Threads and Locks, Part 2

CSCI 5828: Foundations of Software Engineering
Lecture 08 — 09/18/2014

Goals

- Cover the material presented in Chapter 2 of our concurrency textbook
  - In particular, selected material from Days 2 and 3
- java.util.concurrent
  - ReentrantLock
  - AtomicVariables
  - Thread Pools
  - Producer Consumer Example
    - Warning: the book expects you to download a 10GB file that expands to be 46 GB in order to run these examples!
    - I've done this for you. (You're welcome. 😀)

What's the easy way?

- Last lecture, we looked at "doing concurrency the hard way"
  - So, what's the easy way?
- We are not necessarily going to answer that question in this lecture
  - But, we are going to look at the java.util.concurrency library to help make it less painful to create concurrent software
    - <http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html>
- As we move forward, we'll look at other models and techniques that all attempt to reduce the complexity associated with concurrent programming
  - However, each new model/technique will introduce its own set of difficulties such as having to learn Clojure. 😀

Synchronized Keyword => Intrinsic Locks

- We've been using the **synchronized** keyword to try to address the problems we've been seeing in our concurrent programs so far
- The problem with this keyword is that it makes use of an inflexible lock that our book calls **intrinsic locks**
    - It is inflexible because
        - A thread blocked on an intrinsic lock cannot be interrupted
            - It's all or nothing
        - A thread blocked on an intrinsic lock cannot have a time out
        - There's only one type of intrinsic lock (reentrant) and only one way to acquire it (with the synchronized keyword)

java.util.concurrent.locks.ReentrantLock

- The ReentrantLock class from java.util.concurrent.locks allows us to gain the benefit of fine-grained locking without having to use the synchronized keyword
- To use them, you first instantiate a lock (and stash it somewhere)
  - Lock lock = new ReentrantLock();
- Then you follow this pattern
  - `lock.lock();`
  - `try {`
    - `«use shared resource»`
  - `} finally {`
    - `lock.unlock();`
  - `}`

First benefit: Abstraction (I)

- Note this line of code carefully
  - Lock lock = new ReentrantLock();
- The concrete class is `ReentrantLock`. Our variable is of type `Lock`
  - This gives us a nice abstraction with the following methods
    - lock() — acquire the lock and block if needed
    - lockInterruptibly() — acquire the lock, block if needed, allow interruption
    - newCondition() —bind a Condition to this lock
    - tryLock() and tryLock(…) —acquire the lock without blocking or with a time out
    - unlock() — release the lock

First benefit: Abstraction

- Currently java.util.concurrent.locks provide us with two types of locks
  - ReentrantLock and ReentrantReadWriteLock
- But the interface Lock allows us to write code that is not dependent on the concrete classes and could change if better options are added in the future
- This abstraction also addresses the problem of having only one way to acquire an intrinsic lock
  - The Lock interface gives us four different ways of acquiring a lock
    - blocking, non-blocking, with a time out, with the option of being interrupted
- This flexibility allows us to design concurrent systems with more powerful constructs and avoid the problems we saw in Lecture 7

Second Benefit: Thread Interruption

- The book has two programs to demonstrate the ability (or lack thereof) to interrupt threads that are blocked on locks
  - With an intrinsic lock, there's no way to do it
    - if a thread becomes blocked and the semantics of your program will never allow it to become unblocked
      - then it's stuck until you kill the JVM
- With the Lock interface, you simply acquire it with the lockInterruptibly() method and then Thread.interrupt() behaves as you would expect
- DEMO

Third Benefit: Timeouts (I)

- With the Lock interface, you have the option of calling tryLock()
  - This is a nonblocking call that returns TRUE if the lock is acquired
  - `if (lock.tryLock()) {`
    - `Lock acquired`
  - `} else {`
    - `Lock not acquired; act accordingly`
  - `}`
- You also have the option to specify a timeout (from nanoseconds to days!)
  - `if (lock.tryLock(50L, TimeUnit.SECONDS)) {`
    - `Lock acquired`
  - `} else {`
    - `Timeout occurred; act accordingly`
  - `}`

