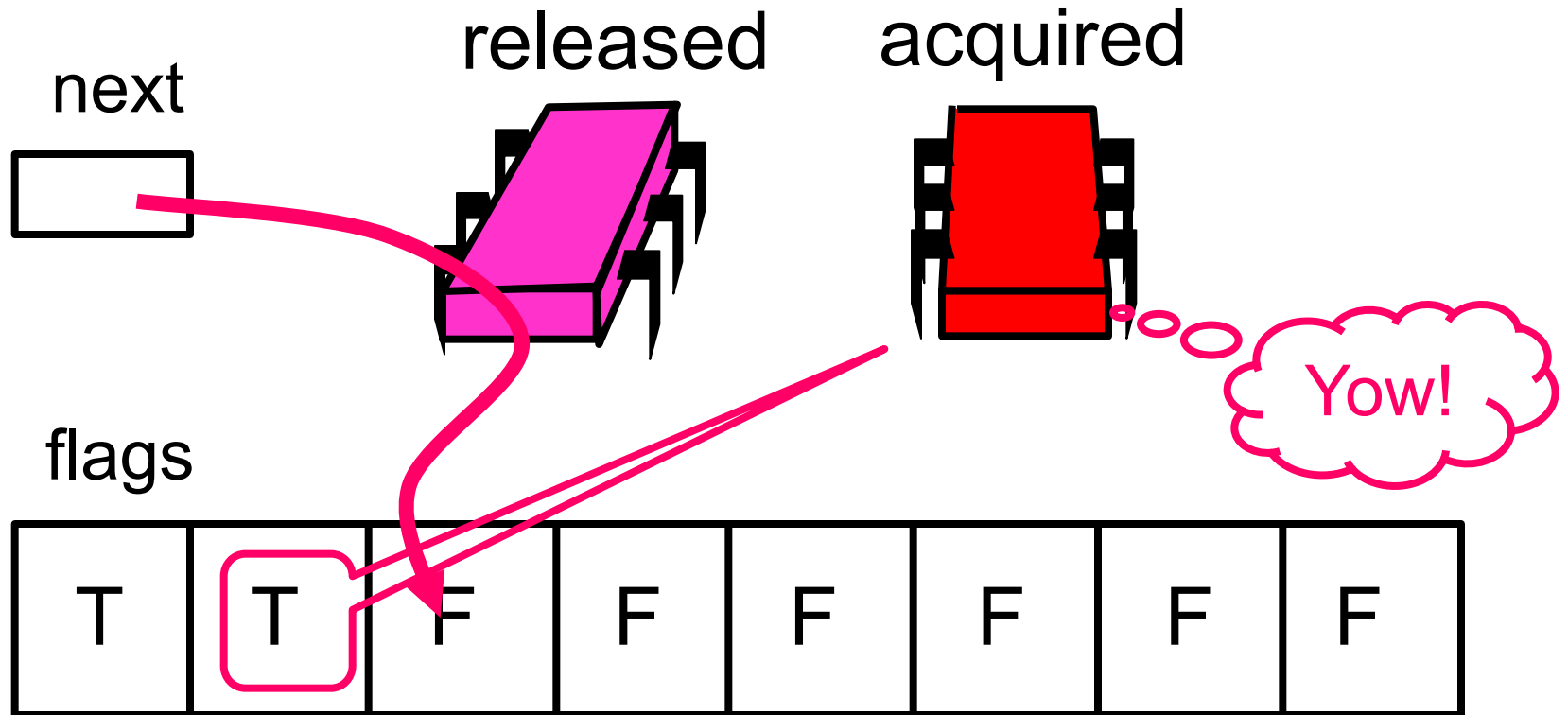
Anderson Queue Lock



Anderson Queue Lock

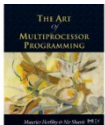


Anderson Queue Lock



Anderson Queue Lock

```
class ALock implements Lock {  
    boolean[] flags={true,false,...,false};  
    AtomicInteger next  
        = new AtomicInteger(0);  
    ThreadLocal<Integer> mySlot;
```



Anderson Queue Lock

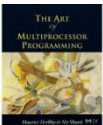
```
class ALock implements Lock {  
    boolean[] flags={true,false,...,false};  
    AtomicInteger next  
        = new AtomicInteger(0);  
    ThreadLocal<Integer> mySlot;
```

One flag per thread

Anderson Queue Lock

```
class ALock implements Lock {  
    boolean[] flags={true,false,...,false};  
    AtomicInteger next  
        = new AtomicInteger(0);  
    ThreadLocal<Integer> mySlot;
```

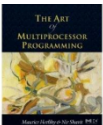
Next flag to use



Anderson Queue Lock

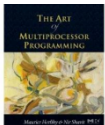
```
class ALock implements Lock {  
    boolean[] flags={true,false,...,false};  
    AtomicInteger next  
        = new AtomicInteger(0);  
    ThreadLocal<Integer> mySlot;
```

Thread-local variable



Anderson Queue Lock

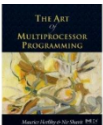
```
public lock() {  
    mySlot = next.getAndIncrement();  
    while (!flags[mySlot % n]) {};  
    flags[mySlot % n] = false;  
}  
  
public unlock() {  
    flags[(mySlot+1) % n] = true;  
}
```



Anderson Queue Lock

```
public lock() {  
    mySlot = next.getAndIncrement();  
    while (!flags[mySlot % n]) {};  
    flags[mySlot % n] = false;  
}  
  
public unlock() {  
    flags[(mySlot+1) % n] = true;  
}
```

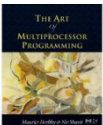
Take next slot



Anderson Queue Lock

```
public lock() {  
    mySlot = next.getAndIncrement();  
    while (!flags[mySlot % n]) {};  
    flags[mySlot % n] = false;  
}  
  
public unlock() {  
    flags[(mySlot+1) % n] = true;  
}
```

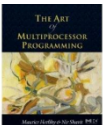
Spin until told to go



Anderson Queue Lock

```
public lock() {  
    myslot = next.getAndIncrement();  
    while (!flags[myslot % n]) {};  
    flags[myslot % n] = false;  
}  
  
public unlock() {  
    flags[(myslot+1) % n] = true;  
}
```

Prepare slot for re-use

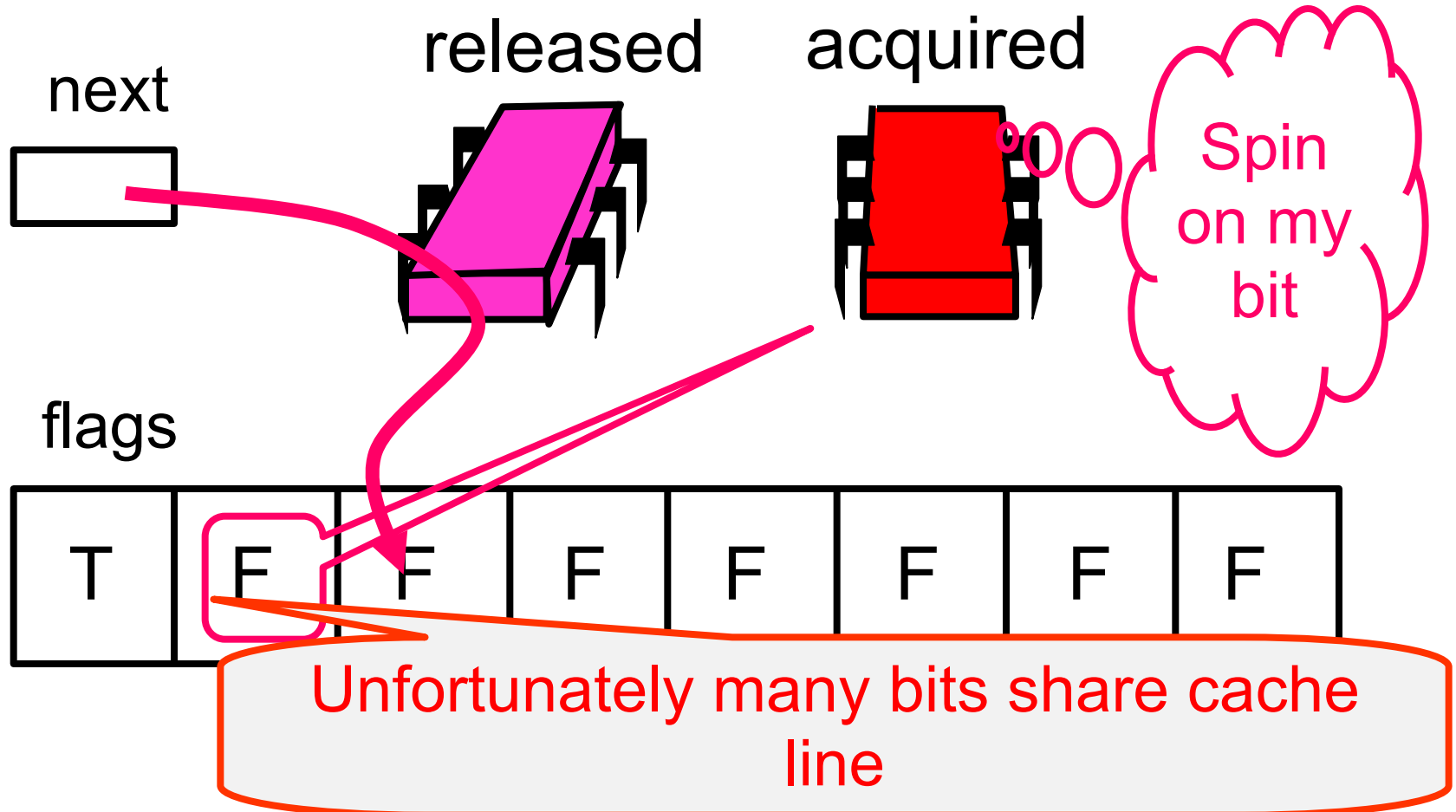


Anderson Queue Lock

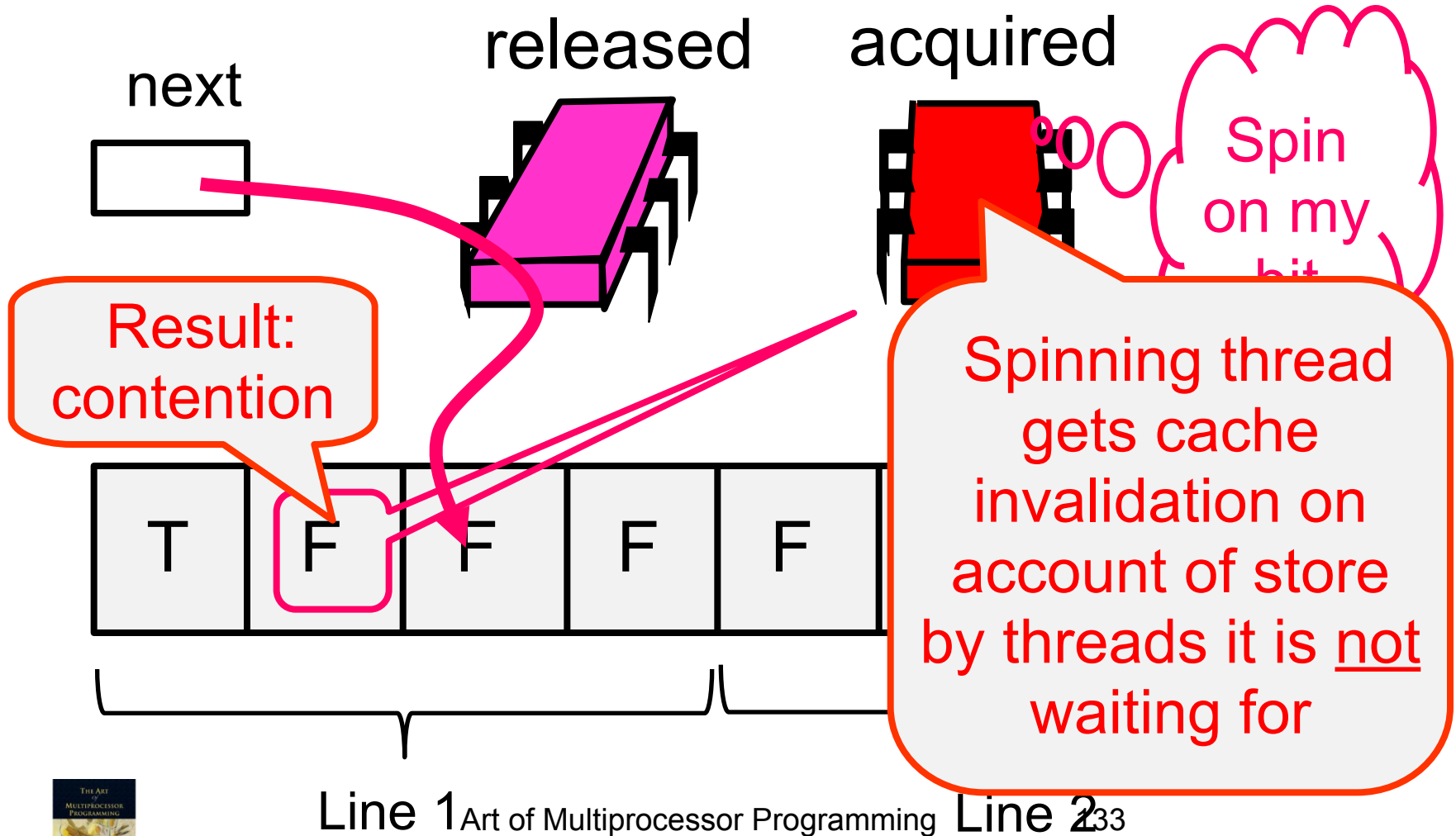
```
public lock() { Tell next thread to go
    mySlot = next.getAndIncrement();
    while (!flags[mySlot % n]) {};
    flags[mySlot % n] = false;
}

public unlock() {
    flags[(mySlot+1) % n] = true;
}
```

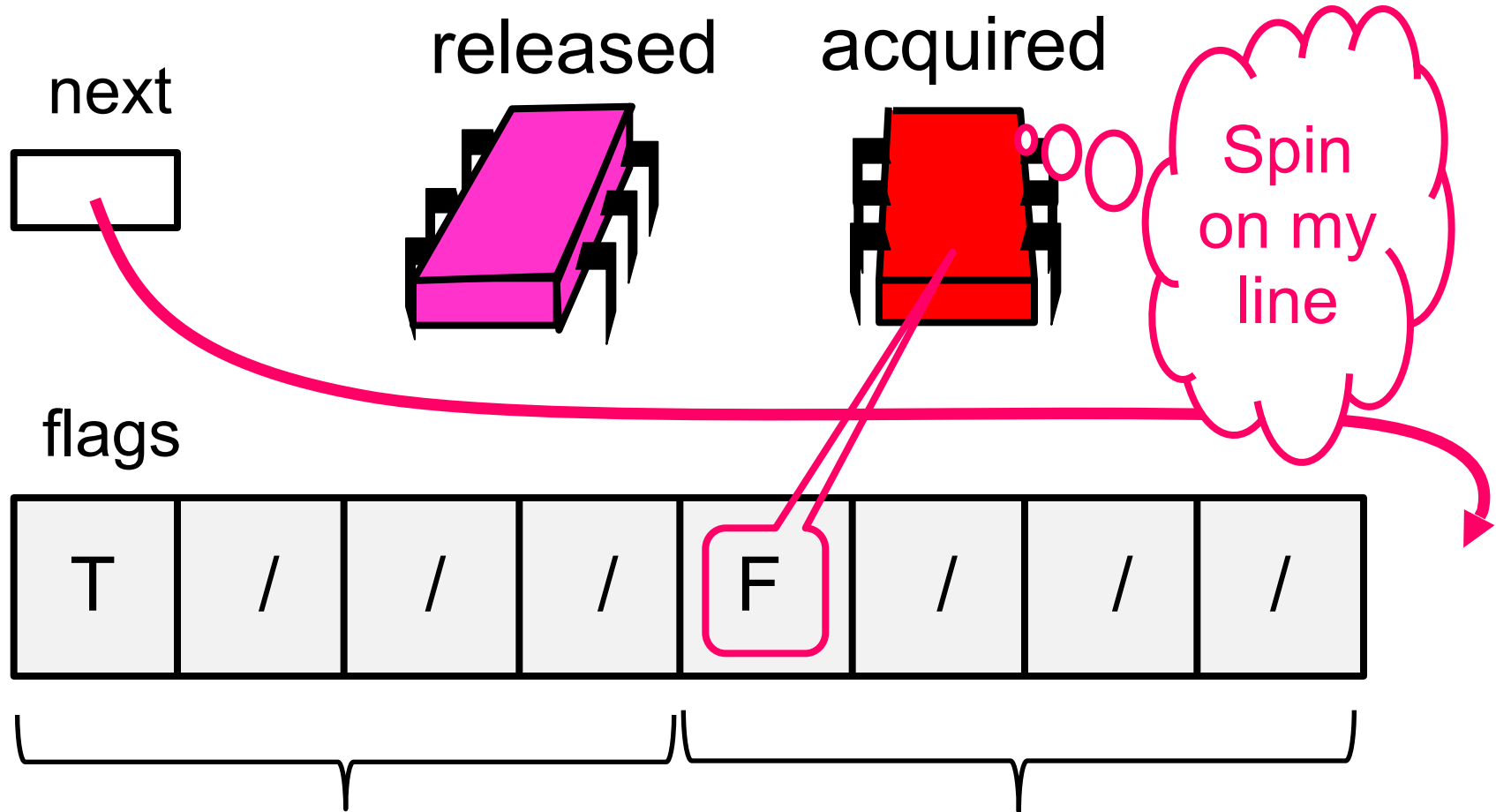
Local Spinning



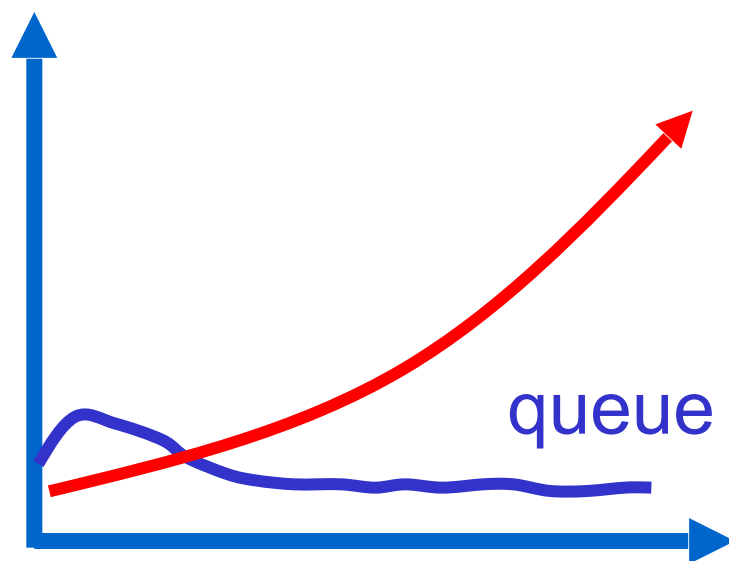
False Sharing



The Solution: Padding



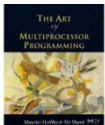
Performance



TTAS

queue

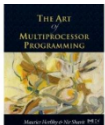
- **Shorter handover than backoff**
- **Curve is practically flat**
- **Scalable performance**



Anderson Queue Lock

Good

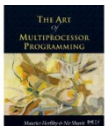
- First truly scalable lock
- Simple, easy to implement
- Back to FIFO order (like Bakery)



Anderson Queue Lock

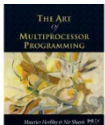
Bad

- Space hog...
- One bit per thread → one cache line per thread
 - What if unknown number of threads?
 - What if small number of actual contenders?



