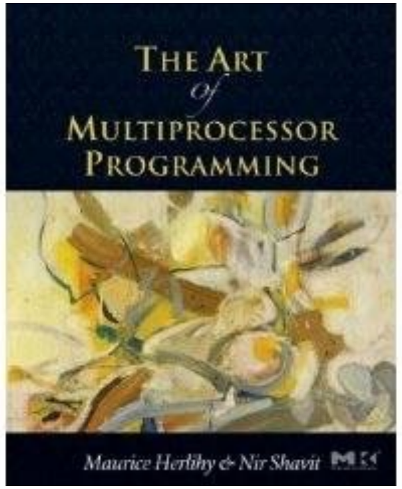
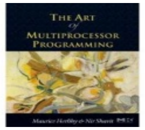
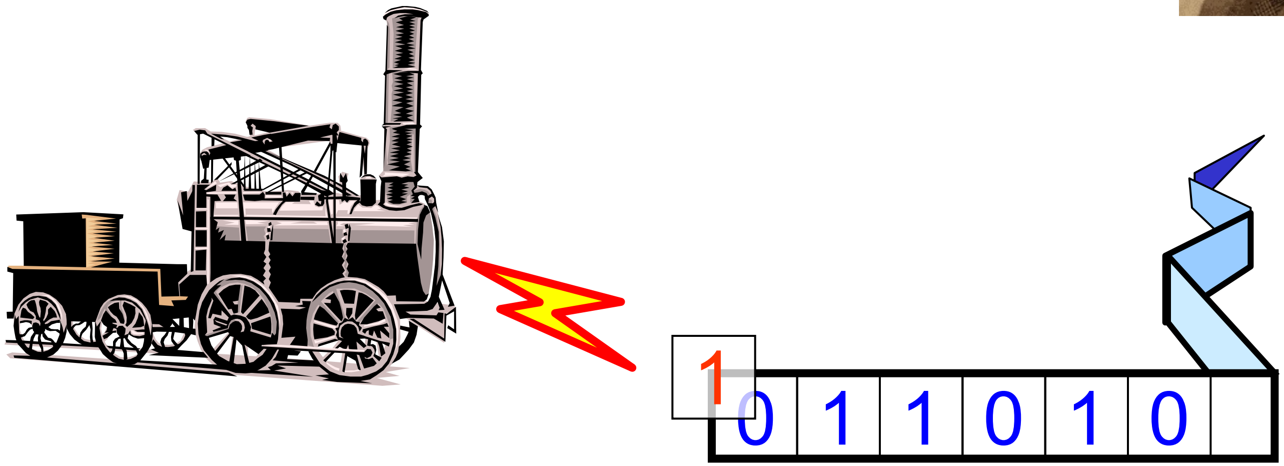
Universality of Consensus



Hyungsoo Jung

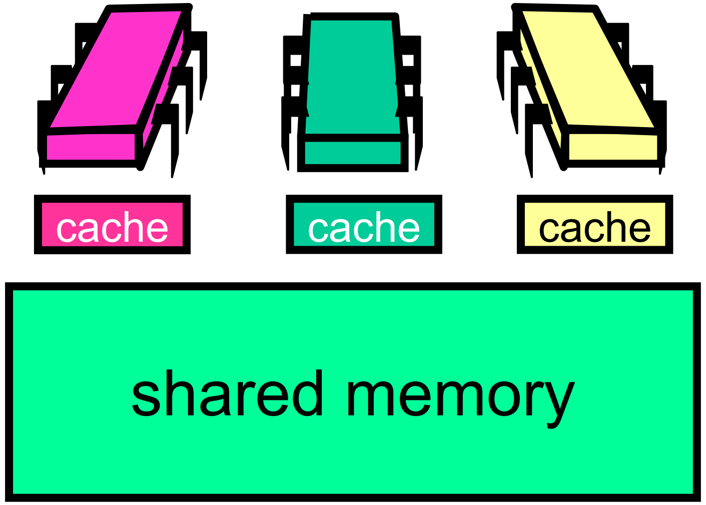


Turing Computability

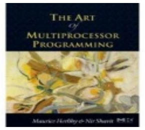


- A mathematical model of computation
- Computable = Computable on a T-Machine

Shared-Memory Computability

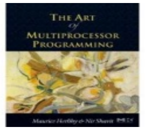


- Model of asynchronous concurrent computation
- Computable = Wait-free/Lock-free computable on a multiprocessor

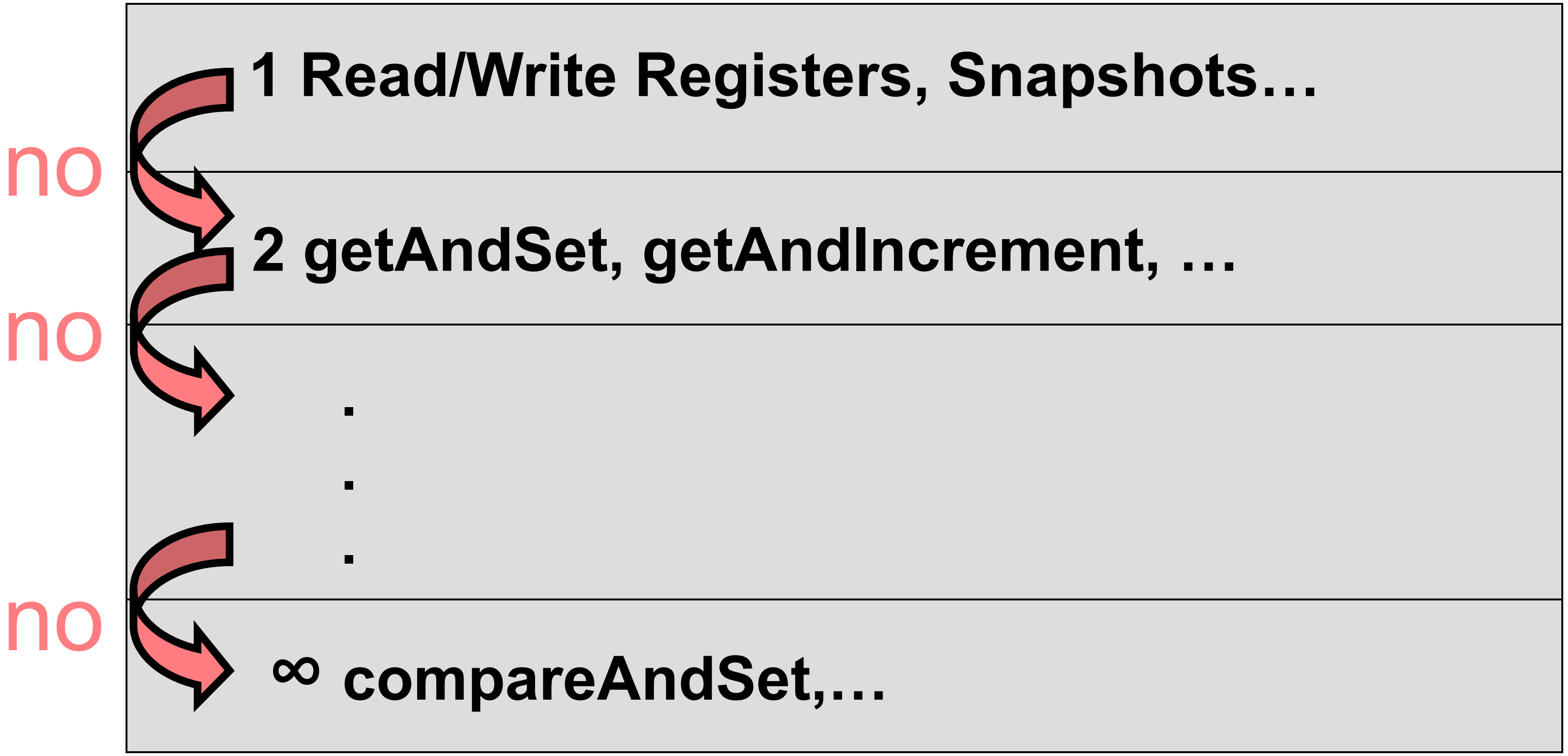


Consensus Hierarchy

1 Read/Write Registers, Snapshots…
2 getAndSet, getAndIncrement, …
. . .
∞ compareAndSet,…



Who Implements Whom?



1 Read/Write Registers, Snapshots…

no

2 getAndSet, getAndIncrement, …

no

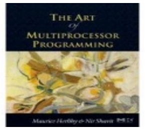
.

.

