Hey, old friend, are you OK old friend?

- Atomic snapshot
  - write to one entry of an array
  - read multiple entries atomically
- Consensus number?

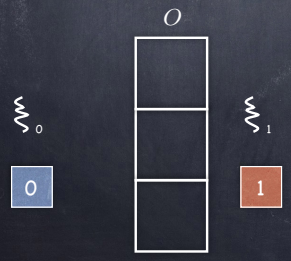
  1

- Multiple assignment
  - write atomically from m out of n entries
  - read from any one entry
- Consensus number?

Atomic registers are not enough

Theorem There is no wait-free implementation of an (m, n)-assignment object by atomic registers for any $n > m > 1$

Show that a (2,3)-assignment object solves 2-thread consensus



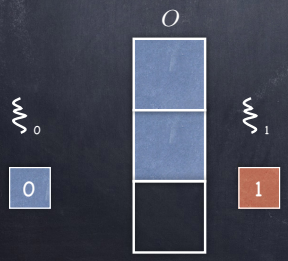
Proof. To decide, T_0 writes atomically to locations 0 and 1; T_1 to locations 1 and 2

- if T_0 and not T_1, then $O[2]$ is empty

Atomic registers are not enough

Theorem There is no wait-free implementation of an (m, n)-assignment object by atomic registers for any $n > m > 1$

Show that a (2,3)-assignment object solves 2-thread consensus



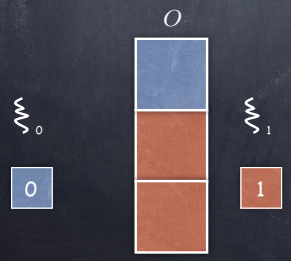
Proof. To decide, T_0 writes atomically to locations 0 and 1; T_1 to locations 1 and 2

- if T_0 and not T_1, then $O[2]$ is empty
  - T_0 decides on its input value
- if T_0 and later T_1, then $O[1]$ and $O[2]$ are red

Atomic registers are not enough

Theorem There is no wait-free implementation of an (m, n)-assignment object by atomic registers for any $n > m > 1$

Show that a (2,3)-assignment object solves 2-thread consensus



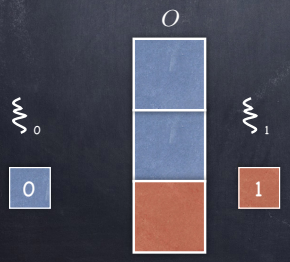
Proof. To decide, T_0 writes atomically to locations 0 and 1; T_1 to locations 1 and 2

- if T_0 and not T_1, then $O[2]$ is empty
  - T_0 decides on its input value
- if T_0 and later T_1, then $O[1]$ and $O[2]$ are red
  - T_0 decides on its input value
- if T_1 and later T_0, then $O[1]$ blue and $O[2]$ red

Atomic registers are not enough

Theorem There is no wait-free implementation of an (m, n)-assignment object by atomic registers for any $n > m > 1$

Show that a (2,3)-assignment object solves 2-thread consensus



Proof. To decide, T_0 writes atomically to locations 0 and 1; T_1 to locations 1 and 2

- if T_0 and not T_1, then $O[2]$ is empty
  - T_0 decides on its input value
- if T_0 and later T_1, then $O[1]$ and $O[2]$ are red
  - T_0 decides on its input value
- if T_1 and later T_0, then $O[1]$ blue and $O[2]$ red
  - T_0 decides on T_1's input value

Hey, old friend, are you OK old friend?

- Atomic snapshot
  - write to one entry of an array
  - read multiple entries atomically
- Consensus number?

  1

- Multiple assignment
  - write atomically from m out of n entries
  - read from any one entry
- Consensus number?

Theorem Atomic $\left(n, \frac{n(n+1)}{2}\right)$-register assignment for $n > 1$ has consensus number at least n

Take away

Writing atomically requires more computational power than reading atomically

Read-Modify-Write registers

- RMW method for a function set $\mathcal{F}$
  - replaces current register value v with $f(v)$ for $f \in \mathcal{F}$
  - returns register's original value v

getAndset(v): sets register value to v; returns old value

fetch-and-increment(): adds 1 to register value; returns old value

fetch-and-add(k): adds k to register value; returns old value

compare-and-swap(e, u): if register value is equal to e, set to u
otherwise, leave unchanged
returns Boolean to indicate if value changed

