Lecture 11: advanced locking

Anderson Queue Lock, CLH, MCS, check-then-act, spin-then-park

Alexander Filatov
filatovaur@gmail.com

https://github.com/Svazars/parallel-programming/blob/main/slides/pdf/l11.pdf

In previous episodes

Hardware optimizations

- Store buffering, Load buffering, Invalidate Queues, Interconnect topology

Compiler optimizations

- reorder/invent/delete memory operations, prevent it via compiler barriers

Abstractions and algorithms

- Events, Precedence, Consistency
- Mutual exclusion: Peterson‘s algorithm, FilterLock
- Progress conditions: wait-free, lock-free, obstruction-free, starvation-free, deadlock-free
- Read-Modify-Write operations: lock-free stack

Language memory model – bridge between

- chaos of hardware, anarchy of compiler optimizations
- consistency requirements for high-level concurrent algorithms

In previous episodes

Hardware optimizations

- Store buffering, Load buffering, Invalidate Queues, Interconnect topology

Compiler optimizations

- reorder/invent/delete memory operations, prevent it via compiler barriers

Abstractions and algorithms

- Events, Precedence, Consistency
- Mutual exclusion: Peterson's algorithm, FilterLock
- Progress conditions: wait-free, lock-free, obstruction-free, starvation-free, deadlock-free
- Read-Modify-Write operations: lock-free stack

Language memory model – bridge between

- chaos of hardware, anarchy of compiler optimizations
- consistency requirements for high-level concurrent algorithms

It is time to approach fundamental problems yet again, but using advanced techniques

Lecture plan

1. Locking revisited
2. Array-based locks: Anderson Queue Lock
3. Queue-based locks: CLH
4. Queue-based locks: MCS
5. Challenges: interrupt/timeout
6. Challenges: memory management
7. Spin-then-park
8. Summary

Lecture plan

Question time

Question: Name 3 key properties of concurrent algorithm that we are studying in this course

Locking revisited

Safety
Progress
Performance

Locking revisited

Safety

- Mutual exclusion

Progress
Performance

Locking revisited

Safety

- Mutual exclusion

Lecture 6:

Mutual exclusion:

- Critical sections of different threads do not overlap. $\forall A, B, k, j\ CS_A^k \rightarrow CS_B^j$ or $CS_B^j \rightarrow CS_A^k$

Progress

Performance

Locking revisited

Safety

- Mutual exclusion
- Deadlock-freedom

Progress
Performance

Locking revisited

Safety

- Mutual exclusion
- Deadlock-freedom

Lecture 6:
Freedom from deadlock:

- If some thread attempts to acquire the lock, then some thread will succeed in acquiring the lock. If thread `A` calls `lock()` but never acquires the lock, then other threads must be completing an infinite number of critical sections.

Progress
Performance

Locking revisited

Safety

- Mutual exclusion
- Deadlock-freedom
- Visibility

Progress
Performance

Locking revisited

Safety

- Mutual exclusion
- Deadlock-freedom
- Visibility

Lecture 10:

- Compiler barriers
- Memory barriers
- Consistency guarantees from language memory model

Progress

Performance

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

Performance

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom

Performance

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom

Lecture 6:

Freedom from starvation (lockout freedom):

- Every thread that attempts to acquire the lock eventually succeeds. Every call to `lock()` eventually returns.

Performance

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom
- Fairness

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom
- Fairness

Lecture 2: admission policy

Competing threads	How to manage EnterSet?	How to manage ArriveSet?
• **owner**	• LIFO	• try-lock-then-wait (ArriveSet > EnterSet)
• **waiters** (EnterSet)	• FIFO	• if-busy-then-wait (ArriveSet < EnterSet)
• **arriving threads** (ArriveSet)	• priority/random	• random or heuristic

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, Fairness

Performance

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, Fairness

Performance

- Memory overhead

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, Fairness

Performance

- Memory overhead

Lecture 6

Theorem

Mutual exclusion for N threads requires read-write of at least N distinct locations

Lecture 8

Theorem

`compareAndSet` has ∞ consensus number

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, Fairness

Performance

- Memory overhead
- Throughput in uncontended mode (fast path)

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, Fairness

Performance

- Memory overhead
- Throughput in uncontended mode (fast path)
- Efficiency in contended mode

Question time

Question: What is lock convoy?

Question time

Question: What is thundering herd problem?

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, Fairness

Performance

- Memory overhead
- Throughput in uncontended mode (fast path)
- Efficiency in contended mode

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, Fairness

Performance

- Memory overhead
- Throughput in uncontended mode (fast path)
- Efficiency in contended mode
- Scalability

Question time

Question: Mutex guarantees only one thread will execute critical section at any moment of time. What do you mean under "scalability" of mutex?

Question time

Question: What is invalidation storm?

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, Fairness

Performance

- Memory overhead
- Throughput in uncontended mode (fast path)
- Efficiency in contended mode
- Scalability

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, Fairness

Performance

- Memory overhead
- Throughput in uncontended mode (fast path)
- Efficiency in contended mode
- Scalability
- Backoff policy

Question time

Question: What is backoff strategy and which trade-offs it implies?

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, Fairness

Performance

- Memory overhead
- Throughput in uncontended mode (fast path)
- Efficiency in contended mode
- Scalability
- Backoff policy

Locking revisited

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, Fairness

Performance

- Memory overhead
- Throughput in uncontended mode (fast path)
- Efficiency in contended mode
- Scalability
- Backoff policy

Simplest non-reenterable mutex on `AtomicBoolean`?

TATAS lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  private boolean tryLock() { return state.getAndSet(LOCKED) == UNLOCKED; }
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (tryLock()) return;
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

TATAS lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  private boolean tryLock() { return state.getAndSet(LOCKED) == UNLOCKED; }
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (tryLock()) return;
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

Safety: Mutual exclusion

TATAS lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  private boolean tryLock() { return state.getAndSet(LOCKED) == UNLOCKED; }
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (tryLock()) return;
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

Safety: Deadlock-freedom

TATAS lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  private boolean tryLock() { return state.getAndSet(LOCKED) == UNLOCKED; }
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (tryLock()) return;
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

Safety: Visibility

TATAS lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  private boolean tryLock() { return state.getAndSet(LOCKED) == UNLOCKED; }
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (tryLock()) return;
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

Progress: Starvation-freedom

TATAS lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  private boolean tryLock() { return state.getAndSet(LOCKED) == UNLOCKED; }
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (tryLock()) return;
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

Progress: Fairness

TATAS lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  private boolean tryLock() { return state.getAndSet(LOCKED) == UNLOCKED; }
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (tryLock()) return;
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

Performance: Backoff policy

TATAS lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  private boolean tryLock() { return state.getAndSet(LOCKED) == UNLOCKED; }
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (tryLock()) return;
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

Performance: Efficiency in contended mode

TATAS lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  private boolean tryLock() { return state.getAndSet(LOCKED) == UNLOCKED; }
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (tryLock()) return;
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

Performance: Efficiency in contended mode
Hint: invalidation storm and quiescence time (Lecture 8)

TATAS lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  private boolean tryLock() { return state.getAndSet(LOCKED) == UNLOCKED; }
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (tryLock()) return;
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

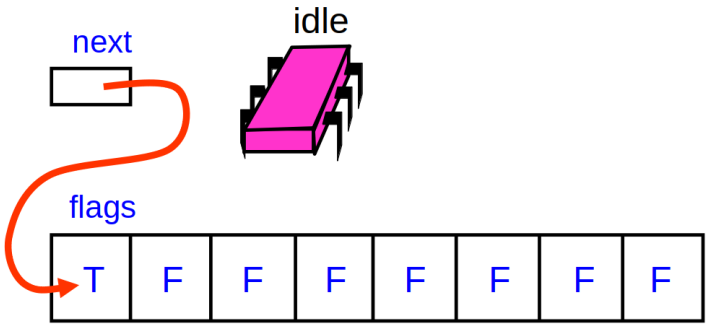
Idea: independent threads should spin on independent memory locations