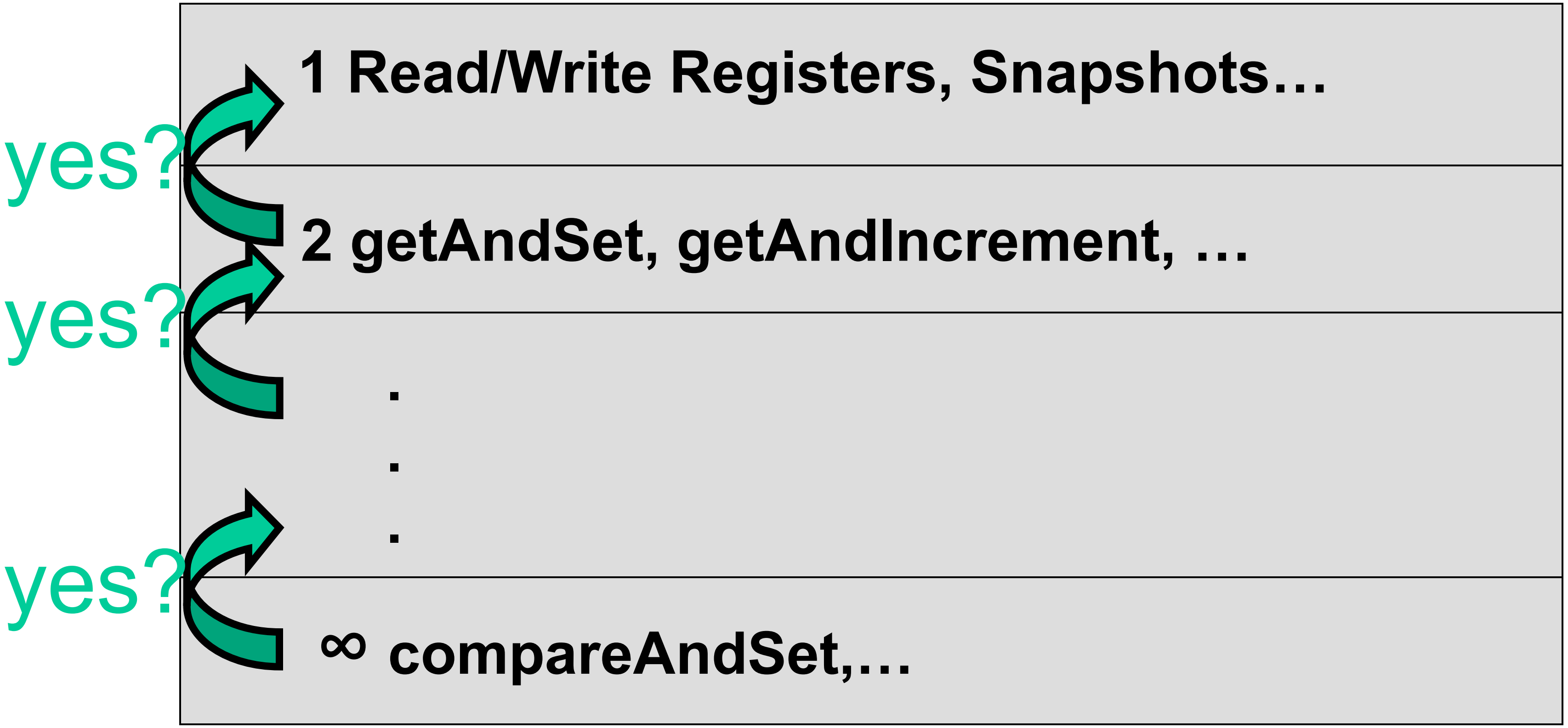
.

no

∞ compareAndSet,…



Hypothesis



1 Read/Write Registers, Snapshots…

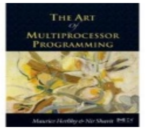
2 getAndSet, getAndIncrement, …

.
.
.

∞ compareAndSet,…

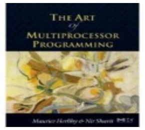
Theorem: Universality

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



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 …

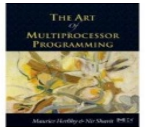


Like a Turing Machine

- 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

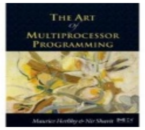
```
public interface SeqObject {
  public abstract
    Response apply(Invocation invoc);
}
```



