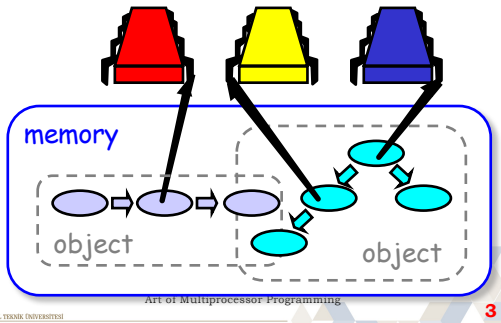
Concurrent Objects

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

Correctness of Concurrent Objects

Concurrent Computaton



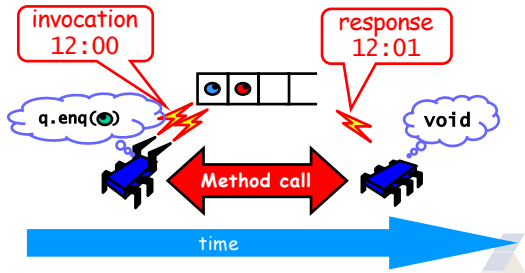
Objectivism

- What is a concurrent object?
  - How do we describe one?
  - How do we implement one?
  - How do we tell if we're right?

Correctness and Progress

- In a concurrent setting, we need to specify both the safety and the liveness properties of an object
- Need a way to define
  - when an implementation is correct
  - the conditions under which it guarantees progress

Methods Take Time

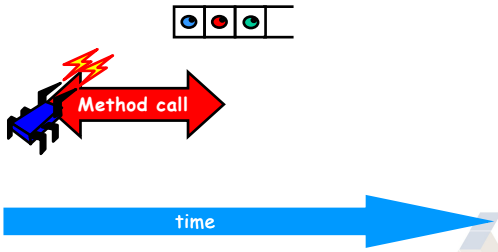


Sequential vs Concurrent

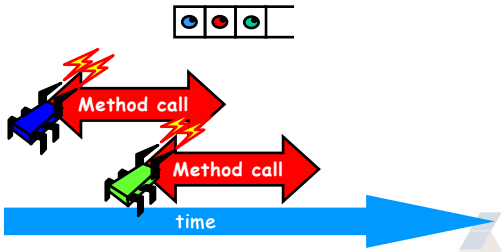
- Sequential
  - Methods take time? Who knew?
- Concurrent
  - Method call is not an event
  - Method call is an interval.

Concurrent Methods Take Overlapping Time

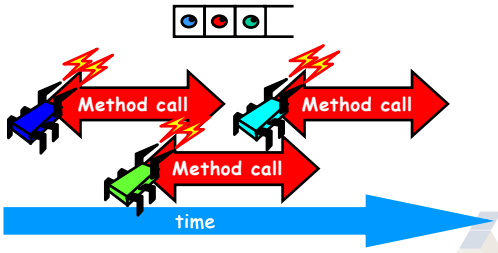
Concurrent Methods Take Overlapping Time



Concurrent Methods Take Overlapping Time



Concurrent Methods Take Overlapping Time



Sequential vs Concurrent

- Sequential:
  - Object needs meaningful state only between method calls
- Concurrent
  - Because method calls overlap, object might never be between method calls

Sequential vs Concurrent

- Sequential:
  - Each method described in isolation
- Concurrent
  - Must characterize all possible interactions with concurrent calls
    - What if two enqs overlap?
    - Two deqs? enq and deq? …

Sequential vs Concurrent

- Sequential:
  - Can add new methods without affecting older methods
- Concurrent:
  - Everything can potentially interact with everything else

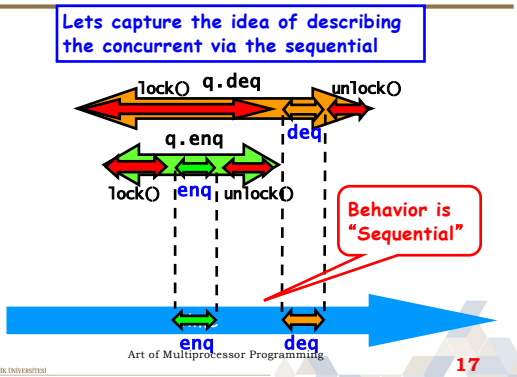
The Big Question

- What does it mean for a concurrent object to be correct?
  - What is a concurrent FIFO queue?
  - FIFO means strict temporal order
  - Concurrent means ambiguous temporal order

Intuitively…

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```


Intuitively



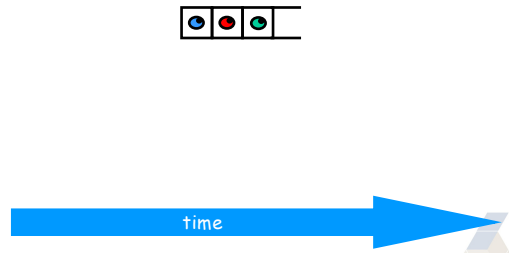
Linearizability

- Each method should
  - "take effect"
  - Instantaneously
  - Between invocation and response events
- Object is correct if this "sequential" behavior is correct
- Any such concurrent object is
  - Linearizable

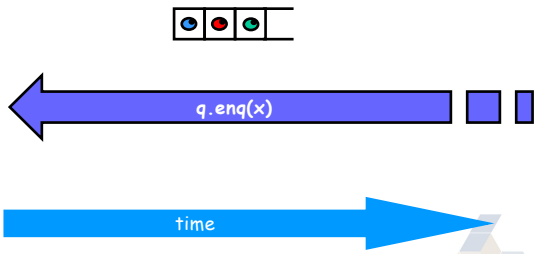
Is it really about the object?

- Each method should
  - "take effect"
  - Instantaneously
  - Between invocation and response events
- Sounds like a property of an execution…
- A linearizable object: one whose all possible executions are linearizable

