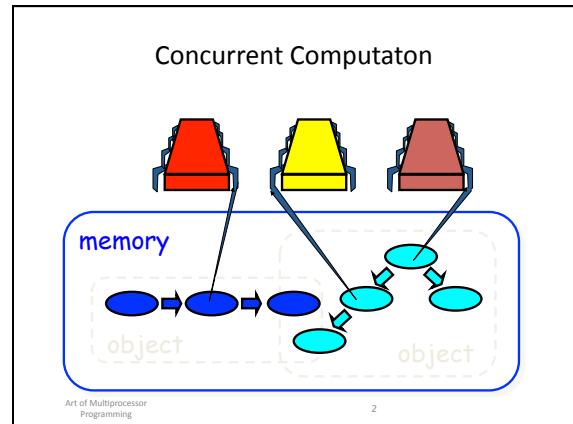


DD245 I
 Parallel and Distributed Computing

FDD3008
 Distributed Algorithms

 Lecture 2
 Concurrent Objects

 Mads Dam
 Autumn/Winter 2011



Objects

Objects have **state**

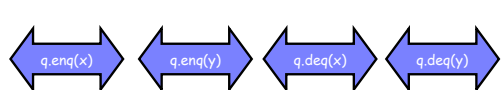
- Usually given by a set of **fields**

Object have **methods**

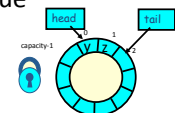
- Only way to manipulate state

Example, a FIFO queue:

- Fields: Sequence of items
- Methods: enq and deq



A Concurrent FIFO Queue

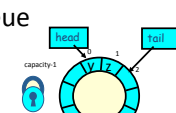


```

class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
  }
}
  
```

4

A Concurrent FIFO Queue



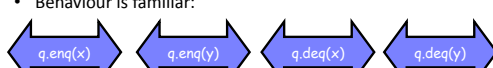
```

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();
  }
}
  
```

5

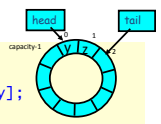
Is The Queue Correct?

- Calls are effectively sequentialized
- Behaviour is familiar:



- (In fact the calls overlap slightly but we ignore that for now)
- Now let's try the same thing without mutual exclusion
- For simplicity, only two threads
 - One thread **enq only**
 - The producer thread
 - The other **deq only**
 - The consumer thread

Wait-free 2-Thread Queue



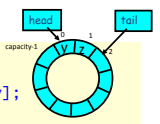
```

public class WaitFreeQueue {
    int head = 0, tail = 0;
    items = (T[]) new Object[capacity];

    public void enq(Item x) {
        while (tail-head == capacity); // busy-wait
        items[tail % capacity] = x; tail++;
    }
    public Item deq() {
        while (tail == head); // busy-wait
        Item item = items[head % capacity]; head++;
        return item;
    }
}
    
```

Art of Multiprocessor Programming

Wait-free 2-Thread Queue



```

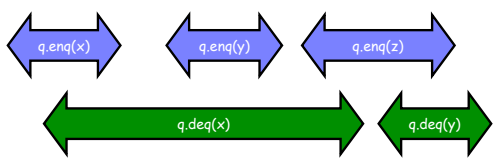
public class WaitFreeQueue {
    int head = 0, tail = 0;
    items = (T[]) new Object[capacity];

    public void enq(Item x) {
        while (tail-head == capacity); // busy-wait
        items[tail % capacity] = x; tail++;
    }
    public Item deq() {
        while (tail == head); // busy-wait
        Item item = items[head % capacity]; head++;
        return item;
    }
}
    
```

Ooops: No locks!

Is the Wait-Free Queue Correct?

The behaviour of the queue may now look something like this:



Do we want this behaviour? Or not? Specify!
 Need to be able to answer this question
 Need to know when an implementation is correct

Sequential Specifications

- If (precondition)
 - the object is in such-and-such a state
 - before you call the method,
- Then (postcondition)
 - the method will return a particular value
 - or throw a particular exception.
- and (postcondition, cont'd)
 - the object will be in some other state
 - when the method returns.

Art of Multiprocessor Programming

Pre and PostConditions for Deq

<ul style="list-style-type: none"> • Precondition: <ul style="list-style-type: none"> – Queue is non-empty • Postcondition: <ul style="list-style-type: none"> – Returns first item in queue • Postcondition: <ul style="list-style-type: none"> – Removes first item in queue 	<ul style="list-style-type: none"> • Precondition: <ul style="list-style-type: none"> – Queue is empty • Postcondition: <ul style="list-style-type: none"> – Throws Empty exception • Postcondition: <ul style="list-style-type: none"> – Queue state unchanged
---	--

Art of Multiprocessor Programming

Why Sequential Specifications Totally Rock

- Interactions among methods captured by side-effects on object state
 - State meaningful between method calls
 - Elsewhere it does not matter
- Documentation size linear in number of methods
 - Each method described in isolation
- Can add new methods
 - Without changing descriptions of old methods

Art of Multiprocessor Programming

What About Concurrent Specifications ?

- Method state potentially matters everywhere
 - Method calls may interfere because of shared variable accesses
 - So intermediate states matter
- Documentation size?
 - In worst case the only meaningful specification is the method itself
- Can add new methods?
 - In worst case only by exploring all possible interactions with existing methods
- Is there any rescue from this morass?

Sequential vs Concurrent

Sequential:

- Methods take time
- Sure, but their *effect* might as well be instantaneous
- Objects need meaningful state only between method calls

Concurrent:

- Method call is not an event
- Method call is an interval
- Since method calls overlap object might never be between method calls

Sequential vs Concurrent

Sequential:

- Each method described in isolation
- Can add new method without affecting old ones

Concurrent:

- Must characterize all possible interactions between concurrent calls
 - What if two deq's overlap? Two enq's? One enq and one deq? Etc? When and how do they overlap?
- Everything can potentially interact with everything else

Sequential vs Concurrent

Sequential:

- Each method described in isolation
- Can add new method without affecting old ones

Concurrent:

- Must characterize all possible interactions between concurrent calls
 - What if two deq's overlap? Two enq's? One enq and one deq? Etc? When and how do they overlap?
- Everything can potentially interact with everything else

Panic!

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();
    }
}
    
```

Art of Multiprocessor Programming 17

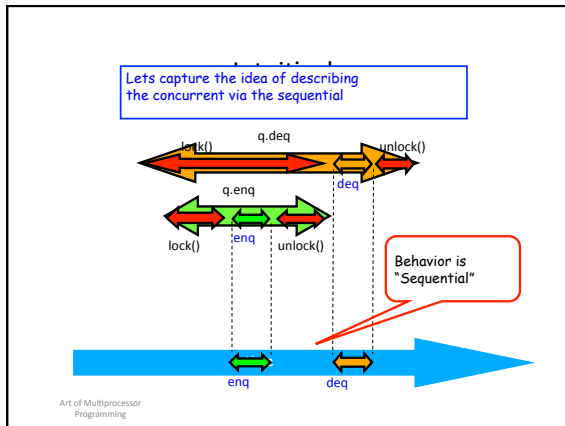
Intuitively...