Lecture plan

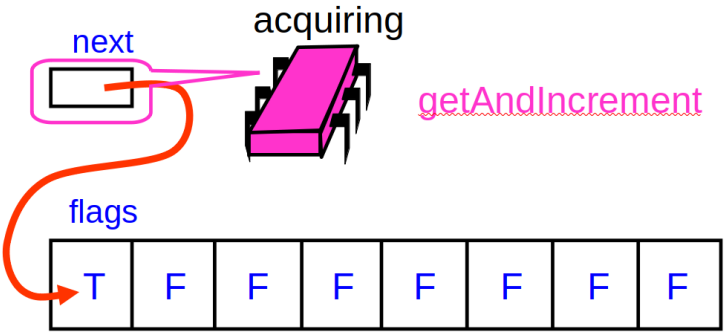
Anderson Queue Lock

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

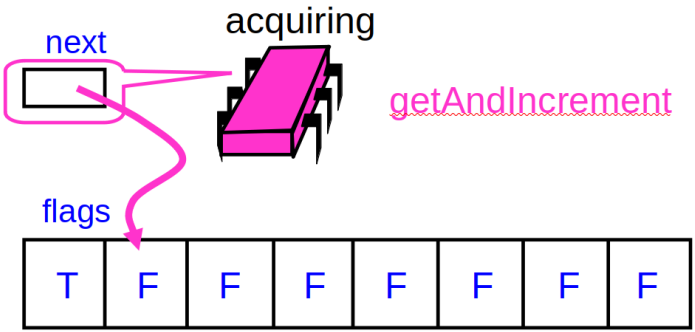
Anderson Queue Lock



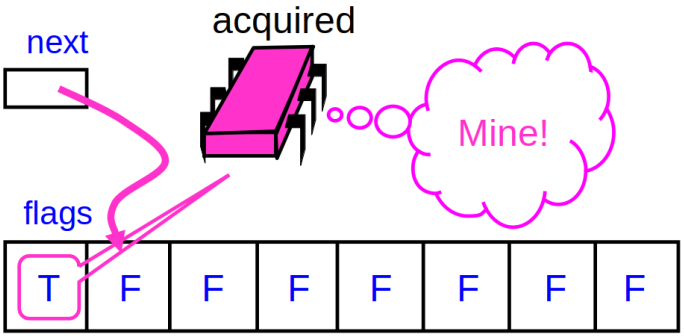
Anderson Queue Lock



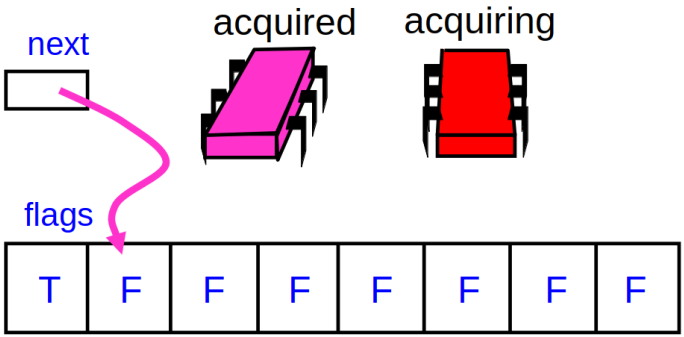
Anderson Queue Lock



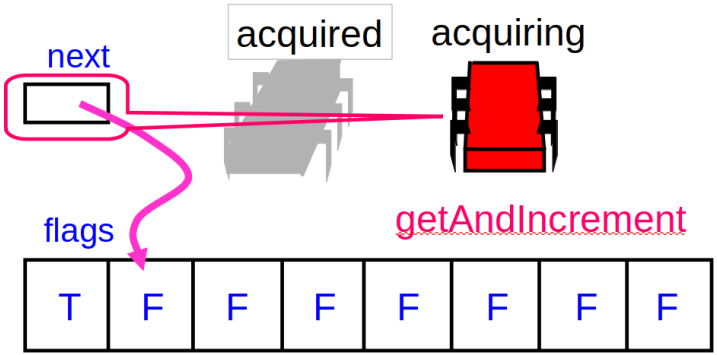
Anderson Queue Lock



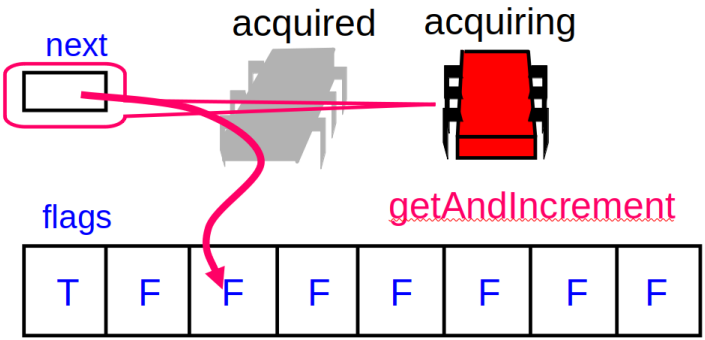
Anderson Queue Lock



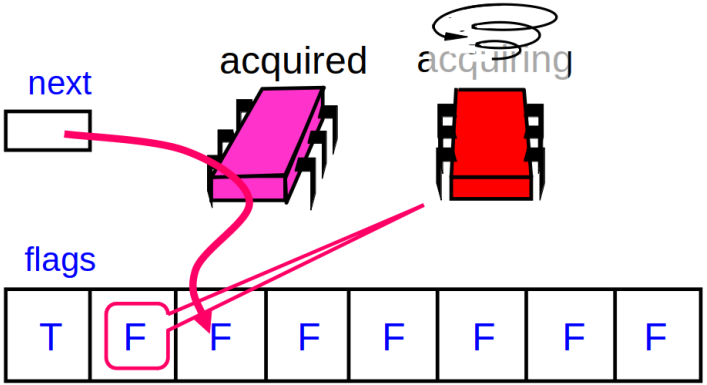
Anderson Queue Lock



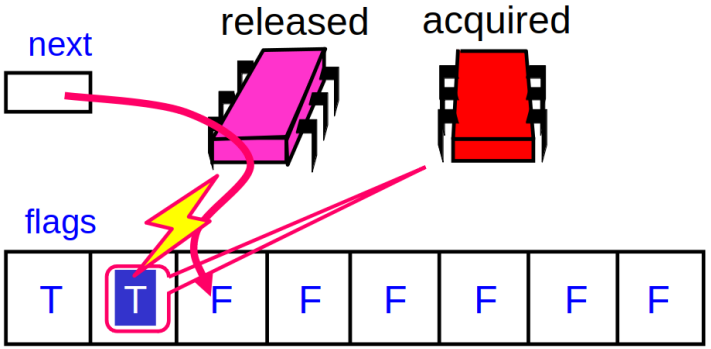
Anderson Queue Lock



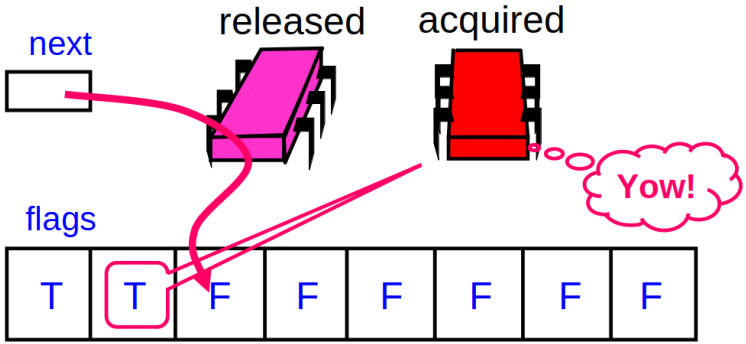
Anderson Queue Lock



Anderson Queue Lock



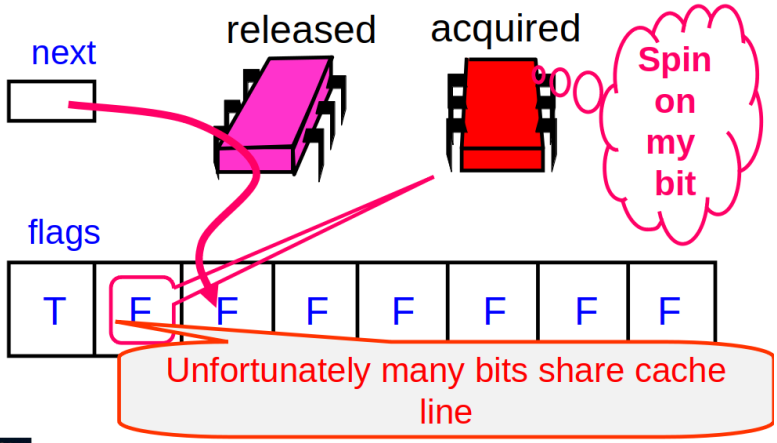
Anderson Queue Lock



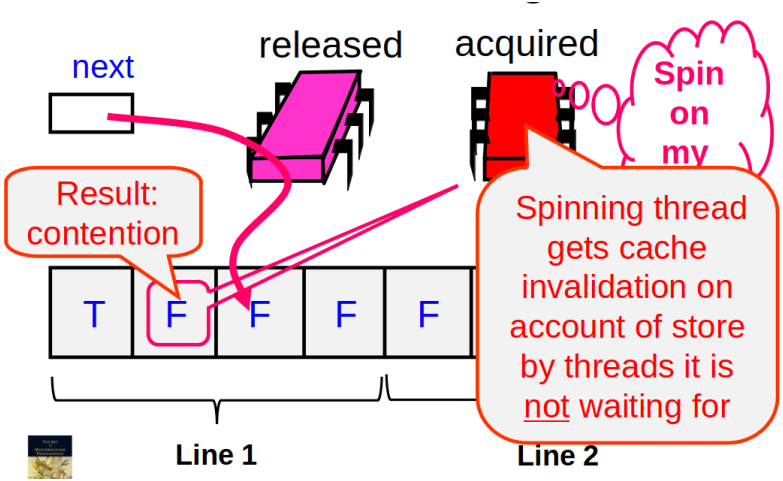
Anderson Queue Lock

```
class ALock implements Lock {
  boolean[] flags = {true,false, ...,false}; // one flag per thread
  AtomicInteger next = new AtomicInteger(0); // next flag to use
  ThreadLocal<Integer> mySlot;               // thread-local variable
  public lock() {
    mySlot = next.getAndIncrement();         // take next slot
    while (flags[mySlot % n] == false) {};   // await
    flags[mySlot % n] = false;               // prepare slot for re-use
  }
  public unlock() {
    flags[(mySlot + 1) % n] = true;          // tell next thread to go
  }
```

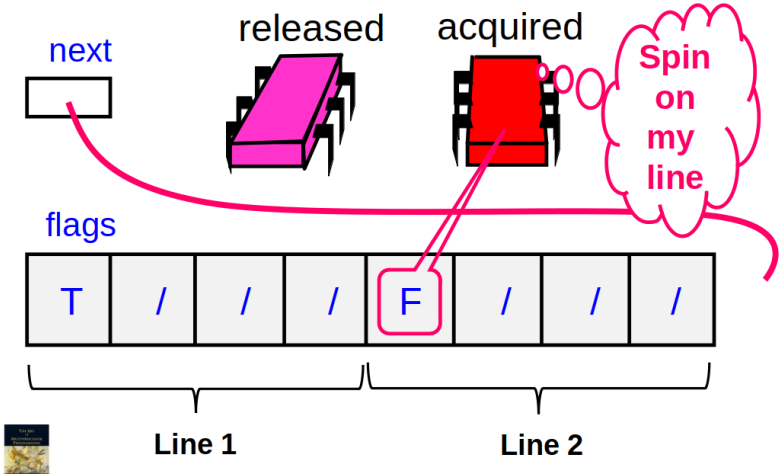
False sharing



False sharing



Solution: padding



Anderson Queue Lock

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, **Fairness** (FIFO)

Performance

- Memory overhead
  - cache line per competitor
  - $O(LN)$ where L = number of locks, N = number of threads
  - what if number of threads is unknown?
- Efficiency in contended mode, Scalability
  - First "truly scalable" lock in our course
- Backoff policy
  - Spin me right round, baby, right round

Anderson Queue Lock

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, **Fairness** (FIFO)

Performance

- Memory overhead
  - cache line per competitor
  - $O(LN)$ where L = number of locks, N = number of threads
  - what if number of threads is unknown?
- Efficiency in contended mode, Scalability
  - First "truly scalable" lock in our course
- Backoff policy
  - Spin me right round, baby, right round

Idea: use O(1) thread-local memory to decrease memory overhead

Lecture plan

CLH lock

- CLH – Craig, Landin, Hagersten
- FIFO
- every thread has thread-private node to be used as a part of a queue

CLH lock

- CLH – Craig, Landin, Hagersten
- FIFO
- every thread has thread-private node to be used as a part of a queue