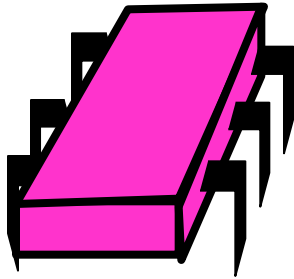
CLH Lock

- FIFO order
- Small, constant-size overhead per thread

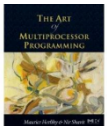
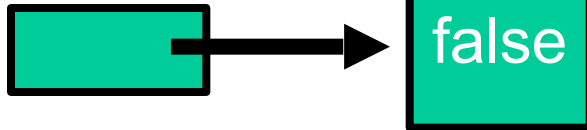


Initially

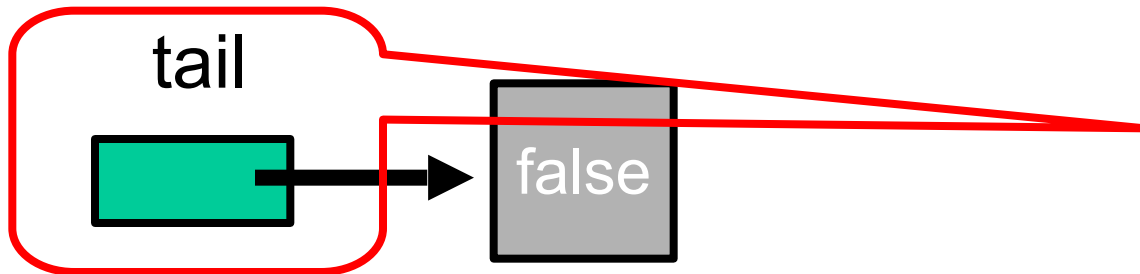
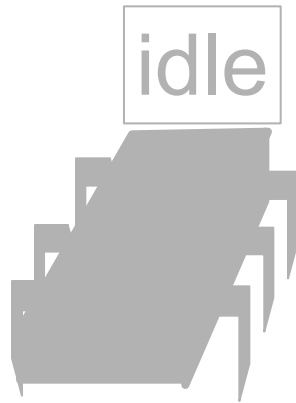
idle



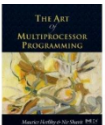
tail



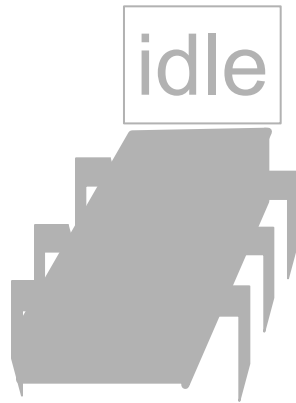
Initially



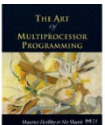
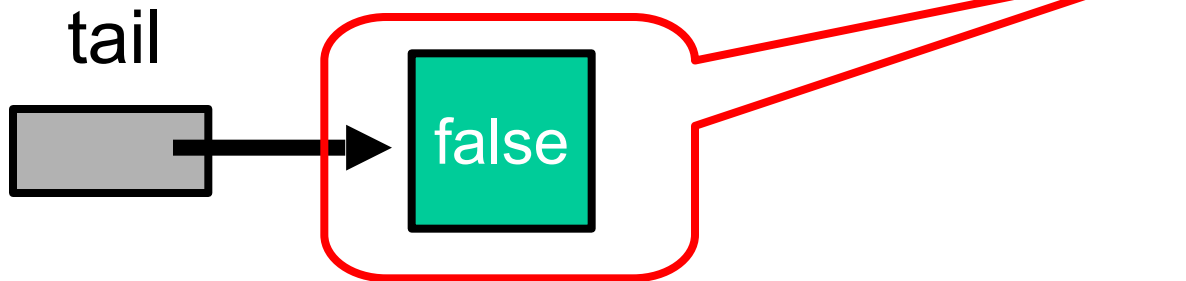
Queue tail



Initially

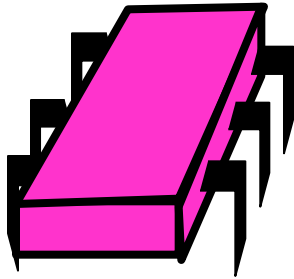


Lock is free

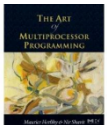
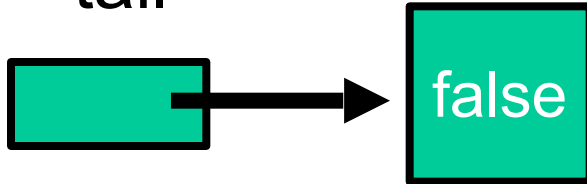


Initially

idle

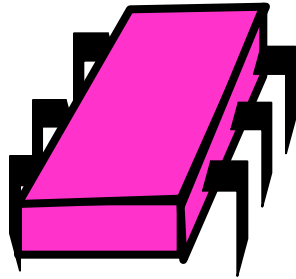


tail

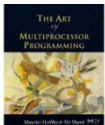
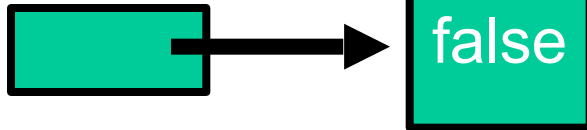


Purple Wants the Lock

acquiring

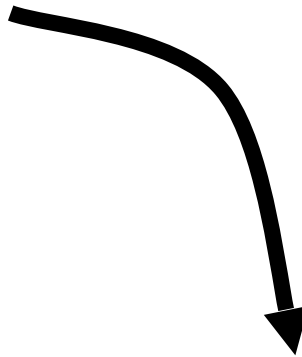
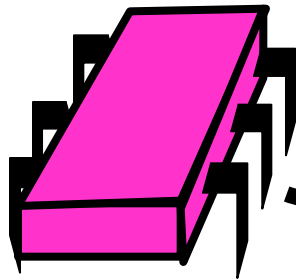


tail

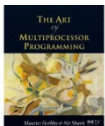
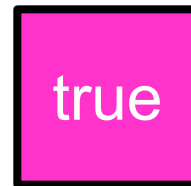
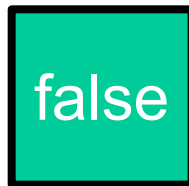


Purple Wants the Lock

acquiring

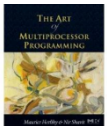
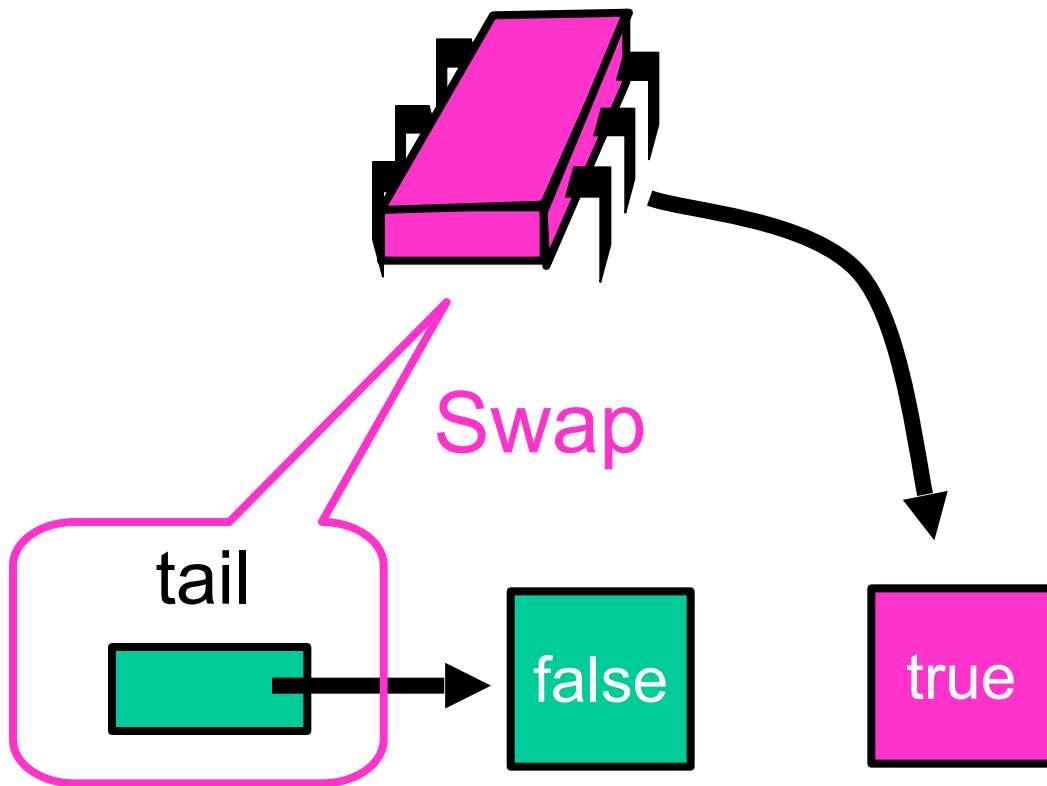


tail



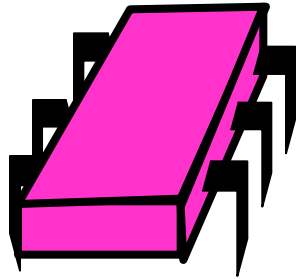
Purple Wants the Lock

acquiring

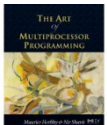
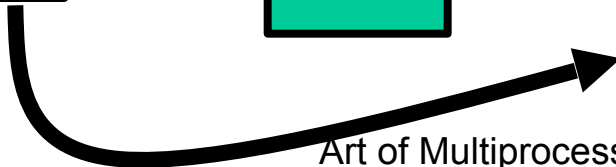
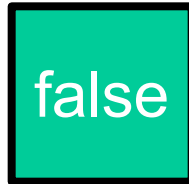


Purple Has the Lock

acquired

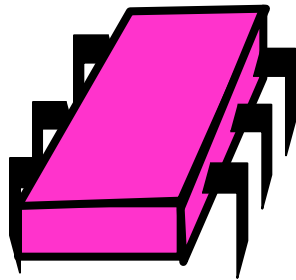


tail

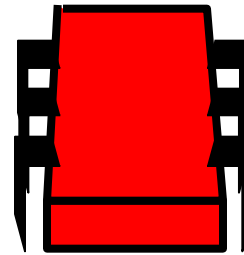


Red Wants the Lock

acquired



acquiring



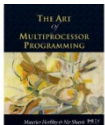
tail



false

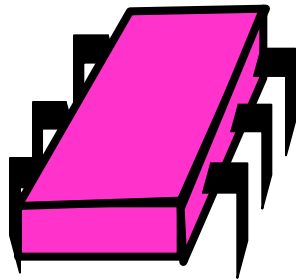
true

true

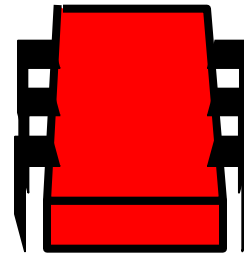


Red Wants the Lock

acquired



acquiring



Swap

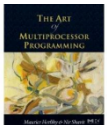
tail



false

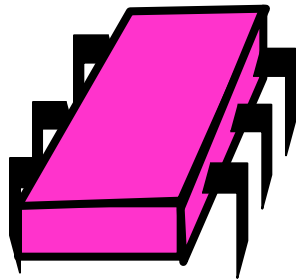
true

true

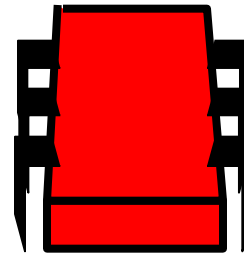


Red Wants the Lock

acquired



acquiring



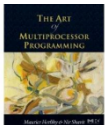
tail



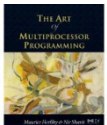
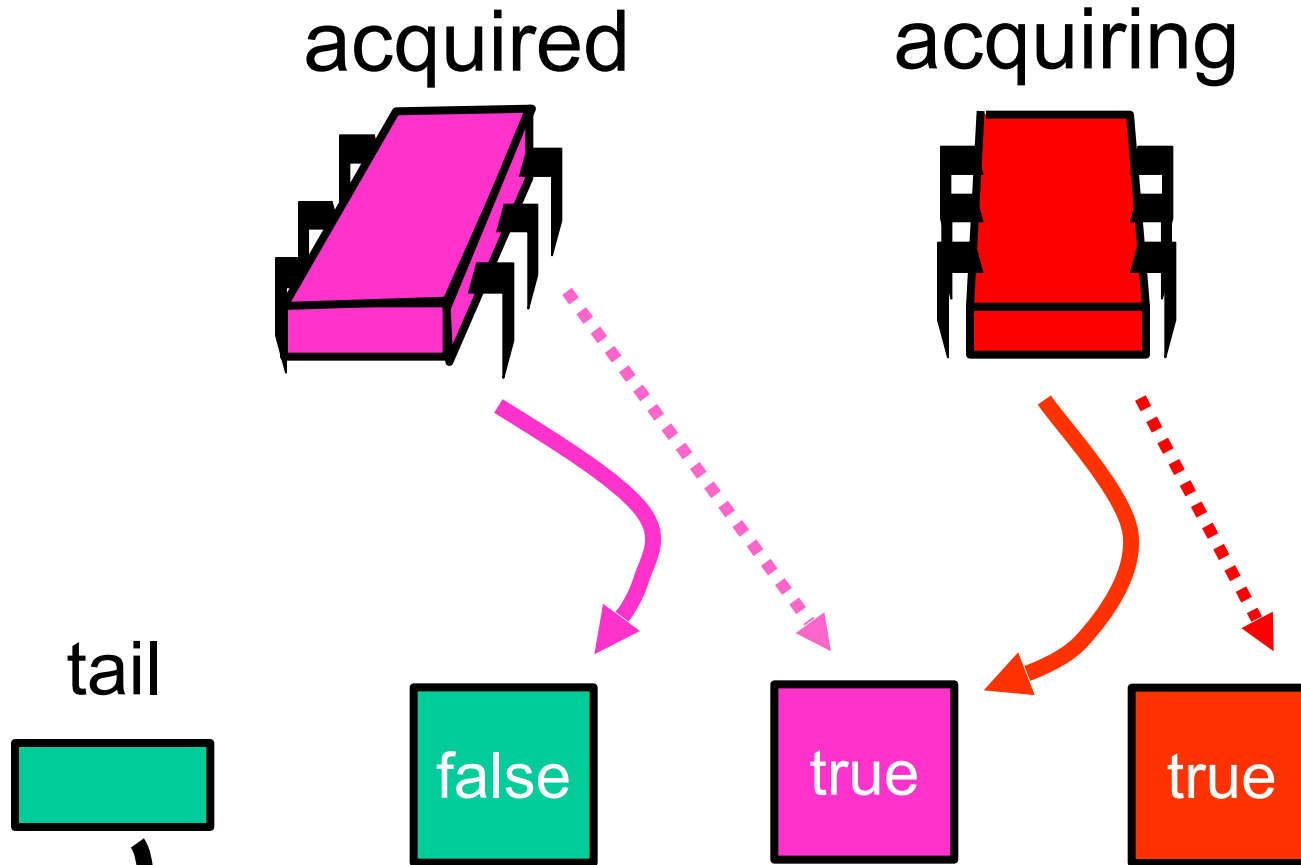
false

true

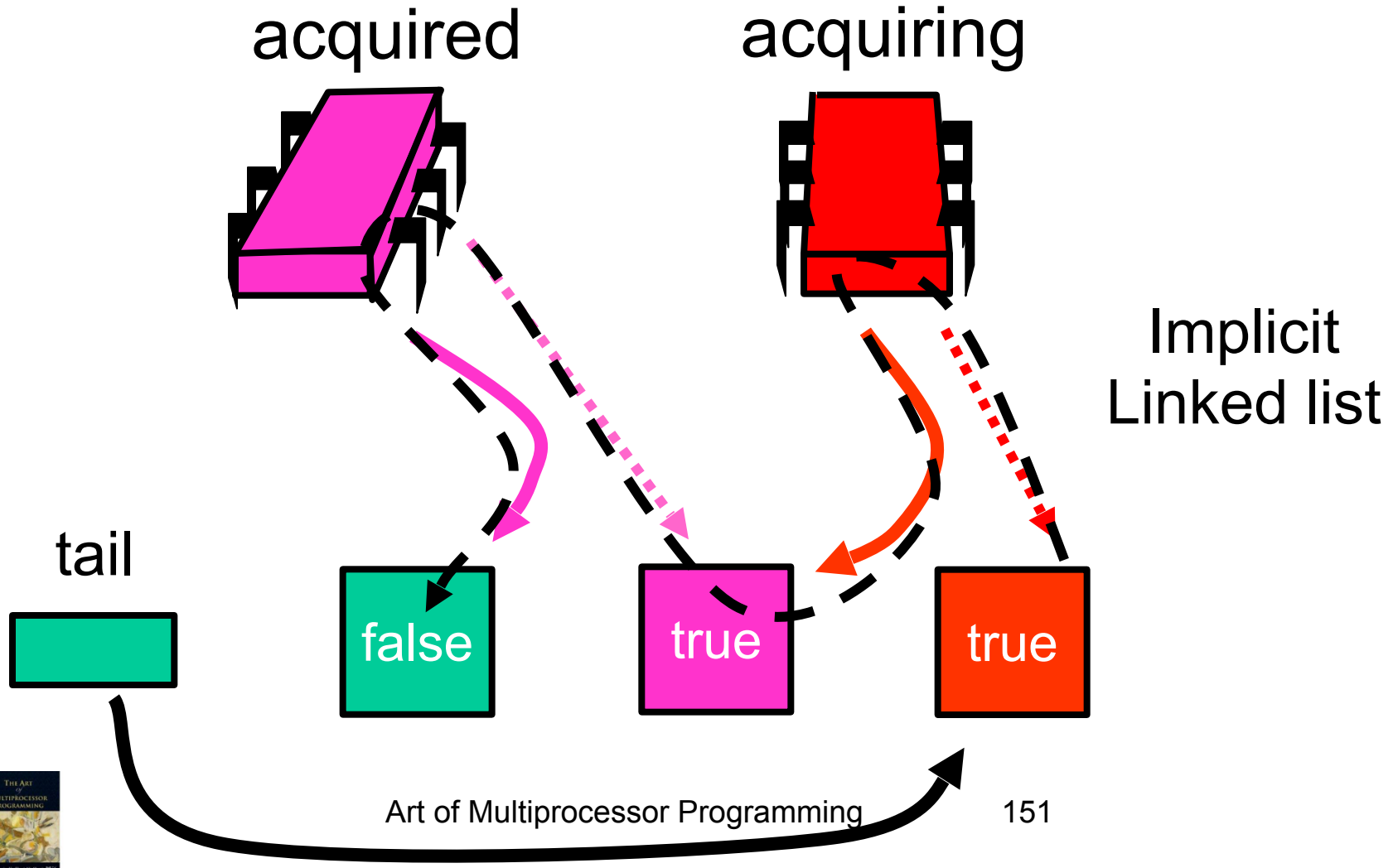
true



Red Wants the Lock

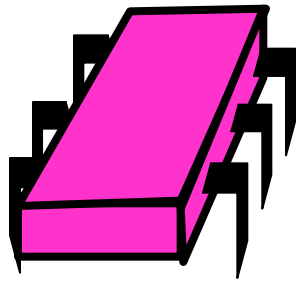


Red Wants the Lock

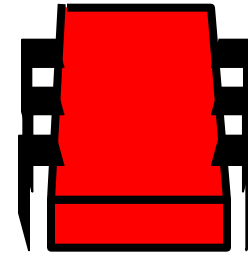


Red Wants the Lock

acquired



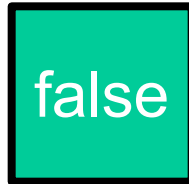
acquiring



tail



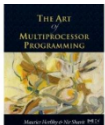
false



true

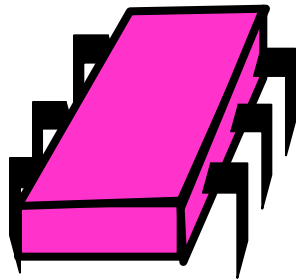


true

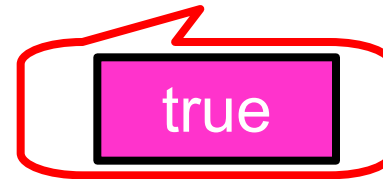
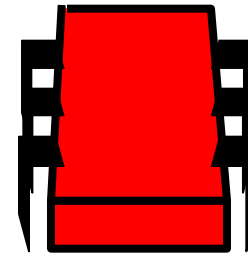


Red Wants the Lock

acquired

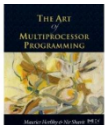
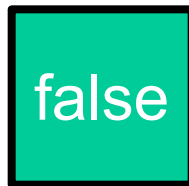


acquiring



Actually, it spins on cached copy

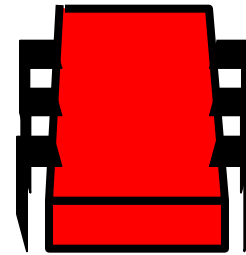
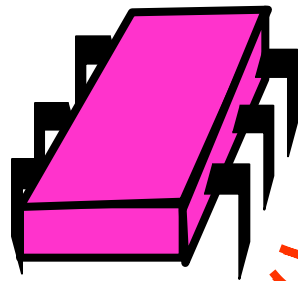
tail



Purple Releases

release

acquiring



false

Bingo!

