

Foundations of Shared Memory

Nir Shavit
Subing for N. Lynch
Fall 2003

Fundamentals

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

© 2003 Herlihy & Shavit

2

Alan Turing

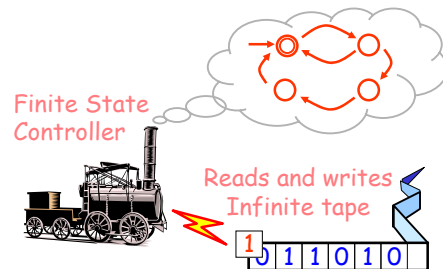


- First (and still the best!) mathematical model of sequential computation
- First to distinguish between what is and is not computable

© 2003 Herlihy & Shavit

3

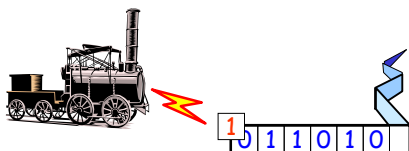
Turing Machine



© 2003 Herlihy & Shavit

4

Turing Computability



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

© 2003 Herlihy & Shavit

5

Shared-Memory Computability?



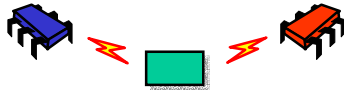
- Mathematical model of concurrent computation
- What is (and is not) concurrently computable
- Efficiency (mostly) irrelevant

© 2003 Herlihy & Shavit

6

Foundations of Shared Memory

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



© 2003 Herlihy & Shavit

7

Foundations of Shared Memory

What is the weakest useful form of shared memory?

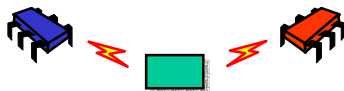


© 2003 Herlihy & Shavit

8

Foundations of Shared Memory

What can it do?

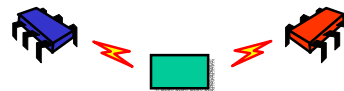


© 2003 Herlihy & Shavit

9

Foundations of Shared Memory

What can't it do?

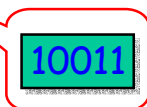


© 2003 Herlihy & Shavit

10

Register

Holds a (binary) value

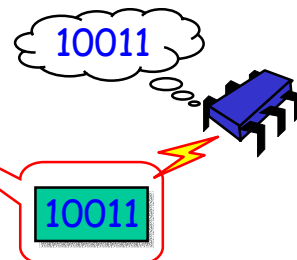


© 2003 Herlihy & Shavit

11

Register

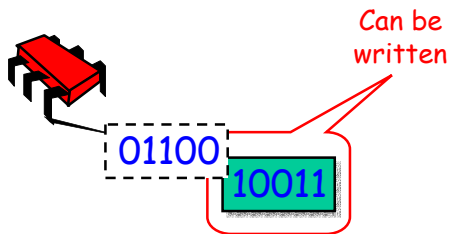
Can be read



© 2003 Herlihy & Shavit

12

Register



© 2003 Herlihy & Shavit

13

Registers

```
public interface BooleanRegister {
    public boolean read();
    public void write(boolean v);
}

public interface Register {
    public int read();
    public void write(int v);
}
```

© 2003 Herlihy & Shavit

14

Registers

```
public interface BooleanRegister {
    public boolean read();
    public void write(boolean v);
}

public interface Register {
    public int read();
    public void write(int v);
}
```

Boolean (1-bit) flavor

© 2003 Herlihy & Shavit

15

Registers

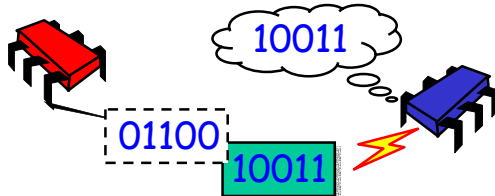
```
Multi-value (M-bit) flavor
public interface Register {
    public boolean read();
    public void write(boolean v);
}

public interface Register {
    public int read();
    public void write(int v);
}
```