```

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();
    }
}
    
```

All modifications of queue are done mutually exclusive

Art of Multiprocessor Programming



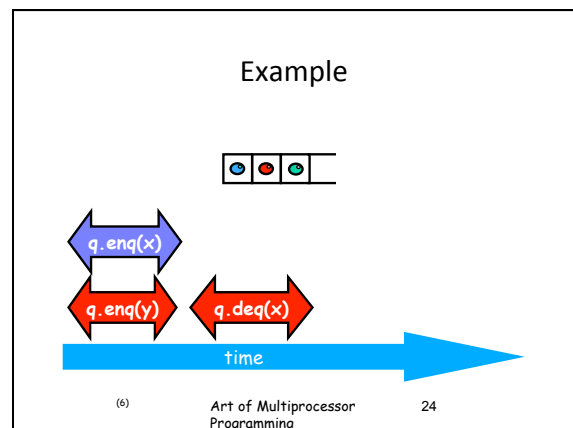
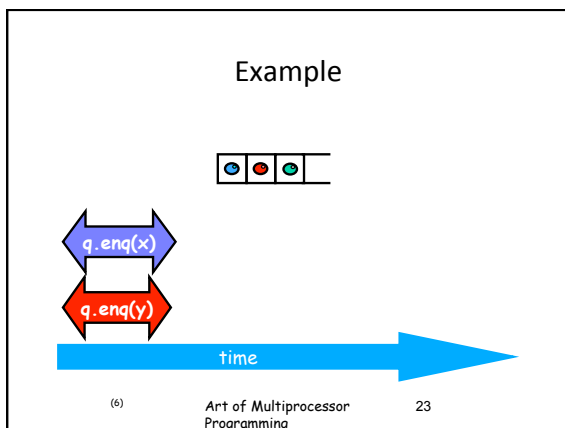
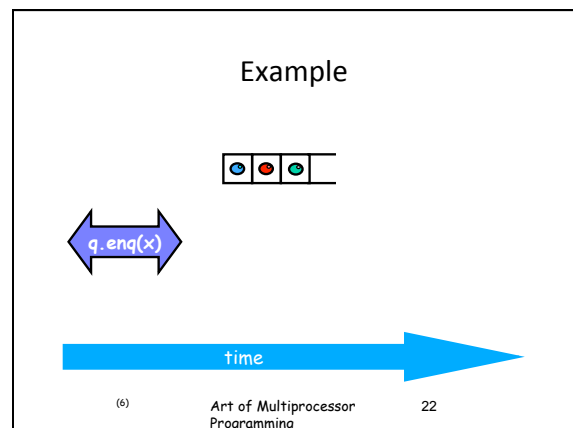
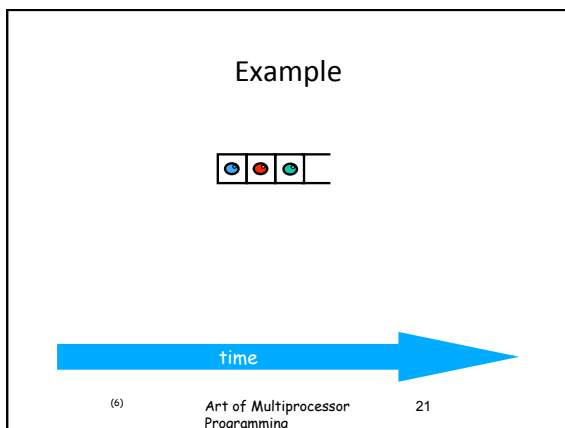
Linearizability

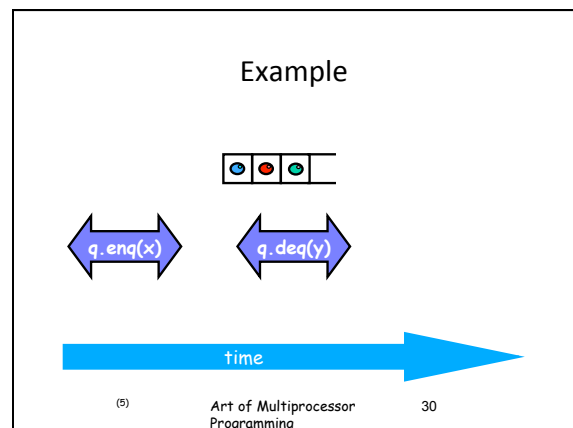
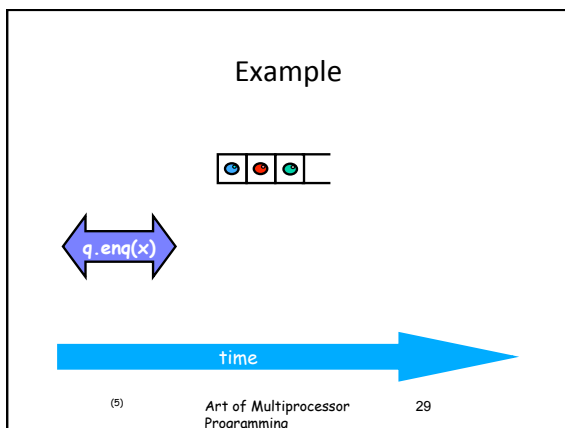
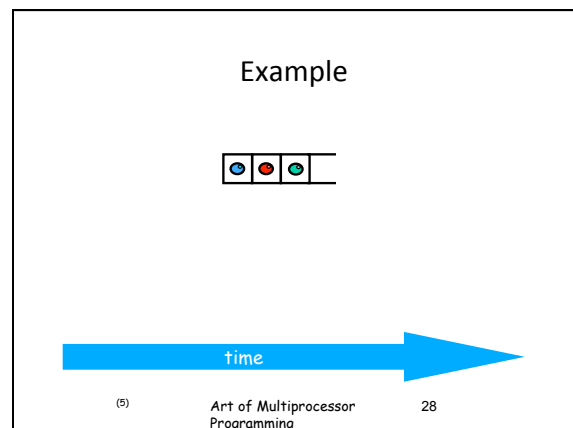
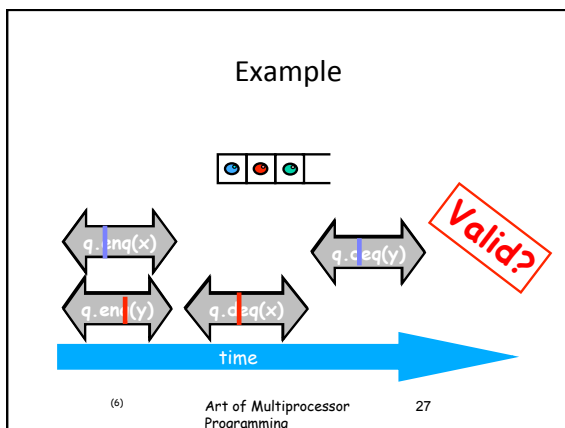
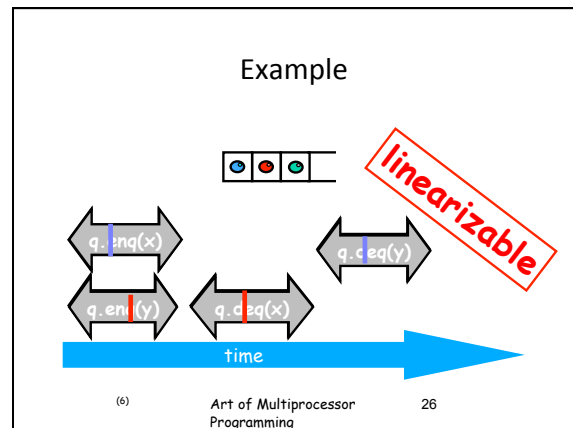
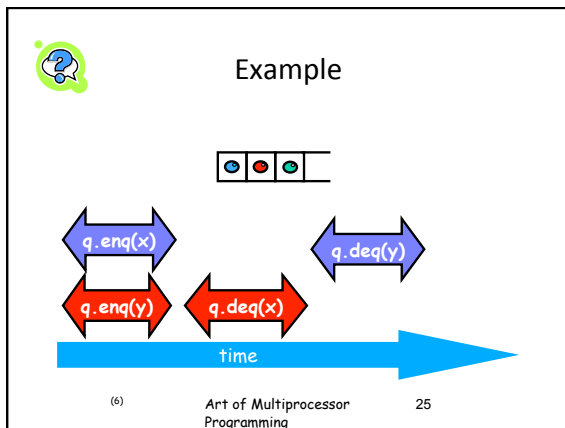
The effect of each method is

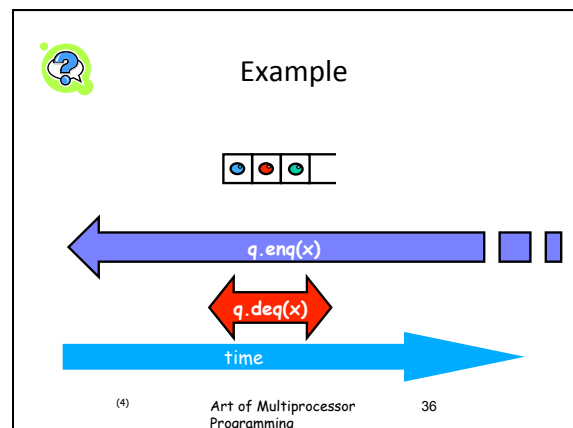
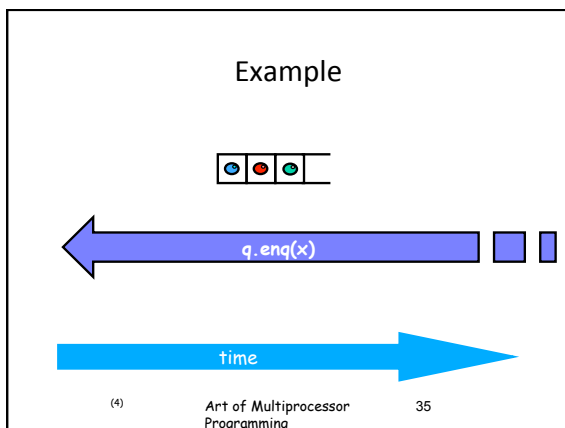
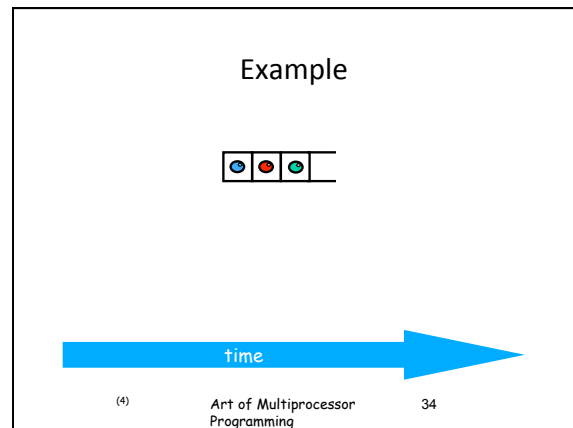