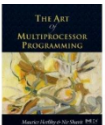
tail



false

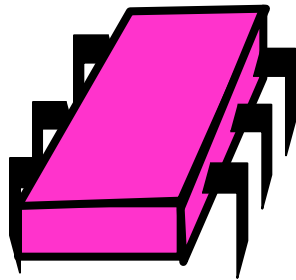
false

true

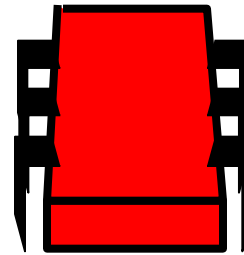


Purple Releases

released



acquired



tail

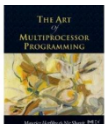


true



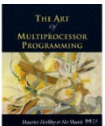
Art of Multiprocessor Programming

155



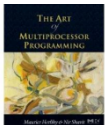
Space Usage

- Let
 - L = number of locks
 - N = number of threads
- ALock
 - $O(LN)$
- CLH lock
 - $O(L+N)$



CLH Queue Lock

```
class Qnode {  
    AtomicBoolean locked =  
        new AtomicBoolean(true);  
}
```



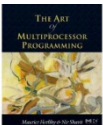
CLH Queue Lock

```
class Qnode {  
    AtomicBoolean locked =  
        new AtomicBoolean(true);  
}
```

Not released yet

CLH Queue Lock

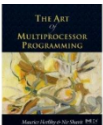
```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```



CLH Queue Lock

```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```

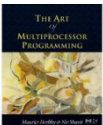
Queue tail



CLH Queue Lock

```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```

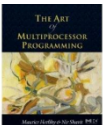
Thread-local Qnode



CLH Queue Lock

```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```

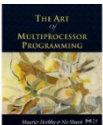
Swap in my node



CLH Queue Lock

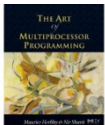
```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```

Spin until predecessor
releases lock



CLH Queue Lock

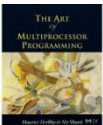
```
Class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```



CLH Queue Lock

```
Class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```

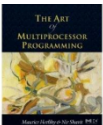
Notify successor



CLH Queue Lock

```
Class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```

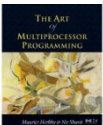
Recycle
predecessor's node



CLH Queue Lock

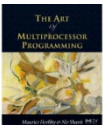
```
Class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```

(we don't actually reuse myNode. Code in book shows how it's done.)



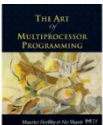
CLH Lock

- Good
 - Lock release affects predecessor only
 - Small, constant-sized space
- Bad
 - Doesn't work for uncached NUMA architectures



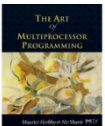
CLH Lock

- Each thread spins on predecessor's memory
- Could be far away ...

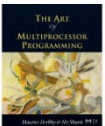
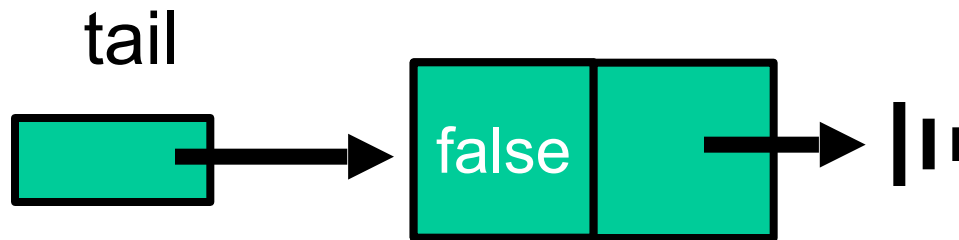
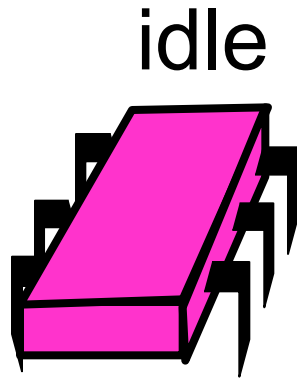


MCS Lock

- FIFO order
- Spin on local memory only
- Small, Constant-size overhead

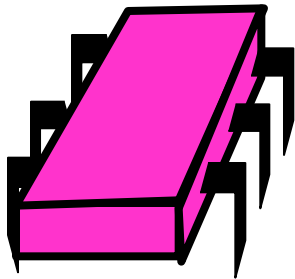


Initially



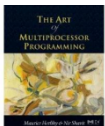
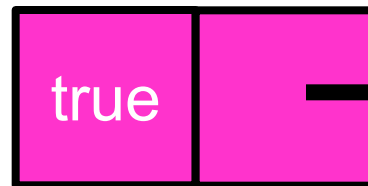
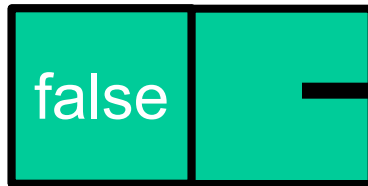
Acquiring

acquiring

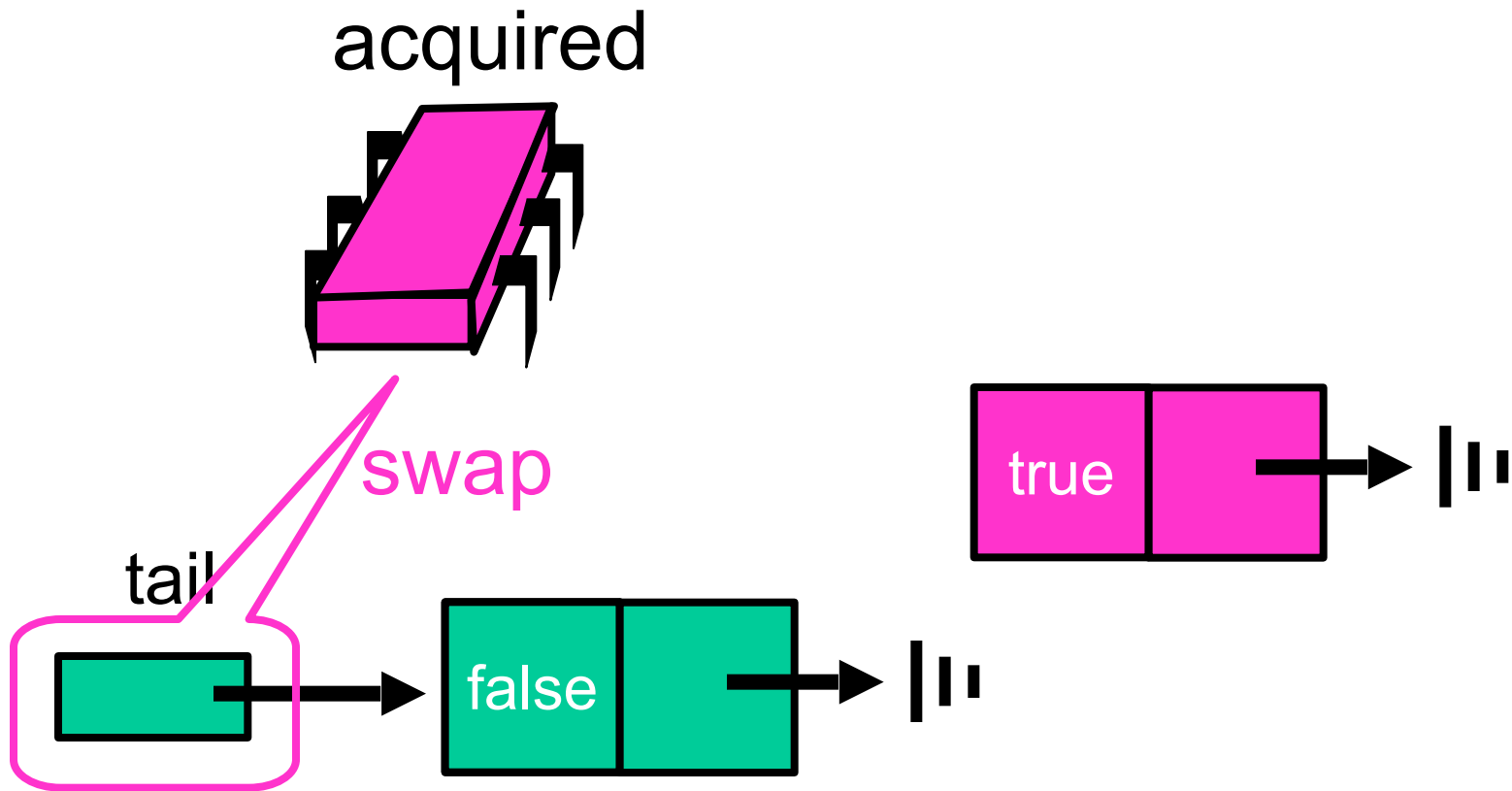


(allocate Qnode)

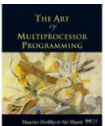
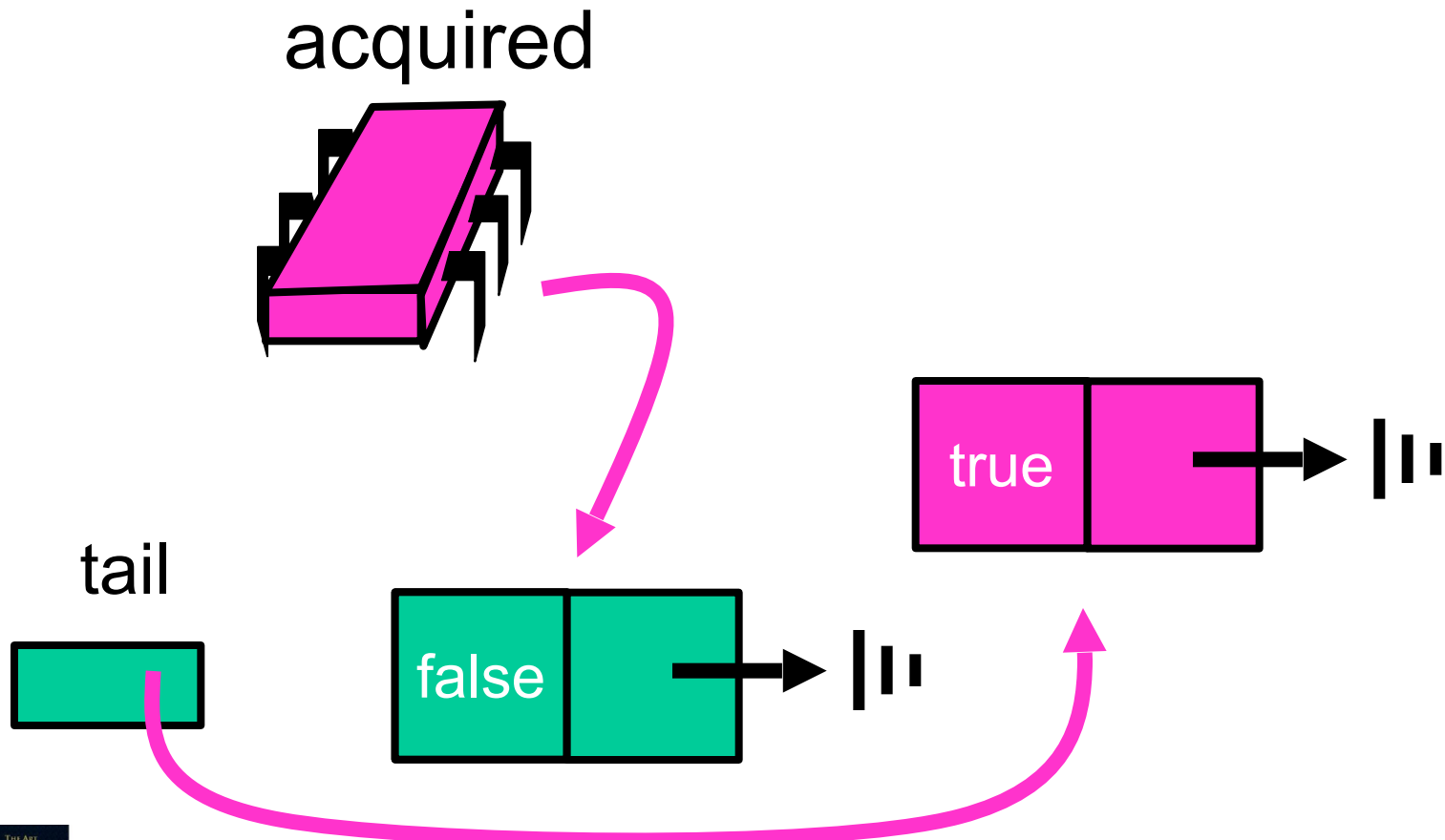
tail



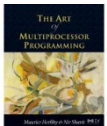
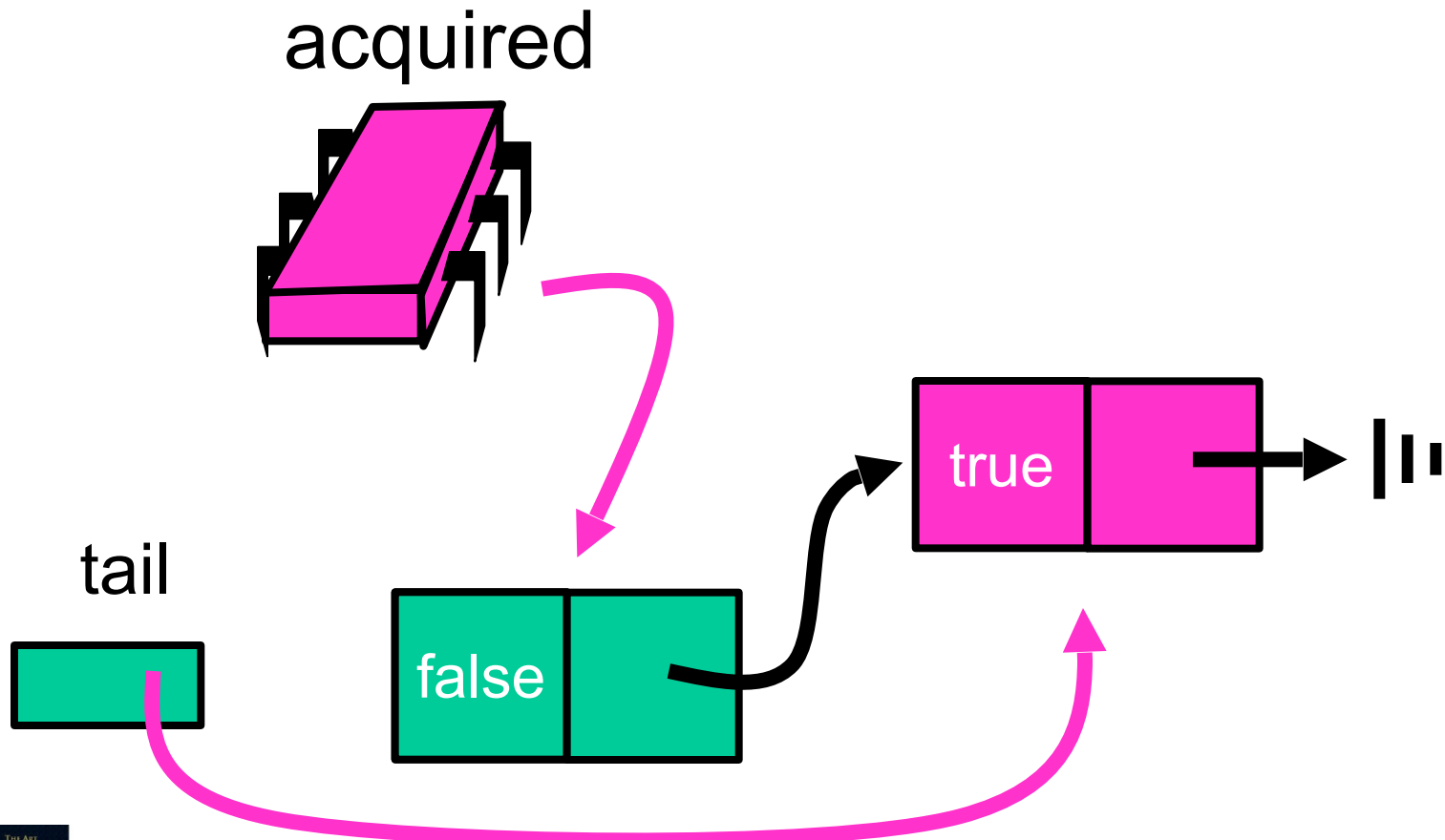
Acquiring



Acquiring

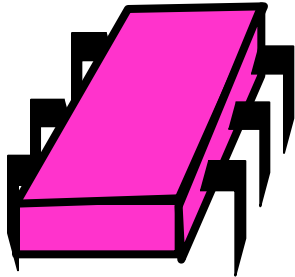


Acquired

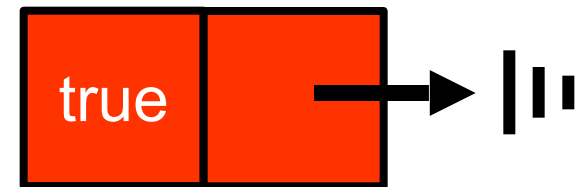
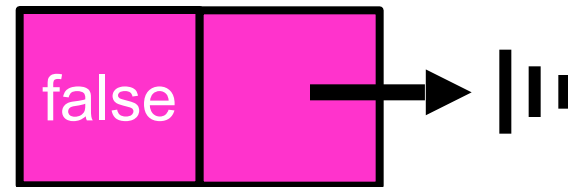
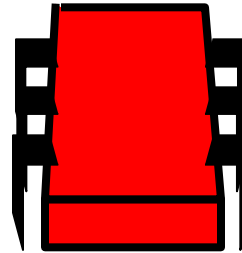


Acquiring

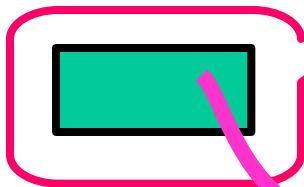
acquired



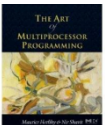
acquiring



tail

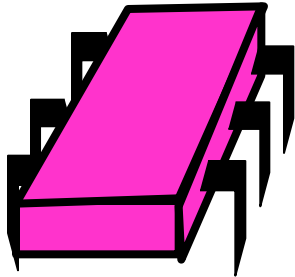


swap

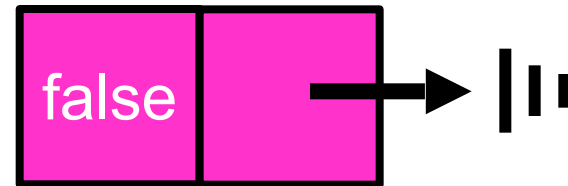
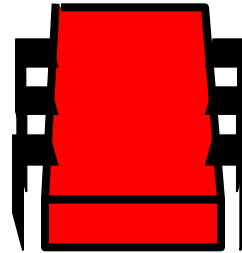


Acquiring

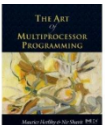
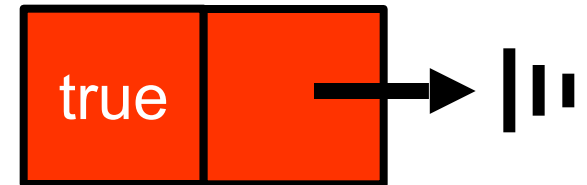
acquired



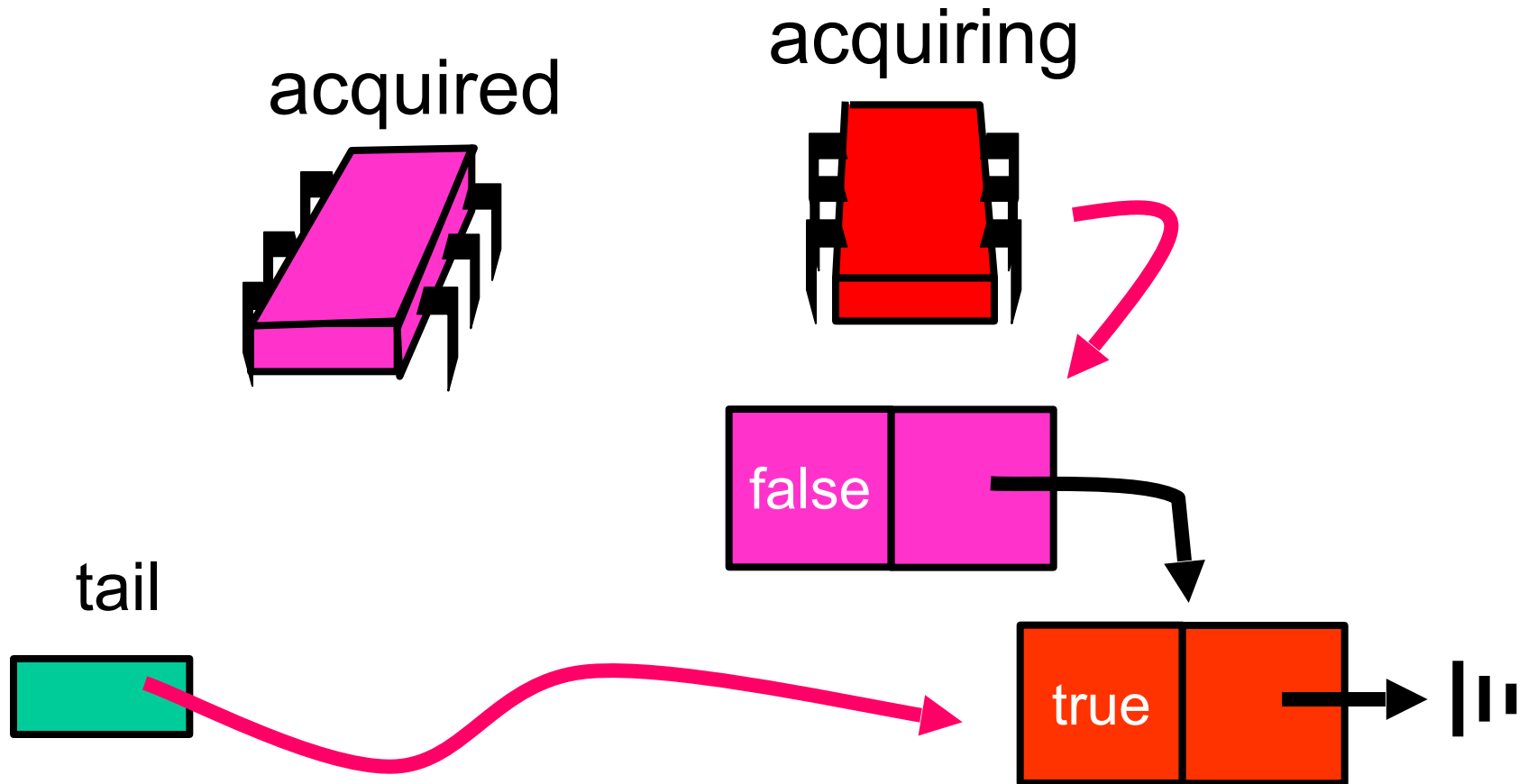
acquiring



tail



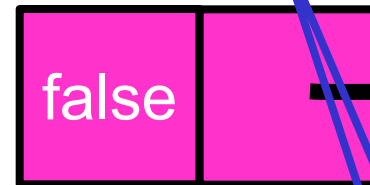
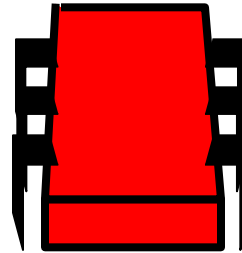
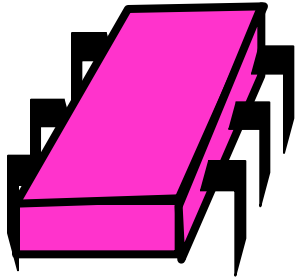
Acquiring



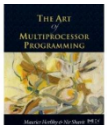
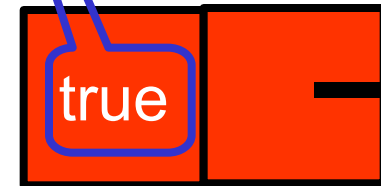
Acquiring

acquiring

acquired



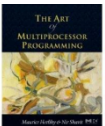
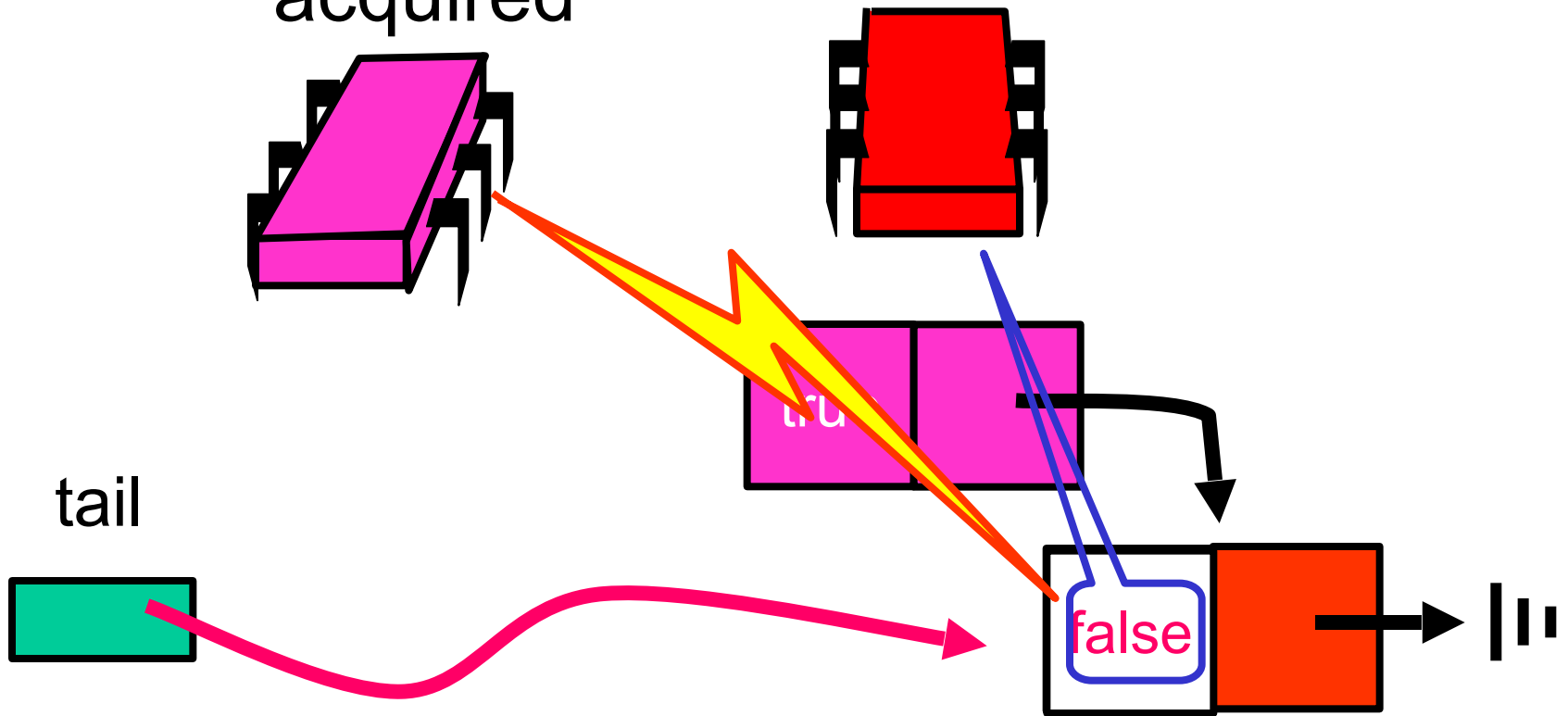
tail



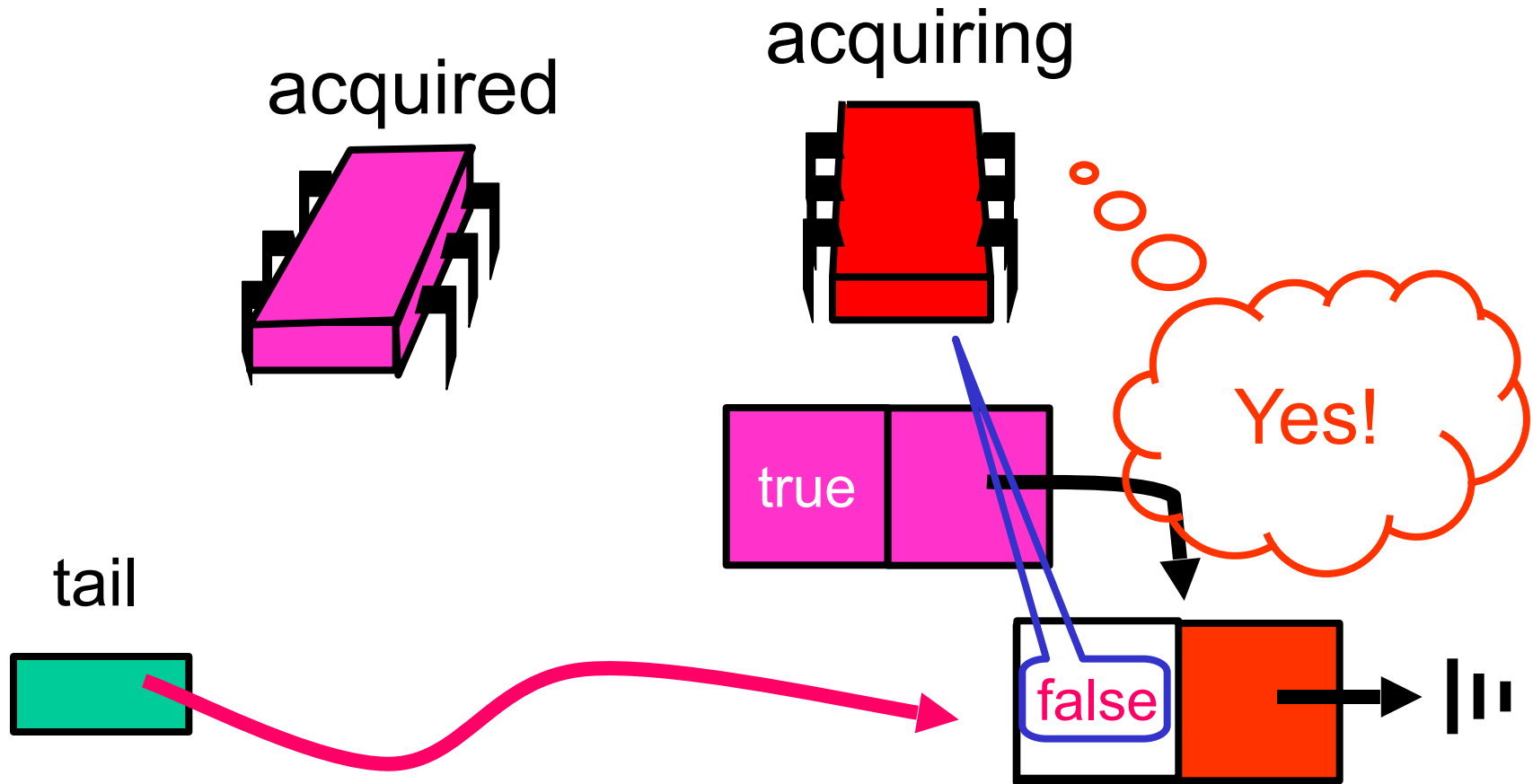
Acquiring

acquiring

acquired

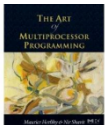


Acquiring



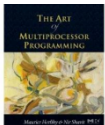
MCS Queue Lock