© 2003 Herlihy & Shavit

16

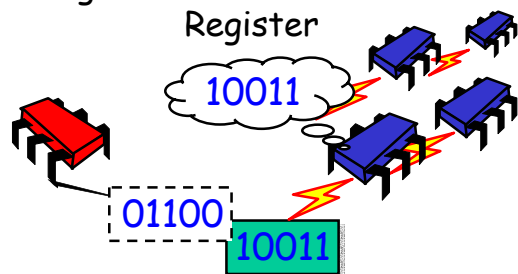
Single-Writer/Single-Reader Register



© 2003 Herlihy & Shavit

17

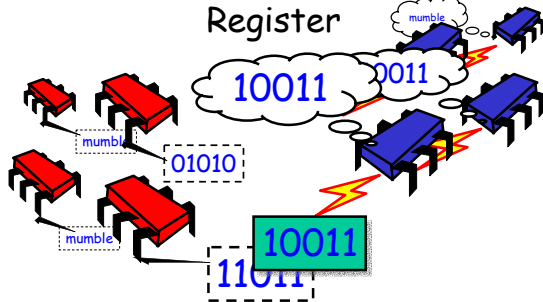
Single-Writer/Multi-Reader Register



© 2003 Herlihy & Shavit

18

Multi-Writer/Multi-Reader Register



© 2003 Herlihy & Shavit

19

Jargon Watch

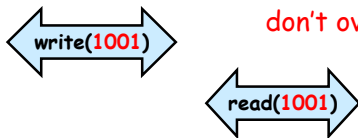
- SRSW
 - Single-reader single-writer
- MRSW
 - Multi-reader single-writer
- MRMW
 - Multi-reader multi-writer

© 2003 Herlihy & Shavit

20

Safe Register

OK if reads and writes don't overlap



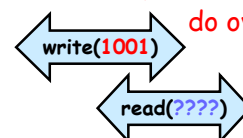
(2)

© 2003 Herlihy & Shavit

21

Safe Register

Effects undefined if reads and writes do overlap

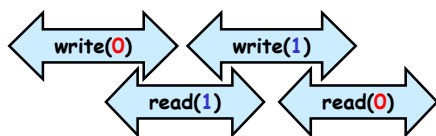


(2)

© 2003 Herlihy & Shavit

22

Regular Register

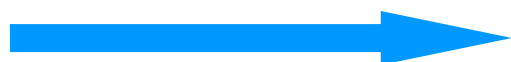
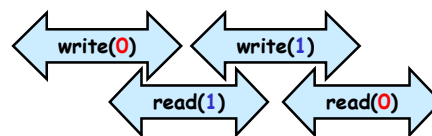


- Single Writer
- Readers return:
 - Old value if no overlap (safe)
 - Old or new value if overlap

© 2003 Herlihy & Shavit

23

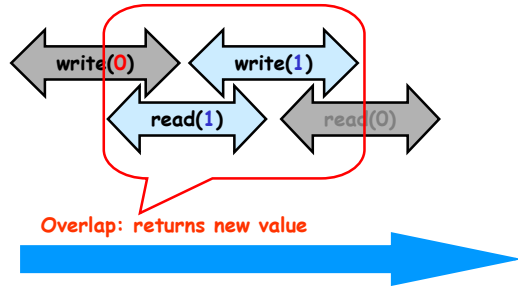
Regular or Not?



© 2003 Herlihy & Shavit

24

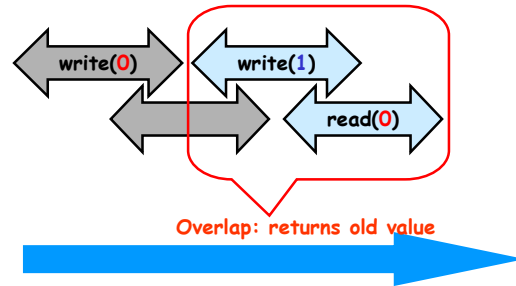
Regular or Not?



© 2003 Herlihy & Shavit

25

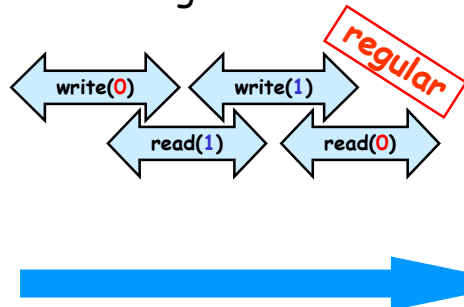
Regular or Not?



© 2003 Herlihy & Shavit

26

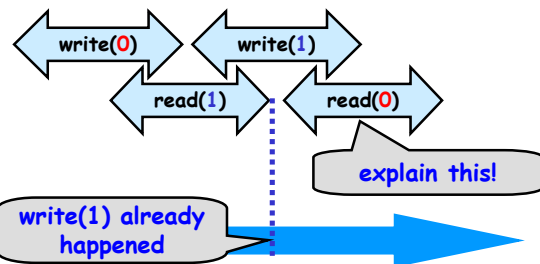
Regular or Not?