A Generic Sequential Object

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

Push:5, Pop:null



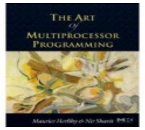
Invocation

```
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

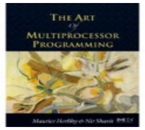
A Generic Sequential Object

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

A Generic Sequential Object

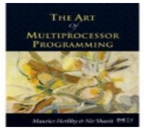
```
public interface SeqObject {
  public abstract
    Response apply(Invocation invoc);
}
```

OK, 4



Response

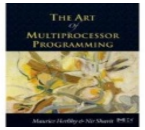
```
public class Response {
  public Object value;
}
```



Response

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

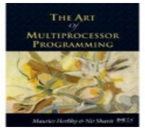
Return value



Universal Concurrent Object

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

A universal concurrent object is linearizable to the generic sequential object



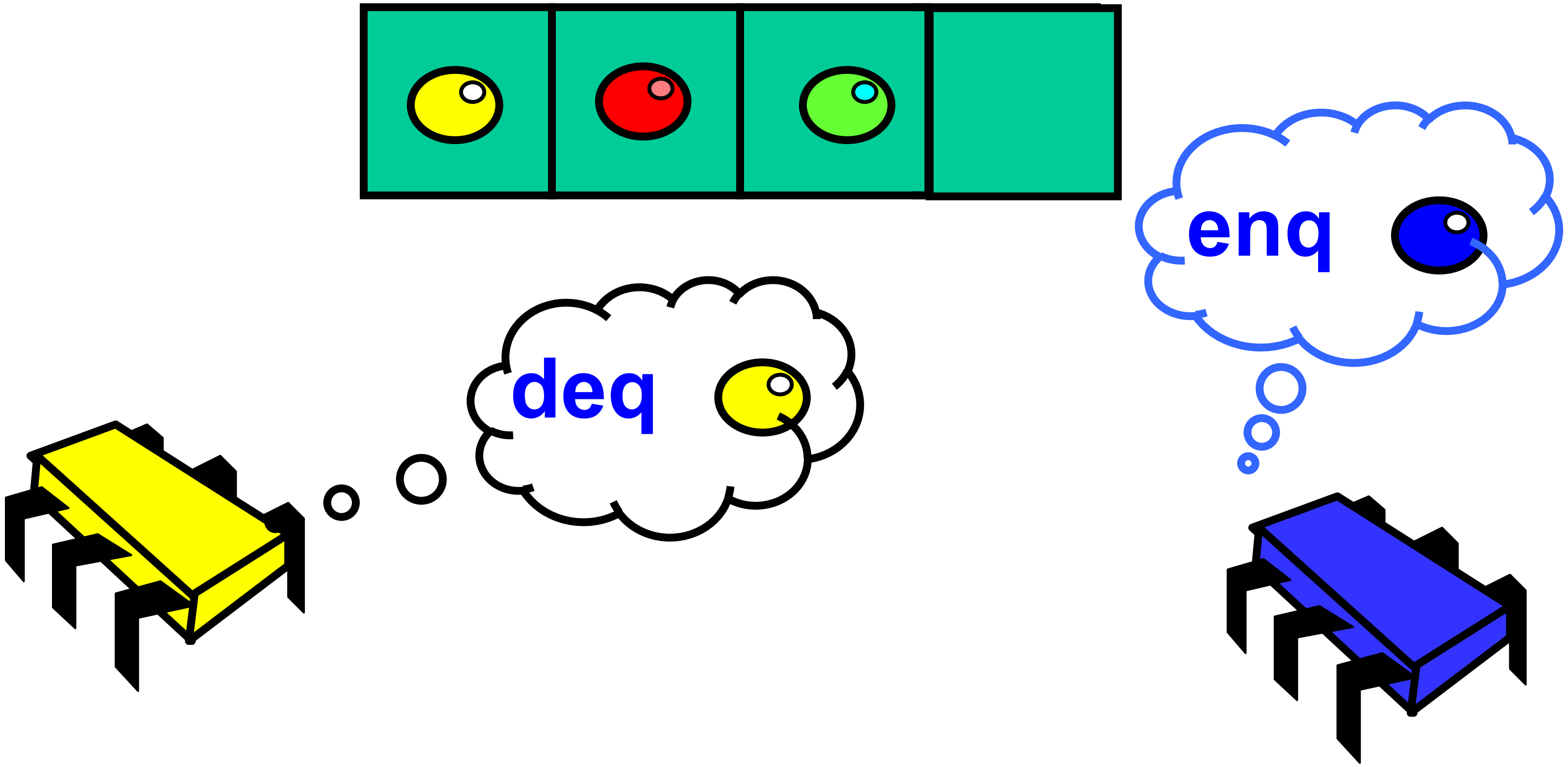
Start with Lock-Free Universal Construction

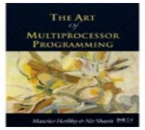
- First Lock-free: infinitely often some method call finishes.
- Then Wait-Free: each method call takes a finite number of steps to finish

Naïve Idea

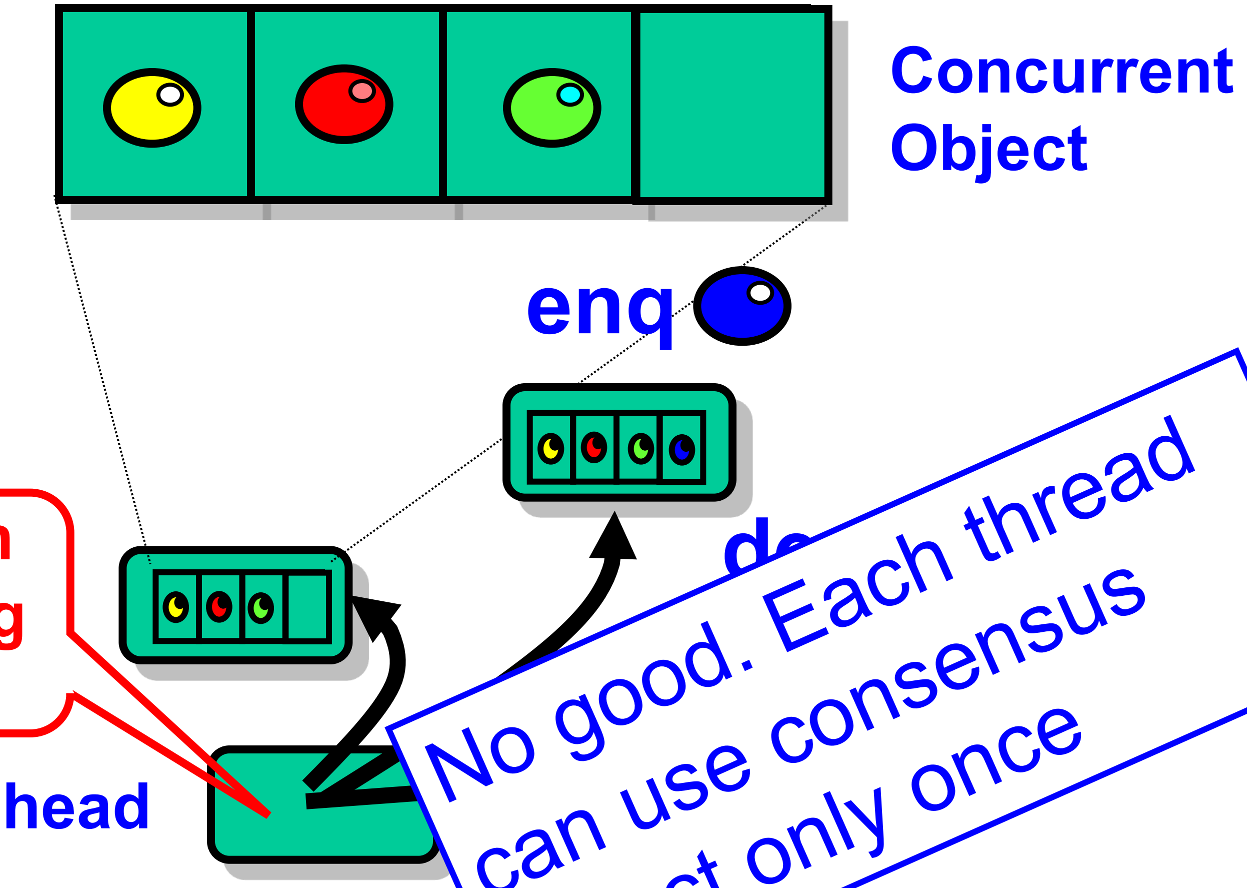
- Consensus object stores reference to cell with current state
- Each thread creates new cell
  - computes outcome,
  - tries to switch pointer to its outcome
- Sadly, no …
  - consensus objects can be used once only

Naïve Idea





Naïve Idea



Decide which to apply using consensus

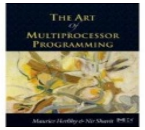
No good. Each thread can use consensus object only once

Once is not Enough?