```
class Qnode {  
    boolean locked = false;  
    qnode next = null;  
}
```



MCS Queue Lock

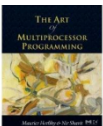
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void lock() {  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked) {}  
        }  
    }  
}
```



MCS Queue Lock

```
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) {}  
        }  
    }  
}
```

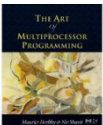
Make a
QNode



MCS Queue Lock

```
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) {}  
        }  
    }  
}
```

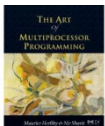
add my Node to
the tail of queue



MCS Queue Lock

```
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) {}  
        }  
    }  
}
```

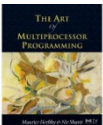
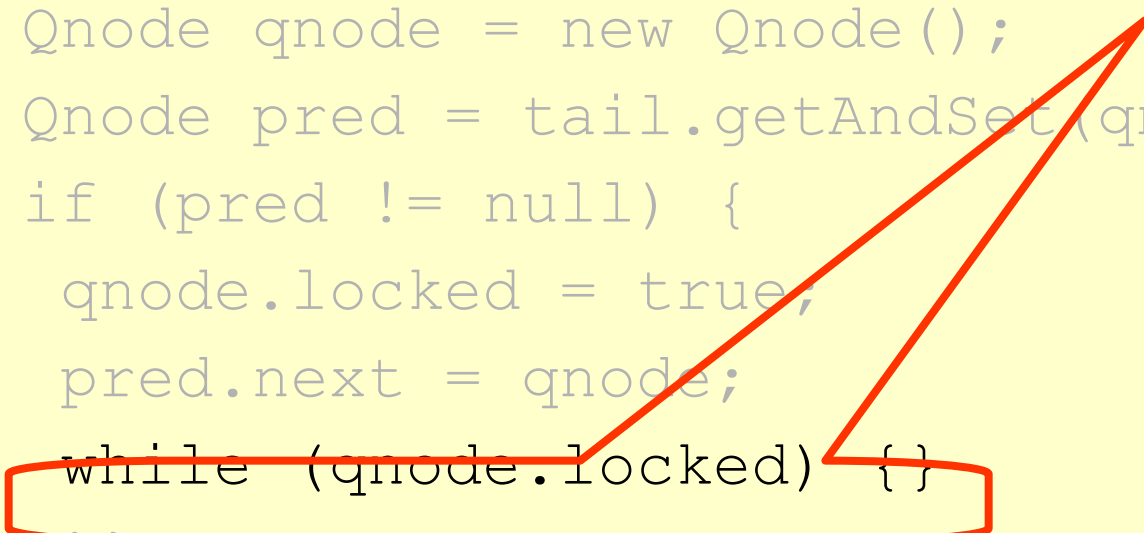
Fix if queue was
non-empty



MCS Queue Lock

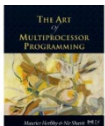
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void lock() {  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked) {}  
        }  
    }  
}
```

**Wait until
unlocked**



MCS Queue Unlock

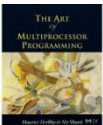
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void unlock() {  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null)  
                return;  
            while (qnode.next == null) {}  
        }  
        qnode.next.locked = false;  
    }  
}
```



MCS Queue Lock

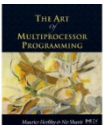
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void unlock() {  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null)  
                return;  
            while (qnode.next == null) {}  
        }  
        qnode.next.locked = false;  
    }  
}
```

**Missing
successor?**



MCS Queue Lock

```
(  
    If really no successor,  
    return  
    k {  
public void unlock() {  
    if (qnode.next == null) {  
        if (tail.CAS(qnode, null)  
            return;  
        while (qnode.next == null) {}  
    }  
    qnode.next.locked = false;  
}}
```



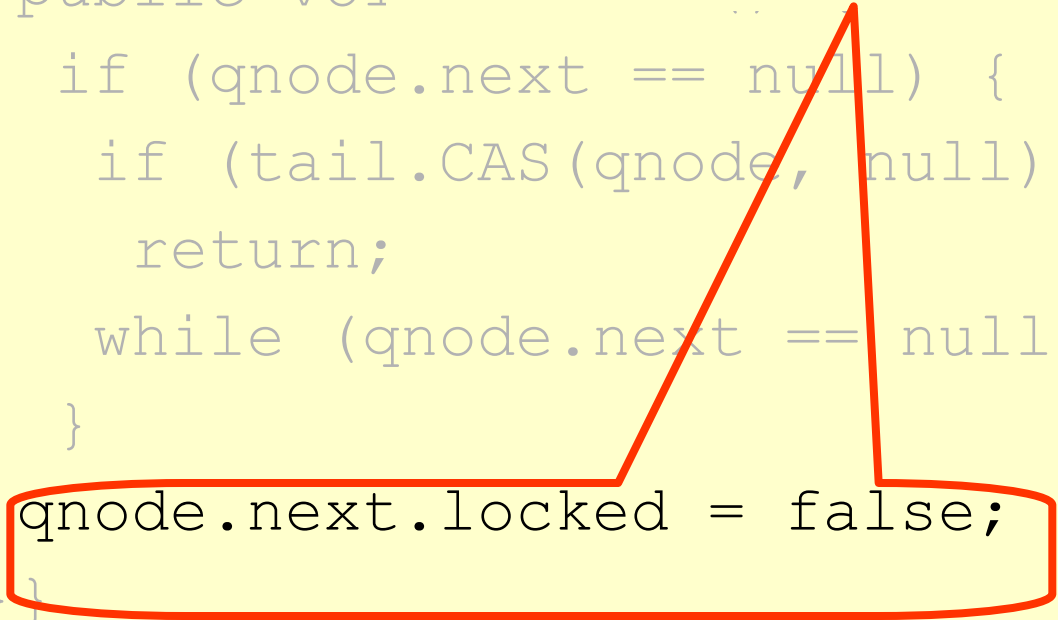
MCS Queue Lock

```
(  
    Otherwise wait for k {  
    successor to catch up  
    public void unlock() {  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null)  
                return;  
            while (qnode.next == null) {}  
        }  
        qnode.next.locked = false;  
    }  
}
```

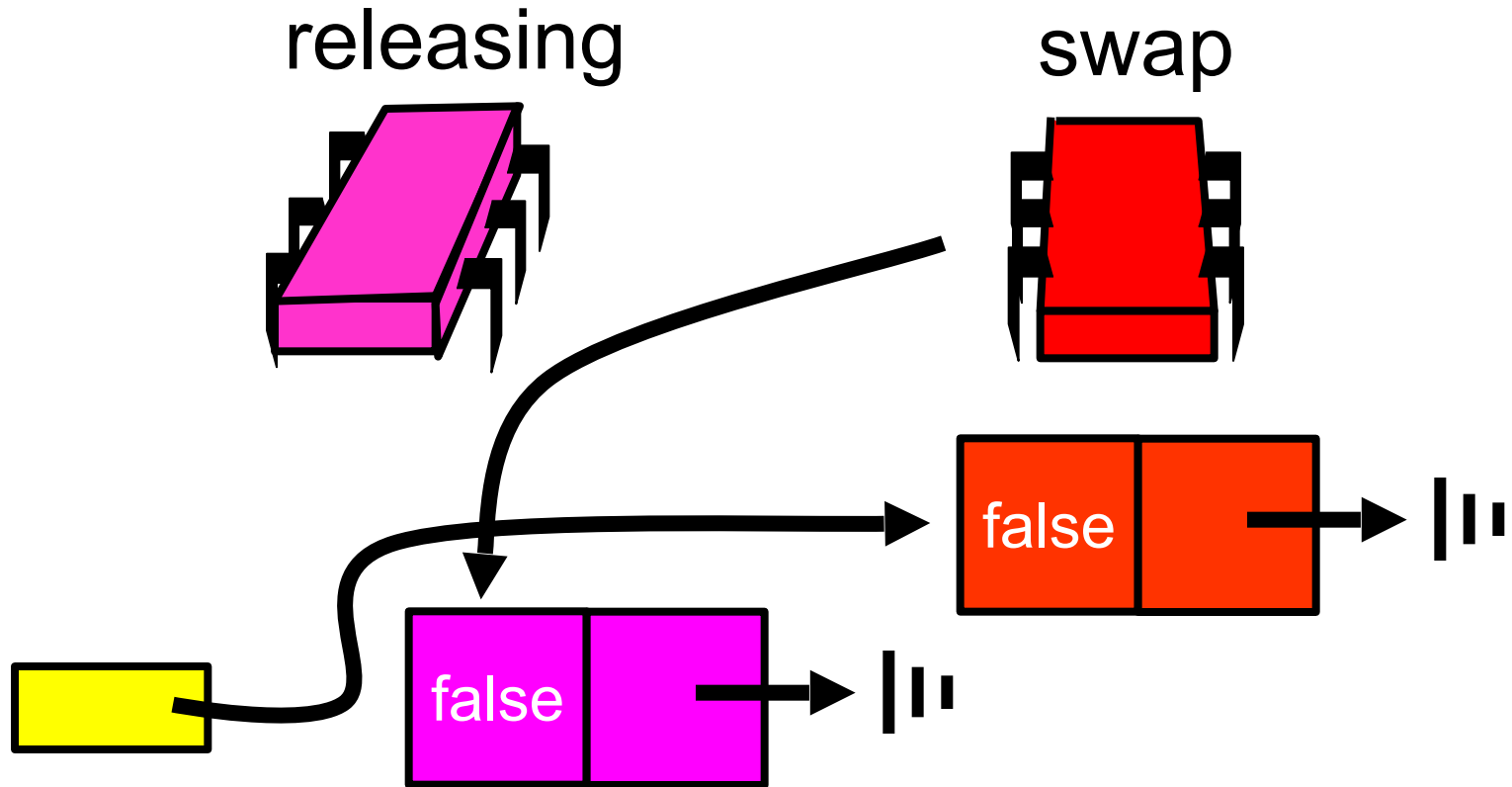
MCS Queue Lock