© 2003 Herlihy & Shavit

27

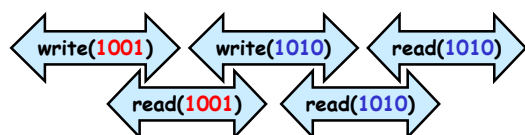
Regular \neq Linearizable



© 2003 Herlihy & Shavit

28

Atomic Register

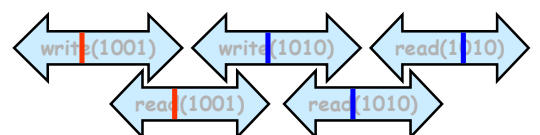


Linearizable to sequential safe register

© 2003 Herlihy & Shavit

29

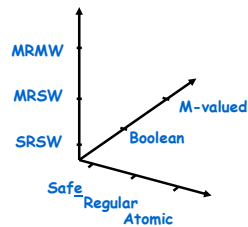
Atomic Register



© 2003 Herlihy & Shavit

30

Register Space



© 2003 Herlihy & Shavit

31

Register Names

```
public class SafeBooleanRegister
implements BooleanRegister {
    public boolean read() { ... }
    public void write(boolean x) { ... }
}
```

(3)

© 2003 Herlihy & Shavit

32

Register Names

```
public class SafeBooleanRegister
implements BooleanRegister {
    public boolean read() { ... }
    public void write(boolean x) { ... }
}
```

property

(3)

© 2003 Herlihy & Shavit

33

Register Names

```
public class SafeBooleanRegister
implements BooleanRegister {
    public boolean read() { ... }
    public void write(boolean x) { ... }
}
```

property

Size matters

(3)

© 2003 Herlihy & Shavit

34

Register Names

```
public class SafeBooleanRegister
implements BooleanRegister {
    public boolean read() { ... }
    public void write(boolean x) { ... }
}
```

property

Size matters

How many readers
& writers?

(3)

© 2003 Herlihy & Shavit

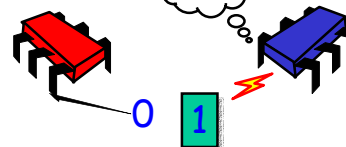
35

Weakest Register

Single writer

1

Single reader



Safe Boolean register

© 2003 Herlihy & Shavit

36

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 ...

(2)

© 2003 Herlihy & Shavit

37

Locking within Registers

- Not interesting to rely on mutual exclusion in register constructions
- We want registers to implement mutual exclusion!
- No fun to use mutual exclusion to implement itself!

© 2003 Herlihy & Shavit

38

Wait-Free Implementations

Definition: An object implementation is *wait-free* if every thread completes a method in a finite number of steps

No mutual exclusion

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

© 2003 Herlihy & Shavit

39


Road Map

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

© 2003 Herlihy & Shavit

40

Road Map

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

© 2003 Herlihy & Shavit

41

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 = Thread.myIndex();
    return r[i].read();
  }
}
```

(2)

© 2003 Herlihy & Shavit

42

Safe Boolean MRSW from Safe Boolean SRSW

```
public class SafeBooleanMRSWRegister
implements BooleanRegister {
    private SafeBooleanSRSWRegister[] r =
        new SafeBooleanSRSWRegister[N];
    public void write(boolean x) {
        for (int j = 0; j < N; j++)
            r[j].write(x);
    }
    public boolean read() {
        int i = Thread.myIndex();
        return r[i].read();
    }
}
```

Each thread has own safe SRSW register

(2)

© 2003 Herlihy & Shavit

43

Safe Boolean MRSW from Safe Boolean SRSW

```
public class SafeBooleanMRSWRegister
implements BooleanRegister {
    private SafeBooleanSRSWRegister[] r =
        new SafeBooleanSRSWRegister[N];
    public void write(boolean x) {
        for (int j = 0; j < N; j++)
            r[j].write(x);
    }
    public boolean read() {
        int i = Thread.myIndex();
        return r[i].read();
    }
}
```

write method

(2)

© 2003 Herlihy & Shavit

44

Safe Boolean MRSW from Safe Boolean SRSW

```
public class SafeBooleanMRSWRegister
implements BooleanRegister {
    private SafeBooleanSRSWRegister[] r =
        new SafeBooleanSRSWRegister[N];
    public void write(boolean x) {
        for (int j = 0; j < N; j++)
            r[j].write(x);
    }
    public boolean read() {
        int i = Thread.myIndex();
        return r[i].read();
    }
}
```

Write each thread's register one at a time

(2)

© 2003 Herlihy & Shavit

45

Safe Boolean MRSW from Safe Boolean SRSW

```
public class SafeBooleanMRSWRegister
implements BooleanRegister {
    private SafeBooleanSRSWRegister[] r =
        new SafeBooleanSRSWRegister[N];
    public void write(boolean x) {
        for (int j = 0; j < N; j++)
            r[j].write(x);
    }
    public boolean read() {
        int i = Thread.myIndex();
        return r[i].read();
    }
}
```

read method

(2)

© 2003 Herlihy & Shavit

46

Safe Boolean MRSW from Safe Boolean SRSW