Third Benefit: Timeouts (II)

- The book provides an example of the Dining Philosophers program that makes use of timeouts
  - We don't have to worry about acquiring chopsticks in the same order
    - We instead adopt the following strategy
      - acquire a lock on the left chopstick
      - acquire a lock on the right chopstick using tryLock() with a timeout
        - If that fails, release the lock on the left chopstick and try again
- This works BUT you can have a problem with LIVELOCK
  - Live lock is the situation in which a program does not make progress because the threads lock, time out, and then immediately lock again
    - You spend your time dealing with locks and not performing work

Note: Skipping Hand-over-Hand Locking

- The book has an example of using ReentrantLocks to allow multiple threads to access a linked list at the same time
  - including allowing multiple insertions to occur without damaging the overall structure of the list
  - and allowing iteration and other operations to occur in parallel on the list
- I'm going to skip this example as it is fairly low-level
  - The key idea is how the locks are acquired as a thread moves through the list
    - You acquire a lock on node i and then try to acquire a lock on node i+1
      - Once you have the lock on node i+1, you can release the lock on node i

Fourth Benefit: Condition Variables (I)

- Condition Variables are used when your thread's logic involves waiting for an event to occur (or a condition to become true) before it does its work
    - If the event has not happened or the condition is false, we want the thread to block and not consume the scheduler's time
- Coding this logic can be quite challenging
    - You may have a lot of threads waiting for the condition to become true
        - As a result, you need a queue or a collection class to hold the threads while they are waiting
        - When the condition becomes true, you need a way to "wake up" the threads that are waiting, and let them try to do their work
- Condition variables make coding this type of logic straightforward

Fourth Benefit: Condition Variables (II)

- To work with a conditional variable, you follow this pattern
  - `ReentrantLock lock = new ReentrantLock();`
  - `Condition condition = lock.newCondition();`
  - `lock.lock()`
  - `try {`
    - `while (!«condition is true») { condition.await(); }`
    - `«use shared resources»`
  - `} finally { lock.unlock(); }`
- The key to this pattern is that the variable and the condition are tied together
  - The thread has to acquire the lock and then check the condition
  - If the condition is false, it needs to wait. The call to await() blocks the thread and **unlocks the lock** *atomically*, allowing other threads to access the shared resources

Fourth Benefit: Condition Variables (III)

- To work with a conditional variable, you follow this pattern
  - `ReentrantLock lock = new ReentrantLock();`
  - `Condition condition = lock.newCondition();`
  - `lock.lock()`
  - `try {`
    - `while (!«condition is true») { condition.await(); }`
    - `«use shared resources»`
  - `} finally { lock.unlock(); }`
- The key to this pattern is that the variable and the condition are tied together
  - If **some other thread** performs work that may have changed the condition, it calls signal() or signalAll() on the condition. The former wakes up a single thread that was blocked on the condition; the latter wakes them all up
    - If signalAll() is used, **all of the threads** battle to **reacquire the lock**

Fourth Benefit: Condition Variables (IV)

- To work with a conditional variable, you follow this pattern
  - `ReentrantLock lock = new ReentrantLock();`
  - `Condition condition = lock.newCondition();`
  - `lock.lock()`
  - `try {`
    - `while (!«condition is true») { condition.await(); }`
    - `«use shared resources»`
  - `} finally { lock.unlock(); }`
- The key to this pattern is that the variable and the condition are tied together
  - It is thus **guaranteed** that when await() returns, the thread once again has acquired the lock. It then checks the condition to see if it is indeed true.
    - If so, it performs its work and releases the lock. If not, it blocks once again with a call to await()

Fourth Benefit: Condition Variables (V)