```
class MCSLock implements Lock {  
    AtomicReference<QueueNode> queue;  
    public void lock() {  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null)  
                return;  
            while (qnode.next == null) {}  
        }  
        qnode.next.locked = false;  
    }  
}
```

Pass lock to successor



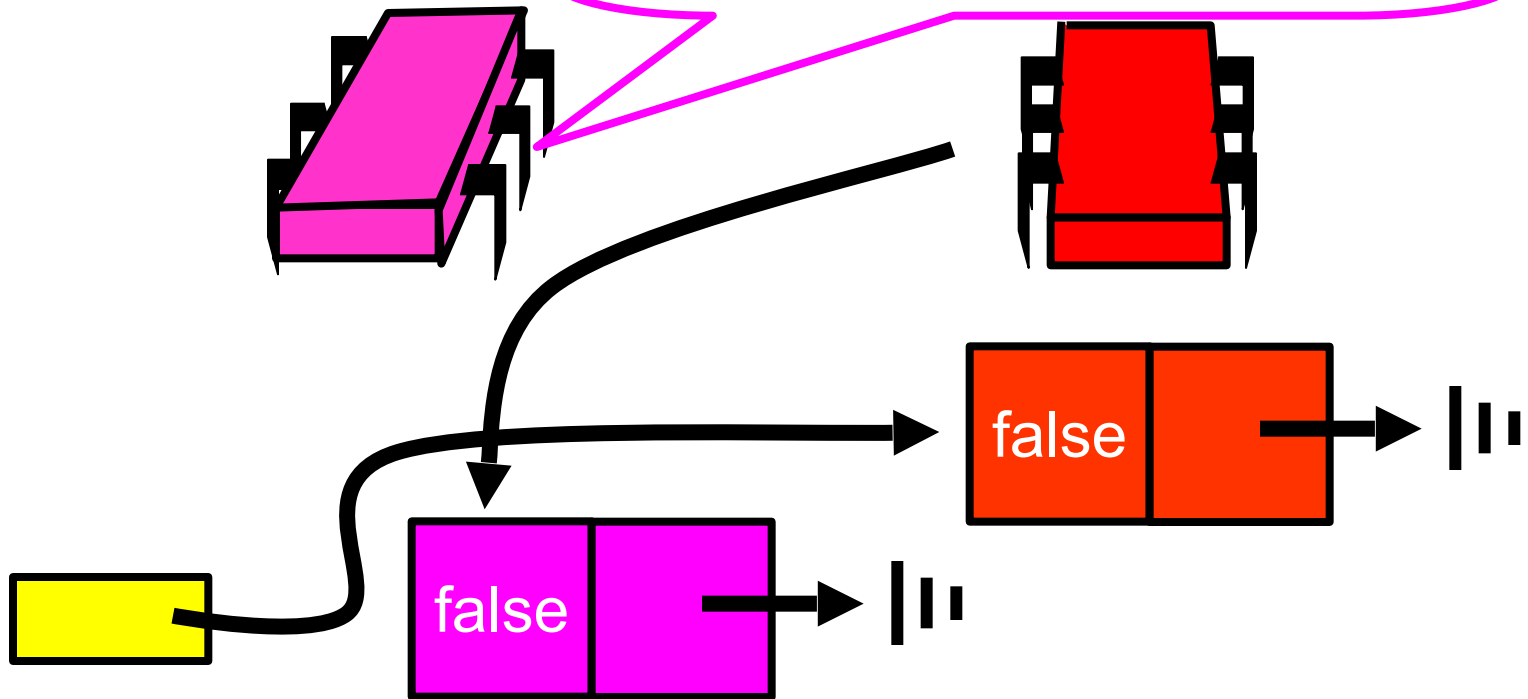
Purple Release



Purple Release

releasin

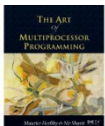
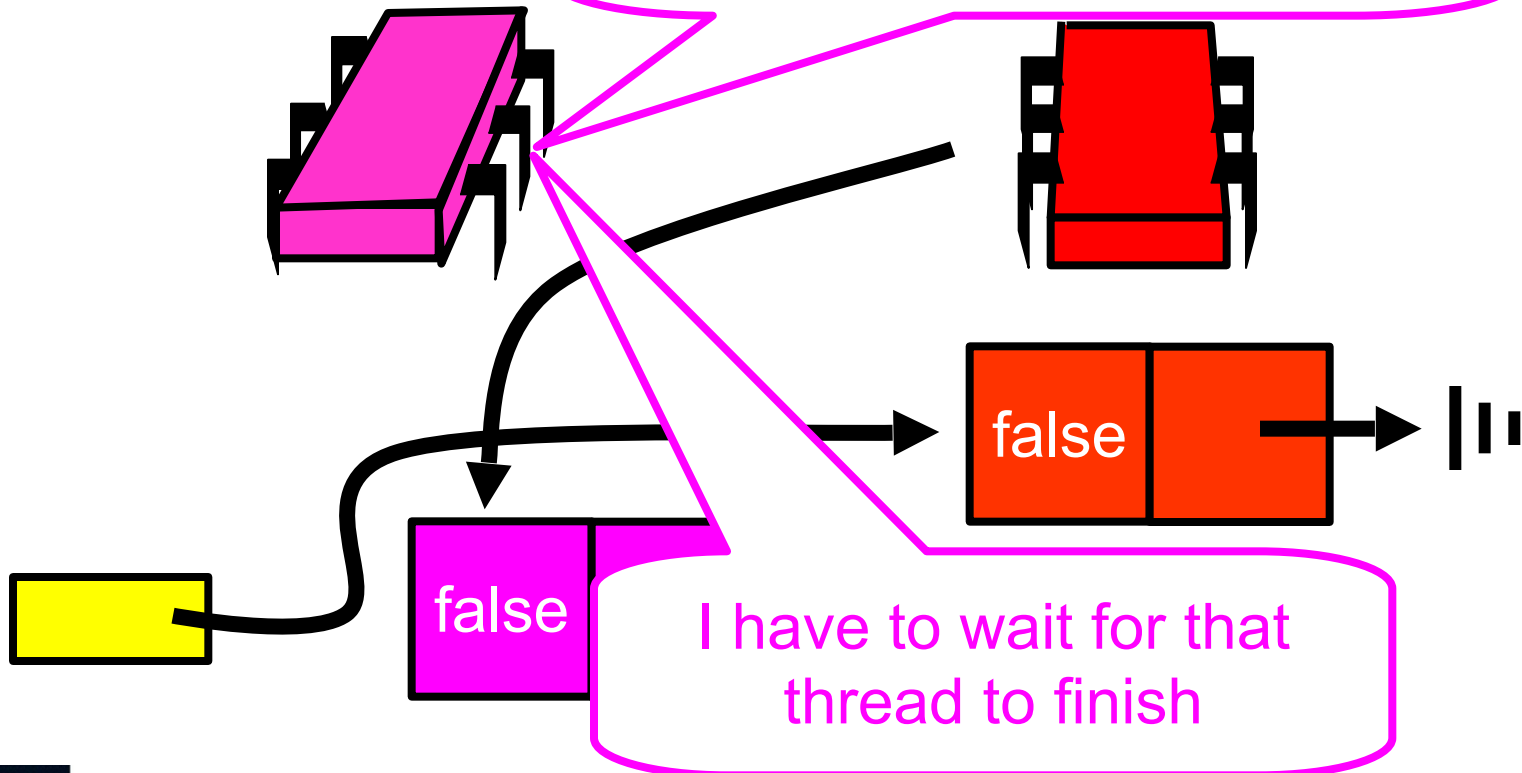
By looking at the queue, I see
another thread is active



Purple Release

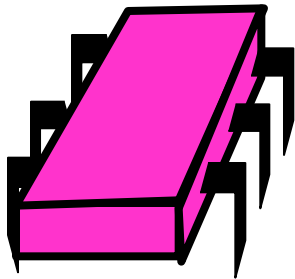
releasin

By looking at the queue, I see another thread is active

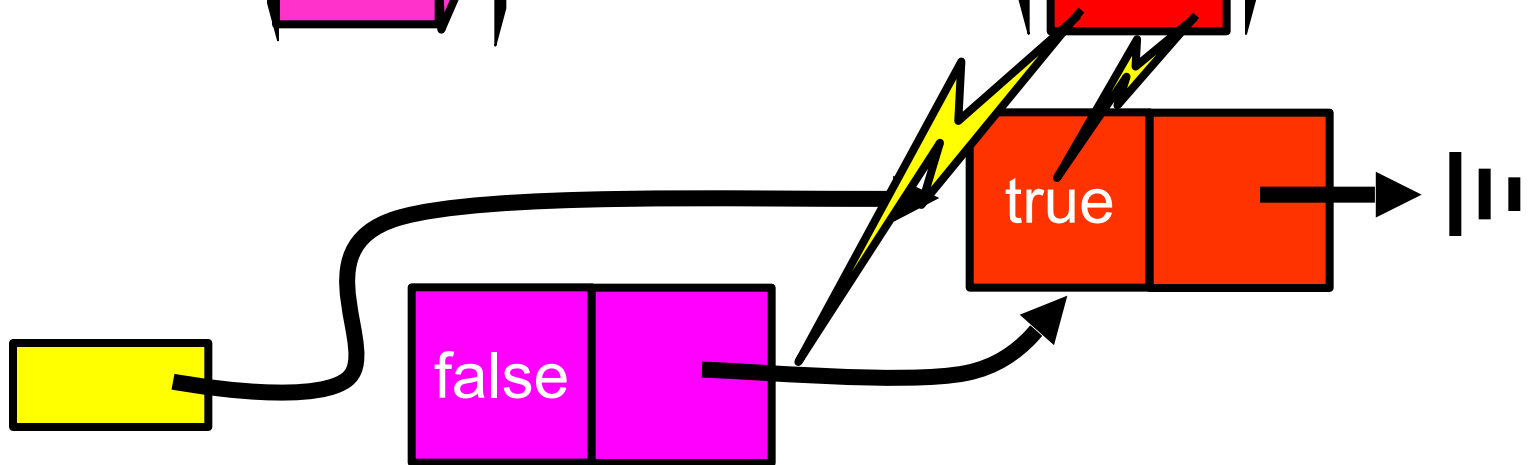
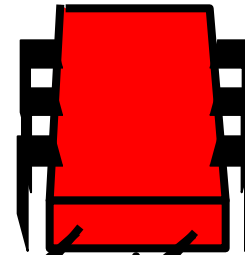


Purple Release

releasing

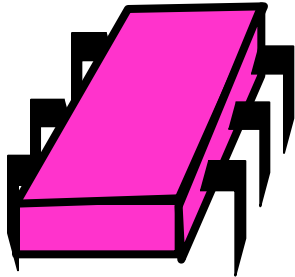


prepare to spin

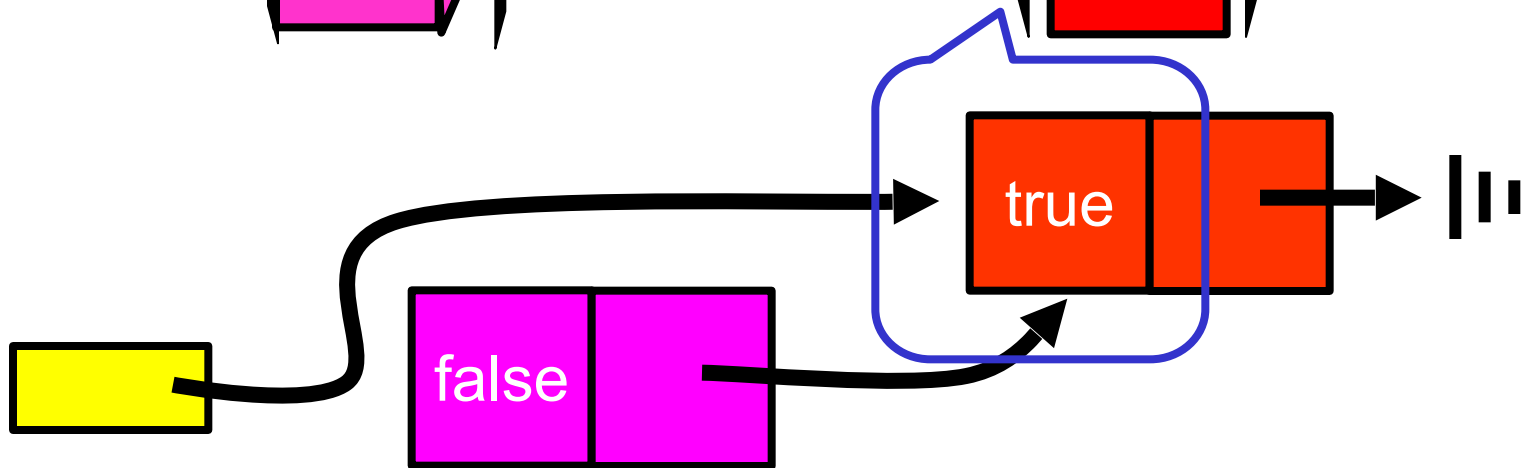
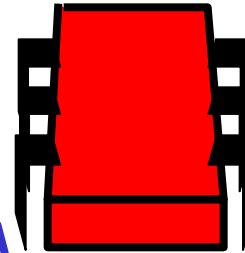


Purple Release

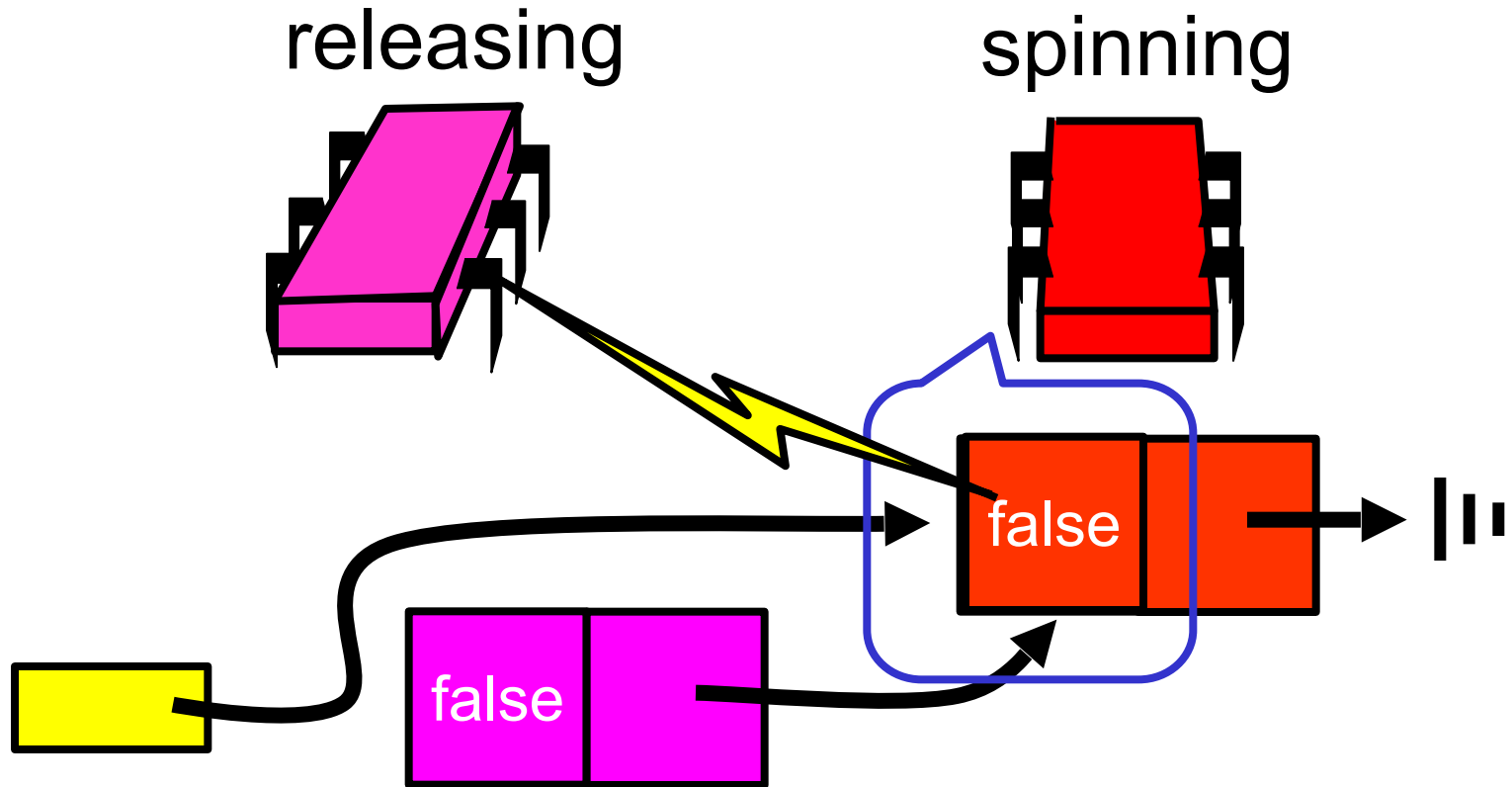
releasing



spinning

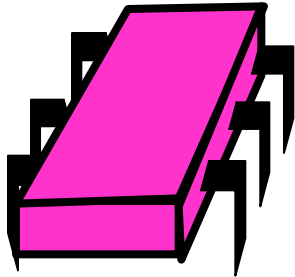


Purple Release

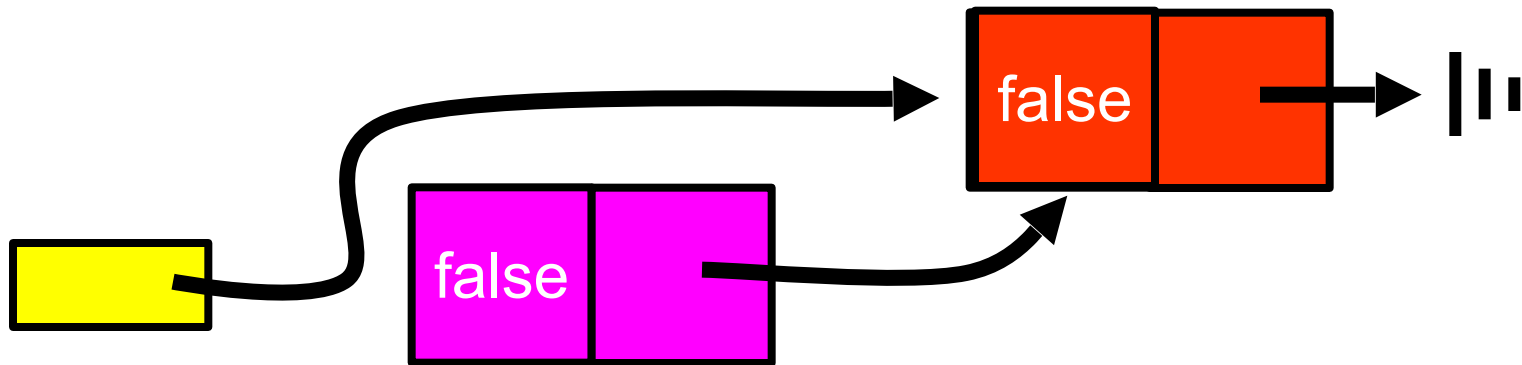
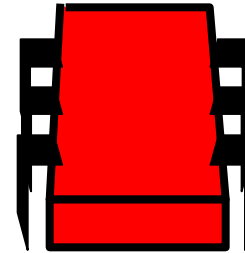


Purple Release

releasing

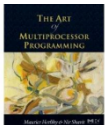


Acquired lock



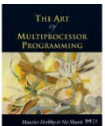
Abortable Locks

- What if you want to give up waiting for a lock?
- For example
 - Timeout