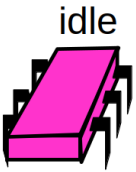
We are discussing simplified implementation in Garbage-collected language (Java) to avoid manual memory management in concurrent environment

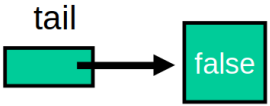
CLH lock

- CLH – Craig, Landin, Hagersten
- FIFO
- every thread has thread-private node to be used as a part of a queue

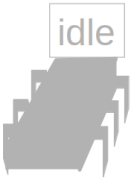
We are discussing simplified implementation in Garbage-collected language (Java) to avoid manual memory management in concurrent environment
Reminder: in Lecture 8 we broke lock-free stack by using node pooling (ABA problem)

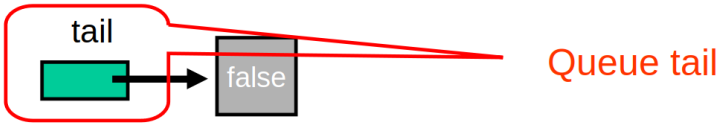
CLH lock: initialization



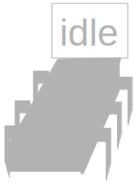


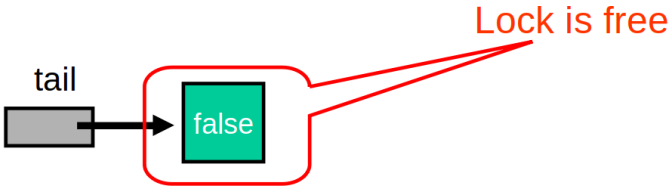
CLH lock: initialization



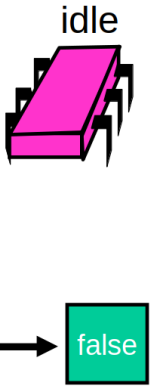


CLH lock: initialization

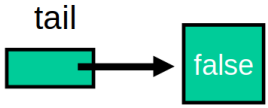




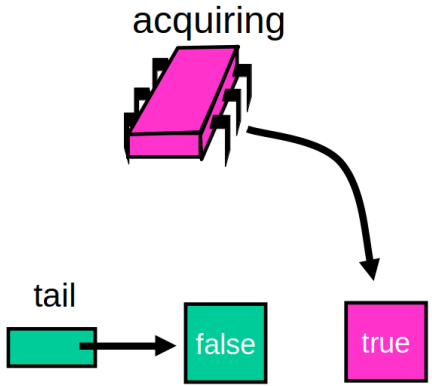
CLH lock: initialization



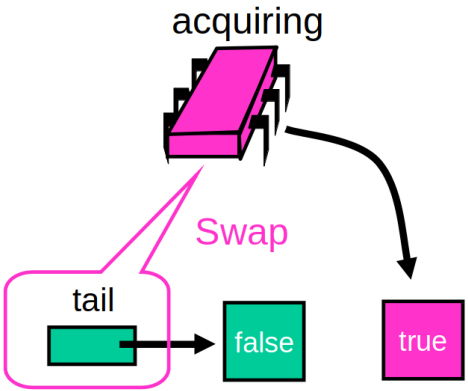
Purple Wants the Lock



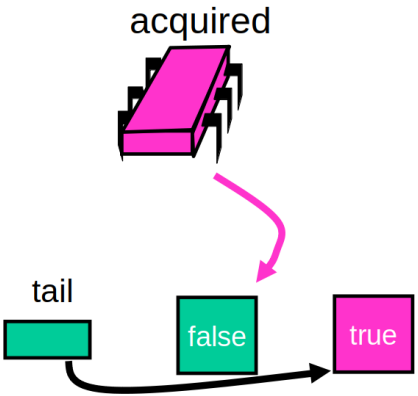
Purple Wants the Lock



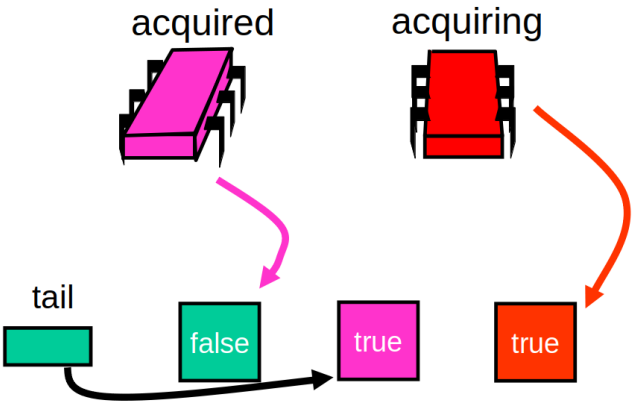
Purple Wants the Lock



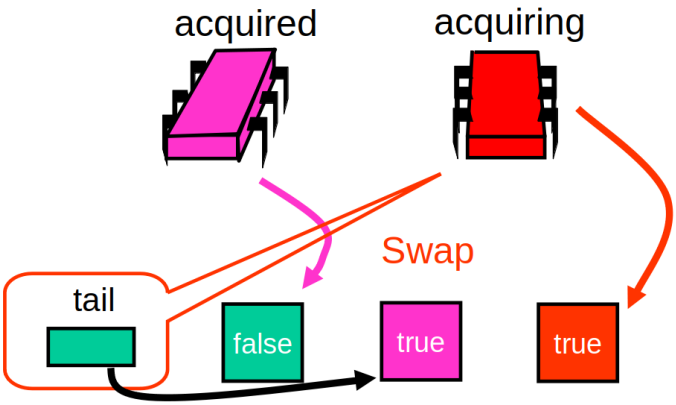
Purple Has the Lock



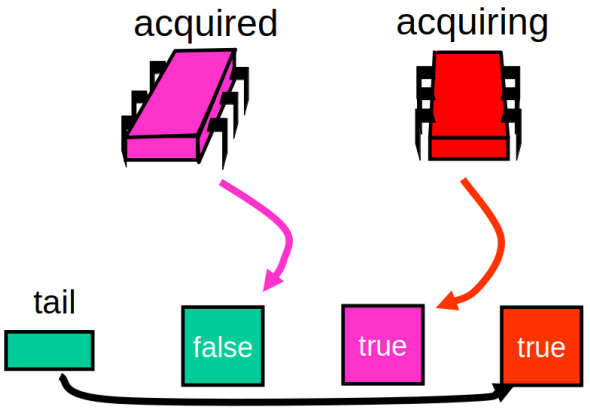
Red Wants the Lock



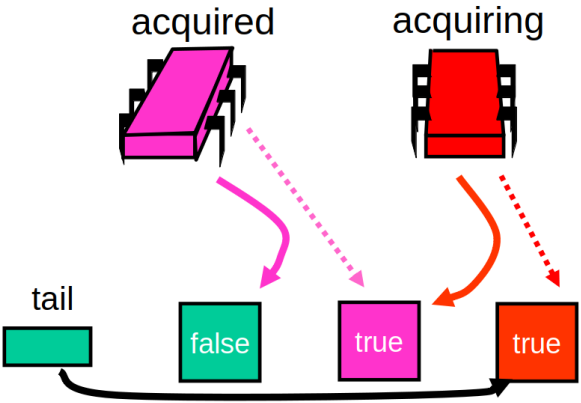
Red Wants the Lock



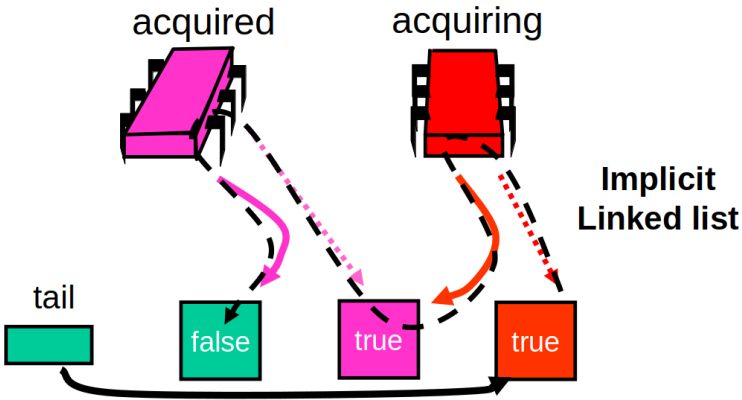
Red Wants the Lock



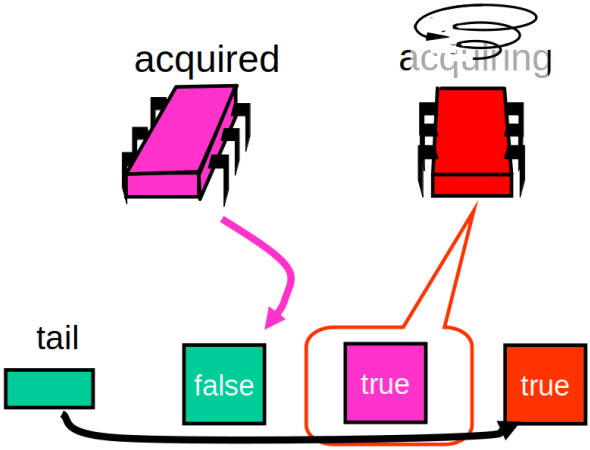
Red Wants the Lock



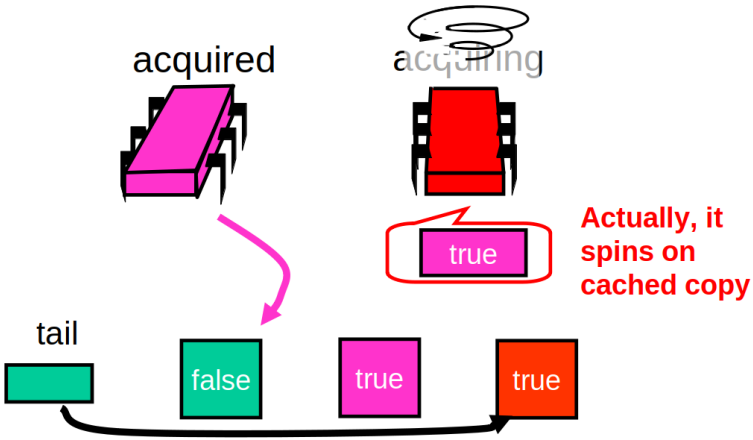
Red Wants the Lock



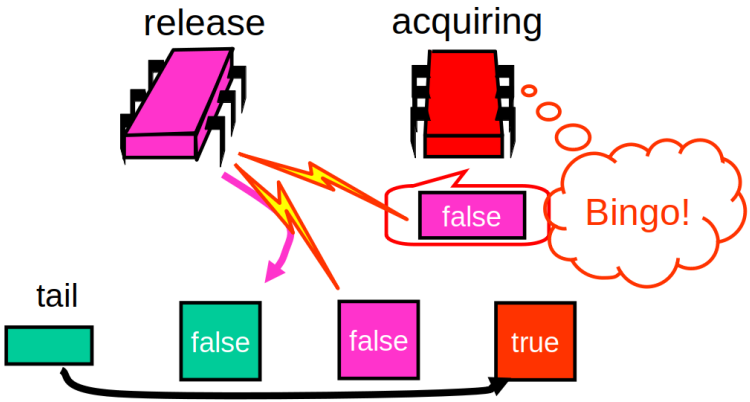
Red Wants the Lock



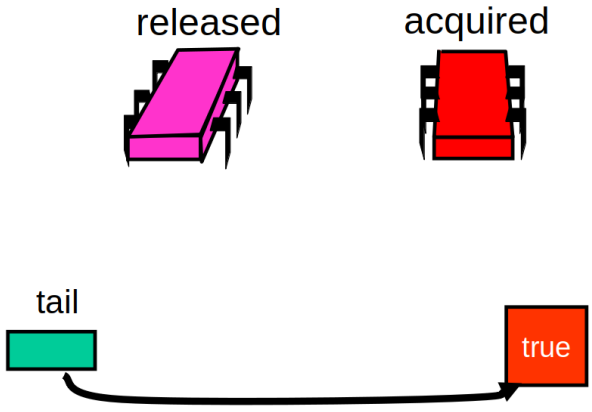
Red Wants the Lock



Purple Releases



Purple Releases



CLH

```
class Qnode { AtomicBoolean locked = new AtomicBoolean(); }
class CLHLock implements Lock {
  AtomicReference<Qnode> tail = new AtomicReference(new Qnode(false));
  ThreadLocal<Qnode> node;      // thread-local variable
  public void lock() {
    node.set(new Qnode(true)); // allocate my node as not released
    Qnode pred = tail.getAndSet(node.get()); // swap in my node
    while (pred.locked.get()) {} // spin until predecessor releases lock
  }
  public void unlock() {
    node.get().locked.set(false); // notify successor
    node.set(null);               // node is not needed by current thread
