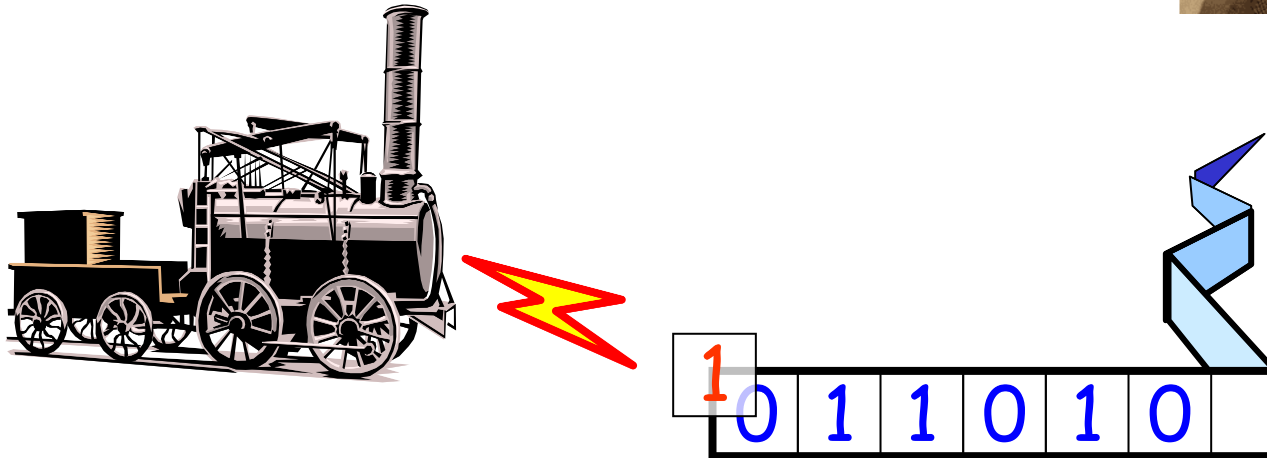
Universality of Consensus

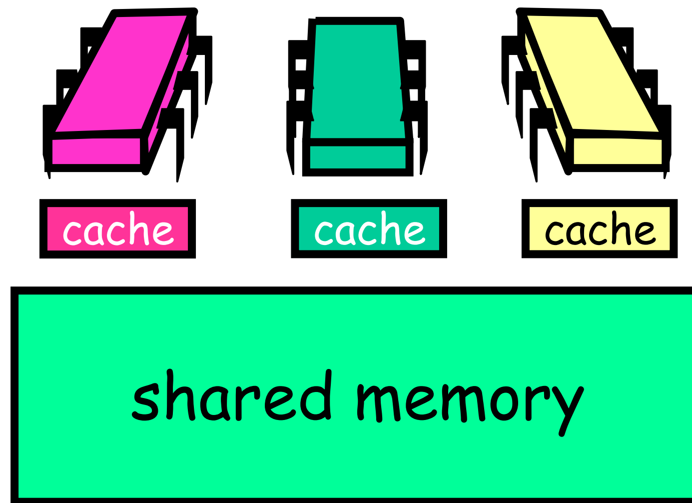
Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Turing Computability



- A mathematical model of computation
- Computable = Computable on a T-Machine

Shared-Memory Computability



- Model of asynchronous concurrent computation
- Computable = Wait-free/Lock-free computable on a multiprocessor

The

Can we implement them from any other object that has consensus number ∞?

Like compareAndSet()...

1	Read/W
2	FIFO Queue, LIFO Stack
. . .	
∞	Multiple Assignment

Theorem: Universality

- Consensus is universal
- From n-thread consensus build a
  - Wait-free
  - Linearizable
  - n-threaded implementation
  - Of any sequentially specified object

BROWN

Proof Outline

- A universal construction
  - From n-consensus objects
  - And atomic registers
- Any wait-free linearizable object
  - Not a practical construction
  - But we know where to start looking …

Like a Turing Machine

- This construction
  - Illustrates what needs to be done
  - Optimization fodder
- Correctness, not efficiency
  - Why does it work? (Asks the scientist)
  - How does it work? (Asks the engineer)
  - Would you like fries with that? (Asks the liberal arts major)

A Generic Sequential Object

```
public interface SeqObject {
  public abstract Response
apply(Invocation invoc);
}
```

BROWN

A Generic Sequential Object

```
public interface SeqObject {
  public abstract Response
apply(Invocation invoc);
}
```

Push:5, Pop:null

Invocation

```
public class Invoc {
 public String method;
 public Object[] args;
}
```

Invocation

```
public class Invoc {
 public String method;
 public Object[] args;
}
```

Method name

Invocation

```
public class Invoc {
 public String method;
 public Object[] args;
}
```

Arguments

A Generic Sequential Object

```
public interface SeqObject {
  public abstract Response
apply(Invocation invoc);
}
```

Response

OK, 4

Response

```
public class Response {
public Object value;
}
```

Object value;

Return value

A Universal Concurrent Object

```
public interface SeqObject {
  public abstract Response
apply(Invocation invoc);
}
```

A concurrent object that is linearizable to the generic sequential object

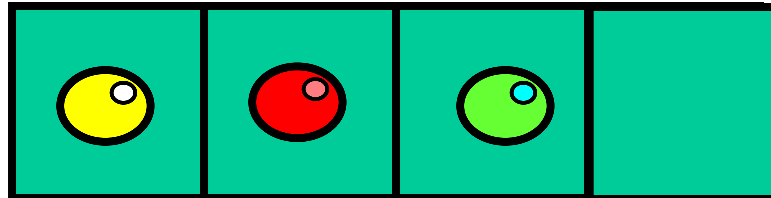
Start with Lock-Free Universal Construction