 - Database transaction aborted by user



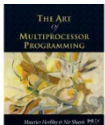
Back-off Lock

- Aborting is trivial
 - Just return from lock() call
- Extra benefit:
 - No cleaning up
 - Wait-free
 - Immediate return

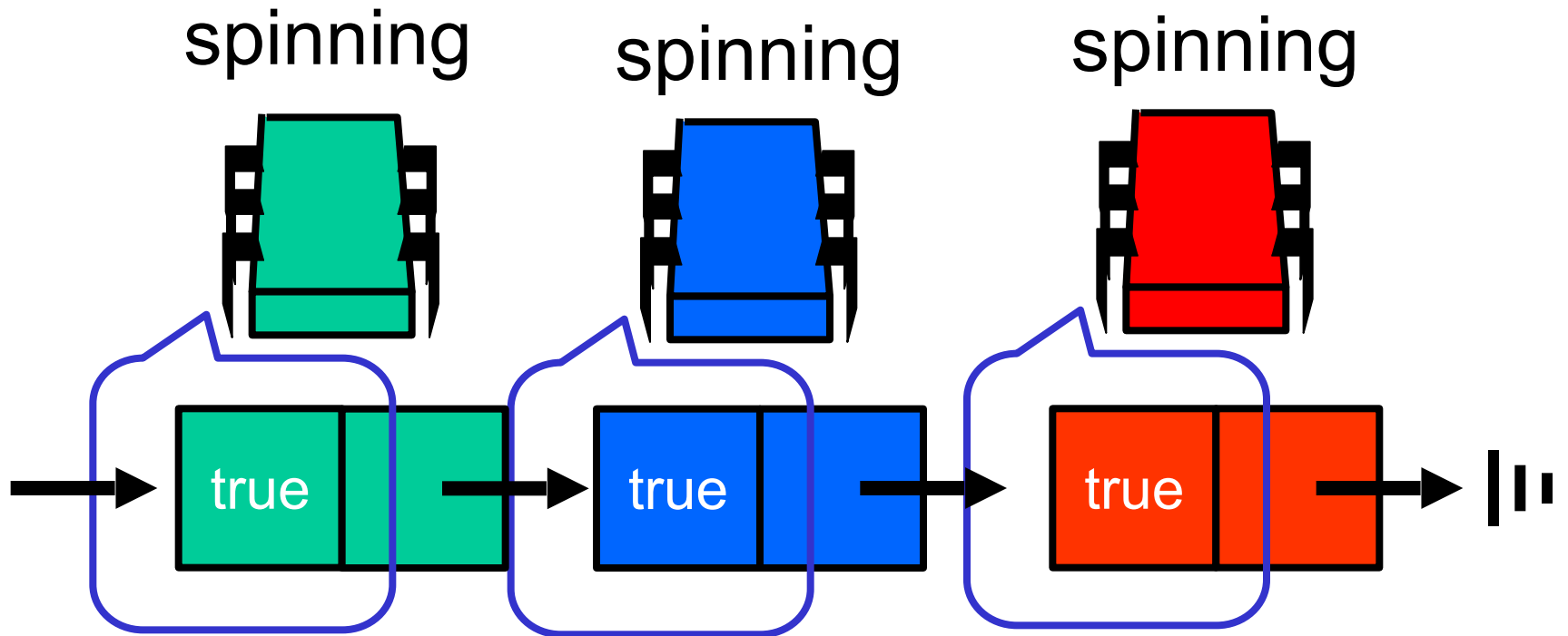


Queue Locks

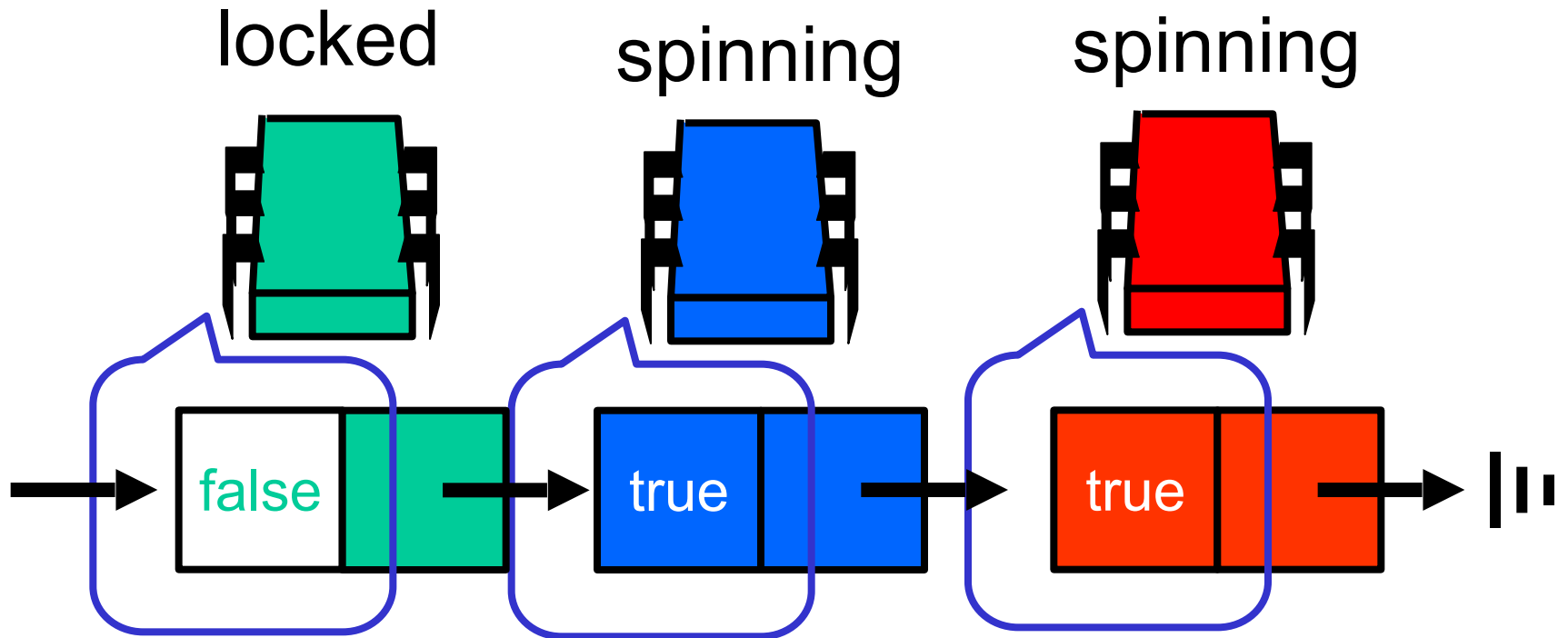
- Can't just quit
 - Thread in line behind will starve
- Need a graceful way out



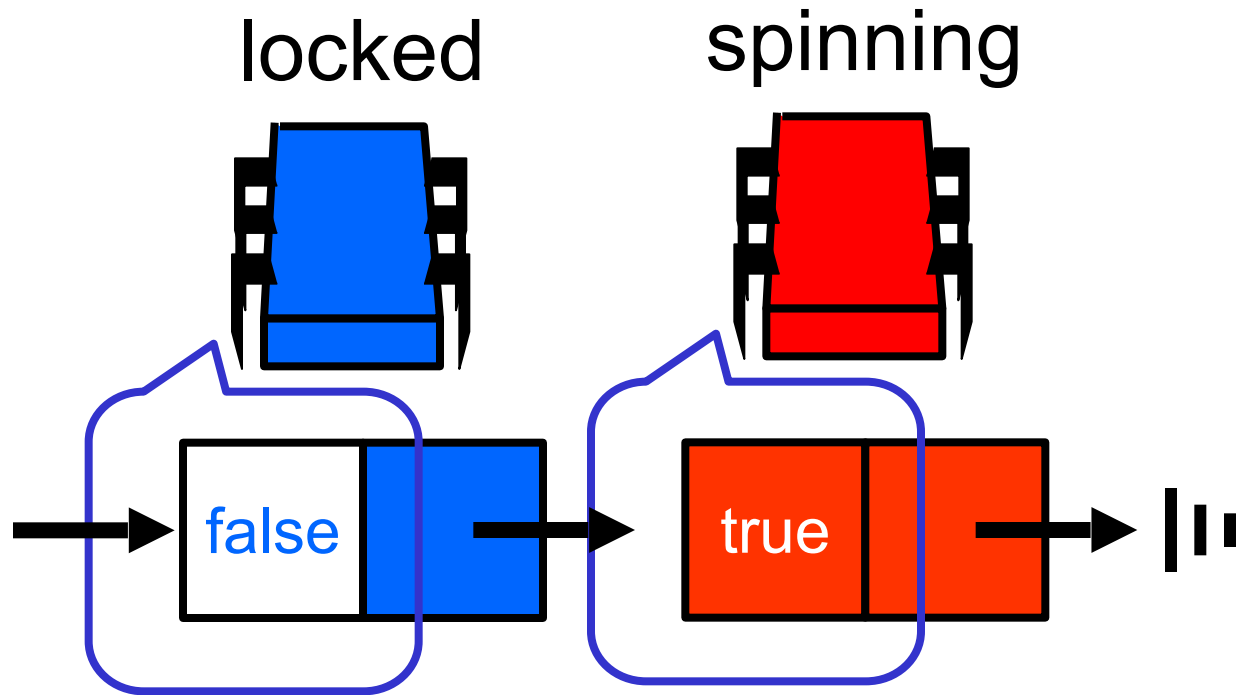
Queue Locks



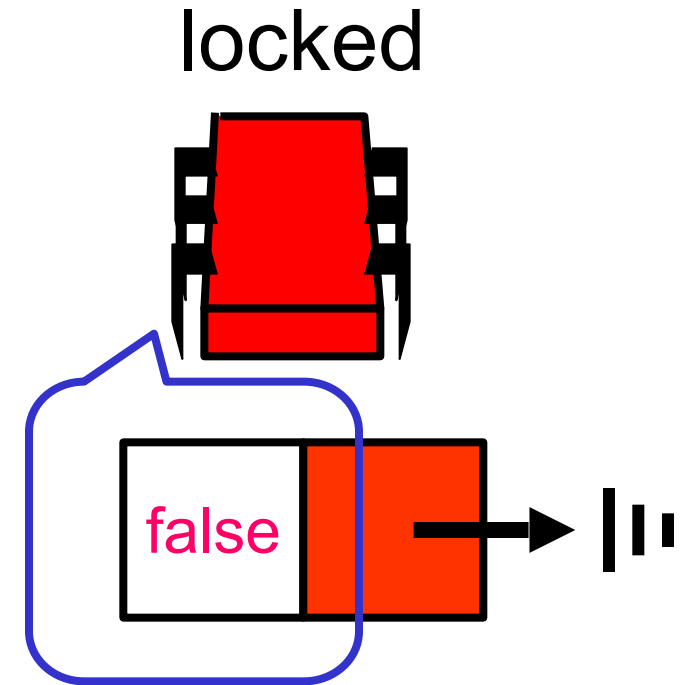
Queue Locks



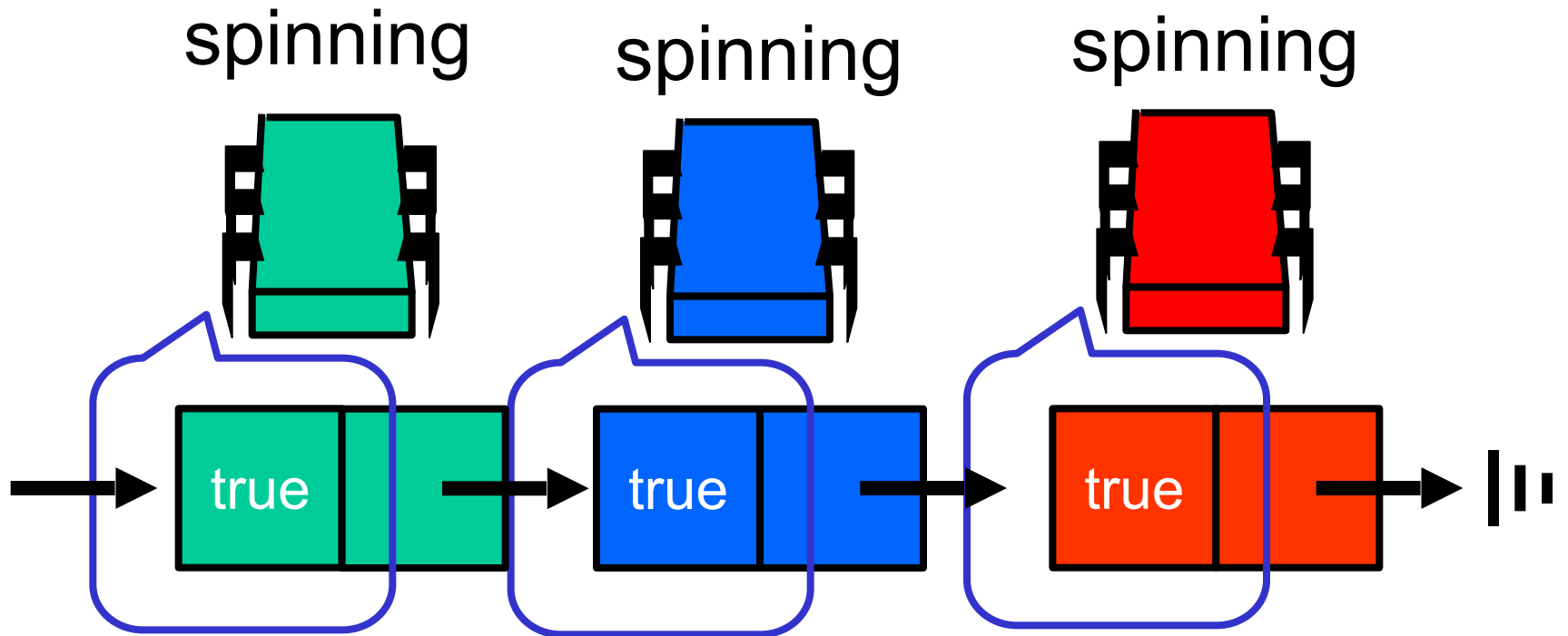
Queue Locks



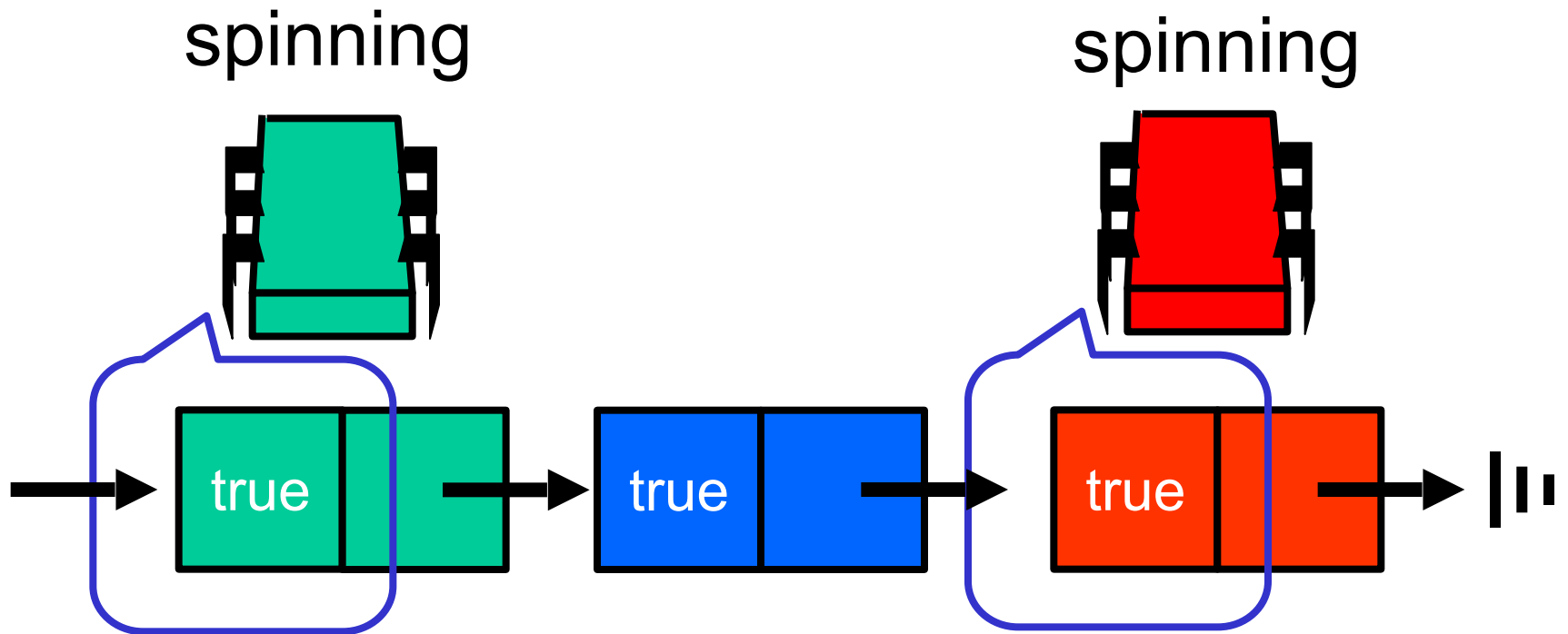
Queue Locks



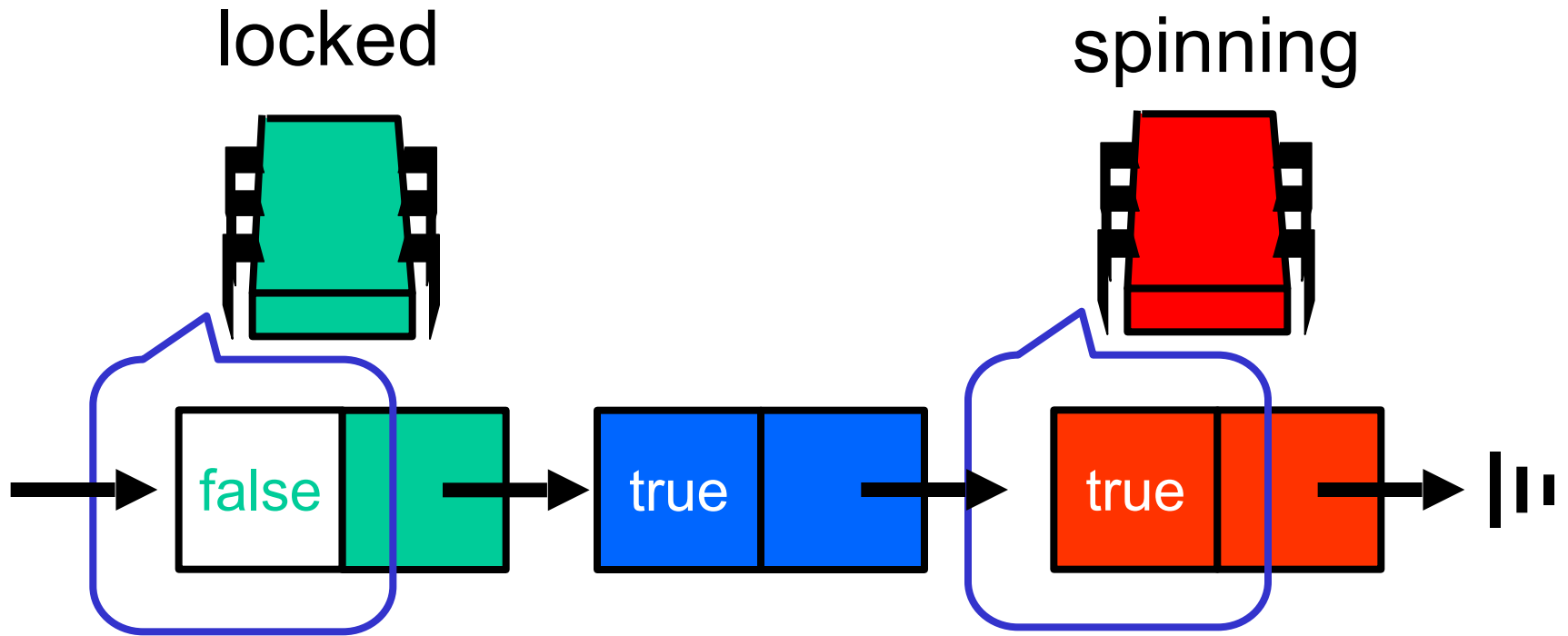
Queue Locks



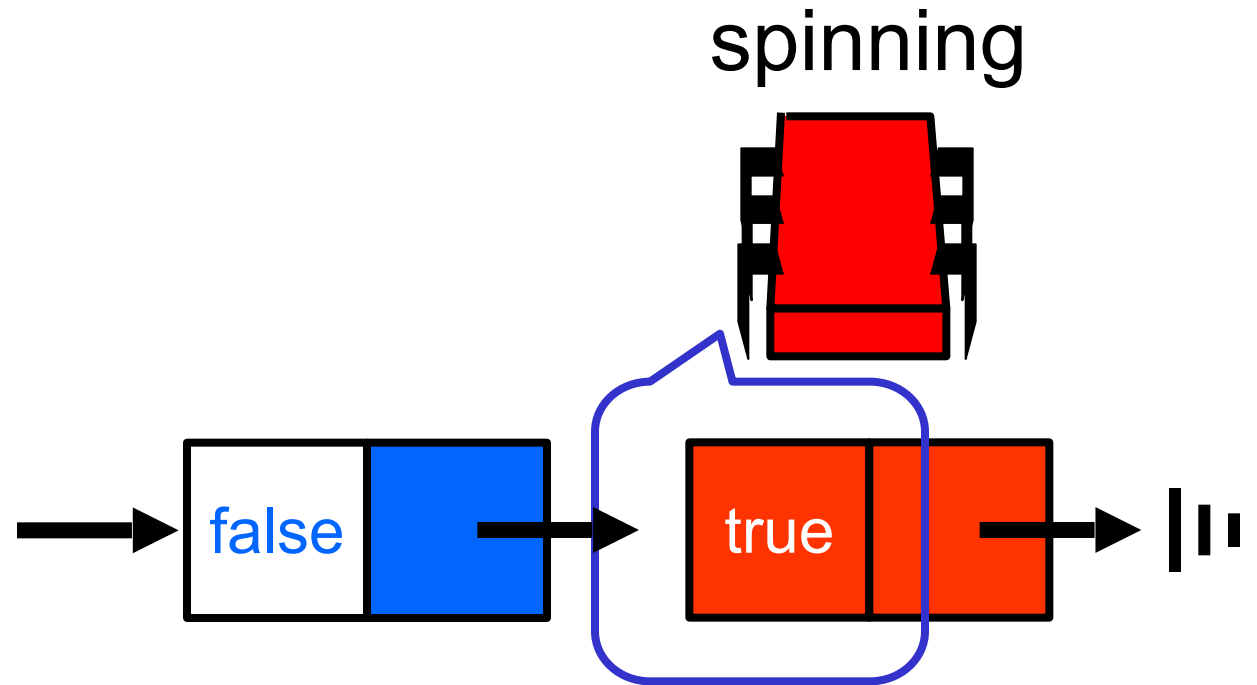
Queue Locks



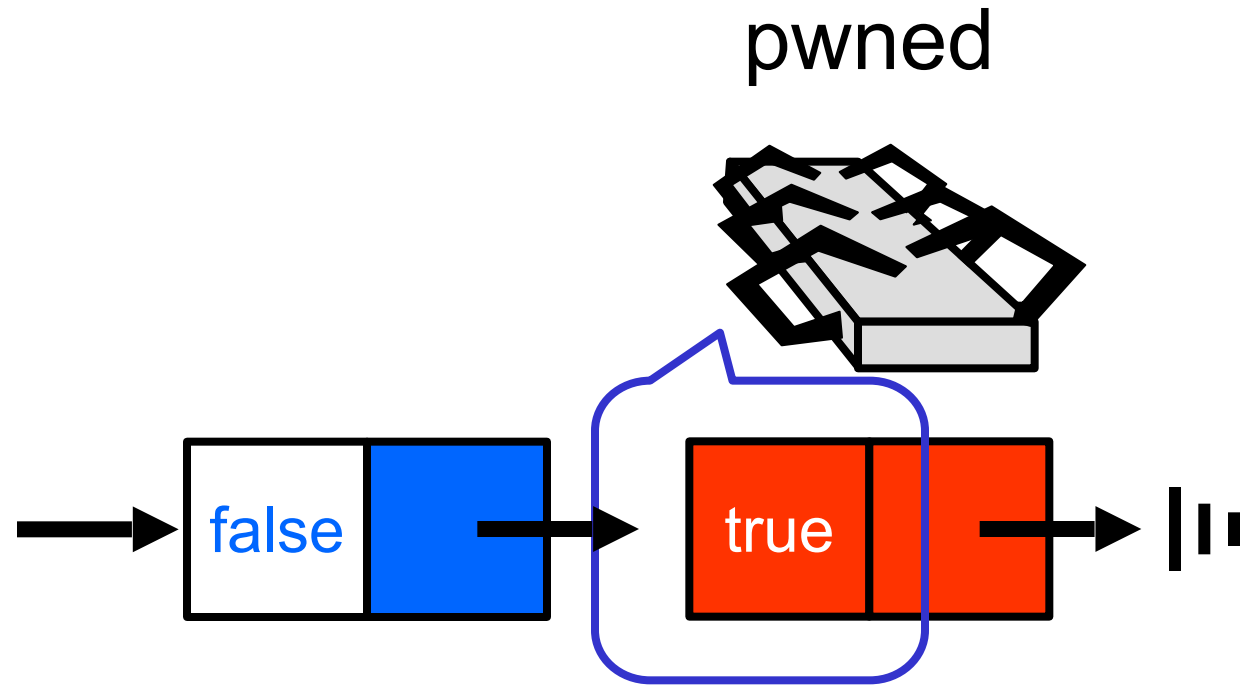
Queue Locks



Queue Locks

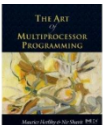


Queue Locks



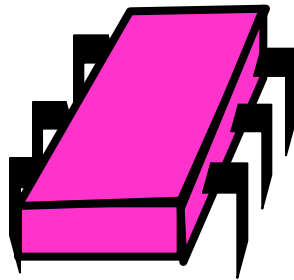
Abortable CLH Lock

- When a thread gives up
 - Removing node in a wait-free (non-locking) way is hard
- Idea:
 - let successor deal with it.



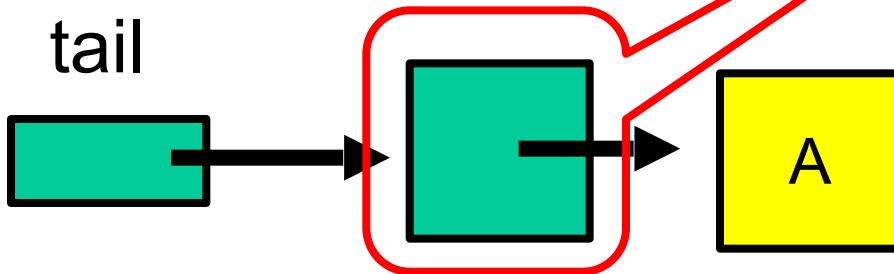
Initially

idle

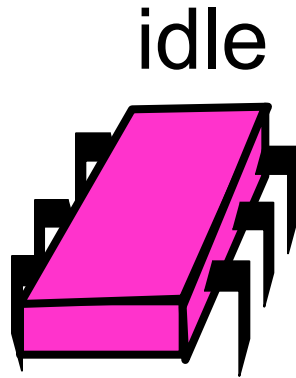


Pointer to
predecessor (or
null)

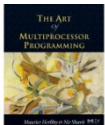
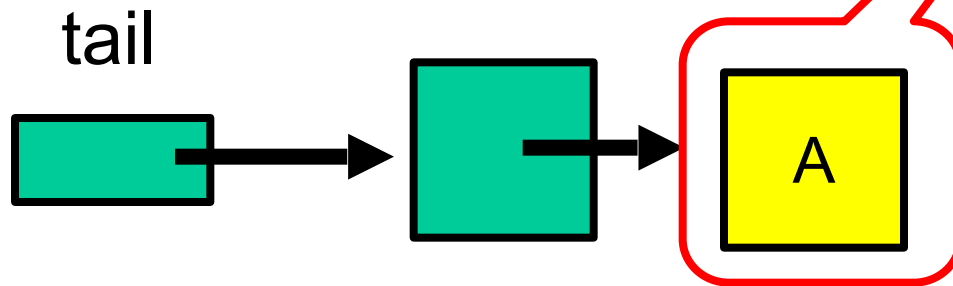
tail



Initially

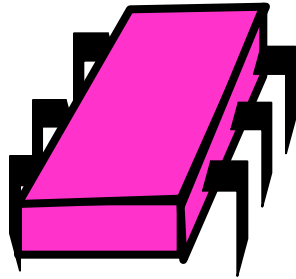


Distinguished
available node
means lock is
free

