Chapter 4
Foundations of Shared Memory

Acknowledgement

Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit

Church-Turing Thesis

- Anything that can be computed, can be computed by a Turing Machine.
- Foundations of sequential computing

Turing Machine

Finite State Controller
Reads and Writes Infinite tape

Concurrent shared-memory computing

- Consists of multiple threads – each a sequential program
- That communicate by calling methods of objects in shared memory

Threads

- Threads are asynchronous
  - They run at different speeds and can be halted for an unpredictable duration at any time
Mathematical model of concurrent computation
What is (and is not) concurrently computable
Efficiency (mostly) irrelevant
public interface Register<T> {
    public T read();
    public void write(T v);
}

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

Jargon Watch
- SRSW
  - Single-reader single-writer
- MRSW
  - Multi-reader single-writer
- MRMW
  - Multi-reader multi-writer

Concurrent registers
- On a multiprocessor, we expect reads and writes to overlap
- How do we specify what a concurrent method call mean?

One approach
- Rely on mutual exclusion:
  - Protect each register with a mutex lock acquired by each `read()` and `write()` call
  - Possible problems?

Different approach: Wait-Free Implementation
Definition: An object implementation is wait-free if every method call completes in a finite number of steps

- No mutual exclusion
- Guarantees independent progress
- We require register implementations to be wait-free

Different kinds of registers
- According to:
  - Range of values
    - Boolean or Integer (M-valued)
  - Number of readers and writers
  - Degree of consistency
Degree of consistency

- Safe
- Regular
- Atomic

Safe Register

- A single-writer, multi-reader register is safe if:
  - A read() that does not overlap a write() returns the last value
  - If a read() overlaps a write() it can return any value within the register’s range

Safe Register

- OK if reads and writes don’t overlap

write(1001)

read(1001)

Regular register

- A single-writer, multi-reader register is regular if:
  - A read() that does not overlap a write() returns the last value
  - If a read() overlaps a write() it returns either the old value or the new value
  - Value being read may “flicker” between the old and new value before finally changing to the new value

Regular or Not?

write(0)

read()

write(1)

read()

write(1)

read(0)
Regular or Not?

Overlap: returns new value

Overlap: returns old value

Regular ≠ Linearizable

Atomic register
- Linearizable implementation of sequential register
- A single-writer, multi-reader register is atomic if:
  - Each read() returns the last value written
Atomic Register

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

Linearizable?

---

Register Space

- Safe
- Regular
- Atomic

- MRMW
- MRSW
- SRSW

Road Map

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

---

Weakest Register

- Single writer
- Single reader

Safe Boolean register

---

Register construction

- We will now build a range of registers from single-reader, single-writer Boolean safe registers
Road Map

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

Register Names

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

Register Names

- Property
- Type
- How many readers & writers?

Safe Boolean MRSW

- Let's Write It!
- 0 or 1

Safe Boolean MRSW

- 0 or 1
Safe Boolean MRSW

Each thread has its own safe SRSW register.

public class SafeBoolMRSWRegister implements Register<boolean> {
    boolean[] s_table; //array of SRSW registers
    public SafeBoolMRSWRegister(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.

Each thread reads its own register.

public class SafeBoolMRSWRegister implements Register<boolean> {
    boolean[] s_table; //array of SRSW registers
    public SafeBoolMRSWRegister(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 Multi-Valued MRSW?

Road Map
- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic

Safe Boolean MRSW vs Regular Boolean MRSW
- Only difference is when newly written value is same as old value:
  - Safe register can return either Boolean value
  - Regular register can return either new value or old value – if both new and old is x, then regular can only return x
- So… write value only if distinct from previous written value