Trivialities

- An RMW method for f is non-trivial if there exists v such that $f(v) \neq v$

Theorem Any non-trivial RMW register has consensus number at least 2

Proof. Initialize register r to $v \neq f(v)$

```
public T decide (T value) {
  int i := ThreadID.get();
  propose(value);
  if (r.rmw = v)
    return proposed[i];
  else
    return propose[1-i];
}
```

Interfering RMW

- Let $\mathcal{F}$ be a set of functions such that for all f_i and f_j, either
  - Commute: $f_i(f_j(v)) = f_j(f_i(v))$
  - Overwrite: $f_i(f_j(v)) = f_i(v)$

Interfering RMW

- Let $\mathcal{F}$ be a set of functions such that for all f_i and f_j, either
  - Commute: $f_i(f_j(v)) = f_j(f_i(v))$
  - Overwrite: $f_i(f_j(v)) = f_i(v)$
- Examples
  - test-and-set: getAndset(1) $f(v) = 1$
    - Overwrite $f_i(f_j(v)) = f_i(v)$
  - swap: getAndset(x) $f(v, x) = x$
    - Overwrite $f_i(f_j(v)) = f_i(v)$
  - fetch-and-inc $f(v) = v + 1$
    - Commute $f_i(f_j(v)) = f_j(f_i(v))$

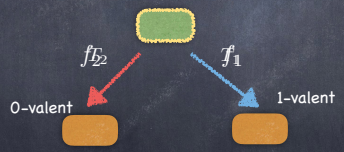
Consensus number of interfering RMW

Theorem Any set of RMW registers whose $\mathcal{F}$ commute or overwrite has consensus number 2

Consensus number of interfering RMW

Theorem Any set of RMW registers whose $\mathcal{F}$ commute or overwrite has consensus number 2

Proof Suppose we had a protocol for 3-thread consensus.
Let's focus on the critical state



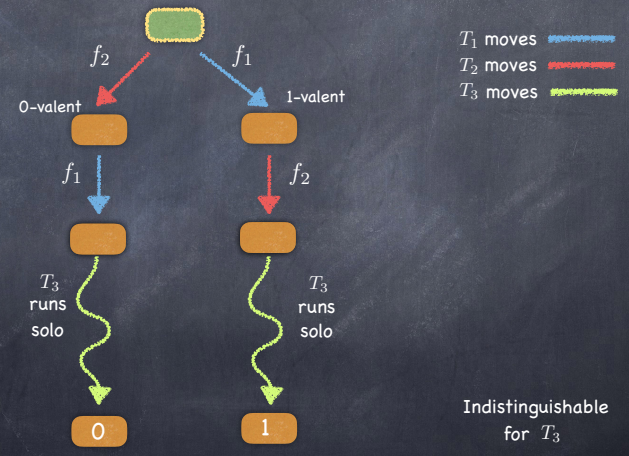
- The methods cannot be reads or writes
- The methods cannot be on separate objects

  or easy contradiction

- Must be RMW methods on a single register

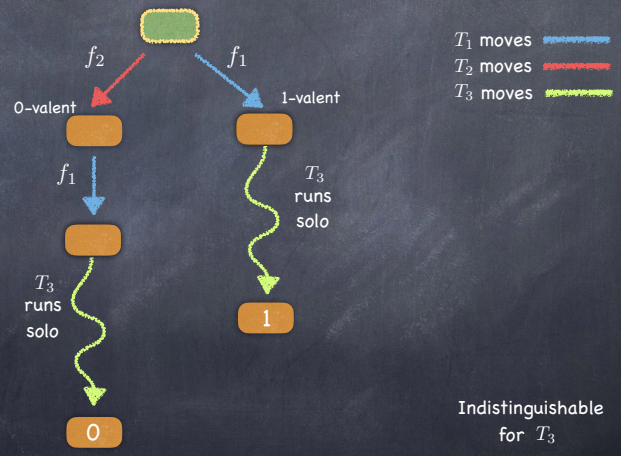
Maybe functions commute

$$f_i(f_j(v)) = f_j(f_i(v))$$



Maybe functions overwrite

$$f_i(f_j(v)) = f_i(v)$$



Impact

- Many early machines provide these "weak" RMW instructions
  - test-and-set (IBM 360)
  - fetch-and-add (NYU Ultracomputer)
  - swap (original SPRCs)
- We now understand their limitations

Compare-and-swap has infinite consensus number

- Initialize RMW atomic register r with a value FIRST that cannot be a thread id

```
public T decide (T value) {
  int i := ThreadID.get();
  propose(value);
  if (r.CaS(FIRST, i))
    return proposed[i];
  else
    return proposed[r.get()];
}
```

A step back

- We have a hierarchy of objects with different consensus number
  - 1: Read/Write registers
  - 2: Stacks, Queues, etc.
  - ....
  - ∞: CAS, Load-link/store-conditional
- But why again do we care about consensus?

