Foundations of Shared Memory

Adapted from the Companion slides for The Art of Multiprocessor Programming by Maurice Herlihy & Nir Shavit
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
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 …
Foundations of Shared Memory

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

* 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);
}
Registers

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

Type of register
(usually Boolean or m-bit Integer)
Registers

If method calls do not overlap:

```java
public class SequentialRegister<T> implements Register<T> {
    private T value;
    public T read() {
        return value;
    }
    public void write(T v) {
        value = v;
    }
}
```
Weakest Register

Single writer

Single reader

Safe Boolean register
Weakest Register

Single writer

Single reader

Get correct reading if not during state transition

Art of Multiprocessor Programming
Registers

If method calls overlap:

```java
public class SequentialRegister<T>
    implements Register<T>
{
    private T value;
    public T read()
    { return value; }
    public void write(T v)
    { value = v; }
}
```
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 \textit{wait-free} if every method call completes in a finite number of steps.

\textbf{No mutual exclusion}

- Thread could halt in critical section
- Build mutual exclusion from registers
Registers

To resolve overlap:

- Mutex lock
  - Registers are primitive
  - We cannot use mutual exclusion to describe registers.
- Even if deadlock or starvation free, depends on OS scheduler to guarantee that thread never get stuck in CS.
The Space of Registers

- Recall, two key properties of shared objects:
  - Safety – consistency conditions
  - Liveness – progress conditions.
- Registers must be wait-free (object is wait-free if each method finishes in a finite number of steps)
- Wait-free rules out Mutex and guarantees independent progress. We desire these in our constructs.
The Space of Registers

Different registers provide different guarantees. The characteristics that differentiate registers include:

• Values supported (boolean, M-valued integers)
• Numbers of readers/writers
• Type of consistency provided
public interface Register\<T\> {
    public T read();
    public void write(T v);
}

Range of value they may encapsulate
(usually Boolean or \(m\)-bit Integer)
Single-Reader/Single-Writer Register

Number of readers/writers

01100

10011

10011

01100
Multi-Reader/Single-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)

Degree of consistency provided
Safe Register

Some valid value if reads and writes do overlap
A register implementation is *safe* if:

- a read() call that does not overlap a write() call returns the value written by the most recent write(),
- otherwise the read() will return any value in the allowed range e.g 0 to (M-1) in a M-valued register.
Safe Register

What is $R_1()$?
Safe Register

What is $R_1()$ and $R_2()$?
Safe Register

What is \( R^1() \), \( R^2() \), and \( R^3() \)?
A register is *regular* if:

- a read() call that does not overlap a write() call returns the value written by the most recent write(),
- otherwise, let $v_0$ be the value written by the latest preceding write() and $v_1...v_k$ be those written by the overlapping write() calls, read() may return any $v_i$ for $i$ in $0...k$
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?

write(0) → write(1) → read(0)

Overlap: returns old value
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!
What is R1()?
Regular Register

What is R1() and R2()?
What is $R^1()$, $R^2()$, and $R^3()$?
Atomic Register

A register is *atomic* if:

- a read() call that does not overlap a write() call returns the value written by the most recent write(),
- otherwise, the value returned by a read() call that overlaps a write() call is either the value written by the overlapping write() call or the last value returned by a read() call.
Atomic Register

Linearizable to sequential safe register

```java
public class SequentialRegister<T> implements Register<T> {
    private T value;
    public T read() {
        return value;
    }
    public void write(T v) {
        value = v;
    }
}
```
Atomic Register

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

Art of Multiprocessor Programming
Atomic Register

What is $R^1()$?
Atomic Register

What is $R^1()$ and $R^2()$?
Atomic Register

What is $R_1()$, $R_2()$, and $R_3()$?
Register Space

- MRMW
- MRSW
- SRSW

Arrows:
- Safe
- Regular
- Atomic
- m-valued
- Boolean
Object Histories

Recall that methods are determined by their invocations and responses:

- if $m_0$ and $m_1$ are method calls then $m_0 \rightarrow m_1$ if $m_0$’s response preceded $m_1$’s invocation.
Formalizing Registers

• Any register implementation define a total order on the write calls or their effect.
• For safe and regular registers, write order is trivial because there’s only one writer at a time.
• For atomic registers, method calls have a linearization order.
• Notation \( W^i, R^i, V^i \) (Single write but possibly many reads)
Regular Registers

