Foundations of Shared Memory

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

Art of Multiprocessor Programming
Last Lecture

- Defined concurrent objects using linearizability and sequential consistency
- **Fact:** implemented linearizable objects (Two thread FIFO Queue) in read-write memory without mutual exclusion
- **Fact:** hardware does not provide linearizable read-write memory
Fundamentals

• What is the weakest form of communication that supports mutual exclusion?
• What is the weakest shared object that allows shared-memory computation?
Alan Turing

• Showed what is and is not computable on a sequential machine.
• Still best model there is.
Turing Computability

- Mathematical model of computation
- What is (and is not) computable
- Efficiency (mostly) irrelevant

Art of Multiprocessor Programming
Shared-Memory Computability?

- Mathematical model of concurrent computation
- What is (and is not) concurrently computable
- Efficiency (mostly) irrelevant
Foundations of Shared Memory

To understand modern multiprocessors we need to ask some basic questions …
To understand modern multiprocessors we need to ask some basic questions.

What is the weakest useful form of shared memory?
Foundations of Shared Memory

To understand modern multiprocessors we need to ask some basic questions...

What is the weakest useful form of shared memory?

What can it do?
Register*

Holds a (binary) value

10011

* A memory location: name is historical
Register

Can be read

10011

10011
Register

Can be written

01100

10011
public interface Register\<T\> {
    public T read();
    public void write(T v);
}
public interface Register\<T> {  
    public T read();  
    public void write(T v);  
}  

Type of register  
(usually Boolean or \textit{m}-bit Integer)
Single-Reader/Single-Writer Register

01100

10011

10011
Multi-Reader/Single-Writer Register

Art of Multiprocessor Programming
Multi-Reader/Multi-Writer Register

Art of Multiprocessor Programming
Jargon Watch

• SRSW
  – Single-reader single-writer

• MRSW
  – Multi-reader single-writer

• MRMW
  – Multi-reader multi-writer
Safe Register

OK if reads and writes don’t overlap

write(1001)

read(1001)
Safe Register

Some valid value if reads and writes do overlap

write(1001)

read(?????)

0000  1001  1111  $*&v
Regular Register

- Single Writer
- Readers return:
  - Old value if no overlap (safe)
  - Old or one of new values if overlap
Regular or Not?

write(0) -> read(1) -> write(1) -> read(0) -> write(0)
Regular or Not?

Overlap: returns new value
Regular or Not?

write(0)  write(1)  read(0)

Overlap: returns old value

Art of Multiprocessor Programming
Regular or Not?

write(0)  write(1)  read(1)  read(0)

regular
Regular ≠ Linearizable

write(0) → write(1) → read(1) → read(0)

write(1) already happened

can’t explain this!
Atomic Register

write(1001) → write(1010) → read(1010) → read(1001) → read(1010)

Linearizable to sequential safe register
Atomic Register

write(1001) -> write(1010) -> read(1010) -> read(1001) -> read(1010)
Register Space

MRMW

MRSW

SRSW

Safe

Regular

Atomic

Boolean

$m$-valued

Art of Multiprocessor Programming
Weakest Register

Single writer

Safe Boolean register

Single reader
Weakest Register

Single writer

Single reader

Get correct reading if not during state transition
Results

- From SRSW safe Boolean register
  - All the other registers
  - Mutual exclusion
- But not everything!
  - Consensus hierarchy

Foundations of the field

The really cool stuff …
Locking within Registers

• Not interesting to rely on mutual exclusion in register constructions
• We want registers to implement mutual exclusion!
• It’s cheating to use mutual exclusion to implement itself!
Definition

An object implementation is *wait-free* if every method call completes in a finite number of steps.

No mutual exclusion

- Thread could halt in critical section
- Build mutual exclusion from registers
From Safe SRSW Boolean to Atomic Snapshots
Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMWW atomic
- Atomic snapshot
Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMWW atomic
- Atomic snapshot
public class SafeBoolMRSWRegister implements Register<Boolean> {
    public boolean read() { ... }
    public void write(boolean x) { ... }
}
public class SafeBoolMRSWRegister implements Register<Boolean> {
    public boolean read() { … } 
    public void write(boolean x) { … } 
}
public class SafeBoolMRSWRegister implements Register<Boolean> {
    public boolean read() {
        ...
    }
    public void write(boolean x) {
        ...
    }
}
Register Names

```java
public class SafeBoolMRSWRegister implements Register<Boolean> {
    public boolean read() { ... }
    public void write(boolean x) { ... }
}
```

- **property**
- **type**
- **how many readers & writers?**
Safe Boolean MRSW from Safe Boolean SRSW

Let’s write 1!
Safe Boolean MRSW from Safe Boolean SRSW
Safe Boolean MRSW from Safe Boolean SRSW
Safe Boolean MRSW from Safe Boolean SRSW
Safe Boolean MRSW from Safe Boolean SRSW

Whew!

