Concurrent Object-Oriented Programming

Prof. Dr. Bertrand Meyer

Lecture 5: Concurrent Objects
Material From

The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
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?
FIFO Queue: Enqueue Method

$q.\text{enq}()$
FIFO Queue: Dequeue Method

$q$.deq()
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];
    }
}

Queue fields Initially head = tail protected by single shared lock

Queue fields protected by single shared lock
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();
    }
}
Now consider the following implementation

• The same thing without mutual exclusion
• For simplicity, only two threads
  • One thread enq only
  • The other deq only
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];
        return item;
    }
}

Queue is updated without

How do we define “correct” when modifications are not mutually exclusive?
Defining concurrent queue implementation

- 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
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 Postcondition for Dequeue

• **Precondition:**
  • Queue is non-empty
• **Postcondition:**
  • Returns first item in queue
• **Postcondition:**
  • Removes first item in queue
Pre- and Postcondition 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

Invocation 12:00

response 12:01

Method call

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
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

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();
    }
}
Intuitively...

• Lets 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

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

Time
Example

```
Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Ex
Example

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

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

`time`
Example

not linearizable
Example

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

linearizable

q.enq(x)

q.deq(x)

time
Example

Comme ci
Comme ça

linearizable
Read/Write Register Example

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

write(1) already happened
Read/Write Register Example

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

write(1) already happened

not linearizable
Read/Write Register Example

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

write(1) already happened
Read/Write Register Example

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

write(1) already happened

not linearizable
Read/Write Register Example

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

Time elapsed:

43
Read/Write Register Example

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

linearizable

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

time
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
Formal Model of Executions

- Define precisely what we mean
  - Ambiguity is bad when intuition is weak
- Allow reasoning
  - Formal
  - But mostly informal
    - In the long run, actually more important
Invocation Notation

```
A q.enq(x)
```

- **thread**
- **method**
- **object**
- **arguments**
Response Notation

- Method is implicit
A q: `empty()`

- Method is implicit
History - Describing an Execution

- Sequence of invocations and responses

\[ H = \begin{cases} 
  A \ q.\text{enq}(3) \\
  A \ q:\text{void} \\
  A \ q.\text{enq}(5) \\
  B \ p.\text{enq}(4) \\
  B \ p:\text{void} \\
  B \ q.\text{deq}() \\
  B \ q:3 
\end{cases} \]
Matching

- Invocation & response match if
  
  Thread names agree
  
  Object names agree

\[ \text{Method call} \]
Object Projections

\[ H|q = \]

\[
A \ q.\text{enq}(3) \\
A \ q:\text{void} \\
B \ p.\text{enq}(4) \\
B \ p:\text{void} \\
B \ q.\text{deq}() \\
B \ q:3
\]
Thread Projections

\[ H \mid B = \]

- A \( q.\text{enq}(3) \)
- A \( q:\text{void} \)
- B \( p.\text{enq}(4) \)
- B \( p:\text{void} \)
- B \( q.\text{deq}() \)
- B \( q:\text{3} \)
Complete Subhistory

- An invocation is pending if it has no matching response.
- The call may or may not have taken effect.
- Discard the pending invocation to get the complete subhistory.

\[
\text{complete}(H) = \begin{cases} 
A \ q.\text{enq}(3) \\
A \ q:\text{void} \\
A \ q.\text{enq}(5) \\
B \ p.\text{enq}(4) \\
B \ p:\text{void} \\
B \ q.\text{deq()} \\
B \ q:3 \\
\text{no matching response}
\end{cases}
\]
Sequential Histories

- In sequential histories method calls of different threads do not interleave.