- First Lock-free: infinitely often some method call finishes.
- Then Wait-Free: each method call takes a finite number of steps to finish

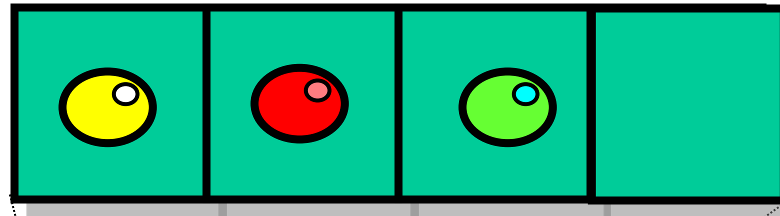
Universal Construction: Naïve Idea

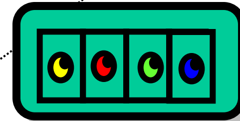
- Use consensus object to store pointer to cell with current state
- Each thread creates new cell
  - computes outcome,
  - and tries to switch pointer to its outcome
- Unfortunately not…
  - consensus objects can be used once only

Naïve Idea



Naïve Idea





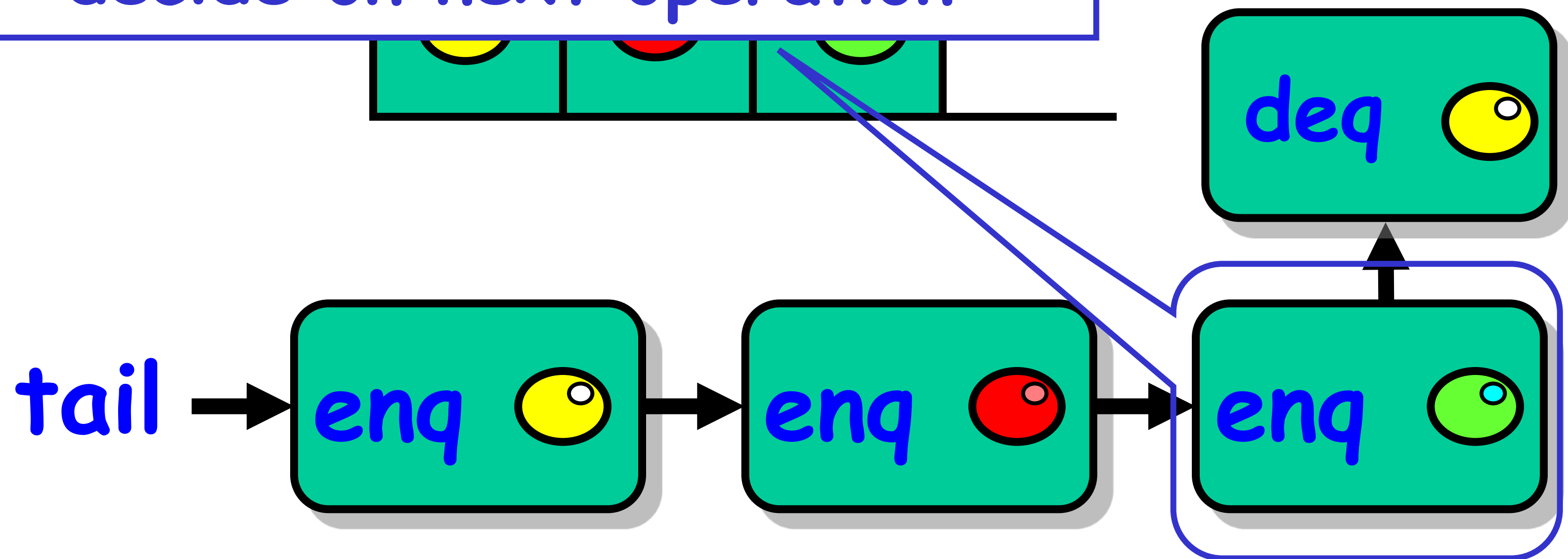
Why only once? Why is consensus object not readable?

Queue based consensus

```
public decide(Object value) {
   propose(value);
   Ball ball = this.queue.deq();
   if (ball == Ball.RED)
    return proposed[i];
   else
    return proposed[1-i];
}
```

Solved one time 2-consensus. Not clear how to allow reuse of object or reading its state…

Improved Idea: Linked-List Representation



Universal Construction

- Object represented as
  - Initial Object State
  - A Log: a linked list of the method calls
- New method call
  - Find end of list
  - Atomically append call
  - Compute response by traversing the log upto the call

Basic Idea

- Use one-time consensus object to decide next pointer
- All threads update actual next pointer based on decision
  - OK because they all write the same value
- Challenges
  - Lock-free means we need to worry what happens if a thread stops in the middle

Basic Data Structures

```
public class Node implements
java.lang.Comparable {
 public Invoc invoc;
 public Consensus<Node> decideNext;
 public Node next;
 public int seq;
 public Node(Invoc invoc) {
    invoc = invoc;
    decideNext = new Consensus<Node>()
    seq = 0;
  }
```

Basic Data Structures

```
public class Node implements
java.lang.Comparable {
 public Invoc invoc;
 public Consensus<Node> decideNext;
 public Node next;
 public int seq;
 p
    decideNext = new Consensus<Node>()
    seq = 0;
  }
```

