Preemptive Scheduling - Mutual Exclusion

Loïc Guégan

Adapted from P. Ha @ UiT, M. Herlihy & N. Shavit @ 2012, J. Kubiatowicz @ 2010 UCB, A. S. Tanenbaum @ 2008, A. Silberschatz @ 2009

UiT The Arctic University of Norway

Spring - 2025

Outline

- Preemptive scheduling
- Mutual exclusion

Recall: Dispatching Loop

- Dispatching loop of the operating system:

```
Loop {
  RunThread();
  ChooseNextThread();
  SaveStateOfCPU(curTCB);
  LoadStateOfCPU(newTCB);
}
```

- This is an infinite loop
  - One could argue that this is all that the OS does
- Should we ever exit this loop?
  - When?

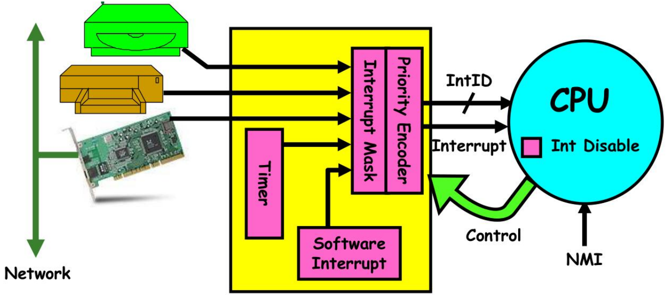
Recall: Running a thread

- Consider the dispatcher first portion: `RunThread()`
- How do I run a thread?
  1. Load its state (registers, PC, stack pointer) into CPU
  2. Load environment (virtual memory space, etc.)
  3. Jump to the PC $\Rightarrow$ done!
- How does the dispatcher get control back?
  1. Internal events: thread returns control voluntarily (non-preempted)
     - ⋆ Software interrupts
     - ⋆ yield()
  2. External events: thread gets *interrupted* (preempted)

External Events – Preemptive scheduling

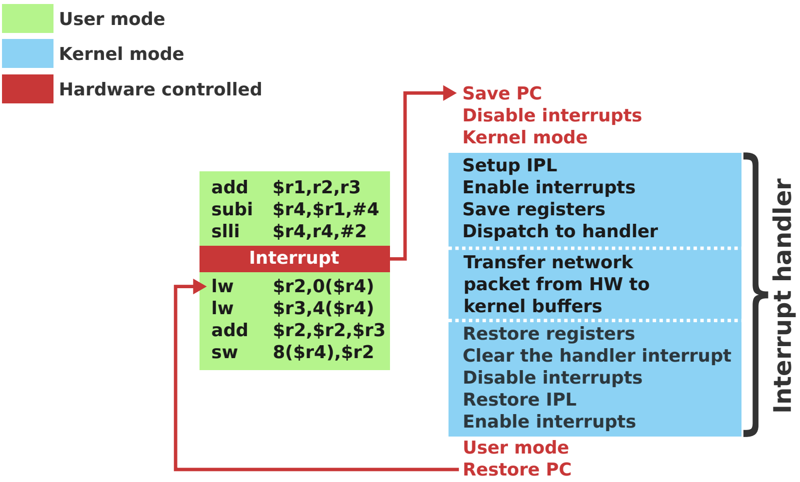
- **Why preemptive scheduling?**
  - Thread never does any I/O, never waits, and never yields control!
    - Could the "ComputePI" program grab all resources and never release the processor?
  - Must find way that dispatcher can regain control!
- **How?**
  - Utilize *external events*
    - Interrupts: signals from hardware or software that stop the running code and jump to kernel
    - Timer: like an alarm clock that goes off some milliseconds
  - If external events occur frequently enough ⇒ ensure dispatcher runs

Interrupt Controller



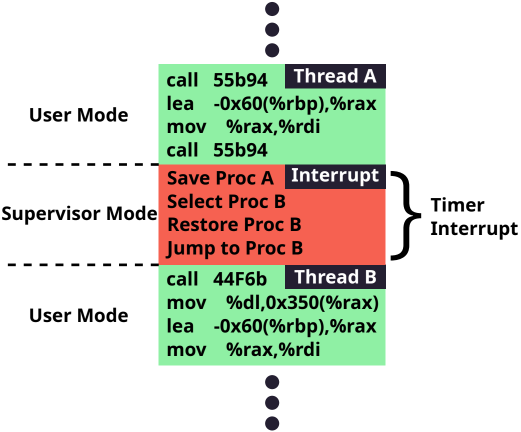
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can't be disabled

Example: Network Interrupt



- An interrupt is a hardware-invoked context switch
  - Always run the interrupt handler immediately

Preemptive scheduling



- Often called **preemptive scheduling**: threads are preempted to achieve scheduling
  - Solves problem of users who do not yield();

Choosing a Thread to Run

- How does Dispatcher decide what to run?
  1. Zero ready threads – dispatcher loops
     - ★ Alternative is to create an "idle thread"
     - ★ Can put machine into low-power mode
  2. Exactly one ready thread – easy
  3. More than one ready thread: use scheduling priorities

- Possible priorities:
  - LIFO (last in, first out):
    - ★ Put ready threads on front of list, remove from front
  - Pick one at random
  - FIFO (first in, first out):
    - ★ Put ready threads on back of list, pull them from front
  - Priority queue:
    - ★ Keep ready list sorted by TCB priority field

Outline

- Preemptive Scheduling
- **Mutual exclusion**
  - Test-and-set locks
  - Queue locks