- The book provides an example of using a Condition Variable to provide a solution to the dining philosopher that
  - a) avoids deadlock
  - b) enables significantly more concurrency than previous solutions
- With the previous solutions, it was highly likely that only one philosopher was able to eat at a time
  - All the other philosophers have one chopstick and are waiting for the other one to become available
- With this solution, at least two philosophers can be eating at one time. That number goes up, if we increase the number of philosophers that are sitting around the table

java.util.concurrent.atomic

- The java.util.concurrent.atomic package contains classes that “support lock-free thread-safe programming on single variables”
  - They provide an easy way to implement single variables that need to be shared between multiple threads and you want to avoid the use of locks to enable maximum concurrency
    - A common use case includes variables that need to be incremented or decremented as events occur without “losing” any of those operations due to race conditions
    - Another use case is the ability to share object references across threads, such that a change in the reference is seen by all threads
      - No “memory barrier” issues, the change is guaranteed to be visible in all threads
- The book updates the counter example using an AtomicInteger. DEMO

Thread Pools

- It takes time to create threads
  - For servers, you want to avoid having to create threads on demand
    - It will slow you down, if you have to wait for a thread to spin up each time you receive a connection
    - Code that creates threads on demand, typically do not have logic to prevent creating one thread per request
      - If that's true and you get 1000s of requests, your code will blindly create 1000s of threads, which will bring your machine to its knees
- A much better strategy is to use a thread pool
  - You create all of the threads you are going to use up front
  - When you need a request to be handled, you wrap it up in a Runnable and hand it to the pool
    - If a thread is available, it executes immediately; otherwise it waits

Using Thread Pools

- The code to create a thread pool is straightforward
  - `public class Task implements Runnable …`
  - `…`
  - `int threadPoolSize = Runtime.getRuntime().availableProcessors() * 2;`
  - `ExecutorService executor = Executors.newFixedThreadPool(threadPoolSize);`
  - `…`
  - `executor.execute(new Task(…));`

- You figure out how large you want your thread pool to be
  - You then create a new thread pool (called an ExecutorService in Java) and pass in that size; when you need something done, call execute()
- How many threads should you allocate
  - Are your tasks computationally intensive? => one thread per core
  - Are your tasks mainly performing I/O? => lots of threads!

Producer-Consumer Example (I)

- The book makes use of a large data file (46 GB!) to provide a real world example of an application that can see performance gains with concurrency
    - I discovered in my testing that there is a problem with page 39904 that causes java.text.RuleBasedBreakIterator to generate an ArrayIndexOutOfBoundsException
        - Rather than try to debug that problem, I restricted my experiments with this file to the first 39,000 pages.
- I also discovered that Java would sometimes run out of memory when processing the XML file. To give the JVM more memory, I would execute the program with this command
    - env MAVEN_OPTS="-Xms2048m -Xmx4096m" mvn -q -e exec:java
- This gives the program 2 to 4 GB of memory to use if needed

Producer-Consumer Example (II)

- The first program, WordCount, is a single threaded version of the program
  - It shows how long it takes to parse the first 39K pages of the XML file
  - On my machine, it averaged 62 seconds across 5 runs
- Nothing special about this program
- The heavy lifting occurs in the Pages.java class, which does the work of instantiating an XML parser and calling it repeatedly for “events” that it recognizes
  - Basically, it looks for <page> tags in the XML file and extracts information about the title and text of the page
    - It ignores all other elements in the XML file
  - When it has a complete page, it stops and returns it to the main() routine, which then counts the words on that page

