Spin Locks and Contention

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Kinds of Architectures

- SISD (Uniprocessor)
  - Single instruction stream
  - Single data stream
- SIMD (Vector)
  - Single instruction
  - Multiple data
- MIMD (Multiprocessors)
  - Multiple instruction
  - Multiple data.

You can classify processors by how they manage data and instruction streams. A single-instruction single-data stream architecture is just a uniprocessor. A few years ago, single-instruction multiple-data stream architectures were popular, for example the Connection Machine CM2. These have fallen out of favor, at least for the time being.

Kinds of Architectures

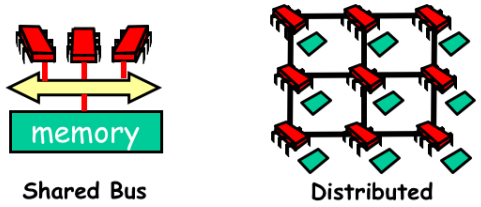
- SISD (Uniprocessor)
  - Single instruction stream
  - Single data stream
- SIMD (Vector)
  - Single instruction
  - Multiple data
- MIMD (Multiprocessors)
  - Multiple instruction
  - Multiple data.

Our space

(1) Art of Multiprocessor Programming 5

Instead, most modern multiprocessors provide *multiple instruction streams*, meaning that processors execute independent sequences of instructions, and *multiple data streams*, meaning that processors issue independent sequences of memory reads and writes. Such architectures are usually called ``MIMD''.

MIMD Architectures



- Memory Contention
- Communication Contention
- Communication Latency

Art of Multiprocessor Programming 6

There are two basic kinds of MIMD architectures. In a *shared bus* architecture, processors and memory are connected by a shared broadcast medium called a *bus* (like a tiny Ethernet). Both the processors and the memory controller can broadcast on the bus. Only one processor (or memory) can broadcast on the bus at a time, but all processors (and memory) can listen. Bus-based architectures are the most common today because they are easy to build.

The principal things that affect performance are (1) contention for the memory: not all the processors can usually get at the same memory location at the same times, and if they try, they will have to queue up, (2) contention for the communication medium. If everyone wants to communicate at the same time, or to the same processor, then the processors will have to wait for one another. Finally, there is a growing *communication latency*, the time it takes for a processor to communicate with memory or with another processor.

Today: Revisit Mutual Exclusion

- Think of performance, not just correctness and progress
- Begin to understand how performance depends on our software properly utilizing the multiprocessor machine's hardware
- And get to know a collection of locking algorithms…

When programming uniprocessors, one can generally ignore the exact structure

and properties of the underlying system. Unfortunately, multiprocessor

programming has yet to reach that state, and at present it is crucial that

the programmer understand the properties and limitations of the underlying

machine architecture. This will be our goal in this chapter. We revisit

the familiar mutual exclusion problem, this time with the aim of devising

mutual exclusion protocols that work well on today's multiprocessors.

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

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

Any mutual exclusion protocol poses the question: what do you do if you cannot acquire the lock? There are two alternatives. If you keep trying, the lock is called a *spin lock*, and repeatedly testing the lock is called *spinning*. or *busy-waiting*. We will use these terms interchangeably. The **filter** and **bakery** algorithms are spin-locks. Spinning is sensible when you expect the lock delay to be short. For obvious reasons, spinning makes sense only on multiprocessors. The alternative is to suspend yourself and ask the operating system's scheduler to schedule another thread on your processor, which is sometimes called *blocking*. Java's built-in synchronization is blocking. Because switching from one thread to another is expensive, blocking makes sense only if you expect the lock delay to be long. Many operating systems mix both strategies, spinning for a short time and then blocking.

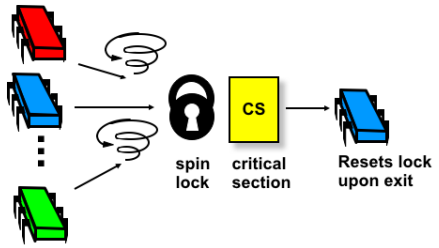
What Should you do if you can't get a lock?

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

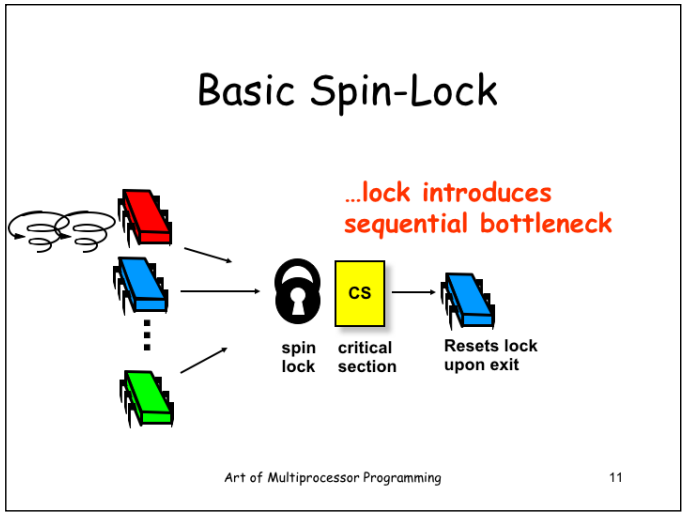
our focus

Both spinning and blocking are important techniques. In this lecture, we turn our attention to locks that use spinning.

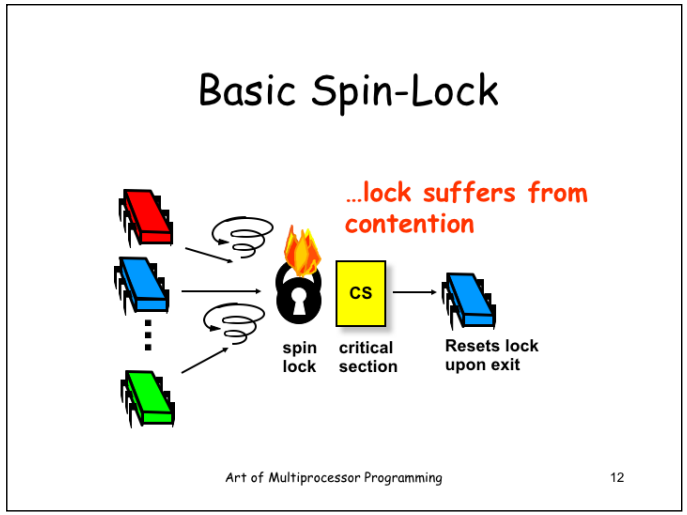
Basic Spin-Lock



With spin locks, synchronization usually looks like this. Some set of threads *contend* for the lock. One of them *acquires* it, and the others spin. The winner enters the critical section, does its job, and *releases* the lock on exit.

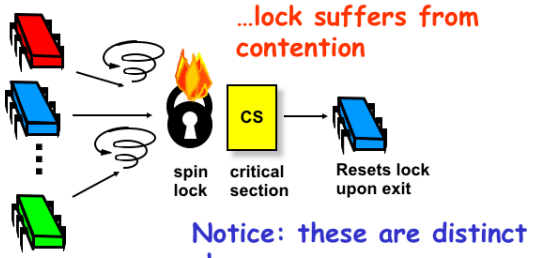


Contention occurs when multiple threads try to acquire a lock at the same time. *High contention* means there are many such threads, and *low contention* means the opposite.



Contention occurs when multiple threads try to acquire a lock at the same time. *High contention* means there are many such threads, and *low contention* means the opposite.

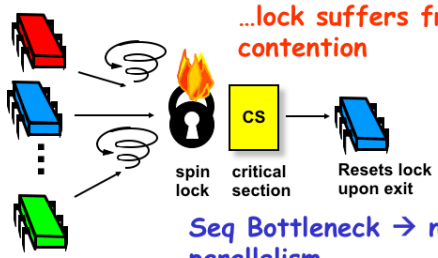
Basic Spin-Lock



Notice: these are distinct phenomena

Our goal is to understand how contention works, and to develop a set of techniques that can avoid or alleviate it. These techniques provide a useful toolkit for all kinds of synchronization problems.

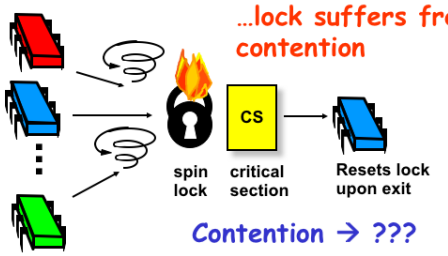
Basic Spin-Lock



Seq Bottleneck → no parallelism

Our goal is to understand how contention works, and to develop a set of techniques that can avoid or alleviate it. These techniques provide a useful toolkit for all kinds of synchronization problems.

Basic Spin-Lock



Our goal is to understand how contention happens, and to develop a set of techniques that can avoid or alleviate it. These techniques provide a useful toolkit for all kinds of synchronization problems.

Review: Test-and-Set

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

The **test-and-set** machine instruction atomically stores *true* in that word, and returns that word's previous value, *swapping* the value *true* for the word's current value. You can reset the word just by writing *false* to it. Note in Java TAS is called getAndSet. We will use the terms interchangeably.

Review: Test-and-Set

```
public class AtomicBoolean {
 boolean value;

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


The **test-and-set** machine instruction, with consensus number two, was the principal synchronization instruction provided by many early multiprocessors.

Review: Test-and-Set

```
public class AtomicBoolean {
 boolean value;

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

Package
java.util.concurrent.atomic

Here we are going to build one out of the **atomic Boolean** class, which is provided as part of Java's standard library of atomic primitives. You can think of this object as a box holding a Boolean value.

Review: Test-and-Set

```
public class AtomicBoolean {
 boolean value;

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

Swap old and new values

The **getAndSet** method swaps a Boolean value with the current contents of the box.

Review: Test-and-Set

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


At first glance, the **AtomicBoolean** class seems ideal for implementing a spin lock.

Review: Test-and-Set

```
AtomicBoolean lock
 = new AtomicBoolean(false)

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

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

If we call **getAndSet(*true*)**, then we have a **test-and-set**.

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

The lock is free when the word's value is *false*, and busy when it is *true*.

Test-and-set Lock

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

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

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


Here is what it looks like in more detail.

Test-and-set Lock

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

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

 void unlock() {
  state
 }}
```

Lock state is AtomicBoolean

The lock is just an **AtomicBoolean** initialized to *false*.

Test-and-set Lock

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

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

 void unlock() {
  sta
 }}
```

Keep trying until lock acquired

The lock() method repeatedly applies test-and-set to the location until that instruction returns *false* (that is, until the lock is free).