```
public class SafeBooleanMRSWRegister
implements BooleanRegister {
    private SafeBooleanSRSWRegister[] r =
        new SafeBooleanSRSWRegister[N];
    public void write(boolean x) {
        for (int j = 0; j < N; j++)
            r[j].write(x);
    }
    public boolean read() {
        int i = Thread.myIndex();
        return r[i].read();
    }
}
```

Read my own register

(2)

© 2003 Herlihy & Shavit

47

Road Map

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

Questions?

© 2003 Herlihy & Shavit

48

Road Map

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

© 2003 Herlihy & Shavit

49

Regular Boolean MRSW from Safe Boolean MRSW

```
public class RegBoolMRSWRegister
implements BooleanRegister {
    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();
    }
}
```

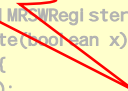
(2)

© 2003 Herlihy & Shavit

50

Regular Boolean MRSW from Safe Boolean MRSW

```
public class RegBoolMRSWRegister
implements BooleanRegister {
    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();
    }
}
```

 Last bit this thread wrote
(OK, we're cheating here on Java syntax)

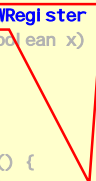
(2)

© 2003 Herlihy & Shavit

51

Regular Boolean MRSW from Safe Boolean MRSW

```
public class RegBoolMRSWRegister
implements BooleanRegister {
    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

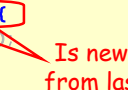
(2)

© 2003 Herlihy & Shavit

52

Regular Boolean MRSW from Safe Boolean MRSW

```
public class RegBoolMRSWRegister
implements BooleanRegister {
    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();
    }
}
```

 Is new value different from last value I wrote?


(2)

© 2003 Herlihy & Shavit

53

Regular Boolean MRSW from Safe Boolean MRSW

```
public class RegBoolMRSWRegister
implements BooleanRegister {
    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();
    }
}
```

 If so, change it

(2)

© 2003 Herlihy & Shavit

54

Regular Boolean MRSW from Safe Boolean MRSW

```
public class RegBoolMRSWRegister
    implements BooleanRegister {
    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();
    }
}
```

•Overlap? No Overlap?
•No problem
•either Boolean value works

(2)

© 2003 Herlihy & Shavit

55

Road Map

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

Questions?

© 2003 Herlihy & Shavit

56

Road Map

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

© 2003 Herlihy & Shavit

57

MRSW Regular M-valued from MRSW Regular Boolean

```
public class RegMRSWRegister implements Register{
    RegBoolMRSWRegister[] blt;

    public void write(int x) {
        for (int i=0; i<M; i++)
            blt[i].write(x>0);
    }

    public int read() {
        if (blt[0].read() != blt[1].read())
            return -1;
        return blt[0].read();
    }
}
```

Viewer discretion advised
(sorry, tuition is non-refundable)

(1)

© 2003 Herlihy & Shavit

58

MRSW Regular M-valued from MRSW Regular Boolean

```
public class RegMRSWRegister implements Register{
    RegBoolMRSWRegister[] blt;

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

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

(1)

© 2003 Herlihy & Shavit

59

MRSW Regular M-valued from MRSW Regular Boolean

```
public class RegMRSWRegister implements Register{
    RegBoolMRSWRegister[] blt;

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

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

Unary representation:
bit[i] means value i

(1)

© 2003 Herlihy & Shavit

60

MRSW Regular M-valued from MRSW Regular Boolean

```
public class RegMRSWRegister implements Register {
    RegBool MRSWRegister[m] blt;

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

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

Set bit x

(1)

© 2003 Herlihy & Shavit

61

MRSW Regular M-valued from MRSW Regular Boolean

```
public class RegMRSWRegister implements Register {
    RegBool MRSWRegister[m] blt;

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

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

Clear lower bits

(1)

