Programming Paradigms for Concurrency
Lecture 3 – Concurrent Objects

Based on companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Modified by
Thomas Wies
New York University
Concurrent Computation

memory

object

object
Objectivism

• What is a concurrent object?
  – How do we describe one?
  – How do we implement one?
  – How do we tell if we’re right?
Objectivism

• What is a concurrent object?
  – How do we describe one?

  – How do we tell if we’re right?
FIFO Queue: Enqueue Method
FIFO Queue: Dequeue Method

$q\text{.deq}()$
Lock-Based Queue

capacity = 8
Lock-Based Queue

Fields protected by single shared lock

capacity = 8
A Lock-Based Queue

```java
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];
    }
}
```

Fields protected by single shared lock
Lock-Based Queue

Initially head = tail
A Lock-Based 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];
    }
}

Initially head = tail
Lock-Based `deq()`
Acquire Lock

Waiting to enqueue...

My turn...
Implementation: `deq()`

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

Acquire lock at method start
Check if Non-Empty

Waiting to enqueue...
Implementation: `deq()`

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

If queue empty throw exception
Modify the Queue

Waiting to enqueue…
Implementation: \texttt{deq()}

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

Queue not empty?
Remove item and update head
Implementation: `deq()`

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

Return result
Release the Lock

head

0
1

tail

7
2

My turn!

6
5
4

3

x
Implementation: \texttt{deq()}

```java
generic 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();
  }
}
```

Release lock no matter what!
Implementation: `deq()`

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

Should be correct because modifications are mutually exclusive...
Now consider the following implementation

- The same thing without mutual exclusion
- For simplicity, only two threads
  - One thread enq only
  - The other deq only
Wait-free 2-Thread Queue

capacity = 8
Wait-free 2-Thread Queue

- head
- tail
- deq() (red)
- enq(z) (purple)
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
Wait-free 2-Thread Queue

\[ \text{result} = x \]

\[ \text{queue[tail]} = z \]
Wait-free 2-Thread Queue

head++

head

tail--

tag

tail

head

x

6

7

0

1

2

3

4

5

6

7
public class WaitFreeQueue {

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

    public void enq(Item x) {
        if (tail - head == capacity) throw new FullException();
        items[tail % capacity] = x; tail++;
    }

    public Item deq() {
        if (tail == head) throw new EmptyException();
        Item item = items[head % capacity]; head++;
        return item;
    }

}
public T deq() throws EmptyException {
    lock.lock();
    try {
        if (tail == head)
            throw new EmptyException();
        T x = items[head++];
        return x;
    } finally {
        lock.unlock();
    }
}
What *is* a Concurrent Queue?

- Need a way to specify a concurrent queue object
- Need a way to prove that an algorithm implements the object’s specification
- Lets talk about object specifications …
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

Let's begin with correctness
Sequential Objects

• Each object has a **state**
  – Usually given by a set of **fields**
  – Queue example: sequence of items

• Each object has a set of **methods**
  – Only way to manipulate state
  – Queue example: `enq` and `deq` methods
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, con’t)
  – the object will be in some other state
  – when the method returns,
Pre and PostConditions for Dequeue

• Precondition:
  – Queue is non-empty

• Postcondition:
  – Returns first item in queue

• Postcondition:
  – Removes first item in queue
Pre and PostConditions for Dequeue

- **Precondition:** Queue is empty
- **Postcondition:** Throws Empty exception
- **Postcondition:** Queue state unchanged
Why Sequential Specifications Totally Rock

• Interactions among methods captured by side-effects on object state
  – State meaningful between method calls

• Documentation size linear in number of methods
  – Each method described in isolation

• Can add new methods
  – Without changing descriptions of old methods
What About Concurrent Specifications?

- Methods?
- Documentation?
- Adding new methods?
Methods Take Time
Methods Take Time

invocation
12:00

q.enq(0)
Methods Take Time

$q$.enq(0)

invocation
12:00
Methods Take Time

Invocation 12:00

Method call

time
Methods Take Time

 invocation 12:00

 q.enq(0)

 Method call

 response 12:01

 void

 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 \textit{between} method calls

• Concurrent
  – Because method calls overlap, object might \textit{never} be \textit{between} method calls
Sequential vs Concurrent

- **Sequential:**
  - Each method described in isolation

- **Concurrent**
  - Must characterize *all* possible interactions with concurrent calls
    - What if two `enq` operations overlap?
    - Two `deq` operations? `enq` and `deq`? ...
Sequential vs Concurrent

• **Sequential:**
  – Can add new methods without affecting older methods

• **Concurrent:**
  – Everything can potentially interact with everything else
Sequential vs Concurrent

• **Sequential:**
  – Can add new methods without affecting older methods

• **Concurrent:**
  – Everything can potentially interact with everything else

Panic!
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
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();
    }
}
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,

Let's capture the idea of describing the concurrent via the sequential.