```
A q.enq(3)  match
A q:void
B p.enq(4)  match
B p:void
B q.deq()   match
B q:3        Final pending
A q:enq(5)   invocation OK
```
Equivalent Histories

- Two histories are equivalent if different threads see the same thing in both histories.
- In case of two histories $H$ and $G$ with threads $A$ and $B$: $H|A = G|A$ and $H|B = G|B$.

$H = q.enq(3)$
$B p.enq(4)$
$B p: void$
$B q.deq()$
$A q: void$
$B q: 3$

$G = B q.deq()$
$A q: void$
$B p.enq(4)$
$B p: void$
$B q: 3$
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 ...
Legal Histories

• A sequential (multi-object) history $H$ is legal if
  • For every object $x$
  • $H|x$ is in the sequential spec for $x$
Precedence

- A method call precedes another if response event precedes invocation event.

\[
\begin{align*}
A & \text{ q.enq(3)} \\
B & \text{ p.enq(4)} \\
B & \text{ p.void} \\
A & \text{ q: void} \\
B & \text{ q.deq()} \\
B & \text{ q: 3}
\end{align*}
\]
Non-Precedence

- We have non-precedence when some method calls overlap one another.

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

• Given
  • History $H$
  • method executions $m_0$ and $m_1$ in $H$
• We say $m_0 \rightarrow_\mathcal{H} m_1$, if
  • $m_0$ precedes $m_1$
• Relation $m_0 \rightarrow_\mathcal{H} m_1$ is a
  • Partial order
  • Total order if $H$ is sequential
Linearizability

• History H is linearizable if it can be extended to G by
  • Appending zero or more responses to pending invocations
  • Discarding other pending invocations
• So that G is equivalent to
  • Legal sequential history S
  • where $\rightarrow_G \subseteq \rightarrow_S$
What is $\rightarrow_G \subset \rightarrow_S$

- It is a limitation on the choice of $S$!

$\rightarrow_G = \{a \rightarrow c, b \rightarrow c\}$

$\rightarrow_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$
Remarks

- Some pending invocations
  - Took effect, so keep them
  - Discard the rest
- Condition $\Rightarrow_G \subset \Rightarrow_S$
  - Means that $S$ respects “real-time order” of $G$
Example

Complete this pending invocation

Equivalent sequential history

discard this one
Concurrency

- How much concurrency does linearizability allow?
- When must a method invocation block?
Concurrent

- Focus on total methods
  - Defined in every state
- Example:
  - `deq()` that throws `Empty` exception
  - Versus `deq()` that waits ...
- Why?
  - Otherwise, blocking unrelated to synchronization
Concurrency

• Question: When does linearizability require a method invocation to block?
• Answer: never.
• Linearizability is non-blocking
Non-Blocking Theorem

- If method invocation $A \ q.inv(...) $ is pending in history $H$, then there exists a response $A \ q.res(...) $ such that $H + A \ q.res(...) $ is linearizable.
Proof

• Pick linearization $S$ of $H$
• If $S$ already contains
  • Invocation $A \text{q.inv}(\ldots)$ and response,
  • Then we are done.
• Otherwise, pick a response such that
  • $S + A \text{q.inv}(\ldots) + A \text{q:res}(\ldots)$
  • Possible because object is total.
Composability Theorem

- History $H$ is linearizable if and only if
  - For every object $x$
  - $H|_x$ is linearizable
- We care about objects only!
  - (Materialism?)
Why does composability matter?

- 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();
    }
}
```

Linearization points are when locks are released
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;
    }
}
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

- History H is Sequentially Consistent if it can be extended to G by
  - Appending zero or more responses to pending invocations
  - Discarding other pending invocations
- So that G is equivalent to a
  - Legal sequential history S
- where $G \subset S$

Differs from linearizability
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

q.enq(x)

q.deq(y)

q.enq(y)

time

79
Example

Sequentially not linearizable
Theorem

- Sequential Consistency is not a local property (and thus we lose composability).
FIFO Queue Example

History H

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

time
H|p Sequentially Consistent

\[ p\.enq(x) \quad q\.enq(x) \quad p\.deq(y) \quad q\.enq(y) \quad p\.enq(y) \quad q\.deq(x) \]

\[ \text{time} \]
H|q Sequentially Consistent

d enq(x) d eq(y)

d enq(y) d eq(x)

time

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

q.enq(y) p.enq(y) q.deq(x)
Ordering imposed by $p$
Ordering imposed by $q$
Ordering imposed by both

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

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

$time$
Fact

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

- Each thread’s view is sequentially consistent
  - It went first
- Entire history isn’t sequentially consistent
  - Can’t both go first
- 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!
Opinion 2: 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
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”
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
Critical Sections

• Easy way to implement linearizability
  • Take sequential object
  • Make each method a critical section

• Problems
  • Blocking
  • No concurrency
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 as 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?
Maximal vs. Minimal

- **Minimal progress:** in *some* suffix of $H$, some pending active invocation has a matching response (some method call eventually completes).
- **Maximal progress:** in *every* suffix of $H$, every pending active invocation has a matching response (every method call always completes).
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>Everyone makes progress</th>
<th>Non-Blocking</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><em>Wait-free</em></td>
<td><em>Starvation-free</em></td>
</tr>
<tr>
<td>Someone makes progress</td>
<td><em>Lock-free</em></td>
<td><em>Deadlock-free</em></td>
</tr>
</tbody>
</table>
Fair Histories

- A history is fair if each thread always continues to take steps.
- On multiprocessors this is controlled by the operating system.
- So fair histories are ones in which the operating system guarantees each thread continues to take steps.