Taming Shared Mutability, Part 1

CSCI 5828: Foundations of Software Engineering
Lecture 11 — 02/21/2012

Goals

- Explore the services of java.util.concurrent
    - ExecutorService
    - Callable/Future
    - Atomic<Type>
    - Queues
    - Concurrent data structures
    - Locks
- This is a review of the material in Chapter 4 of our concurrency textbook

Threads in Java prior to JDK 1.5

- Use of Threads in Java used to be painful
  - Low level abstractions
    - Thread with run() routine
    - wait(), notify() to have Threads block and wait for each other
    - synchronized keyword on methods and synchronized blocks
  - Concurrent versions of Java collections that
    - were optimized for safety not performance
      - contention on locks forced programs back to sequential levels of performance

Threads in Java after JDK 1.4

- With the release of JDK 1.5, new features were added to ease the difficulty of dealing with threads
  - All of the low-level mechanisms are still available if needed
  - But, java.util.concurrent provides higher level abstractions
    - Thread Pools, Tasks, and Locks
    - Concurrent data structures that offer both safety and performance

Consider AtomicInteger

- AtomicInteger is an addition to JDK 1.5 that is perfect for those situations in which multiple threads must access and update a shared integer value
  - In Lecture 4, I provided an example of
    - 10 threads each incrementing the value of a shared integer three times
    - The final value of the integer should be 30
    - but we demonstrated that it could be as low as 2 or 3 due to the race condition that occurred between each thread reading/writing the value
  - AtomicInteger provides a way to do those updates without interference
    - more importantly, the synchronized keyword is not used

DEMO

      - in our code or the JDKs; instead finer grain locks are being used to maximize performance

Thread Safety and Performance

- By giving us constructs like AtomicInteger
  - the JDK raises the level of abstraction that we work at
  - while providing us with the best possible performance given the needs of the application
- Hence the primary addition of java.util.concurrent is the ExecutorService
  - We've seen the ExecutorService in action previously
    - Concurrent Portfolio Calculator and Concurrent Prime Finder
  - We'll look at it in further depth with a new example
    - Concurrent File Size Calculator
  - and examine various constructs provided by java.util.concurrent

Coordinating Threads (I)

- A key challenge in the design of concurrent systems is the coordination of threads
  - We may want to
    - start them
    - wait for them to finish
    - assign tasks to them
    - retrieve results from them
    - allow threads to exchange data
    - etc.

Coordinating Threads (II)

- With the ExecutorService, the most typical case now involves
  - submitting a task to a thread pool of type Callable
  - receiving a Future in response
  - when ready, calling get() on the Future to retrieve the result
- Let's see this in action with the File Size Calculator
  - First, let's take a look at the sequential version of this program
    - See Section 4.2 of your concurrency textbook for details
  - Design is straightforward; recursive function that returns either the size for a single file or for directories, the combined size of all of its children

Disk Cache

- With programs that target the disk, performance will vary
  - The first time through a particular section of the disk, the time will be slower than subsequent runs on the same section of the disk
- The reason for this is the disk cache
  - The operating system will
    - take the most recently read sections of disk
    - and cache them in memory
    - under the assumption that they will be read again fairly soon
- The difference may not be major but it will be there
  - First run sequential on /usr: 34.1 seconds; Second run: 30.9 seconds

First Stab at Concurrency

- Creates a thread pool of 100 threads
- Makes use of recursive function to calculate size of files and directories
  - If its handed a file, return the file size
  - If its handed a directory
    - loop through children
      - submit() a task to the thread pool to calculate the size of the child
        - Each task is a Callable<Long>
        - Thread pool returns a Future<Long> that gets added to an array
    - loop through array calling get() on each Future to add up subtotals
    - return the result

Result? DEADLOCK!