© 2003 Herlihy & Shavit

62

MRSW Regular M-valued from MRSW Regular Boolean

```
public class RegMRSWRegister implements Register {
    RegBool MRSWRegister[m] blt;

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

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

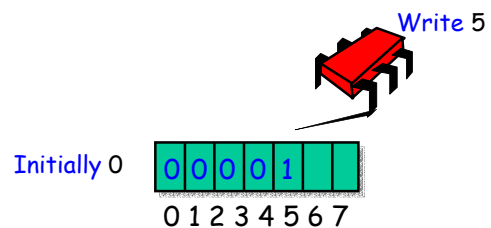
Find & return first bit set

(1)

© 2003 Herlihy & Shavit

63

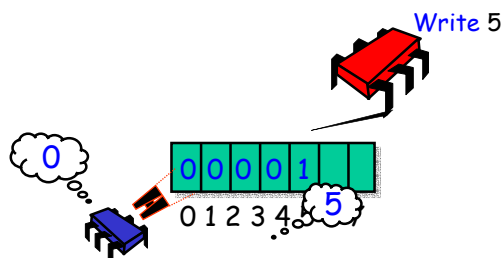
Writing M-Valued



© 2003 Herlihy & Shavit

64

Writing M-Valued



© 2003 Herlihy & Shavit

65

Road Map



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

Questions?

© 2003 Herlihy & Shavit

66


Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular  ~~Complicated and boring~~
- MRSW atomic  Of interest mostly to specialists
- MRMW atomic
- Atomic snapshot

© 2003 Herlihy & Shavit

67

Road Map

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

© 2003 Herlihy & Shavit

68

Multi-writer Atomic from Single-Writer Atomic

```
public class Label edVal ue {
    public Int label;
    public Int val ue;

    public static Label edVal ue
        MI N_VALUE = new Label edVal ue(0, 0);

    public Label edVal ue(Int label , Int val ue) {
        this. label = label;
        this. val ue = val ue;
    }
}
```

(2)

© 2003 Herlihy & Shavit

69

Multi-writer Atomic from Single-Writer Atomic

```
public class Label edVal ue {
    public Int label;
    public Int val ue;

    public static Label edVal ue
        MI N_VALUE = new Label edVal ue(0, 0);

    public Label edVal ue(Int label , Int val ue) {
        this. label = label;
        this. val ue = val ue;
    }
}
```

counter incremented on each write

(2)

© 2003 Herlihy & Shavit

70

Multi-writer Atomic from Single-Writer Atomic

```
public class Label edVal ue {
    public Int label;
    public Int val ue;

    public static Label edVal ue
        MI N_VALUE = new Label edVal ue(0, 0);

    public Label edVal ue(Int label , Int val ue) {
        this. label = label;
        this. val ue = val ue;
    }
}
```

actual value

(2)

© 2003 Herlihy & Shavit

71

Multi-writer Atomic from Single-Writer Atomic

```
public class Label edVal ue {
    public Int label;
    public Int val ue;

    public static Label edVal ue
        MI N_VALUE = new Label edVal ue(0, 0);

    public Label edVal ue(Int label , Int val ue) {
        this. label = label;
        this. val ue = val ue;
    }
}
```

constructor

(2)

© 2003 Herlihy & Shavit

72

Multi-writer Atomic from Single-Writer Atomic

```
public class LabeledValue {
    public int label;
    public int value;

    public static LabeledValue
    MIN_VALUE = new LabeledValue(0, 0);

    public LabeledValue(int label, int value) {
        this.label = label;
        this.value = Smallest label ever
    }
}
```

(2)

© 2003 Herlihy & Shavit

73

Writing Labeled Values

1101111010100010

label value

(2)

© 2003 Herlihy & Shavit

74

Multi-Writer Atomic from Single-Writer Atomic

```
public class AtomicMRMWRRegister {
    private AtomicMRSWRegister[] r =
        new AtomicMRSWRegister[n];

    public AtomicMRMWRRegister() {
        for (int j = 0; j < n; j++) {
            r[j] = new AtomicMRSWRegister();
            r[j].write(LabeledValue.MIN_VALUE);
        }
    }
}
```

(2)

© 2003 Herlihy & Shavit

75

Multi-Writer Atomic from Single-Writer Atomic

```
public class AtomicMRMWRRegister {
    private AtomicMRSWRegister[] r =
        new AtomicMRSWRegister[n];

    public AtomicMRMWRRegister() {
        for (int j = 0; j < n; j++) {
            r[j] = new AtomicMRSWRegister();
            r[j].write(One Single-Writer Register per thread);
        }
    }
}
```

(2)

© 2003 Herlihy & Shavit

76

Multi-Writer Atomic from Single-Writer Atomic

```
public class AtomicMRMWRRegister {
    Initialize all to min label
    private AtomicMRSWRegister[] r =
        new AtomicMRSWRegister[n];

    public AtomicMRMWRRegister() {
        for (int j = 0; j < n; j++) {
            r[j] = new AtomicMRSWRegister();
            r[j].write(LabeledValue.MIN_VALUE);
        }
    }
}
```

(2)

© 2003 Herlihy & Shavit

77

Multi-Writer Atomic from Single-Writer Atomic