Universality of Consensus

- A class C is universal for n threads if one can construct a wait-free linearizable implementation of any object with some objects of class C and some read/write registers

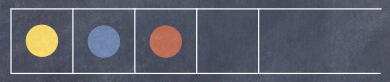
Theorem An object with consensus number $\mathcal{C}$ is universal for $\mathcal{C}$ threads

Coming up...

- An impractical universal construction
  - aims for correctness, not efficiency
  - applies only to deterministic objects
  - can be optimized
- Step 1: A lock-free universal construction
  - some method finishes infinitely often in a finite number of steps
- Step 2: A wait-free universal construction
  - every method finishes infinitely often in a finite number of steps

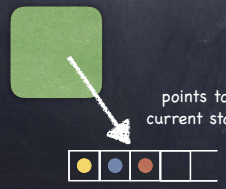
Lock-free Universal Construction: Try I

- A queue



Threads concurrently trying to enqueue, dequeue

Consensus object



points to current state

wants to dequeue

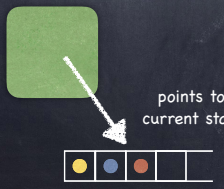
wants to enqueue

Lock-free Universal Construction: Try I

- A queue

Threads concurrently trying to enqueue, dequeue

Consensus object



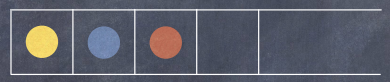
points to current state

wants to dequeue

wants to enqueue

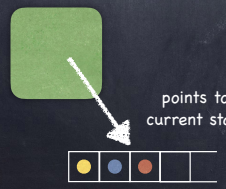
Lock-free Universal Construction: Try I

- A queue



Threads concurrently trying to enqueue, dequeue

Consensus object



points to current state

wants to dequeue

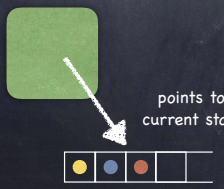
wants to enqueue

Lock-free Universal Construction: Try I

- A queue

Threads concurrently trying to enqueue, dequeue

Consensus object



points to current state

Propose state to consensus object

Pointer switches to winner

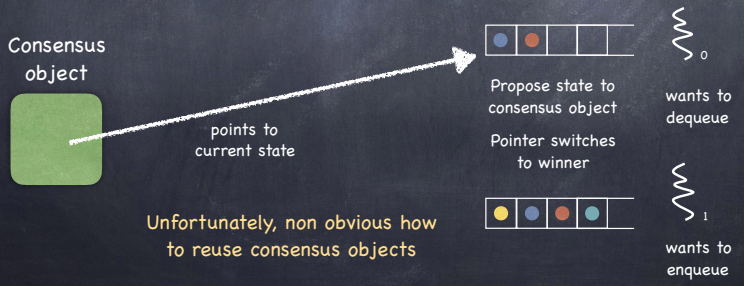
wants to dequeue

wants to enqueue

Lock-free Universal Construction: Try I

- A queue

Threads concurrently trying to enqueue, dequeue

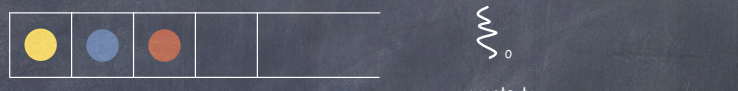


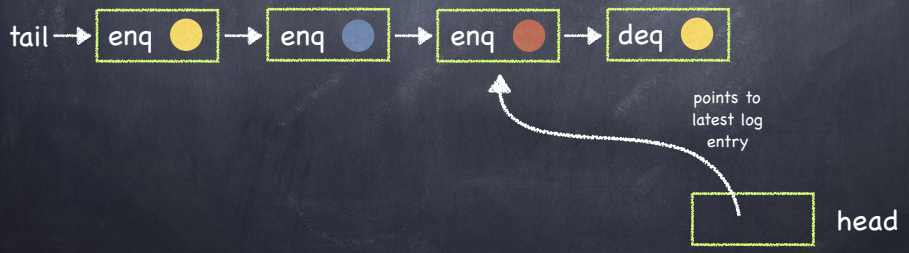
Unfortunately, non obvious how to reuse consensus objects