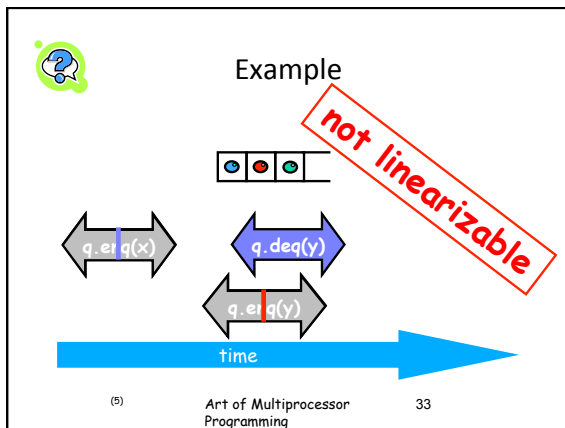
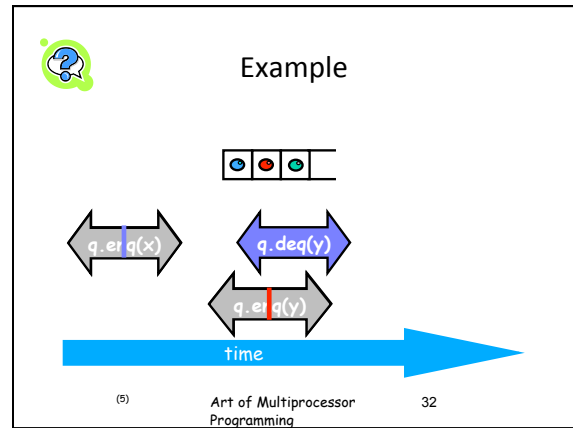
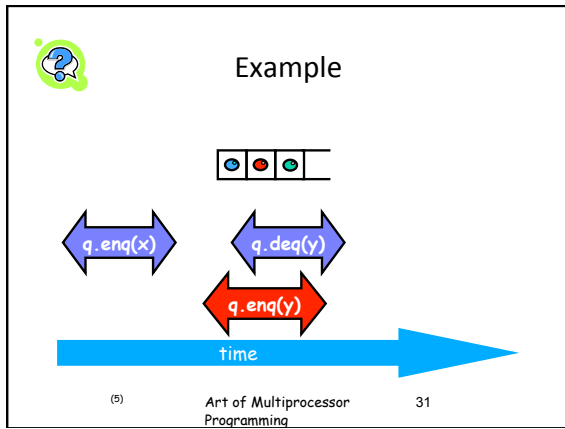
- Instantaneous
- Takes place inbetween invocation and response events
- Methods can use locks to achieve this, or not
- Call such an execution linearizable

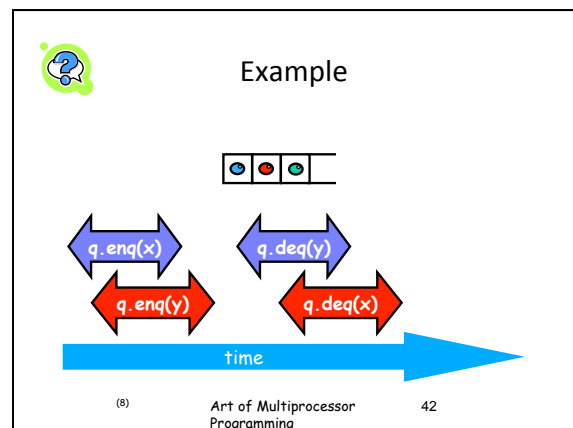
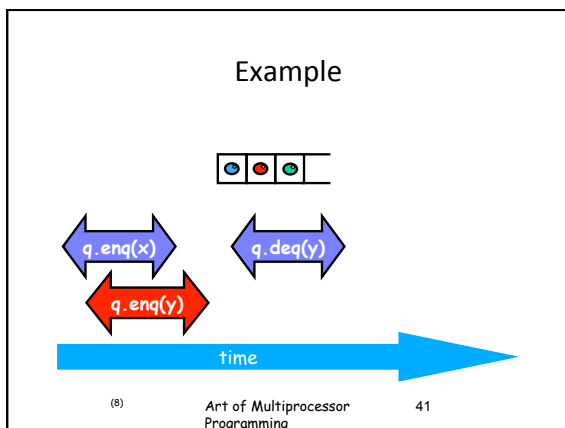
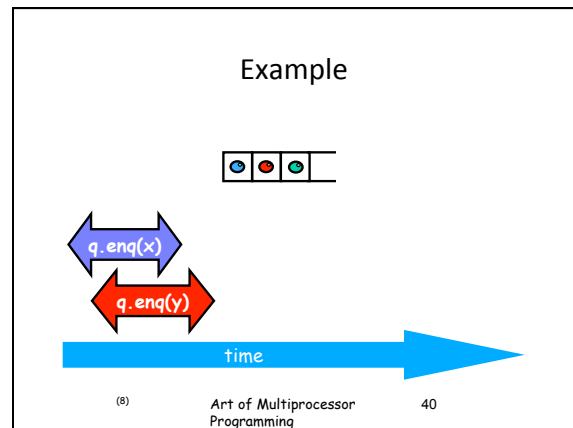
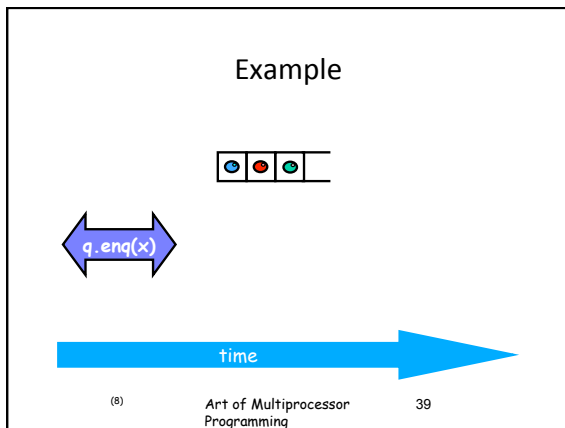
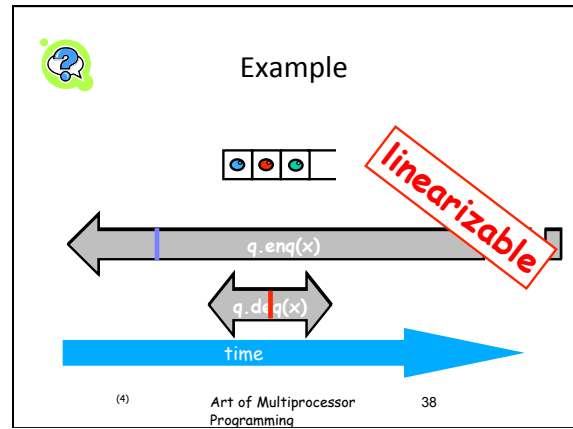
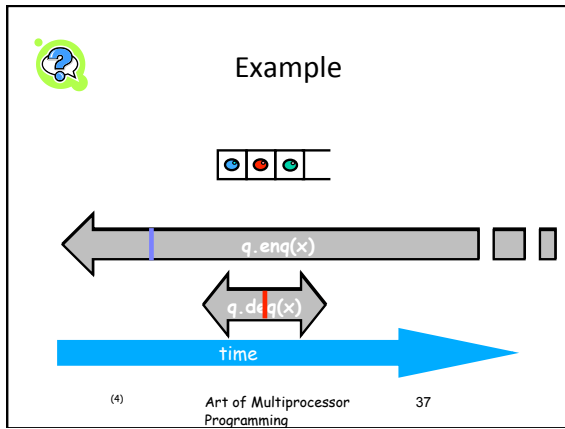
Linearizable object:

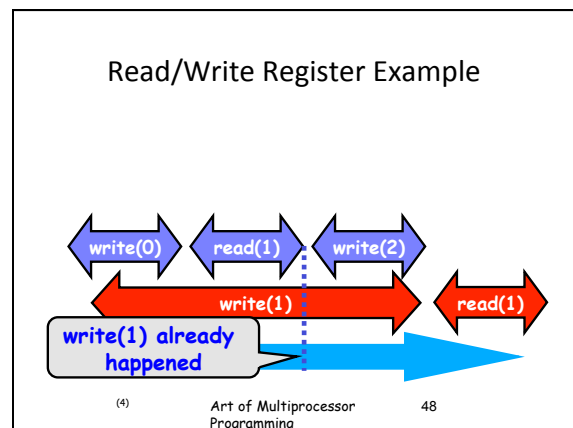
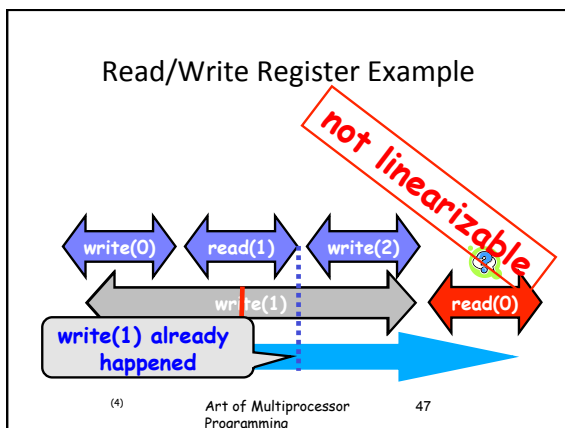
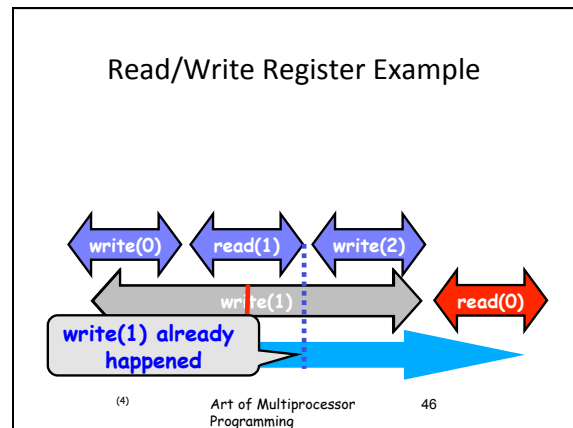
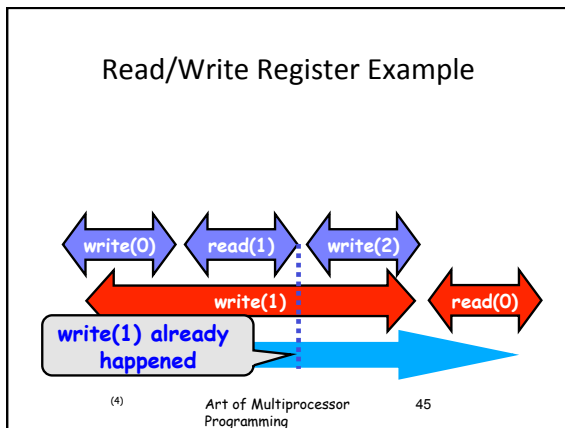
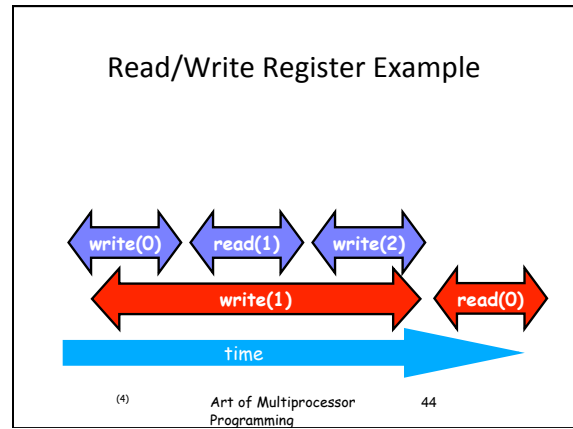
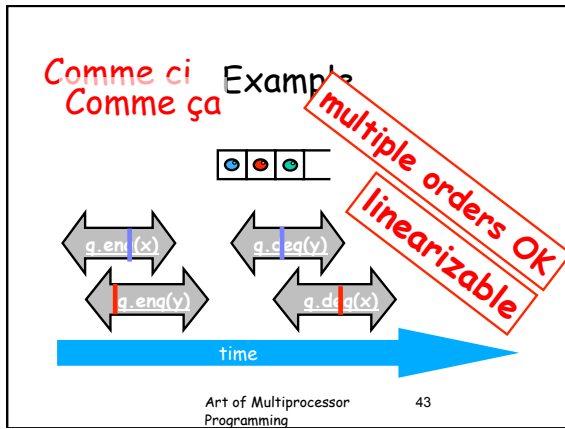