Art of Multiprocessor Programming
public class SafeBoolMRSWRegister implements Register<Boolean> {
    private SafeBoolSRSWRegister[] r = new SafeBoolSRSWRegister[N];
    public void write(boolean x) {
        for (int j = 0; j < N; j++)
            r[j].write(x);
    }
    public boolean read() {
        int i = ThreadID.get();
        return r[i].read();
    }
}
Safe Boolean MRSW from Safe Boolean SRSW

public class SafeBoolMRSWRegister implements BooleanRegister {
    private SafeBoolSRSWRegister[] r = new SafeBoolSRSWRegister[N];
    public void write(boolean x) {
        for (int j = 0; j < N; j++)
            r[j].write(x);
    }

    public boolean read() {
        int i = ThreadID.get();
        return r[i].read();
    }
}

Each thread has own safe SRSW register
Safe Boolean MRSW from Safe Boolean SRSW

```java
public class SafeBoolMRSWRegister implements BooleanRegister {
    private SafeBoolSRSWRegister[] r = new SafeBoolSRSWRegister[N];

    public void write(boolean x) {
        for (int j = 0; j < N; j++)
            r[j].write(x);
    }

    public boolean read() {
        int i = ThreadID.get();
        return r[i].read();
    }
}
```

write method
Safe Boolean MRSW from Safe Boolean SRSW

```java
public class SafeBoolMRSWRegister implements BooleanRegister {
    private SafeBoolSRSWRegister[] r = new SafeBoolSRSWRegister[N];
    public void write(boolean x) {
        for (int j = 0; j < N; j++)
            r[j].write(x);
    }
    public boolean read() {
        int i = ThreadID.get();
        return r[i].read();
    }
}
```

Write each thread’s register one at a time
Safe Boolean MRSW from Safe Boolean SRSW

```java
public class SafeBoolMRSWRegister implements BooleanRegister {
    private SafeBoolSRSWRegister[] r = new SafeBoolSRSWRegister[N];
    public void write(boolean x) {
        for (int j = 0; j < N; j++)
            r[j].write(x);
    }

    public boolean read() {
        int i = ThreadID.get();
        return r[i].read();
    }
}
```

**read method**

Art of Multiprocessor Programming
public class SafeBoolMRSWRegister implements BooleanRegister {
    private SafeBoolSRSWRegister[] r =
        new SafeBoolSRSWRegister[N];
    public void write(boolean x) {
        for (int j = 0; j < N; j++)
            r[j].write(x);
    }
    public boolean read() {
        int i = ThreadID.get();
        return r[i].read();
    }
}
Safe Multi-Valued MRSW from Safe Multi-Valued SRSW?

Yes, it works!

any value in range
Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

Questions?
Road Map

• SRSW safe Boolean
• MRSW safe Boolean
• MRSW regular Boolean
• MRSW regular
• MRSW atomic
• MRMW atomic
• Atomic snapshot
Regular Boolean MRSW from Safe Boolean MRSW
Regular Boolean MRSW

Safe Boolean MRSW

Uh, oh!
Regular Boolean MRSW from Safe Boolean MRSW
Regular Boolean MRSW from Safe Boolean MRSW

```java
public class RegBoolMRSWRegister implements Register<Boolean> {
    private boolean old;
    private SafeBoolMRSWRegister value;
    public void write(boolean x) {
        if (old != x) {
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
```
Regular Boolean MRSW from Safe Boolean MRSW

public class RegBoolMRSWRegister implements Register<Boolean> {
    threadLocal boolean old;
    private SafeBoolMRSWRegister value;
    public void write(boolean x) {
        if (old != x) {
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
Regular Boolean MRSW from Safe Boolean MRSW

```java
public class RegBoolMRSWRegister implements Register<Boolean> {
    threadLocal boolean old;
    private SafeBoolMRSWRegister value;
    public void write(boolean x) {
        if (old != x) {
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
```

Actual value
Regular Boolean MRSW from Safe Boolean MRSW

```java
public class RegBoolMRSWRegister implements Register<Boolean> {
    threadLocal boolean old;
    private SafeBoolMRSWRegister value;
    public void write(boolean x) {
        if (old != x) {
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
```

