Methods Take Time

invocation 12:00

q.enq(

...) (method call)

response 12:01

void

Method call

time
Concurrent Methods Take **Overlapping** Time
Example: Linearizable?

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

• From our model of strict happens-before to… how programming languages really work
• Plus condition variables, monitors and semaphores
• Reading: H&S 3.8, 8.1-8.5
• Note: HW1 coming up: https://www.jonbell.net/gmu-cs-475-fall-2019/homework-1/
Alternative: Sequential Consistency

• No need to preserve real-time order
  – Cannot re-order operations done by the same thread
  – Can re-order operations done by different threads without affecting each individual threads' order
• Often used to describe multiprocessor memory architectures
• Formulation:
  • There is some total order of operations so that:
    • Each CPUs operations appear in order
    • All CPUs see results according to that order (read most recent writes)
Example
Example

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

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

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

\[
q.e\!\!q(x) \quad q.e\!\!q(y) \quad q.d\!\!q(y) \quad q.e\!\!q(y)
\]
Example

Not linearizable!
Example

- Reminder, sequentially consistent if there is some total order of operations so that:
  - Each CPUs operations appear in order
  - All CPUs see results according to that order (read most recent writes)
Sequential Consistency is not a local property
(and thus we lose composability...)

Theorem
FIFO Queue Example

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

Time
FIFO Queue Example

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

Time progression from left to right.
FIFO Queue Example

History H
H∥p Sequentially Consistent

- p.enq(x)
- q.enq(x)
- p.deq(y)
- q.enq(y)
- p.enq(y)
- 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)
p.enq(y)  q.enq(y)  p.enq(y)
q.enq(y)  p.enq(y)  q.deq(x)
```
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)`
Sequentially Consistent Overall?

The only way to make this work is if these operations are reordered: but we can not reorder operations within a thread!

P: must enq y before x to make deq right

Q: must enq x before y to make deq right!
Sequential Consistency vs Linearizability

• Linearizability can be composed:
  • If p’s execution and q’s execution are both linearizable, then the combination must also be linearizable

• Sequential consistency can not be composed:
  • If p’s execution and q’s execution are both sequential, then the combination MAY also be sequential (but not guaranteed!)

• Why use sequential consistency?
  • Does not require global clock
Sequential Consistency

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

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

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
    • We used this to implement our lock…

• 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 Operations are Slow

![Diagram showing memory operations between CPUs and caches]

- CPU 1
  - thread0(): 7ns
  - CPU 1 Cache: 100ns
- CPU 2
  - thread1():
  - CPU 2 Cache:
- Main Memory: 7ns
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 the 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 share that variable?
Who knew you wanted to share the variable?

• Writing to memory = mailing a letter
• Vast majority of reads & writes
  – Not shared between CPUs
  – No need to idle waiting for post office
• If you want to make sure that there variable is shared
  – Announce it explicitly
  – Pay for it only when you need it
• Foreshadowing: Writing to network = mailing a letter to the moon
Explicitly Sharing Variables

- 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”
Volatile Keyword
Linearizability, Sequential Consistency, Java, and Us

• What guarantees can we get, and how?
  • Different threads’ access to the same variable is sequential -> volatile
  • Different threads’ access to different variables is linear -> a lock around every read/write to those variables
Revisionist History

class Bouncer {
    public static final int DOWN = 0;
    public static final int RIGHT = 1;
    public static final int STOP = 2;
    private boolean goRight = false;
    private ThreadLocal<Integer> myIndex;
    private int last = -1;

    int visit() {
        int i = myIndex.get();
        last = i;
        if (goRight)
            return RIGHT;
        goRight = true;
        if (last == i)
            return STOP;
        else
            return DOWN;
    }
}

Without volatile, it would actually be possible for ALL threads to see STOP
Revisionist History

class Bouncer {
    public static final int DOWN = 0;
    public static final int RIGHT = 1;
    public static final int STOP = 2;
    private volatile boolean goRight = false;
    private ThreadLocal<Integer> myIndex;
    private volatile int last = -1;

    int visit() {
        int i = myIndex.get();
        last = i;
        if (goRight)
            return RIGHT;
        goRight = true;
        if (last == i)
            return STOP;
        else
            return DOWN;
    }
}

Without volatile, it would actually be possible for ALL threads to see STOP
Synchronization beyond locks and flags
Return to the FIFO queue

• What if we want our lock based queue to be bounded
• Recall: what we did was use a lock around the enqueue and dequeue methods
  
  ```java
  mutex.lock()
  try{
    queue.enq(x);
  } finally{
    mutex.unlock();
  }
  