Test-and-set Lock

```
class TA
 AtomicB
  new At

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

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

Release lock by resetting state to false

The unlock() method simply writes the value *false* to that word.

Space Complexity

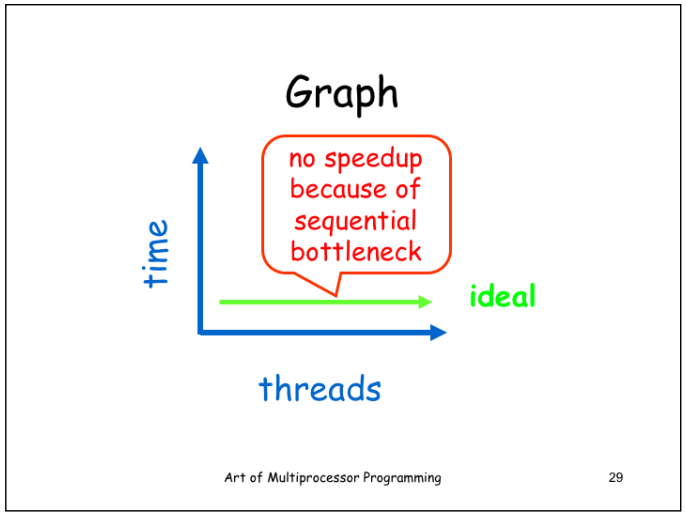
- TAS spin-lock has small "footprint"
- N thread spin-lock uses O(1) space
- As opposed to O(n) Peterson/Bakery
- How did we overcome the Ω(n) lower bound?
- We used a RMW operation…

We call real world space complexity the "footprint". By using TAS we are able to reduce the footprint from linear to constant.

Performance

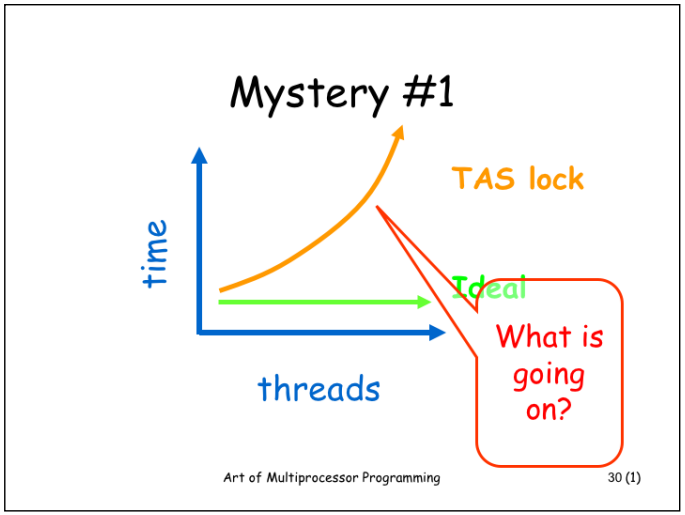
- Experiment
  - n threads
  - Increment shared counter 1 million times
- How long should it take?
- How long does it take?

Let's do an experiment on a real machine. Take *n* threads and have them collectively acquire a lock, increment a counter, and release the lock. Have them do it collectively, say, one million times. Before we look at any curves, let's try to reason about how long it *should* take them.



Ideally the curve should stay flat after that. Why? Because we have a sequential bottleneck so no matter

How many threads we add running in parallel, we will not get any speedup (remember Amdahl's law).



The curve for TAS lock looks like this. In fact, if you do the experiment you have to give up because it takes so long beyond a certain number of processors? What is happening? Let us try this again.

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

Let's try a slightly different approach. Instead of repeatedly trying to **test-and-set** the lock, let's split the locking method into two phases. In the *lurking* phase, we wait until the lock looks like it's free, and when it's free, we *pounce*, attempting to acquire the lock by a call to **test-and-set**. If we win, we're in, if we lose, we go 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;
 }
}
```


Here is what the code looks like.

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

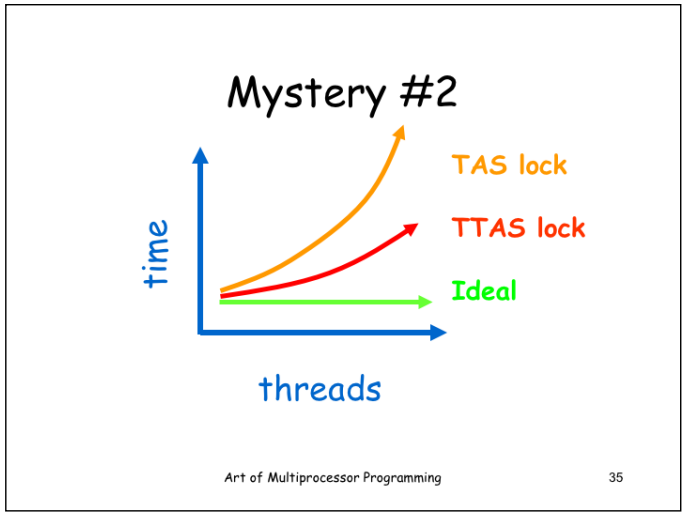
First we spin on the value, repeatedly reading it until it looks like the lock is free. We don't try to modify it, we just read it.