Is new value different from last value I wrote?
Regular Boolean MRSW from Safe Boolean MRSW

```java
public class RegBoolMRSWRegister implements Register<Boolean> {
    threadLocal boolean old;
    private SafeBoolMRSWRegister value;
    public void write(boolean x) {
        if (old != x) {
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
```

If so, change it (otherwise don’t!)
Regular Boolean MRSW from Safe Boolean MRSW

```java
public class RegBoolMRSWRegister implements Register<Boolean>{
    threadLocal boolean old;
    private SafeBoolMRSWRegister value;
    public void write(boolean x) {
        if (old != x) {
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
```

Overlap? What overlap? No problem either Boolean value works
Regular Multi-Valued MRSW from Safe Multi-Valued MRSW?

Safe register can return any value in range when value changes

Does not work!

Regular register can return only old or new when value changes
Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

Questions?
Road Map

• SRSW safe Boolean
• MRSW safe Boolean
• MRSW regular Boolean
• MRSW regular
• MRSW atomic
• MRMW atomic
• Atomic snapshot
Representing $m$ Values

Unary representation:
bit[$i$] means value $i$

Initially 0

1 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7
Writing $m$-Valued Register

Write 5

1 0 0 0 0

0 1 2 3 4 5 6 7
Writing $m$-Valued Register

Initially 0

Write 5

0 0 0 0 1

0 1 2 3 4 5 6 7
Writing \( m \)-Valued Register

Write 5

Art of Multiprocessor Programming
public class RegMRSWRegister implements Register{
    RegBoolMRSWRegister[M] bit;

    public void write(int x) {
        this.bit[x].write(true);
        for (int i=x-1; i>=0; i--)
            this.bit[i].write(false);
    }

    public int read() {
        for (int i=0; i < M; i++)
            if (this.bit[i].read())
                return i;
    }
}
MRSW Regular $m$-valued from MRSW Regular Boolean

```java
public class RegMRSWRegister implements Register{
    RegBoolMRSWRegister[M] bit;

    public void write(int x) {
        bit[x].write(true);
        for (int i=x-1; i>=0; i--)
            bit[i].write(false);
    }

    public int read() {
        for (int i=0; i < M; i++)
            if (bit[i].read())
                return i;
    }
}
```

Unary representation: bit[i] means value i
public class RegMRSWRegister implements Register {
    RegBoolMRSWRegister[m] bit;

    public void write(int x) {
        bit[x].write(true);
        for (int i=x-1; i>=0; i--)
            bit[i].write(false);
    }

    public int read() {
        for (int i=0; i < M; i++)
            if (bit[i].read())
                return i;
    }
}
MRSW Regular $m$-valued from MRSW Regular Boolean

```java
public class RegMRSWRegister implements Register {
    RegBoolMRSWRegister[m] bit;

    public void write(int x) {
        bit[x].write(true);
        for (int i=x-1; i>=0; i--)
            bit[i].write(false);
    }

    public int read() {
        for (int i=0; i < M; i++)
            if (bit[i].read())
                return i;
    }
}
```

Clear bits from higher to lower
public class RegMRSWRegister implements Register {
    RegBoolMRSWRegister[m] bit;

    public void write(int x) {
        bit[x].write(true);
        for (int i=x-1; i>=0; i--)
            bit[i].write(false);
    }

    public int read() {
        for (int i=0; i < M; i++)
            if (bit[i].read())
                return i;
    }
}
Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

Questions?
Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot
Road Map (Slight Detour)

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot
Concurrent Reading

Regular writer

5678

Concurrent Reading

1234

Instead of 5678…

When is this a problem?

SRSW Atomic From SRSW Regular
SRSW Atomic From SRSW Regular

Initially 1234

Regular writer

Regular reader

Same as Atomic

write(5678)
SRSW Atomic From SRSW
Regular

Regular writer

Initially
1234

write(5678)

read(1234)

Same as Atomic

Regular reader

Instead of 5678…

time

Art of Multiprocessor Programming
SRSW Atomic From SRSW Regular

Initially 1234

Regular writer

Regular reader

write(5678)

read(1234)

not Atomic!