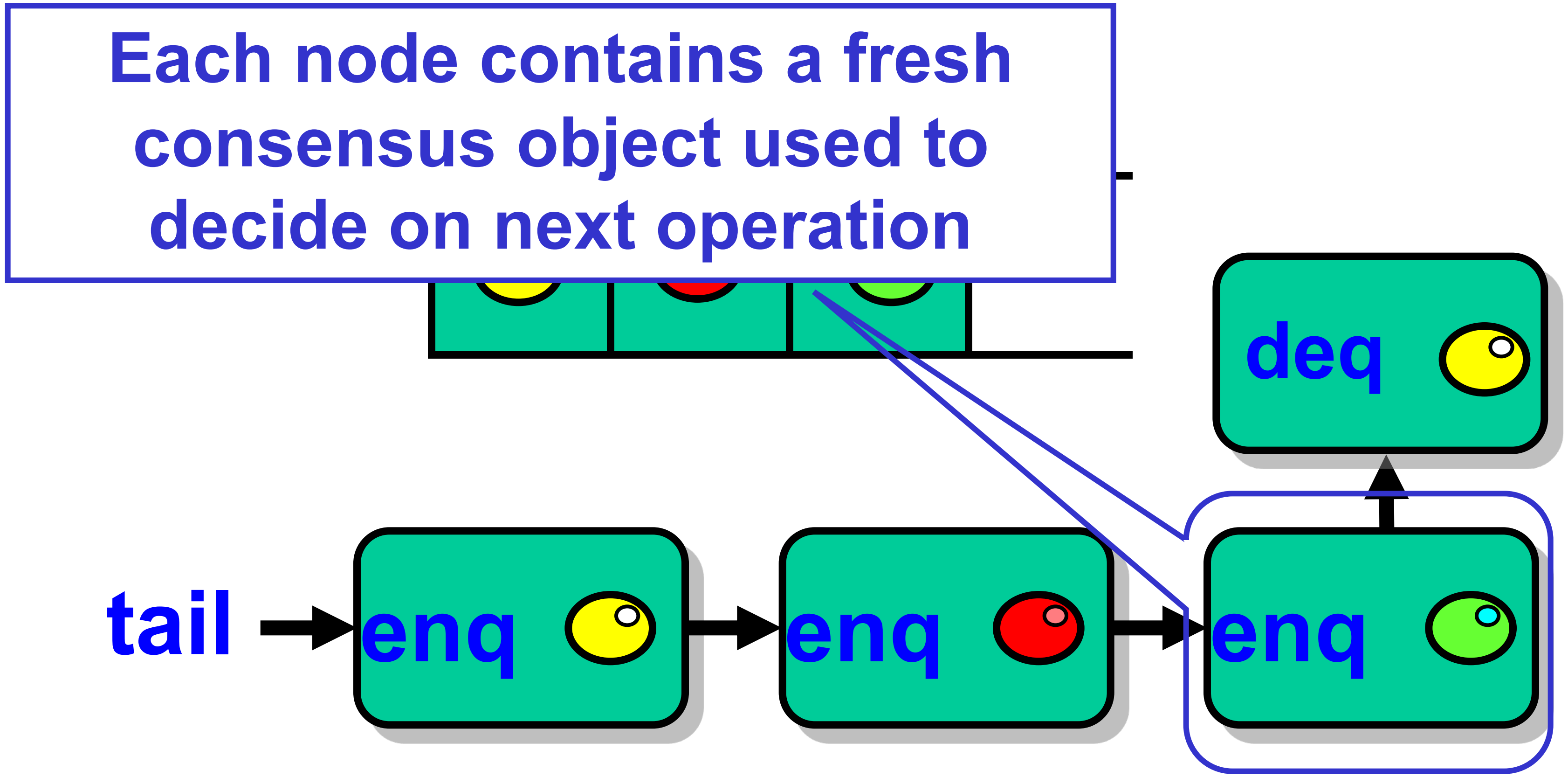
Queue based consensus

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

Solved one-shot 2-consensus.
Not clear how to reuse or reread …



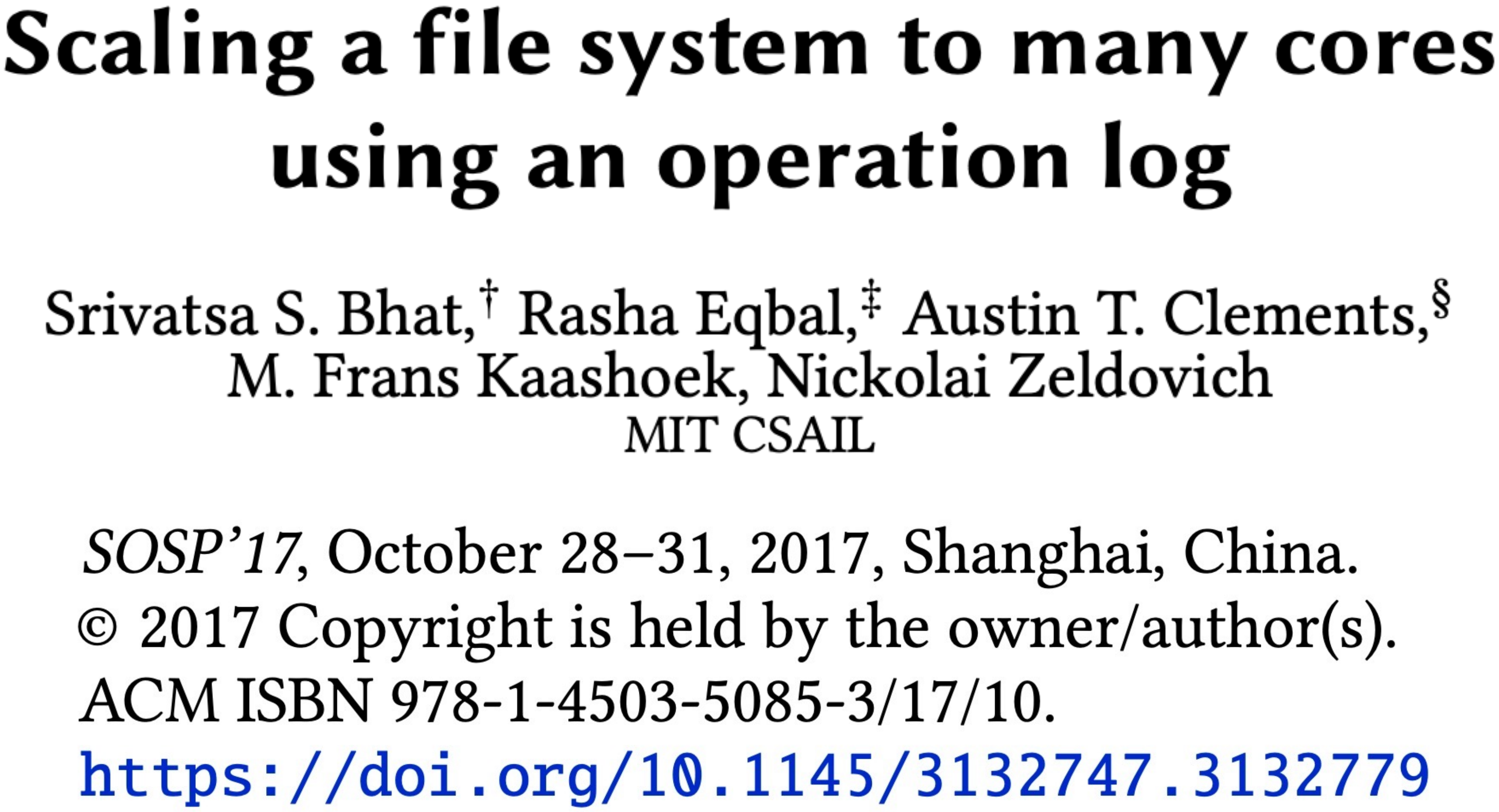
Improved Idea: Linked-List Representation



Each node contains a fresh consensus object used to decide on next operation

Universal Construction

- Object represented as
  - Initial Object State
  - A Log: a linked list of the method calls

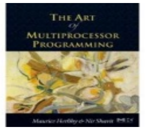


Scaling a file system to many cores using an operation log

Srivatsa S. Bhat,[†] Rasha Eqbal,[‡] Austin T. Clements,[§]
M. Frans Kaashoek, Nickolai Zeldovich
MIT CSAIL

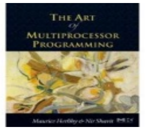
SOSP'17, October 28–31, 2017, Shanghai, China.

ACM ISBN 978-1-4503-5085-3/17/10.
https://doi.org/10.1145/3132747.3132779



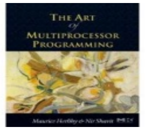
Basic Idea

- Use one-time consensus object to decide next pointer



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

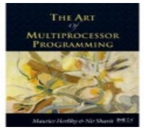


Basic Idea

- Threads use one-time consensus object to decide which node goes next
- Threads update actual next field to reflect consensus outcome
  - 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

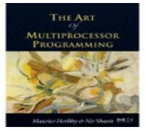
```
public class Node implements
java.lang.Comparable {
 public Invoc invoc;
 public Consensus<Node> decideNext;
 public Node next;
 public int seq;
 public Node(Invoc invoc) {
    invoc = invoc;
    decideNext = new Consensus<Node>()
    seq = 0;
  }
```



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;
  }
```