Mutual exclusion

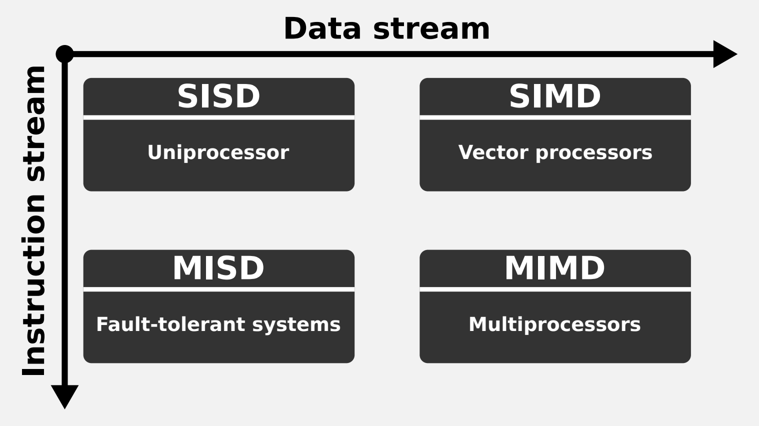
Current knowledge

- Monitor
- Semaphore
- Compare&Swap
- Locks

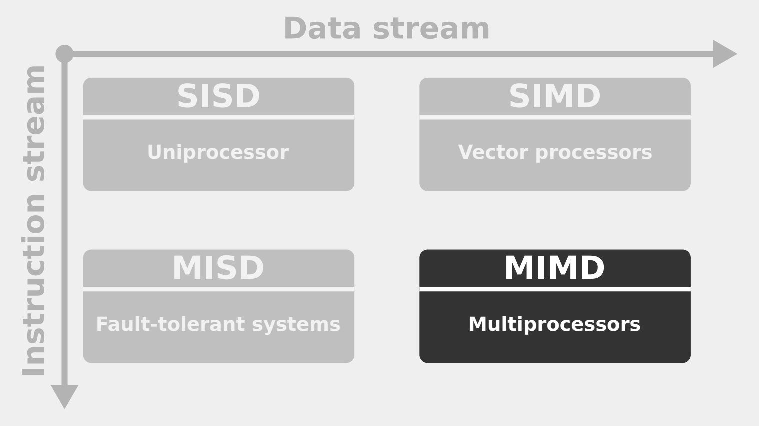
Going further

- How locks are implemented?
  - Various locking algorithms
- What about performance?

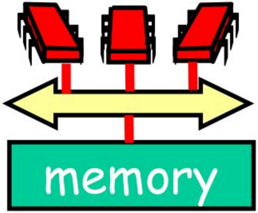
Types of architectures



Types of architectures



MIMD architecture



Shared bus

- Communication contention
- Communication latency
- Memory contention

Locking strategies

What to do if we cannot acquire the lock?

1. Give up the processor
   - Called **block**
   - Good if delays are long $\Rightarrow$ Always good with SISD
2. Keep trying
   - Called **spin** or **busy-wait**
   - Good if delays are short

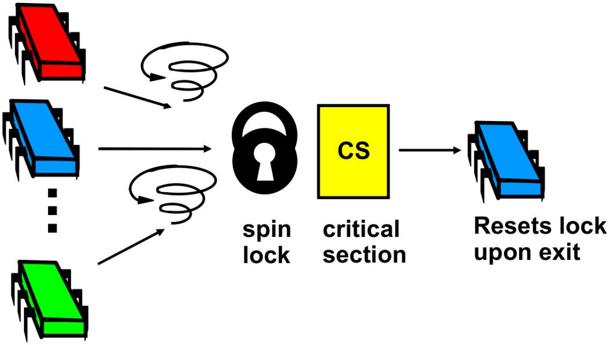
Locking strategies

What to do if we cannot acquire the lock?

1. Give up the processor
   - Called **block**
   - Good if delays are long $\Rightarrow$ Always good with SISD
2. Keep trying
   - Called **spin** or **busy-wait**
   - Good if delays are short

Our focus

Spin-lock principle



Performance issues:

- Sequential bottleneck ⇒ No parallelism
- Contention ⇒ Sensible to large number of threads

Test-and-set locks

- Operates on **boolean value**
- Test-and-set (TAS)
  - Swap **true** with current value
  - Return value tells if prior value was **true** or **false**
- Reset by writting **false**
- TAS a.k.a "getAndSet"

Test-and-set implementation

Java sample

```java
public class AtomicBoolean {
  boolean value;

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

Test-and-set implementation

Package: java.util.concurrent.atomic

Java sample

```
public class AtomicBoolean {
  boolean value;

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

Test-and-set implementation

Java sample

```
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 implementation

Java sample

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

Test-and-set implementation

Java sample

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

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

Test-and-set locks

- Acquire lock by calling TAS
  - If result is false, you win
  - If result is true, you lose
- Locking
  - Value is false $\Rightarrow$ free
  - Value is true $\Rightarrow$ taken
- Release lock by writing false

Test-and-set locks

Java sample

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

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

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

Test-and-set locks

Java sample

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

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

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

Lock state is AtomicBoolean

Test-and-set locks

Java sample