Lock-free Universal Construction: Try II

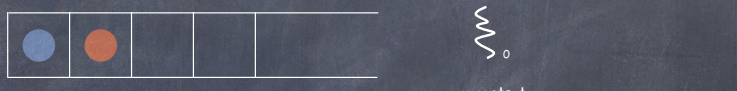
- Represent object as
  - initial state +
  - a log (linked list) of method calls
- New method call
  - finds end of list
  - atomically appends new method to log
  - determines current state by locally re-executing log

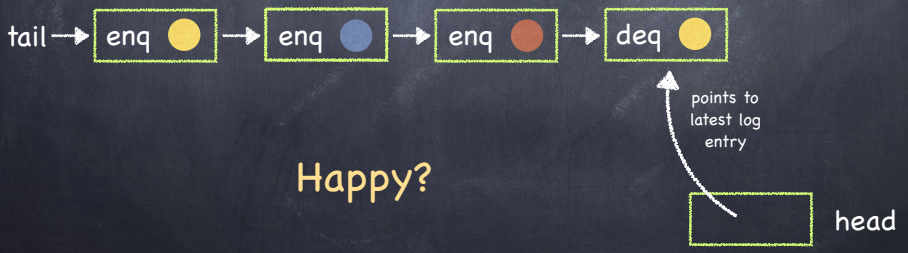
Lock-free Universal Construction: Try II





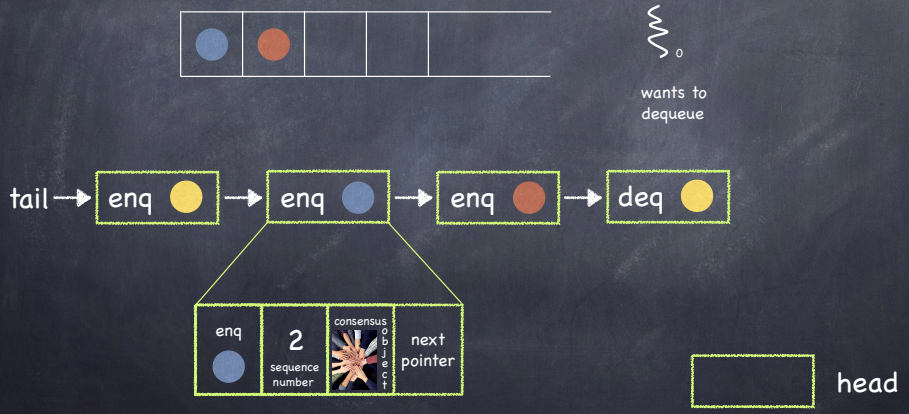
Lock-free Universal Construction: Try II



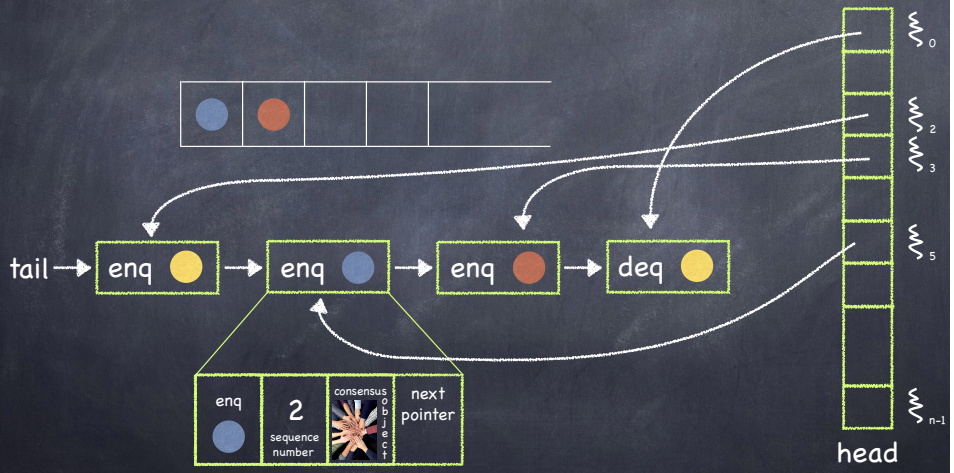


Happy?