The following conditions precisely define a regular register:

• it is never the case that $R^i \rightarrow W^i$ (no read calls return a value from the future)
• it is never the case that for some $j$
  $W^i \rightarrow W^j \rightarrow R^i$ (no read call returns a value from the distant past)
Atomic Registers

An atomic register must satisfy one more condition:
• If $R^i \rightarrow R^j$ then $i \leq j$

To show that a register implementation is atomic, we need first to define a write order and then show that it’s histories satisfy these conditions.
From Safe SRSW Boolean to Atomic Snapshots

- MRMW
- MRSW
- SRSW

- Safe
- Regular
- Atomic
- M-valued
- Boolean
- Snapshot
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
- MRSW regular Boolean
- MRSW regular
- MRSW 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**

Art of Multiprocessor Programming
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!
public class SafeBooleanMRSWRegister implements Register<Boolean> {
    boolean[] s_table;
    public SafeBooleanMRSWRegister(int capacity) {
        s_table = new boolean[capacity];
    }
    public Boolean read() {
        return s_table[ThreadID.get()];
    }
    public void write(Boolean x) {
        for (int i = 0; i < s_table.length; i++)
            s_table[i] = x;
    }
}
Safe Boolean MRSW from Safe Boolean SRSW

```java
public class SafeBooleanMRSWRegister implements Register<Boolean> {
    boolean[] s_table;
    public SafeBooleanMRSWRegister(int capacity) {
        s_table = new boolean[capacity];
    }
    public Boolean read() {
        return s_table[ThreadID.get()];
    }
    public void write(Boolean x) {
        for (int i = 0; i < s_table.length; i++)
            s_table[i] = x;
    }
}
```

Each thread has own safe SRSW register
public class SafeBooleanMRSWRegister implements Register<Boolean> {
    boolean[] s_table;
    public SafeBooleanMRSWRegister(int capacity) {
        s_table = new boolean[capacity];
    }
    public Boolean read() {
        return s_table[ThreadID.get()];
    }
    public void write(Boolean x) {
        for (int i = 0; i < s_table.length; i++)
            s_table[i] = x;
    }
}
Safe Boolean MRSW from Safe Boolean SRSW

```java
public class SafeBooleanMRSWRegister implements Register<Boolean> {
    boolean[] s_table;
    public SafeBooleanMRSWRegister(int capacity) {
        s_table = new boolean[capacity];
    }
}

public Boolean read() {
    return s_table[ThreadID.get()];
}

public void write(Boolean x) {
    for (int i = 0; i < s_table.length; i++)
        s_table[i] = x;
}
```

Write each thread’s register one at a time.
public class SafeBooleanMRSWRegister implements Register<Boolean> {
    boolean[] s_table;
    public SafeBooleanMRSWRegister(int capacity) {
        s_table = new boolean[capacity];
    }
}

public Boolean read() {
    return s_table[ThreadId.get()];
}

public void write(Boolean x) {
    for (int i = 0; i < s_table.length; i++)
        s_table[i] = x;
}
public class SafeBooleanMRSWRegister implements Register<Boolean> {
    boolean[] s_table;
    public SafeBooleanMRSWRegister(int capacity) {
        s_table = new boolean[capacity];
    }
    public Boolean read() {
        return s_table[ThreadID.get()];
    }
    public void write(Boolean x) {
        for (int i = 0; i < s_table.length; i++)
            s_table[i] = x;
    }
}

Read my own register
Safe Boolean MRSW from Safe Boolean SRSW

Proof:

• If A’s read() call does not overlap any write() call, then the read() call returns the value of $s_{table}[A]$, which is the most recently written value.
• For overlapping method calls, the reader may return any value because the component registers are safe.
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 from Safe Boolean MRSW

Uh, oh!
Regular Boolean MRSW from Safe Boolean MRSW

Art of Multiprocessor Programming
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

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

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 Boolean MRSW from Safe Boolean MRSW

**Proof:**

- A read() call that does not overlap any write() call returns the most recently written value.
- If the calls do overlap, there are two cases to consider:
Regular Boolean MRSW from Safe Boolean MRSW

