Universality of Consensus

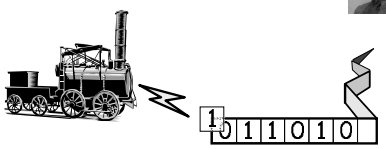
BROWN

Maurice Herlihy

CS176

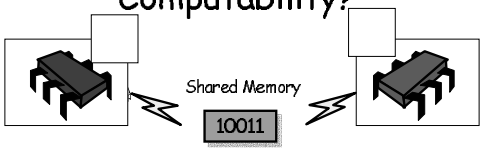
Fall 2003

Turing Computability



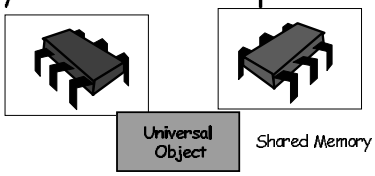
- A mathematical model of computation
- Complexity irrelevant to real machines
- Computable = Computable on a T-Machine

Shared-Memory Computability?



- A model of concurrent computation
- Complexity irrelevant to real machines
- Wait-free/Lock-free computable = ????

Asynchronous Computability



- A model of concurrent computation
- Complexity irrelevant to real machines
- Wait-free/Lock-free computable = ????

Theorem: Universality

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

Proof Outline

- Constrruct a universal object
  - From n-consensus objects
  - And atomic registers
- Object implements any sequentially specified object canonically

Like a Turing Machine

- This construction
  - Not intended to be practical
  - But enlightening
- 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)

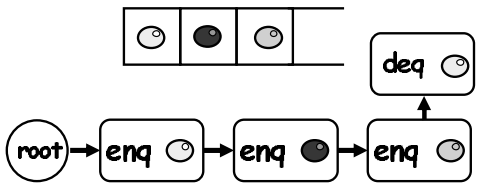
Universal Construction

- Object implemented as a
  - List of method calls
- Method call
  - Find end of list
  - Atomically append your call

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

Basic Idea: Linked-List Representation



Basic Idea

- Use one-time consensus object as next pointer
- Challenges
  - How to avoid starvation?
  - What if a thread stops in the middle?

Stylized Sequential Objects

- Object itself is *immutable*
  - State never changes
- Method call produces
  - New copy of object in new state
  - Return value (or exception)

Stylized Sequential Objects

- Invocation
  - Method name
  - Arguments
- Response
  - Copy of modified object
  - Return value
- Assume methods are deterministic
  - Only one response (can relax restriction)

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

Response

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


Response

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

New object state

Response

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

Return value

Sequential Objects

```
public class SeqObject {
  public Response apply(Invoc invoc) {
    …
  }}
```


Sequential Objects

```
public class SeqObject {
  public Response apply(Invoc invoc) {
    …
  }}
```

Applies method invocation, returns new state and result

Basic Data Structures

```
public class Cell {
 int seq;
 Invoc invoc;
 Response response;
 Consensus next;
}
```


Basic Data Structures

```
public class Cell {
 int seq;
 Invoc invoc;
 Response response;
 SeqObject next;
}
```

Sequence number: zero while in play

Basic Data Structures

```
public class Cell {
 int seq;
 Invoc invoc;
 Response response;
 Consensus next;
}
```

Method name & args

Basic Data Structures

```
public class Cell {
 int seq;
 Invoc invoc;
 Response response;
 Consensus next;
}
```

New object state & return value

Basic Data Structures

```
public class Cell {
 int seq;
 Invoc invoc;
 Response response;
 Consensus next;
}
```

Consensus object that indicates next cell in list

Cell Constructor

```
public Cell(Invoc invoc) {
  this.seq      = 0;
  this.invoc    = invoc;
  this.response = null;
  this.next     = new Consensus();
}
```


Cell Constructor

```
public Cell(Invoc invoc) {
  this.seq      = 0;
  this.invoc    = invoc;
  this.response = null;
  this.next     = new Consensus();
}
```

Sequence number zero means call still incomplete

Cell Constructor

```
public Cell(Invoc invoc) {
  this.seq      = 0;
  this.invoc    = invoc;
  this.response = null;
  this.next     = new Consensus();
}
```

Record method name & args

Cell Constructor

```
public Cell(Invoc invoc) {
  this.seq      = 0;
  this.invoc    = invoc;
  this.response = null;
  this.next     = null;
}
```

Winning thread decides

Cells are Comparable

```
class Cell implements
  java.util.Comparable {
 int compareTo(Cell other) {
  if (this.seq > aCell.seq)
   return 1;
  else if (aCell.seq > this.seq)
   return -1
  else
   return 0;
}}
```


Cells are Comparable

```
class Cell implements
  java.util.Comparable {
 int compareTo(Cell other) {
  if (this.seq > aCell.seq)
   return 1;
  else if (aCell.seq > this.seq)
   return -1
  else
   return 0;
}}
```