Write 5678 happened
Timestamped Values

Writer writes value and stamp together

Reader saves last value & stamp read returns new value only if stamp is higher
SRSW Atomic From SRSW Regular

writer

reader

Same as Atomic

write(2:00 5678)

read(2:00 5678)

read(1:45 1234)

write(1:45 1234)

read(2:00 5678)

read(1:45 1234)

Art of Multiprocessor Programming
Atomic Single-Reader to Atomic Multi-Reader

<table>
<thead>
<tr>
<th>stamp</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
</tbody>
</table>

One per reader
Another Scenario

Writer starts write…

<table>
<thead>
<tr>
<th>stamp</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:00</td>
<td>5678</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
</tbody>
</table>
Another Scenario

Yellow was completely after Blue but read earlier value…not linearizable!
## Multi-Reader Redux

Each thread reads one value:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:45 1234</td>
<td>1:45 1234</td>
<td>1:45 1234</td>
</tr>
<tr>
<td>1:45 1234</td>
<td>1:45 1234</td>
<td>1:45 1234</td>
</tr>
<tr>
<td>1:45 1234</td>
<td>1:45 1234</td>
<td>1:45 1234</td>
</tr>
</tbody>
</table>

---

Art of Multiprocessor Programming 90
Writer writes column...

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:45</td>
<td>1234</td>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>2:00</td>
<td>5678</td>
<td>1234</td>
<td>1234</td>
</tr>
<tr>
<td>2:00</td>
<td>5678</td>
<td>1234</td>
<td>1234</td>
</tr>
<tr>
<td>2:00</td>
<td>5678</td>
<td>1234</td>
<td>1234</td>
</tr>
</tbody>
</table>

reader reads row
zzz...after second write

reader writes column to notify others of what it read

Yellow reader will read new value in column written by earlier Blue reader

Art of Multiprocessor Programming
Can’t Yellow Miss Blue’s Update?  
… Only if Readers Overlap…

In which case it’s OK to read 1234
Bad Case Only When Readers Don’t Overlap

In which case Blue will complete writing 2:00 5678 to its column.
Road Map

• SRSW safe Boolean
• MRSW safe Boolean
• MRSW regular Boolean
• MRSW regular
• MRSW atomic
• MRMW atomic
• Atomic snapshot
Multi-Writer Atomic From Multi-Reader Atomic

Each writer reads all then writes Max+1 to its register

Readers read all and take max (Lexicographic like Bakery)

Max is 2:15, return XYZW

Art of Multiprocessor Programming
Atomic Execution
Means it is Linearizable
Linearization Points

write(1)  Read(max = 2)  write(4)  
write(2)  write(3)  Read(max = 3)  
Read (max = 1)  write(2)  Read(max = 4)  

Time

Art of Multiprocessor Programming
Linearization Points

Look at Writes First

write(1) → write(2) → write(2) → write(2) → write(3) → write(4)

Art of Multiprocessor Programming
Linearization Points

Order writes by TimeStamp

write(1)
write(2)
write(3)
write(4)
Linearization Points

Order reads by max stamp read

write(1)
write(2)
write(3)
write(4)
Read(max = 1)
Read(max = 2)
Read(max = 3)
Read(max = 4)

Art of Multiprocessor Programming
Linearization Points

Order reads by max stamp read

write(1)
write(4)
write(2)
write(3)
write(2)
Read(max = 2)
Write
Read(max = 3)
Read(max = 1)
Read(max = 4)

Art of Multiprocessor Programming
The linearization point depends on the execution (not a line in the code)!
Road Map

• SRSW safe Boolean
• MRSW safe Boolean
• MRSW regular Boolean
• MRSW regular
• MRSW atomic
• MRMW atomic
• Atomic snapshot

Questions?
Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot
Atomic Snapshot

update

scan
Atomic Snapshot

- Array of SWMR atomic registers
- Take instantaneous snapshot of all
- Generalizes to MRMW registers …
public interface Snapshot {
    public int update(int v);
    public int[] scan();
}
Snapshot Interface

Thread \( i \) writes \( v \) to its register

```java
public interface Snapshot {
    public int update(int v);
    public int[] scan();
}
```
Snapshot Interface

Instantaneous snapshot of all thead's registers

```java
public interface Snapshot {
    public int update(int v);
    public int[] scan();
}
```
Atomic Snapshot

• **Collect**
  – Read values one at a time

• **Problem**
  – Incompatible concurrent collects
  – Result not linearizable
Clean Collects

- **Clean Collect**
  - Collect during which nothing changed
  - Can we make it happen?
  - Can we detect it?
Simple Snapshot

- Put increasing labels on each entry
- Collect twice
- If both agree, we're done
- Otherwise, try again

Problem: Scanner might not be collecting a snapshot!

