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

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

```c
q.deq();
```
Lock-Based Queue

capacity = 8
Fields protected by single shared lock
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];
    }
}
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];
    }
}

Fields protected by single shared lock
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 \textbf{head} = \textbf{tail}
Lock-Based `deq()`
Acquire Lock

My turn ...

Waiting to enqueue...
Implementation: \texttt{deq()}
Check if Non-Empty

Not equal?

head

0

1

x

y

2

3

4

5

6

7

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();
    }
}
```

throw exception if queue empty
Modify the Queue

Waiting to enqueue...
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();
    }
}
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

y

2

3

7

6

5

4

Waiting…

x
Release the Lock

x

head

y

7

6

5

4

3

2

1

0

tail

My turn!
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();
    }
}
An alternative implementation

The same code 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

capacity = 8
Wait-free 2-Thread Queue

head

result = x

queue[tail] = z

Art of Multiprocessor Programming
Wait-free 2-Thread Queue
Wait-free 2-Thread Queue

```java
public class WaitFreeQueue {
    int head = 0, tail = 0;
    Object[] 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;
    }
}
```

No lock needed!

Art of Multiprocessor Programming
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;
  }
  public Item deq() {
    if (tail-head == 0) throw new EmptyException();
    Item item = items[head % capacity];
    return item;
    head++;
  }
}

How do we define “correct” when modifications are not mutually exclusive?
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
- Let’s talk about object specifications …
Correctness and Progress

In a concurrent setting, must specify both safety and liveness properties.

Need a way to define when an implementation is correct.

When 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: \texttt{enq} and \texttt{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 (also postcondition)
The object will be in some other state
Pre and PostConditions for Dequeue

**Precondition**
- Queue is non-empty

**Postcondition**
- Returns first item in queue
- 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

Documentation size linear in number of methods

Can add new methods without combinatorial blow-up
What About Concurrent Specifications?

Methods?

Documentation?

Adding new methods?
Methods Take Time
Methods Take Time

invocation
12:00

q.enq(Ø)
Methods Take Time

invocation 12:00

q.enq(⊙)

Method call

time
Methods Take Time

invocation 12:00

q.enq(○)
Methods Take Time

**invocation 12:00**

**response 12:01**

void

q.enq( )
Sequential vs Concurrent

**Sequential**
Method calls 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

Method call

**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 \texttt{enq()} calls overlap?

Two \texttt{deq()} calls? \texttt{enq()} and \texttt{deq()}? …
Sequential vs Concurrent

**Sequential**

Can add new methods without affecting older methods’ specs

**Concurrent**

Adding a method can potentially rewrite every other method’s spec

Panic!
Today’s Big Question

What does it even 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...

```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();
    }
}
Intuitively

Behavior acts “Sequential”
Linearizability

Each method call should "take effect" instantaneously between invocation and response events. Object is correct if "sequential" behavior is correct. Any such concurrent object is *Linearizable™*. 
Linearizability

Each method call should

“take effect”

Instantaneously

Between invocation and response events

Actually a property of an execution

A linearizable object: one all of whose possible executions are linearizable
Example

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

time
Example

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

Time progression

Art of Multiprocessor Programming
Example

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

time
Example

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

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
Example

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

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

time
Example

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

Time
Example

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

time
Example

not linearizable

\texttt{q.enq(x)} \quad \texttt{q.enq(y)} \quad \texttt{q.deq(y)} \quad \texttt{q.enq(x)} \quad \texttt{q.enq(y)}

\textbf{time}
Example
Example

q.enq(x)

time
Example

$q$.enq(x)

$q$.deq(x)

time
Example

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

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

\[ \text{time} \]
Example

q.enq(x)

q.deq(x)

linearizable

time
Example

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

time
Example

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

Time
Example

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

Time

Art of Multiprocessor Programming
Example

\[ q.enq(x) \]  \[ q.deq(y) \]  \[ q.enq(y) \]  \[ q.deq(x) \]  \[ \text{time} \]
Comme ci, Comme ça

multiple orders OK linearizable

g.enq(x) g.deq(y) g.enq(y) g.deq(x)

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)

write(1)

write(1) already happened

read(0)
Read/Write Register Example

write(0) → read(1) → write(2)

write(1) already happened

read(0)
Read/Write Register Example

write(0) → read(1) → write(2)

write(1) already happened
Read/Write Register Example

write(0) → read(1) → write(2)

write(1) already happened
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(1) --> write(2) --> read(1)

write(1) already happened

not linearizable
Read/Write Register Example

write(0)  write(2)

write(1)

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)

write(1)  read(1)

time
Linearization Points

Can we specify an operation’s linearization point without describing an execution?