  • What happens if the queue is full?
    • Throw an exception?
    • Can’t we make it wait until the queue has space?
Synchronized methods in Java

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

Result: Before entering `increment()`, thread gets a lock on the Class object of `increment()`
Synchronized methods in Java

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

Result: Before entering `increment()`, thread gets a lock on the Class object of `increment()`

```java
public synchronized static void incrementOther()
{
    j = j + 1;
}
```

Result: Before entering `incrementOther()`, thread gets a lock on the Class object of `incrementOther()`

Problem?
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
Two mechanisms to enable coordination between multiple threads using the same monitor (target of synchronized)

While holding a monitor on an object, a thread can wait on that monitor, which will temporarily release it, and put that thread to sleep

Another thread can then acquire the monitor, and can notify a waiting thread to resume and re-acquire the monitor
wait and notify() example

public class BlockingQueue<T> {
    private Queue<T> queue = new LinkedList<T>();
    private int capacity;

    public BlockingQueue(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void put(T element) throws InterruptedException {
        while (queue.size() == capacity) {
            wait();
        }
        queue.add(element);
        notify(); // notifyAll() for multiple producer/consumer threads
    }

    public synchronized T take() throws InterruptedException {
        while (queue.isEmpty()) {
            wait();
        }
        T item = queue.remove();
        notify(); // notifyAll() for multiple producer/consumer threads
        return item;
    }
}

Only one thread can be in put or take of the same queue
Java Lock API

• **Synchronized** gets messy: what happens when you need to synchronize many operations? What if we want more complicated locking?

• **ReentrantLock**: same semantics as synchronized, also supports conditions...

```java
static ReentrantLock lock = new ReentrantLock();
public static void increment()
{
    lock.lock();
    try{
        i = i + 1;
    } finally{
        lock.unlock();
    }
}
```
Conditions

• When a thread is waiting for something to happen, it might want to release the lock and be notified when that thing has happened
• Ex: while queue is full, release the lock to let someone else empty it
• This is what a condition does
• Key methods: `await`, `signal`

```java
Condition condition = lock.newCondition();
lock.lock();
try{
    while(!property)
        condition.await();
} catch(InterruptedException e){
    //Application-dependent response, property may still be false
}
//at this point, property must be true and we have the lock again
condition.signal(); //wake up one thread await'ing
condition.signalAll(); //wake up all threads await'ing
```
Conditions

• An awakened thread must:
  • try to reclaim the lock;
  • when this has happened, retest the property it is waiting for;
  • if the property doesn’t hold, release the lock by calling await().

• wrong: if “boolean expression” condition.await()
• correct: while “boolean expression” condition.await()
Monitors

• Java’s **synchronized** keyword creates a *monitor*, which is both a lock and a condition variable
• Three relevant methods that have been there all along, and you may not have known what for:
  • `object.notify();`
  • `object.notifyAll();`
  • `object.wait();`
Non-mutual exclusion locks

• The strict mutual exclusion property of locks is often relaxed. Three examples are:
  • Readers-writers lock: Allows concurrent readers, while a writer disallows concurrent readers and writers.
  • Reentrant lock: Allows a thread to acquire the same lock multiple times, to avoid deadlock (we have mostly used reentrant locks)
  • Semaphore: Allows at most c concurrent threads in their critical section, for some given capacity c.
Readers and Writers Problem

- Commonly called a read-write lock: data can be read by an unlimited number of threads at a time, written by at most one
- If a thread wants to write, nobody else can be reading (Alice + Bob with the billboards)
- High level approach:
  - Build on top of mutual exclusion locks with condition variables
class ReadLock implements Lock{
    public void lock()
    {
        lock.lock();
        try{
            while(writer){
                condition.await();
            }
            readers++;
        } finally {
            lock.unlock();
        }
    }

    @Override
    public void unlock() {
        lock.lock();
        try{
            readers--;
            if(readers == 0)
                condition.signalAll();
        } finally{
            lock.unlock();
        }
    }
}

class WriteLock implements Lock{
    public void lock()
    {
        lock.lock();
        try{
            while(readers > 0 || writer)
                condition.await();
            writer = true;
        } finally{
            lock.unlock();
        }
    }

    @Override
    public void unlock() {
        writer = false;
        condition.signalAll();
    }
}
Readers and Writers Lock

- Note: not fair - many readers can prevent a single writer from getting in
- A fair solution is outlined in the textbook
Where we go next

• How should we apply these different locking primitives?
• HW1 due on Weds
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.