Behavior is "Sequential"
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 all of whose possible executions are linearizable
Example

...
Example

q.enq(x)

time
Example

q.enq(x)
q.enq(y)

time
Example

q.enq(x)
q.enq(y)
q.deq(x)

time
Example

```
q.enq(x)
q.enq(y)
q.deq(x)
q.deq(y)
```

```
[ ] [ ] [ ]
```
Example

Example time

q.enq(x)
q.enq(y)
q.deq(x)
q.deq(y)
q.enq(x)
q.enq(y)
q.deq(x)
q.deq(y)

linearizable
Example

q.enq(x)
q.enq(y)
q.deq(x)
q.deq(y)
q.enq(x)
q.enq(y)
q.deq(x)
q.deq(y)

Valid?
Example

time
Example

q.enq(x)

time
Example

\[ q \text{.} \text{enq}(x) \quad \text{time} \quad q \text{.} \text{deq}(y) \]
Example

$q.enq(x)$
$q.deq(y)$
$q.enq(y)$

$q.enq(x)$
Example

\[ q.\text{enq}(x) \]
\[ q.\text{enq}(y) \]
\[ q.\text{deq}(y) \]
\[ q.\text{enq}(x) \]
\[ q.\text{enq}(y) \]

\text{time}
Example

not linearizable
Example
Example

q.enq(x)

time
Example

\[ q\text{.enq}(x) \]

\[ q\text{.deq}(x) \]

time
Example

\[
\begin{align*}
q.\text{enq}(x) \\
q.\text{deq}(x)
\end{align*}
\]
Example

- `q.enq(x)`
- `q.deq(x)`

时间为线性可重放的。
Example

\texttt{q.enq(x)}

\textit{time}
Example
Example

1. `q.enq(x)`
2. `q.enq(y)`
3. `q.deq(y)`

Time line showing the operations:
- `q.enq(x)`
- `q.enq(y)`
- `q.deq(y)`
Example

\[ q\text{.enq}(x) \quad q\text{.enq}(y) \quad q\text{.deq}(y) \quad q\text{.deq}(x) \]

\text{time}
Example

```
q.enq(x)
q.enq(y)
q.deq(y)
q.deq(x)
```

Comme ci  Comme ça

```
```

multiple orders OK linearizable

time
Read/Write Register Example

write(0) → read(1) → write(2) → write(1) ⏰ read(0)

time
Read/Write Register Example

write(0) → read(1) → write(2) → read(0)

write(1) already happened
Read/Write Register Example

write(0) → read(1) → write(2) → read(0)

write(1) already happened
Read/Write Register Example

write(0) → read(1) → write(2)

write(1) already happened

not linearizable

read(0)
Read/Write Register Example

write(0) → read(1) → write(2) → read(1)

write(1) already happened
Read/Write Register Example

write(0) → read(1) → write(2) → read(1)

write(1) already happened
Read/Write Register Example

write(0) → read(1) → write(1) → write(2) → read(1)

write(1) already happened

not linearizable
Read/Write Register Example

write(0) write(1) write(2) read(1)

time
Read/Write Register Example

write(0)

write(1)

write(2)

read(1)

time
Read/Write Register Example

write(0)

write(1)
write(2)

read(1)

linearizable

time
Read/Write Register Example

write(0) → read(1) → write(2) → write(1) → read(1)
Read/Write Register Example
Read/Write Register Example
Read/Write Register Example

write(0) → read(1) → write(1) → write(2) → read(2)
Talking About Executions

• Why?
  – Can’t we specify the linearization point of each operation without describing an execution?

• Not Always
  – In some cases, linearization point depends on the execution
Linearizable Objects are Composable

- Modularity
- Can prove linearizability of objects in isolation
- Can compose independently-implemented objects
Reasoning About Linearizability: Locking

```java
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();
    }
}
```
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();
    }
}
public class WaitFreeQueue {

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

    public void enq(Item x) {
        if (tail-head == capacity) throw
            new FullException();
        items[tail % capacity] = x; tail++;
    }

    public Item deq() {
        if (tail == head) throw
            new EmptyException();
        Item item = items[head % capacity]; head++;
        return item;
    }
}
public class WaitFreeQueue {

    int head = 0, tail = 0;
    Item[] items = (T[]) new Object[capacity];

    public void enq(Item x) {
        if (tail - head == capacity) throw new FullException();
        items[tail % capacity] = x;
        tail++;
    }

    public Item deq() {
        if (tail == head) throw new EmptyException();
        Item item = items[head % capacity];
        head++;
        return item;
    }
}

Linearization order is order head and tail fields modified

Remember that there is only one enqueuer and only one dequeuer.