Standard interface for class whose objects are totally ordered

Basic Data Structures

```
public class Node implements
java.lang.Comparable {
 public Invoc invoc;
 public Consensus<Node> decideNext;
 public Node next;
 public int seq;
 p
    invoc = invoc;
    decideNext = new Consensus<Node>()
    seq = 0;
  }
```

the invocation

Basic Data Structures

```
public class Node implements
java.lang.Comparable {
 public Invoc invoc;
 public Consensus<Node> decideNext;
 public Node next;
 public int seq;
 public Node(Invoc invoc) {
   invoc = invoc;
  }
```

Decide on next node (next method applied to object)

Basic Data Structures

```
public class Node implements
java.lang.Comparable {
 public Invoc invoc;
 public Consensus<Node> decideNext;
 public Node next;
 public int seq;
 public Node(Invoc invoc) {
   invoc = invoc;
```

Traversable pointer to next node (needed because you cannot repeatedly read a consensus object)

Basic Data Structures

```
public class Node implements
java.lang.Comparable {
 public Invoc invoc;
 public Consensus<Node> decideNext;
 public Node next;
 public int seq;
 public Node(Invoc invoc) {
   invoc = invoc;

   seq = 0;
 }
```

Seq number

Basic Data Structures

Create a new node for a given method invocation

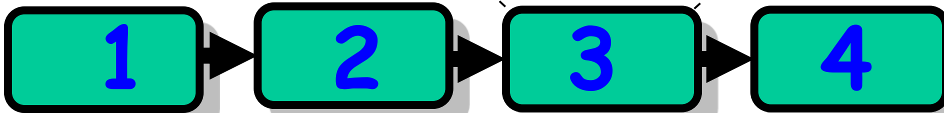
```
pu
ja
 public Invoc invoc;
 public Consensus<Node> decideNext;
 public Node next;
 public int seq;
 public Node(Invoc invoc) {
    invoc = invoc;
    decideNext = new Consensus<Node>()
    seq = 0;
  }
```

Universal Object

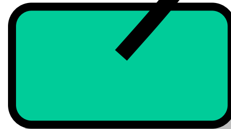
Seq number, Invoc

next

node

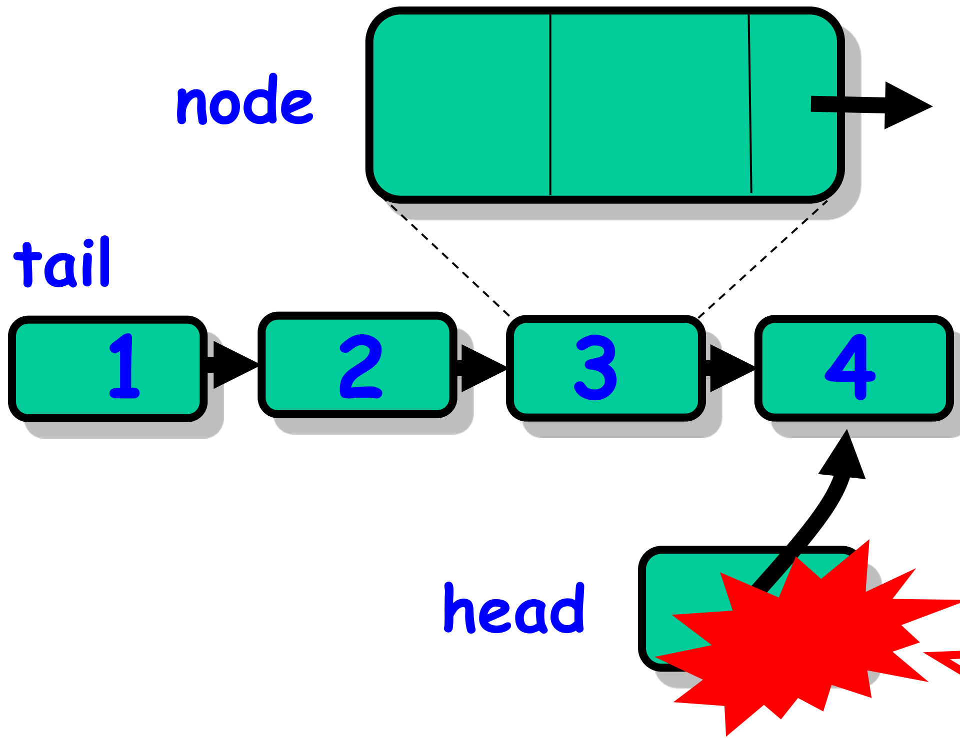


decideNext (Consensus Object)