```
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 locks

Java sample

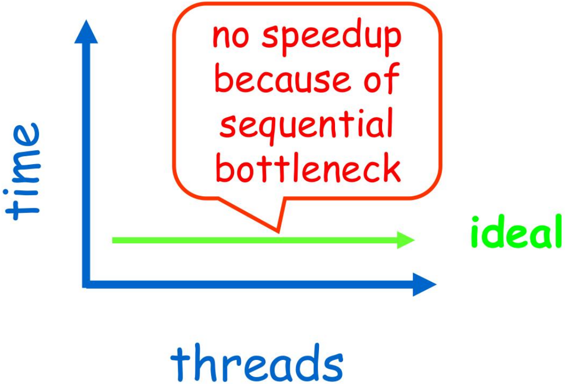
```
class TASlock {
  AtomicBoolean state = new AtomicBoolean(false);

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

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

Release lock by resetting state to false

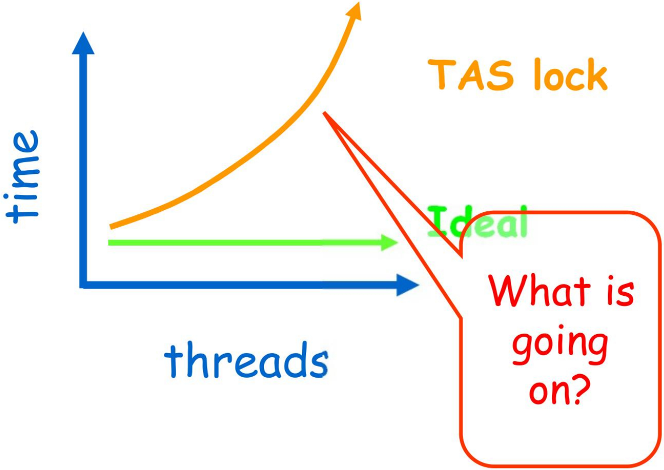
Performance

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

How long should it take?



How long it take?



Outline

- Preemptive Scheduling
- Mutual exclusion
  - Test-and-set locks
    - **Test-and-test-and-set lock**
    - Exponential backoff
  - Queue locks

Test-and-Test-and-Set Locks

Two stages:

1. Lurking stage
   - Wait until lock "looks" free
     - Spin while read returns true (lock taken)
2. 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 Locks

Java sample

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

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

Test-and-Test-and-Set Locks

Java sample

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

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

Wait until lock looks free

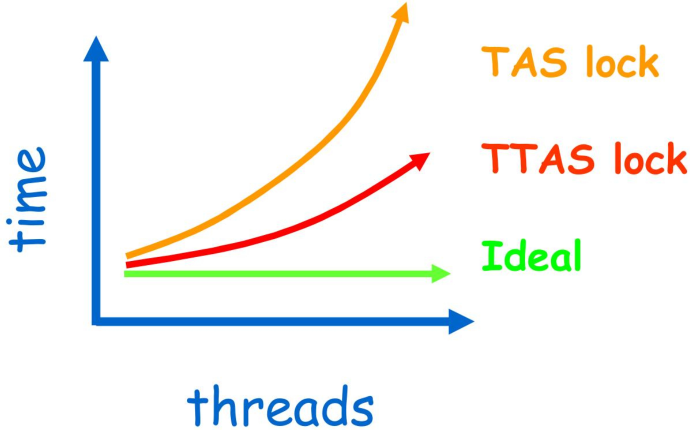
Test-and-Test-and-Set Locks

Java sample

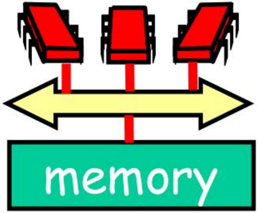
```
class TTASlock {
  AtomicBoolean state = new AtomicBoolean(false);

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

Then try to acquire it

Mystery

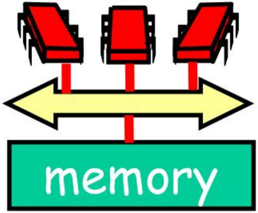


Mystery



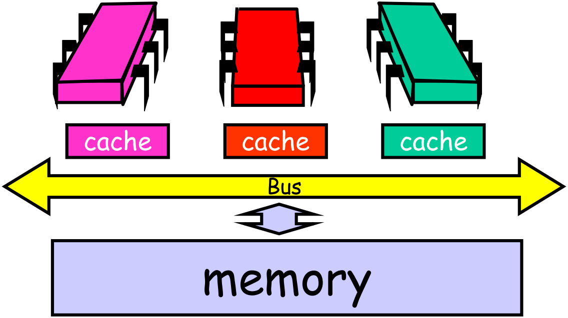
- TAS and TTAS $\Rightarrow$ Do the same thing (in our model)
- Except that performance wise:
  - TTAS performs much better than TAS
- Why is that?

Hypothesis

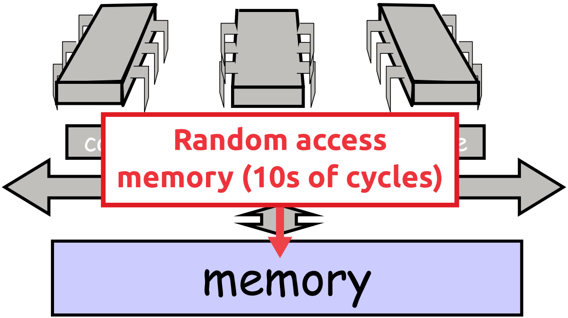


- TAS & TTAS methods
  - Are the same (in our model)
  - But on the performance side they are not (experimentally)
- Our memory abstraction is broken
- Need a more detailed architecture!