                                  // but could still be read by other threads
                                  // cannot `free` it
  }
```

CLH: summary

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, **Fairness** (FIFO)

Performance

- Memory overhead
  - fixed-size node per competitor
  - $O(L + N)$ where L = number of locks, N = number of threads
- Efficiency in contended mode, Scalability
  - Second "truly scalable" lock in our course
  - Spins on random **remote** memory location
- Backoff policy
  - Spin me right round, baby, right round

CLH: locality

Contender of CLH lock spins on random **remote** memory location

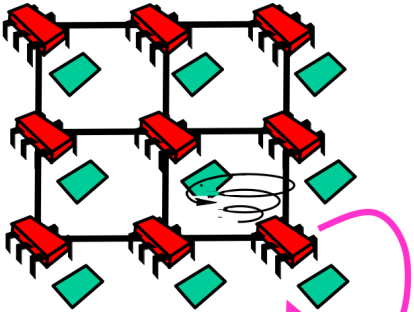
CLH: locality

Contender of CLH lock spins on random **remote** memory location
NUMA Architectures:

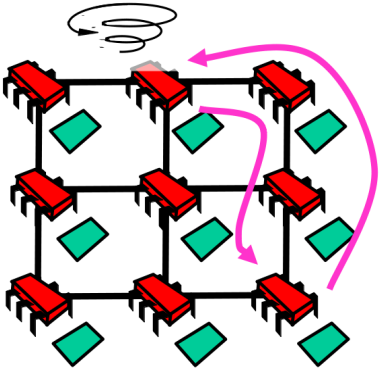
- **N**on-**U**niform **M**emory **A**rchitecture
- Illusion: flat shared memory
- Truth:
  - No caches (sometimes)
  - Some memory regions faster than others

NUMA and locking

Contender of CLH lock spins on random **remote** memory location



Spinning on local memory is fast

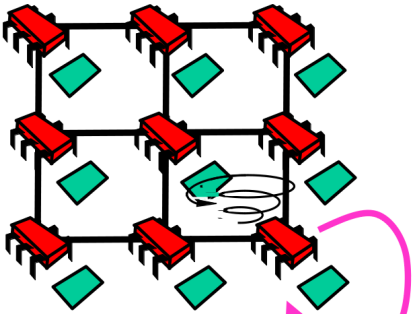


Spinning on remote memory is slow

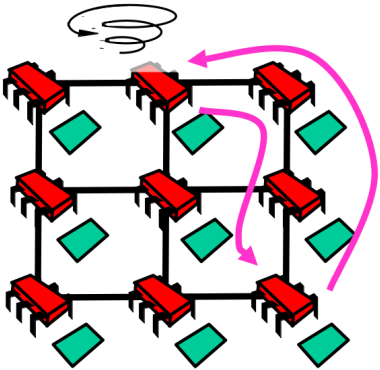
NUMA and locking

Contender of CLH lock spins on random **remote** memory location

Idea: spin on thread-private memory



Spinning on local memory is fast



Spinning on remote memory is slow

Lecture plan

MCS

- MCS – Mellor-Crummey and Scott
- FIFO
- every thread has thread-private node to be used as a part of a queue
- thread spins on this thread-local memory

MCS

- MCS – Mellor-Crummey and Scott
- FIFO
- every thread has thread-private node to be used as a part of a queue
- thread spins on this thread-local memory

We are discussing simplified implementation in Garbage-collected language (Java) to avoid manual memory management in concurrent environment

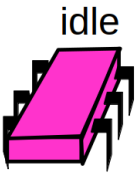
MCS

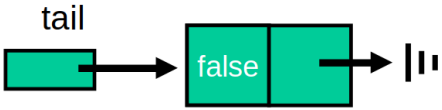
- MCS – Mellor-Crummey and Scott
- FIFO
- every thread has thread-private node to be used as a part of a queue
- thread spins on this thread-local memory

We are discussing simplified implementation in Garbage-collected language (Java) to avoid manual memory management in concurrent environment

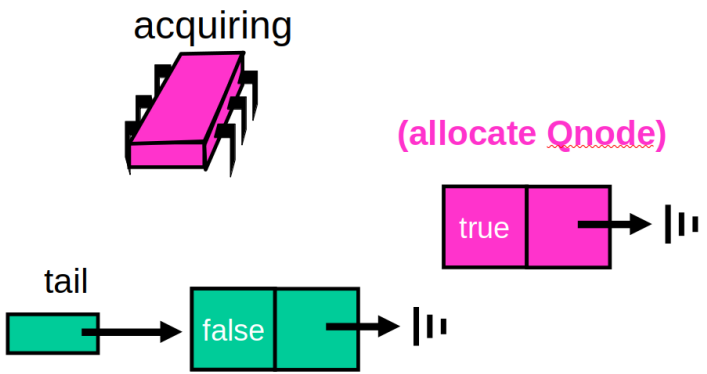
Reminder: in Lecture 8 we broke lock-free stack by using node pooling (ABA problem)