- This approach to the program has a flaw that appears on “deep directories”
  - Each task adds new tasks to the thread pool and then waits for those tasks to return
  - That means that the calling task is STILL ON THE POOL
    - blocked waiting for its subtasks to complete
  - If your directory has lots of subdirectories (more than 100 in this case)
    - You can get into the situation where each of the 100 threads in the thread pool are blocked waiting for subdirectory calculations to complete
      - when this happens, the program deadlocks
        - or thanks to the timeout that we set, eventually the timeout fires and the program terminates

Discussion

- This problem is unfortunately because
  - the approach is straightforward and understandable
    - you'd likely come up with it on a first pass design
- But, a machine's resources are finite
  - you might be able to make this code work on more directories by upping the number of threads
  - but that approach is not generic
    - eventually you'll run into the limit concerning the number of threads the operating system will allow a single process to create
    - and you'll be stuck

New Approach: Find Directories, Total Later

- To make progress, we need an approach that
  - submits tasks for sub-directories
  - but doesn't require the submitting task to hang around for the results
- New Approach
  - Create a data structure that holds the total size of a directory's files and a list of all of that directory's sub-directories
  - Tasks now calculate the size of files in their assigned directory and create a list of all subdirectories; allowing them to complete and not stick around
  - The main thread takes care of submitting new tasks and totaling results
    - Performance: First Run: 22.7 seconds; Second Run: 12.4 seconds (!)

Terrific Results But...

- increased complexity!
  - We got great results but the approach we used is not intuitive
    - Creating a class to store partial (immutable) results
    - Creating the function executed by tasks such that it completes quickly
    - Adopting a while loop strategy in main to iterate while there were directories to process
      - and then ensuring that the while loop would not terminate until all directories had been processed
- Let's look at features that java.util.concurrent has that might reduce the complexity of the code

CountDownLatch (I)

- The next approach examines the use of a CountDownLatch
  - plus it relaxes our constraint to avoid shared mutablity
- but it achieves the same results with simpler code
  - Simplicity is not to be discounted
    - it has significant impacts on the ability to maintain software systems

CountDownLatch (II)

- What's a CountDownLatch?
  - It is a synchronization aid to help coordinate threads
  - It maintains a count and has three primary methods
    - CountDownLatch(n) - creates the latch with a specific count
    - await() - block the calling thread until the latch's count == 0
    - countDown() -- decrement the count of the latch
- Typical scenario:
  - create a bunch of threads and start() them
  - but don't let them run() until some point in the future
    - i.e. have their first line in run() call await() **DEMO**

New Approach

- Instead of returning subdirectories, we let each task update two shared variables
    - each an instance of AtomicLong (like AtomicInteger but stores long value)
- One AtomicLong stores the total file size
- The second AtomicLong stores the number of "pending file visits"
    - This value gets incremented each time we find a subdirectory to visit
    - It gets decremented each time we are done processing a subdirectory
    - When this value equals zero, we call countDown() on the latch
- The main thread initializes the latch to a value of 1, starts the directory search, and calls await()

Performance

- First Run: 24.5 seconds
- Second Run: 10.7 seconds
- Comparable performance to previous approach
  - but with simpler code
- We actually anticipate that this approach would be slightly slower than the previous approach due to the extra thread synchronization
  - Each call to AtomicLong involves thread synchronization
    - threads do not necessarily block
      - (only happens when there is contention)
    - but a monitor of some sort will be checked and that slows things down

Third Approach: Queue (I)

- We have seen two approaches for exchanging data between threads
  - Callable/Future and Atomic<Type>
- both techniques ensured that we could pass information between threads
- A third approach is to use a data structure such as a queue to pass information between threads
  - as long as their is space in the queue, producers will not block
  - as long as their are items on the queue, consumers will not block
  - contention will occur only when the queue is full (producers) or when it is empty (consumers)

Third Approach: Queue (II)

- This version of the program creates a blocking queue with 500 slots
- An atomic long is used to keep track of pending file visits
- Tasks traverse the directories as normal, adding file sizes to the queue and updating the atomic long as they submit more tasks to the thread pool
- The main program kicks off the traversal and then sits in a loop
  - that reads items off the queue until there are no more file visits pending and the queue is empty
- Performance:
  - First Run: 24.6 seconds; Second Run: 10.9 seconds
  - Same performance, just slightly different abstractions, perhaps simpler
    - not by much