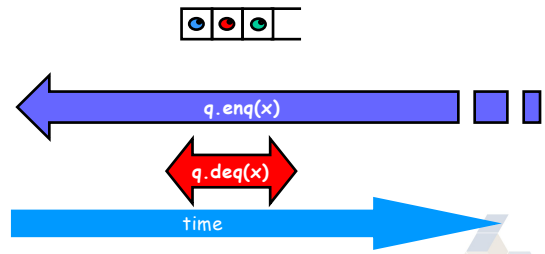
Example



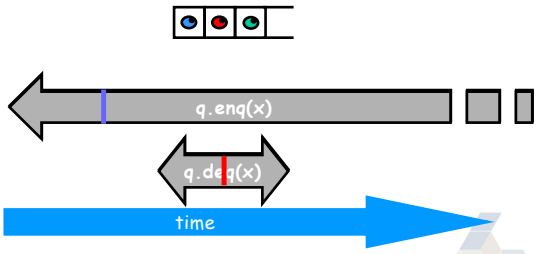
Example



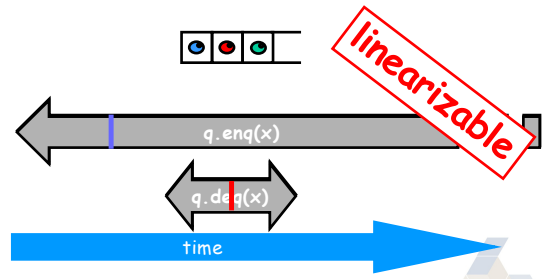
Example



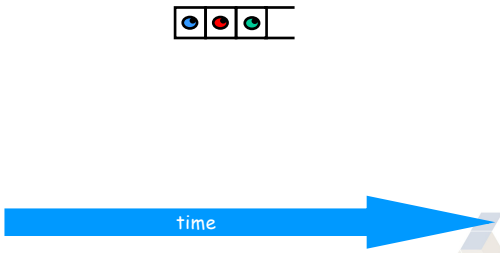
Example



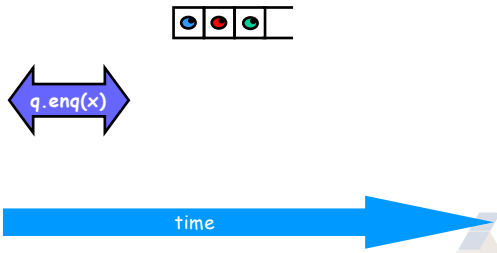
Example



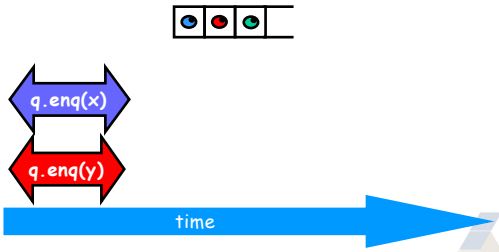
Example



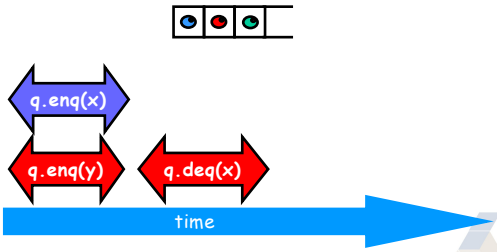
Example



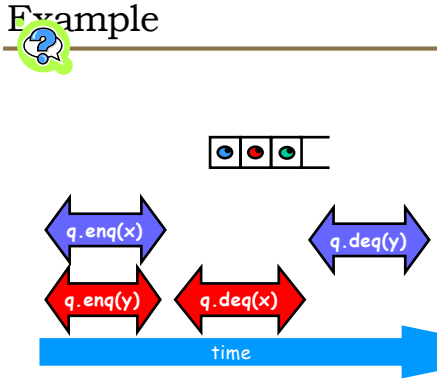
Example



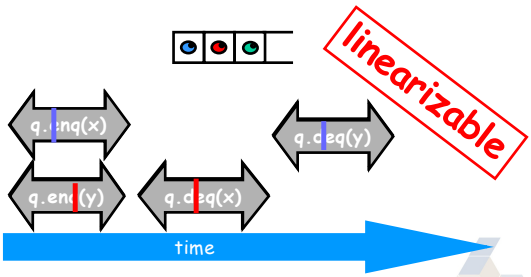
Example



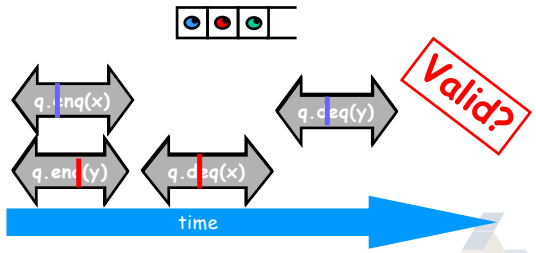
Example



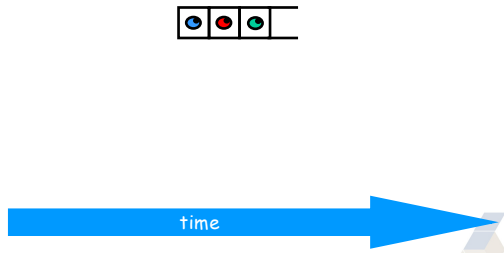
Example



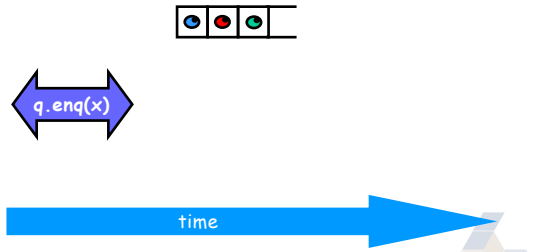
Example



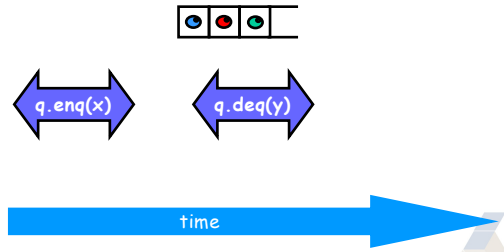
Example



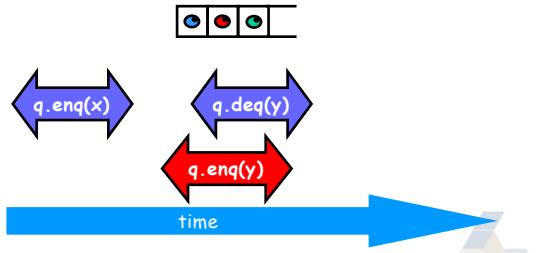
Example



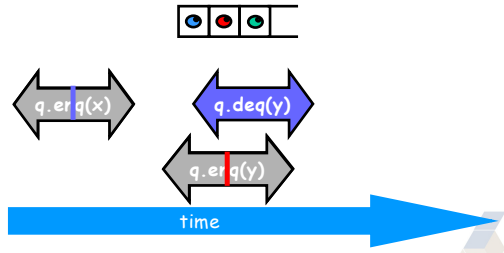
Example



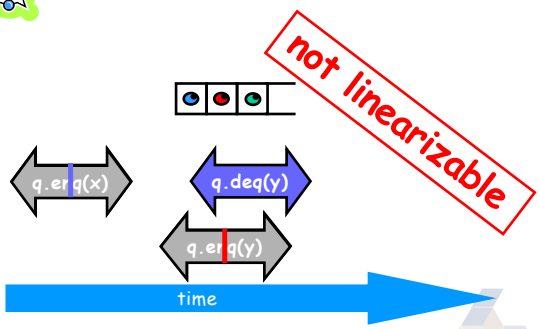
Example



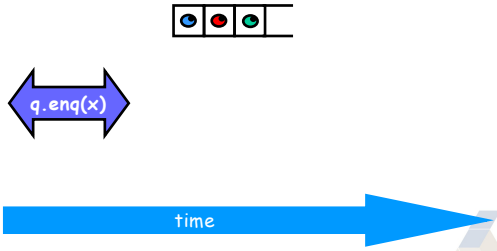
Example



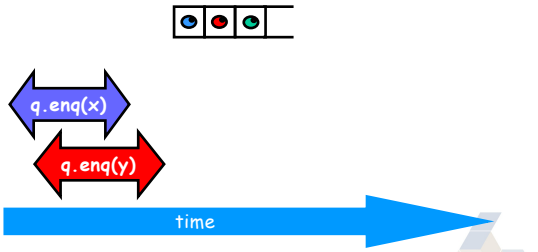
Example



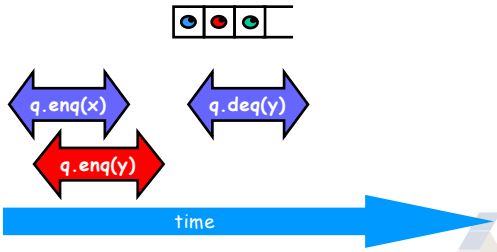
Example



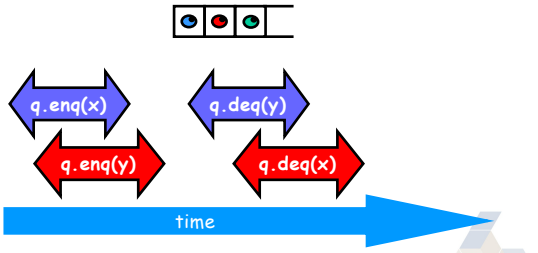
Example



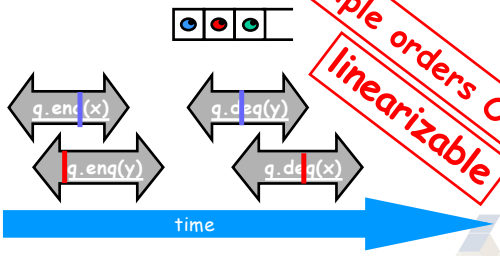
Example



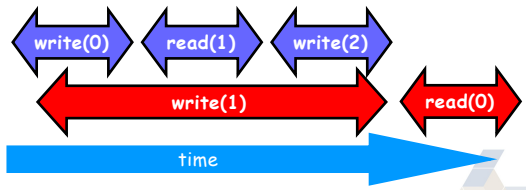