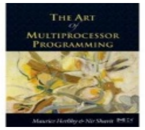
Standard interface for class whose objects are totally ordered



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;
  }
```

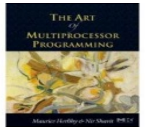
the invocation



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;
  }
```

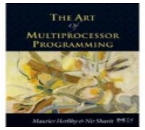
Decide on next node
(next method applied to object)



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;
```

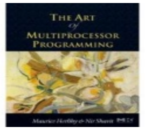
Traversable pointer to next node (needed because you cannot repeatedly read a consensus object)



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;
    decideNext = new Consensus<Node>()
    seq = 0;
  }
```

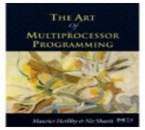
Seq number



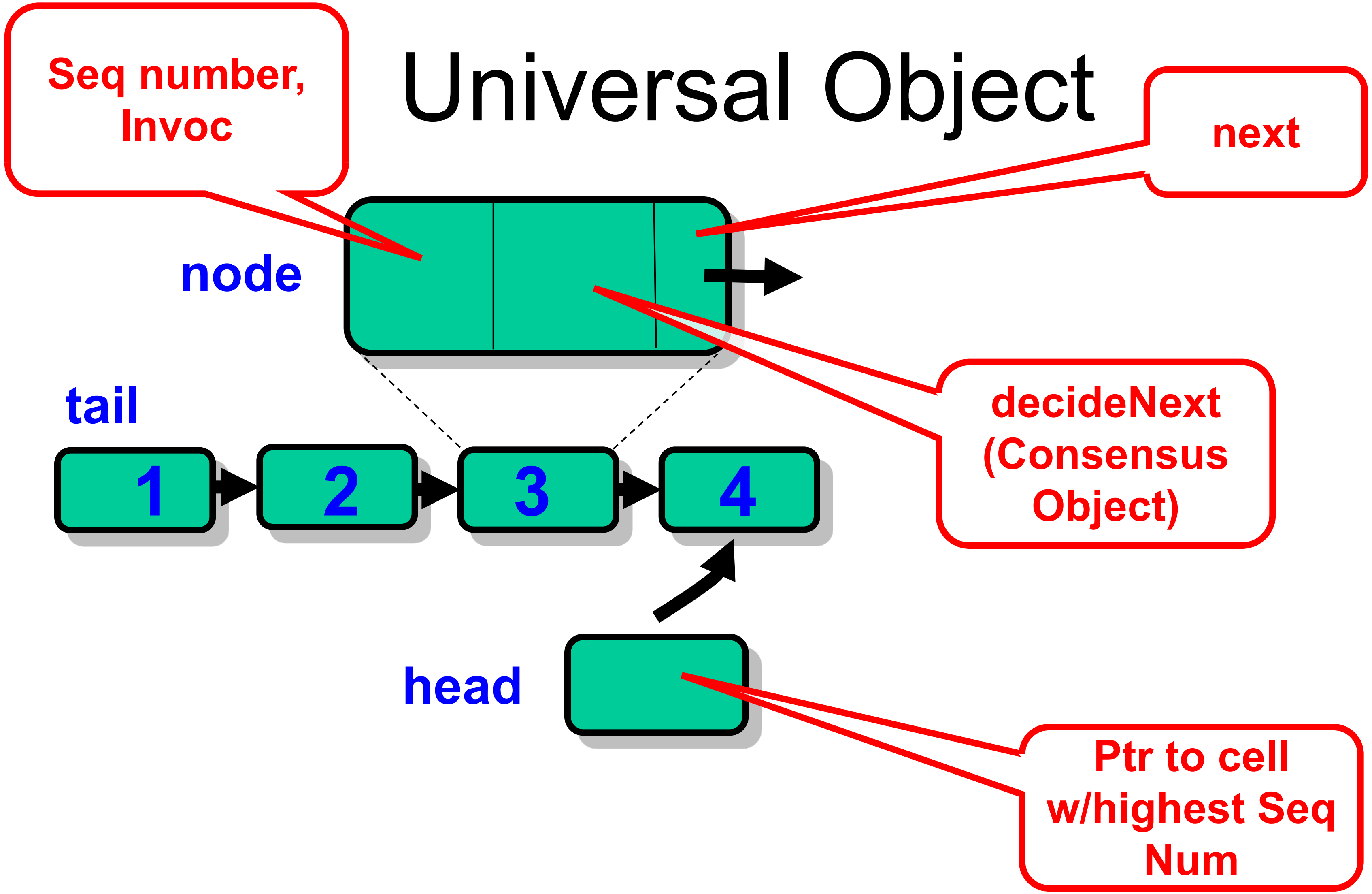
Basic Data Structures

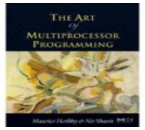
Create new node for this method invocation

```
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;
 }
```

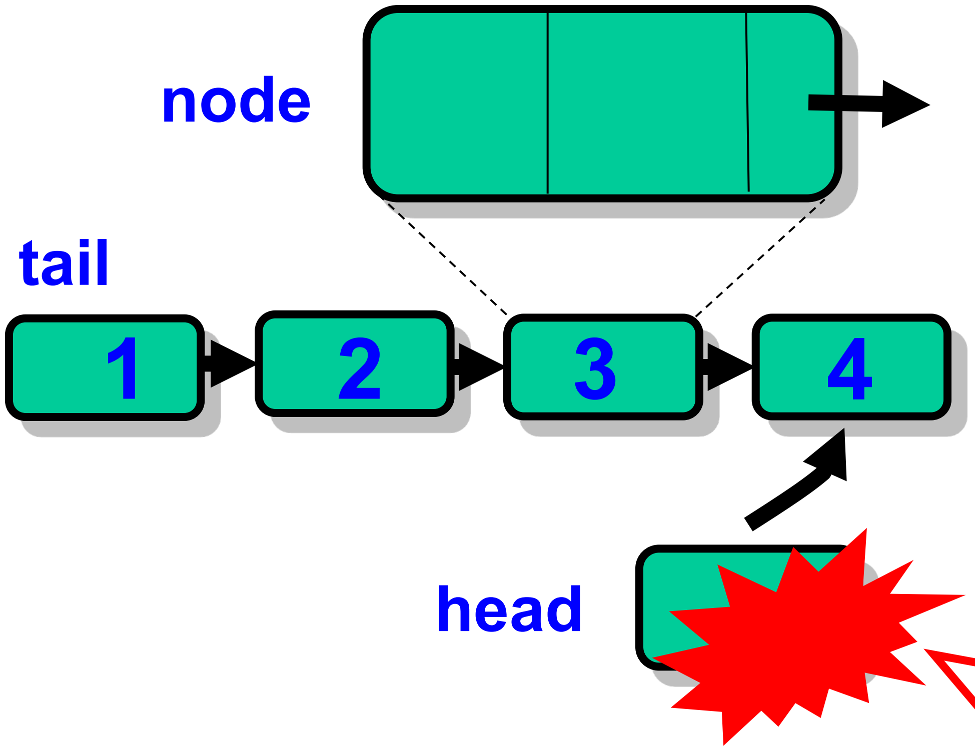


Universal Object

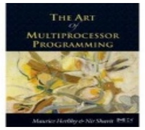




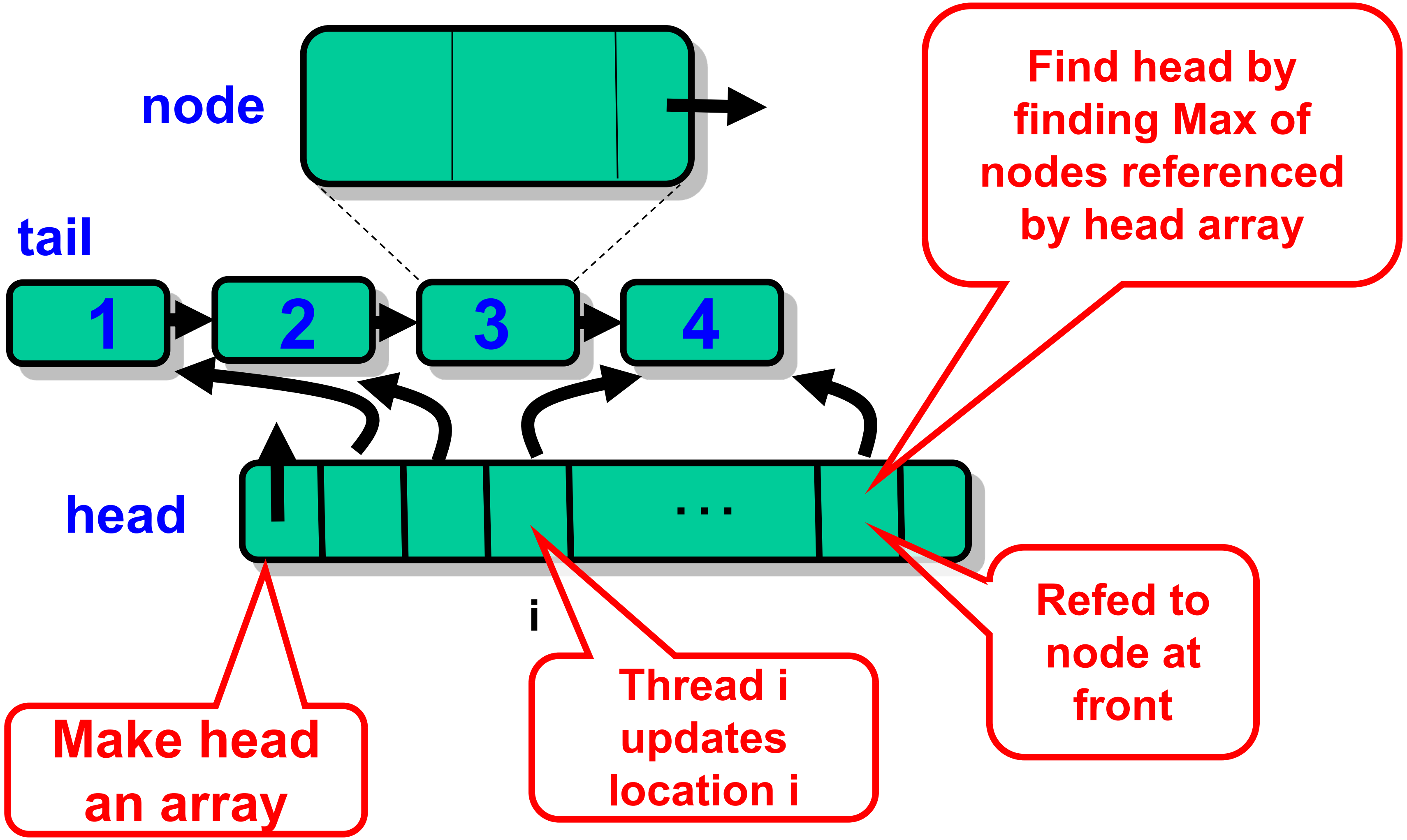
Universal Object



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



The Solution

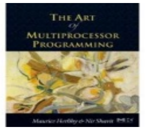


Find head by finding Max of nodes referenced by head array

Refed to node at front

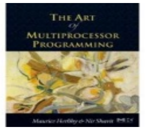
Thread i updates location i

Make head an array



Universal Object

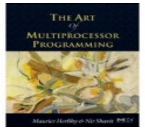
```
public class Universal {
  private Node[] head;
  private Node tail = new Node();
  tail.seq = 1;
  for (int j=0; j < n; j++){
          head[j] = tail
   }
```