Test-and-test-and-set Lock

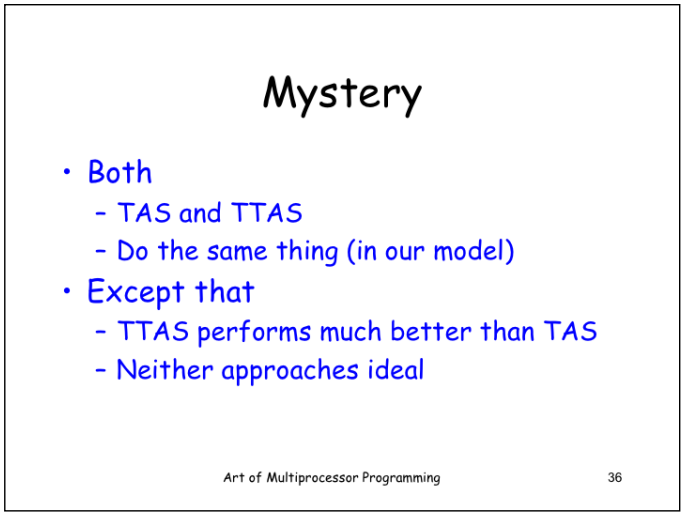
```
class TTASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

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

Then try to acquire it

As soon as it looks like the lock is free, we call **test-and-set** to try to acquire it. If we are first and we succeed, the lock() method returns, and otherwise, if someone else got there before us, we go back to lurking, to repeatedly rereading the variable.



The difference is dramatic. The TTAS lock performs much better than the TAS lock, but still much worse than we expected from an ideal lock.



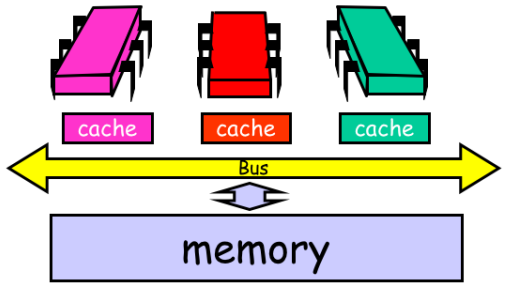
There are two mysteries here: why is the TTAS lock so good (that is, so much better than TAS), and why is it so bad (so much worse than ideal)?

Opinion

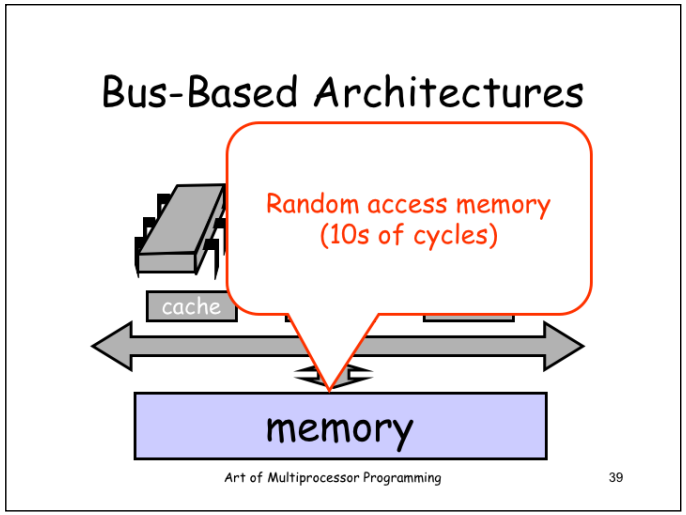
- Our memory abstraction is broken
- TAS & TTAS methods
  - Are provably the same (in our model)
  - Except they aren't (in field tests)
- Need a more detailed model …

From everything we told you in the earlier section on mutual exclusion, you would expect the TAS and TTAS locks to be the same --- after all, they are *logically* equivalent programs. In fact, they are equivalent with respect to *correctness* (they both work), but very different with respect to *performance*. The problem here is that the shared memory abstraction is broken with respect to performance. If you don't understand the underlying architecture, you will never understand why your reasonable-looking programs are so slow. It's a little like not being able to drive a car unless you know how an automatic transmission works, but there it is.

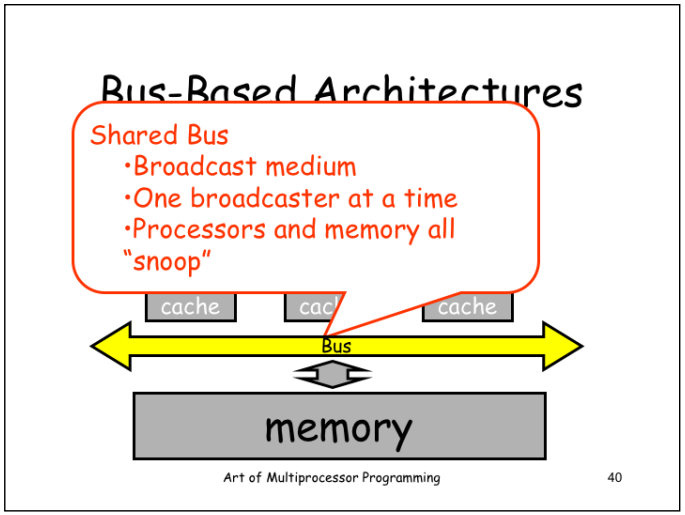
Bus-Based Architectures



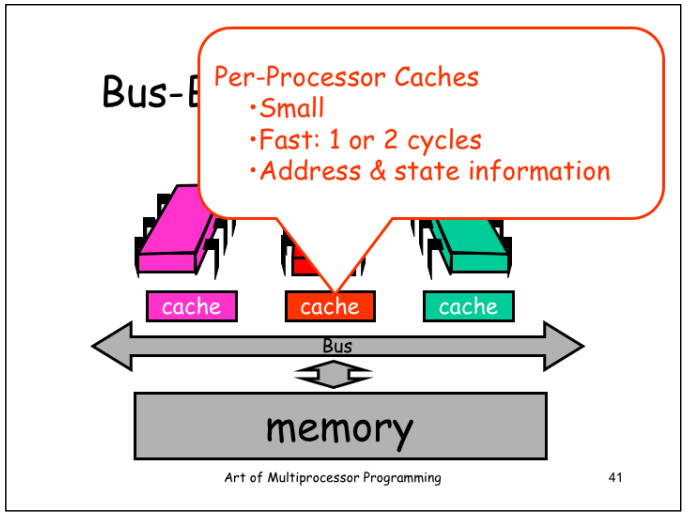
We are going to do yet another review of multiprocessor architecture.



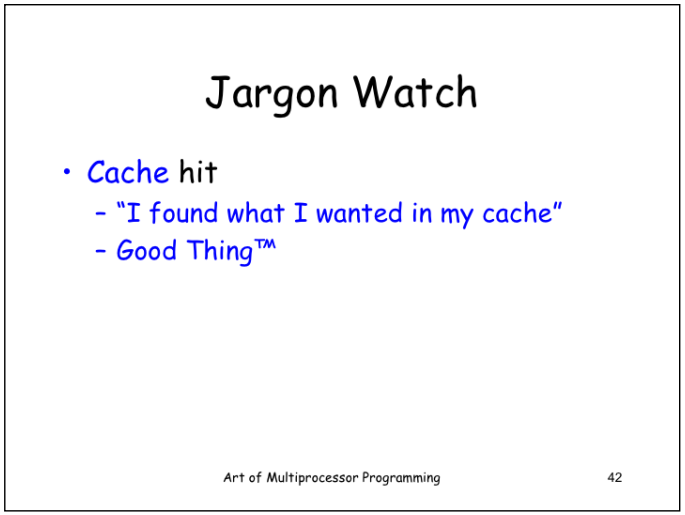
The processors share a memory that has a high latency, say 50 to 100 cycles to read or write a value. This means that while you are waiting for the memory to respond, you can execute that many instructions.



Processors communicate with the memory and with one another over a shared bus. The bus is a broadcast medium, meaning that only one processor at a time can send a message, although everyone can passively listen.



Each processor has a *cache*, a small high-speed memory where the processor keeps data likely to be of interest. A cache access typically requires one or two machine cycles, while a memory access typically requires tens of machine cycles. Technology trends are making this contrast more extreme: although both processor cycle times and memory access times are becoming faster, the cycle times are improving faster than the memory access times, so cache performance is critical to the overall performance of a multiprocessor architecture.



If a processor finds data in its cache, then it doesn't have to go all the way to memory. This is a very good thing, which we call a *cache hit*.

Jargon Watch

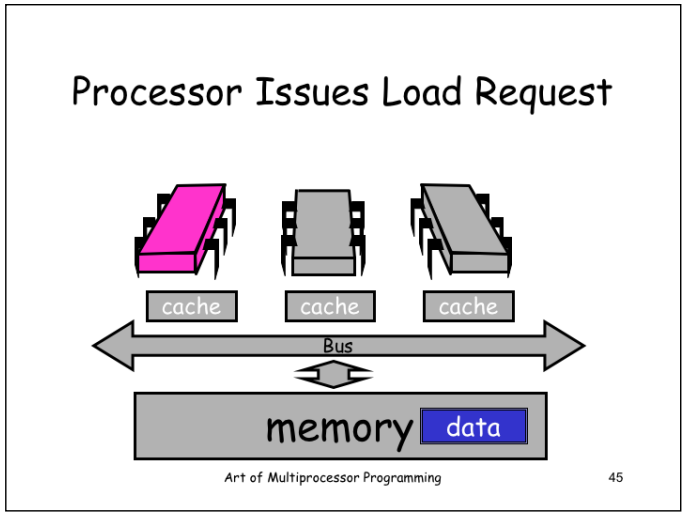
- Cache hit
  - "I found what I wanted in my cache"
  - Good Thing™
- Cache miss
  - "I had to shlep all the way to memory for that data"
  - Bad Thing™

If the processor doesn't find what it wants in its cache, then we have a *cache miss*, which is very time-consuming. How well a synchronization protocol or concurrent algorithm performs is largely determined by its cache behavior: how many hits and misses.

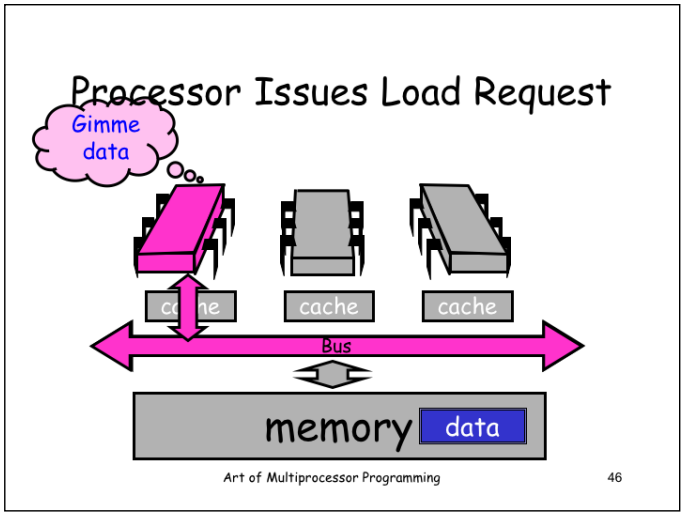
Cave Canem

- This model is still a simplification
  - But not in any essential way
  - Illustrates basic principles
- Will discuss complexities later

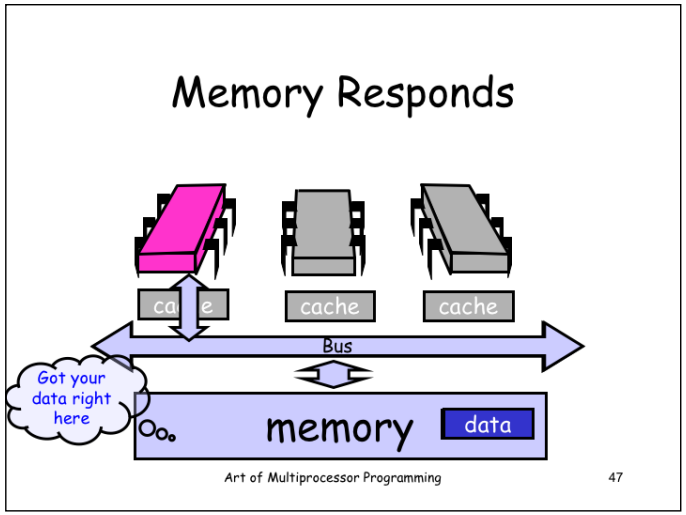
Real cache coherence protocols can be very complex. For example, modern multiprocessors have multi-level caches, where each processor has an on-chip \textit{level-one} (L1) cache, and clusters of processors share a \textit{level-two} (L2) cache. The L2 cache is on-chip in some modern architectures, and off chip in others, a detail that greatly changes the observed performance. We are going to avoid going into too much detail here because the basic principles don't depend on that level of detail.



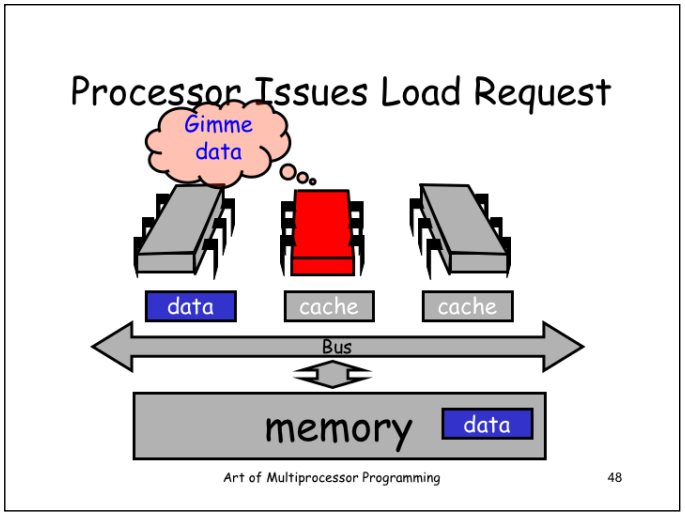
Here is one thing that can happen when a processor issues a load request.



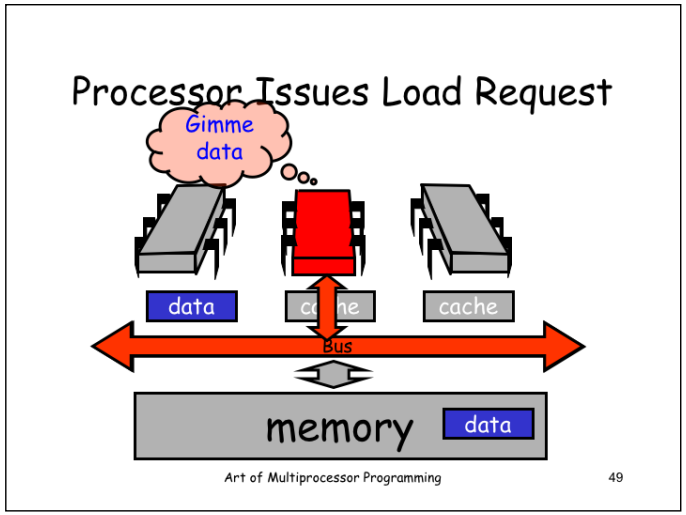
It broadcasts a message asking for the data it needs. Notice that while it is broadcasting, no one else can use the bus.



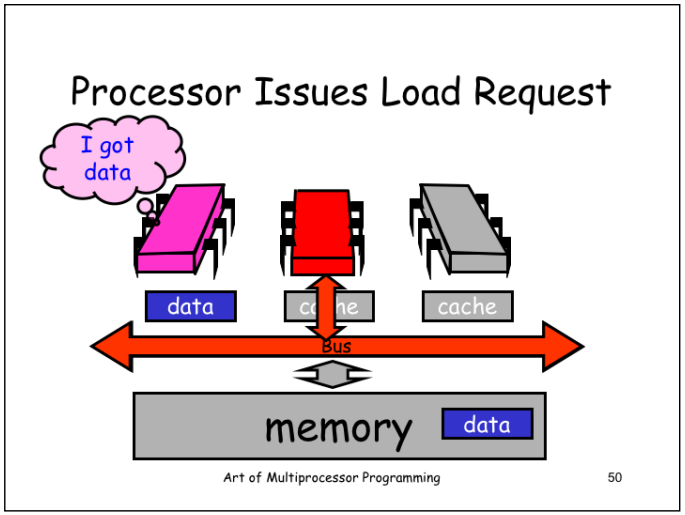
In this case, the memory responds to the request, also over the bus.