```
public void write(int value) {
    int i = Thread.myIndex();
    LabeledValue max = LabeledValue.MIN_VALUE;
    for (int j = 0; j < n; j++) {
        LabeledValue other = r[j].read();
        if (other.label > max.label)
            max = other;
        r[i].write(new LabeledValue(max.label + 1,
                                    value));
    }
}
```

(2)

© 2003 Herlihy & Shavit

78

Multi-Writer Atomic from Single-Writer Atomic

```

Find highest label
public void write(int value) {
    int i = Thread.mvIndex();
    Label edVal ue max = Label edVal ue. MIN_VALUE;
    for (int j = 0; j < n; j++) {
        Label edVal ue other = r[j].read();
        if (other.label > max.label)
            max = other;
    }
    r[i].write(new Label edVal ue(max.label + 1,
                                value));
}

```

(2)

© 2003 Herlihy & Shavit

79

Multi-Writer Atomic from Single-Writer Atomic

```

public void write(int value) {
    int i = Thread.mvIndex();
    Write new value with higher label
    Label edVal ue max = Label edVal ue. MIN_VALUE;
    for (int j = 0; j < n; j++) {
        Label edVal ue other = r[j].read();
        if (other.label > max.label)
            max = other;
    }
    r[i].write(new Label edVal ue(max.label + 1,
                                value));
}

```

(2)

© 2003 Herlihy & Shavit

80

Multi-Writer Atomic from Single-Writer Atomic

```

int read() {
    Label edVal ue max = Label edVal ue. MIN_VALUE;
    for (int j = 0; j < n; j++) {
        Label edVal ue other = r[j].read();
        if (other.label > max.label)
            max = other;
    }
    return max.val ue;
}

```

(2)

© 2003 Herlihy & Shavit

81

Multi-Writer Atomic from Single-Writer Atomic

```

int read() {
    Label edVal ue max = Label edVal ue. MIN_VALUE;
    for (int j = 0; j < n; j++) {
        Label edVal ue other = r[j].read();
        if (other.label > max.label)
            max = other;
    }
    return max.val ue;
}
Find highest label

```

(2)

© 2003 Herlihy & Shavit

82

Multi-Writer Atomic from Single-Writer Atomic

```

int read() {
    Label edVal ue max = Label edVal ue. MIN_VALUE;
    for (int j = 0; j < n; j++) {
        Label edVal ue other = r[j].read();
        if (other.label > max.label)
            max = other;
    }
    Return value with highest label
    return max.val ue;
}

```

(2)

© 2003 Herlihy & Shavit

83

Road Map

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

© 2003 Herlihy & Shavit

84

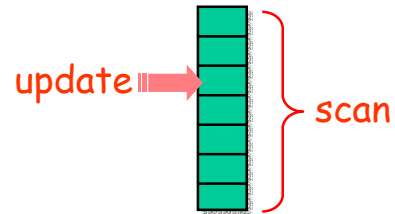
Road Map

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

© 2003 Herlihy & Shavit

85

Atomic Snapshot



© 2003 Herlihy & Shavit

86

Atomic Snapshot

- Array of SWMR atomic registers
- Take instantaneous snapshot of all
- Generalizes to MRMW registers ...

© 2003 Herlihy & Shavit

87

Snapshot Interface

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

(2)

© 2003 Herlihy & Shavit

88

Snapshot Interface

Thread i writes v to its register

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

(2)

© 2003 Herlihy & Shavit

89

Snapshot Interface

Instantaneous snapshot of all threads' registers

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

(2)

© 2003 Herlihy & Shavit

90

Atomic Snapshot

- Collect
 - Read values one at a time
- Problem
 - Incompatible concurrent collects
 - Result not linearizable

© 2003 Herlihy & Shavit

91

Clean Collects

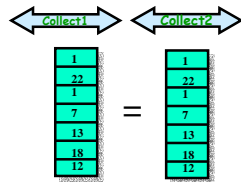
- Clean Collect
 - Collect during which nothing changed
 - Can we make it happen?
 - Can we detect it?

© 2003 Herlihy & Shavit

92

Simple Snapshot

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



© 2003 Herlihy & Shavit

93

Simple Snapshot: Update

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

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

(1)

© 2003 Herlihy & Shavit

94

Simple Snapshot: Update

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

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

One single-writer register per thread

(1)

© 2003 Herlihy & Shavit

95

Simple Snapshot: Update

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

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

Write each time with higher label

(1)

© 2003 Herlihy & Shavit

96

Simple Snapshot: Collect

```
private Label edVal ue[] collect() {
    Label edVal ue[] copy =
        new Label edVal ue[n];
    for (Int j = 0; j < n; j++)
        copy[j] = this.s.register[j].read();
    return copy;
}
```

(1)

© 2003 Herlihy & Shavit

97

Simple Snapshot