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
   }
```

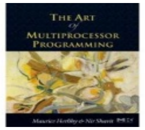
Head Pointers Array



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



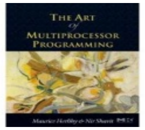
Universal Object

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

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

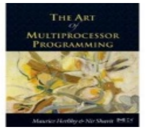
```
public static Node max(Node[] array) {
    Node max = array[0];
    for (int i = 0; i < array.length; i++)
      if (max.seq < array[i].seq)
        max = array[i];
      return max;
  }
```

Compare the seq nums of nodes pointed to by the array

Find Max Head Value

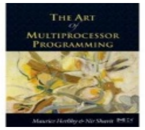
```
public static Node max(Node[] array) {
    Node max = array[0];
    for (int i = 0; i < array.length; i++)
      if (max.seq < array[i].seq)
        max = array[i];
      return max;
  }
```

Return node with maximal sequence number



Universal Application Part I

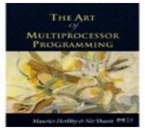
```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
    }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



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;
  …
```

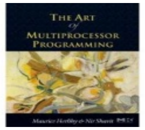
Apply has invocation as input and returns the appropriate response



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;
    he
    }
  …
```

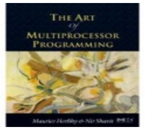
my ID



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

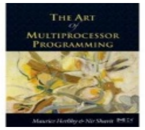
```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
  while (prefer.seq == 0) {
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.
    after.s
    head[i] = after;
    }
  …
```

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.
    after.s
    head[i] = after;
    }
  …
```

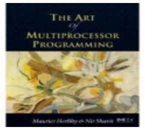
Head of list to which we will try to append



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
  }
}
```

Propose next node

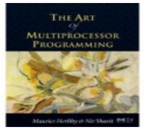


Universal Application

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new node(invoc);
```

Set next field to consensus winner

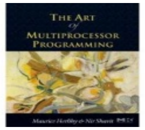
```
    Node before = Node.max(head);
    Node after =
     before.decideNext.decide(prefer);
    before.next = after;
    after.seq = before.seq + 1;
    head[i] = after;
  }
}
```



Universal Application Part I

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

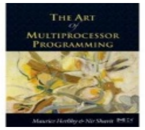
Set seq number
(indicating node was appended)



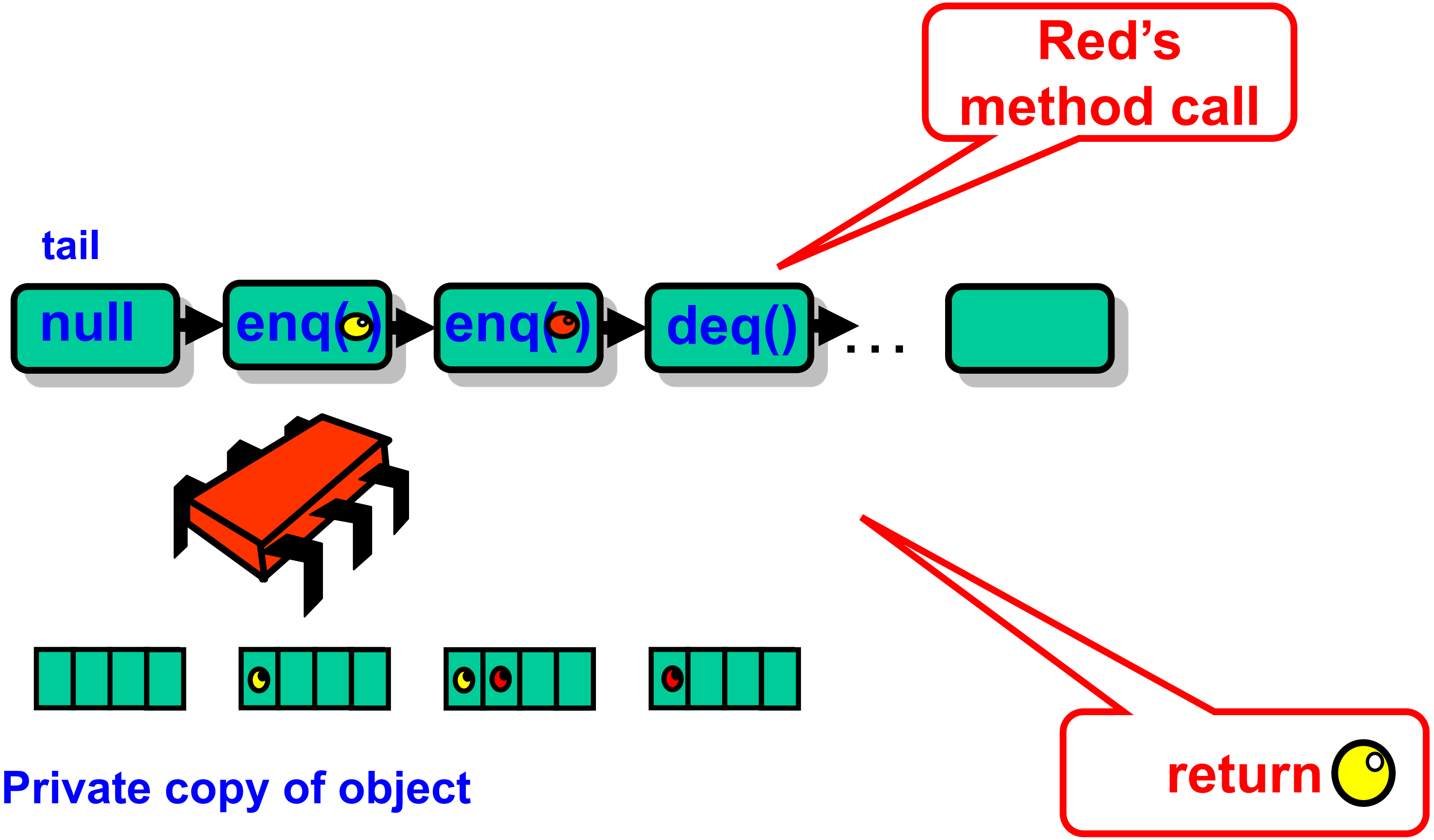
Universal Application Part I

```
public Response apply(Invoc invoc) {
  int i =
  Node pre
  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 2 – Compute Response



Red's method call

tail

null

enq()

enq()

deq()

...

return

Private copy of object

Universal Application Part 2

```
...
//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

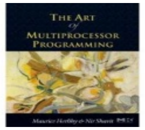
```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
   MyObject.apply(current.invoc);
   current = current.next;
  }
```

Compute result by sequentially applying method calls in list to a private copy

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 copy of 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);
}
```

new method call appended after tail

Universal Application Part II

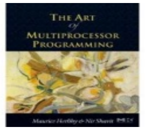
```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
  MyObject.apply(current.invoc);
  current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

While my method call not linked …

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);
}
```

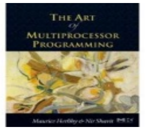
Apply current node’s method



Universal Application Part II

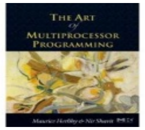
```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;
while (current != prefer){
   MyObject.apply(current.invoc);
   current = current.next;
  }
return MyObject.apply(current.invoc);
}
```

Return result after my method call applied



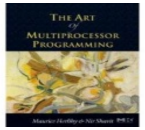
Correctness

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



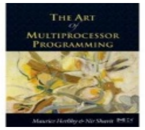
Lock-freedom

- Lock-free because
  - A thread moves forward in list
  - Can repeatedly fail to win consensus on “real” head only if another succeeds
  - Consensus winner adds node and completes within a finite number of steps