- We're done

- Otherwise, try again
Claim: We Must Use Labels

But scanner sees \( x \) and \( z \) together!

\( x \) and \( z \) are never in memory together.
Scanner reads x and z with different labels and recognizes collect not clean.
Simple Snapshot

• Collect twice
• If both agree,
  – We’re done
• Otherwise,
  – Try again
public class SimpleSnapshot implements Snapshot {
    private AtomicMRSWRegister[] register;

    public void update(int value) {
        int i = Thread.myIndex();
        LabeledValue oldValue = register[i].read();
        LabeledValue newValue =
            new LabeledValue(oldValue.label+1, value);
        register[i].write(newValue);
    }
}
Simple Snapshot: Update

```java
public class SimpleSnapshot implements Snapshot {
    private AtomicMRSWRegister[] register;

    public void update(int value) {
        int i = Thread.myIndex();
        LabeledValue oldValue = register[i].read();
        LabeledValue newValue =
            new LabeledValue(oldValue.label+1, value);
        register[i].write(newValue);
    }
}
```

One single-writer register per thread
public class SimpleSnapshot implements Snapshot {
    private AtomicMRSWRegister[] register;

    public void update(int value) {
        int i = Thread.myIndex();
        LabeledValue oldValue = register[i].read();
        LabeledValue newValue =
            new LabeledValue(oldValue.label + 1, value);
        register[i].write(newValue);
    }

    Write each time with higher label
private LabeledValue[] collect() {
    LabeledValue[] copy =
        new LabeledValue[n];
    for (int j = 0; j < n; j++)
        copy[j] = this.register[j].read();
    return copy;
}
private LabeledValue[] collect() {
    LabeledValue[] copy = new LabeledValue[n];
    for (int j = 0; j < n; j++)
        copy[j] = this.register[j].read();
    return copy;
}
public int[] scan() {
    LabeledValue[] oldCopy, newCopy;
    oldCopy = collect();
    collect: while (true) {
        newCopy = collect();
        if (!equals(oldCopy, newCopy)) {
            oldCopy = newCopy;
            continue collect;
        }
    }
    return getValues(newCopy);
}
public int[] scan() {
    LabeledValue[] oldCopy, newCopy;
    
    oldCopy = collect();
    collect: while (true) {
        newCopy = collect();
        if (!equals(oldCopy, newCopy)) {
            oldCopy = newCopy;
            continue collect;
        }
    }
    return getValues(newCopy);
}
public int[] scan() {
  LabeledValue[] oldCopy, newCopy;
  oldCopy = collect();
  collect: while (true) {
    newCopy = collect();
    if (!equals(oldCopy, newCopy)) {
      oldCopy = newCopy;
      continue collect;
    }
  }
  return getValues(newCopy);
}
Simple Snapshot: Scan

```java
public int[] scan() {
    LabeledValue[] oldCopy, newCopy;
    oldCopy = collect();
    collect: while (true) {
        newCopy = collect();
        if (!equals(oldCopy, newCopy)) {
            oldCopy = newCopy;
            continue collect;
        }
        return getValues(newCopy);
    }
}
```

**Collect once**

**Collect twice**

**On mismatch, try again**
public int[] scan() {
    LabeledValue[] oldCopy, newCopy;
    oldCopy = collect();
    collect: while (true) {
        newCopy = collect();
        if (!equals(oldCopy, newCopy)) {
            oldCopy = newCopy;
            continue collect;
        }
        return getValues(newCopy);
    }
}
Simple Snapshot

• Linearizable
• Update is wait-free
  – No unbounded loops
• But Scan can starve
  – If interrupted by concurrent update
Wait-Free Snapshot

• Add a scan before every update
• Write resulting snapshot together with update value
• If scan is continuously interrupted by updates, scan can take the update’s snapshot
Wait-free Snapshot

If A’s scan observes that B moved \textit{twice}, then B completed an update while A’s scan was in progress.
Wait-free Snapshot
Wait-free Snapshot

Art of Multiprocessor Programming
Wait-free Snapshot

B’s 1\textsuperscript{st} update must have written during 1\textsuperscript{st} collect

So scan of B’s second update must be within interval of A’s scan

So A can steal result of B’s scan
Wait-free Snapshot

But no guarantee that scan of B’s 1st update can be used…
Why?

Art of Multiprocessor Programming
Once is not Enough

Why can’t A steal B’s scan?