```
private Label edVal ue[] collect() {
    Label edVal ue[] copy =
        new Label edVal ue[n];
    for (Int j = 0; j < n; j++)
        copy[j] = this.s.register[j].read();
    return copy;
}
```

Just read each register into array

(1)

© 2003 Herlihy & Shavit

98

Simple Snapshot: Scan

```
public Int[] scan() {
    Label edVal ue[] ol dCopy, newCopy;
    ol dCopy = collect();
    collect: while (true) {
        newCopy = collect();
        if (!equals(ol dCopy, newCopy)) {
            ol dCopy = newCopy;
            continue collect;
        }
        return getVal ues(newCopy);
    }
}
```

(1)

© 2003 Herlihy & Shavit

99

Simple Snapshot: Scan

```
public Int[] scan() {
    Label edVal ue[] ol dCopy, newCopy;
    ol dCopy = collect();
    collect: while (true) {
        newCopy = collect();
        if (!equals(ol dCopy, newCopy)) {
            ol dCopy = newCopy;
            continue collect;
        }
        return getVal ues(newCopy);
    }
}
```

Collect once

(1)

© 2003 Herlihy & Shavit

100

Simple Snapshot: Scan

```
public Int[] scan() {
    Label edVal ue[] ol dCopy, newCopy;
    ol dCopy = collect();
    collect: while (true) {
        newCopy = collect();
        if (!equals(ol dCopy, newCopy)) {
            ol dCopy = newCopy;
            continue collect;
        }
        return getVal ues(newCopy);
    }
}
```

Collect once

Collect twice

(1)

© 2003 Herlihy & Shavit

101

Simple Snapshot: Scan

```
public Int[] scan() {
    Label edVal ue[] ol dCopy, newCopy;
    ol dCopy = collect();
    collect: while (true) {
        newCopy = collect();
        if (!equals(ol dCopy, newCopy)) {
            ol dCopy = newCopy;
            continue collect;
        }
        return getVal ues(newCopy);
    }
}
```

Collect once

Collect twice

On mismatch, try again

(1)

© 2003 Herlihy & Shavit

102

Simple Snapshot: Scan

```
public int[] scan() {
    Label edValue[] 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

(1)

© 2003 Herlihy & Shavit

103

Simple Snapshot

- Linearizable
- Update is wait-free
 - No unbounded loops
- But Scan can starve
 - If interrupted by concurrent update

© 2003 Herlihy & Shavit

104

Wait-Free Snapshot

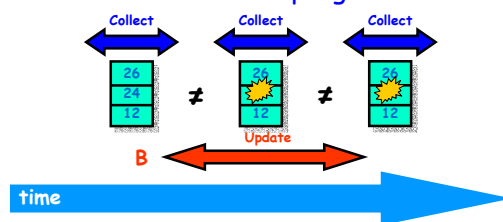
- Add a scan before every update
- Write resulting snapshot together with update value
- If scan is continuously interrupted by updates, scan can take the update's snapshot

© 2003 Herlihy & Shavit

105

Wait-free Snapshot

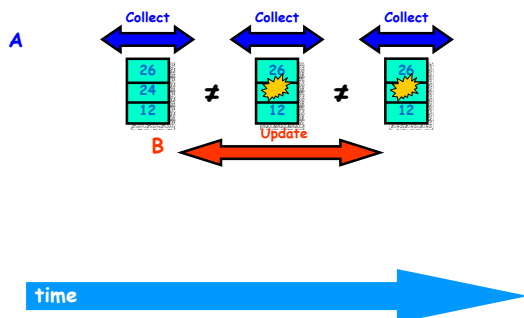
If A's scan observes that B moved twice, then B completed an update while A's scan was in progress