Bus-based architecture

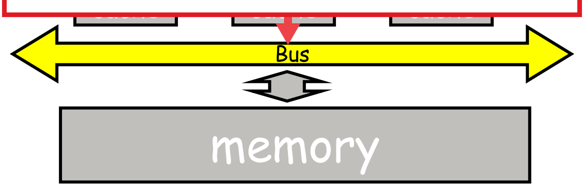


Bus-based architecture

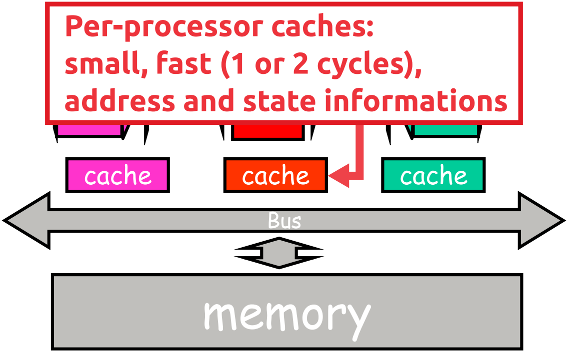


Bus-based architecture

Shared bus
- **Broadcast medium**
- **One broadcaster at a time**
- **Processors and memory all "snoop"**

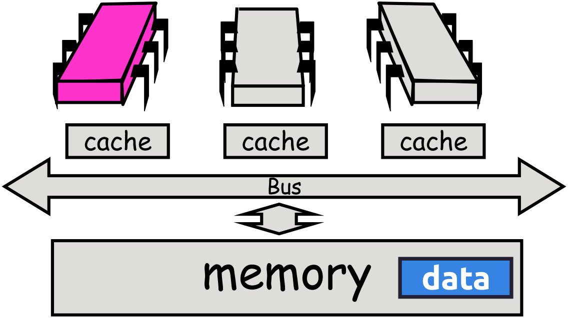


Bus-based architecture

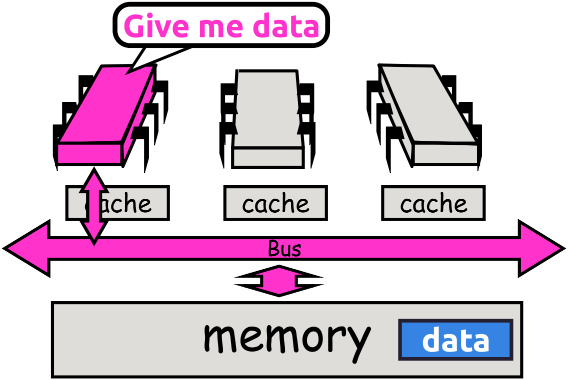


Example of memory accesses with our new architecture

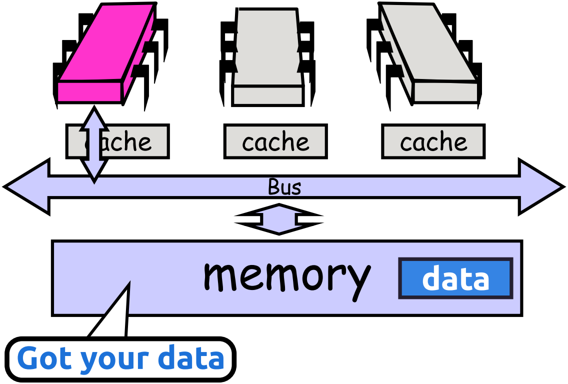
Processor issues load request



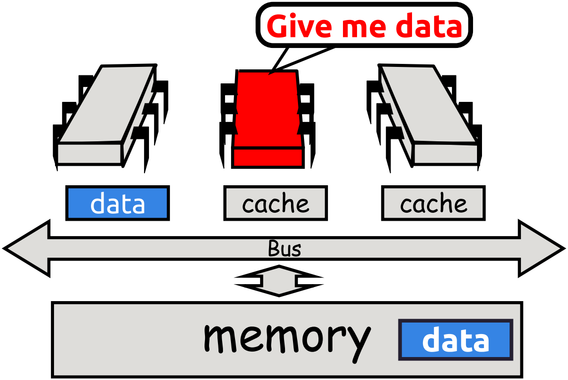
Processor issues load request



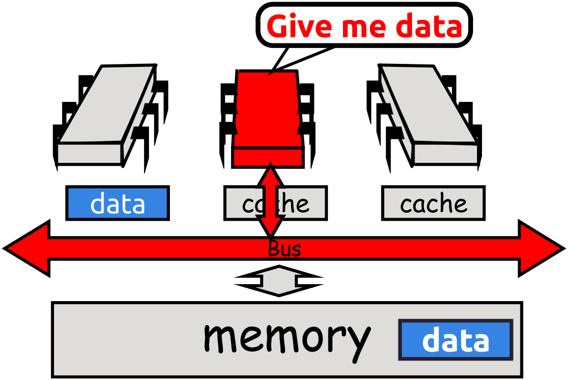
Processor issues load request



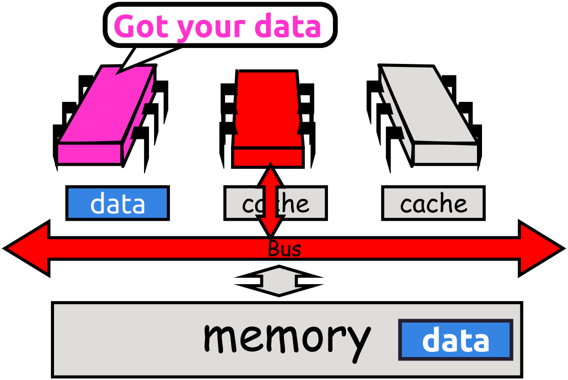
Processor issues load request



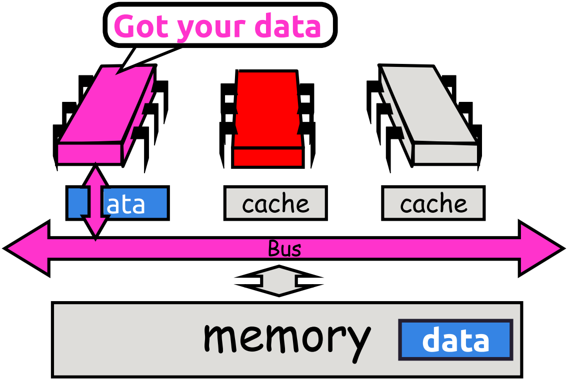
Processor issues load request



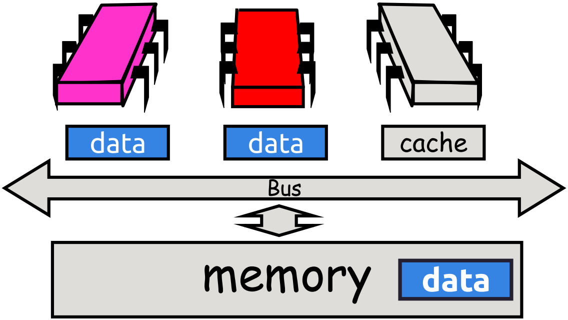
Processor issues load request



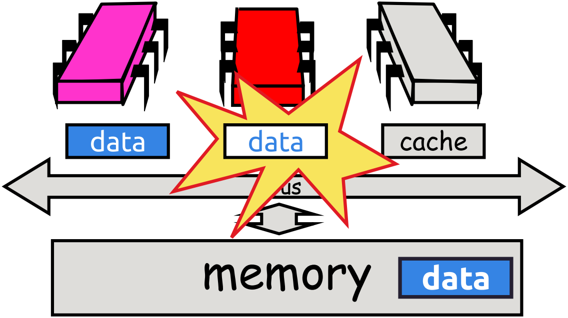
Other processor responds



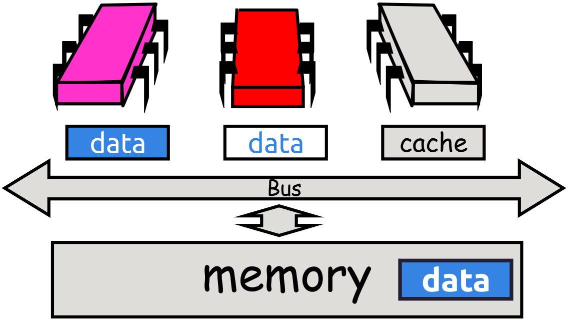
Modify cached data



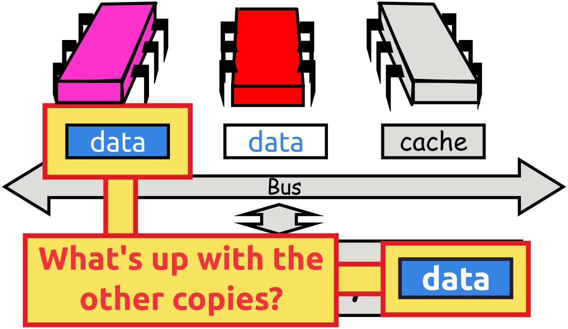