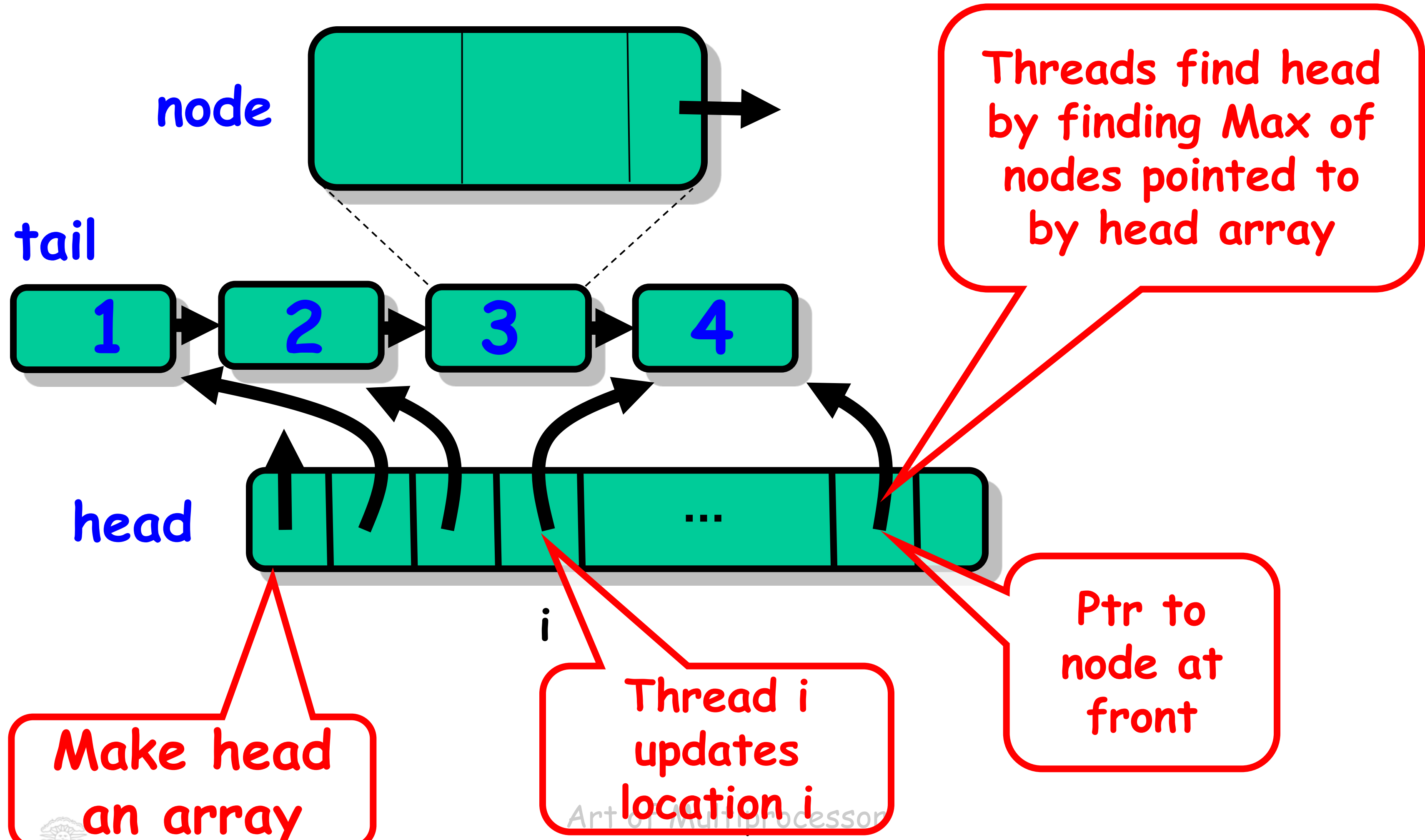
Ptr to cell w/highest Seq Num

Universal Object



All threads repeatedly modify head…back to where we strated?

The Solution



Universal Object

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

Universal Object

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

Head Pointers Array

Universal Object

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

Tail is a sentinel node with sequence number 1

Universal Object

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

Tail is a sentinel node with sequence number 1

Universal Object

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

Initially head points to tail

Find Max Head Value

```
public static Node max(Node[] array) {
    Node max = array[0];
    for (int i = 1; i < array.length; i++)
      if (max.seq < array[i].seq)
        max = array[i];
      return max;
  }
```

Find Max Head Value

```
public static Node max(Node[] array) {
    Node max = array[0];
    for (int i = 0; i < array.length; i++)
      if (max.seq < array[i].seq)
        max = array[i];
      return max;
  }
```

Traverse the array

Find Max Head Value

```
public static Node max(Node[] array) {
    Node max = array[0];
    for (int i = 0; i < array.length; i++)
      if (max.seq < array[i].seq)
        max = array[i];
      return max;
  }
```

Compare the seq nums of nodes pointed to by the array

Find Max Head Value

```
public static Node max(Node[] array) {
    Node max = array[0];
    for (int i = 0; i < array.length; i++)
      if (max.seq < array[i].seq)
        max = array[i];
      return max;
  }
```

Compare the seq nums of nodes pointed to by the array

Universal Application Part I

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
    }
  …
```

BROWN

Universal Application Part I

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
  }
  …
```

Apply will have invocation as input and return the appropriate response

Universal Application Part I

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
  }
  …
```

My id

Universal Application Part I

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
    }
  …
```

My method call

Universal Application Part I

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
  }
 …
```

As long as I have not been threaded into list

Universal Application Part I

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
    }
  …
```

Node at head of list that will try and to append to

Universal Application Part I

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
  }
}
```

Decide winning node; could have already been decided

Universal Application

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
  }
}
```