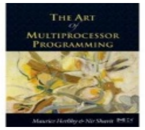
Wait-free Construction

- 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

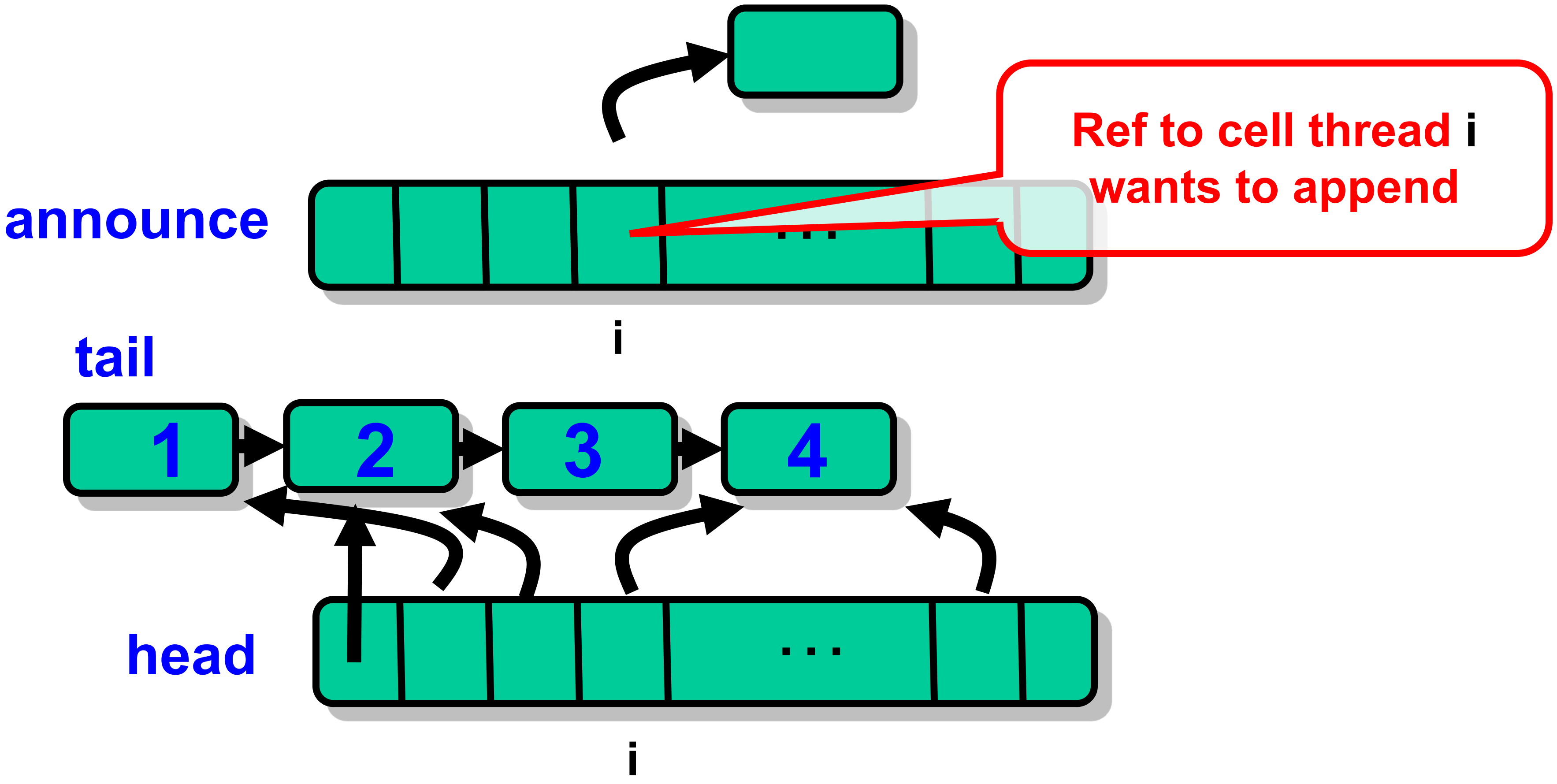


Helping

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



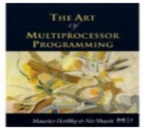
Wait-free Construction



Ref to cell thread i wants to append

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
  };
```



The Announce Array

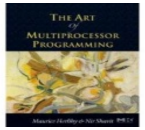
```
public class Universal {
  private Node[] announce;
  private Node[] head;
  private Node tail = new node();
  tail.seq = 1;
  for (int j=0; j < n; j++){
    head[j] = tail; announce[j] = tail
  };
```

New field: 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



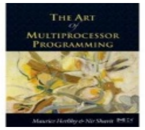
A Cry For Help

```
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
  …
  }
```

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
  …
  }
```

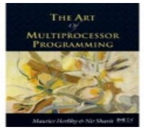
Announce new method call, asking help from others



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
  …
  }
```

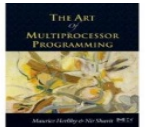
Look for end of list



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 helper)

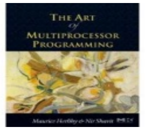


Main Loop

- Non-zero sequence # means success

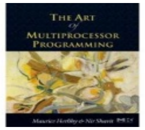
Main Loop

- Non-zero sequence # means success
- Thread keeps helping append nodes



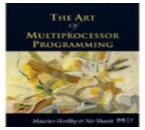
Main Loop

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



Main Loop

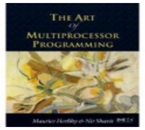
```
while (announce[i].seq == 0) {
 Node before = head[i];
 Node help = announce[(before.seq + 1) % n];
 if (help.seq == 0)
     prefer = help;
    else
     prefer = announce[i];
…
```



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
```

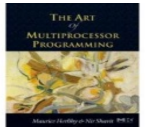
Keep trying until my cell gets a sequence number