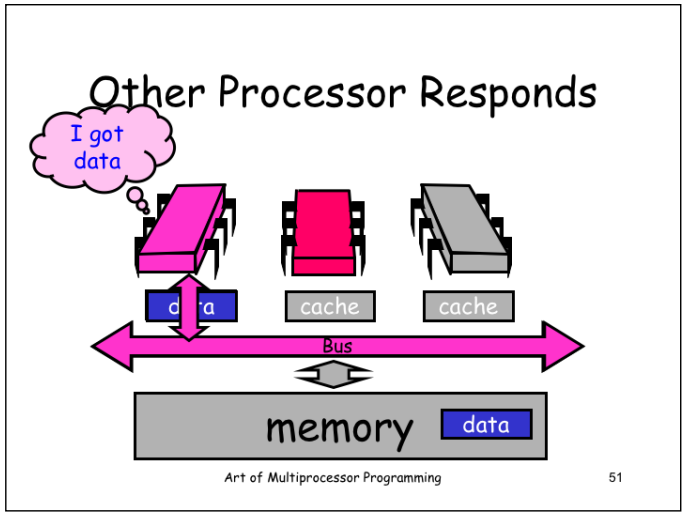
Now suppose another processor issues a load request for the same data.



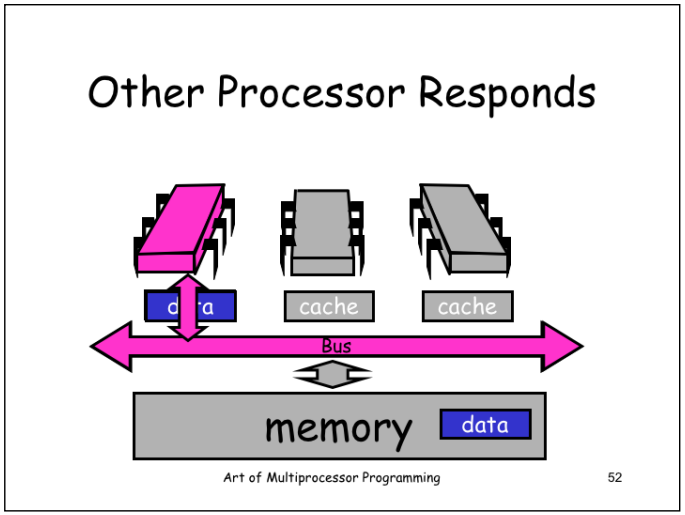
It broadcasts its request over the bus.



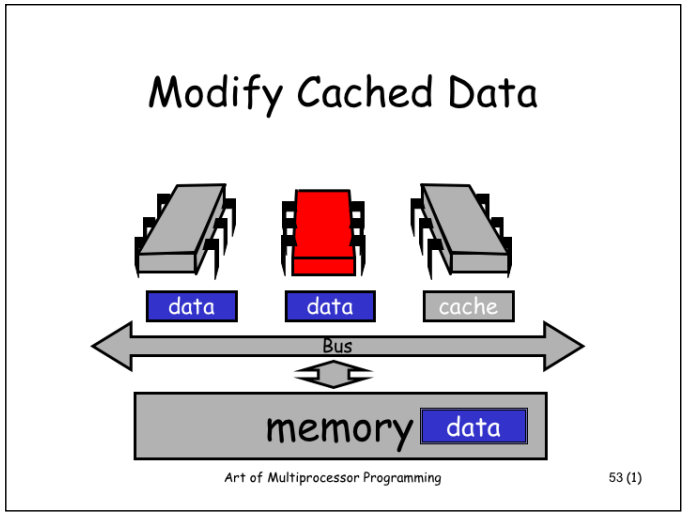
This time, however, the request is picked up by the first processor, which has the data in its cache. Usually, when a processor has the data cached, it, rather than the memory, will respond to load requests.



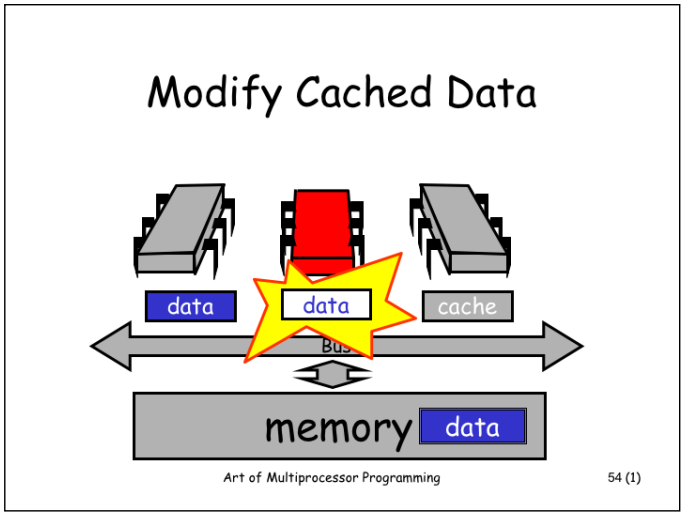
The processor puts the data on the bus.



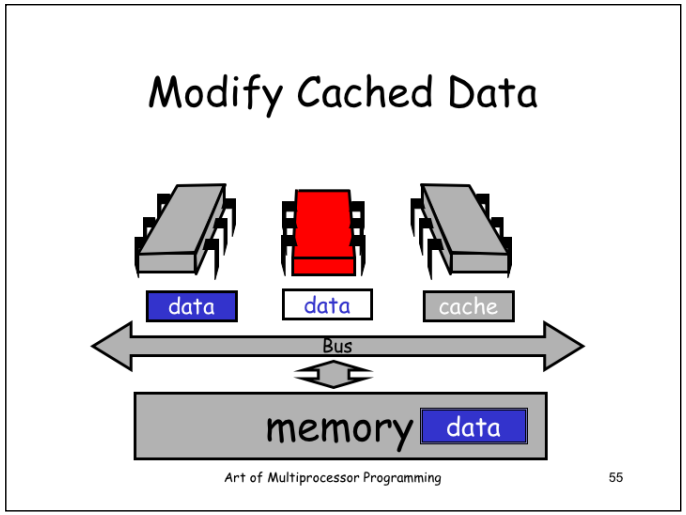
Now both processors have the same data cached.

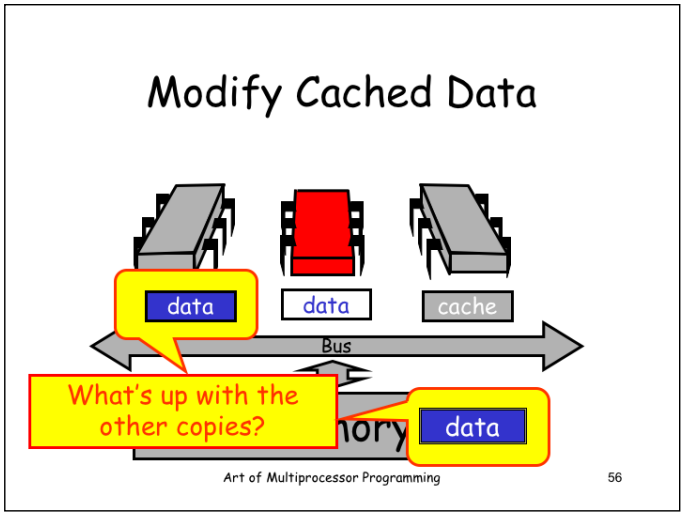


Now what happens if the red processor decides to modify the cached data?



It changes the copy in its cache (from blue to white).





Now we have a problem. The data cached at the red processor disagrees with the same copy of that memory location stored both at the other processors and in the memory itself.

Cache Coherence

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

The problem of keeping track of multiple copies of the same data is called the cache coherence problem, and ways to accomplish it are called cache coherence protocols.

Write-Back Caches

- Accumulate changes in cache
- Write back when needed
  - Need the cache for something else
  - Another processor wants it
- On first modification
  - Invalidate other entries
  - Requires non-trivial protocol …

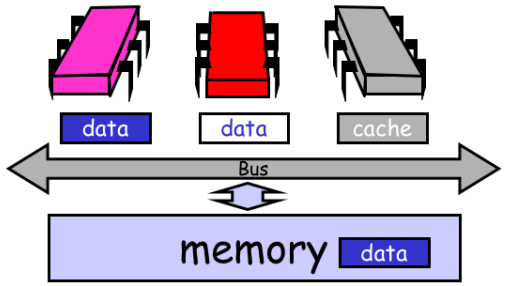
A write-back coherence protocol sends out an invalidation message when the value is first modified, instructing the other processors to discard that value from their caches. Once the processor has invalidated the other cached values, it can make subsequent modifications without further bus traffic. A value that has been modified in the cache but not written back is called dirty. If the processor needs to use the cache for another value, however, it must remember to write back any dirty values.

Write-Back Caches

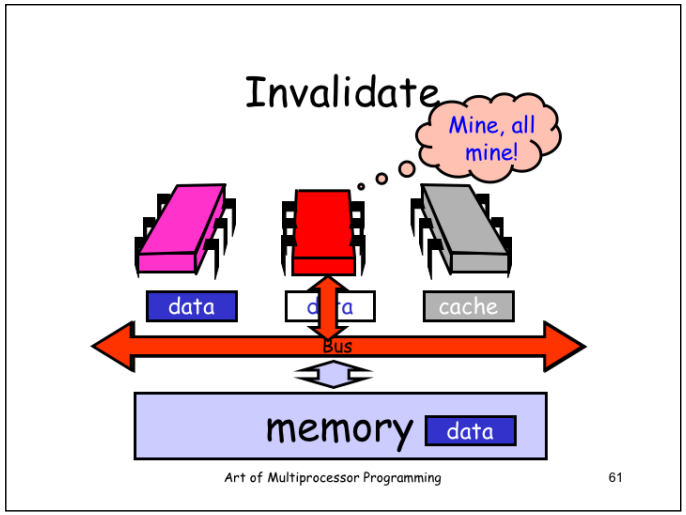
- Cache entry has three states
  - Invalid: contains raw seething bits
  - Valid: I can read but I can't write
  - Dirty: Data has been modified
    - Intercept other load requests
    - Write back to memory before using cache

Instead, we keep track of each cache's state as follows. If the cache is *invalid*, then its contents are meaningless. If it is *valid*, then the processor can read the value, but does not have permission to write it because it may be cached elsewhere. If the value is *dirty*, then the processor has modified the value and it must be written back before that cache can be reused.

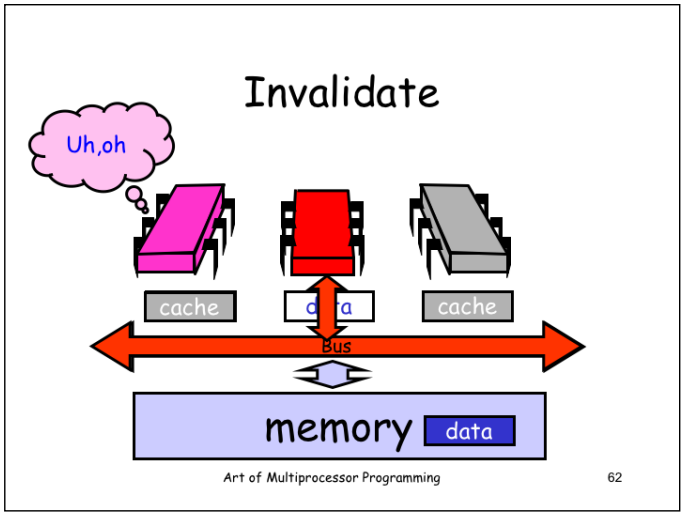
Invalidate



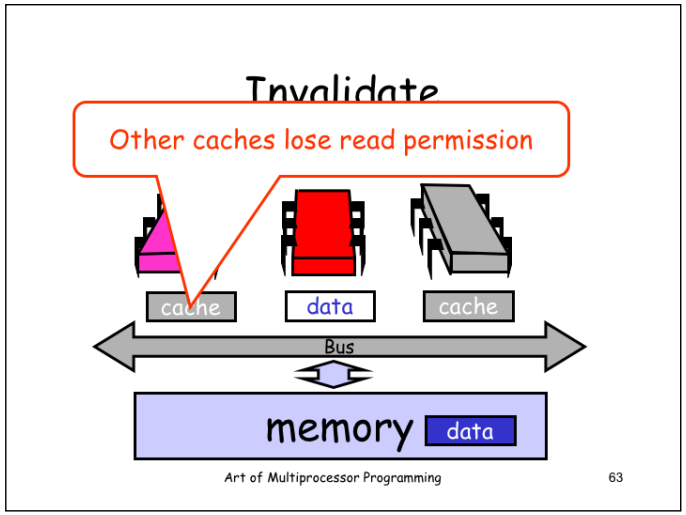
Let's rewind back to the moment when the red processor updated its cached data.



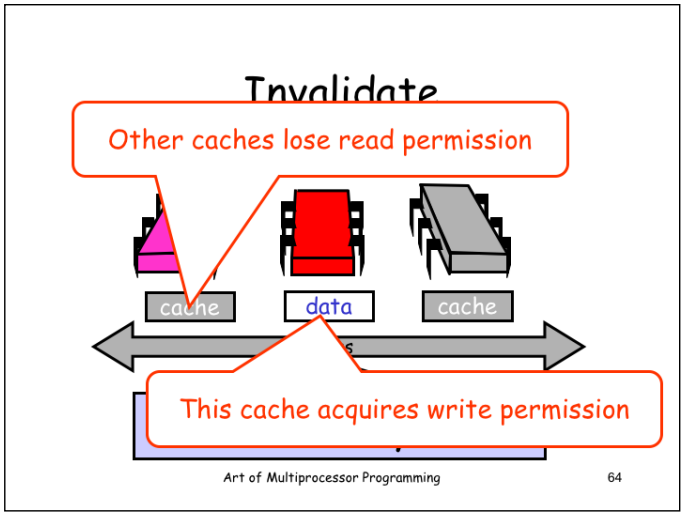
It broadcasts an *invalidation* message warning the other processors to invalidate, or discard their cached copies of that data.



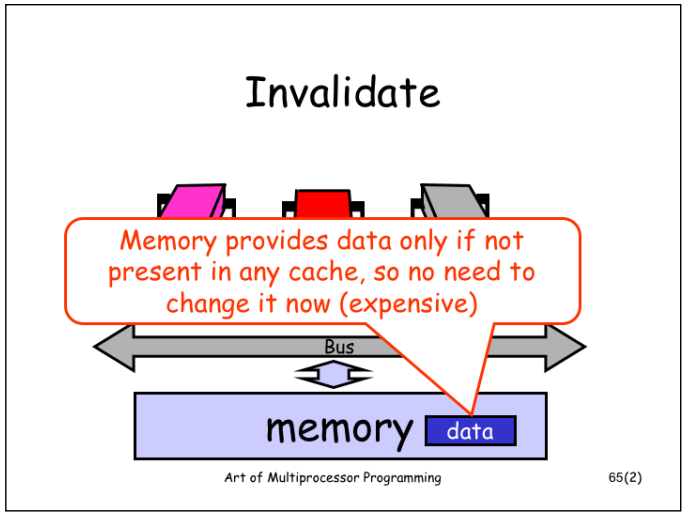
When the other processors hear the invalidation message, they set their caches to the *invalid* state.



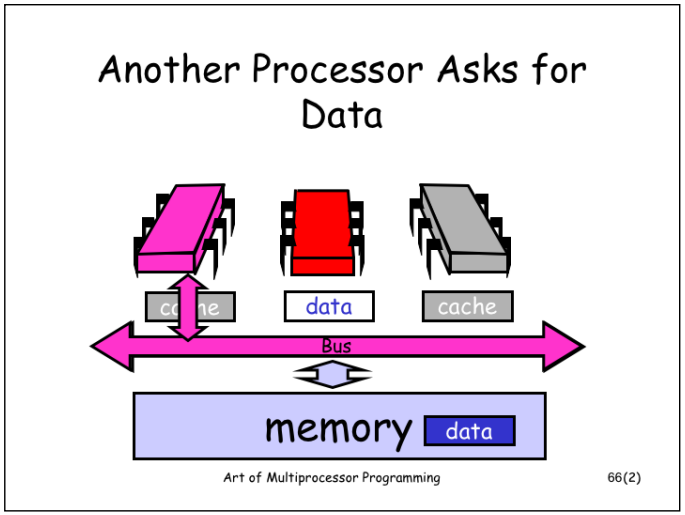
From this point on, the red processor can update that data value without causing any bus traffic, because it knows that it has the only cached copy. This is much more efficient than a write-through cache because it produces much less bus traffic.



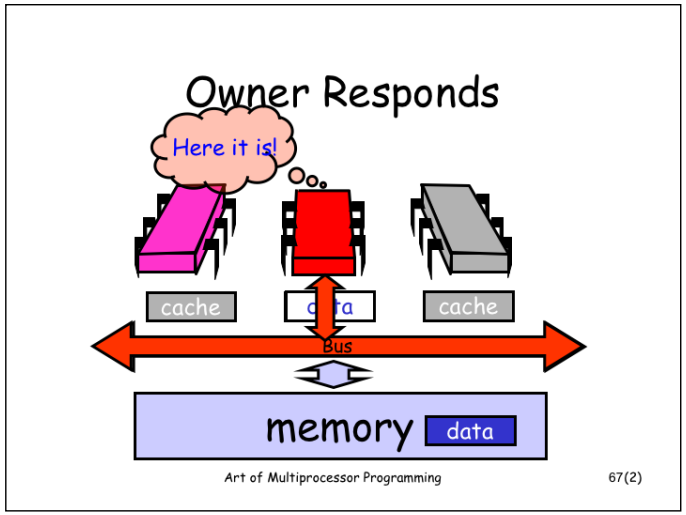
From this point on, the red processor can update that data value without causing any bus traffic, because it knows that it has the only cached copy. This is much more efficient than a write-through cache because it produces much less bus traffic.

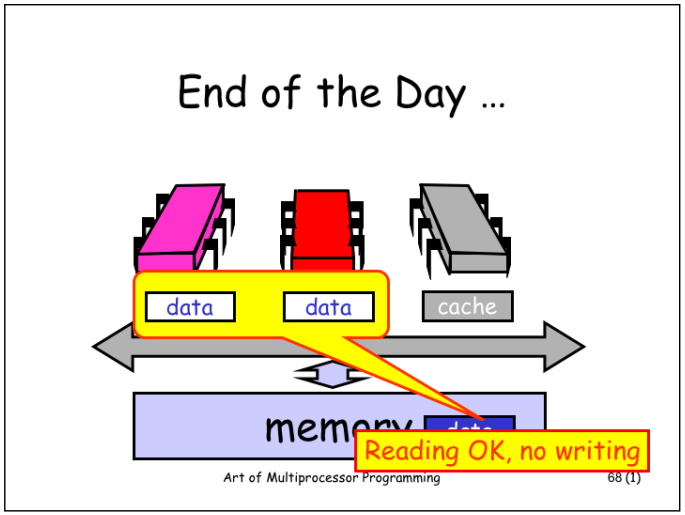


Finally, there is no need to update memory until the processor wants to use that cache space for something else. Any other processor that asks for the data will get it from the red processor.



If another processor wants the data, it asks for it over the bus.





And the owner responds by sending the data over.

Mutual Exclusion

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

Note that optimizing a spin lock is not a simple question, because we have to figure out exactly what we want to optimize: whether it's the bus bandwidth used by spinning threads, the latency of lock acquisition or release, or whether we mostly care about uncontended locks.

Simple TASLock

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

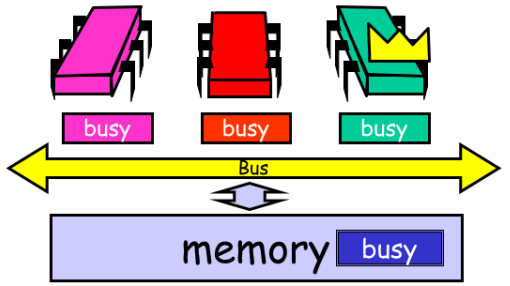