- Each execution is linearizable
- More formal definition to come ...

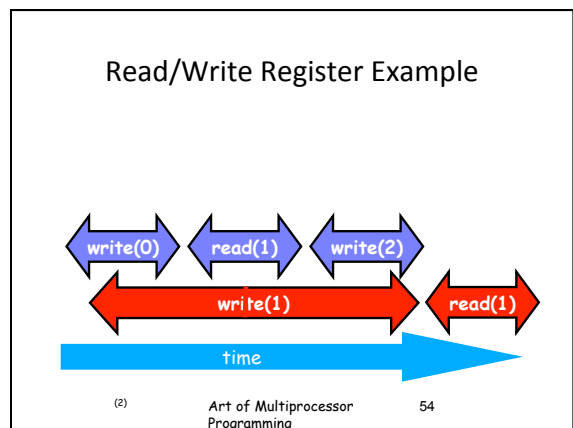
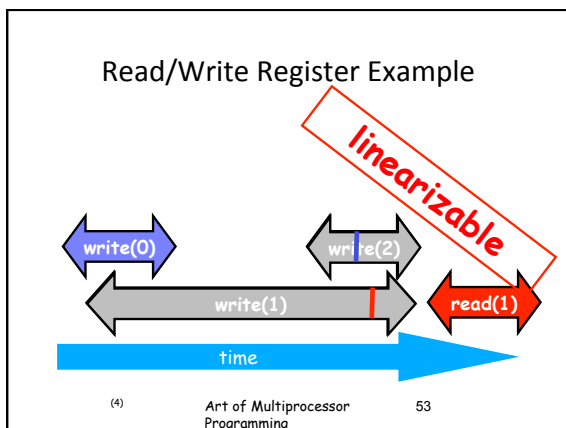
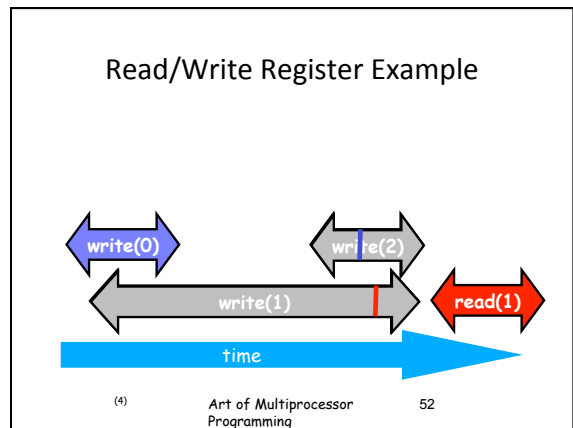
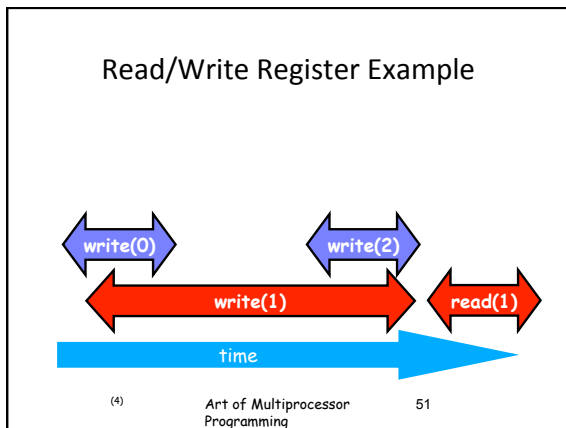
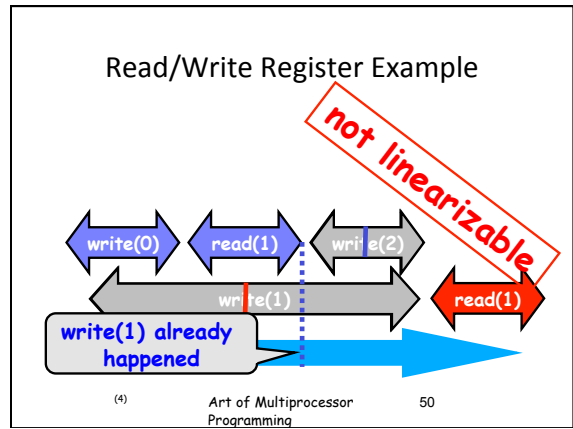
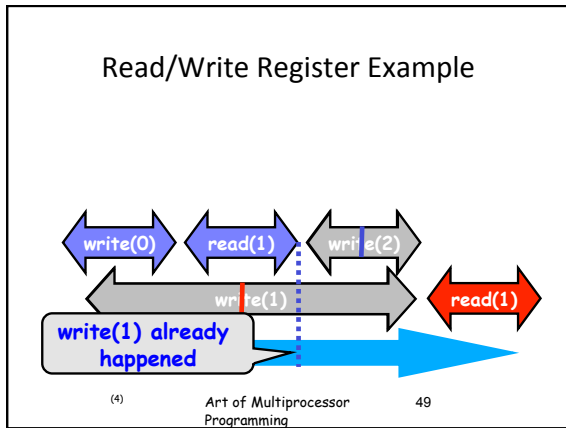


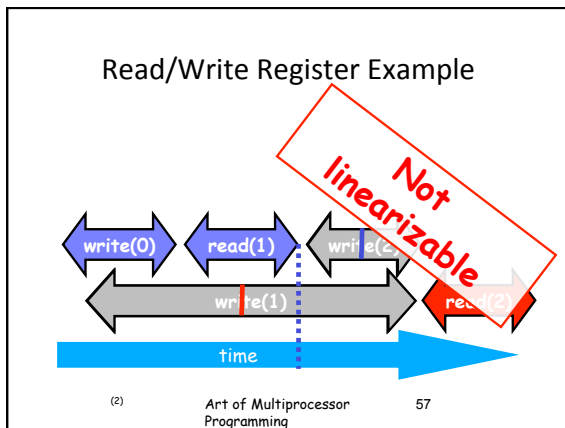
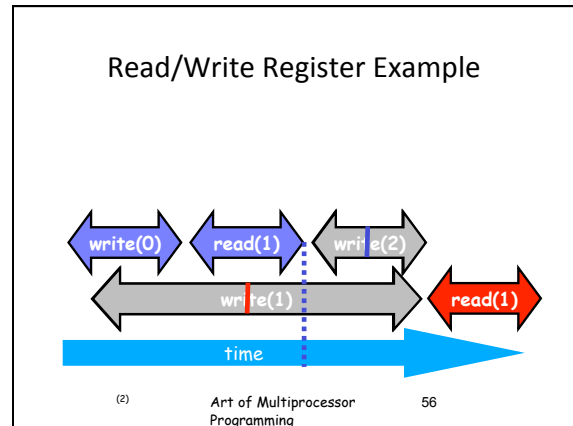
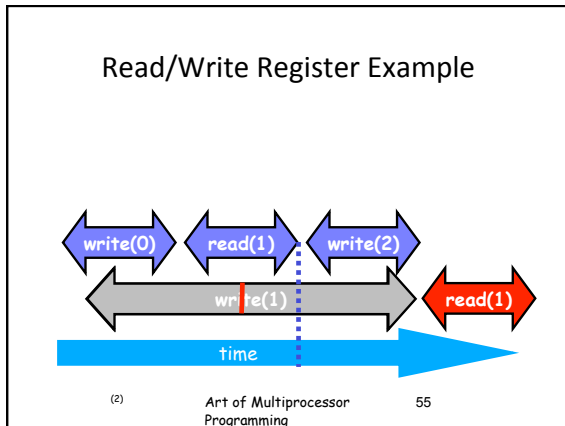












- ### Formal Model
- Goal: Build formal model of computations on shared objects
 - So that we pin down concepts precisely
 - And can prove properties about them
 - Formalization:
 - Can be a goal by itself
 - Here: Tool for understanding
 - Do not go overboard!
 - Our terminology slightly different from Herlihy/Shavit's
 - You can use either
 - Good to be clear about this *when doubt may arise*

- ### States and Executions
- Object state:
- Assignment to fields
- Thread state:
- Assignment to local variables
 - Assignment to program counter, operand stacks and activation record stacks
- System state:
- State of all objects and threads
- Execution:
- Sequences of system/thread states
 - Can be finite or infinite – only finite for now
 - Produced by running the program

Example

Initial state: head = 0, tail = 0, items array allocated + initialized
 Producer and consumer threads initialized, program counters at initial states

Execution: Sequence of transitions from state to state

- Control in producer or consumer thread
- Inside or outside enq/deq methods