Modify cached data



Modify cached data



Modify cached data



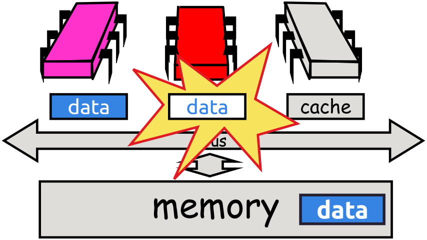
Cache coherence

The issue:

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

Write-Back Caches

- Accumulate changes in cache
- Write back when:
  1. Need the cache entry for something else
  2. Another processor wants it
- On first modification
  - Invalidate other entries
  - Requires non-trivial protocol...



Simple TASLock

Problem:

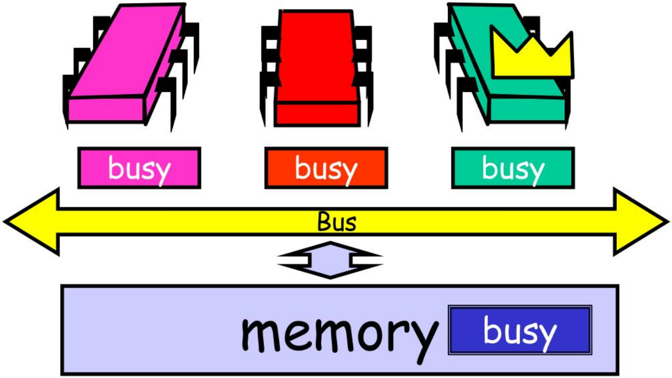
- TAS invalidates cache lines
- Spinners (threads that perform busy-waiting)
  - Miss in cache because of other spinner
  - Go to bus $\Rightarrow$ Congestion
- Thread wants to release lock
  - Delayed behind spinners!!

Test-and-test-and-set

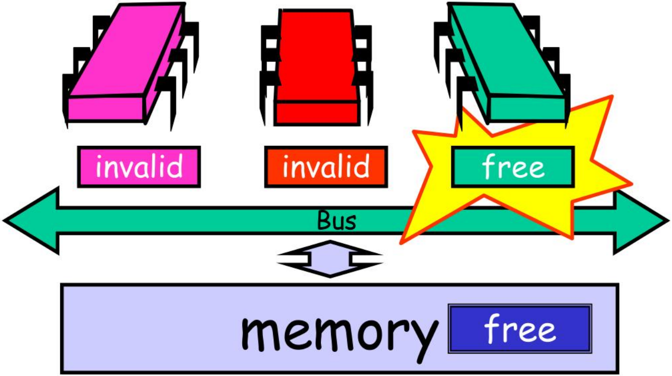
How to solve this problem?

- Wait until lock “looks” free
  - Spin on local cache (no invalidation)
  - No bus use while lock busy
- This is exactly what TTAS does!
- Still a problem $\Rightarrow$ when lock is released
  - Invalidation storm...