- If the value being written is the same as the last value written, then the writer avoids writing to the safe register, ensuring that the reader reads the correct value.
- If the value written now is distinct from the last value written, then those values must be *true* and *false* because the register is Boolean. A concurrent read returns some value in the range of the register, namely either *true* or *false*, either of which is correct.
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

Does not work!
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
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

0 0 0 0 1

0 1 2 3 4 5 6 7

Write 5
Writing $m$-Valued Register

Write 5
Writing $m$-Valued Register

Write 5

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

Write 5

0 0 0 0 1 0 1
0 1 2 3 4 5 6 7
Writing $m$-Valued Register
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;
    }
}
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;
    }
}
```
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;
    }
}
Lemma 4.2.3: The read() call always returns a value corresponding to a bit in 0..M - 1 set by some write call.

Proof: The following property invariant: if a reading thread is reading bit[j] then some bit at j or higher is true.
MRSW Regular $m$-valued from MRSW Regular Boolean

Proof:

• For any read, let $x$ be the value written by the most recent non-overlapping write().
• At the time the write() completed, bit[$x$] was set to true, and bit[$i$] is false for $i < x$. 
MRSW Regular $m$-valued from MRSW Regular Boolean

Proof:

- By Lemma 4.2.3, if the reader returns a value that is not $x$, then it observed some bit[$j$], $j \neq x$, to be true and that bit must have been set by a concurrent write(), proving conditions 4.1.1 and 4.1.2.
Regular Registers

The following conditions precisely define a regular register:

• it is never the case that $R_i \rightarrow W_i$ (no read calls return a value from the future)

• it is never the case that for some $j$
  $W_i \rightarrow W_j \rightarrow R_i$ (no read call returns a value from the distant past)
Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMWW 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
SRSW Atomic From SRSW Regular

Recall that a regular register must satisfy the following, two conditions:

1. it is never the case that $R^i \rightarrow W^i$
2. it is never the case that for some $j$ $W^i \rightarrow W^j \rightarrow R^i$

An atomic register must satisfy a single, additional condition:

1. if $R^i \rightarrow R^j$ then $i \leq j$
To construct a SRSW atomic register, we will require:

1. a regular register
2. to satisfy the condition for atomic registers
SRSW Atomic From SRSW Regular

Regular writer

Concurrent Reading

Regular reader

Instead of 5678…

When is this a problem?
SRSW Atomic From SRSW
Regular

Regular writer

Initially
1234

write(5678)

Regular reader

Same as Atomic

write(5678)

5678

5678
SRSW Atomic From SRSW
Regular

Regular writer

Regular reader

Initially
1234

write(5678)
read(1234)

Same as
Atomic

Art of Multiprocessor Programming
Initially 1234

Regular writer

Regular reader

Write 5678 happened

not Atomic!

Art of Multiprocessor Programming
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

1:45 1234
read(2:00 5678)

2:00 5678

write(2:00 5678)

read(1:45 1234)

reader

Same as Atomic

1:50 5678

Art of Multiprocessor Programming
122
public class StampedValue<T> {
    public long stamp;
    public T value;
}
public static StampedValue max(
    StampedValue x, StampedValue y)
{
    if (x.stamp > y.stamp)
        return x;
    else
        return y;
}

public static StampedValue MIN.VALUE =
    new StampedValue(null);
public void write(T v) {
    long stamp = lastStamp.get() + 1;
    r_value = new StampedValue(stamp, v);
    lastStamp.set(stamp);
}
public T read()
{
    StampedValue<T> value = r_value;
    StampedValue<T> last = lastRead.get();
    StampedValue<T> result =
        StampedValue.max(value, last);
    lastRead.set(result);
    return result.value;
}
**Lemma 4.2.5**: The construction is an atomic SRSW register.

**Proof**: `r_value` is a regular register so the first two conditions for an atomic register are satisfied. The third condition for an atomic register is also satisfied because reads are totally ordered by the stamped values.
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

<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>

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

One per thread

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

Art of Multiprocessor Programming
**Multi-Reader Redux**

Writer writes column...

Reader reads row

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

---

*Art of Multiprocessor Programming*
### Multi-Reader Redux

#### Reader Writes Column to Notify Others of What It Read

- **zzz…after second write**
- **2:00, 5678**

<table>
<thead>
<tr>
<th>Time</th>
<th>Reader</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:00</td>
<td>Green</td>
</tr>
<tr>
<td>2:00</td>
<td>Green</td>
</tr>
<tr>
<td>1:45</td>
<td>Red</td>
</tr>
<tr>
<td>2:00</td>
<td>Yellow</td>
</tr>
<tr>
<td>1:45</td>
<td>Red</td>
</tr>
<tr>
<td>2:00</td>
<td>Yellow</td>
</tr>
<tr>
<td>1:45</td>
<td>Red</td>
</tr>
<tr>
<td>2:00</td>
<td>Yellow</td>
</tr>
<tr>
<td>1:45</td>
<td>Red</td>
</tr>
</tbody>
</table>

**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…

1:45
1234

In which case its 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
Atomic MRSW Register