```

public class waitFreeQueue {
    int head = 0, tail = 0;
    Item[] items = (Item[]) new Object[capacity];

    public void enq(Item x) {
        while (tail == head) // busy-wait
            items[tail % capacity] = x; tail++;
    }

    public Item deq() {
        while (tail == head) // busy-wait
            Item item = items[head % capacity]; head++;
        return item;
    }
}
    
```

Histories/Traces

Executions produce histories – or traces – of events
 How – next slides
 Here:

$$\text{behaviour(program } P) = \{ \text{traces } t \mid \text{exists execution of } P \text{ such that } t = \text{trace(execution)} \}$$

Specification *Spec* = set of traces
 Correctness of program *P*: $R(\text{behaviour}(P), \text{Spec})$
 Question is: What is *R*?

Events

This lecture: Events = method calls and returns

Call event – method invocation:

$$A: x.m(a_1, \dots, a_n)$$

thread object method arguments

Normal return event:

$$A: x.ok(r_1, \dots, r_m)$$

result values No thread id given

Exceptional return event:

$$A: x.e(r_1, \dots, r_m)$$

exception

Traces - Matching Events

Trace: Finite or infinite sequence of events

A: p.enq(2) B: q.enq(7) A: p.ok() C: p.deq() C: p.ok(2)

Matching response:
 $A: x.m(a_1, \dots, a_n)$ matches $B: y.ok(r_1, \dots, r_m)$ iff
 $A = B$ and $x = y$

Method call: Matching call-return pair – interval (lecture 1)

Complication – Incomplete Calls

Pending invocation

A: p.enq(2) B: q.enq(7) A: p.ok() C: p.deq() C: p.ok(2)

Pending: No matching response

Extension of trace *H*:

- H* followed by matching responses to unmatched calls in *H*

Complete(*H*):

- Subsequence of *H* with unmatched calls removed

Example - Extension

A: p.enq(2) B: q.enq(7) A: p.ok() C: p.deq() C: p.ok(2)

Extension:

A: p.enq(2) B: q.enq(7) A: p.ok() C: p.deq() C: p.ok(2) B: q.error(full)

Example - Completion

$H = A: p.enq(2) \ B: q.enq(7) \ A: p.ok() \ C: p.deq() \ C: p.ok(2)$

complete(*H*):

A: p.enq(2) A: p.ok() C: p.deq() C: p.ok(2)

Subtraces and Equivalence

Thread subtrace $H|A$:

- Subsequence of H with all events not from A removed

$H = A: p.enq(2) B: q.enq(7) C: p.deq() A: p.ok() C: p.ok(2)$

$H|A = A: p.enq(2) B: q.enq(7) C: p.deq() A: p.ok() C: p.ok(2)$

Object subtrace $H|x$:

- Do. with all events not on x removed:

$H|p = A: p.enq(2) B: q.enq(7) C: p.deq() A: p.ok() C: p.ok(2)$

Sequentiality

Sequential trace:

- First event is call. All calls immediately followed by matching response

$A: p.enq(2) A: p.ok() C: p.deq() C: p.ok(2)$

Claim: Thread subtraces are always sequential
 – In the absence of recursion

Claim: Object subtraces may not be sequential. Why?

Sequential Specifications

A sequential specification is some way of telling whether a

- Single-thread, single-object history
- Is legal

For example:

- Pre and post-conditions
- But plenty of other techniques exist

For a sequential **multi-object** trace H :

- H is legal iff
- $H|x$ is in the sequential specification of x , for all x

Equivalence

Traces H_1 and H_2 are *equivalent* if $H_1|A = H_2|A$ for all A

Equivalence: Threads see the same subtraces in both

$H_1 = A: p.enq(2) A: p.ok() C: p.deq() C: p.ok(2)$

$H_2 = A: p.enq(2) C: p.deq() A: p.ok() C: p.ok(2)$

Condition:

- $H_1|A = H_2|A$
- $H_1|B = H_2|B$

Linearizability

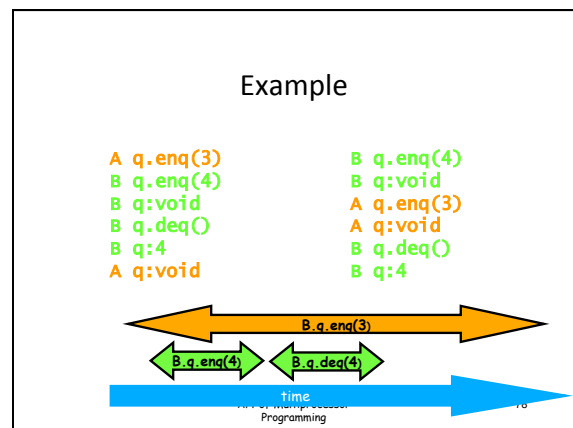
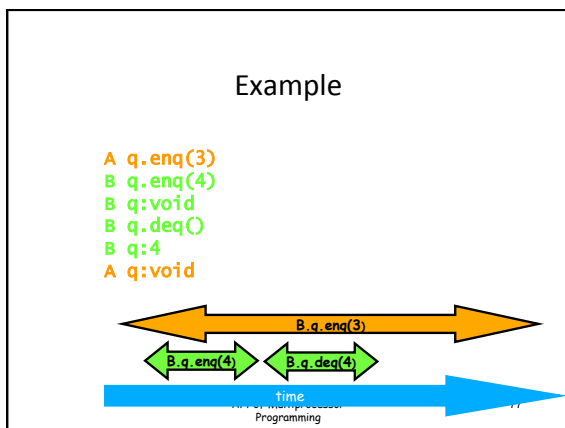
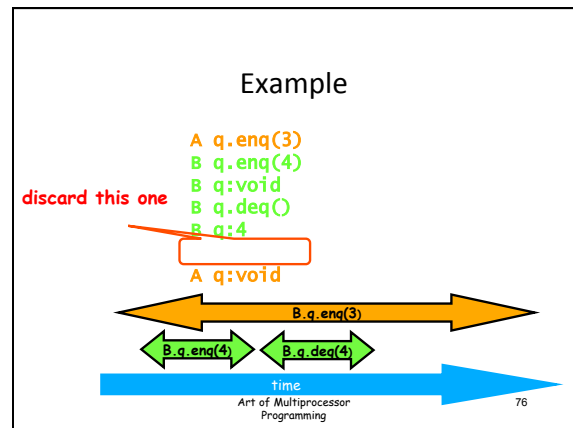
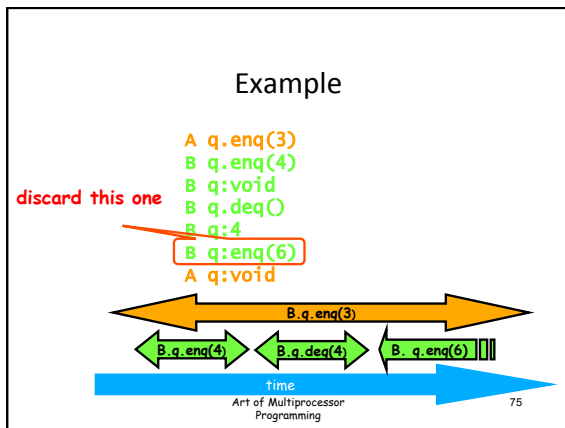
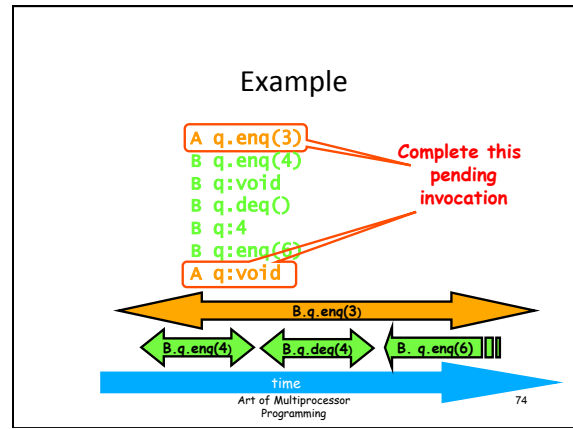
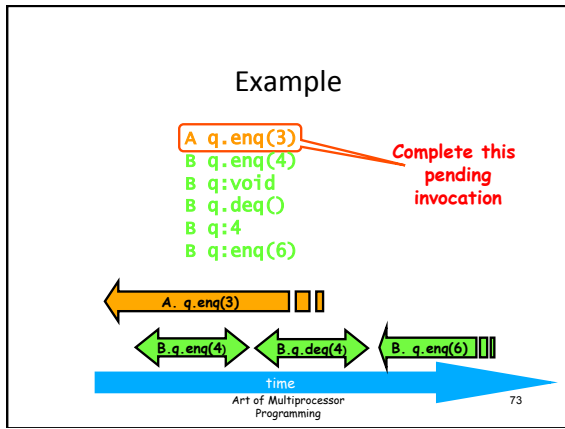
Definition: Trace H is linearizable if

- There is a legal sequential multi-object trace S
- H has an extension H'
 - (So pending calls can get response)
- $complete(H')$ is equivalent to S
 - (Not all pending calls need get a response)
- If $m_0 \rightarrow m_1$ in H then $m_0 \rightarrow m_1$ in S
 - Method calls that are sequential in H must appear in the right order
 - Recall the interval ordering \rightarrow from last lecture

Example

$A: q.enq(3)$
 $B: q.enq(4)$
 $B: q.void$
 $B: q.deq()$
 $B: q:4$
 $B: q:enq(6)$

Art of Multiprocessor Programming 72



Example

Equivalent sequential history

A q.enq(3)
 B q.enq(4)
 B q:void
 B q.deq()
 B q:4
 A q:void

B q.enq(4)
 B q:void
 A q.enq(3)
 A q:void
 B q.deq()
 B q:4

time
Programming

Composability

Theorem: H is linearizable iff for each object x , $H \upharpoonright x$ is linearizable

Proof: See Herlihy-Shavit

This is important!

- Linearizable objects => linearizable systems
- Otherwise might have to impose extra scheduler or locks to ensure linearizability at system level

Proving Linearizability

Recall: Trace H is linearizable if

- There is a legal sequential multi-object trace S
- H has an extension H'
- $complete(H')$ is equivalent to S
- If $m_0 \rightarrow m_1$ in H then $m_0 \rightarrow m_1$ in S

Strategy:

- Consider each object in turn – ok by composability
- Identify a single linearization point in each method – (Not always possible, alas)
- Show that all method events can be “shifted” to the linearization point

Proving Linearizability – with Locking

