Foundations of Shared Memory

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
Adapted by Larry Wittie for CSE391 S14

Art of Multiprocessor Programming Ch04 CSE391
Multicore Programming S14  2/25,27; 3/4/14
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
Why Hardware R/W Memory Not Linearizable

<table>
<thead>
<tr>
<th>Memory Hierarchy</th>
<th>Memory Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>193</td>
<td>194</td>
</tr>
<tr>
<td>• On modern multiprocessors, processors do not read and write directly to memory.</td>
<td>• To read a memory location,</td>
</tr>
<tr>
<td>• Memory accesses are very slow compared to processor speeds,</td>
<td>– load data into cache.</td>
</tr>
<tr>
<td>• Instead, each processor reads and writes directly to a cache</td>
<td>• To write a memory location</td>
</tr>
<tr>
<td></td>
<td>– update cached copy,</td>
</tr>
<tr>
<td></td>
<td>– lazily write cached data back to memory</td>
</tr>
</tbody>
</table>

Slides from Ch03 CSE391 Multicore Programming S14 2/11, 18, 20/14

<table>
<thead>
<tr>
<th>Real-World Hardware Memory</th>
<th>Linearizability</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>201</td>
</tr>
<tr>
<td>• Weaker than sequential consistency</td>
<td>• Linearizability</td>
</tr>
<tr>
<td>• But you can get sequential consistency at a price</td>
<td>– Operation takes effect instantaneously between invocation and response</td>
</tr>
<tr>
<td>• OK for expert, tricky stuff</td>
<td>– Uses sequential specification, locality implies composability</td>
</tr>
<tr>
<td>– assembly language, device drivers, etc.</td>
<td>– Good for high level objects</td>
</tr>
<tr>
<td>• Linearizability more appropriate for high-level software</td>
<td></td>
</tr>
</tbody>
</table>
Fundamentals

• What is the weakest form of communication that supports mutual exclusion?

• What is the weakest shared object that allows shared-memory computation?
  – What is the weakest kind of communication or synchronization that allows useful work?
Alan Turing

- Showed what is and is not computable on a sequential machine.
- Provided what is still the best model there is.
Turing Computability

- Mathematical model of computation
- What is (and is not) computable
- Efficiency (mostly) irrelevant
Shared-Memory Computability?

- Mathematical model of concurrent computation
- What is (and is not) concurrently computable
- Efficiency (mostly) irrelevant
To understand modern multiprocessors we need to ask some basic questions …

A shared-memory computation consists of multiple threads, each a sequential program. Threads communicate by calling methods of objects that reside in a shared memory. Threads are asynchronous; they run at different speeds and any thread can halt for an unpredictable duration at any time. The notion of asynchrony reflects the realities of multicore architectures; thread delays are unpredictable, ranging from microseconds (cache misses), to milliseconds (page faults), to seconds (scheduling interruptions).
Foundations of Shared Memory

To understand modern multiprocessors, we need to ask some basic questions. What is the weakest useful form of shared memory?

The first question to ask is what is the **weakest useful form** of shared memory. **Not** because we want to save money by buying the least amount of machinery, like a standard transmission and manual windows on an automobile, but rather to figure out what is essential to perform computations and from there what is useful to enhance efficiency, convenience, and other desirable attributes.
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?

After identifying the weakest useful model of concurrent computation, we next can ask how far can we go with the model? For example, in sequential computability, we can decide that finite state machines are useful, and we can use them to parse arithmetic expressions, control traffic lights, and do other simple tasks.
Register*

Holds a (binary) value

* A memory location: the name is historical
Register

Can be read

10011
Register

Can be written

01100

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

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

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

01100 10011

10011

01100
Multi-Reader/Single-Writer Register
Multi-Reader/Multi-Writer Register

Example diagram showing the concept of a Multi-Reader/Multi-Writer Register.
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

Any 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) write(1) read(1) read(0)
Regular or Not?

Overlap: returns new value
Regular or Not?

Overlap: returns old value
Regular or Not?

Both reads, which return one and zero, overlap the write that is writing 1, so the history is regular. The way to think about this is that while the write of 1 is going on, the value is flickering between 0 and 1, and each read samples the value at some instant while the write from 0 to 1 is running.
Regular ≠ Linearizable

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

write(1) already happened

can’t explain this!
Atomic Register

Linearizable to sequential safe register
Atomic Register

write(1001) → read(1001) → write(1010) → read(1010) → read(1010)
For all correctness conditions, a read returns the most recently completed write value if the read does not overlap any new write(s).