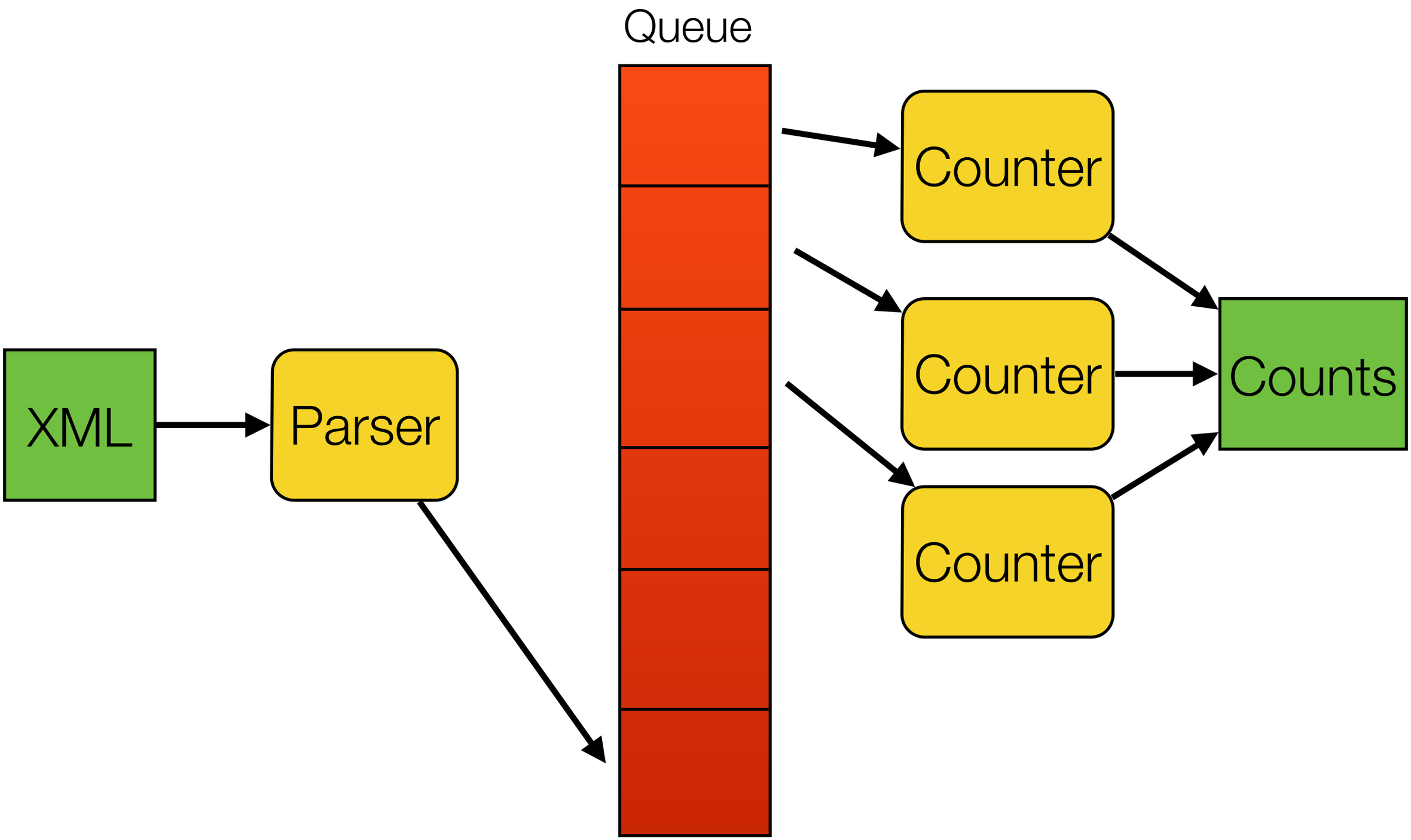
Producer-Consumer Example (III)

- The first attempt to make this program concurrent adopts a typical producer/consumer architecture
  - The parser is in one thread, sticking pages into a bounded queue
    - 100 pages max
  - The consumer is in another thread pulling pages from the queue and counting them
    - A “poison pill” page is inserted into the queue when the parser is done
      - The consumer processes pages until it finds the poison pill and then it terminates
- The program now averages 52 seconds per run; this is an improvement but not by much. Activity Monitor doesn’t report much in the way of concurrency, the CPU utilization is only 110%; the queue provides some separation but not by much

What's taking so long?