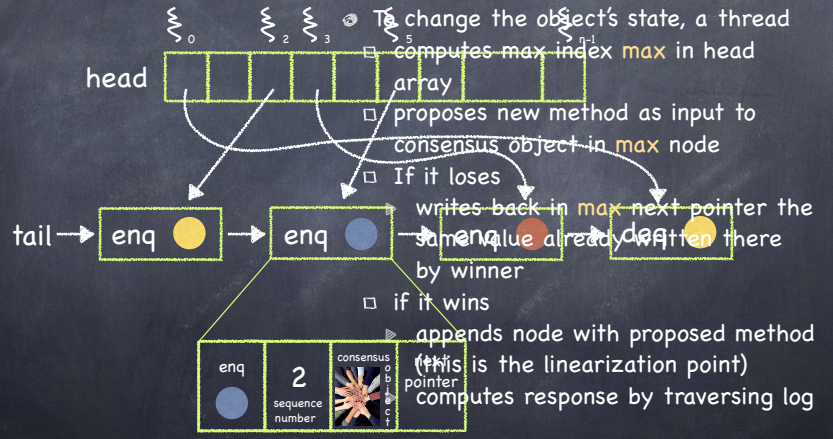
Lock-free Universal Construction: Try II



Lock-free Universal Construction: Try II



Lock-free Universal Construction: Try II



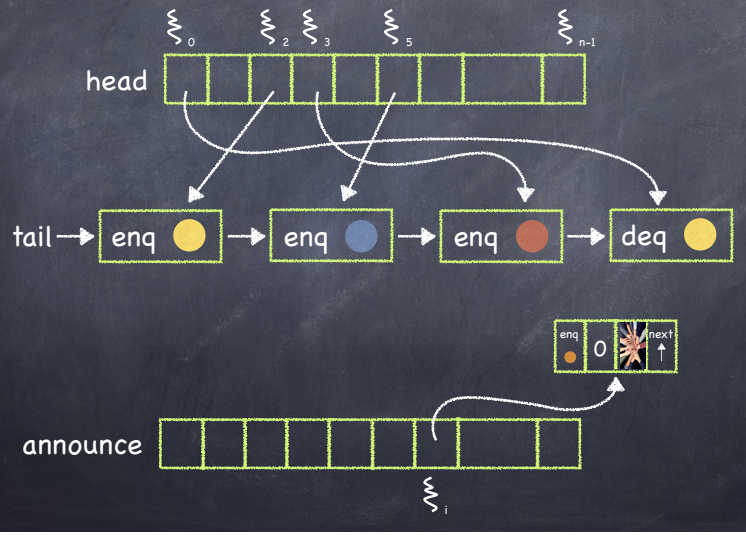
- To change the object's state, a thread
  - computes max index max in head array
  - proposes new method as input to consensus object in max node
  - If it loses
    - writes back in max next pointer the same value already written there by winner
  - if it wins
    - appends node with proposed method (this is the linearization point)
    - computes response by traversing log

A node can lose consensus only if another succeeds: lock free

Wait-free Universal Construction

- A little help from your friends
  - threads announce their intentions
  - before trying to enqueue its own method, thread tries to help enqueueing someone else's announced method
  - mechanism ensures every method will be enqueued after a finite number of steps

The ingredients



The Universal class

```
public class Universal {
  private Node[] announce;
  private Node[] head;
  private Node tail = new node();
  tail.seq := 1;
  for (int j:=0; j < n; j++) {
    head[j] := tail; announce[j] := tail
  };
```

I need somebody's help...

```
public Response apply(Invoc invoc) {
  int i := ThreadID.get();
  announce[i] := new Node(invoc);
  head[i] := Node.max(head);
  while (announce[i].seq = 0) {
  ...
  // while node not appended to list
  ...
  }
... // create response by traversing log
}
```

announce new method call, asking help from others

I need somebody's help...

```
public Response apply(Invoc invoc) {
  int i := ThreadID.get();
  announce[i] := new Node(invoc);
  head[i] := Node.max(head);
  while (announce[i].seq = 0) {
  ...
  // while node not appended to list
  ...
  }
... // create response by traversing log
}
```

tentatively compute end of log

I need somebody's help...

```
public Response apply(Invoc invoc) {
  int i := ThreadID.get();
  announce[i] := new Node(invoc);
  head[i] := Node.max(head);
  while (announce[i].seq = 0) {
  ...
  // while node not appended to list
  ...
  }
... // create response by traversing log
}
```

Execute loop until Node has not been added to log (by thread i or some other thread)

I need somebody's help...

```
while (announce[i].seq = 0) {
...
// while node not appended to list
...
}
```

I need somebody's help...

```
while (announce[i].seq = 0) {
  Node before := head[i];
  Node help := announce[(before.seq + 1 % n)];
  if (help.seq = 0)
    prefer := help;
  else
    prefer := announce[i];
  ...
}
```

tentative end of log

I need somebody's help...