If read overlaps write(s), returns:
- Any valid value;
- Old or any new value;
- Most recent value in a linearized sequence.
Weakest Register

Single writer

Single reader

Safe Boolean register
Weakest Register

Single writer

Single reader

Get correct reading if not during state transition

flipflop
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!
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
- MRMW atomic
- Atomic snapshot
Road Map

• SRSW safe Boolean
• MRSW safe Boolean
• MR SW regular Boolean
• MR SW regular
• MR SWW atomic
• MRMW 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) { ... }
}
public class SafeBoolMRSWRegister implements Register<Boolean> {
    public boolean read() { … }
    public void write(boolean x) { … }
}
Safe Boolean MRSW from Safe Boolean SRSW
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!
Safe Boolean MRSW from Safe Boolean SRSW

```java
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

```java
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 its 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

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 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();
    }
}
```
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 my own register
Safe Multi-Valued MRSW from Safe Multi-Valued SRSW?

Yes, it works!

any value in range
Road Map

• SR SW safe Boolean
• MR SW safe Boolean
• MR SW regular Boolean
• MR SW regular
• MR SW atomic
• MR MW atomic
• Atomic snapshot

Questions?
Road Map

- SRSW safe Boolean
- MR SW safe Boolean
- MR SW regular Boolean
- MR SW regular
- MR SW atomic
- MRMW atomic
- Atomic snapshot
Regular Boolean MRSW from Safe Boolean MRSW
Regular Boolean MRSW from Safe Boolean MRSW

Uh, oh!
Regular Boolean MRSW from Safe Boolean MRSW

Last written: 0

Art of Multiprocessor Programming Ch04 CSE391
Multicore Programming S14  2/25,27; 3/4/14
public class RegBoolMRSWRegister implements Register<Boolean> {
    private boolean old; //OK for 1 writer thread
    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();
    }
}
```

Last bit this thread wrote

(made-up syntax) MRMW needs; >1 writer threads
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.

Regular register can return only old or new when value changes.
Regular Multi-Valued MRSW from Safe Multi-Valued MRSW?

Safe register can return any value in range when value changes.

Regular register can return only old or new when value changes.
Regular Multi-Valued MRSW from Safe Multi-Valued MRSW?

Safe register can return any value in range when value changes

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
- MRW 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
Writing $m$-Valued Register

Write 5

\[ 100000 \]