Standard interface for class whose objects are totally ordered

Cells are Comparable

```
class Cell implements
  java.util.Comparable {
 int compareTo(Cell other) {
  if (this.seq > aCell.seq)
   return 1;
  else if (aCell.seq > this.seq)
   return -1
  else
   return 0;
}}
```

Returns +1 if I'm greater, -1 if I'm lesser, and 0 otherwise

Later is Greater

```
class Cell implements
  java.util.Comparable {
 int compareTo(Cell other) {
  if (this.seq > aCell.seq)
   return 1;
  else if (aCell.seq > this.seq)
   return -1
  else
   return 0;
}}
```

I'm greater if my sequence number is higher

Cells are Comparable

```
class Cell implements
  java.util.Comparable {
 int compareTo(Cell other) {
  if (this.seq > aCell.seq)
   return 1;
  else if (aCell.seq > this.seq)
   return -1
  else
   return 0;
}}
```

And vice-versa …

Utility method

```
class Cell implements Comparable {
 …
 static Cell max(Cell[] array) {
  Cell max = array[0];
  for (int i=0; i<array.length; i++)
   if (max.compareTo(array[i]) < 0)
    max = array[i];
  return max;
 }
}
```

Utility method

```
class Cell implements Comparable {
 …
 static Cell max(Cell[] array) {
  Cell max = array[0];
  for (int i=0; i<array.length; i++)
   if (max.compareTo(array[i]) < 0)
    max = array[i];
  return max;
 }
}
```

Find latest cell in array

Universal Object

```
public class Universal {
  private Cell[] announce;
  private Cell[] head;
  …
}
```


Universal Object

```
public class Universal {
  private Cell[] announce;
  private Cell[] head;
  …
}
```

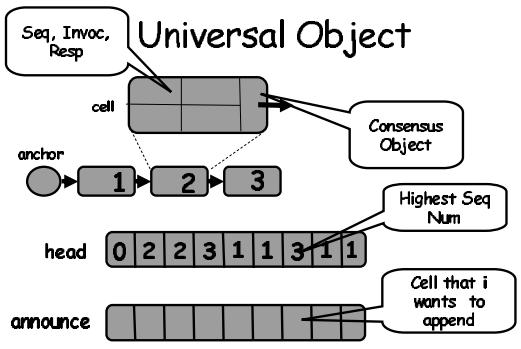
If this thread does not succeed, another thread will help out

Universal Object

```
public class Universal {
  private Cell[] announce;
  private Cell[] head;
  …
}
```

Find the end of the list

Universal Object



Thread A

- Allocates cell to represent call
- Stores (pointer to) cell in announce
  - If A doesn't execute it
  - Another thread will
- Looks for thread near end of list
  - By scanning head array
  - Choosing cell with largest sequence num

Helping

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

A Cry For Help

```
public class Universal {
 public Object apply(Invoc invoc) {
   int i = Thread.myIndex();
   this.announce[i] = new Cell(invoc);
   for (int j = 0; j < N; j++)
     this.head[i] =
       max(this.head[i], this.head[j]);
   …
```


A Cry For Help

```
public class Universal {
 public Object apply(Invoc invoc) {
   int i = Thread.myIndex();
   this.announce[i] = new Cell(invoc);
   for (int j = 0; j < N; j++)
     this.head[i] =
       max(this.head[i], this.head[j]);
   …
```

Announce my intention to append cell to list

Where does it End?

- Need to find end of list
  - Can't start from anchor
  - Starvation for slow threads
- Each thread records last cell seen
- If all collect from that array
  - Some thread has last cell

Assume the Position

```
public class Universal {
 public Object apply(Invoc invoc) {
   int i = Thread.myIndex();
   this.announce[i] = new Cell(invoc);
   this.head[i] = max(this.head);
   …
```


Assume the Position

```
public class Universal {
 public Object apply(Invoc invoc) {
   int i = Thread.myIndex();
   this.announce[i] = new Cell(invoc);
   this.head[i] = max(this.head);
   …
```

Look for end of list

Are We Done Yet?

- Non-zero sequence number indicates success
- Thread keeps appending cells
- Until its own cell is done

Keep on Keeping on

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


Keep on Keeping on

```
while (this.announce[i].seq == 0) {
 Cell before, help, prefer;
 before = this.head[i];
 help = announce[(before.seq+1) % n];
 if (help.seq == 0)
   prefer = help;
   Keep trying until my cell gets a
          sequence number
}
```


Keep on Keeping on

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

Possible end of list

Altruism

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

Help!

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

Help!