```java
public class AtomicMRSWRegister<T>
    implements Register<T>
{
    ThreadLocal<Long> lastStamp;
    private StampedValue<T>[][] a_table;
```
Atomic MRSW Register

```java
public void write(T v)
{
    long stamp = lastStamp.get() + 1;
    lastStamp.set(stamp);
    StampedValue<T> value =
        new StampedValue<T>(stamp, v);

    for (int i = 0; i < a_table.length; i++)
        a_table[i][i] = value;
}
```
public T read() {
    int me = ThreadID.get();
    StampedValue<T> value = a_table[me][me];

    for (int i = 0; i < a_table.length; i++)
        value = StampedValue.max(value, a_table[i][me]);

    for (int i = 0; i < a_table.length; i++)
        a_table[me][i] = value;

    return value;
}
public T read() {
    int me = ThreadID.get();
    StampedValue<T> value = a_table[me][me];

    for (int i = 0; i < a_table.length; i++)
        value = StampedValue.max(value, a_table[i][me]);

    for (int i = 0; i < a_table.length; i++)
        a_table[me][i] = value;

    return value;
}
Atomic MRSW Register
Atomic MRSW Register

A table representing the Atomic MRSW Register with columns labeled as 'reader 1', 'i', '0', '1', '2', '3', and rows labeled as '0', '1', '2', '3'. The contents of the table are:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>t+1</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>1</td>
<td>t</td>
<td>t+1</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>2</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>3</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
</tbody>
</table>
Atomic MRSW Register

```
  i   0   1   2   3
  0 t+1 t   t   t
  1 t+1 t+1 t+1 t+1
  2 t   t   t   t
  3 t   t   t   t
```
Atomic MRSW Register
Atomic MRSW Register

Lemma 4.2.6: The construction is an atomic MRSW register.

Proof:
- No reader returns a value from the future so the first condition of a regular register is satisfied.
- If A writes v with timestamp t then any read by B where A’s call completely precedes B’s call reads a timestamp greater than or equal to t, satisfying the second condition of a regular register.
Atomic MRSW Register

Recall that a regular register must satisfy the following, two conditions:

1. it is never the case that $R^i \rightarrow W^i$
2. it is never the case that for some $j$ $W^i \rightarrow W^j \rightarrow R^i$

An atomic register must satisfy a single, additional condition:

1. if $R^i \rightarrow R^j$ then $i \leq j$
Atomic MRSW Register

- If a read call by A completely precedes a read call by B then A will write t to B’s column so B chooses a timestamp greater than or equal to t, satisfying the additional condition required by atomic registers.
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

To construct a **multiple-reader, multiple-writer** atomic register, we use the MRSW atomic register created previously.
Multi-Writer Atomic From Multi-Reader Atomic

Each writer reads all and 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
Multi-Writer Atomic From Multi-Reader Atomic

```java
public class StampedValue<T>
{
    public long stamp;
    public T value;
}
```
Multi-Writer Atomic From Multi-Reader Atomic

```java
public static StampedValue max(
    StampedValue x, StampedValue y)
{
    if (x.stamp > y.stamp)
        return x;
    else
        return y;
}

public static StampedValue MIN_VALUE =
    new StampedValue(null);
```
Multi-Writer Atomic From Multi-Reader Atomic

```java
public class AtomicMRMWRegister<T>
    implements Register<T>
{
    private StampedValue<T>[] a_table;
```
Multi-Writer Atomic From Multi-Reader Atomic

```java
public void write(T v)
{
    int me = ThreadID.get();