MCS lock: initialization

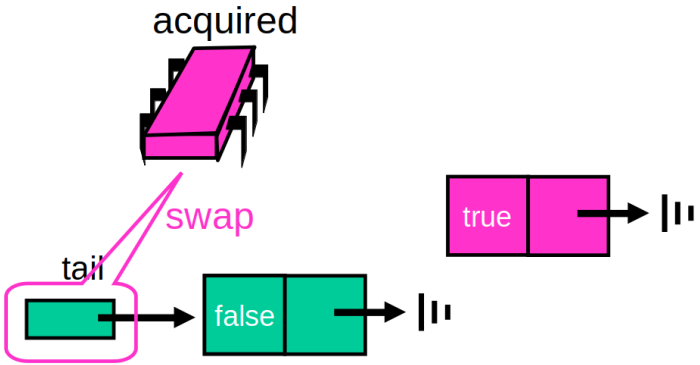




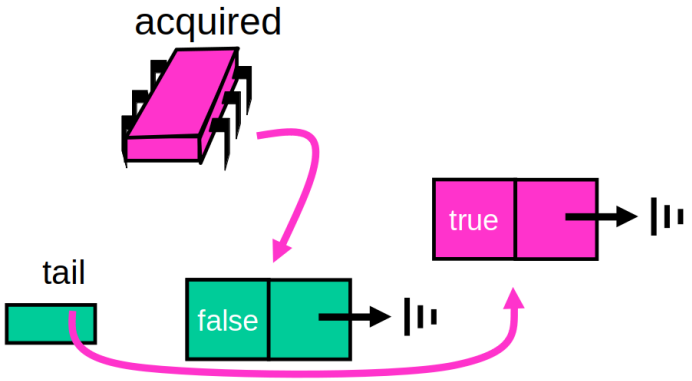
Acquiring



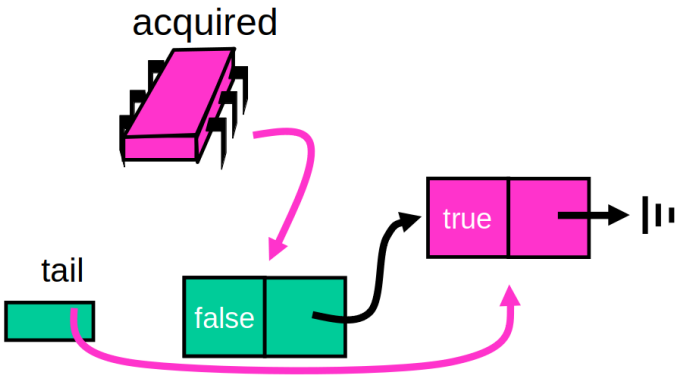
Acquiring



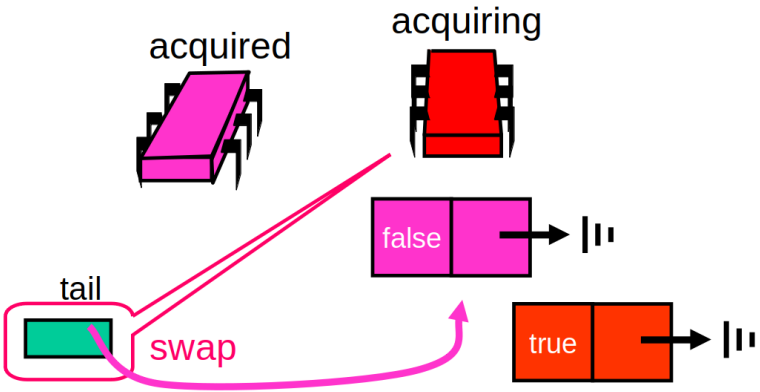
Acquiring



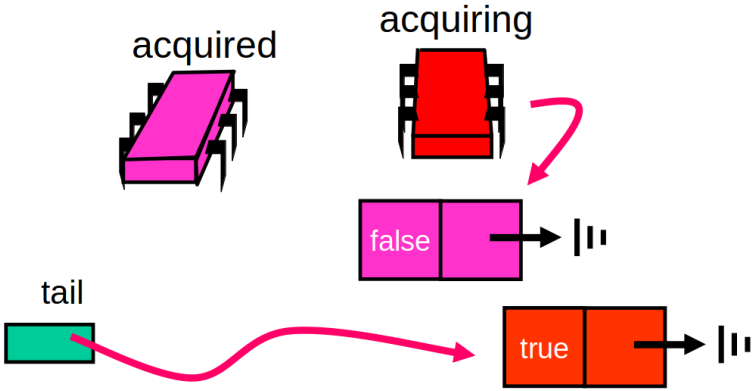
Acquired



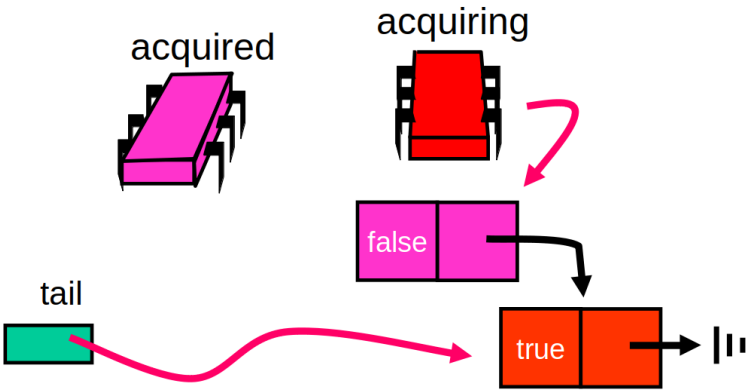
Acquiring



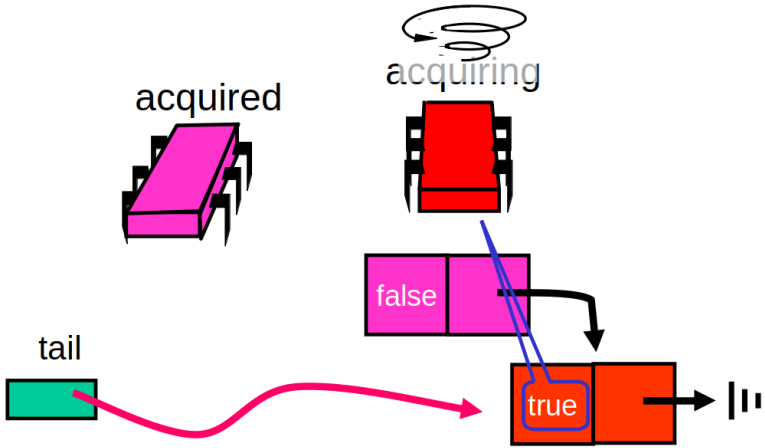
Acquiring



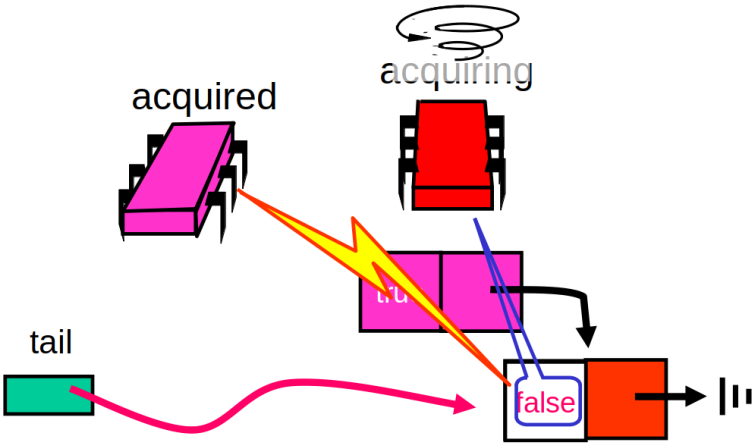
Acquiring



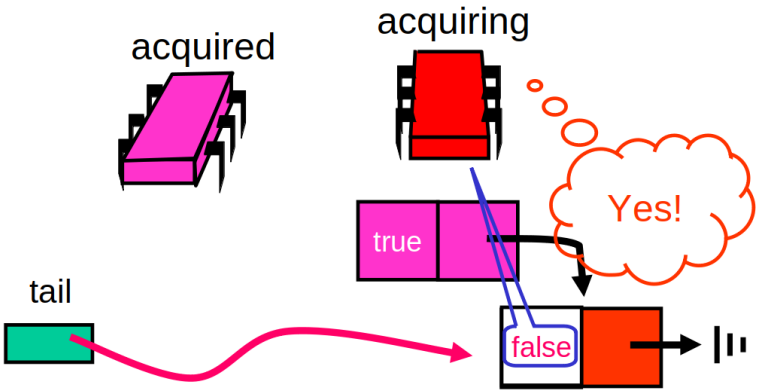
Acquiring



Acquiring



Acquiring



MCS: definitions

```
class Qnode {
  AtomicBoolean locked = new AtomicBoolean(false);
  AtomicReference<Qnode> next = new AtomicReference<Qnode>(null);
}
Qnode qnode, pred;
// shortcut: boolean assignment
  qnode.locked = true;    // pseudocode
  qnode.locked.set(true) // java code
// shortcut: reference assignment
  pred.next = qnode;      // pseudocode
  pred.next.set(qnode)    // java code
```

MCS: definitions

```
class Qnode {  var locked = new AtomicBoolean(false);
               var next = new AtomicReference<Qnode>(null); }
ThreadLocal<Qnode> node = ...
Qnode qnode, pred;
// shortcut: init thread locals
  node = new Qnode(...);     // pseudocode
  node.set(new Qnode(...)); // java code
// shortcut: use thread locals
  pred.next = node;           // pseudocode
  pred.next.set(node.get()); // java code
```

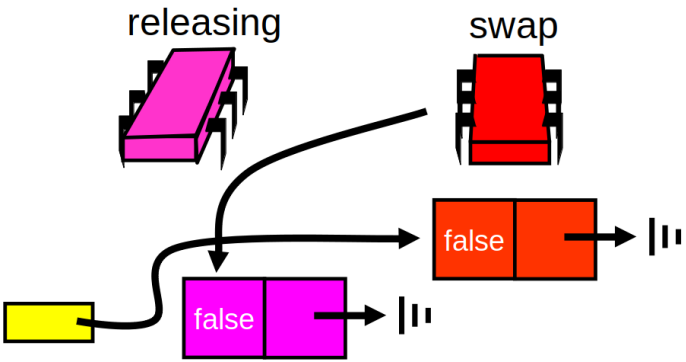
MCS: lock

```
class MCSLock implements Lock {
  AtomicReference<Qnode> tail = new AtomicReference<>(null);
  ThreadLocal<Qnode> node; // thread-local variable
  public void lock() {
    node = new Qnode(locked = false, next = null);
    Qnode pred = tail.getAndSet(node); // swap in my node
    if (pred != null) { // fix if queue was non-empty
      node.locked = true;
      pred.next = node; // now other thread could see my node
      while (node.locked) {} // wait until unlocked
    }
  }
}
```

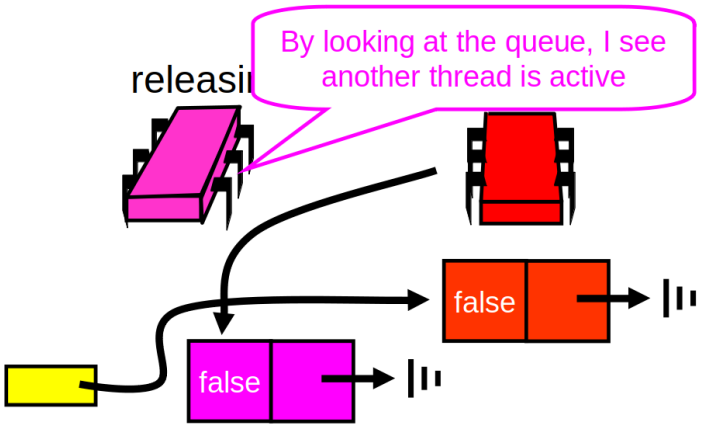
MCS: unlock

```
class MCSLock implements Lock {
  AtomicReference<Qnode> tail = new AtomicReference<>(null);
  ThreadLocal<Qnode> node; // thread-local variable
  public void unlock() {
    if (node.next == null) { // missing successor?
      if (tail.CAS(node, null)) return; // if really no successor, return
      while (node.next == null) {} // otherwise wait for successor to catch up
    }
    node.next.locked = false; // pass lock to successor
    node.set(null); // not needed by current thread
                    // but could be read by others
  }
}
```