```

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();
    }
}
    
```

Linearization point

The Lock-free Case

Linearization points where tail/head updated

Remember: Only one producer and one consumer!

```

public class LockFreeQueue {
    int head = 0, tail = 0;
    items = (T[]) new Object[capacity];

    public void enq(Item x) {
        while (tail-head == capacity); // busy-wait
        items[tail % capacity] = x; tail++;
    }

    public Item deq() {
        while (tail == head); // busy-wait
        Item item = items[head % capacity]; head++;
        return item;
    }
}
    
```

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

Definition: Trace H is sequentially consistent if

- There is a legal sequential multi-object trace S
- H has an extension H'
- $complete(H')$ is equivalent to S
- if $m_0 \rightarrow m_1$ in H then $m_0 \rightarrow m_1$ in S

Not necessary to preserve precedence order between threads

But **program order** must be preserved !

Program order: Per thread precedence order - equivalence

Example: Sequential Consistency

(5) Art of Multiprocessor Programming 86

Example: Sequential Consistency

(5) Art of Multiprocessor Programming 87

Example: Sequential Consistency

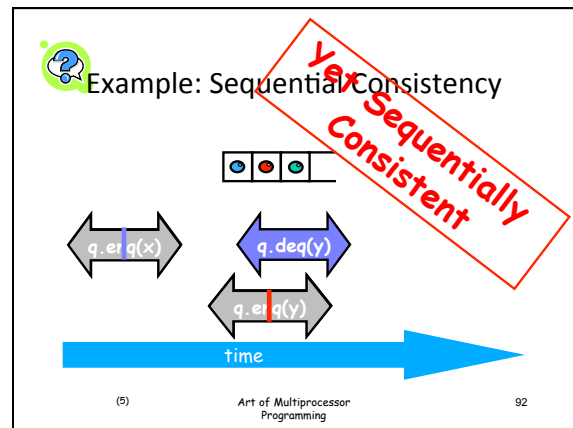
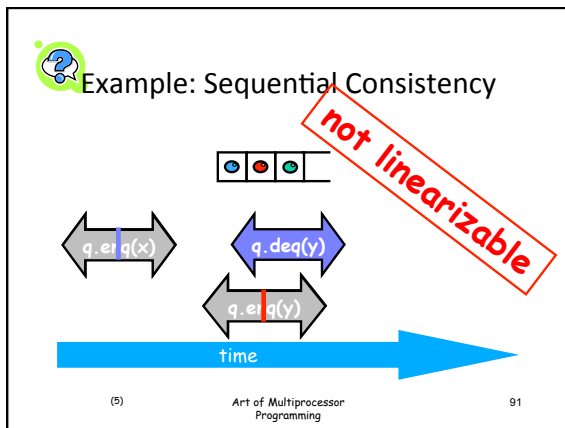
(5) Art of Multiprocessor Programming 88

Example: Sequential Consistency

(5) Art of Multiprocessor Programming 89

Example: Sequential Consistency

(5) Art of Multiprocessor Programming 90

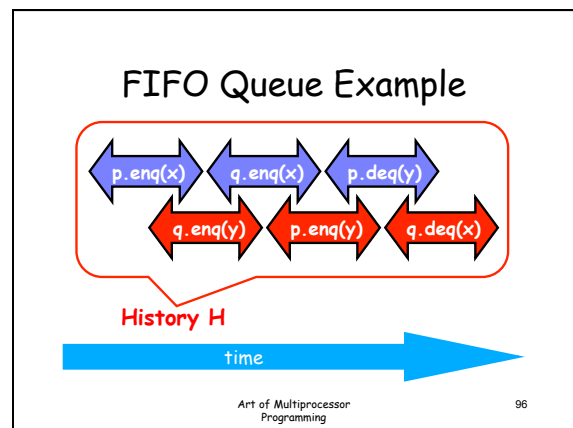
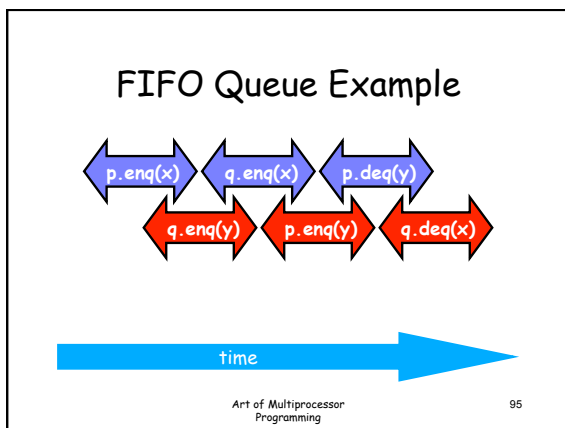
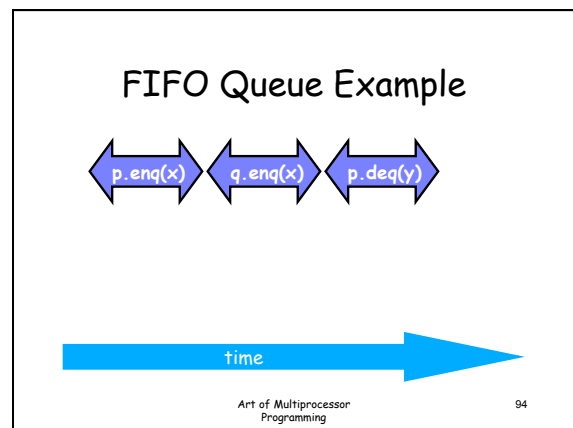


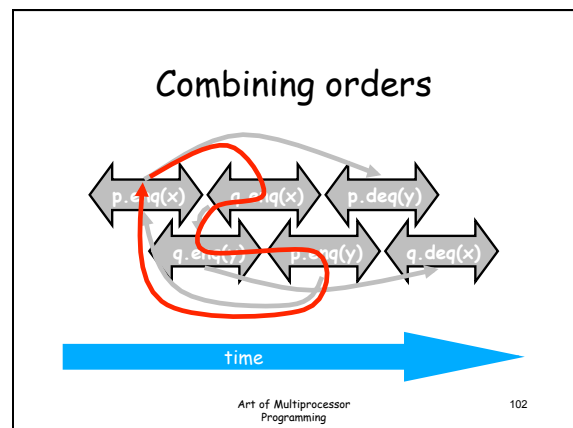
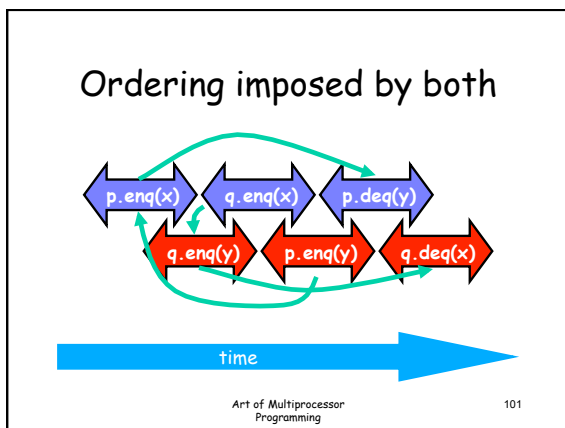
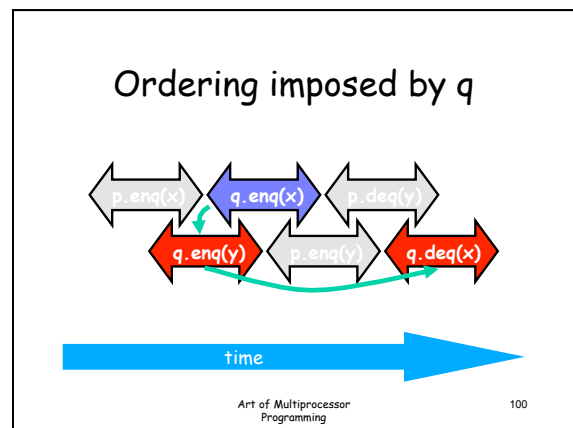
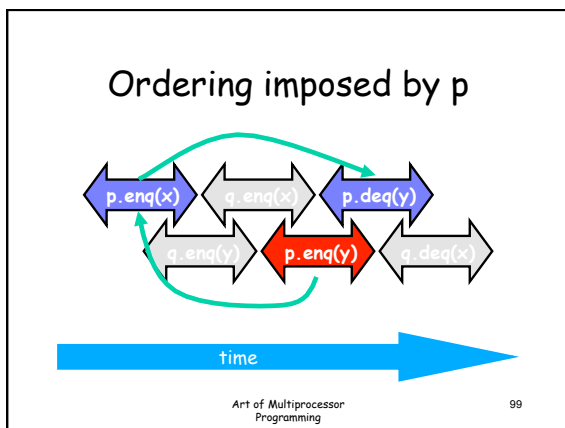
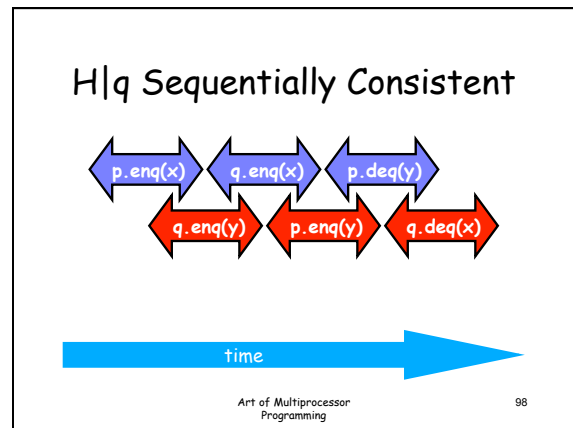
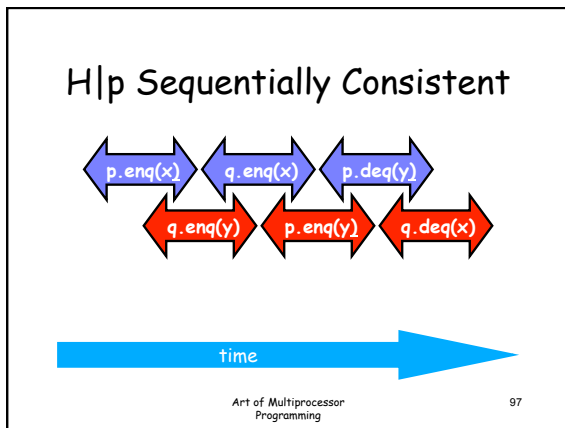
Theorem

Sequential Consistency is not a local property

(and thus we lose composability...)

Art of Multiprocessor Programming 93





Fact

- Most hardware architectures don't support sequential consistency
- Because they think it's **too strong**
- Here's another story ...

Art of Multiprocessor Programming 103

Example

Art of Multiprocessor Programming 104

Example

Each thread's view is sequentially consistent
– It went first

Art of Multiprocessor Programming 105

Example

Entire history isn't sequentially consistent
– Can't both go first
– Highly counterintuitive

Art of Multiprocessor Programming 106

Example

But what if each thread executes on its own core?
Read and writes address cores own cache
Writes and reads to main memory buffered

Art of Multiprocessor Programming 107

Example

But what if each thread executes on its own core?
Read and writes address cores own cache
Writes and reads to main memory buffered

Art of Multiprocessor Programming 108

Example

Is this behavior wrong?
– We can argue either way ...

Art of Multiprocessor Programming 109