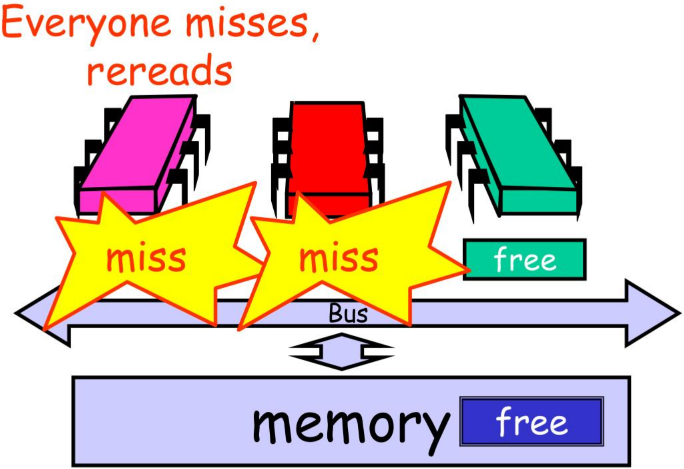
Local spinning while lock is busy



On release

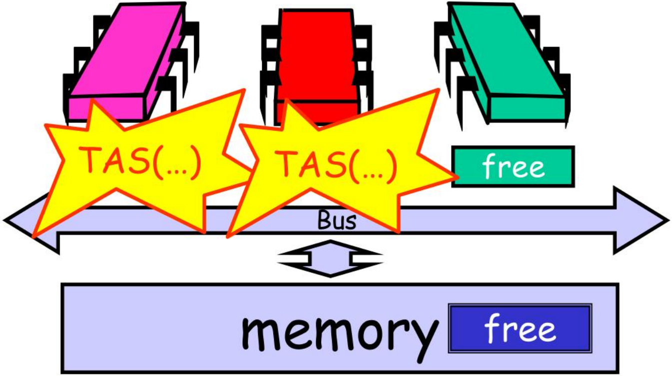


On release



On release

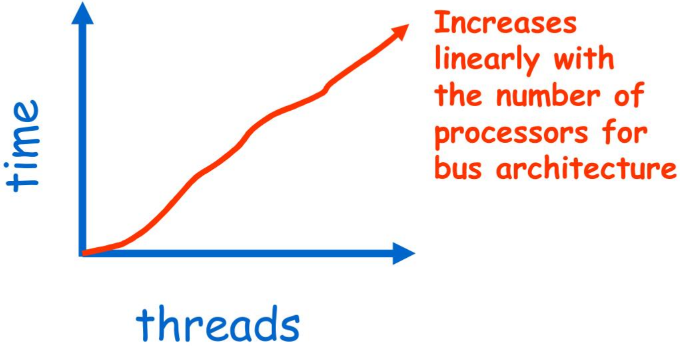
Everyone tries TAS



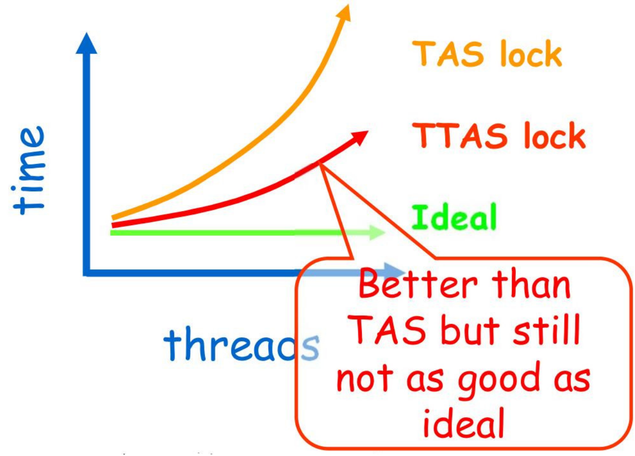
Problems

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

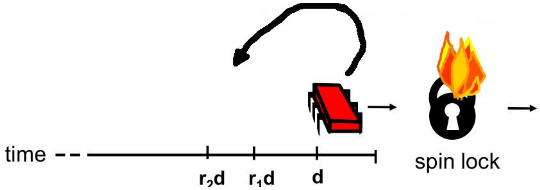
Quiescence time



Mystery explained



Solution: Introduce delay

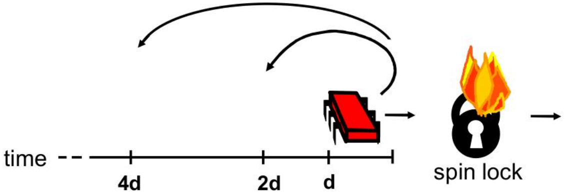


- If the lock looks free
  - But I fail to get it
- There must be lots of contention
  - Better to back off than to collide again

Outline

- Preemptive Scheduling
- Mutual exclusion
  - Test-and-set locks
    - Test-and-test-and-set lock
    - **Exponential backoff**
  - Queue locks

Dynamic Example: Exponential Backoff



If I fail to get lock:

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

Exponential backoff lock

Java sample

```
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

Java sample

```
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

Java sample

```
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 free

Exponential backoff lock

Java sample

```
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

Java sample

```
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;
    }
  }
}
```

Backoff for random duration

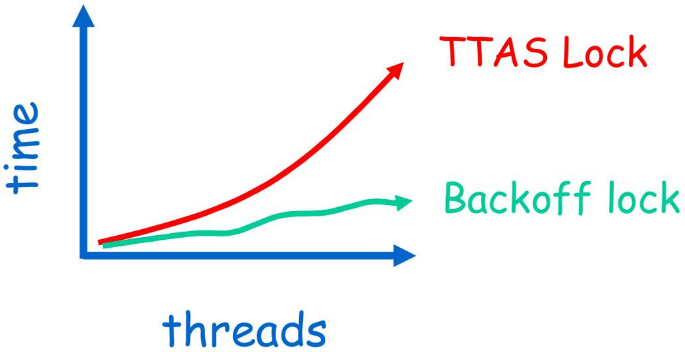
Exponential backoff lock

Java sample

```
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;
    }
  }
}
```