Candidate thread for help

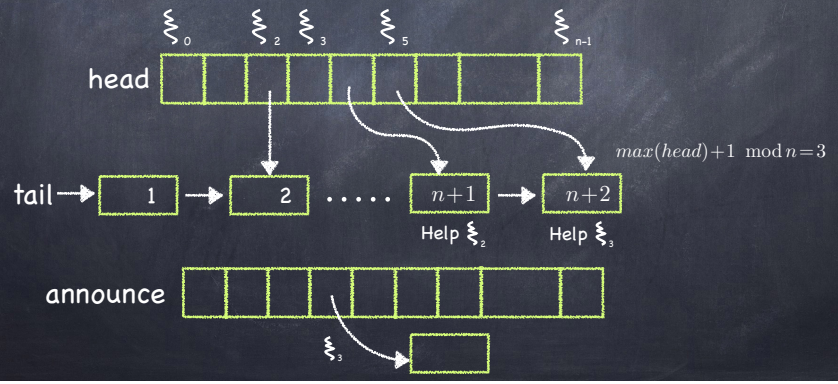
```
while (announce[i].seq = 0) {
  Node before := head[i];
  Node help := announce[(before.seq + 1 % n)];
  if (help.seq = 0)
    prefer := help;
  else
    prefer := announce[i];
  ...
}
```

Managing altruism

- Chosen thread a function of log's length
  - effectively, each log position is preferentially assigned to a thread
    - period equal to number of threads
  - if thread i has announced a method and failed to append it by the time "its" log position comes around , then every thread tries to append it on its behalf
  - specifically...
    - If log's last node has sequence number k, every thread that wants to enqueue a node checks whether thread $k+1 \mod n$ needs help

Progress guarantee

After thread i announces $node_i$, no more than n other nodes can be added to the log before $node_i$ is appended to the log



I need somebody's help...

Pick new thread in each iteration

```
while (announce[i].seq = 0) {
  Node before := head[i];
  Node help := announce[(before.seq + 1 % n)];
  if (help.seq = 0)
    prefer := help;
  else
    prefer := announce[i];
  ...
}
```

I need somebody's help...

If they need help, lend a hand

```
while (announce[i].seq = 0) {
  Node before := head[i];
  Node help := announce[(before.seq + 1 % n)];
  if (help.seq = 0)
    prefer := help;
  else
    prefer := announce[i];
  ...
}
```

I need somebody's help...

```
while (announce[i].seq = 0) {
  Node before := head[i];
  Node help := announce[(before.seq + 1 % n)];
  if (help.seq = 0)
    prefer := help;
  else
    prefer := announce[i];
  ...
}
```

otherwise, the chance is MINE!

I need somebody's help...

```
while (announce[i].seq = 0) {
  Node before := head[i];
  Node help := announce[(before.seq + 1 % n)];
  if (help.seq = 0)
    prefer := help;
  else
    prefer := announce[i];
  Node after := before.decNext.decide(prefer);
  before.next := after;
  after.seq := before.seq + 1;
  head[i] := after;
}
```

decide next node to append

I need somebody's help...

```
while (announce[i].seq = 0) {
  Node before := head[i];
  Node help := announce[(before.seq + 1 % n)];
  if (help.seq = 0)
    prefer := help;
  else
    prefer := announce[i];
  Node after := before.decNext.decide(prefer);
  before.next := after;
  after.seq := before.seq + 1;
  head[i] := after;
}
```

add appropriate pointer to before

I need somebody's help...

```
while (announce[i].seq = 0) {
  Node before := head[i];
  Node help := announce[(before.seq + 1 % n)];
  if (help.seq = 0)
    prefer := help;
  else
    prefer := announce[i];
  Node after := before.decNext.decide(prefer);
  before.next := after;
  after.seq := before.seq + 1;
  head[i] := after;
}
```

make visible that a node has been appended

Computing the response

```
SeqObject MyObject := new SeqObject();
current := tail.next;
while (current != announce[i]){
      MyObject.apply(current.invoc);
      current := current.next;
  }
head[i] :=announce[i];
return MyObject.apply(current.invoc);
}
```

Builds local copy by traversing the log and computes response

All together now

```
public Response apply(Invoc invoc) {
  int i := ThreadID.get();
  announce[i] := new Node(invoc);
  head[i] := Node.max(head);
  while (announce[i].seq = 0) {
     Node before := head[i];
     Node help := announce[(before.seq + 1 % n)];
     if (help.seq = 0)
        prefer := help;
     else
        prefer := announce[i];
     Node after := before.decNext.decide(prefer);
     before.next := after;
     after.seq := before.seq + 1;
     head[i] := after;
  }
  SeqObject MyObject := new SeqObject();
  current := tail.next;
  while (current != announce[i]){
        MyObject.apply(current.invoc);
        current := current.next;
    }
  head[i] :=announce[i];
  return MyObject.apply(current.invoc);
  }
}
```

Back to mutual exclusion

(but with an eye to performance)

Spin locks: if at first you don't succeed, try try again

- Threads contend for the lock
  - one succeeds, entering CS
  - upon finishing, releases lock
- Two effects on performance
  - lock serialize accesses
    - loss of parallelism
  - contention resolution
    - what implications?