The Hardware View

- Many hardware architects think that sequential consistency is **too strong**
- Too expensive to implement in modern hardware
- OK if fiction of sequentiality
 - Violated by default
 - Honored by explicit request

Art of Multiprocessor Programming 110

Memory Hierarchy

- On modern multiprocessors, processors do not read and write directly to memory
- Memory accesses are **very slow** compared to processor speeds
- Instead, each processor reads and writes directly to a cache
- To **read** a memory location, load data into cache.
- To **write** a memory location, update cached copy,
 - Lazily write cached data back to memory

Art of Multiprocessor Programming 111

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
- Processors delay writing to memory
- Until reads have been issued
- Otherwise get severe performance degradation
- If you need to synchronize, say so

Art of Multiprocessor Programming 112

Who Knew You Wanted to Synchronize?

- Writing to memory = mailing a letter
- Vast majority of reads & writes
- Not for synchronization
- No need to idle waiting for post office
- If you want to synchronize
 - Announce it explicitly
 - Pay for it only when you need it

Art of Multiprocessor Programming 113

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
- In **Java**, can ask compiler to keep a variable up-to-date with **volatile** keyword
- Also inhibits reordering, removing from loops, & other "optimizations"

Art of Multiprocessor Programming 114

Real-World Hardware Memory

- Weaker than sequential consistency
- But you can get sequential consistency at a price
- OK for experts, tricky stuff
 - assembly language, device drivers, etc.
- Linearizability more appropriate for high-level software

Art of Multiprocessor Programming 115

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.

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

Art of Multiprocessor Programming 116

Summary

- Concurrent access to objects need thought
- Linearizability:
 - Useful high-level model
 - Local and composable
- Locks: Easy way to obtain linearizability
 - But expensive
 - Deadlock freedom and starvation freedom can be obtained
- Can do better without locks
 - Trickier, but can get better progress properties
- For better performance yet need to use weaker memory models
 - Even trickier

Art of Multiprocessor Programming 117

CC BY-SA

This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](http://creativecommons.org/licenses/by-sa/3.0/).

- You are free:
 - to Share — to copy, distribute and transmit the work
 - to Remix — to adapt the work
- Under the following conditions:
 - Attribution. You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
 - Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
 - <http://creativecommons.org/licenses/by-sa/3.0/>.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author’s moral rights.

Art of Multiprocessor Programming 118