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];
```

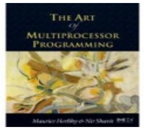
Possible end of list



Main Loop

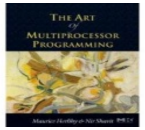
```
while (announce[i].seq == 0) {
 Node before = head[i];
 Node help = announce[(before.seq + 1) % n];
 if (help.seq == 0)
     prefer = help;
    else
     prefer = announce[i];
```

Whom do I help?



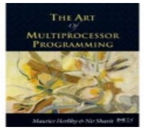
Altruism

- Choose a thread to “help”



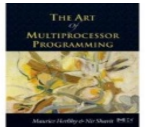
Altruism

- Choose a thread to “help”
- If that thread needs help
  - Try to append its node
  - Otherwise append your own



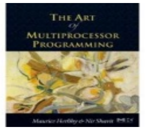
Altruism

- 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



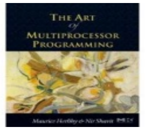
Help!

- When last node in list has sequence number *k*



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



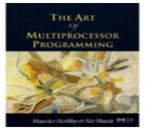
THE ART of MULTIPROCESSOR PROGRAMMING

Help!

- First time after thread *k*+1 announces
  - No guarantees

Help!

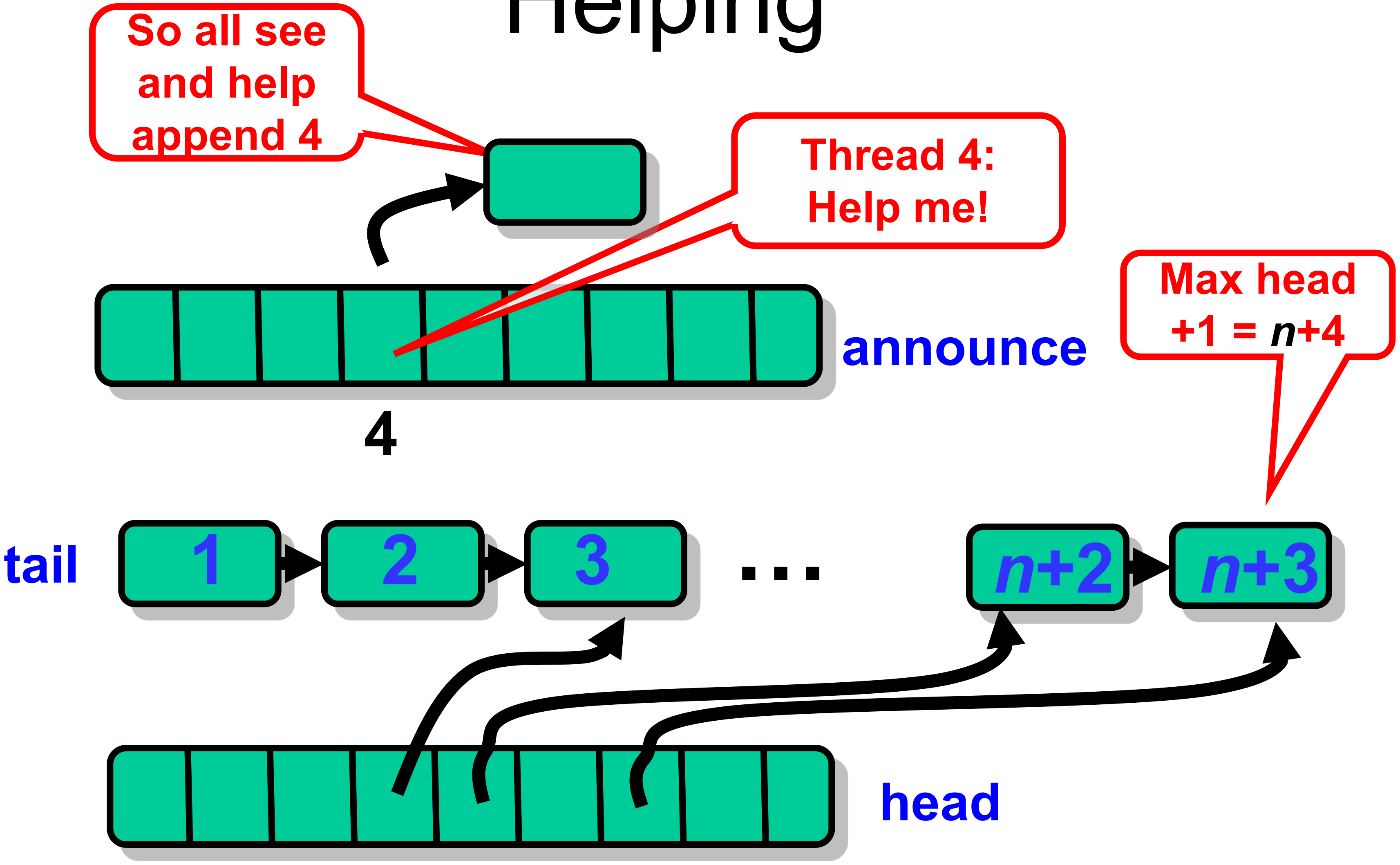
- First time after thread *k*+1 announces
  - No guarantees
- After *n* more nodes appended
  - Everyone sees that thread *k*+1 wants help
  - Everyone tries to append that node
  - Someone succeeds

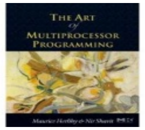


Sliding Window Lemma

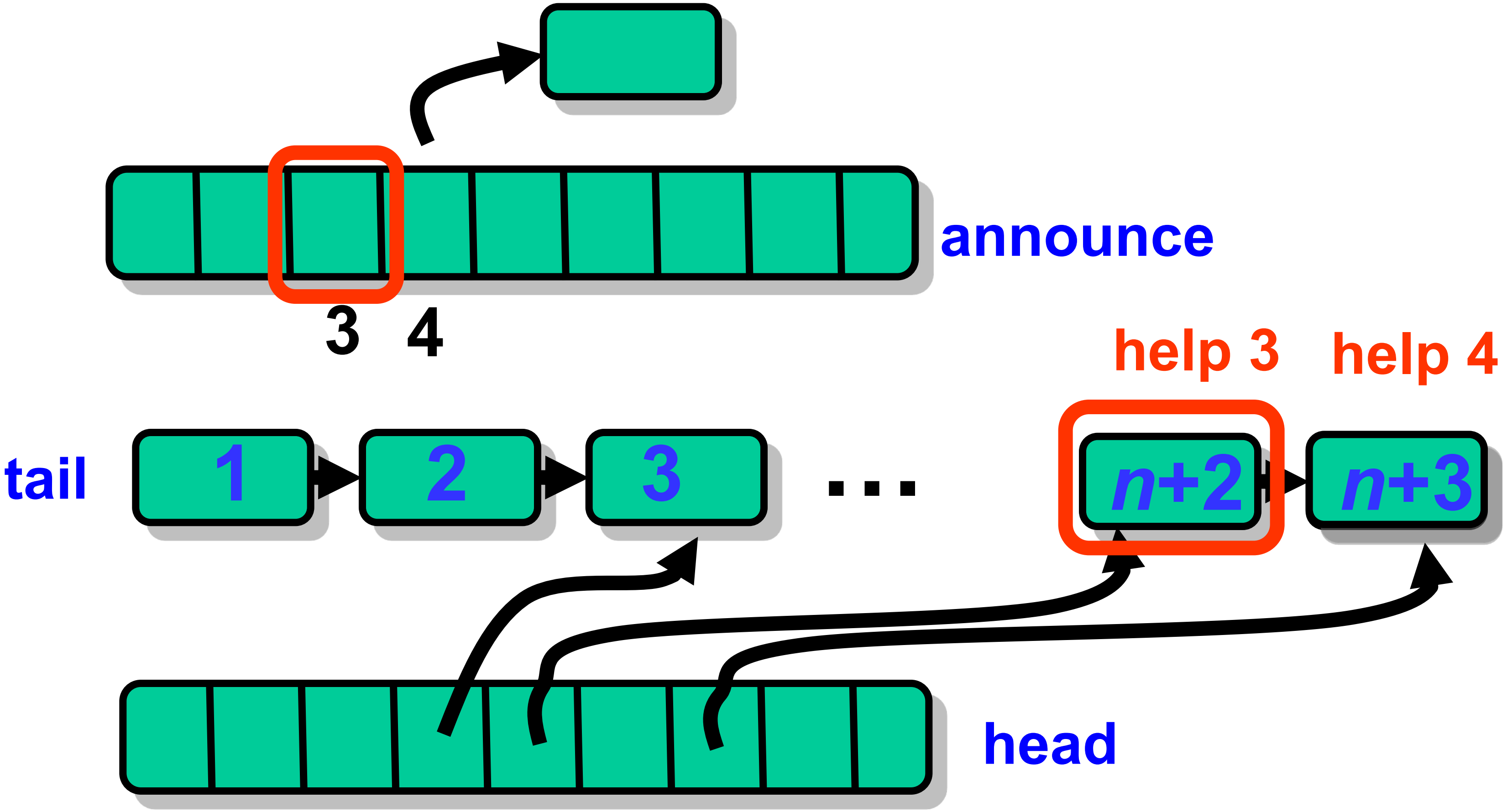
- After thread A announces its node
- No more than *n* other calls
  - Can start and finish
  - Without appending A’s node

Helping


So all see and help append 4
Thread 4: Help me!
Max head +1 = n+4
announce
4
tail
1
2
3
...
n+2
n+3
head



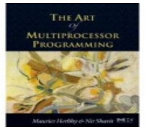
The Sliding Help Window



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



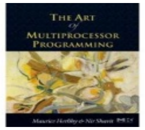
Sliding Help Window

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

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

Rest is Same as Lock-free

```
while (announce[i].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

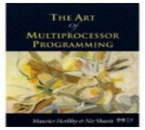
```
while (announce[i].seq == 0) {
  …
  Node after =
   before.decideNext.decide(prefer);
  before.next = after;
  after.seq = before.seq + 1;
  head[i] = after;
 }
```

Call consensus to attempt to append

Rest is Same as Lock-free

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

cache consensus result for later use



Rest is Same as Lock-free

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

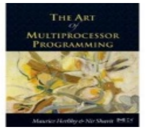
Tell world that node is appended

Finishing the Job

- Once thread’s node is linked …

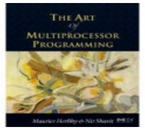
Finishing the Job

- Once thread’s node is linked…
- The rest same as lock-free algorithm



Finishing the Job

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



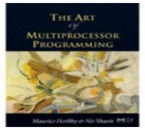
Then Same Part II

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

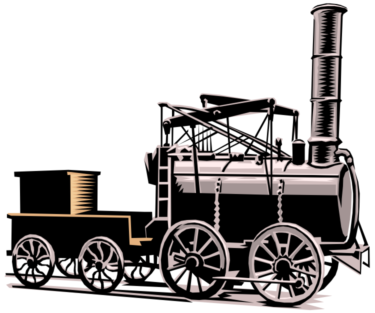
Universal Application Part II

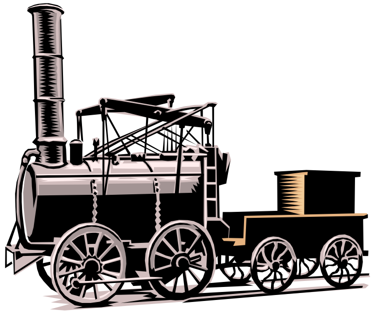
```
...
//compute my response
SeqObject MyObject = new SeqObject();
current = tail.next;

  MyObject.apply(current.invoc);
  current = current.next;
 }
return MyObject.apply(current.invoc);
}
```

Return result after applying my method



Shared-Memory Computability





Universal Construction

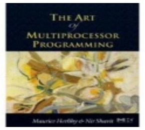
Wait-free/Lock-free computable

=

Solving *n*-consensus

Veni, Vidi, Vici

- We saw
  - how to define concurrent objects



Veni, Vidi, Vici

- We saw
  - how to define concurrent objects
- We discussed
  - computational power of machine instructions

Veni, Vidi, Vici

- We saw
  - how to define concurrent objects
- We discussed
  - computational power of machine instructions
- Next
  - use these foundations to understand the real world