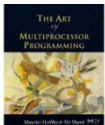
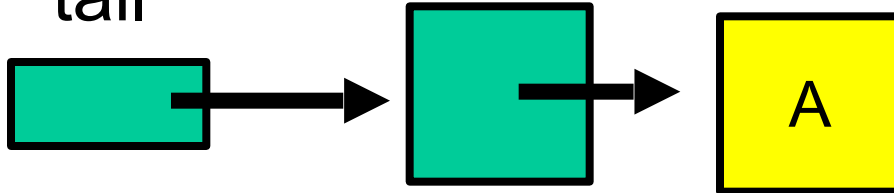


Acquiring

acquiring



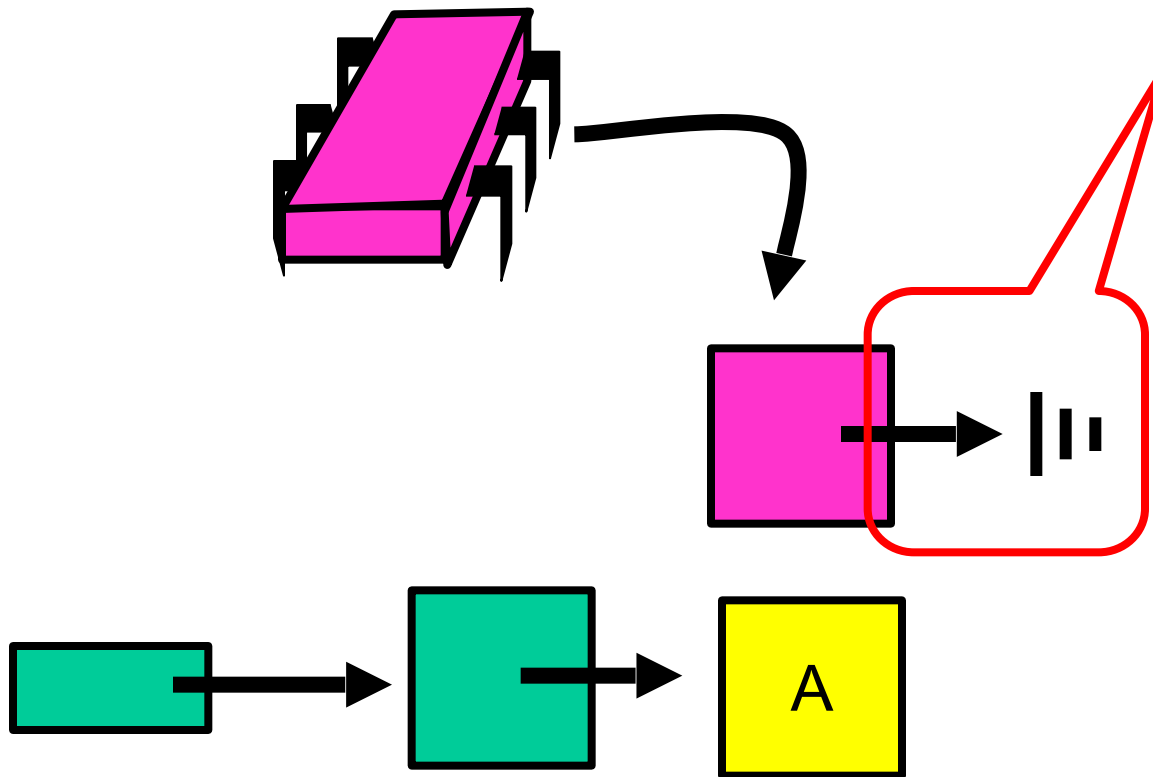
tail



Acquiring

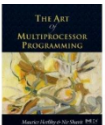
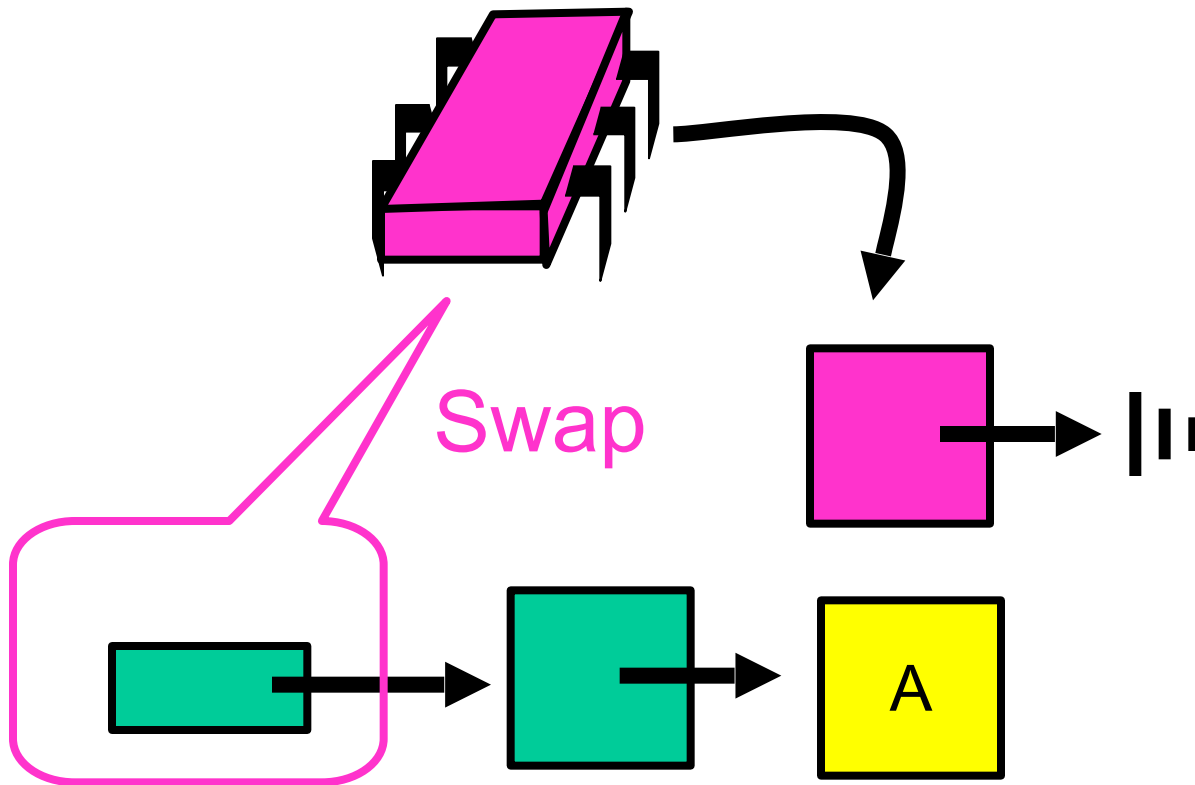
Null predecessor
means lock not
released or aborted

acquiring



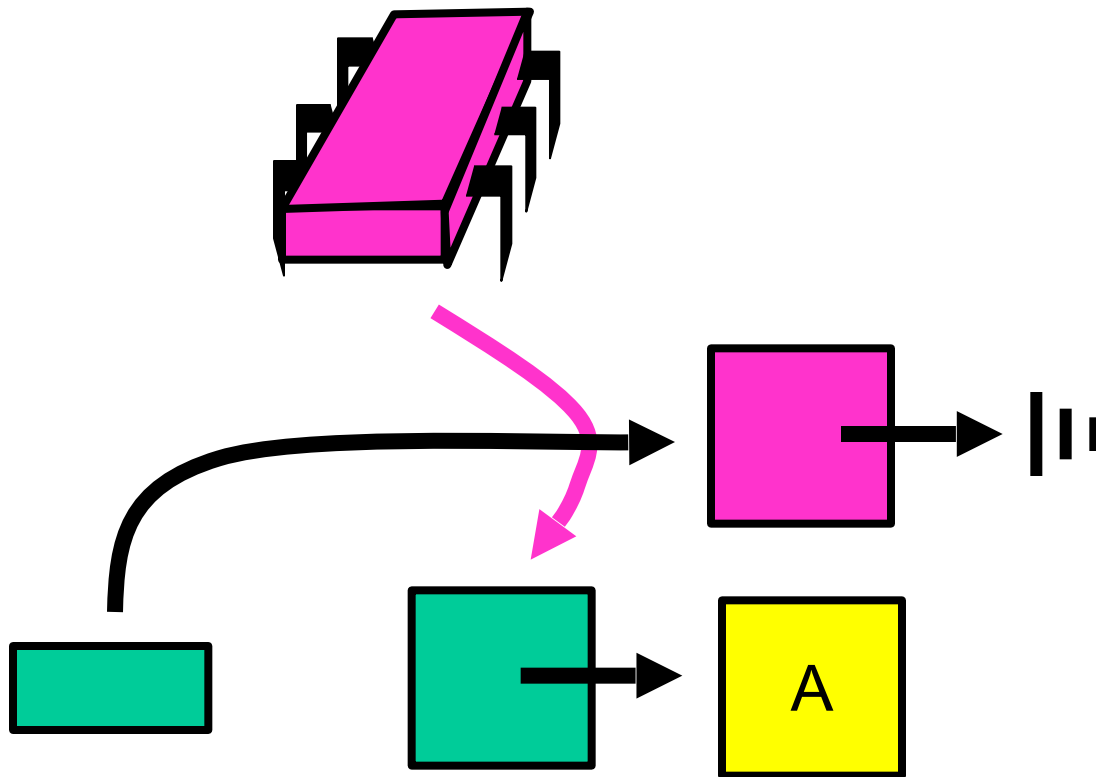
Acquiring

acquiring



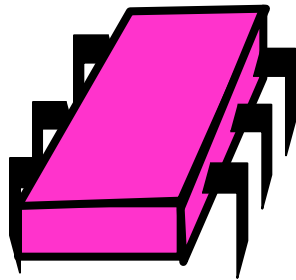
Acquiring

acquiring

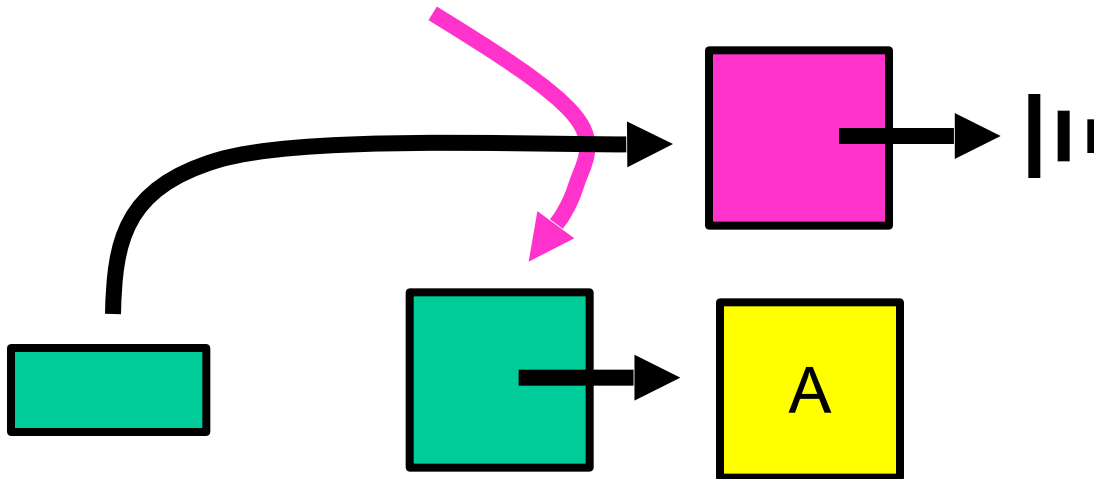


Acquired

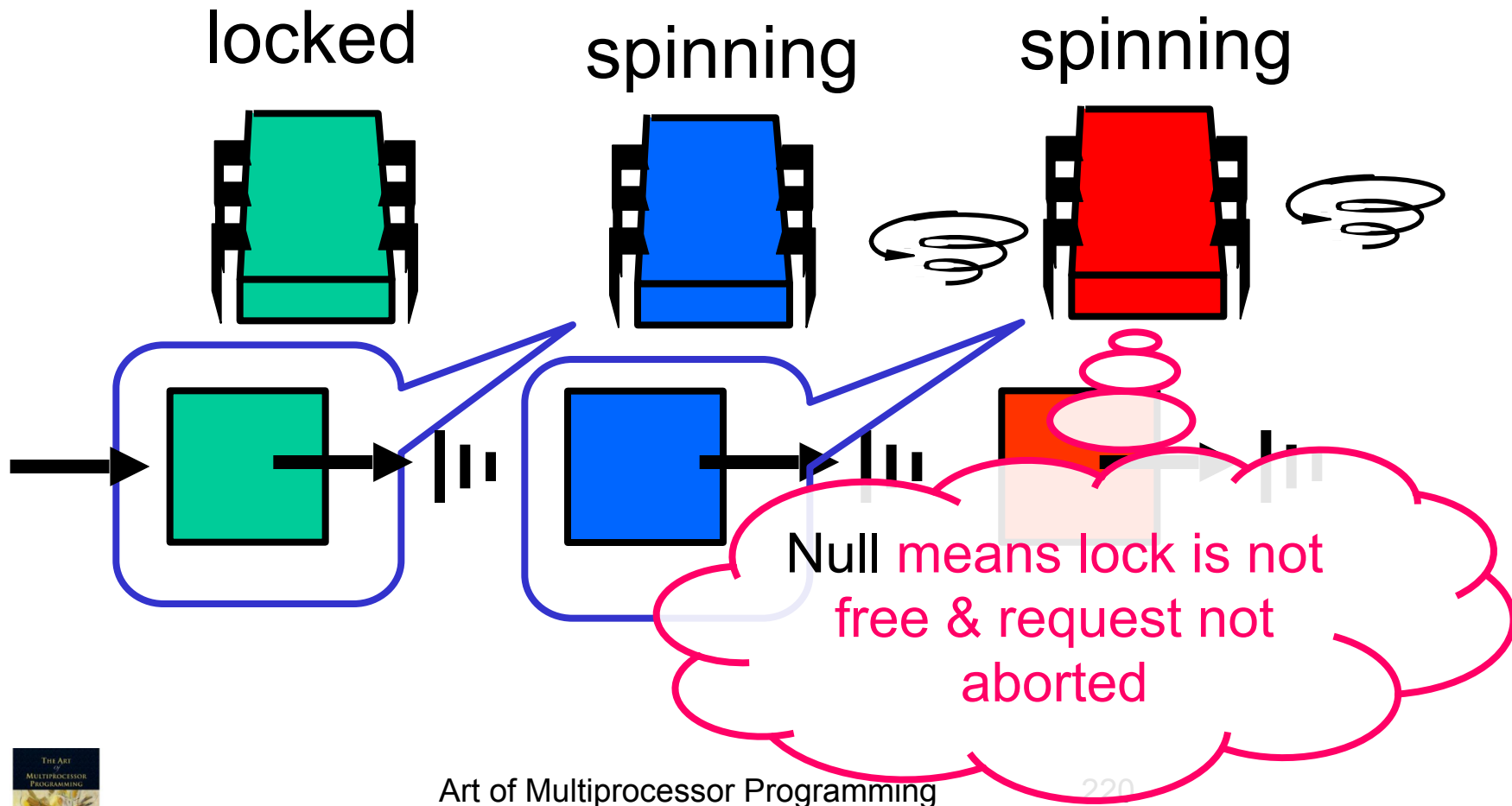
locked



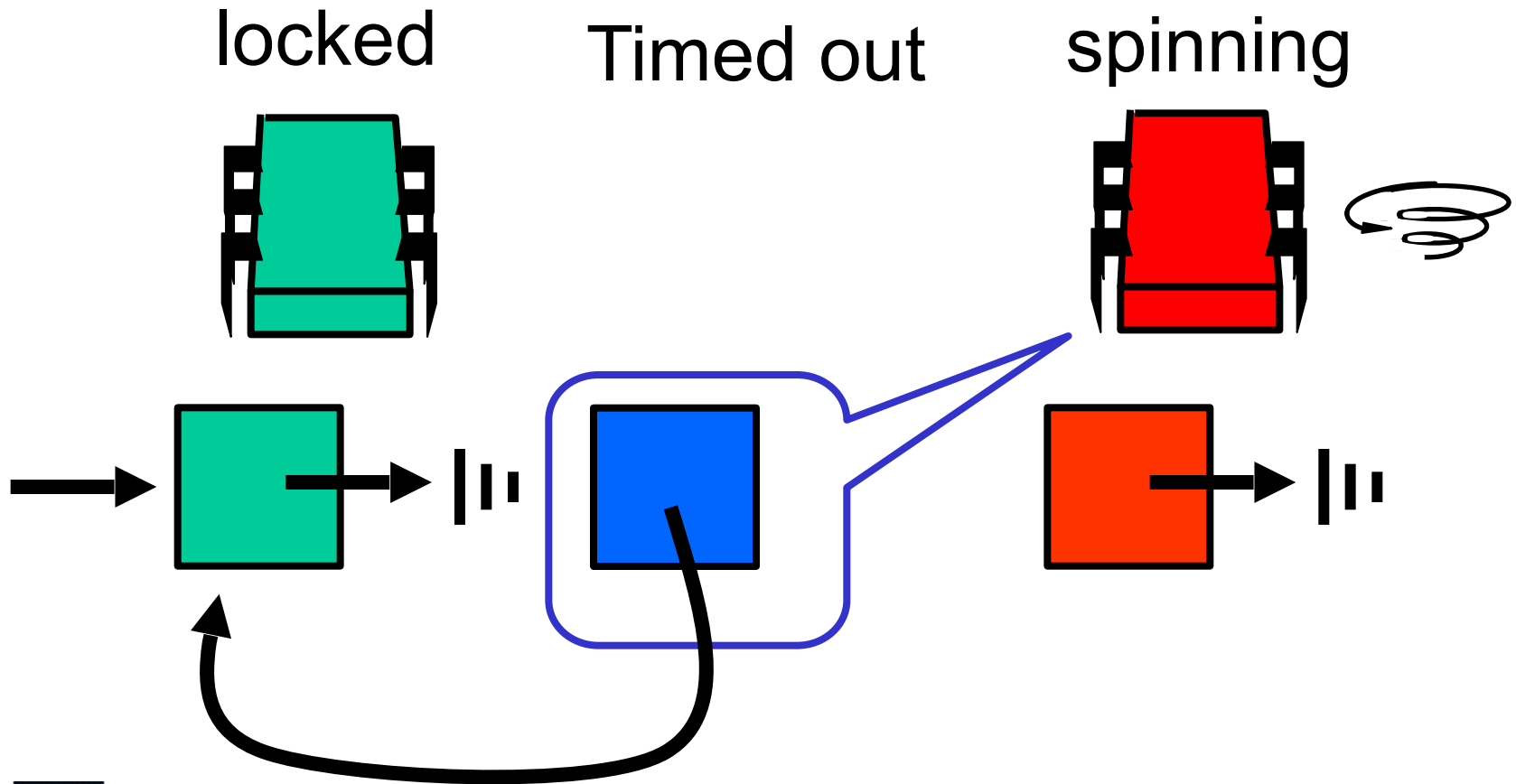
Reference to
AVAILABLE means
lock is free.



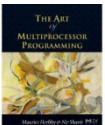
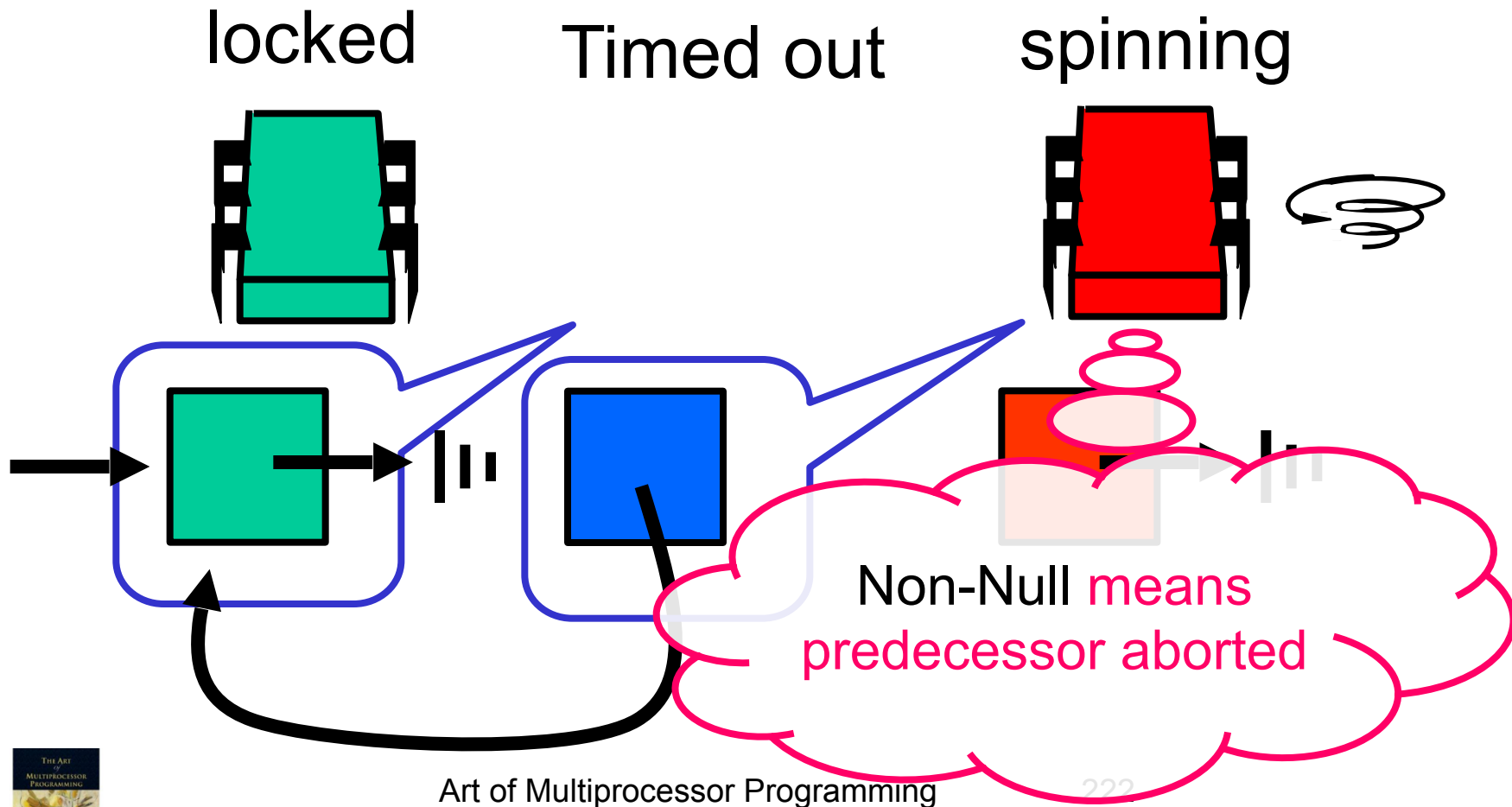
Normal Case



One Thread Aborts

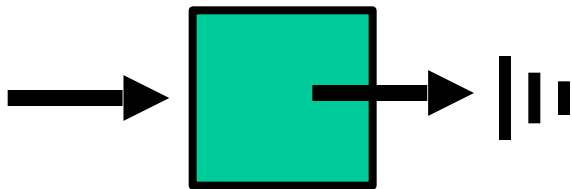
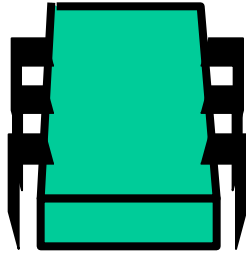


Successor Notices

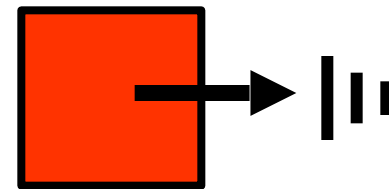
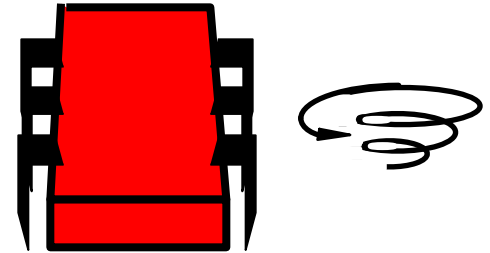


Recycle Predecessor's Node

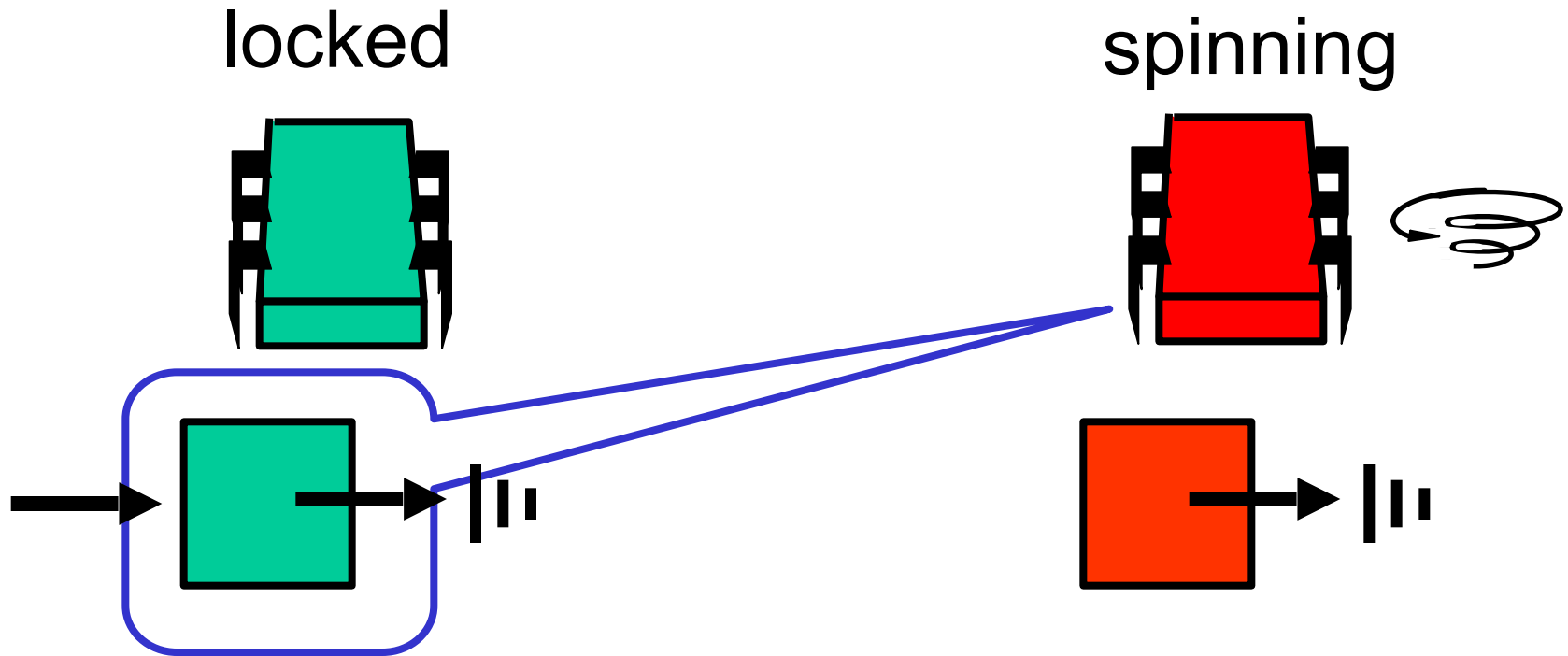
locked



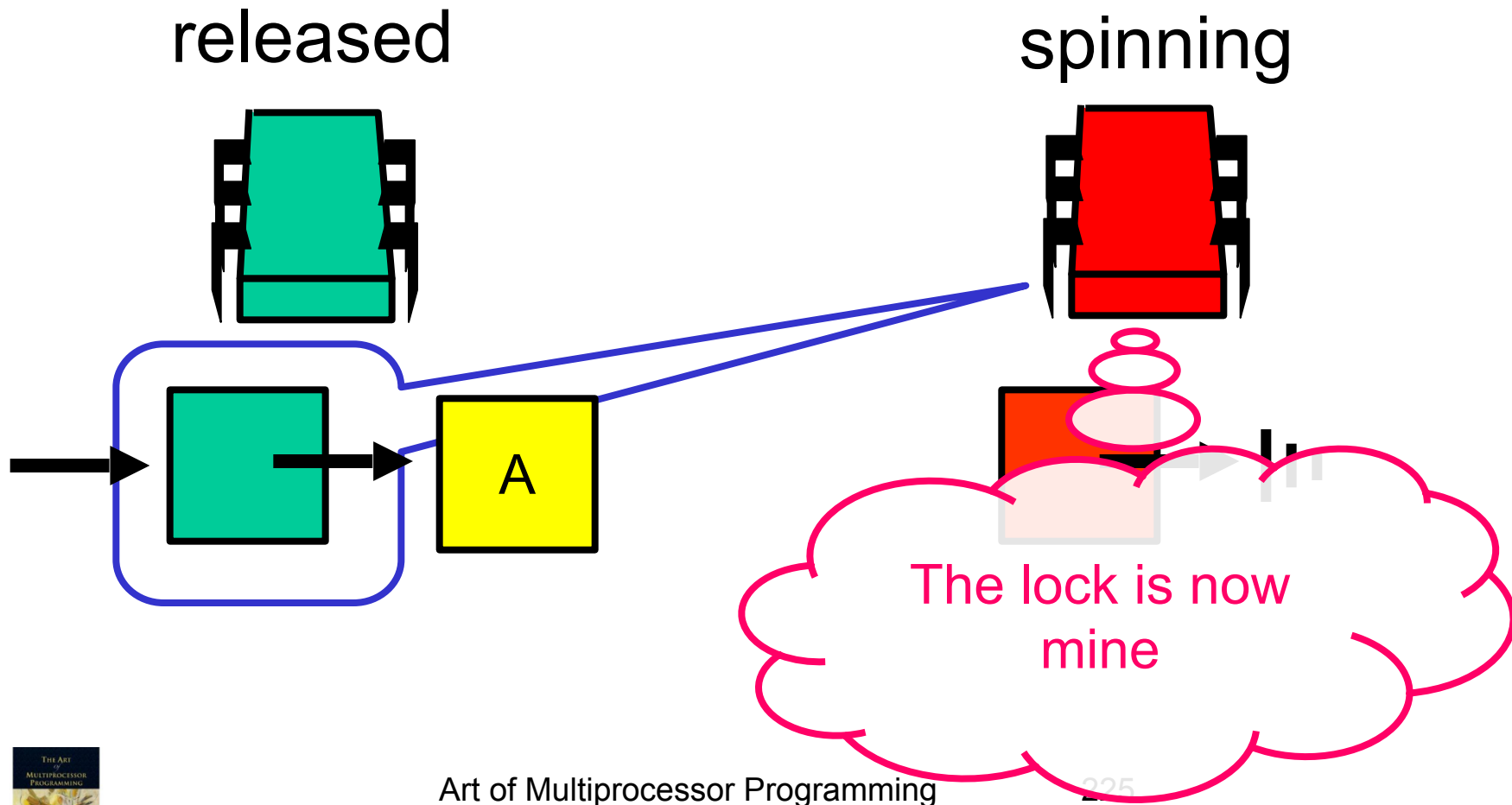
spinning



Spin on Earlier Node

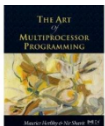


Spin on Earlier Node



Time-out Lock

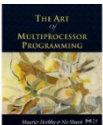
```
public class TOLock implements Lock {  
    static Qnode AVAILABLE  
        = new Qnode();  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode;
```



Time-out Lock