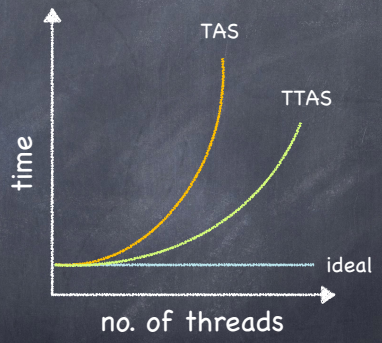
Basic spin locks

- Use R/W registers
  - Peterson's algorithm
  - Bakery algorithm
- Large footprint
  - Ω(n) R/W registers
- Sometimes do not guarantee mutual exclusion
  - modern processors do not provide sequentially consistent memory

Test-and set locks

```
public class TASLock implements Lock {
  AtomicBoolean state := new AtomicBoolean(false);
  public void lock() {
    while (state.getAndSet(true)) {};
  }
  public void unlock() {
     state.set(false);
  }
}
```

```
public class TTASLock implements Lock {
  AtomicBoolean state := new AtomicBoolean(false);
  public void lock() {
    while (true) {
      while (state.get()) {};
      if (!state.getAndSet(true))
        return;
    }
  }
  public void unlock() {
     state.set(false);
  }
}
```



Why are TAS and TTAS performing poorly?

Why is TTAS performing much better than TAS?

Cache effects

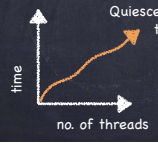
- Simple TAS
  - all TAS go to bus
    - everybody waits
  - each TAS invalidates cached copies of lock
    - every thread must use bus to retrieve new lock value even if unchanged
    - lot of bus traffic!
  - lock release may be delayed
    - by spinner hogging the bus

Cache effects

- TTAS
  - Suppose T_2 reads a lock held by T_1
    - cache miss the first time
    - but, as long as T_1 has the lock, all reads are from the cache
  - When T_1 releases lock
    - all cached copies of lock are invalidated
    - all spinning thread reread lock value
    - call getAndSet at about same time
    - burst of bus traffic, before quiescence

Measuring quiescence

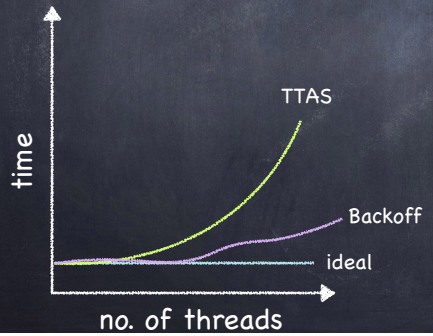
- Acquire lock. While in CS
  - pause for Δ without using bus
  - use bus heavily
- Tune Δ
  - if Δ > quiescence time, then time in CS is independent of the number of threads
  - otherwise, time in CS grows with thread number



explains TTAS line

Solutions

- Exponential backoff
  - introduces random delay if contention
  - delays double every time contention detected



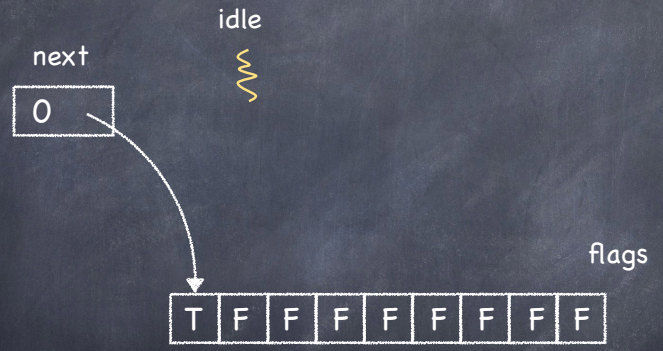
BUT...