Double max delay, within reason

Spin-waiting overhead



Backoff: Other issues

- Good
  - Easy to implement
  - Beats TTAS lock
- Bad
  - Must choose parameters carefully (MIN and MAX time)
  - Not portable across platforms

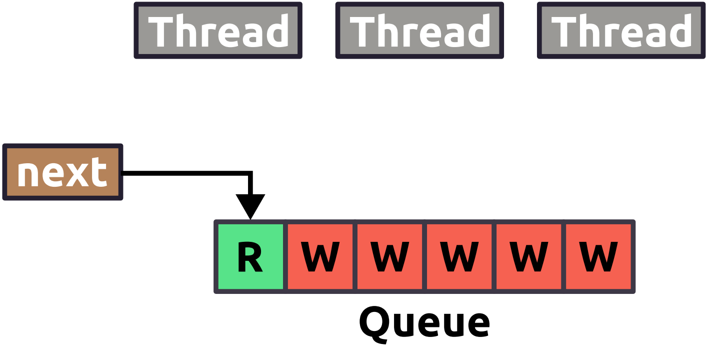
How to overcome these issues?

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

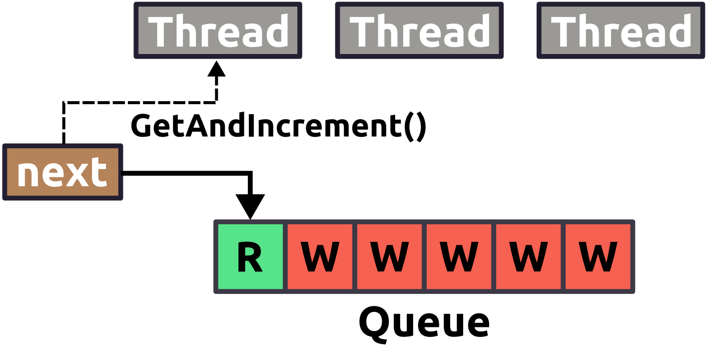
Outline

- Preemptive Scheduling
- Mutual exclusion
  - Test-and-set locks
    - Test-and-test-and-set lock
    - Exponential backoff
  - **Queue locks**