Example



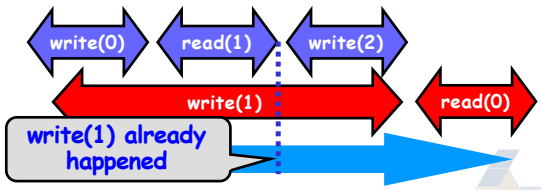
Comme ci Comme ça Example



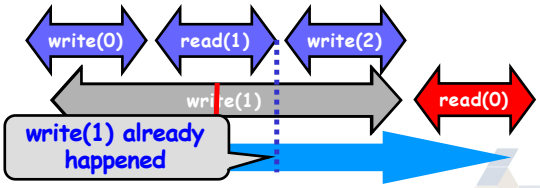
Read/Write Register Example



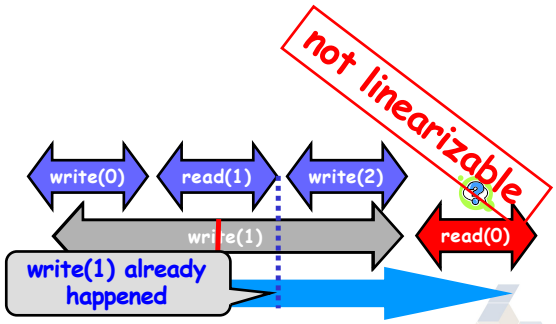
Read/Write Register Example



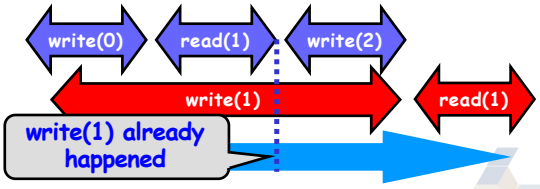
Read/Write Register Example



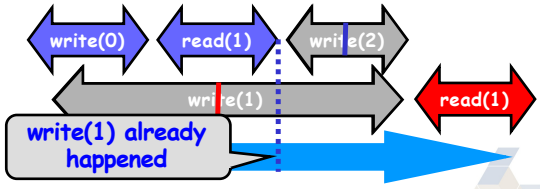
Read/Write Register Example



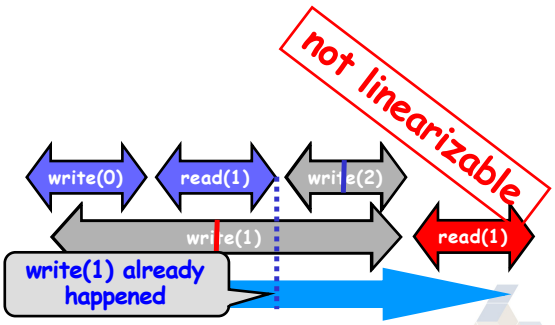
Read/Write Register Example



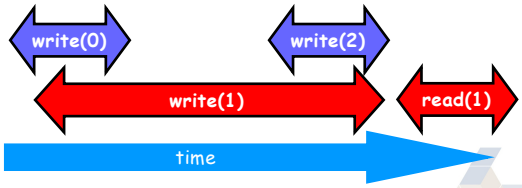
Read/Write Register Example



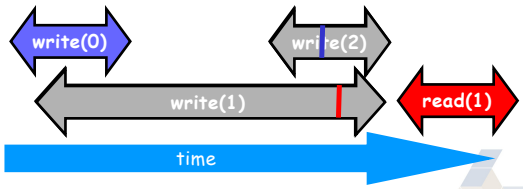
Read/Write Register Example



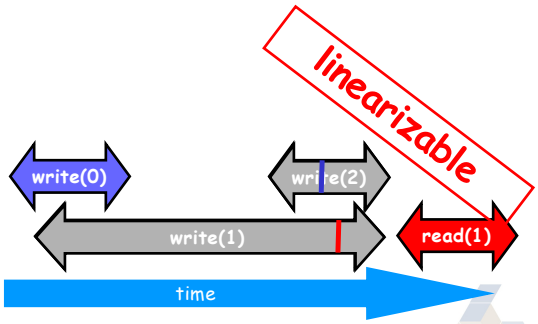
Read/Write Register Example



Read/Write Register Example



Read/Write Register Example



Strategy

- Identify one atomic step where method "happens"
  - Critical section
  - Machine instruction
- Doesn't always work
  - Might need to define several different steps for a given method

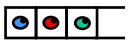
Linearizability: Summary

- Powerful specification tool for shared objects
- Allows us to capture the notion of objects being "atomic"
- Don't leave home without it