Could have already been set by winner…in which case no affect

Set next pointer based on decision

Universal Application Part I

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
    }
  …
```

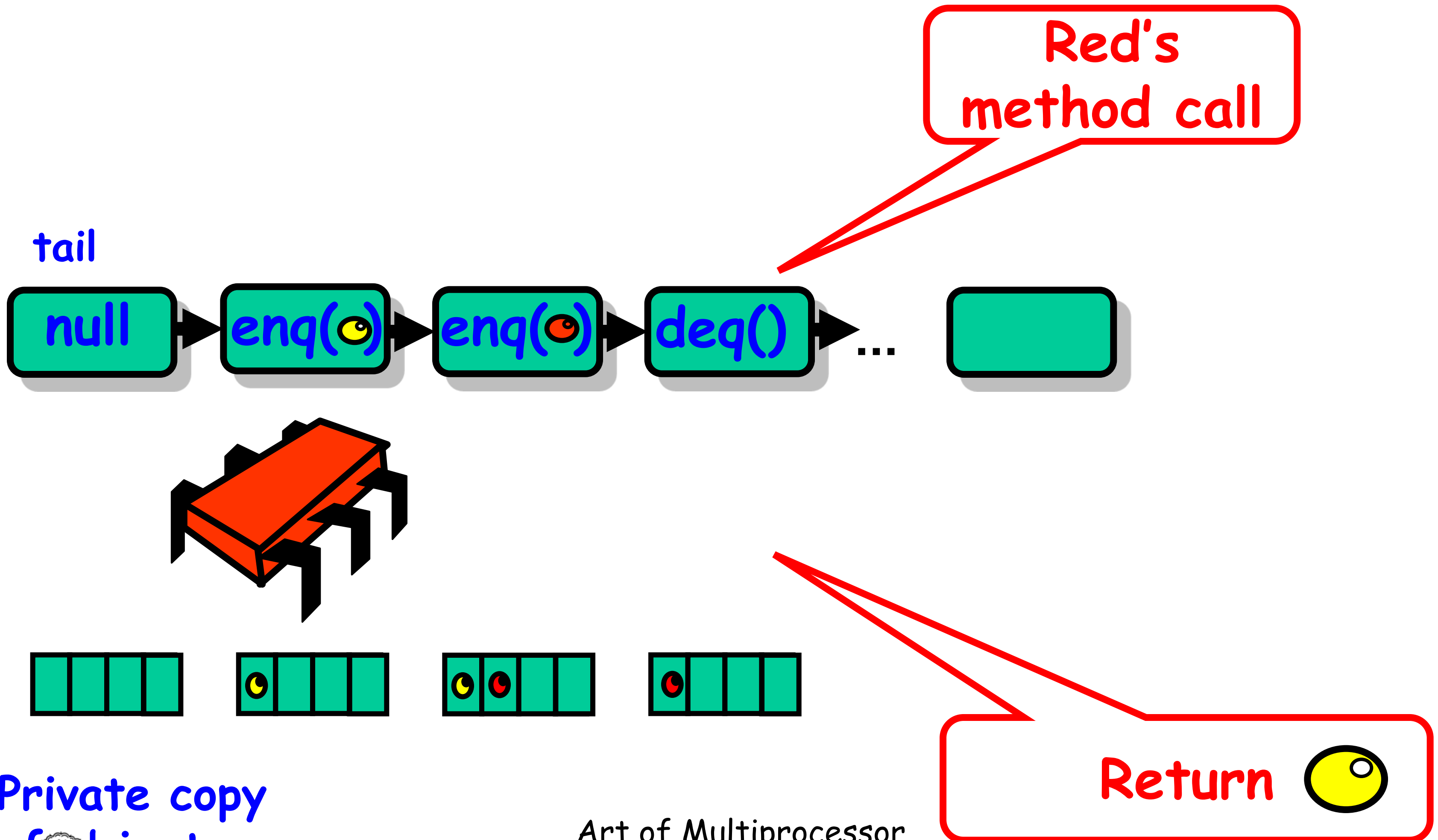
Set seq number indicating node was appended

Universal Application Part I

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
    }
  …
```

add to head array so new head will be found

Part II – Compute Response



Universal Application Part II

```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
   MyObject.apply(current.invoc);
   current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

Universal Application Part II

```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
   MyObject.apply(current.invoc);
   current = current.next;
  }
...                        .invoc);
```

Compute the result by sequentially applying the method calls in the list to a private copy of the object starting from the initial state

Universal Application Part II

```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
   MyObject.apply(current.invoc);
   current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

Start with initialized copy of the sequential object

Universal Application Part II

```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
   MyObject.apply(current.invoc);
   current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

First new method call is appended after the tail

Universal Application Part II

```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
   MyObject.apply(current.invoc);
   current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

While not reached my own method call

Universal Application Part II

```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
  MyObject.apply(current.invoc);
  current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

Apply the current nodes method to object

Universal Application Part II

```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
   MyObject.apply(current.invoc);
   current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

Return the result after applying my own method call

Correctness

- List defines linearized sequential history
- Thread returns its response based on list order

Lock-freedom

- Lock-free because
- New winner node is added into the head array within a finite number of steps
- A thread moves forward in list
- Can repeatedly fail to win consensus on "real" head only if another succeeds