Usually, but not always

In some cases, linearization point depends on the execution
Formal Model of Executions

Why?

Indicate precisely what you mean to say ...

We want to *reason* about objects

Sometimes formally ...

Usually informally
Split Method Calls into Two Events

Invocation
- method name & args
- `q.enq(x)`

Response
- result or exception
- `q.enq(x)` returns void
- `q.deq()` returns `x`
- `q.deq()` throws empty
Invocation Notation

A q.enq(x)
Invocation Notation

$q\text{.enq}(x)$

thread
Invocation Notation

A q.enq(x)

thread method
Invocation Notation

A \texttt{q.enq(x)}

thread

method

object
Invocation Notation

A q.enq(x)

thread

object

method

arguments
Response Notation

A q: void
Response Notation

A q: void

thread
Response Notation

thread

A q: void

result
Response Notation

thread

object

A q: void

result
Response Notation

Method is implicit

thread

object

A q: void

result
Response Notation

Method is implicit

A q: empty()
History - Describing an Execution

H =
- A q.enq(3)
- A q: void
- A q.enq(5)
- B p.enq(4)
- B p: void
- B q.deq()
- B q: 3

Sequence of invocations and responses
Definition

Invocation & response *match* if

Thread names agree

Object names agree

A q.enq(3)
A q:void

Method call
Object Projections

\[ H = \]

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

\[ H|q = \]

A \( q.\text{enq}(3) \)
A \( q:\text{void} \)

B \( q.\text{deq}() \)
B \( q:3 \)
Thread Projections

\[
H = \begin{align*}
A & \ q.\text{enq}(3) \\
A & \ q:\text{void} \\
B & \ p.\text{enq}(4) \\
B & \ p:\text{void} \\
B & \ q.\text{deq}() \\
B & \ q:3
\end{align*}
\]
Thread Projections

$$H | B = \begin{align*}
B & \ p. enq(4) \\
B & \ p: void \\
B & \ q. deq() \\
B & \ q: 3
\end{align*}$$
Complete Subhistory

An invocation is *pending* if it has no matching response

H =

A q.enq(3)
A q: void

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

An invocation is *pending* if it has no matching response
Complete Subhistory

\[ H = \]

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

May or may not have taken effect
Complete Subhistory

\[ H = \]

A q.enq(3)
A q: void

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

discard pending invocations
Complete Subhistory

A q.enq(3)
A q:void

Complete(H) =
B p:void
B q.deq()
B q:3
Sequential Histories

A q.enq(3)  
A q: void

B p.enq(4)  
B p: void

B q.deq()   
B q: 3

A q.enq(5)
Well-Formed Histories

Per-thread projections sequential

\[ H | B = \]

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

\[ H | A = \]

\[ A \ q.\text{enq}(3) \]
\[ A \ q:\text{void} \]
Equivalent Histories

Threads see the same thing in both

\[ H | A = G | A \]
\[ H | B = G | B \]
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 any style will do …
Legal Histories

A sequential (multi-object) history $H$ is \textit{legal} if

For every object $x$

$H|x$ is in the sequential spec for $x$
Precedence

A \texttt{q.enq(3)}
B \texttt{p.enq(4)}
B \texttt{p.void}
A \texttt{q.void}
B \texttt{q.deq()}
B \texttt{q:3}

A method call \textit{precedes} another if response event precedes invocation event

Method call
Concurrency

Some method calls overlap one another

Method call

Method call

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

History $H$

Method executions $m_0$ and $m_1$ in $H$

We say $m_0 \rightarrow_H m_1$, if

$m_0$ precedes $m_1$

Relation $m_0 \rightarrow_H m_1$ 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 $\rightarrow_G \subseteq \rightarrow_S$
Remarks