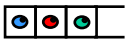
Alternative: Sequential Consistency

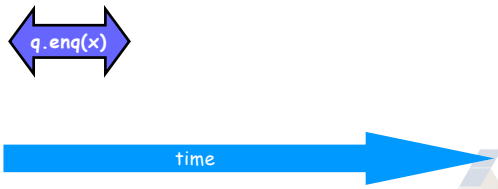
- No need to preserve real-time order
  - Cannot re-order operations done by the same thread
  - Can re-order non-overlapping operations done by different threads
- Often used to describe multiprocessor memory architectures

Example

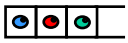


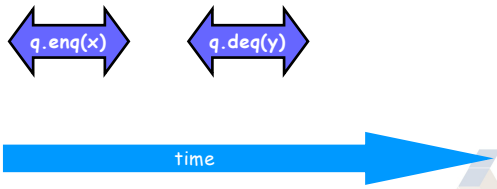
Example



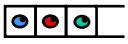


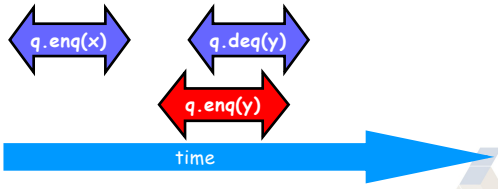
Example



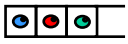


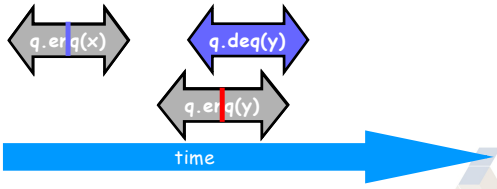
Example



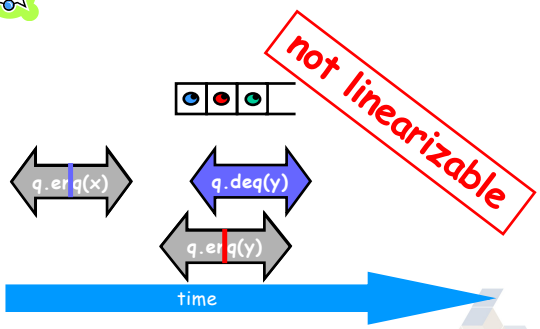


Example

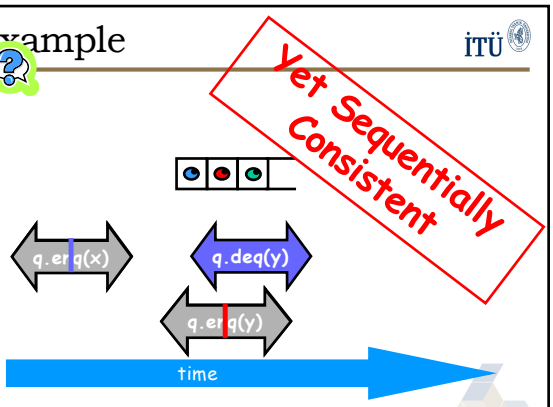




Example



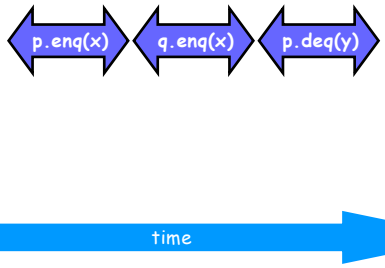
Example



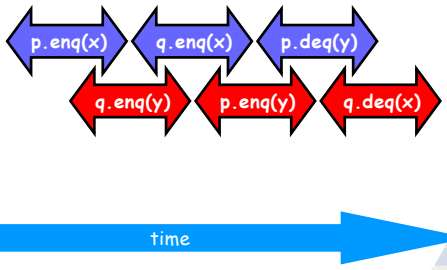
Theorem

- Sequential Consistency is not a composable property

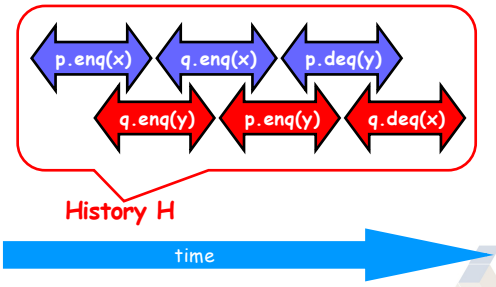
FIFO Queue Example



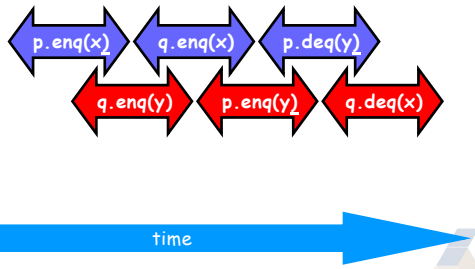
FIFO Queue Example



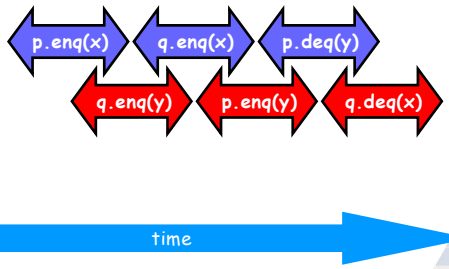
FIFO Queue Example



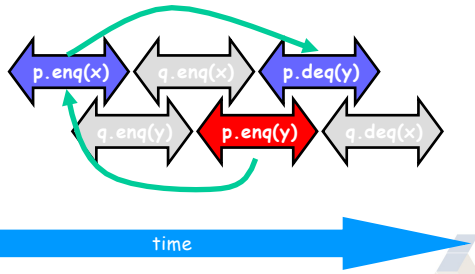
H | p Sequentially Consistent



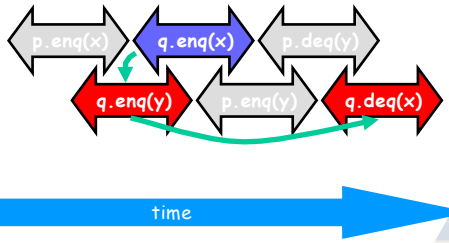
H | q Sequentially Consistent



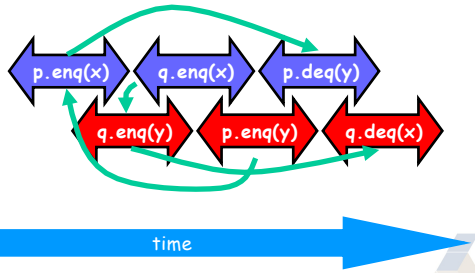
Ordering imposed by p



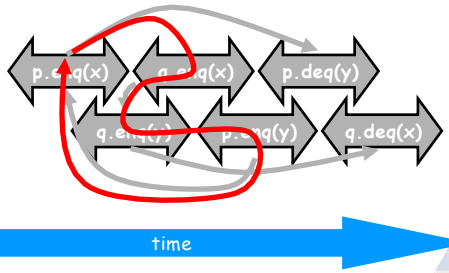
Ordering imposed by q



Ordering imposed by both



Combining orders



Fact

- Most hardware architectures don't support sequential consistency
- Because they think it's too strong

Memory Operations

- To read a memory location,
  - load data into cache.
- To write a memory location
  - update cached copy,
  - Lazily write cached data back to memory

While Writing to Memory

- A processor can execute hundreds, or even thousands of instructions
- Why delay on every memory write?
- Instead, write back in parallel with rest of the program.