    StampedValue<T> max = StampedValue.MIN_VALUE;

    for (int i = 0; i < a_table.length; i++)
        max = StampedValue.max(max, a_table[i]);

    a_table[me] =
        new StampedValue(max.stamp + 1, value);
}
```
Multi-Writer Atomic From Multi-Reader Atomic

```java
public T read()
{
    StampedValue<T> max = StampedValue.MIN_VALUE;

    for (int i = 0; i < a_table.length; i++)
        max = StampedValue.max(max, a_table[i]);

    return max.value;
}
```
Atomic MRMW Register

Lemma 4.2.7: The construction is an atomicMRMWRegister.

Proof: Define the write order among write() calls based on a lexicographic order of their timestamps and thread ids.

First Condition: A reader can not read a value written after the read completes. A write, completely preceded by a read, will have a higher timestamp than all the writes before the read completes.
Atomic MRMW Register

Recall that a regular register must satisfy the following, two conditions:

1. it is never the case that $R^i \rightarrow W^i$
2. it is never the case that for some $j$ $W^i \rightarrow W^j \rightarrow R^i$

An atomic register must satisfy a single, additional condition:

1. if $R^i \rightarrow R^j$ then $i \leq j$
Atomic MRMW Register

Second Condition: Assume that:

\[ write_A \rightarrow write_B \rightarrow read_C. \]

- If \( A = B \) then \( write_B \) overwrites table\([A]\).
- If \( A <> B \) then since \( t_A < t_B \), \( C \) will return B’s value.
Atomic MRMW Register

**Third Condition**: Assume that
\[ read_A \rightarrow read_B \text{ and } write_C \rightarrow write_D. \]

We must show that if A returned D’s value then B could not have returned C’s value.

- If \( t_C < t_D \) then if A read \( t_D \) from table[D] then B will also read \( t_D \) from table[D].
- If \( t_C = t_D \) then from the write order C < D so if A read \( t_D \) from table[D] then B will also read \( t_D \) from table[D].
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)
Linearization Points

Look at Writes First

write(1) -> write(2) -> write(2) -> write(2) -> write(3) -> write(4)

time
Linearization Points

Order writes by TimeStamp

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

Art of Multiprocessor Programming
Linearization Points

Order reads by max stamp read

write(1) time
write(2) write(3) write(2)
Read(max = 2) Read(max = 3) Read(max = 4)
Read (max = 1) write(2) Write
Linearization Points

Order reads by max stamp read

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

Linearization Points

Write

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

Time
Road Map

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

Questions?
 Registers - Summary

The behaviour of primitives must be defined in the presence of concurrent read/write operations.

- **Safe registers:**
  - Reliable only when reads and writes are separated by quiescence.

- **Regular registers:**
  - At least as reliable as safe registers.
  - Guarantee also that values flicker between old and new.
Registers - Summary

• Atomic registers:
  • Behave like regular registers.
  • Guarantee also that reads separated by quiescence are ordered.
Linearizability - Example

```java
public class HWQueue<T>
{
    AtomicReference<T>[] items;
    AtomicInteger tail;
    static final int CAPACITY = 1024;
}
```
public HWQueue()
{
    items = (AtomicReference<T>[])Array.newInstance(
        AtomicReference.class, CAPACITY);

    for (int i = 0; i < items.length; i++)
        items[i] = new AtomicReference<T>(null);

    tail = new AtomicInteger(0);
}
public void enq(T x)
{
    int i = tail.getAndIncrement();
    items[i].set(x);
}
public T deq()
{
    while (true)
    {
        int range = tail.get();

        for (int i = 0; i < range; i++)
        {
            T value = items[i].getAndSet(null);
            if (value != null)
                return value;
        }
    }
}
Linearizability - Example

Give an example execution showing that the linearization point for enq() cannot occur where:

```java
int i = tail.getAndIncrement();
```

**Hint:** Give an execution where two enq() calls are not linearized in the order they execute the above line.
Linearizability - Example
Linearizability - Example

Give an example execution showing that the linearization point for enq() cannot occur where:

```java
items[i].set(x);
```
Linearizability - Example

A \quad W'(x) \quad B \quad W^2(y) \quad \text{Reader} \quad R^1()
Next lecture – Chapter 5