```
public class TOLock implements Lock {  
    static Qnode AVAILABLE  
        = new Qnode();  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode;
```

AVAILABLE node signifies
free lock



Time-out Lock

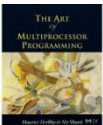
```
public class TOLock implements Lock {  
    static Qnode AVAILABLE  
        = new Qnode();  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode;
```

Tail of the queue

Time-out Lock

```
public class TOLock implements Lock {  
    static Qnode AVAILABLE  
        = new Qnode();  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode;
```

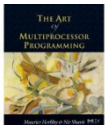
Remember my node ...



Time-out Lock

```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred = tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
}
```

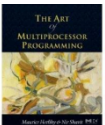
...



Time-out Lock

```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred = tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
}
```

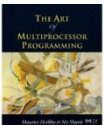
Create & initialize node



Time-out Lock

```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred = tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
}
```

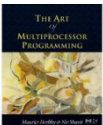
Swap with tail



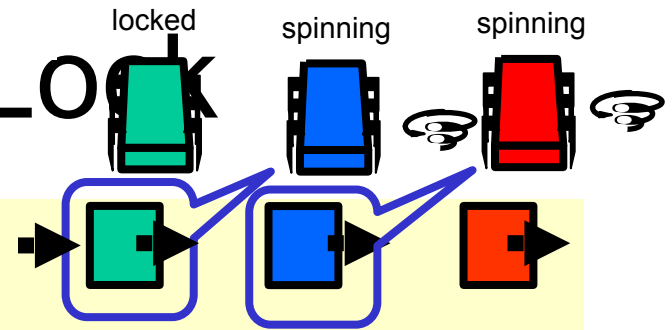
Time-out Lock

```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred = tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
    ...  
}
```

If predecessor absent or
released, we are done



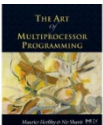
Time-out Lock



...

```
long start = now();
while (now() - start < timeout) {
    Qnode predPred = myPred.prev;
    if (predPred == AVAILABLE) {
        return true;
    } else if (predPred != null) {
        myPred = predPred;
    }
}
```

...



Time-out Lock

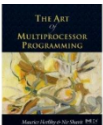
...

```
long start = now();  
while (now() - start < timeout) {
```

```
    Qnode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}
```

...

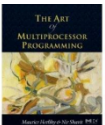
Keep trying for a while ...



Time-out Lock

```
...  
long start = now();  
while (now() - start < timeout) {  
    Qnode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}
```

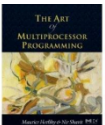
Spin on predecessor's prev
field



Time-out Lock

```
...  
long start = now();  
while (now() - start < timeout) {  
    Qnode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}  
...
```

Predecessor released lock



Time-out Lock

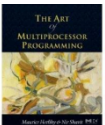
```
...  
long start = now();  
while (now() - start < timeout) {  
    Qnode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}  
...
```

Predecessor aborted,
advance one

Time-out Lock

```
...  
if (!tail.compareAndSet(qnode, myPred))  
    qnode.prev = myPred;  
    return false;  
}  
}
```

What do I do when I time out?



Time-out Lock

```
...  
if (!tail.compareAndSet(qnode, myPred))  
    qnode.prev = myPred;  
return false;  
}  
}
```

Do I have a successor?
If CAS fails, I do.
Tell it about myPred

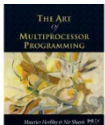
Time-out Lock

```
...  
if (!tail.compareAndSet(qnode, myPred))  
    qnode.prev = myPred;  
    return false;  
}  
}
```

If CAS succeeds: no successor,
simply return false

Time-Out Unlock

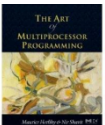
```
public void unlock() {  
    Qnode qnode = myNode.get();  
    if (!tail.compareAndSet(qnode, null))  
        qnode.prev = AVAILABLE;  
}
```



Time-out Unlock

```
public void unlock() {  
    Qnode qnode = myNode.get();  
    if (!tail.compareAndSet(qnode, null))  
        qnode.prev = AVAILABLE;  
}
```

If CAS failed:
successor exists,
notify it can enter



Timing-out Lock

```
public void unlock() {  
    Qnode qnode = myNode.get();  
    if (!tail.compareAndSet(qnode, null))  
        qnode.prev = AVAILABLE;  
}
```

CAS successful: set tail to null, no clean up since no successor waiting

One Lock To Rule Them All?

- TTAS+Backoff, CLH, MCS, ToLock...
- Each better than others in some way
- There is no one solution
- Lock we pick really depends on:
 - the application
 - the hardware
 - which properties are important