© 2003 Herlihy & Shavit

106

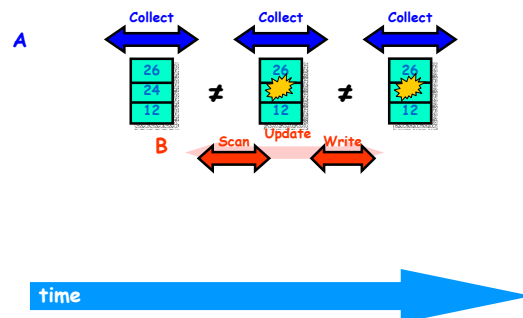
Wait-free Snapshot



© 2003 Herlihy & Shavit

107

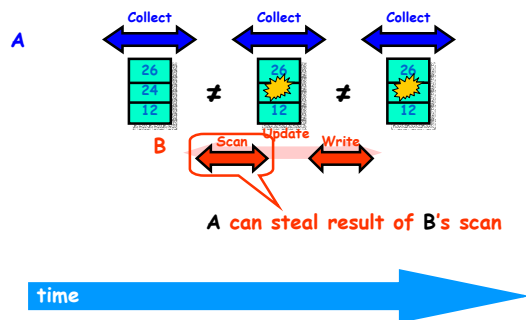
Wait-free Snapshot



© 2003 Herlihy & Shavit

108

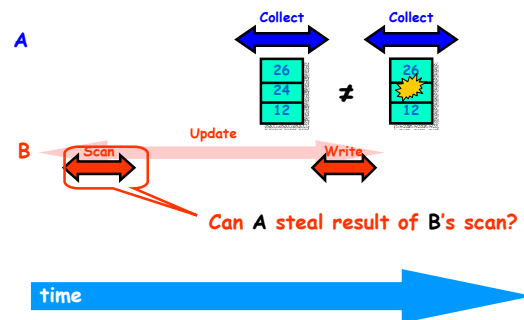
Wait-free Snapshot



© 2003 Herlihy & Shavit

109

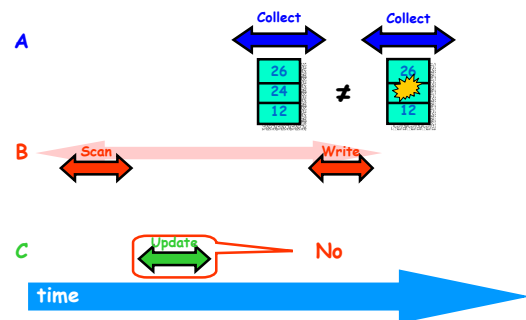
Once is not Enough



© 2003 Herlihy & Shavit

110

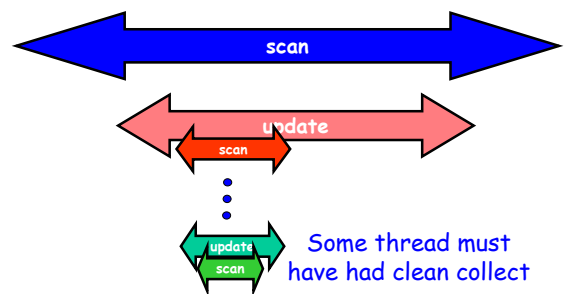
Once is not Enough



© 2003 Herlihy & Shavit

111

Wait-free



© 2003 Herlihy & Shavit

112

Wait-Free Snapshot Label

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

(2)

© 2003 Herlihy & Shavit

113

Wait-Free Snapshot Label

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

Counter incremented with each snapshot

(2)

© 2003 Herlihy & Shavit

114

Wait-Free Snapshot Label

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

Actual value

(2)

© 2003 Herlihy & Shavit

115

Wait-Free Snapshot Label

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

most recent snapshot

(2)

© 2003 Herlihy & Shavit

116

Wait-Free Snapshot Label

11011110101000101100...00

label

value

Last
snapshot

(3)

© 2003 Herlihy & Shavit

117

Wait-free 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);
}
```

(2)

© 2003 Herlihy & Shavit

118

Wait-free 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);
}
```

Take scan

(2)

© 2003 Herlihy & Shavit

119

Wait-free 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);
}
```

Label value with scan

(2)

© 2003 Herlihy & Shavit

120

Wait-free Snapshot

```
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);
    }
}
```

(2)

© 2003 Herlihy & Shavit

121

Wait-free Snapshot

```
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

(2)

© 2003 Herlihy & Shavit

122

Mismatch Detected

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

(2)

© 2003 Herlihy & Shavit

123

Mismatch Detected

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

If thread moved twice,
just steal its snapshot

(2)

© 2003 Herlihy & Shavit

124

Mismatch Detected

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

Remember that
thread moved

(2)

© 2003 Herlihy & Shavit

125

Observations

- Uses unbounded counters
 - can be replaced with 2 bits
- Assumes SWMR registers
 - for labels
 - Can be extended to MRMW

© 2003 Herlihy & Shavit

126

Grand Challenge

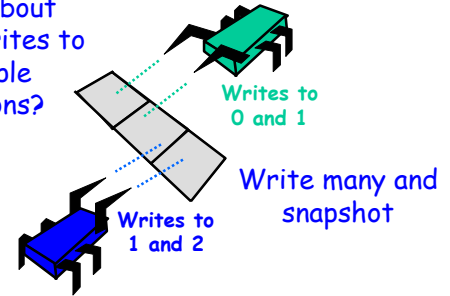
- Snapshot means
 - Write any one array element
 - Read multiple array elements

© 2003 Herlihy & Shavit

127

Grand Challenge

What about
atomic writes to
multiple
locations?



© 2003 Herlihy & Shavit

128