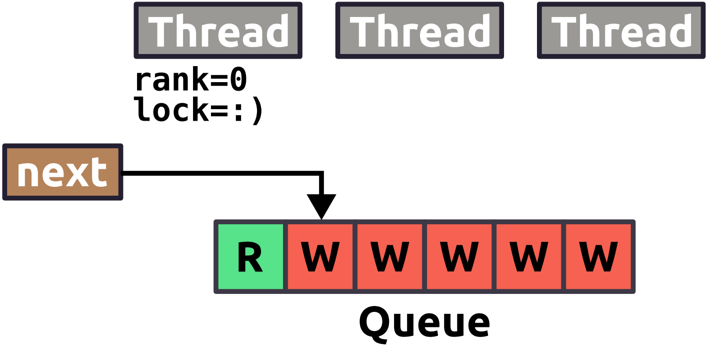
Anderson Queue Lock



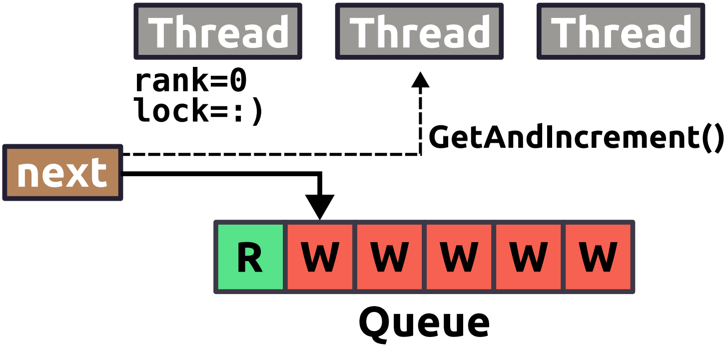
Anderson Queue Lock



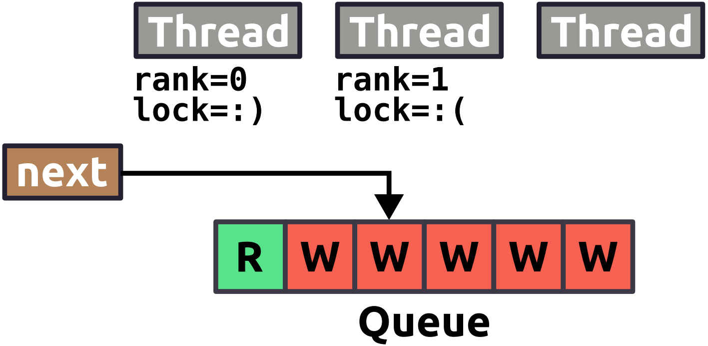
Anderson Queue Lock



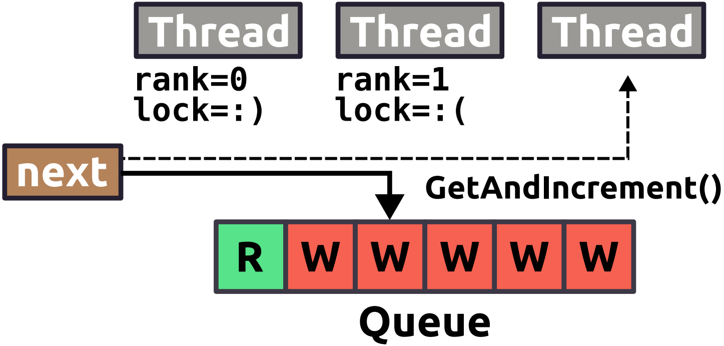
Anderson Queue Lock



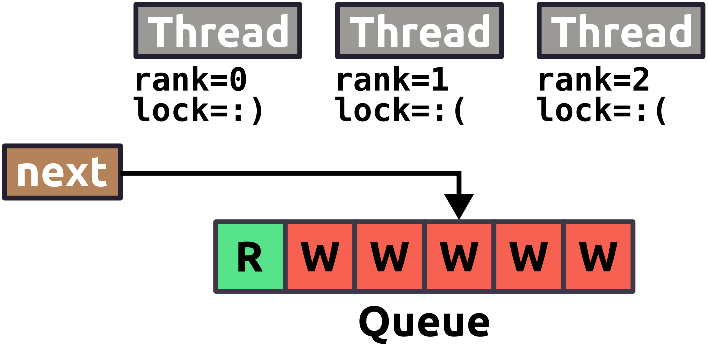
Anderson Queue Lock



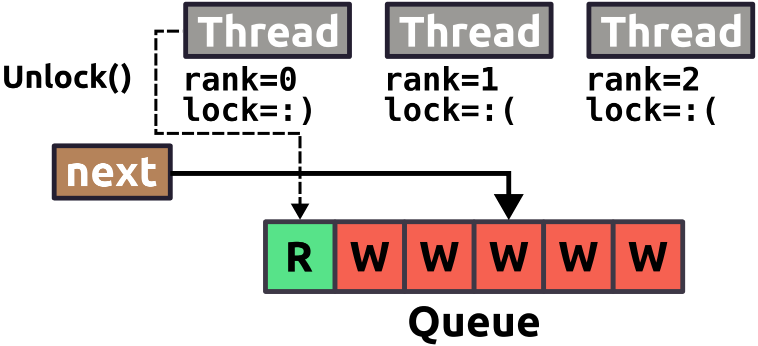
Anderson Queue Lock



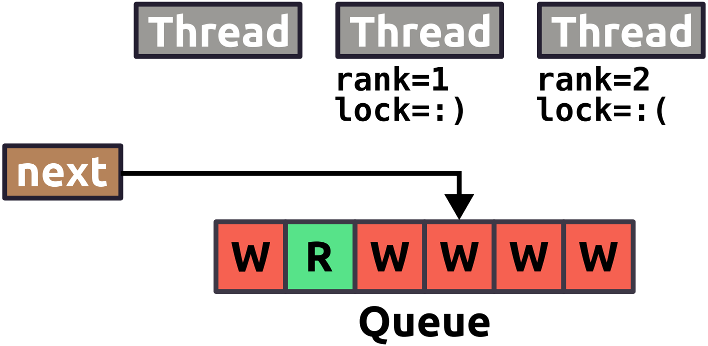
Anderson Queue Lock



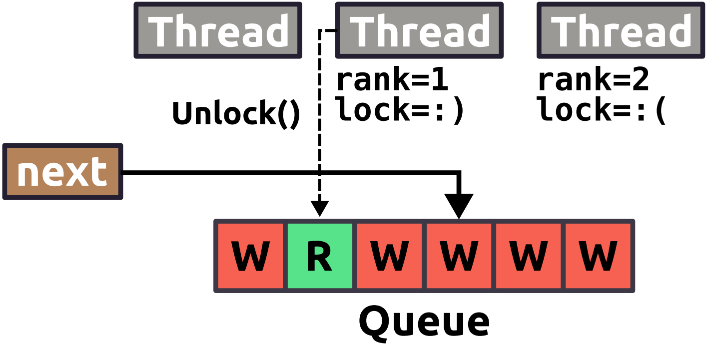
Anderson Queue Lock



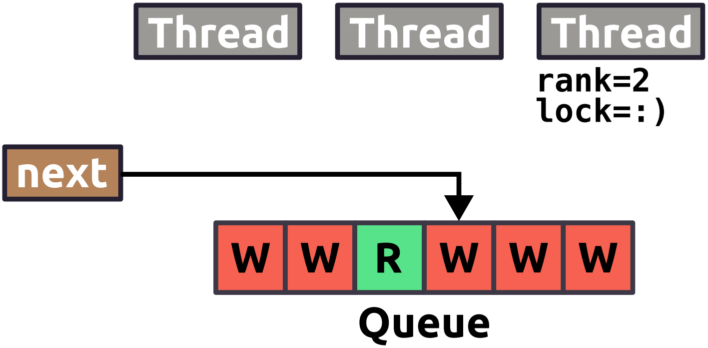
Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock

Java sample

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

Anderson Queue Lock

Java sample

One flag per thread

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

Anderson Queue Lock

Java sample

Next flag to use

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

Anderson Queue Lock

Java sample

Thread-local variable

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

Anderson Queue Lock

Java sample

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

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

Anderson Queue Lock

Java sample

Take next slot

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

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

Anderson Queue Lock

Java sample

Spin until told to go

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

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

Anderson Queue Lock

Java sample

Prepare slot for re-use

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

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

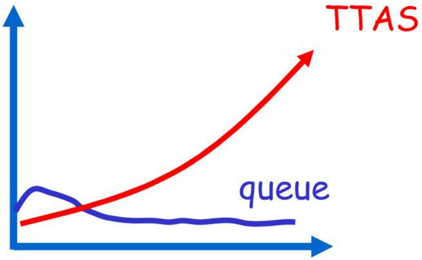
Anderson Queue Lock

Java sample

```java
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

Performance



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

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?
- Solutions:
  - CLH and MCS queue locks (in the book)

References

- A. S. Tanenbaum, Modern Operating Systems.
- A. Silberschatz et. al., Operating System Concepts.
- M. Herlihy et. al., The Art of Multiprocessor Programming.

Thanks for your attention!

Questions?