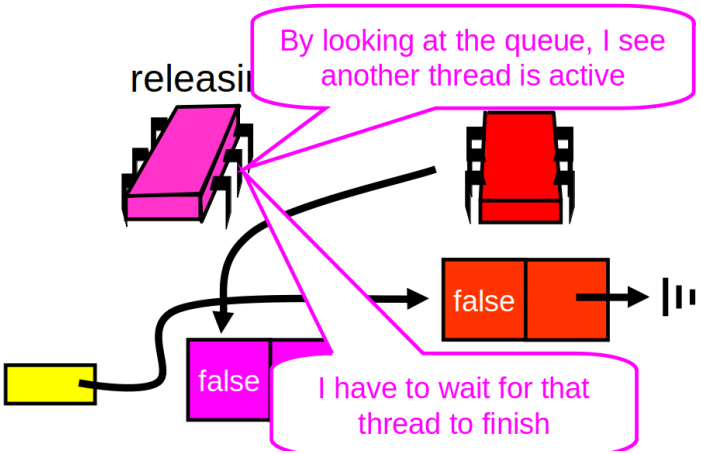
Purple release



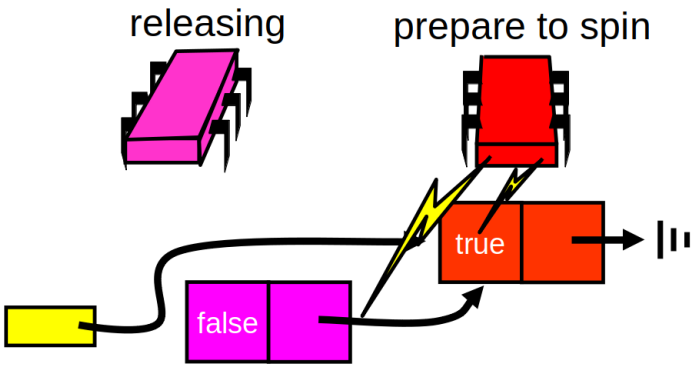
Purple release



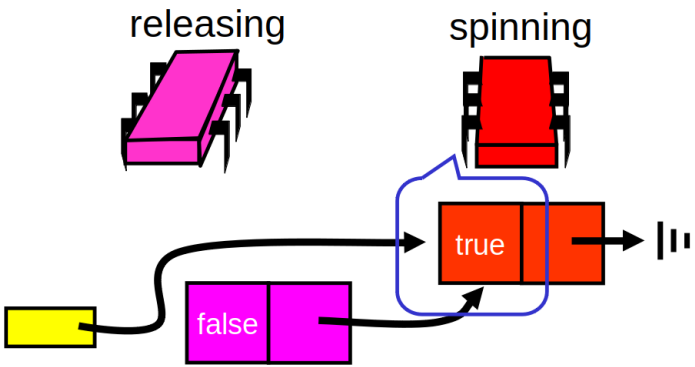
Purple release



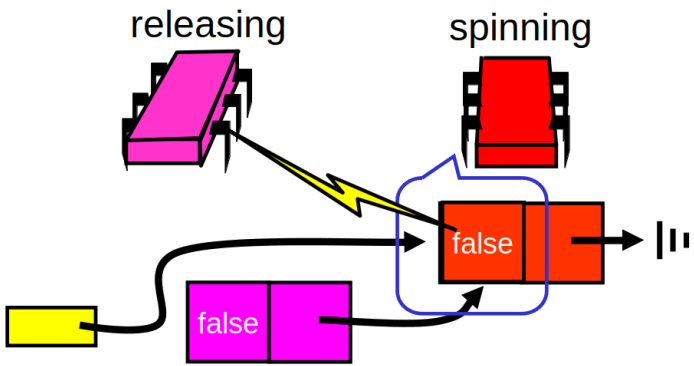
Purple release



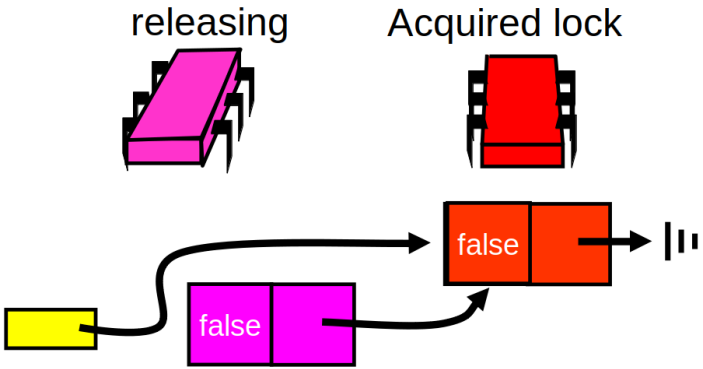
Purple release



Purple release



Purple release



MCS: summary

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, **Fairness** (FIFO)

Performance

- Memory overhead
  - fixed-size node per competitor
  - $O(L + N)$ where L = number of locks, N = number of threads
- Efficiency in contended mode, Scalability
  - Third "truly scalable" lock in our course
  - Spins on **local** memory location
- Backoff policy
  - Spin me right round, baby, right round

MCS: summary

Safety

- Mutual exclusion, Deadlock-freedom, Visibility

Progress

- Starvation-freedom, **Fairness** (FIFO)

Performance

- Memory overhead
  - fixed-size node per competitor
  - $O(L + N)$ where L = number of locks, N = number of threads
- Efficiency in contended mode, Scalability
  - Third "truly scalable" lock in our course
  - Spins on **local** memory location
- Backoff policy
  - Spin me right round, baby, right round

The best lock ever?

Lecture plan

Abortable locks

What if you want to give up waiting for a lock?

- Timeout
- Database transaction aborted by user
- Task of current thread was cancelled

Abortable locks

What if you want to give up waiting for a lock?

- Timeout
- Database transaction aborted by user
- Task of current thread was cancelled
- Lecture 4: cancellation is useful concurrent pattern

Abortable locks

What if you want to give up waiting for a lock?

- Timeout
- Database transaction aborted by user
- Task of current thread was cancelled
- Lecture 4: cancellation is useful concurrent pattern

TATAS: just return, no problem

Abortable locks

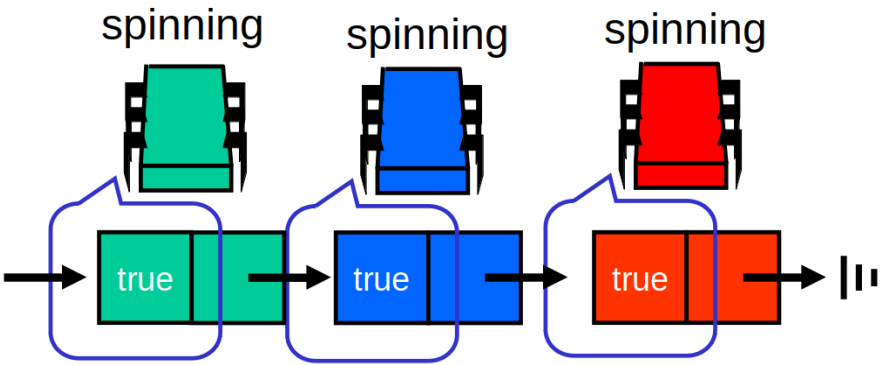
What if you want to give up waiting for a lock?

- Timeout
- Database transaction aborted by user
- Task of current thread was cancelled
- Lecture 4: cancellation is useful concurrent pattern

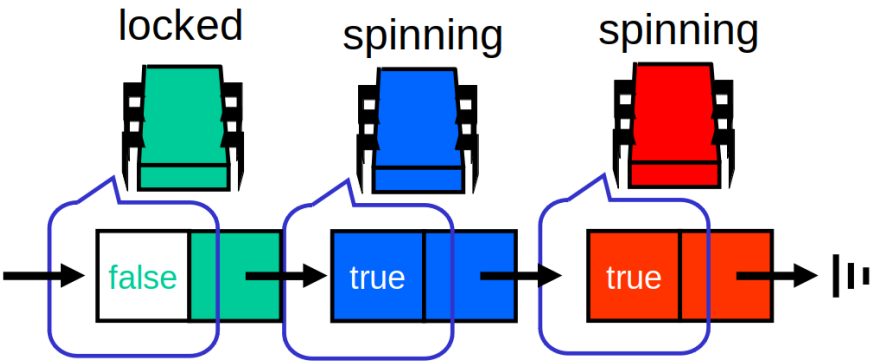
TATAS: just return, no problem
Queue-based locks: can't just quit