105
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
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

time
Example

q.enq(x)

time
Example

q.enq(x)  q.deq(y)

time
Example

q.enq(x)  q.deq(y)
q.enq(y)
Example

\[ q.\text{enq}(x) \]
\[ q.\text{enq}(y) \]
\[ q.\text{deq}(y) \]
\[ q.\text{enq}(x) \]
\[ q.\text{enq}(y) \]
Example

\[ q.\text{enq}(x) \]
\[ q.\text{enq}(y) \]
\[ q.\text{deq}(y) \]

\[ q.\text{enq}(x) \]
\[ q.\text{enq}(y) \]

not linearizable

\( \text{time} \)
Example

Yet Sequentially Consistent

q.enq(x)
q.enq(y)
q.deq(y)
q.enq(x)
q.enq(y)

time
Theorem

Sequential Consistency is not composable
FIFO Queue Example

\[ \text{time} \]

\[ \text{p.enq(x)} \quad \text{q.enq(x)} \quad \text{p.deq(y)} \]
FIFO Queue Example

p.enq(x)  q.enq(x)  p.deq(y)  q.enq(y)  p.enq(y)  q.deq(x)

time
FIFO Queue Example

History $H$
Sequentially Consistent
Sequentially Consistent

- $p.enq(x)$
- $q.enq(x)$
- $p.deq(y)$
- $q.enq(y)$
- $p.enq(y)$
- $q.deq(x)$

Time flow from left to right.
Ordering imposed by p

Art of Multiprocessor Programming
Ordering imposed by q

- p.enq(x)
- q.enq(x)
- p.deq(y)
- q.enq(y)
- p.enq(y)
- q.deq(x)

Time
Ordering imposed by both

```
p.enq(x)
q.enq(x)
p.deq(y)
q.enq(y)
p.enq(y)
q.deq(x)
```
Combining orders

- `p.enq(x)`
- `q.enq(x)`
- `p.deq(y)`
- `q.enq(y)`
- `p.enq(y)`
- `q.deq(x)`

Time arrow indicates the progression of events.
Fact

• Most hardware architectures don’t support sequential consistency
• Because they think it’s too strong
• Here’s another story …
The Flag Example

\begin{align*}
\text{x.write}(1) & \quad \text{y.read}(0) \\
\text{y.write}(1) & \quad \text{x.read}(0)
\end{align*}

\text{time}
The Flag Example

- Each thread’s view is sequentially consistent
  - It went first
The Flag Example

- Entire history isn’t sequentially consistent
  - Can’t both go first
The Flag Example

- Is this behavior really so wrong?
  - We can argue either way ...
Opinion 1: It’s Wrong

- This pattern
  - Write mine, read yours
- Is exactly the flag principle
  - Beloved of Alice and Bob
  - Heart of mutual exclusion
    - Peterson
    - Bakery, etc.
- It’s non-negotiable!
Opinion2: But It Feels So Right …

• Many hardware architects think that sequential consistency is too strong
• Too expensive to implement in modern hardware
• OK if flag principle
  – violated by default
  – Honored by explicit request
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
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.
Revisionist History

• Flag violation history is actually OK
  – processors delay writing to memory
  – until after reads have been issued.
• Otherwise unacceptable delay between read and write instructions.
• Who knew you wanted to synchronize?
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
Double-Checked Locking

```java
public class Singleton {
    private static Singleton instance;

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```
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
Volatile

- In Java, can ask compiler to keep a variable up-to-date with volatile keyword
- Also inhibits reordering, removing from loops, & other “optimizations”
Bakery Algorithm revisited

class Bakery implements Lock {
    volatile boolean[] flag;
    volatile Label[] label;
    public Bakery (int n) {
        flag = new boolean[n];
        label = new Label[n];
        for (int i = 0; i < n; i++) {
            flag[i] = false; label[i] = 0;
        }
    }
}
Real-World Hardware Memory

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

- Linearizability
  - Operation takes effect instantaneously between invocation and response
  - Uses sequential specification, locality implies composability
  - Good for high level objects
Correctness: Linearizability

• Sequential Consistency
  – Not composable
  – Harder to work with
  – Good way to think about hardware models

• We will use *linearizability* in the remainder of this course unless stated otherwise
Progress

• We saw an implementation whose methods were lock-based (deadlock-free)
• We saw an implementation whose methods did not use locks (lock-free)
• How do they relate?
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

<table>
<thead>
<tr>
<th></th>
<th>Non-Blocking</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Everyone</td>
<td><strong>Wait-free</strong></td>
<td><strong>Starvation-free</strong></td>
</tr>
<tr>
<td>makes progress</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Someone</td>
<td><strong>Lock-free</strong></td>
<td><strong>Deadlock-free</strong></td>
</tr>
<tr>
<td>makes progress</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

• We will look at *linearizable blocking* and *non-blocking* implementations of objects.
This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- 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.