We now consider how the simple test-and-set algorithm performs using a bus-based write-back cache (the most common case in practice). Each test-and-set call goes over the bus, and since all of the waiting threads are continually using the bus, all threads, even those not waiting for the lock, must wait to use the bus for each memory access. Even worse, the test-and-set call invalidates all cached copies of the lock, so every spinning thread encounters a cache miss almost every time, and has to use the bus to fetch the new, but unchanged value. Adding insult to injury, when the thread holding the lock tries to release it, it may be delayed waiting to use the bus that is monopolized by the spinners. We now understand why the TAS lock performs so poorly.

Test-and-test-and-set

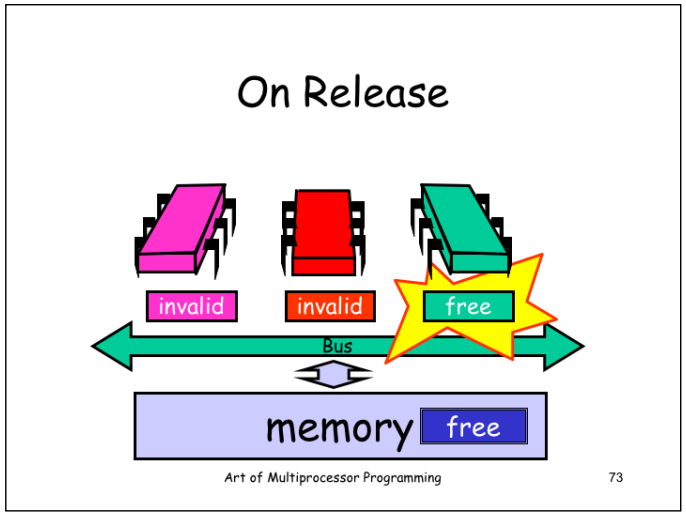
- Wait until lock "looks" free
  - Spin on local cache
  - No bus use while lock busy
- Problem: when lock is released
  - Invalidation storm …

Now consider the behavior of the TAS lock algorithm while the lock is held by a thread A. The first time thread B reads the lock it takes a cache miss, forcing B to block while the value is loaded into B's cache. As long as A holds the lock, B repeatedly rereads the value, but each time B hits in its cache (finding the desired value). B thus produces no bus traffic, and does not slow down other threads' memory accesses. Moreover, a thread that releases a lock is not delayed by threads spinning on that lock.

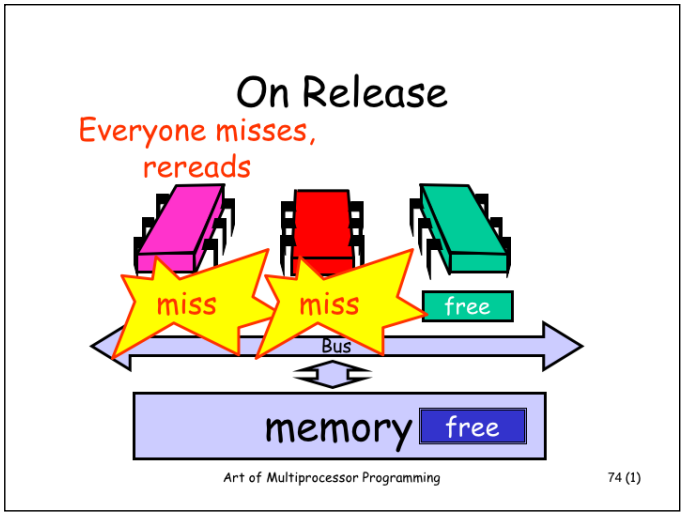
Local Spinning while Lock is Busy



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



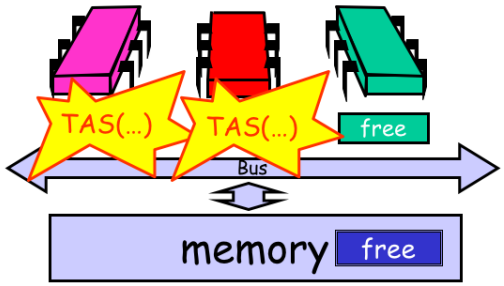
Things deteriorate, however, when the lock is released. The lock holder releases the lock by writing false to the lock variable…



… which immediately invalidates the spinners' cached copies. Each one takes a cache miss, rereads the new value,

On Release

Everyone tries TAS



and they all (more-or-less simultaneously) call test-and-set to acquire the lock. The first to succeed invalidates the others, who must then reread the value, causing a storm of bus traffic.

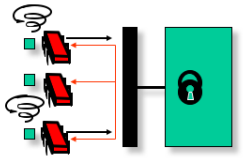
Problems

- Everyone misses
  - Reads satisfied sequentially
- Everyone does TAS
  - Invalidates others' caches
- Eventually quiesces after lock acquired
  - How long does this take?

Eventually, the processors settle down once again to local spinning. This could explain why TATAS lock cannot get to the ideal lock. How long does this take?

Measuring Quiescence Time

X = time of ops that don't use the bus
Y = time of ops that cause intensive bus traffic

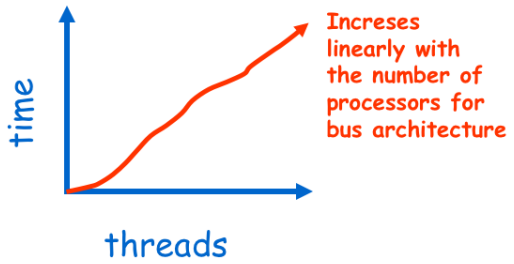


In critical section, run ops X then ops Y. As long as Quiescence time is less than X, no drop in performance.

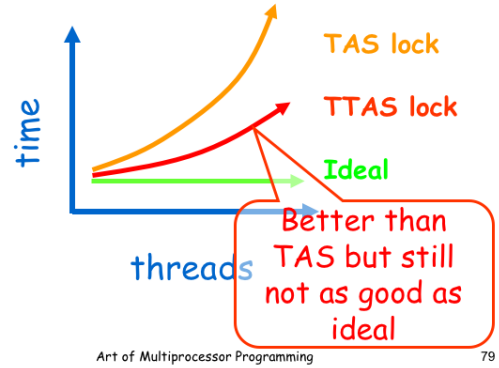
By gradually varying X, can determine the exact time to quiesce.

This is the classical experiment conducted by Anderson. We decrease X until the bus intensive operations in Y interleave with the quiescing of the lock release opeartion, at which point we will see a drop in throughput or an increase in latency.

Quiescence Time



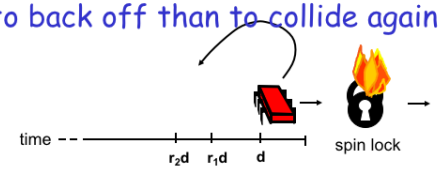
Mystery Explained



So now we understand why the TTAS lock performs much better than the TAS lock, but still much worse than an ideal lock.

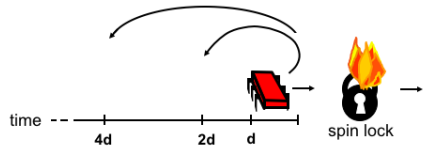
Solution: Introduce Delay

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



Another approach is to introduce a delay. If the lock looks free but you can't get it, then instead of trying again right away, it makes sense to back off a little and let things calm down. There are several places we could introduce a delay --- after the lock rl

Dynamic Example: Exponential Backoff



If I fail to get lock
- wait random duration before retry
- Each subsequent failure doubles expected wait

For how long should the thread back off before retrying? A good rule of thumb is that the larger the number of unsuccessful tries, the higher the likely contention, and the longer the thread should back off. Here is a simple approach. Whenever the thread sees the lock has become free but fails to acquire it, it backs off before retrying. To ensure that concurrent conflicting threads do not fall into ``lock-step'', all trying to acquire the lock at the same time, the thread backs off for a random duration. Each time the thread tries and fails to get the lock, it doubles the expected time it backs off, up to a fixed maximum.

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;
 }}}
```