- All threads try to help k+1 (mod n)
- First time after k+1 announces
  - Some may see announcement
  - Some may not
- "Many are cold but few are frozen"

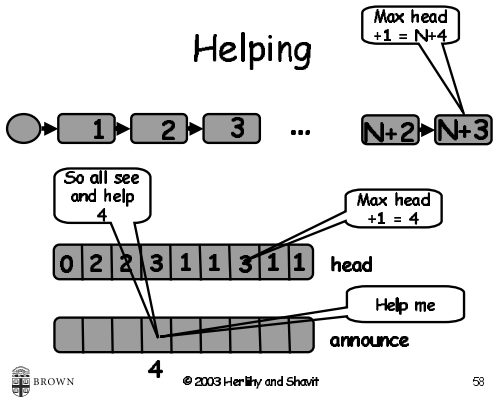
Help!

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

Lemma

- After A announces cell
- No more than n other calls
  - Can start and finish
  - Without appending A's cell

Helping



Pull Together

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


Pull Together

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

Pick another thread to help

Pull Together

```
While (1
 Cell be
 before = this.head[i];
 help = announce[(before.seq % n) +1];
 if (help.seq == 0)
   prefer = help;
 else
   prefer = this.announce[i];
 …
}
```

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

Quality is Job 1.1

```
While (this.announce[i].seq == 0) {
 …
 before.next.propose(prefer);
 Cell after = before.next.decide();
 SeqObject oldObject =
  before.response.object;
 Response response =
  oldObject.apply(after.invoc);
 after.nextState.propose(response);
 after.response =
  (Response)after.nextState.decide();
 …}
```


Quality is Job 1.1

```
While (this.announce[i].seq == 0) {
 …
 before.next.propose(prefer);
 Cell after = before.next.decide();
 SeqObject oldObject =
  before.response.object;
 Response response =
  oldObject.apply(after.invoc);
 …}
```

Propose my favorite, then find out who actually won

Finishing the Job

- Once we have linked in a cell
- Make sure remaining fields are filled in before moving on
- New object state in response field

Quality is Job 1.1

```
while (this.announce[i].seq == 0) {
 …
 before.next.propose(prefer);
 Cell after = before.next.decide();
 SeqObject oldObject =
  before.response.object;
 after.response =
  oldObject.apply(after.invoc);
 after.seq = before.seq + 1;
 this.head[i] = after;
}
```


Quality is Job 1.1

```
while (this.announce[i].seq == 0) {
 …
 before.next.propose(prefer);
 Cell after = before.next.decide();
 SeqObject oldObject =
  before.response.object;
 after.response =
  oldObject.apply(after.invoc);
 after.seq = before.seq + 1;
 this.he
 }
```

Get object's prior state

Quality is Job 1.1

```
while (this.announce[i].seq == 0) {
 Apply method and
  fill in response
 SeqObject oldObject =
  before.response.object;
 after.response =
  oldObject.apply(after.invoc);
 after.seq = before.seq + 1;
 this.head[i] = after;
 }
```


Quality is Job 1.1

```
while (this.announce[i].seq == 0) {
   Fill in sequence
      number
 SeqObject oldObject =
  before.response.object;
 after.response =
  oldObject.apply(after.invoc);
 after.seq = before.seq + 1;
 this.head[i] = after;
 }
```


Quality is Job 1.1

```
while (this.annou
 …
 before.next.propose(prefer);
 Cell after = before.next.decide();
 SeqObject oldObject =
  before.response.object;
 after.response =
  oldobject.apply(after.invoc);
 after.seq = before.seq + 1;
 this.head[i] = after;
 }
```

Note: Multiple threads could fill in these fields

Quality is Job 1.1

```
while (this.announce[i
 …
 before.next.propose(p
 Cell after = before.next.decide();
 SeqObject oldObject =
  before.response.object;
 after.response =
  oldObject.apply(after.invoc);
 after.seq = before.seq + 1;
 this.head[i] = after;
 }
```

Why is such duplication OK?

Quality is Job 1.1

```
while (this.announce[i].seq == 0) {
 …
 before.next.propose(prefer);
 Cell after = before.next.decide();
 SeqObject oldObject =
  before.respc
 after.response =
  oldObject.apply(after.invoc);
 after.seq = before.seq + 1;
 this.head[i] = after;
}
```

Advance my end of list

Asynchronous 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;
  }
 return false
 }}
```

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 …

Till Now: Correctness

- Models
  - Accurate (we never lied to you)
  - But idealized (so we forgot to mention a few things)
- Protocols
  - Elegant
  - Important
  - But naïve

Next Time: Performance

- Models
  - More complicated (not the same as complex!)
  - Still focus on principles (not soon obsolete)
- Protocols
  - Elegant (in their fashion)
  - Important (why else would we pay attention?)
  - And realistic (your mileage may vary)

Comments on Universality

Maurice: I removed non-determinism from the
Code but there are still redundent fields in each cell.
Please fix code. Also, from comments and
questions we need to think of a better way to define
The max function of the seq numbers in the nodes since
It confused several students who came up
with counter examples simply because they missed the fact
That the max is on the seq numbers…