- Thread in line behind will starve
- Need a graceful way out

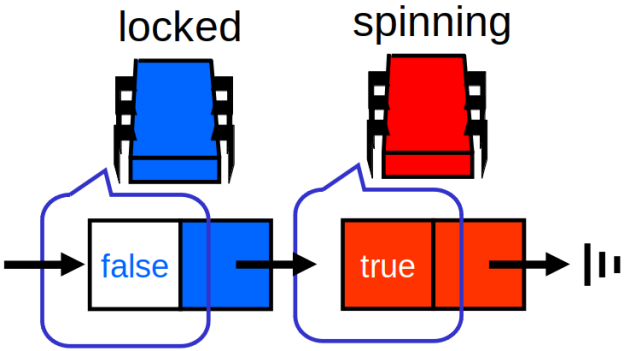
Abortable queue locks: problem



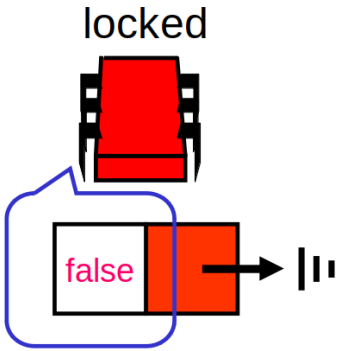
Abortable queue locks: problem



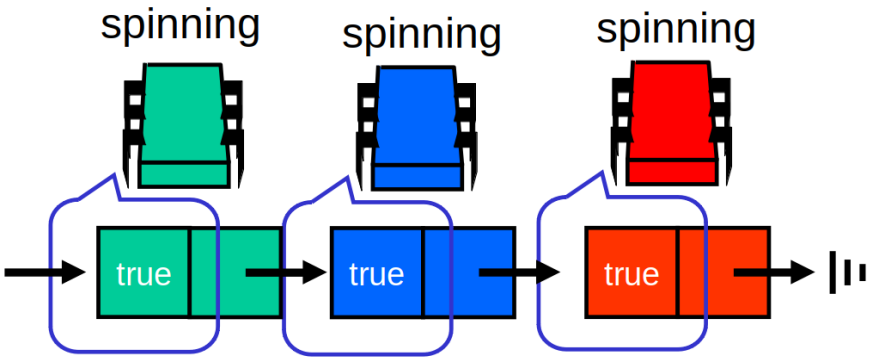
Abortable queue locks: problem



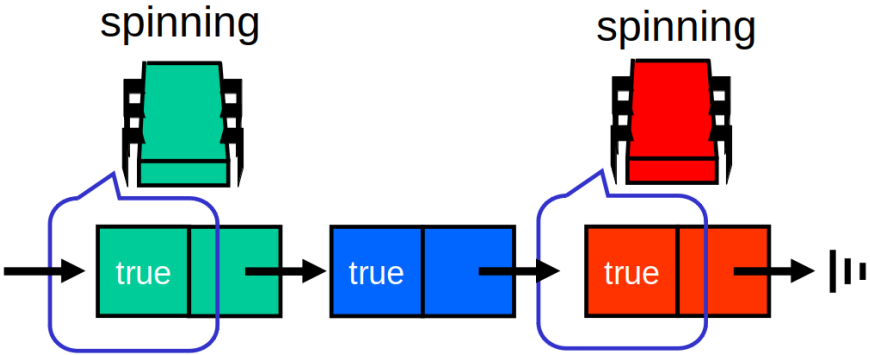
Abortable queue locks: problem



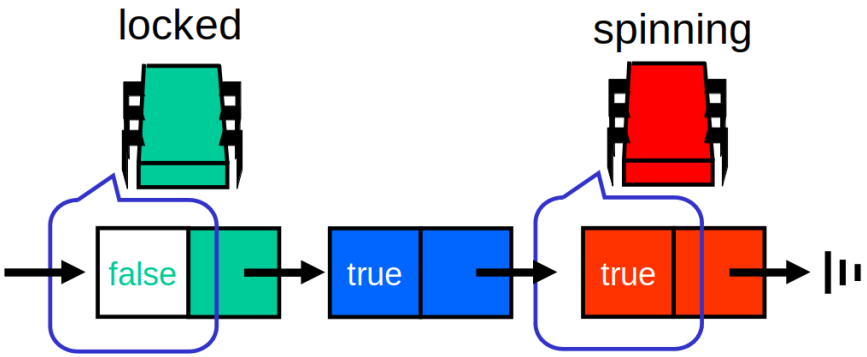
Abortable queue locks: problem



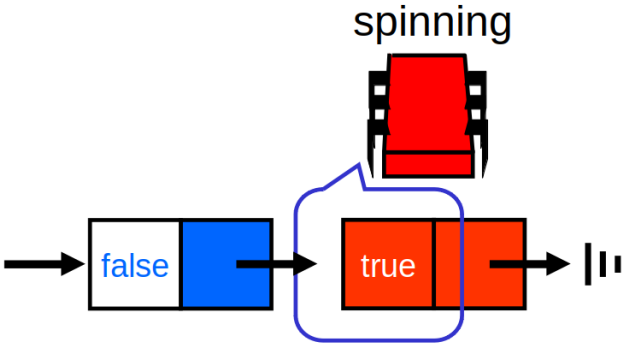
Abortable queue locks: problem



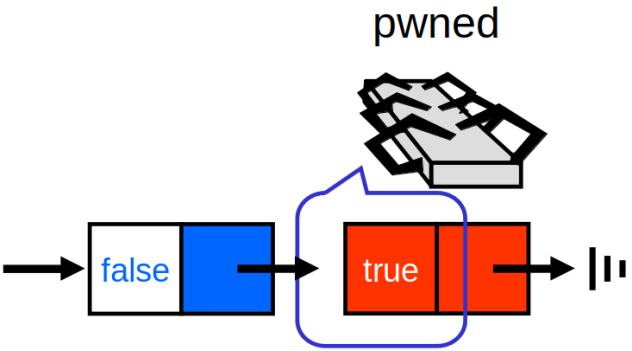
Abortable queue locks: problem



Abortable queue locks: problem



Abortable queue locks: problem



Abortable queue locks: solutions

Abortable queue locks: solutions

- Exist

Abortable queue locks: solutions

- Exist for some algorithms

Abortable queue locks: solutions

- Exist for some algorithms
- Have advantages and disadvantages

Abortable queue locks: solutions

- Exist for some algorithms
- Have advantages and disadvantages
- Add complexity

Abortable queue locks: solutions

- Exist for some algorithms
- Have advantages and disadvantages
- Add complexity

See Section 7.6 "A Queue Lock with Timeouts" in "The Art of Multiprocessor Programming"

Abortable queue locks: solutions

- Exist for some algorithms
- Have advantages and disadvantages
- Add complexity

See Section 7.6 "A Queue Lock with Timeouts" in "The Art of Multiprocessor Programming"
We will highly appreciate if you will not do THIS in your 3rd critical task

Lecture plan

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
  - reused node makes some `CAS` operation to succeed

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
  - reused node makes some `CAS` operation to succeed
  - which is incorrect, "identity" of node changed (it is "other" node)

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
  - Thread A observed node as "previous" in queue

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
  - Thread A observed node as "previous" in queue
    - saved to local variable

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
  - Thread A observed node as "previous" in queue
    - saved to local variable
    - plan to write `node.locked = false`

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
  - Thread A observed node as "previous" in queue
    - saved to local variable
    - plan to write `node.locked = false`
    - context switch

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
  - Thread A observed node as "previous" in queue
    - saved to local variable
    - plan to write `node.locked = false`
    - context switch
  - Thread B advanced queue, marked node as removed

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
  - Thread A observed node as "previous" in queue
    - saved to local variable
    - plan to write `node.locked = false`
    - context switch
  - Thread B advanced queue, marked node as removed
  - Thread C reused node: `node.locked = true`

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
  - Thread A observed node as "previous" in queue
    - saved to local variable
    - plan to write `node.locked = false`
    - context switch
  - Thread B advanced queue, marked node as removed
  - Thread C reused node: `node.locked = true`
  - Thread A wake-ups and "spoils" node

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
- Use-after-free

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
- Use-after-free
  - Thread A published thread-local node

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
- Use-after-free
  - Thread A published thread-local node
  - Thread B observed node as "previous" in queue

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
- Use-after-free
  - Thread A published thread-local node
  - Thread B observed node as "previous" in queue
    - saved to local variable

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
- Use-after-free
  - Thread A published thread-local node
  - Thread B observed node as "previous" in queue
    - saved to local variable
    - plan to read

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
- Use-after-free
  - Thread A published thread-local node
  - Thread B observed node as "previous" in queue
    - saved to local variable
    - plan to read
    - context switch

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
- Use-after-free
  - Thread A published thread-local node
  - Thread B observed node as "previous" in queue
    - saved to local variable
    - plan to read
    - context switch
  - Thread A entered lock

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
- Use-after-free
  - Thread A published thread-local node
  - Thread B observed node as "previous" in queue
    - saved to local variable
    - plan to read
    - context switch
  - Thread A entered lock
  - Thread A released lock

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
- Use-after-free
  - Thread A published thread-local node
  - Thread B observed node as "previous" in queue
    - saved to local variable
    - plan to read
    - context switch
  - Thread A entered lock
  - Thread A released lock
  - Thread A finished

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
- Use-after-free
  - Thread A published thread-local node
  - Thread B observed node as "previous" in queue
    - saved to local variable
    - plan to read
    - context switch
  - Thread A entered lock
  - Thread A released lock
  - Thread A finished
  - Thread B reads from already freed memory

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
- Use-after-free

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem
- Inconsistent state (pending write)
- Use-after-free
- Many more...

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem, Inconsistent state (pending write), Use-after-free, ...

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem, Inconsistent state (pending write), Use-after-free, ...

Automatic memory management tremendously simplify development of concurrent algorithms

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem, Inconsistent state (pending write), Use-after-free, ...

Automatic memory management tremendously simplify development of concurrent algorithms

- There are implementations of CLH/MCS with node reuse[1]

[1] Chapter 7 in "The Art of Multiprocessor Programming"

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem, Inconsistent state (pending write), Use-after-free, ...

Automatic memory management tremendously simplify development of concurrent algorithms

- There are implementations of CLH/MCS with node reuse[1]
- Developing custom algorithms is hard

[1] Chapter 7 in "The Art of Multiprocessor Programming"

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem, Inconsistent state (pending write), Use-after-free, ...

Automatic memory management tremendously simplify development of concurrent algorithms

- There are implementations of CLH/MCS with node reuse[1]
- Developing custom algorithms is hard see KSUH story
  - Hydraconf "Java Path Finder: going to Mars without bugs and deadlocks"[2]
  - Heisenbug "Java Path Finder: летим на Марс без багов и дедлоков"[3]

[1] Chapter 7 in "The Art of Multiprocessor Programming"

[2] https://youtu.be/dgHbSL_aDsO?si=vi81xieQ4zKRECQG

[3] https://youtu.be/sQSwShW_IlI?si=ZMlKKLQxMZYhk1T7

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem, Inconsistent state (pending write), Use-after-free, ...

Automatic memory management tremendously simplify development of concurrent algorithms

- There are implementations of CLH/MCS with node reuse
- Developing custom algorithms is hard

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem, Inconsistent state (pending write), Use-after-free, ...

Automatic memory management tremendously simplify development of concurrent algorithms

- There are implementations of CLH/MCS with node reuse
- Developing custom algorithms is hard

My opinion: **advanced** concurrency in languages with manual memory management is forbidden forest for you

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem, Inconsistent state (pending write), Use-after-free, ...

Automatic memory management tremendously simplify development of concurrent algorithms

- There are implementations of CLH/MCS with node reuse
- Developing custom algorithms is hard

My opinion: **advanced** concurrency in languages with manual memory management is forbidden forest for you

- "Atomic weapons" from Herb Sutter is a good way to dive in[4]

[4] https://youtu.be/A8eCGOqgvH4

Reusing node from queue-based lock

Common problems with memory management in concurrent environments

- ABA problem, Inconsistent state (pending write), Use-after-free, ...

Automatic memory management tremendously simplify development of concurrent algorithms

- There are implementations of CLH/MCS with node reuse
- Developing custom algorithms is hard

My opinion: **advanced** concurrency in languages with manual memory management is forbidden forest for you

- "Atomic weapons" from Herb Sutter is a good way to dive in[4]
- Using `std::shared_ptr` everywhere could be a bad idea[5]

[4] https://youtu.be/A8eCGOqgvH4

[5] https://pvs-studio.ru/ru/blog/posts/cpp/1174/#ID1DE383AA76

Lecture plan

Backoff policy revisited

```
class TATAS_Exponential_Backoff {
  void lock() {
    int delay = MIN_DELAY;
    while (true) {
      while (state.get() == LOCKED) {} // read-only polling
      if (lock.getAndSet(LOCKED) == UNLOCKED) return; // return on success
      sleep(random() % delay); // delay next write request on failure
      delay = Math.min(delay * 2, MAX_DELAY); // keep delay within bounds
    }}
  void unlock() { state.set(UNLOCKED); }
}
```

Backoff policy revisited

```
class TATAS_Exponential_Backoff {
  void lock() {
    int delay = MIN_DELAY;
    while (true) {
      while (state.get() == LOCKED) {} // read-only polling
      if (lock.getAndSet(LOCKED) == UNLOCKED) return; // return on success
      sleep(random() % delay); // delay next write request on failure
      delay = Math.min(delay * 2, MAX_DELAY); // keep delay within bounds
    }}
  void unlock() { state.set(UNLOCKED); }
}
```

Trade-off: time-to-react (latency) vs throughput (CPU)

Backoff policy revisited

```
class TATAS_Exponential_Backoff {
  void lock() {
    int delay = MIN_DELAY;
    while (true) {
      while (state.get() == LOCKED) {} // read-only polling
      if (lock.getAndSet(LOCKED) == UNLOCKED) return; // return on success
      sleep(random() % delay); // delay next write request on failure
      delay = Math.min(delay * 2, MAX_DELAY); // keep delay within bounds
    }}
  void unlock() { state.set(UNLOCKED); }
}
```

Trade-off: time-to-react (latency) vs throughput (CPU)

Idea: wake-up as soon as lock ready but not earlier

Question time

Question: Which concurrent primitive helps one thread to inform other thread that some event has happened?