Here is the code for an exponential back-off lock.

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

The constant MIN_DELAY indicates the initial, shortest limit (it makes no sense for the thread to back off for too short a duration),

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

As in the TTAS algorithm, the thread spins testing the lock until the lock appears to be 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

Then the thread tries to acquire the lock.

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;
 }}}
```

Back off for random duration

If it fails, then it computes a random delay between zero and the current limit.

and then sleeps for that delay before retrying.

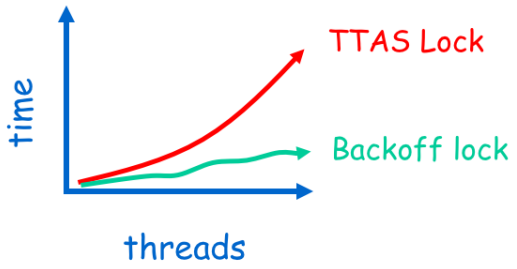
Exponential Backoff Lock

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

Double max delay, within reason

It doubles the limit for the next back-off, up to MAX_DELAY. It is important to note that the thread backs off only when it fails to acquire a lock that it had immediately before observed to be free. Observing that the lock is held by another thread says nothing about the level of contention.

Spin-Waiting Overhead



The graph shows that the backoff lock outperforms the TTAS lock, though it is far from the ideal curve which is flat. The slope of the backoff curve varies greatly from one machine to another, but is invariably better than that of a TTAS lock.

Backoff: Other Issues

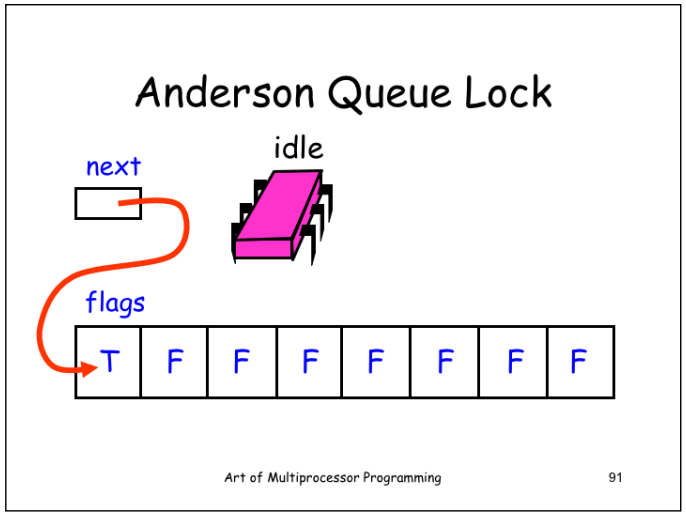
- Good
  - Easy to implement
  - Beats TTAS lock
- Bad
  - Must choose parameters carefully
  - Not portable across platforms

The Backoff lock is easy to implement, and typically performs significantly better than TTAS lock on many architectures. Unfortunately, its performance is sensitive to the choice of minimum and maximum delay constants. To deploy this lock on a particular architecture, it is easy to experiment with different values, and choose the ones that work best. Experience shows, however, that these optimal values are sensitive to the number of processors and their speed, so it is not easy to tune the back-off lock class to be portable across a range of different machines.

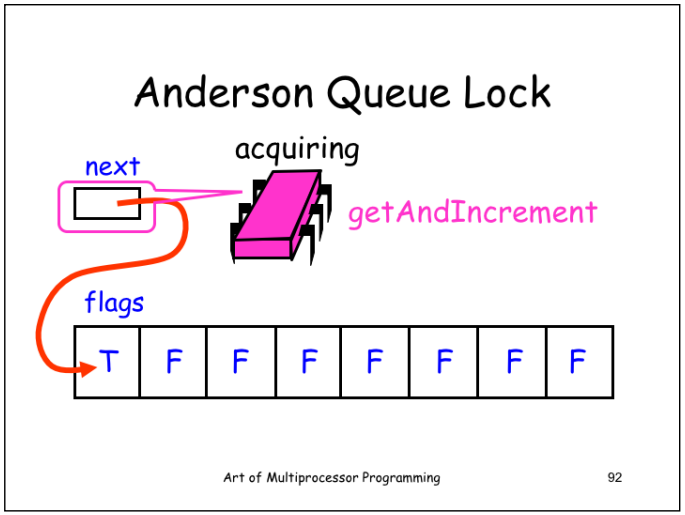
Idea

- Avoid useless invalidations
  - By keeping a queue of threads
- Each thread
  - Notifies next in line
  - Without bothering the others

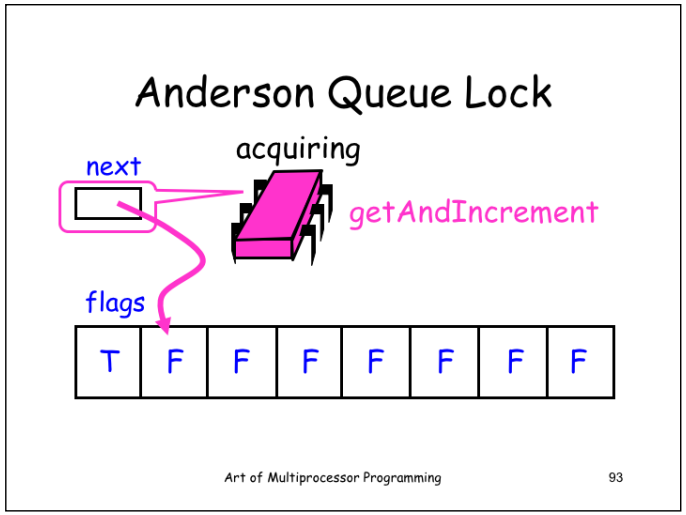
We now explore a different approach to implementing spin locks, one that is a little more complicated than backoff locks, but inherently more portable. One can overcome these drawbacks by having threads form a line, or queue. In a queue, each thread can learn if its turn has arrived by checking whether its predecessor has been served. Invalidation traffic is reduced by having each thread spin on a different location. A queue also allows for better utilization of the critical section since there is no need to guess when to attempt to access it: each thread is notified directly by its predecessor in the queue. Finally, a queue provides first-come-first-served fairness, the same high level of fairness achieved by the Bakery algorithm. We now explore different ways to implement queue locks, a family of locking algorithms that exploit these insights.

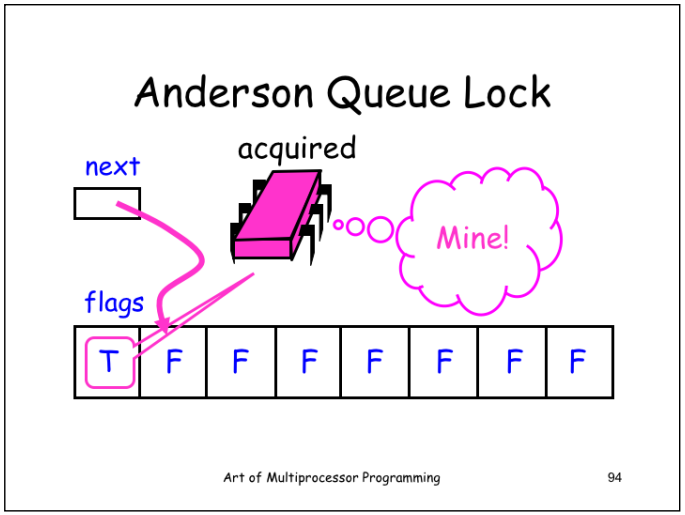


Here is the Anderson queue lock, a simple array-based queue lock. The threads share an atomic integer tail field, initially zero. To acquire the lock, each thread atomically increments the tail field. Call the resulting value the thread's slot. The slot is used as an index into a Boolean flag array. If flag[j] is true, then the thread with slotj has permission to acquire the lock. Initially, flag[0] is true.

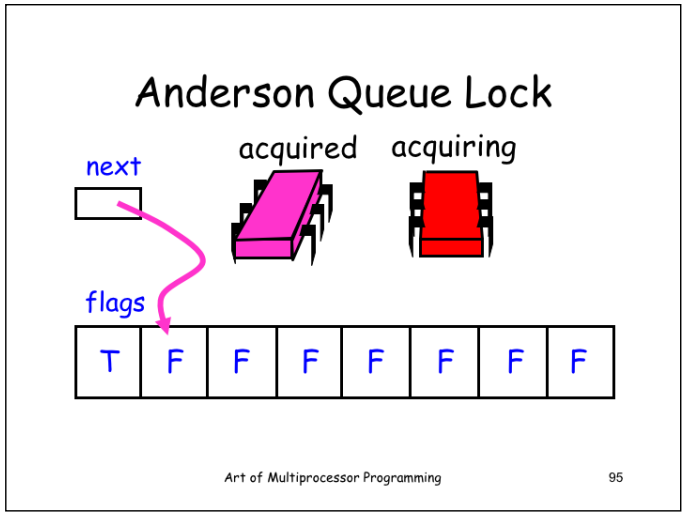


To acquire the lock, each thread atomically increments the tail field. Call the resulting value the thread's slot.

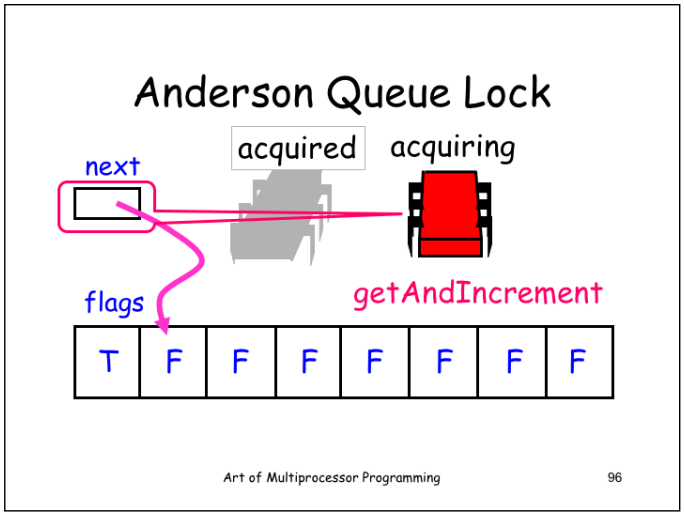




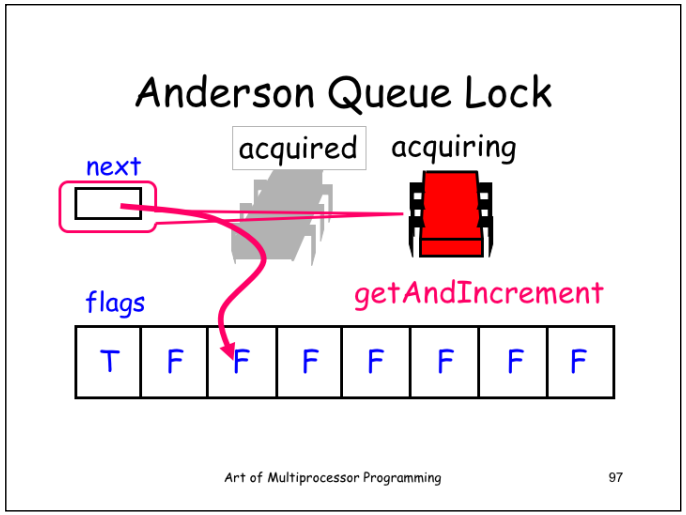
The slot is used as an index into a Boolean flag array. If flag[j] is true, then the thread with slot j has permission to acquire the lock. Initially, flag[0] is true. To acquire the lock, a thread spins until the flag at its slot becomes true.



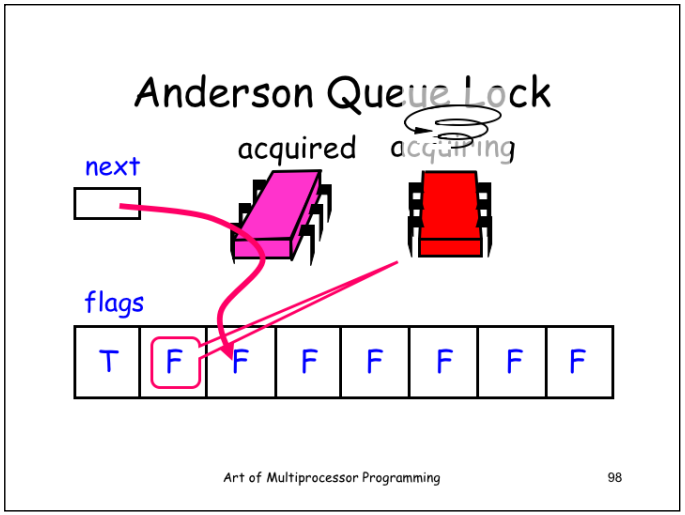
Here another thread wants to acquire the lock.



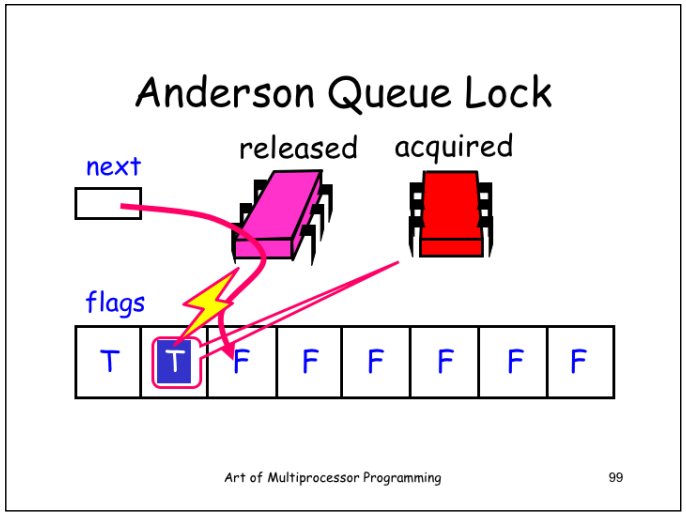
It applies get-and-increment to the next pointer.



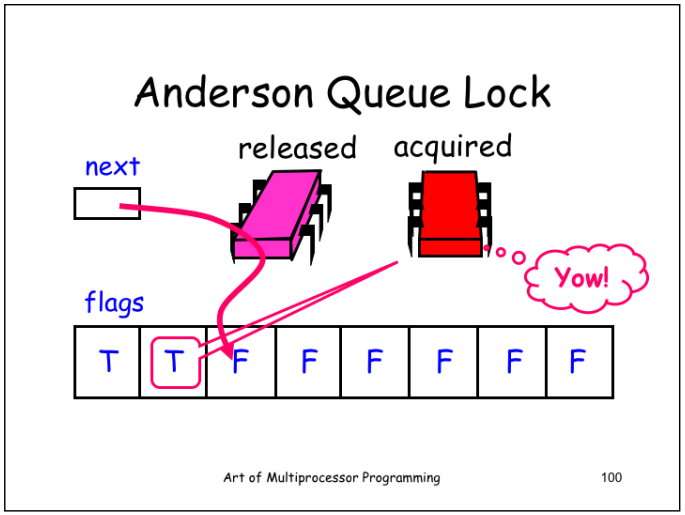
Advances the next pointer to acquire its own slot.



Then it spins until the flag variable at that slot becomes true.



The first thread releases the lock by setting the next slot to true.



The second thread notices the change, and enters its critical section.

Anderson Queue Lock