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

Initially 0

0 0 0 0 0 1

0 1 2 3 4 5 6

Write 5
Writing $m$-Valued Register

Write 5

0 1 2 3 4 5 6

0 0 0 0 0 1

0
MRSW Regular $m$-valued from MRSW Regular Boolean

```java
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;
    }
}
```
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;
    }
}
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
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;
    }
}
```

Scan from lower to higher & return value as index for first true bit
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

Art of Multiprocessor Programming Ch04 CSE391
Multicore Programming S14  2/25,27; 3/4/14
Concurrent Reading

SRSW Atomic From SRSW Regular

Regular writer

Concurrent Reading

Regular reader

When is this a problem?

Instead of 5678…
SRSW Atomic From SRSW Regular

Initially 1234

Regular writer

Regular reader

write(5678)

read(5678)

Same as Atomic

time
SRSW Atomic From SRSW Regular

Regular writer

Regular reader

Initially 1234

write(5678)

read(1234)

Same as Atomic

Instead of 5678...

time
SRSW Atomic From SRSW Regular

Initially\n\text{1234}\n
\text{Reg read(5678)} \quad \text{Reg read(5678)} \quad \text{Reg read(5678)}

\text{write(5678)} \quad \text{write(5678)} \quad \text{write(5678)}

\text{read(1234)} \quad \text{read(1234)}\quad \text{read(1234)}

\text{Write 5678 happened}

\text{not Atomic!}
Timestamped Values

Writer writes each 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

1:45 1234 < 2:00 5678
So stick with 5678

Same as Atomic

write(2:00 5678)
read(2:00 5678)
read(1:45 1234)

Art of Multiprocessor Programming Ch04 CSE391
Multicore Programming S14 2/25,27; 3/4/14
Atomic Single-Reader to Atomic Multi-Reader – First Try (Bad)

<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 register per each of 4 readers
Another Scenario, About to Fail

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, Failed

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

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

one per thread
Writer writes column...

Reader reads row

2:00, 5678
reader writes column to notify others of what it read

zzz…after second write

Yellow reader will read new value in column written by earlier Blue reader
Can’t Yellow Miss Blue’s Update? … Only if Readers Overlap…

Prior
1:45 1234

write(2:00 5678)

read(2:00 5678)

read(1:45 1234)

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

Prior
1:45 1234

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

Questions?
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 Single-Writer Atomic

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

<table>
<thead>
<tr>
<th>time stamp</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>2:00</td>
<td>5678</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>2:15</td>
<td>XYZW</td>
</tr>
</tbody>
</table>

Readers read all and take max time (Lexicographic like Bakery).

Max is 2:15, return XYZW.
Atomic Execution
Means it is Linearizable

Numbers in call arrows are time stamps.

Writer IDs
0  write(1)  Read(max = 2)  write(4)
1  write(2)  write(3)  Read(max = 3)
2  Read (max = 1)  write(2)  Read(max = 4)

Art of Multiprocessor Programming Ch04 CSE391
Multicore Programming S14 2/25,27; 3/4/14
Linearization Points

Numbers in call arrows are time stamps.

Writer IDs

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

time
Look at Writes First

Numbers in call arrows are time stamps. Two write(2)s saw max=1, so ID=1 is 1\textsuperscript{st}.
Linearization Points

Order writes by TimeStamp

Writer IDs
0
1
2

write(0)
write(2)
write(2)
write(1)
write(4)
write(3)
Linearization Points

Order reads by max stamp read

Writer IDs

0
write(1)  Read(max= 2)  write(2)

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

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

time
Linearization Points

Order reads by max stamp read

Writer IDs

0
write(1) Read(max = 2) write(4)

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

2
Read(max = 1) write(2) Read(max = 4)
Linearization Points

The linearization point depends on the execution (not a line in the code)!

Writer IDs
0  write(1)  read(max= 2)  write(3)  read(max = 3)
1  write(2)  write(3)  read(max = 3)
2  read (max = 1)  write(2)  read(max = 4)
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 MRSW 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 theads’ 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 scan 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

- We’re done

• Otherwise,
  - Try again

Problem: Scanner might not be collecting a snapshot!
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 in labels and values,
  - We’re done
- Otherwise,
  - Try again
Simple Snapshot: Update
{change one register[i] value & label}

```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);
    }
}
```
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
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);
    }
}
```

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;
}
Simple Snapshot: Scan
{collect new copies of all regs until two equal}

```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);
    }
}
```
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);
    }
}
```
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);
    }
}
```
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 match, return values
Simple Snapshot

• Linearizable
• Update is wait-free
  – No unbounded loops
• But Scan can starve
  – If overlapped repeatedly by other threads’ concurrent updates
Wait-Free Snapshot

• Add a scan before every update
• Write resulting snapshot together with update value and label
• If scan is continuously interrupted by updates, scan can eventually take an update’s snapshot
• A thread “moves [on]” when it completes an update() including its own clean scan().
Wait-free Snapshot

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

A `scan()`   Collect ≠ Collect ≠ Collect

B

26 24 12 ≠ 26 12 ≠ 26 12

Update

time
Wait-free Snapshot

A scan()

Collect ≠ Collect ≠ Collect

26 24 12

B

Scan
Write
Update

time

Art of Multiprocessor Programming Ch04 CSE391 Multicore Programming S14 2/25,27; 3/4/14
Wait-free Snapshot

B’s 1st update must have written during 1st collect

A scan()

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?

A  scan()

Collect

B  Write

Scan

Write

26
24
12

26
12

26
12

time
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).

{A thread “moves [on]” when it completes an update() including a clean scan() of its own.}
Scan is Wait-free

One of the n-1 threads (excluding A) must have a clean collect before an nth scan in an update to A’s scan. A can take 2nd copy in the 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
Wait-Free Snapshot Label

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

Actual value
Wait-Free Snapshot Label

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

most recent snapshot
Wait-Free Snapshot Label

11011110101000101100...00

label

101000101100...00

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);
}
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);
    }
}
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);
    }}

Repeated double collect
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) {
                ... //See next slide
            }
        }
        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);
}}
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 MRSW registers
  – for labels
  – can be extended to MRMW
Summary

• We saw we could implement MRMW multi valued 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
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.