Some pending invocations

Took effect, so keep them

Discard the rest

Condition $\Rightarrow_G \subseteq \Rightarrow_S$

Means that S respects “real-time order” of G
Ensuring $\rightarrow^G \subseteq \rightarrow^S$

$\rightarrow^G = \{a \rightarrow c, b \rightarrow c\}$
$\rightarrow^S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$

A limitation on the Choice of $S$!
Example

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

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

Complete this pending invocation

A q.enq(3)
B q.enq(4)
B q.deq(3)
B q.enq(6)

time

Art of Multiprocessor Programming
Example

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

Complete this pending invocation

B q.enq(4)  B q.deq(4)  B q.enq(6)

time

Art of Multiprocessor Programming
Example

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

discard this one
Example

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

discard this one

Art of Multiprocessor Programming
Example

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

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
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

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

time
Concurrency

How *much* concurrency does linearizability allow?

When *must* a method invocation block?
Concurrency

Focus on *total* methods

Defined in every state

Example:

\texttt{deq()} throws \texttt{Empty} exception

Versus \texttt{deq()} that waits …

Why?

Otherwise, blocking unrelated to synchronization
Concurrency

When does linearizability require a method invocation to block?

never

Linearizability is non-blocking
Non-Blocking Theorem

If a 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 a linearization $S$ of $H$

If $S$ already contains

Invocation $A.q.inv(...)$ and response

Then we are done!

Otherwise, pick a response $res$ such that

$S + A.q.inv(...) + A.q:res(...)$

Possible because object is total.
Composability Theorem

(Multi-object) history $H$ is linearizable if & only if

For every object $x$ ...

$H|x$ is linearizable
Why Does Composability Matter?

Modularity!

Can prove object linearizability in isolation

Composition of linearizable objects is linearizable
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();
    }
}
```
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 when locks are released
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 {
    private int head = 0, tail = 0;
    private T[] items = 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;
    }
}
Strategy

Identify 1 step where method “takes effect”

Critical section

Machine instruction

Does not always work

A method might have alternative steps
Linearizability: Summary

- Standard notion of correctness
- Captures 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

Does *not* require \( G \subseteq S \)
Sequential Consistency

Does *not* preserve real-time order

Not allowed to re-order same-thread operations

OK to re-order operations by different threads

Often used for multiprocessor memories
Example

time
Example

\textbf{q.enq(x)}

\textbf{time}
Example

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

Time
Example

q.enq(x)

q.deq(y)

q.enq(y)

time
Example

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

not linearizable

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

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

Yet Sequentially Consistent
Theorem

Sequential Consistency is not composable
FIFO Queue Example

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

p.enq(x)  q.enq(x)  p.deq(y)
q.enq(y)  p.enq(y)  q.deq(x)
H|p Sequentially Consistent

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

time
H|q Sequentially Consistent

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

time
Ordering imposed by $p$

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

Arrows indicate the order of operations with time moving from left to right.
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)

time
Combining orders

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

Art of Multiprocessor Programming
Fact

Most hardware architectures don’t support even sequential consistency!

SC considered *too strong*
The Flag Example

- x.write(1)
- y.write(1)
- x.read(0)
- y.read(0)
The Flag Example

Each thread’s view is sequentially consistent

It went first

Art of Multiprocessor Programming
The Flag Example

Entire history isn’t sequentially consistent

Can’t both go first

Time
The Flag Example

x.write(1)  
y.write(1)  
x.read(0)  
y.read(0)

Is this behavior really so wrong?

time
Opinion: It’s Wrong

This pattern: Write mine, read yours

Is exactly the *flag principle*

Beloved of Alice and Bob

The very heart of mutual exclusion

Relied on by Peterson, Bakery, etc.

non-negotiable!
Peterson's Algorithm