```
class ALock implements Lock {
 boolean[] flags={true,false,…,false};
 AtomicInteger next
  = new AtomicInteger(0);
 int[] slot = new int[n];
```


Here is what the code looks like.

Anderson Queue Lock

```
class ALock implements Lock {
 boolean[] flags={true,false,…,false};
 AtomicInteger next
  = new AtomicInteger(0);
 int[] slot = new int[n];
```

One flag per thread

We have one flag per thread (this means we have to know how many threads there are).

Anderson Queue Lock

```
class ALock implements Lock {
 boolean[] flags={true,false,…,false};
 AtomicInteger next
  = new AtomicInteger(0);
 int[] slot = new int[n];
```

Next flag to use

The next field tells us which 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

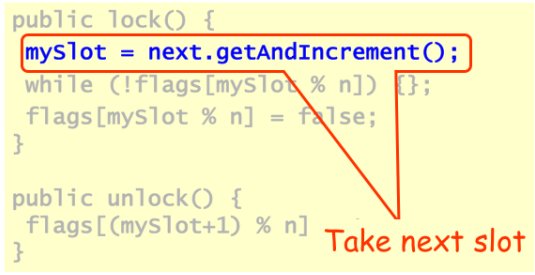
Each thread has a *thread-local* variable that keeps track of its slot.

Anderson Queue Lock

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

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


Here is the code for the lock and unlock methods.

Anderson Queue Lock



First, claim a slot by atomically incrementing the next field.

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

Next wait until my predecessor has released the lock.

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

Reset my slot to false so that it can be used the next time around.

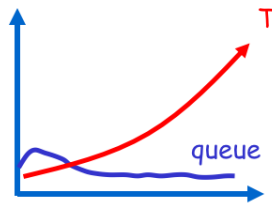
Anderson Queue Lock

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

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

Tell next thread to go

To release the lock, set the slot after mine to true, being careful to wrap around.

Performance



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

The Anderson queue lock improves on Back-off locks because it reduces invalidations to a minimum and schedules access to the critical section tightly, minimizing the interval between when a lock is freed by one thread and when is acquired by another. There is also a theoretical benefit: unlike the TTAS and backoff lock, this algorithm guarantees that there is no lockout, and in fact, provides first-come-first-served fairness, which we actually loose in TTAS and TTAS with backoff.

Anderson Queue Lock

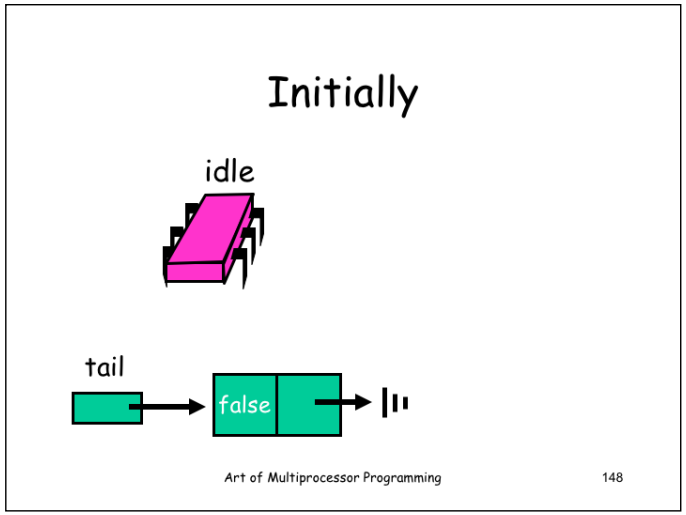
- Good
  - First truly scalable lock
  - Simple, easy to implement
- Bad
  - Space hog
  - One bit per thread
    - Unknown number of threads?
    - Small number of actual contenders?

The Anderson lock has two disadvantages. First, it is not space-efficient. It requires knowing a bound N on the maximum number of concurrent threads, and it allocates an array of that size per lock. Thus, L locks will require O(LN) space even if a thread accesses only one lock at a given time. Second, the lock is poorly suited for uncached architectures, since any thread may end up spinning on any array location, and in the absence of caches, spinning on a remote location may be very expensive.

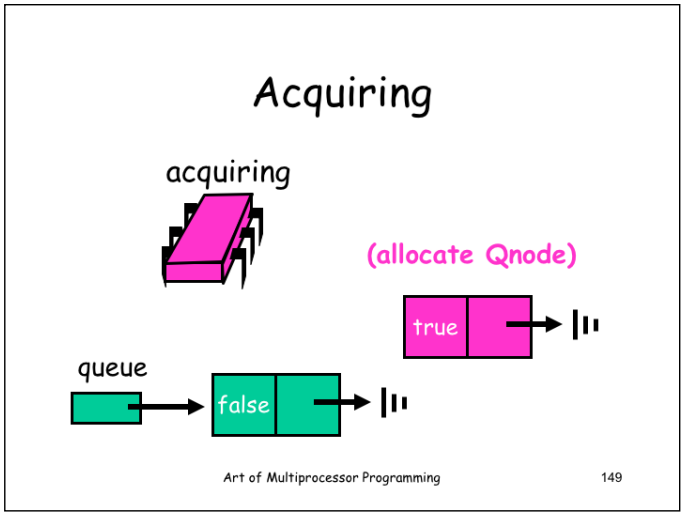
MCS Lock

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

The MCS lock is another kind of queue lock that ensures that processes always spin on a fixed location, so this one works well for cacheless architectures. Like the CLH lock, it uses only a small fixed-size overhead per thread.

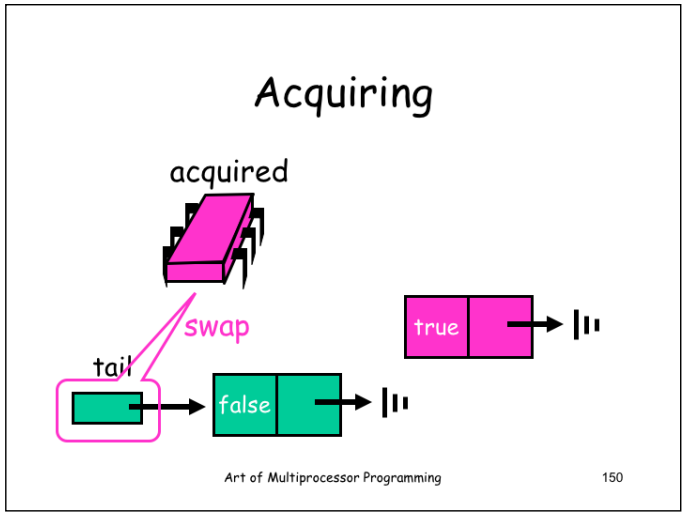


Here, too, the lock is represented as a linked list of QNodes, where each QNode represents either a lock holder or a thread waiting to acquire the lock. Unlike the CLH lock, the list is explicit, not virtual.



To acquire the lock, a thread places its own QNode} at the tail of the list. (Notice that the

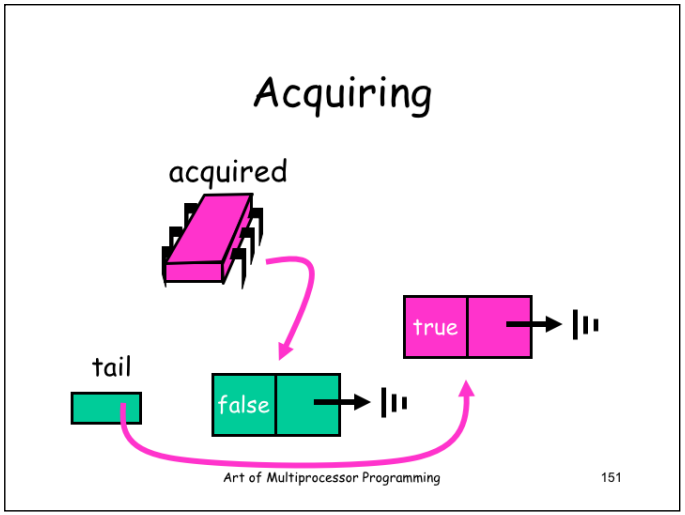
Setting of the value to true is a simplification of the presentation. In the code, we allocate the node with its value false so that we can get out of the with loop immediately if there is no predecessor. But if there is a predecessor, a thread sets the value of its flag to true before it redirects the other thread's pointer to it, so in effect its true whenever its used...thus, in order not to confuse the viewer with the minor issue, we diverge slightly from the code and describe the node as being true here).



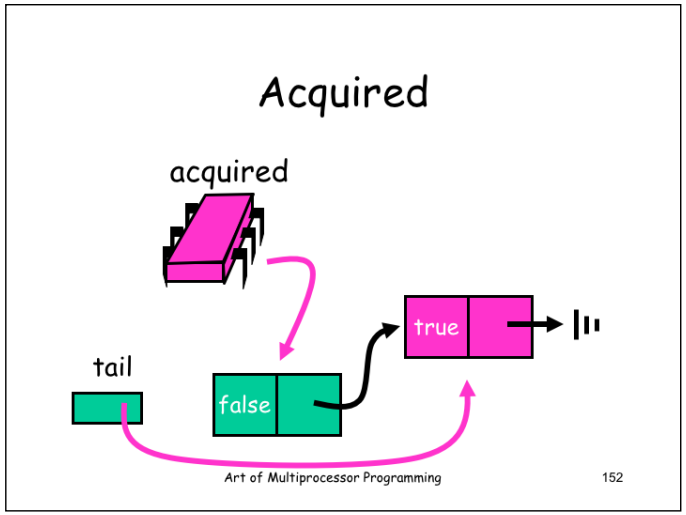
The node then swaps in a reference to its own QNode.

(Notice that the

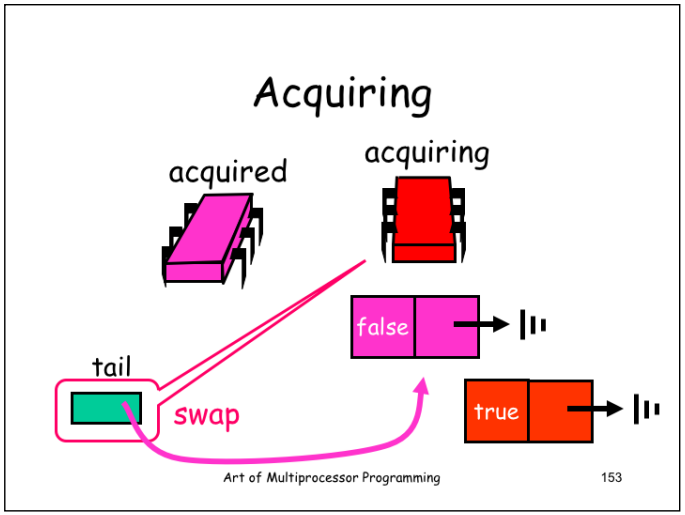
Setting of the value to true is a simplification of the presentation. In the code, we allocate the node with its value false so that we can get out of the with loop immediately if there is no predecessor. But if there is a predecessor, a thread sets the value of its flag to true before it redirects the other thread's pointer to it, so in effect its true whenever its used...thus, in order not to confuse the viewer with the minor issue, we diverge slightly from the code and describe the node as being true here).

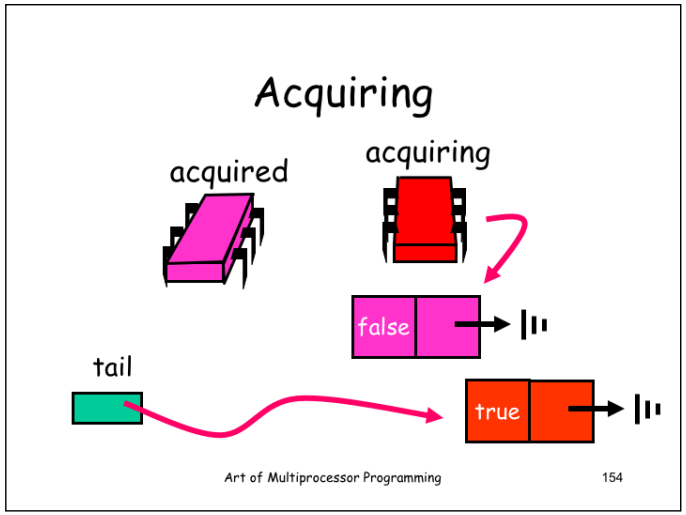


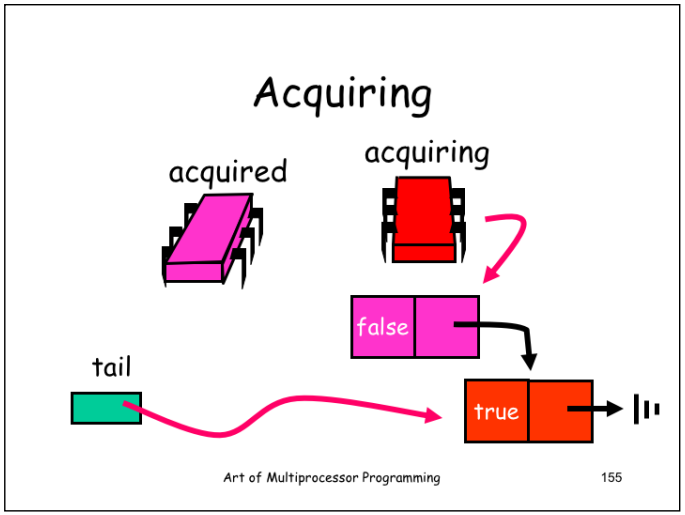
At this point the swap is completed, and the queue variable points to the tail of the queue.

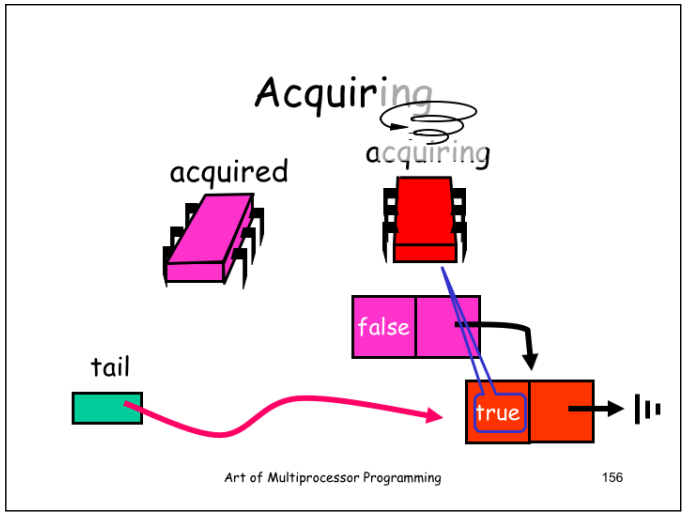


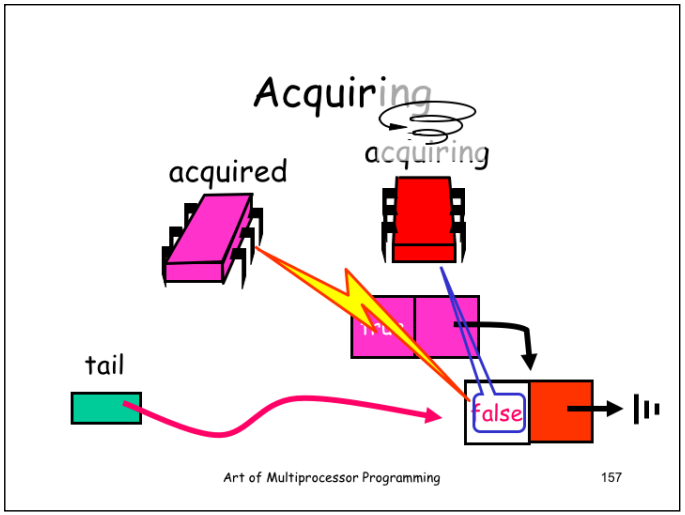
To acquire the lock, a thread places its own QNode} at the tail of the list. If it has a predecessor, it modifies the predecessor's node to refer back to its own QNode.



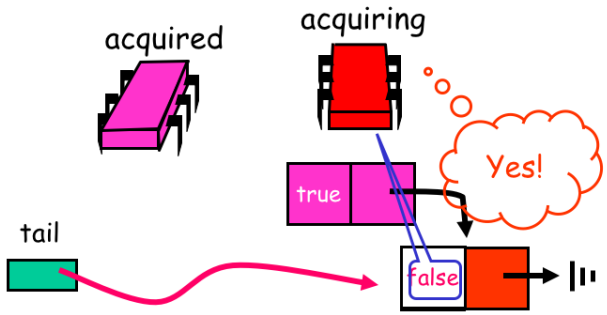

Acquiring
acquired
acquiring
false
tail
true
Art of Multiprocessor Programming
154


Acquiring
acquired
acquiring
false
tail
true
Art of Multiprocessor Programming
155


Acquiring
acquired
acquiring
false
tail
true
Art of Multiprocessor Programming
156


Acquiring
acquired
acquiring
tail
false
Art of Multiprocessor Programming
157

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) {}
  }}}
```