Safe Boolean MRSW → Regular Boolean MRSW

Safe Boolean MRSW → Regular Boolean MRSW
Safe Boolean MRSW → Regular 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();
    }
}
```

Actual value

Is new value different from last value I wrote?

If so, change it (otherwise don't!)

Overlap? No Overlap?

No problem - either Boolean value works

Last bit this thread wrote (made-up syntax)
Safe Multi-Valued MRSW →
Regular Multi-Valued MRSW

Regular M-Valued MRSW Register

- Values are represented using unary notation
- An M-valued register is implemented as an array of \( m \) regular MRSW Boolean registers
- Initially the register is set to 0

Writing M-Valued

Unary representation:
\( \text{bit}[i] \) means value \( i \)

Road Map

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

Writing M-Valued

Initial Value:

0 1 2 3 4 5 6 7

Write:

Write 5

Initial 0:

0 0 0 0 0 1

0 1 2 3 4 5 6 7

Next

Reader

Writer
Writing M-Valued

MRSW Regular Boolean → MRSW Regular M-valued

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;
    }
}

Unary representation: bit[i] means value i

Set bit x

Clear bits from higher to lower

Scan from lower to higher & return first bit set
Regular Register Conditions

- Further conditions for a register to be regular:
  - No read() call should return a value from the future
  - No read() call should return a value from the distant past – only the most recently written non-overlapping value must be returned

Road Map

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

Road Map (Slight Detour)

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

Atomic Register Conditions

- Together with the conditions for a regular register, an additional condition for an atomic register is:
  - An earlier read() cannot return a value later that returned by a later read()
  - In other words, values read() should be in the correct order

SRSW register

- Since a SRSW register has no concurrent reads, the only way that the condition for an atomic register can be violated is when two reads that overlap the same write read values out of order

SRSW Regular

- Concurrent Reading
- When could this happen?
Initially 1234

Timestamps

- Solution is to for each value to have an added tag – a timestamp
- Timestamps are used to order concurrent calls

Timestamped Values

- The writer writes a timestamp to a value
- Each reader remembers the latest timestamp/value pair ever read
- If a later read() then returns an earlier value the value is discarded and the reader uses the last value
SRSW Regular → SRSW Atomic

Atomic SRSW

```
public class StampedValue<T> {
    public long stamp;
    public T value;
    public StampedValue(T init) {
        stamp = 0;
        value = init;
    }
    public StampedValue max(StampedValue x, StampedValue y) {
        if (x.stamp > y.stamp)
            return x;
        else return y;
    }
}
```

Atomic SRSW

```
public class AtomicSRSWRegister<T> implements Register<T> {
    long lastStamp;
    StampedValue<T> lastRead;
    StampedValue<T> value;
    public T read() {
        StampedValue<T> result = StampedValue.max(value, lastRead);
        lastRead = result;
        return result.value;
    }
    public void write(T v) {
        long stamp = lastStamp + 1;
        value = new StampedValue(stamp, v);
        lastStamp = stamp;
    }
}
```

Atomic SRSW → Atomic MRSW

- Can the atomic SRSW be used to build an atomic MRSW?
- Solution of Safe MRSW Registers:
  - Every thread in array
  - Write starts at the beginning of the array and iterates through array
  - Read reads only its own array location

Atomic Single Reader → Atomic Multi-Reader

Atomic MRSW → Atomic SRSW
Atomic SRSW $\rightarrow$ Atomic MRSW

Atomic MRSW

- n-threads share a n-by-n array of stamped values
- Read() calls determine latest threads by timestamps
- Similar to the Safe MRSW Register implementation, the writer writes the new values to the array, but only on the diagonals

We address this problem by having earlier reader threads help out later threads, by telling them which value they read.

If not then the current value is the latest
Write 5

Reader 1
Is there a value in this column with a higher timestamp?

Read value with highest timestamp

Reader 1
Read 5

Reader 1
Write highest timestamp to row

Reader 3
Is there a value in this column with a higher timestamp?
public class AtomicMRSWRegister implements Register{
    long lastStamp;
    StampedValue<T>[] a_table;
    
    public T read() {
        int me = ThreadID.get();
        StampedValue<T> value = a_table[me][me];
        for (int i = 0; i < n; i++)
            value = StampedValue.max(value, a_table[i][me]);
        for (int i = 0; i < n; i++)
            a_table[me][i] = value;
    }
    
    public void write(T v) {
        long stamp = lastStamp + 1;
        lastStamp = stamp;
        StampedValue<T> value = new StampedValue<T>(stamp, v);
        for (int i = 0; i < n; i++)
            a_table[i][i] = value;
    }
}

Can't Yellow Miss Blue's Update? ...
... Only if Readers Overlap...
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

Multi-Writer Atomic From Multi-Reader Atomic

Readers read all and take max (Lexicographic like Bakery)

1:45 1234
Max is 2:15, return XYZW

MRMW Atomic

public class AtomicMRMWRegister implements Register{
    StampedValue<T>[] a_table;
    public void write (T value)  {
        int me = ThreadID.get();
        StampedValue<T> max = StampedValue.MIN;
        for (int i = 0; i < n; i++)
            max = StampedValue.max(max, a_table[i]);
        a_table[me] = new StampedValue(max.stamp + 1, value);
    }
}

Write new value to array
public T read() {
    StampedValue<T> max = StampedValue.MIN;
    for (int i = 0; i < n; i++)
        max = StampedValue.max(max, a_table[i]);
    return max.value;
}

Find highest timestamp

Conclusion

- One can construct a wait-free MRMW atomic register from SRSW Safe Boolean registers