Explicit Synchronization

- Memory barrier instruction
  - Flush unwritten caches
  - Bring caches up to date
- Compilers often do this for you
  - Entering and leaving critical sections
- Expensive

Critical Sections

- Easy way to implement linearizability
  - Take sequential object
  - Make each method a critical section
- Problems
  - Blocking
  - No concurrency

Progress Conditions

- Deadlock-free: some thread trying to acquire the lock eventually succeeds.
- Starvation-free: every thread trying to acquire the lock eventually succeeds.
- Lock-free: some thread calling a method eventually returns.
- Wait-free: every thread calling a method eventually returns.

Progress Conditions

	Non-Blocking	Blocking
Everyone makes progress	Wait-free	Starvation-free
Someone makes progress	Lock-free	Deadlock-free

Art of Multiprocessor Programming

Concurrent Registers

Registers

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


Registers

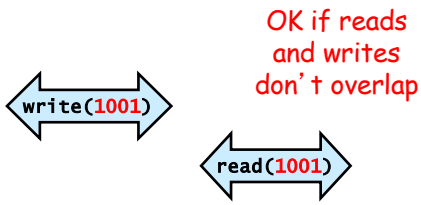
```
public interface Register<T> {
  public T read();
  public void write(T v);
}
```

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

Jargon Watch

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

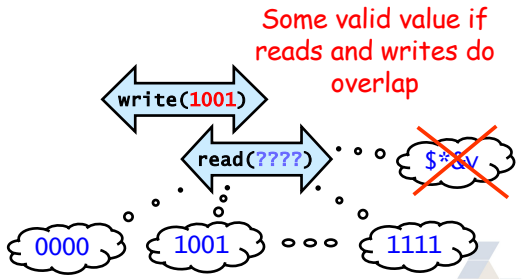
Safe Register



OK if reads and writes don't overlap

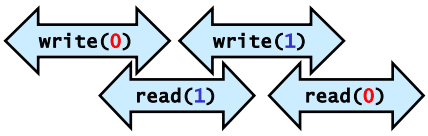
(2)

Safe Register

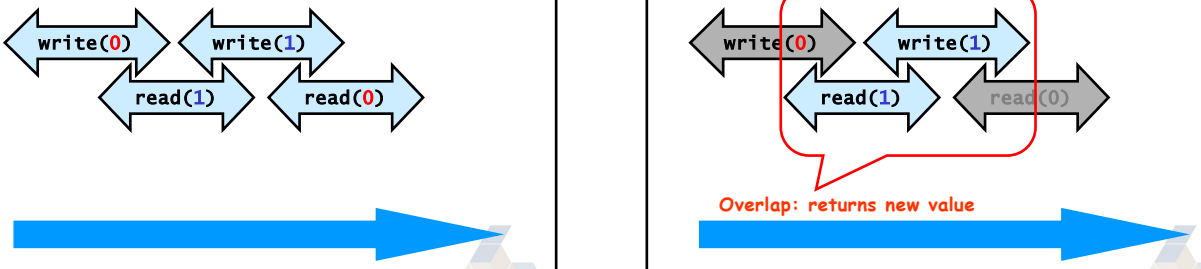


Regular Register

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

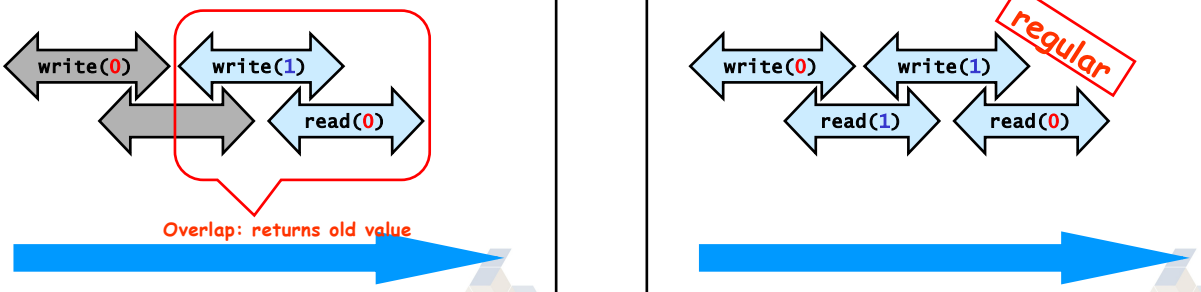


Regular or Not?



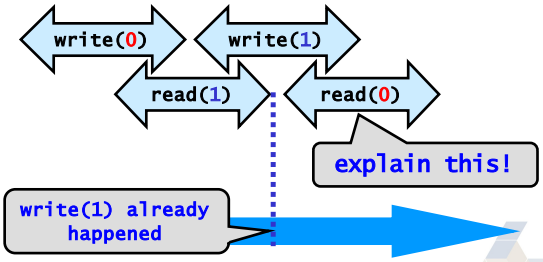
Regular or Not?

Regular or Not?

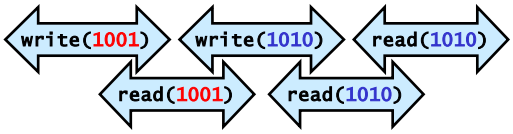


Regular or Not?

Regular ≠ Atomic

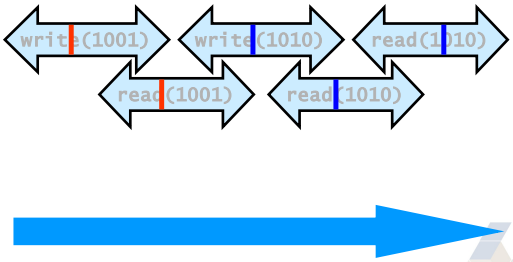


Atomic Register

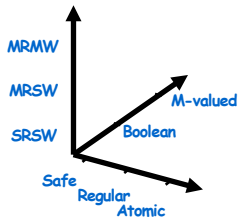


Linearizable to sequential safe register

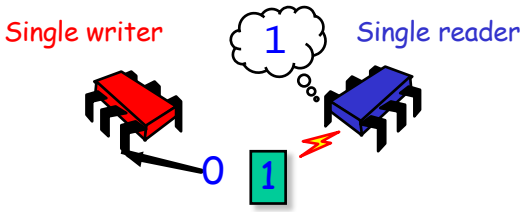
Atomic Register



Register Space



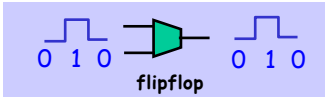
Weakest Register



Safe Boolean register

Weakest Register

Single writer Single reader



Get correct reading if not during state transition

Results

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

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!