- What about queue size? Let's try increasing the queue's size from 100 to 1000? Maybe the producer got blocked on a full queue?
    - No impact, processing time is still 52 seconds (on average)
- Let's determine if the producer is the problem
    - Let's modify the consumer to pull pages off the queue without counting them
        - It takes about 6 seconds to parse 39 thousands pages
- So, in order to speed this program up, we need to see if we can make the counting side (i.e. consumers) of the equation more concurrent

The new approach



Producer-Consumer Example (IV)

- We need to create multiple consumers and have them work together to pull pages off the queue and count the words in parallel
    - The problem here is that we will have a shared resource between the consumers => the hash table that stores the counts for each word
- We will protect the hash table with a static lock shared by all consumers
    - How did we do?
        - Two threads => 109 seconds; 160% CPU utilization
        - Three threads => 133 seconds; 167% CPU utilization
- There's too much contention for the lock that guards the hash table, we're seeing a slow down not a speed up!

Producer-Consumer Example (V)

- Since synchronization is slowing us down, perhaps we can avoid explicit locking and use a hash table designed for concurrent access
  - java.util.concurrent.ConcurrentHashMap may do the trick!
- This data structure tries to allow parallel reads and writes on the hash table without explicit locking
- To do this correctly, however, in the presence of multiple threads, we have to adopt a different style of use
  - When we have an update, we loop until one of these conditions is true
    - The map has not seen this word before and we set the count to 1
    - The map has seen this word and we successfully increment by 1
  - To do this, we use putIfAbsent() and replace() rather than put()

Producer-Consumer Example (VI)

- The insertion code into the ConcurrentHashMap looks like this
  - private void countWord(String word) {
    - while (true) {
      - Integer currentCount = counts.get(word);
      - if (currentCount == null) {
        - if (counts.putIfAbsent(word, 1) == null) {
          - break;
        - }
      - } else if (counts.replace(word, currentCount, currentCount + 1)) {
        - break;
      - }
    - }
  - }
- The loop is there since at any point some other thread may update counts out from underneath us! e.g. currentCount == null, but putIfAbsent() != null

Producer-Consumer Example (VII)

- The results?
  - 2 consumers: 50.5 seconds, 200% CPU utilization
  - 3 consumers: 40.5 seconds, 280% CPU utilization
  - 4 consumers: 40 seconds, 370% CPU utilization
  - 8 consumers: 47.2 seconds (!), 560% CPU utilization
- We can get some speed up, but we still run into contention issues
  - We're seeing the LIVELOCK in action, the threads are active (leading to higher CPU utilization) but they keep failing to update the hash table and so they are spinning around the loop, not making progress
- To get faster speedups, we need to reduce the contention close to zero

Producer-Consumer Example (VIII)

- The final solution is to have each consumer thread maintain its own counts separate from all other threads
  - We can use a plain old HashTable and no need for locks since there will be no contention with this data structure; one HashTable per Consumer
  - When the consumer is done counting words for its pages, it then merges its counts into a shared ConcurrentHashMap in the same way as the previous solution
  - There might be some contention near the end of the run as each Consumer performs its merge, but for most of the run, there will be NO contention while counting

Producer-Consumer Example (IX)

- The results?
  - 2 consumers: 32.6 seconds, 200% CPU utilization
  - 3 consumers: 24.8 seconds, 275% CPU utilization
  - 4 consumers: 22.7 seconds, 330% CPU utilization
  - 8 consumers: 21.1 seconds, 480% CPU utilization
- We don’t see a strict linear speed-up with increased cores BUT
  - this solution is ~3x faster than the single threaded version
  - this solution is ~2x faster than the previous version
- Not bad! There’s still some contention at the end, but significantly reduced

There's more!

- We've covered most of the major classes of functionality hiding in java.util.concurrent
  - BUT there is still more to see
    - Futures
    - CountdownLatch and CyclicBarrier
    - ForkJoinPool (work stealing)
- We'll return to java.util.concurrent later in the semester
  - For now, we'll forge on and look at the functional approach to concurrency, the actor model, and more

Summary

- We reviewed a wide range of functionality provided by the java.util.concurrent library
  - ReentrantLock
  - AtomicVariables
  - Thread Pools
  - Producer Consumer Example
- These constructs help to reduce the sting of programming with Threads and Locks and make it easier to write and reason about concurrent programs

Coming Up Next

- Lecture 9: User Stories, Part 3
- Lecture 10: User Stories, Part 4