- must choose parameters carefully
- not portable across platforms

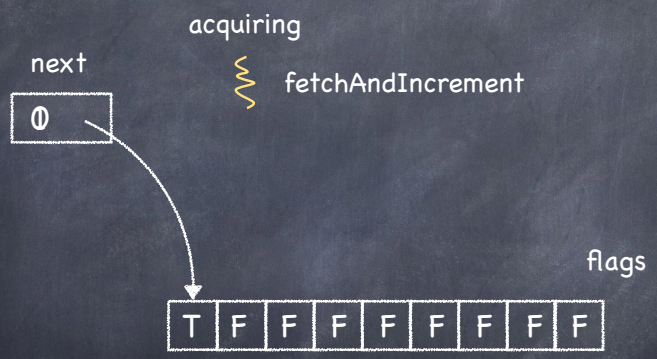
Solutions

- Keep a queue!
  - avoids useless invalidations
  - on release, thread notifies next in line
    - no one else needs to know...
  - FIFO: same high level of fairness as Bakery's algorithm

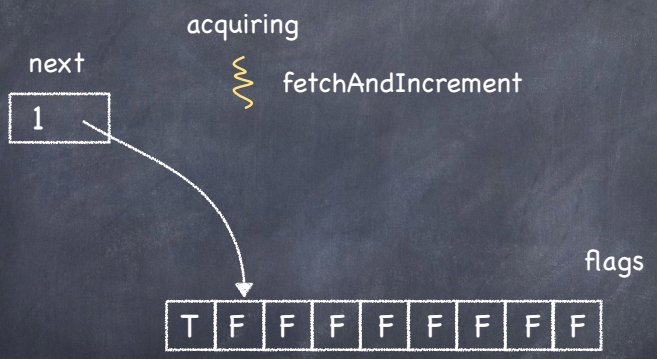
Anderson Queue Lock



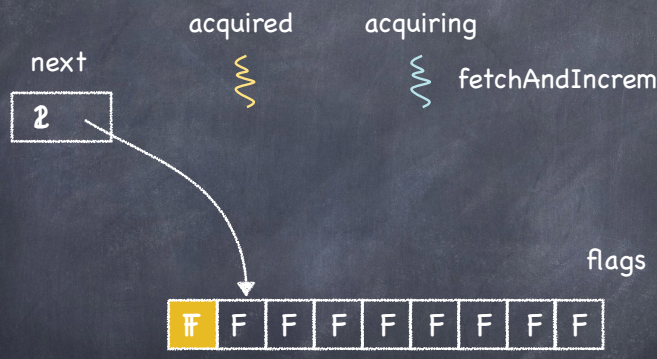
Anderson Queue Lock



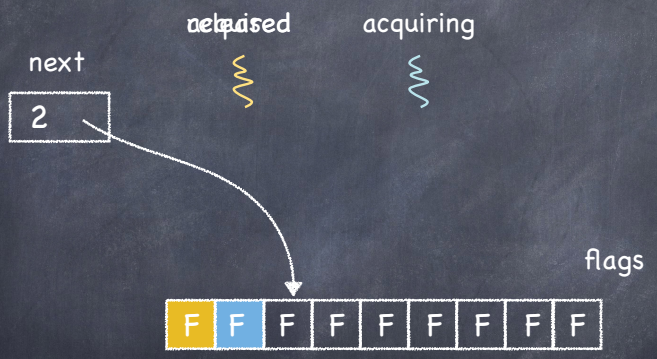
Anderson Queue Lock



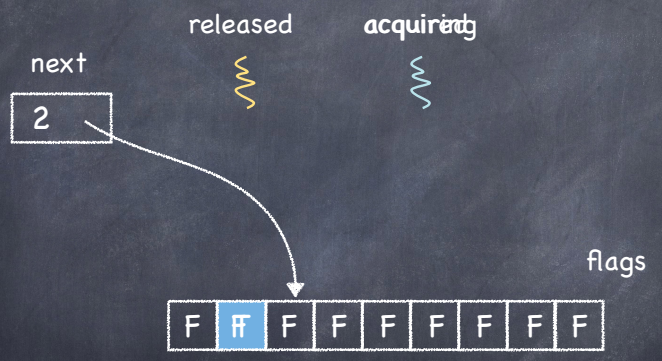
Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock

- FIFO
- Simple
- Reduces invalidations
  - but what if a line holds multiple thread bits?
    - false sharing
    - reduced by padding (each bits in separate cache line)
    - but space hog (especially if few contenders)

MCS Queue Lock

- Lock's state represented as a queue of QNodes
- Each thread can choose location of queue element it spins on (good for cache-less NUMA)
- release invalidates only successor's cache entry

```
class QNode {
 volatile boolean locked := false;
 volatile qnode   next   := null;
}
```

MCS Queue Lock

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

MCS Queue 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;
```