Road Map

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

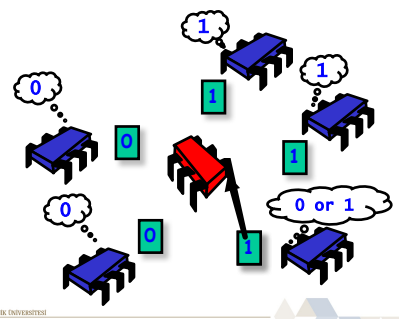
Road Map

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

Register Names

```
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

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


Safe Boolean MRSW from Safe Boolean SRSW

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

Each thread has own safe SRSW register

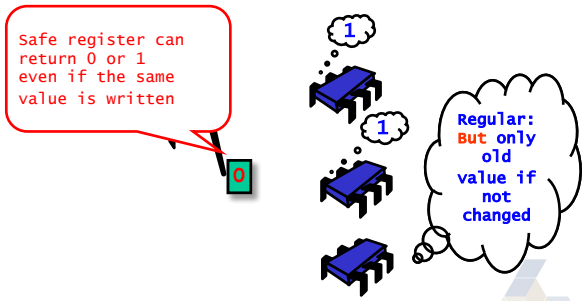
(2)

Road Map

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

Next

Regular Boolean MRSW from Safe Boolean MRSW



Regular Boolean MRSW from Safe Boolean MRSW

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


Regular Boolean MRSW from Safe Boolean MRSW

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

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

Regular Boolean MRSW from Safe Boolean MRSW

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

Actual value

Regular Boolean MRSW from Safe Boolean MRSW

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

Is new value different from last value I wrote?

Regular Boolean MRSW from Safe Boolean MRSW

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

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

Regular Boolean MRSW from Safe Boolean MRSW

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

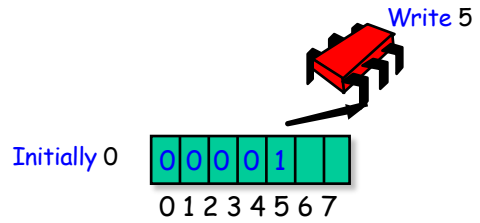
- Overlap? No Overlap?
  - No problem
  - either Boolean value works

Road Map

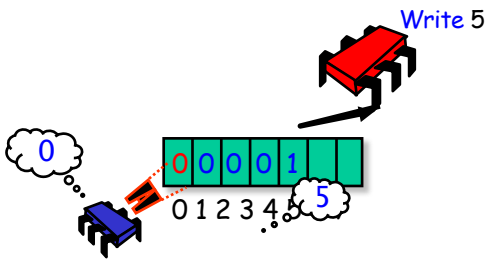
- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

Next

Writing M-Valued



Writing M-Valued



MRSW Regular M-valued from MRSW Regular Boolean

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

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

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


MRSW Regular M-valued from MRSW Regular Boolean

```
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

MRSW Regular M-valued from MRSW Regular Boolean

```
public class RegMRSWRegisterimplements 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;
  }}
```

Set bit x

(1)

MRSW Regular M-valued from MRSW Regular Boolean

```
public class RegMRSWRegisterimplements 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;
  }}
```

Clear bits from higher to lower

(1)

MRSW Regular M-valued from MRSW Regular Boolean

```
public class RegMRSWRegisterimplements 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;
  }}
```

Scan from lower to higher & return first bit set

(1)

Road Map

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

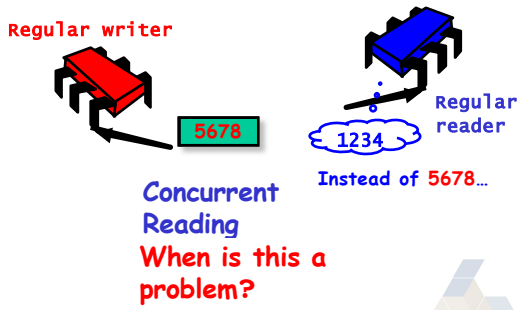
Next

Road Map (Slight Detour)

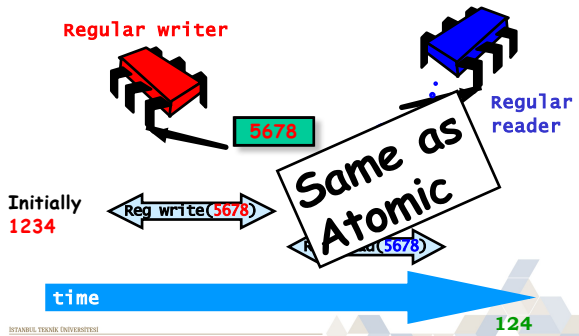
- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic

SRSW Atomic

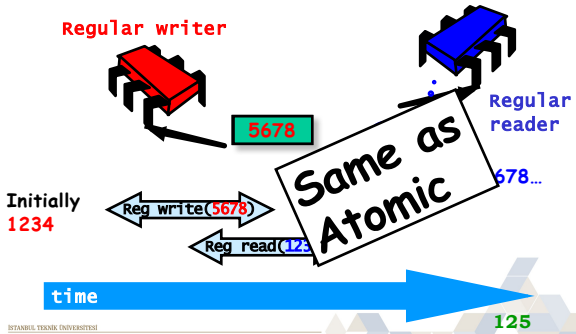
SRSW Atomic From SRSW Regular



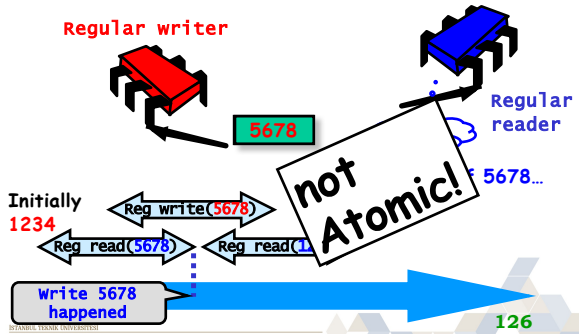
SRSW Atomic From SRSW Regular



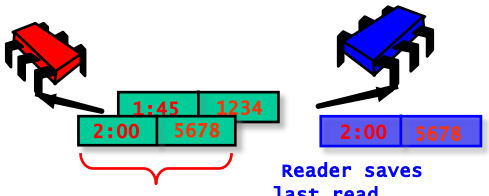
SRSW Atomic From SRSW Regular



SRSW Atomic From SRSW Regular



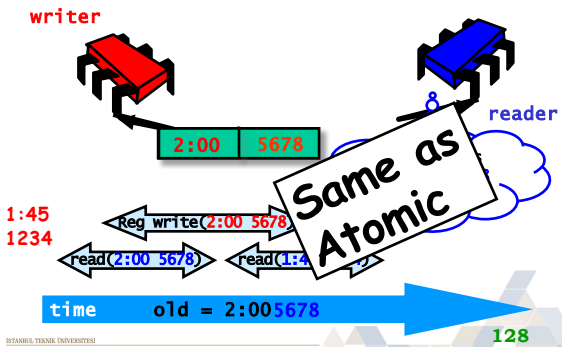
Timestamped Values



Writer writes value and stamp together

Reader saves last read (value,stamp) and returns new value only if higher stamp

SRSW Atomic From SRSW Regular

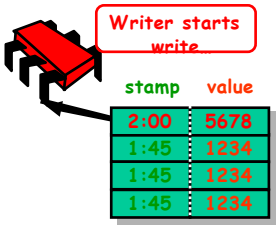


Atomic Single Reader to Atomic Multi-Reader

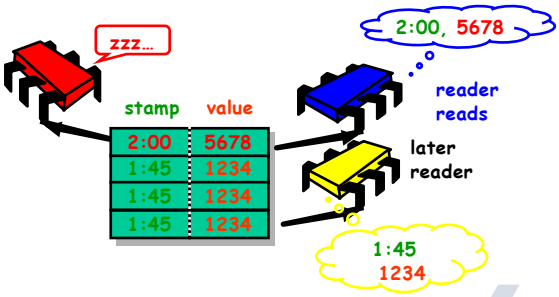
stamp	value
1:45	1234
1:45	1234
1:45	1234
1:45	1234

One per reader

Another Scenario



Another Scenario



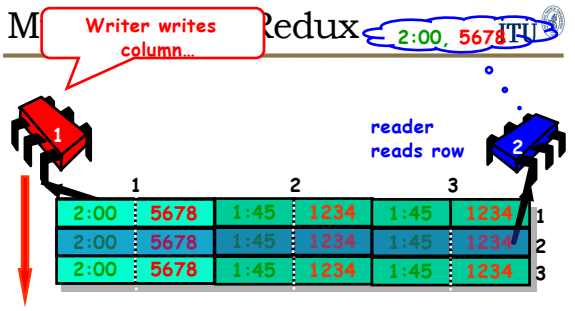
Yellow was completely after blue but read earlier value, not linearizable!

Multi-Reader Redux

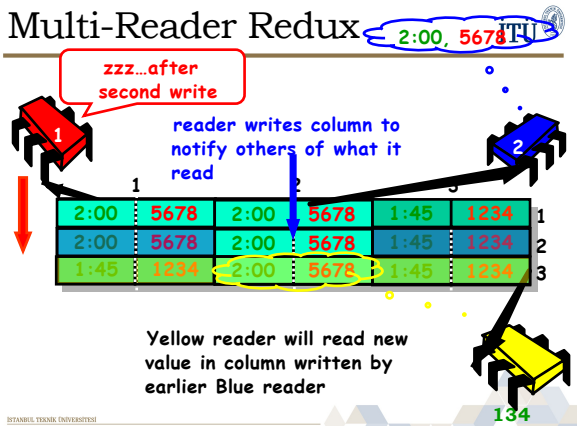
One per thread

1		2		3		
1:45	1234	1:45	1234	1:45	1234	1
1:45	1234	1:45	1234	1:45	1234	2
1:45	1234	1:45	1234	1:45	1234	3

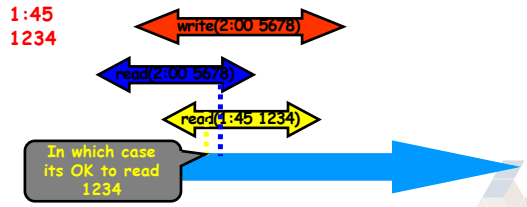
M Redux



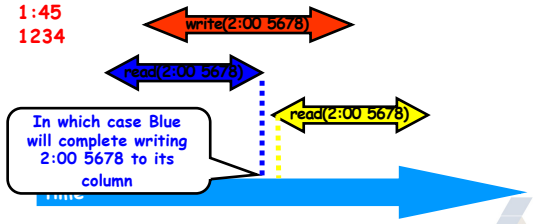
Multi-Reader Redux



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



Bad Case Only When Readers Don't Overlap

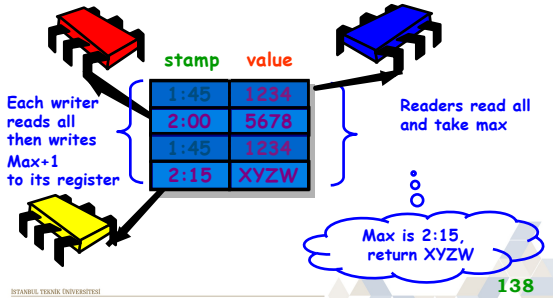


Road Map

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

Next

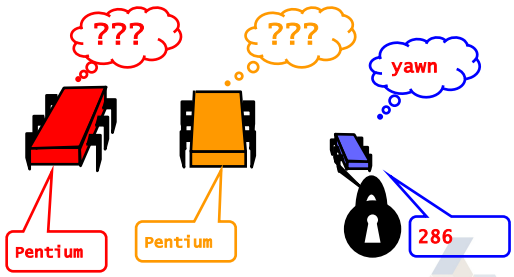
Multi-Writer Atomic From Multi-Reader Atomic



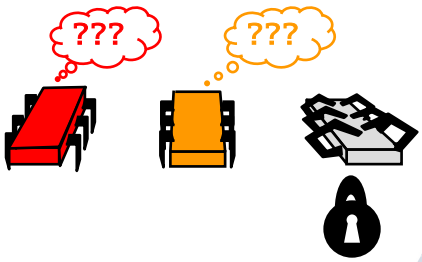
Consensus Values

Why is Mutual Exclusion so wrong?

Heterogeneous Processors



Fault-tolerance

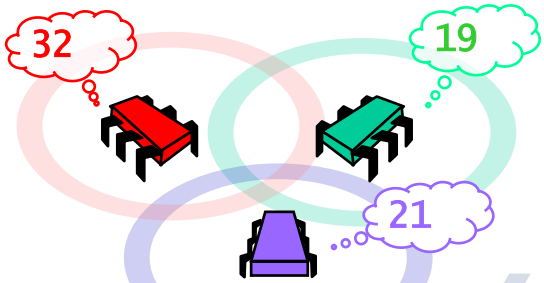


(2)

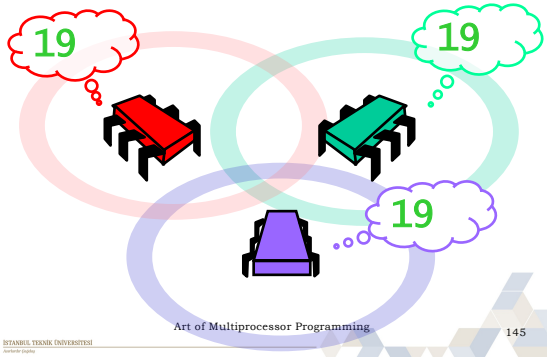
Basic Questions

- Wait-Free synchronization might be a good idea in principle
- But how do you do it
  - Systematically?
  - Correctly?
  - Efficiently?

