Concurrency and Correctness

CS 475, Spring 2019
Concurrent & Distributed Systems

With material from Herlihy & Shavit, Art of Multiprocessor Programming
Review: Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent (why?)
Review: Locks (Mutual Exclusion)

public interface Lock {
    public void lock();
    public void unlock();
}
Review: Mutual Exclusion, Formally

- Let $CS_i^k$ be thread $i$'s $k$-th critical section execution.
- And $CS_j^m$ be thread $j$'s $m$-th execution.
- Then either
  - $CS_i^k \rightarrow CS_j^m$
  - $CS_j^m \rightarrow CS_i^k$

Aka: it is guaranteed that one critical section happens before the other.
Review: Synchronized blocks in Java

• Can also use any object as that monitor

```java
static Object someObject = new Object();
public static void increment()
{
    synchronized(someObject){
        i = i + 1;
    }
}

static Object someOtherObject = new Object();
public static void incrementOther()
{
    synchronized(someOtherObject){
        j = j + 1;
    }
}
```

Now, two different threads could call `increment()` and `incrementOther()` at the same time
Review: Peterson’s Alg: Mutual Exclusion

public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}

• If thread 1 in critical section,
  - flag[1]=true,
  - victim = 0  
  
• If thread 0 in critical section,
  - flag[0]=true,
  - victim = 1  
  
Cannot both be true, hence yes: it is safe!
Review: Amdahl's Law

• Identifies performance gains from adding additional cores to an application that has both serial and parallel components
• $S$ is serial portion
• $N$ processing cores
• That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
• As $N$ approaches infinity, speedup approaches $1 / S$
• Serial portion of an application has disproportionate effect on performance gained by adding additional cores
Today

- Concurrent specifications
- Linearizability
- Reasoning about correctness
- Reading: H&S 2.1-2.3
- HW1 due soon! https://www.jonbell.net/gmu-cs-475-spring-2019/homework-1/
Concurrent Computation
FIFO Queue: Enqueue Method

q.enq(○)
FIFO Queue: Dequeue Method

q.deq()
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];
    }
}

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();
    }
}
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
- Remember Amdahl’s law?
- 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]; head++;
        return item;
    }
}
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;
    }
}
Lock-free 2-Thread Queue

```java
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;
    }
}
```

Queue is updated without a lock!

How do we define “correct” when modifications are not mutually exclusive?
public class WaitFreeQueue {
    int head = 0, tail = 0;
    Item[] 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;
    }
}

Argument for why this is OK (for now):
No two threads ever write the same variable

Wrote items, writes tail
Wrote head

Defining concurrent queue implementations

- 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, 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 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(...)
Methods Take Time

invocation
12:00

q.enq(…)

time
Methods Take Time

Method call

q.enq(...)
Methods Take Time

Method call

invocation 12:00

response 12:01

void

q.enq( )

time
Sequential vs Concurrent

- **Sequential**
  - Methods take time? Who knew?
- **Concurrent**
  - Method call is not an event
  - Method call is an interval.
Concurrent Methods Take **Overlapping** Time
Concurrent Methods Take Overlapping Time
Concurrent Methods Take Overlapping Time
Concurrent Methods Take **Overlapping** Time
Sequential vs Concurrent

- Sequential:
  - Object needs meaningful state only between method calls

- Concurrent:
  - Because method calls overlap, object might never be between method calls
Sequential vs Concurrent

- Sequential:
  - Each method described in isolation

- Concurrent
  - Must characterize all possible interactions with concurrent calls
    - What if two enqs overlap?
    - Two deqs? enq and deq? …
Sequential vs Concurrent

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

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

```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();
    }
}
```

All modifications of queue are done mutually exclusive.
Intuitively

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

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

time
Example: Linearizable?

Reminder: Linearizable means: each method takes effect instantaneously, sometime in its observed time window

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

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 Lineraizability: 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 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;
    }
}
public class LockFreeQueue {
    int head = 0, tail = 0;
    Object[] items = 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;
    }
}
What’s next?

• Weds: One more consistency model: *sequential* (is included in reading for this lecture though, book covers it in a different order), plus more Java-specific implementation fun!

• Reminder for Monday: HW1 Due!!!
This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/
- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - for any purpose, even commercially.
- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.