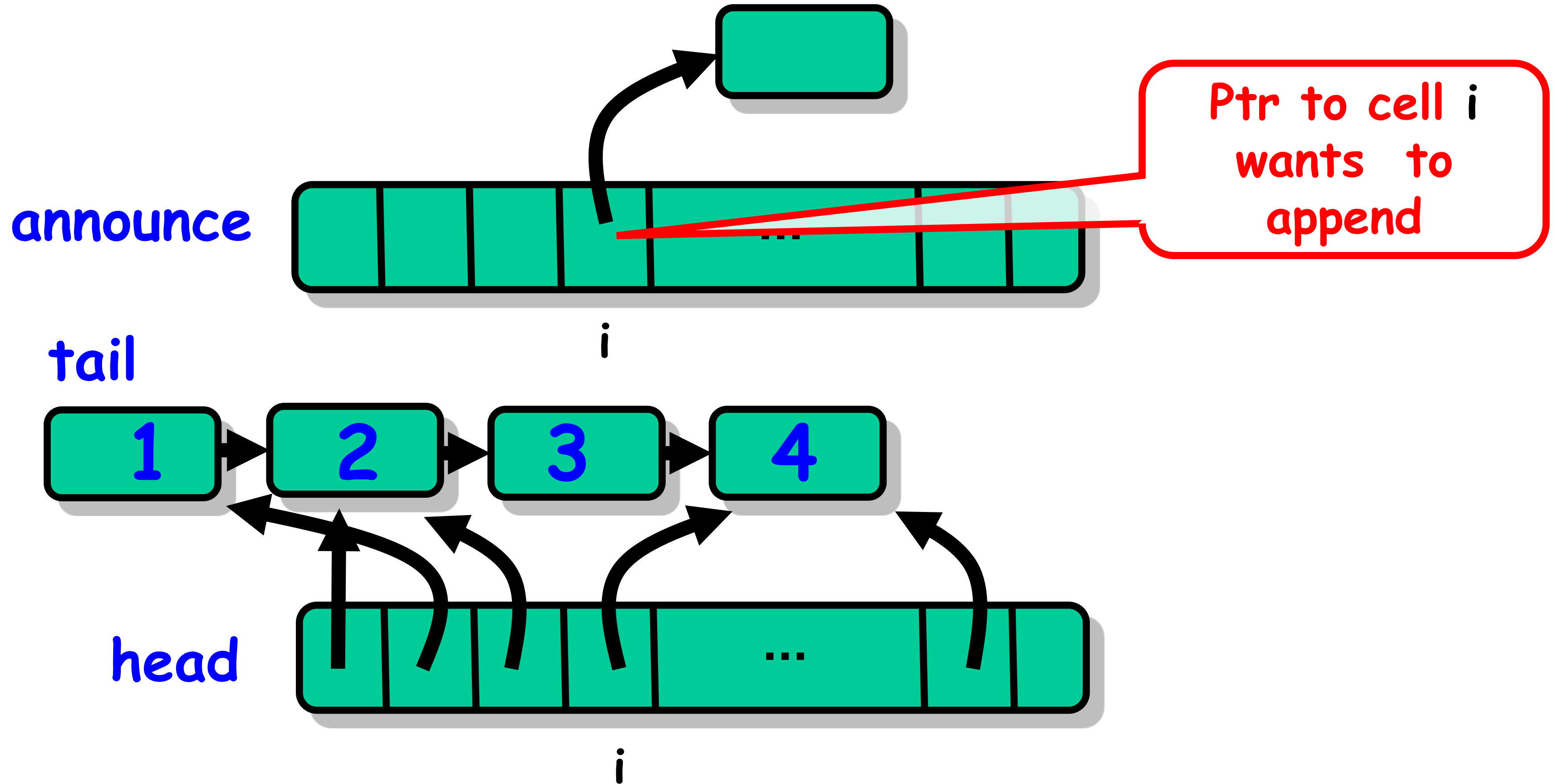
Wait-free Construction

- Lock-free construction + **announce** array
- Stores (pointer to) node in **announce**
  - If a thread doesn't append its node
  - Another thread will see it in array and ***help*** append it

Helping

- "Announcing" my intention
  - Guarantees progress
  - Even if the scheduler hates me
  - My method call will complete
- Makes protocol wait-free
- Otherwise starvation possible

Wait-free Construction



The Announce Array

```
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
  };
```

BROWN

The Announce Array

```
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
  };
```

Announce array

The Announce Array

```
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
  };
```

All entries initially point to tail

BROWN

A Cry For 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
  …
  }
```

BROWN

A Cry For 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
  …
  }
```

Announce new method call (node), asking help from others

A Cry For 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
  …
  }
```

Look for end of list

A Cry For 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
  …
  }
```

Main loop, while node not appended (either by me or some thread helping me)

Main Loop

- Non-zero sequence number indicates success
- Thread keeps helping append nodes
- Until its own node is appended

Main Loop

```
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];
…
```

Main Loop

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

Keep trying until my cell gets a sequence number

Main Loop

```
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];
```

Possible end of list

Main Loop

```
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];
```

Who do I help?

Altruism

- Choose a thread to "help"
- If that thread needs help
  - Try to append its node
  - Otherwise append your own
- Worst case
  - Everyone tries to help same pitiful loser
  - Someone succeeds

Help!

- When last node in list has sequence number k
- All threads check …
  - Whether thread k+1 mod n wants help
  - If so, try to append her node first