Consensus: Each Thread has a Private Input



They Agree on One Thread's Input



Formally: Consensus

Consistent: all threads decide the same value

Valid: the common decision value is some thread's input

Consensus Object

```
public interface Consensus {
 Object decide(Object value);
}
```

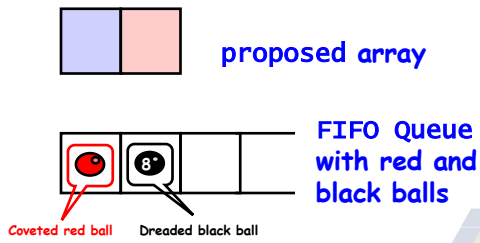

Java Jargon Watch

- Define Consensus protocol as an abstract class
- We implement some methods

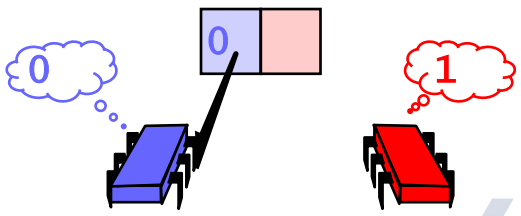
Generic Consensus Protocol

```
abstract class ConsensusProtocol implements Consensus {

 protected Object[] proposed = new Object[N];

 protected void propose(Object value) {
  proposed[ThreadID.get()] = value;
 }

 abstract public Object decide(Object value);

}
```

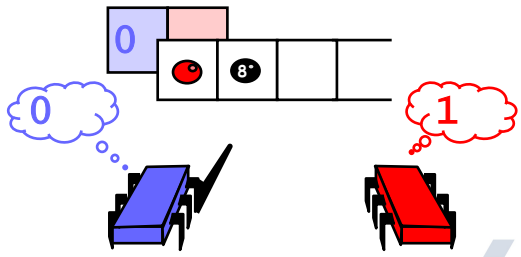

FIFO Consensus



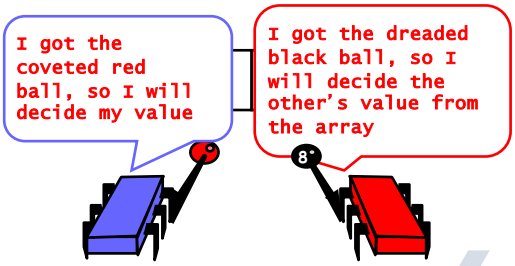
Protocol: Write Value to Array



Protocol: Take Next Item from Queue



Protocol: Take Next Item from Queue



Consensus Using FIFO Queue

```
public class QueueConsensus
                    extends ConsensusProtocol {
 private Queue queue;
 public QueueConsensus() {
  queue = new Queue();
  queue.enq(Ball.RED);
  queue.enq(Ball.BLACK);
 }
 …
}
```


Who Won?

```
public class QueueConsensus
                    extends ConsensusProtocol {
 private Queue queue;
 …
 public decide(object value) {
  propose(value);
  Ball ball = this.queue.deq();
  if (ball == Ball.RED)
     return proposed[i];
  else
     return proposed[1-i];
 }
}
```


Why does this Work?

- If one thread gets the red ball
- Then the other gets the black ball
- Winner decides her own value
- Loser can find winner's value in array
  - Because threads write array
  - Before dequeueing from queue

Consensus Numbers

- An object X has consensus number n
  - If it can be used to solve n-thread consensus
    - Take any number of instances of X
    - together with atomic read/write registers
    - and implement n-thread consensus
  - But not (n+1)-thread consensus

Consensus Numbers

- Theorem
  - Atomic read/write registers have consensus number 1
- Theorem
  - Multi-dequeuer FIFO queues have consensus number 2

Consensus Numbers Measure Synchronization Power

- Theorem
  - If you can implement X from Y
  - And X has consensus number c
  - Then Y has consensus number at least c

Synchronization Speed Limit

- Conversely
  - If X has consensus number c
  - And Y has consensus number d < c
  - Then there is no way to construct a wait-free implementation of X by Y

Read-Modify-Write Objects

- Method call
  - Returns object's prior value **x**
  - Replaces **x** with **mumble(x)**

RMW Everywhere!

- Most synchronization instructions
  - are RMW methods
- The rest
  - Can be trivially transformed into RMW methods

Example: getAndSet

```
public abstract class RMWRegister {
 private int value;

 public int synchronized getAndSet(int v)
 {
   int prior  = this.value;
   this.value = v;
   return prior;
 }
 …
}
```


getAndIncrement

```
public abstract class RMWRegister {
 private int value;

 public int synchronized getAndIncrement()
 {
   int prior  = this.value;
   this.value = this.value + 1;
   return prior;
 }
 …
}
```


getAndAdd

```
public abstract class RMWRegister {
 private int value;

 public int synchronized getAndAdd(int a)
 {
   int prior  = this.value;
   this.value = this.value + a;
   return prior;
 }
 …
}
```


compareAndSet

```
public abstract class RMWRegister {
 private int value;
 public boolean synchronized compareAndSet(
                 int expected, int update) {

    int prior = this.value;
    if (this.value==expected) {
         this.value = update;
         return true;
    }
  return false;
  } …
}
```


compareAndSet Has ∞ Consensus Number

```
public class RMWConsensus extends ConsensusProtocol {

 private AtomicInteger r = new AtomicInteger(-1);
 public Object decide(object value) {
   propose(value);
   r.compareAndSet(-1,i);
   return proposed[r.get()];
 }
}
```


The Consensus Hierarchy

1 Read/Write Registers, Snapshots…
2 getAndSet, getAndIncrement, …
. . .
∞ compareAndSet,…

Atomic variables

- `java.util.concurrency.atomic` contains classes representing scalars supporting "CAS"

  `boolean compareAndSet(expectedV, newV)`
  - Atomically set to new value if holding expected value
  - Always used in a loop
- Essential for writing efficient code on MPs
  - Nonblocking data structures, optimistic algorithms, reducing overhead and contention when updates center on a single field
- JVMs use best construct available on platform
  - Compare-and-swap, Load-linked/Store-conditional, Locks

Atomic Variables

- Java concurrency package also supplies reflection-based classes that allow CAS on given volatile fields of other classes
- Also provides methods for getting and unconditionally setting values
- Instance of classes AtomicInteger, AtomicBoolean, AtomicLong, AtomicReference provide access and update to single variables of that type

Atomic Variable Example

- Faster and less contention in programs with a single `Random` accessed by many threads
- Let's see `java.util.Random`

```
class Random {        // snippets
  private AtomicLong seed;
  Random(long s) {
    seed = new AtomicLong(s);
  }
  long next(){
    for(;;) {
      long s = seed.get();
      long nexts = s * ... + ...;
      if (seed.compareAndSet(s,nexts))
        return s;
    }
  } }
```