```java
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}
public void unlock() {
    flag[i] = false;
}
```
Crux of Peterson Proof

\[ \text{write}_B(\text{flag}[B]=\text{true}) \]

\[ \Rightarrow \text{write}_B(\text{victim}=B) \]

\[ \Rightarrow \text{write}_A(\text{victim}=A) \]

\[ \Rightarrow \text{read}_A(\text{flag}[B]) \]

\[ \Rightarrow \text{read}_A(\text{victim}) \]

Assumes if a variable is written, a later read will see same or later value.
But It Feels So Right …

Many hardware architects think that sequential consistency is *too strong*.

Too expensive for modern hardware.

OK if flag principle.

violated by *default*.

Honored by *explicit request*. 
Hardware Consistency

Initially, $a = b = 0$.

Core 0
- `mov 1, a ;Store`
- `mov b, %ebx ;Load`

Core 1
- `mov 1, b ;Store`
- `mov a, %eax ;Load`

What are the possible values of `%eax` and `%ebx` registers after both processors have executed?

Sequential consistency implies that no execution ends with `%eax = %ebx = 0`
No modern-day processor implements sequential consistency.

Hardware actively reorders instructions.

Compilers too.

Because performance is dominated by single-thread unsynchronized execution.
Q. Why might the hardware or compiler decide to reorder these instructions?

A. Higher performance by covering load latency — *instruction-level parallelism*. 
Instruction Reordering

Q. When is it safe for the hardware or compiler to perform this reordering?

A. When \( a \neq b \).

A’. And there’s no concurrency.
Processor can issue stores faster than the network can handle them ⇒ queued in store buffer.

Loads take priority, bypassing the store buffer.

Except if a load addr matches store buffer addr
X86: Memory Consistency

1. Loads are not reordered with loads.
2. Stores are not reordered with stores.
3. Stores are not reordered with prior loads.
4. A load may be reordered with a prior store to a different location but not with a prior store to the same location.
5. Stores to the same location respect a global total order.
1. Loads are not reordered with loads.
2. Stores are not reordered with stores.
3. Stores are not reordered with prior loads.
4. A load may be reordered with a prior store to a different location but not with a prior store to the same location.
5. Stores to the same location respect a global total order.

X86: Memory Consistency

Thread’s Code

Store1
Store2
Load1
Load2
Store3
Store4
Load3
Load4
Load5

Total Store Ordering (TSO)...weaker than sequential consistency

OK!

Art of Multiprocessor Programming
A memory barrier (or fence) is a hardware operation that enforces ordering between the instructions before and after the fence.

A memory barrier can be an explicit instruction (x86: mfence).

The typical cost of a memory fence is comparable to that of an L2-cache access.
X86: Memory Consistency

Thread's Code

Store1
Store2
Load1
Load2
Store3
Store4
Barrier
Load3
Load4
Load5

Loads are not reordered with loads.

Stores are not reordered with stores.

Stores are not reordered with prior loads.

A load may be reordered with a prior store to a different location but not with a prior store to the same location.

Stores to the same location respect a global total order.

Total Store Ordering + properly placed memory barriers = sequential consistency
Memory Barriers

Memory barrier instruction will:

- Flush write buffer
- Bring caches up to date
- Compilers often do this for you
- Entering and leaving critical sections
Volatile Variables

In Java, can ask compiler to keep a variable up-to-date by declaring it \texttt{volatile}

Adds a memory barrier after each store

Inhibits reordering, removing from loops, & other “compiler optimizations”

Will discuss in detail later
Summary: Real-World

Hardware *weaker* than sequential consistency

Can get sequential consistency at a price

Linearizability better fit for high-level software
Linearizability

- Operation takes effect instantaneously between invocation and response
- Based on sequential spec
- Composable, non-blocking
- "gold standard" for concurrent data structures
Summary: Correctness

Sequential Consistency

- Not composable
- Harder to work with
- Baseline for low-level hardware

We will use *linearizability* as our consistency condition unless stated otherwise.
We saw a lock-based queue

We saw a lock-free queue

How are they related?
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>Non-Blocking</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait-free</td>
<td>Starvation-free</td>
</tr>
<tr>
<td>Lock-free</td>
<td>Deadlock-free</td>
</tr>
</tbody>
</table>

Everyone makes progress

Someone makes progress
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.
Processor can issue stores faster than the network can handle them ⇒ queued in store buffer.

Loads take priority, bypassing the store buffer except if a load addr matches an addr in the store buffer.

*ff-free*: every thread calling a method eventually returns.

We will use *linearizability* as our consistency condition unless stated otherwise.