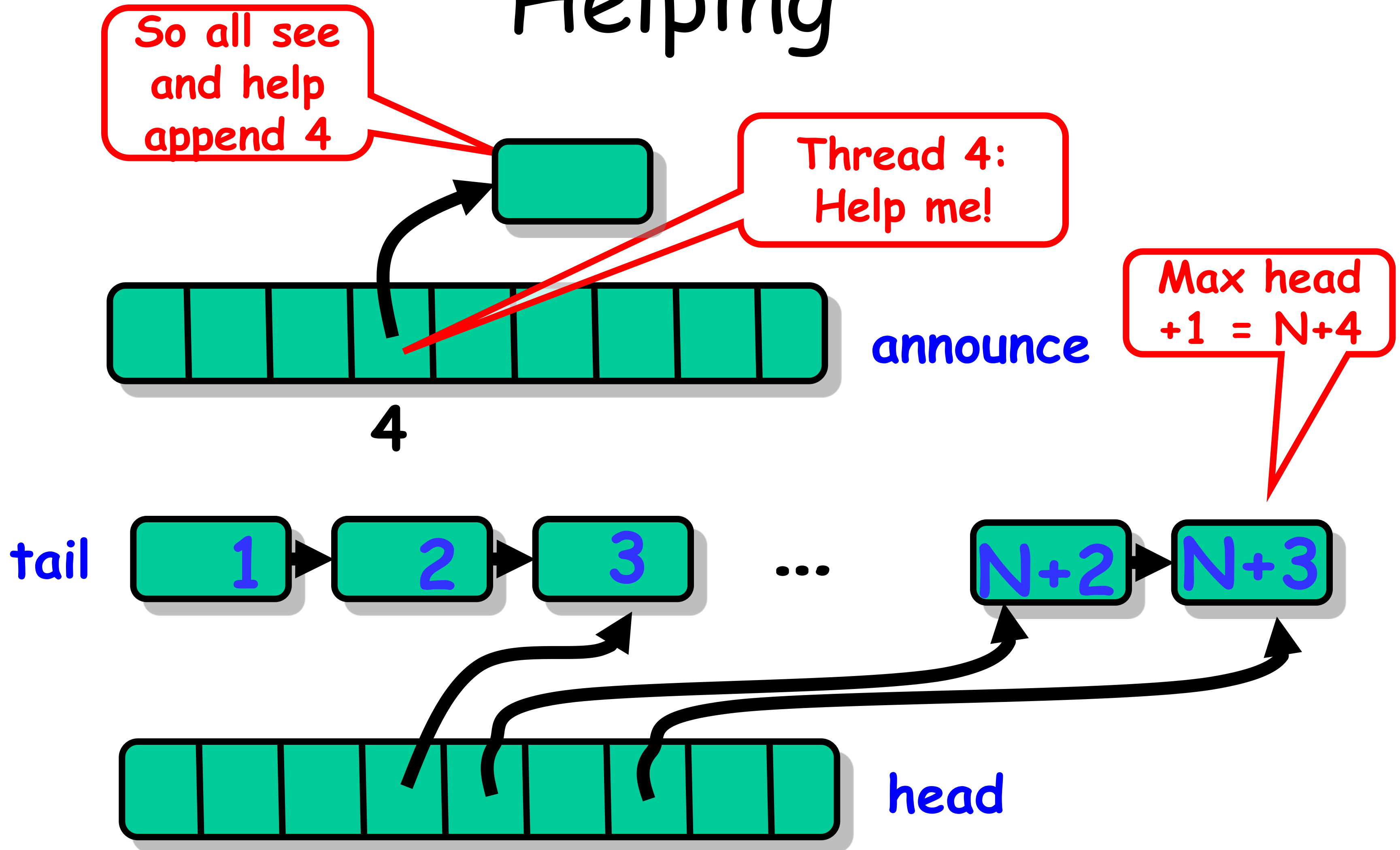
Help!

- First time after thread k+1 announces
  - No guarantees
- After n more nodes appended
  - Everyone sees that thread k+1 wants help
  - Everyone tries to append that node
  - Someone succeeds