Because another update might have interfered before the scan.
Someone Must Move Twice

If we collect $n$ times…some thread must move twice (pigeonhole principle)
Scan is Wait-free

At most n-1 depth

So some thread must have had clean collect
Wait-Free Snapshot Label

```java
public class SnapValue {
    public int label;
    public int value;
    public int[] snap;
}
```
Wait-Free Snapshot Label

public class SnapValue {
    public int label;
    public int value;
    public int[] snap;
}

Counter incremented with each snapshot
public class SnapValue {
    public int label;
    public int value;
    public int[] snap;
}
public class SnapValue {
    public int label;
    public int value;
    public int[] snap;
}

most recent snapshot
Wait-Free Snapshot Label

11011110101000101100...00

label

value

Last snapshot
public void update(int value) {
    int i = Thread.myIndex();
    int[] snap = this.scan();
    SnapValue oldValue = r[i].read();
    SnapValue newValue =
        new SnapValue(oldValue.label+1,
                      value, snap);
    r[i].write(newValue);
}
public void update(int value) {
  int i = Thread.myIndex();
  int[] snap = this.scan();
  SnapValue oldValue = r[i].read();
  SnapValue newValue =
      new SnapValue(oldValue.label+1,
                    value, snap);
  r[i].write(newValue);
}
public void update(int value) {
    int i = Thread.myIndex();
    int[] snap = this.scan();
    SnapValue oldValue = r[i].read();
    SnapValue newValue =
      new SnapValue(oldValue.label+1, value, snap);
    r[i].write(newValue);
}

Wait-free Scan

Take scan

Label value with scan
public int[] scan() {
    SnapValue[] oldCopy, newCopy;
    boolean[] moved = new boolean[n];
    oldCopy = collect();
    collect: while (true) {
        newCopy = collect();
        for (int j = 0; j < n; j++) {
            if (oldCopy[j].label != newCopy[j].label) {
                ...
            }
        }
        return getValues(newCopy);
    }
}
public int[] scan() {
    SnapValue[] oldCopy, newCopy;
    boolean[] moved = new boolean[n];
    oldCopy = collect();
    collect: while (true) {
        newCopy = collect();
        for (int j = 0; j < n; j++) {
            if (oldCopy[j].label != newCopy[j].label) {
                ...}
        }
    }
    return getValues(newCopy);
}}

Keep track of who moved
public int[] scan() {
    SnapValue[] oldCopy, newCopy;
    boolean[] moved = new boolean[n];

    oldCopy = collect();
    collect: while (true) {
        newCopy = collect();
        for (int j = 0; j < n; j++) {
            if (oldCopy[j].label != newCopy[j].label) {
                ...
            }
        }
        return getValues(newCopy);
    }
}}
public int[] scan() {
    SnapValue[] oldCopy, newCopy;
    boolean[] moved = new boolean[n];
    oldCopy = collect();
    collect: while (true) {
        newCopy = collect();
        for (int j = 0; j < n; j++) {
            if (oldCopy[j].label != newCopy[j].label) {
                ...
            }
        }
        return getValues(newCopy);
    }
}
if (oldCopy[j].label != newCopy[j].label) {
    if (moved[j]) { // second move
        return newCopy[j].snap;
    } else {
        moved[j] = true;
        oldCopy = newCopy;
        continue collect;
    }
}]
return getValues(newCopy);
}}

Mismatch Detected

Art of Multiprocessor Programming
Mismatch Detected

```java
if (oldCopy[j].label != newCopy[j].label) {
    if (moved[j]) {
        return newCopy[j].snap;
    } else {
        moved[j] = true;
        oldCopy = newCopy;
        continue collect;
    }
}
return getValues(newCopy);
```
if (oldCopy[j].label != newCopy[j].label) {
    if (moved[j]) { // second move
        return newCopy[j].snap;
    } else {
        moved[j] = true;
        oldCopy = newCopy;
        continue collect;
    }
}}

return getValues(newCopy);
}}
Observations

• Uses unbounded counters
  – can be replaced with 2 bits
• Assumes SWMR registers
  – for labels
  – can be extended to MRMW
Summary

• We saw we could implement MRMW multivalued snapshot objects
• From SRSW binary safe registers (simple flipflops)
• But what is the next step to attempt with read-write registers?
Grand Challenge

• **Snapshot means**
  – Write any one array element
  – Read multiple array elements
Grand Challenge

What about atomic writes to multiple locations?

Write many and snapshot

Write to 0 and 1

Write to 1 and 2

Art of Multiprocessor Programming

155
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.