Spin-then-park: idea

`LockSupport.park`, `LockSupport.unpark`[6]

```
boolean tryLock() { return lock.getAndSet(LOCKED) == UNLOCKED; }
void lock() {
  try { while (true) {
    for (int i = 0; i < SPIN; i++) { if (tryLock()) return; }
    enterSet.registerWaiter(currentThread()); // "wake-up me on unlock"
    currentThread().park(); // sleep
  }} finally { enterSet.removeIfPresent(currentThread()); }}
void unlock() {
  state.set(UNLOCKED);
  enterSet.unparkAllWaiters(); } // wake-up all registered threads
```

[6] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/LockSupport.html

Spin-then-park: idea

`LockSupport.park, LockSupport.unpark`[6]

```
boolean tryLock() { return lock.getAndSet(LOCKED) == UNLOCKED; }
void lock() {
  try { while (true) {
    for (int i = 0; i < SPIN; i++) { if (tryLock()) return; }
    enterSet.registerWaiter(currentThread()); // "wake-up me on unlock"
    currentThread().park(); // sleep
  }} finally { enterSet.removeIfPresent(currentThread()); }}
void unlock() {
  state.set(UNLOCKED);
  enterSet.unparkAllWaiters(); } // wake-up all registered threads
```

Do you see thundering herd problem here?

[6] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/LockSupport.html

Spin-then-park: idea

```
boolean tryLock() { return lock.getAndSet(LOCKED) == UNLOCKED; }
void lock() {
  try { while (true) {
    for (int i = 0; i < SPIN; i++) { if (tryLock()) return; }
    enterSet.registerWaiter(currentThread()); // "wake-up me on unlock"
    currentThread().park(); // sleep
  }} finally { enterSet.removeIfPresent(currentThread()); }}
void unlock() {
  state.set(UNLOCKED);
  enterSet.unparkOneWaiter(); } // wake-up one of registered threads
```

Spin-then-park: idea

```
boolean tryLock() { return lock.getAndSet(LOCKED) == UNLOCKED; }
void lock() {
  try { while (true) {
    for (int i = 0; i < SPIN; i++) { if (tryLock()) return; }
    enterSet.registerWaiter(currentThread()); // "wake-up me on unlock"
    currentThread().park(); // sleep
  }} finally { enterSet.removeIfPresent(currentThread()); }}
void unlock() {
  state.set(UNLOCKED);
  enterSet.unparkOneWaiter(); } // wake-up one of registered threads
```

Do you see race condition here?

Check-then-act race condition

Race condition:

- Thread A sees occupied lock (`check`), decides to park
- Thread B releases lock, notifies all pending threads (e.g. nobody)
- Thread A sleeps on condition variable (act)

Question time

Question: Concurrent protocol "fires" message in inappropriate time (when `EnterSet` is empty). How do we call it?

Check-then-act

One of the most common high-level errors in concurrent algorithms

Check-then-act

One of the most common high-level errors in concurrent algorithms

- Thread A observes system state (`check`)

Check-then-act

One of the most common high-level errors in concurrent algorithms

- Thread A observes system state (`check`)
- Thread B changes system state

Check-then-act

One of the most common high-level errors in concurrent algorithms

- Thread A observes system state (`check`)
- Thread B changes system state
- Thread A does something (`act`) which is incorrect for new system state

Check-then-act

One of the most common high-level errors in concurrent algorithms

- Thread A observes system state (`check`)
- Thread B changes system state
- Thread A does something (`act`) which is incorrect for new system state

Solutions:

Check-then-act

One of the most common high-level errors in concurrent algorithms

- Thread A observes system state (`check`)
- Thread B changes system state
- Thread A does something (`act`) which is incorrect for new system state

Solutions:

- atomic transaction (check-then-act under lock)

Check-then-act

One of the most common high-level errors in concurrent algorithms

- Thread A observes system state (`check`)
- Thread B changes system state
- Thread A does something (`act`) which is incorrect for new system state

Solutions:

- atomic transaction (check-then-act under lock)
  - but we are implementing mutex right now!

Check-then-act

One of the most common high-level errors in concurrent algorithms

- Thread A observes system state (`check`)
- Thread B changes system state
- Thread A does something (`act`) which is incorrect for new system state

Solutions:

- atomic transaction (check-then-act under lock)
  - but we are implementing mutex right now!
- use RMW operations (e.g. `compareAndExchange`)

Check-then-act

One of the most common high-level errors in concurrent algorithms

- Thread A observes system state (`check`)
- Thread B changes system state
- Thread A does something (`act`) which is incorrect for new system state

Solutions:

- atomic transaction (check-then-act under lock)
  - but we are implementing mutex right now!
- use RMW operations (e.g. `compareAndExchange`)
  - does not work for compound state (multiple variables)

Check-then-act

One of the most common high-level errors in concurrent algorithms

- Thread A observes system state (`check`)
- Thread B changes system state
- Thread A does something (`act`) which is incorrect for new system state

Solutions:

- atomic transaction (check-then-act under lock)
  - but we are implementing mutex right now!
- use RMW operations (e.g. `compareAndExchange`)
  - does not work for compound state (multiple variables)
- algorithm tolerates "inconsistent actions"

Check-then-act

One of the most common high-level errors in concurrent algorithms

- Thread A observes system state (`check`)
- Thread B changes system state
- Thread A does something (`act`) which is incorrect for new system state

Solutions:

- atomic transaction (check-then-act under lock)
  - but we are implementing mutex right now!
- use RMW operations (e.g. `compareAndExchange`)
  - does not work for compound state (multiple variables)
- algorithm tolerates "inconsistent actions"
  - "missed signal is not a problem"

Check-then-act

One of the most common high-level errors in concurrent algorithms

- Thread A observes system state (`check`)
- Thread B changes system state
- Thread A does something (`act`) which is incorrect for new system state

Solutions:

- atomic transaction (check-then-act under lock)
  - but we are implementing mutex right now!
- use RMW operations (e.g. `compareAndExchange`)
  - does not work for compound state (multiple variables)
- algorithm tolerates "inconsistent actions"
  - "missed signal is not a problem"
- rework protocol

Check-then-act

One of the most common high-level errors in concurrent algorithms

- Thread A observes system state (`check`)
- Thread B changes system state
- Thread A does something (`act`) which is incorrect for new system state

Solutions:

- atomic transaction (check-then-act under lock)
  - but we are implementing mutex right now!
- use RMW operations (e.g. `compareAndExchange`)
  - does not work for compound state (multiple variables)
- algorithm tolerates "inconsistent actions"
  - "missed signal is not a problem"
- rework protocol
  - "never miss signal"

Spin-then-park with timeouts

"Missed signal is not a problem"

```
void lock() {
  try { while (true) {
    for (int i = 0; i < SPIN; i++) { if (tryLock()) return; }
    enterSet.registerWaiter(currentThread()); // "wake-up me on unlock"
    currentThread().park(1_000); // fixed time sleep
  }} finally { enterSet.removeIfPresent(currentThread()); }}
void unlock() {
  state.set(UNLOCKED);
  enterSet.unparkOneWaiter(); } // wake-up one of registered threads
```

Spin-then-park with timeouts

"Missed signal is not a problem"

```
void lock() {
  try { while (true) {
    for (int i = 0; i < SPIN; i++) { if (tryLock()) return; }
    enterSet.registerWaiter(currentThread()); // "wake-up me on unlock"
    currentThread().park(1_000); // fixed time sleep
  }} finally { enterSet.removeIfPresent(currentThread()); }}
void unlock() {
  state.set(UNLOCKED);
  enterSet.unparkOneWaiter(); } // wake-up one of registered threads
```

Problem: many waiting threads will repeatedly wake-up and sleep

Spin-then-park with double check

"Never miss signal"

```
void lock() {
  try { while (true) {
    for (int i = 0; i < SPIN; i++) { if (tryLock()) return; }
    enterSet.registerWaiter(currentThread()); // "wake-up me on unlock"
    if (tryLock()) return;  // maybe unlock happened before register?
    currentThread().park(); // sleep and be sure not to miss notification
  }} finally { enterSet.removeIfPresent(currentThread()); }}
void unlock() {
  state.set(UNLOCKED);
  enterSet.unparkOneWaiter(); } // wake-up one of registered threads
```

Spin-then-park with double check

"Never miss signal"

```
void lock() {
  try { while (true) {
    for (int i = 0; i < SPIN; i++) { if (tryLock()) return; }
    enterSet.registerWaiter(currentThread()); // "wake-up me on unlock"
    if (tryLock()) return;  // maybe unlock happened before register?
    currentThread().park(); // sleep and be sure not to miss notification
  }} finally { enterSet.removeIfPresent(currentThread()); }}
void unlock() {
  state.set(UNLOCKED);
  enterSet.unparkOneWaiter(); } // wake-up one of registered threads
```

Requires intricate reasoning for all possible interleavings of

- `tryLock`, `registerWaiter`, `park`, `unparkOneWaiter`

and assumes all of them are linearizable

Summary

Advanced alternatives for single-location TATAS

- Array-based lock: Anderson Queue Lock (spin on cache line indexed by `AtomicInteger`)
- Queue-based lock: CLH (spin on remote location, implicit linked list)
- Queue-based lock: MCS (spin on local location, explicit linked list)

Challenges with advanced locks

- cancellation/timeout support
- manual memory management

Spin-then-park optimization

- Better utilization of CPU and scheduling quantum
- Check-then-act race conditions should be avoided in implementation