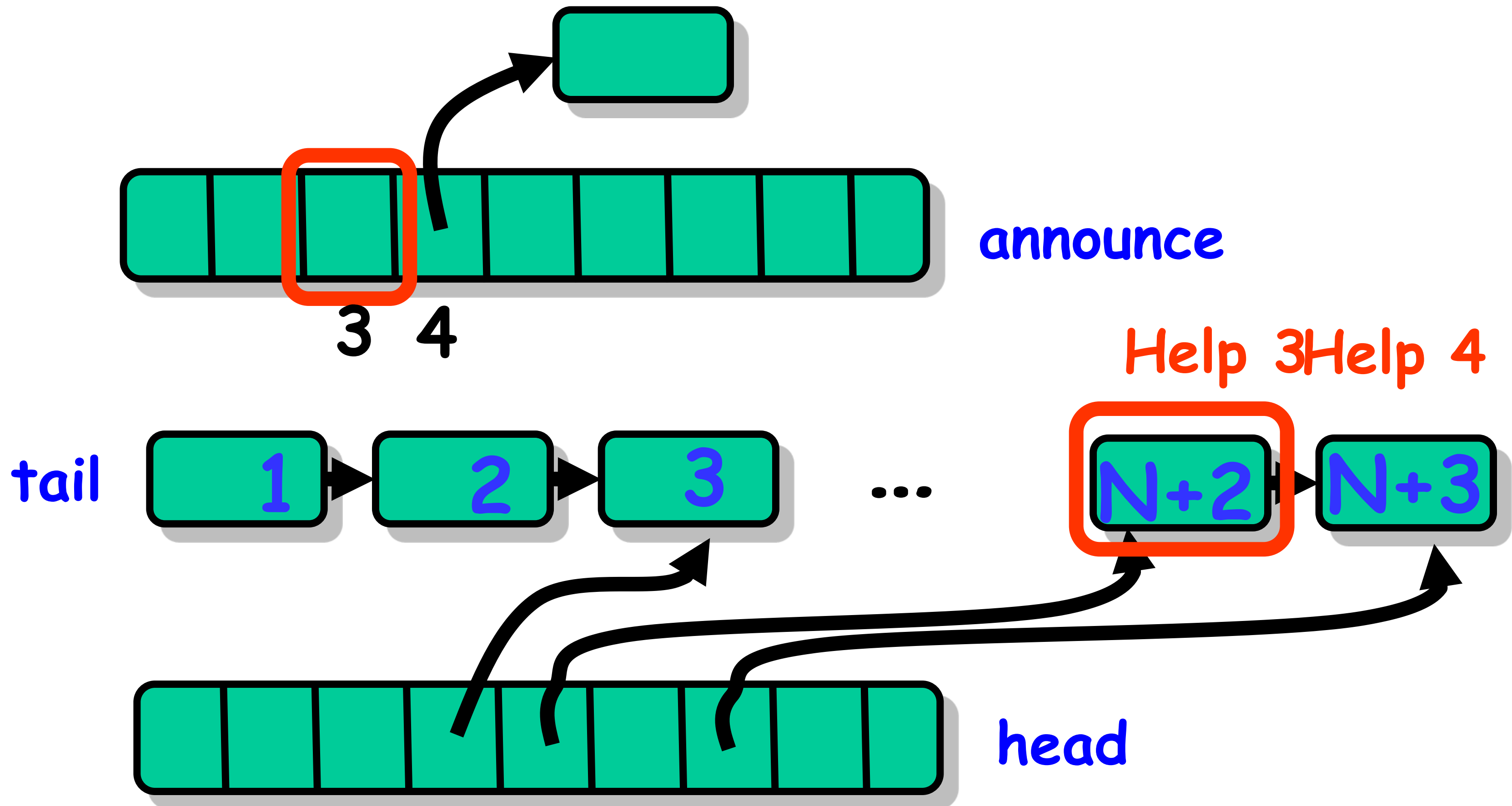
Sliding Window Lemma

- After thread *A* announces its node
- No more than n other calls
  - Can start and finish
  - Without appending *A*'s node

Helping



The Sliding Help Window



Sliding Help Window

```
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];
```

In each main loop iteration pick another thread to help

Sliding Help Window

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

Help if help required, but otherwise it's all about me!

Rest is Same as Lock-free

```
while (prefer.seq == 0) {
 …
 Node after =
  before.decideNext.decide(prefer);
 before.next = after;
 after.seq = before.seq + 1;
 head[i] = after;
}
```

Rest is Same as Lock-free

```
while (prefer.seq == 0) {
 …
 Node after =
  before.decideNext.decide(prefer);
 before.next = after;
 after.seq = before.seq + 1;
 head[i] = after;
}
```

Decide next node to be appended

Rest is Same as Lock-free

```
while (prefer.seq == 0) {
  Node after =
   before.decideNext.decide(prefer);
  before.next = after;
  after.seq = before.seq + 1;
  head[i] = after;
}
```

Update next based on decision

Rest is Same as Lock-free

```
while (prefer.seq == 0) {
  ...
  Node after =
   before.decideNext.decide(prefer);
  before.next = after;
  after.seq = before.seq + 1;
  head[i] = after;
}
```

Tell world that node is appended

Finishing the Job

- Once thread's node is linked
- The rest is again the same as in lock-free alg
- Compute the result by sequentially applying the method calls in the list to a private copy of the object starting from the initial state

Then Same Part II

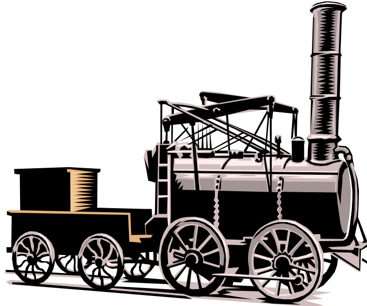
```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
   MyObject.apply(current.invoc);
   current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

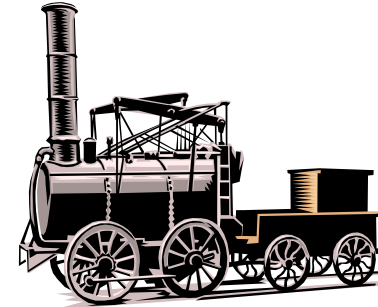
Universal Application Part II

```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
   MyObject.apply(current.invoc);
   current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

Return the result after applying my own method call

Shared-Memory Computability





Universal Object

Wait-free/Lock-free computable
=
Threads with methods that solve n-consensus

GetAndSet is not Universal

```
public class RMWRegister {
 private int value;
 public boolean getAndSet(int update)
 {
  int prior = this.value;
  this.value = update;
  return prior;
 }
}
```

GetAndSet is not Universal

```
public class RMWRegister {
 private int value;
 public boolean getAndSet(int update)
 {
  int prior = this.value;
  this.value = update;
  return prior;
 }
}
```

Consensus number 2

GetAndSet is not Universal

```
public class RMWRegister {
 private int value;
 public boolean getAndSet(int update)
 {
  int prior = this.value;
  this.value = update;
  return prior;
 }
}
```

Not universal for ≥ 3 threads

CompareAndSet is Universal

```
public class RMWRegister {
 private int value;
 public boolean
   compareAndSet(int expected,
                 int update) {
  int prior = this.value;
  if (this.value == expected) {
   this.value = update;
   return true;
  }
 return false;
 }}
```

CompareAndSet is Universal

```
public class RMWRegister {
 private int value;
 public boolean
   compareAndSet(int expected,
                 int update) {
  int prior = this.value;
  if (this.value == expected) {
   this.value = update;
   return true;
  }
 return false
 }}
```

Consensus number ∞

CompareAndSet is Universal

```
public class RMWRegister {
 private int value;
 public boolean
   compareAndSet(int expected,
                 int update) {
  int prior = this.value;
  if (this.value == expected) {
   this.value = update;
   return true;
  }
 }}
```

Universal for any number of threads

Practical Implications

- Any architecture that does not provide a universal primitive has inherent limitations
- You cannot avoid locking for concurrent data structures …

Older Architectures

- IBM 360
  - testAndSet (getAndSet)
- NYU UltraComputer
  - getAndAdd
- Neither universal
  - Except for 2 threads

Newer Architectures

- Intel x86, Itanium, SPARC
  - compareAndSet
- Alpha AXP, PowerPC
  - Load-locked/store-conditional
- All universal
  - For any number of threads
- Trend is clear …

SOME RIGHTS RESERVED

This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- **You are free:**
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work
- **Under the following conditions:**
  - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - http://creativecommons.org/licenses/by-sa/3.0/.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.