getAndSet = TAS (atomically set the value, return old value)

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

getAndSet = TAS (atomically set the value, return old value)

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

getAndSet = TAS (atomically set the value, return old value)

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

```
 if (qnode.next == null) {
  if (tail.CAS(qnode, null)
   return;
  while (qnode.next == null) {}
 }
qnode.next.locked = false;
}}
```

MCS Queue Lock

Otherwise wait for successor to catch up

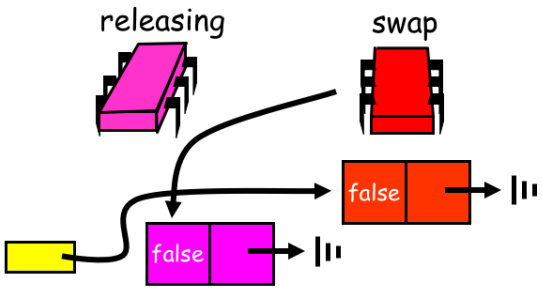
```
:k {
 if (qnode.next == null) {
  if (tail.CAS(qnode, null)
   return;
  while (qnode.next == null) {}
 }
 qnode.next.locked = false;
}}
```


MCS Queue Lock

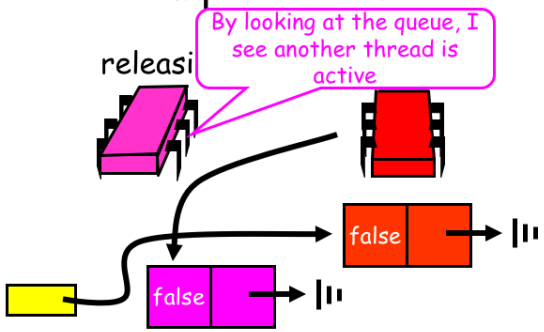
Pass lock to successor

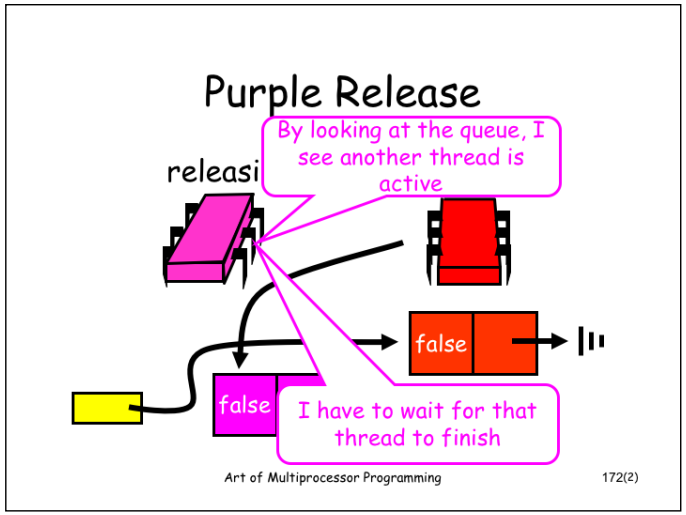
```
class MCSLock implements Lock {
 AtomicRef
 public vo
  if (qnode.next == null) {
   if (tail.CAS(qnode, null)
    return;
   while (qnode.next == null) {}
  }
 qnode.next.locked = false;
}}
```

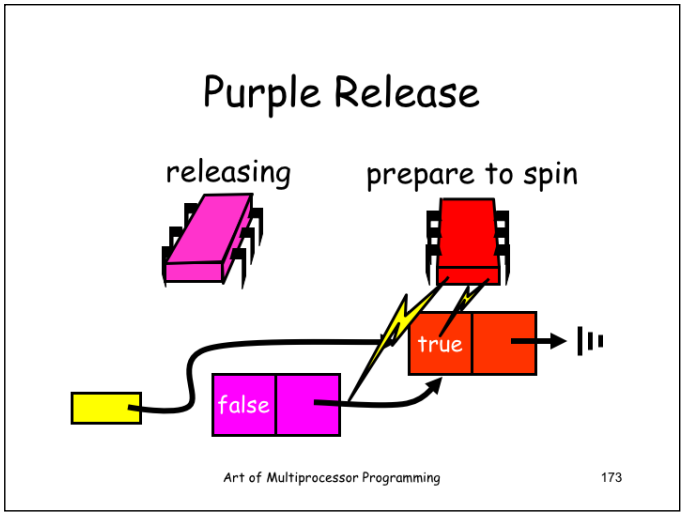
Purple Release



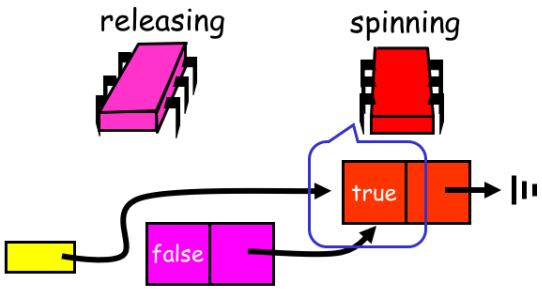
Purple Release



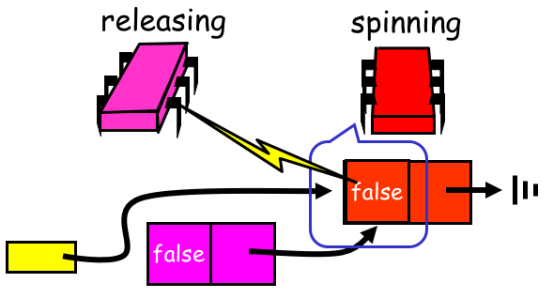

Purple Release
releasi
By looking at the queue, I see another thread is active
false
false
I have to wait for that thread to finish
Art of Multiprocessor Programming
172(2)


Purple Release
releasing
prepare to spin
true
false
Art of Multiprocessor Programming
173

Purple Release



Purple Release



Purple Release