Java 7: Fork-Join API

- The latest version of Java comes with a new type of thread pool and task
  - ForkJoinPool and ForkJoinTask
- The key benefit of this new thread pool is that threads can steal tasks generated by other active tasks
  - This solves the problem we encountered with the first approach to the concurrent file size calculator
  - When a task generates a bunch of other tasks and blocks, it’s thread can let it go and work on the other tasks
    - The book shows that this approach is the fastest of all seen so far
    - Unfortunately, I can’t run Java 7 (yet) on MacOS X

Performance Vs. Safety (I)

- Another problem addressed by the java.util.concurrent library is the performance of certain data structures when accessed by multiple threads
  - In the past, you had to synchronize threads before they updated a shared data structure, such as a hash map or a queue
    - to ensure that you didn't access a key that was being removed by another thread
  - In addition, you were not allowed to change the collection while iterating over it
    - this problem leads to weird strategies, where you have to iterate over a map or list to search for items you wanted to delete
      - but wait until after the iteration was over before you performed the deletions

Performance Vs. Safety (II)

- The primary problem with this past approach was that it valued safety over performance
  - If you had a bunch of threads accessing the data structure
    - performance slowed to a crawl since they all had to take turns accessing and modifying the data structure
- But, there are certain situations where “eventual consistency” is fine
  - that is, the fact that Thread A doesn’t see the key being inserted by Thread B during its current iteration is fine
    - since Thread A will see it on its next iteration through the map
  - furthermore, the fact that Thread A processed a key that was being removed by Thread B is fine, since it will catch up on the next iteration
- Think Facebook: it’s okay if the number of people who “like” a post is not current

Performance Vs. Safety (III)

- For these situations, a concurrent version of these data structures vastly improves performance in concurrent programs
  - and allows the data structure to be modified during an iteration leading to simpler code
- How much faster? The book provides an example of a program that
    - has a task that will randomly read, insert, and delete keys into a map
    - it takes a read on how long the task takes for a single thread to complete
    - and then compares performance as the number of threads goes from one thread to sixteen;
  - Throughput can be 30% higher with multiple threads using concurrent data structures and can be 70% slower with synchronized data structures

DEMO

Lock vs synchronized

- The last improvement that java.util.concurrent provides is related to locking
  - Before java.util.concurrent, locking was provided by
    - synchronized methods
    - synchronized blocks
    - wait() and notify() -- note: so awful, I’m not going to cover them
  - These methods are hard to get right and are slow
    - adding synchronized to a method can cause it to run 10 to 100 times slower!
- To combat this, java.util.concurrent.locks provides the Lock interface
  - different Locks are then provided by various concrete implementations

Lock methods

- lock() -- acquires the lock
- tryLock() -- acquires lock only if it is free
- tryLock(...) -- acquires lock but will time out if the lock is not available
    - tryLock() is an improvement over synchronized's all or nothing approach
- unlock() - release the lock

- One last method is newCondition() -- this produces a condition object associated with this lock that allows threads to block on a lock until a given condition is true
    - We may return to this style of concurrent programming later in the semester; see java.util.concurrent.locks.Condition for more details

Types of Locks

- Given that Lock is an interface, what types of locks are available?
    - Just one: ReentrantLock
- This one covers most of the bases
- What's does "reentrant" mean in this context
    - If Thread A acquires Lock B
        - if Thread C tries to acquire Lock B, it blocks
        - but if Thread A tries to acquire Lock B again, it does so and continues
        - the lock will keep track of how many times Thread A calls lock() and will look for the corresponding unlock() calls

ReadWriteLock

- java.util.concurrent.locks also provides a ReadWriteLock interface that simply groups two Locks together, a Read lock and a Write lock
  - The package provides only one implementation of this interface
    - ReentrantReadWriteLock
  - that provides standard semantics
- This type of lock allows a resource to be accessed by lots of readers and writers
  - writers will block until all readers are done;
  - readers will block if there is a writer updating the resource
    - otherwise multiple readers can acquire a read lock at the same time

Use of the Lock interface

- The Lock interface allows us
  - to stop using the synchronized block technique
    - discussed at the beginning of Lecture 6 and used in Homework 2
- Instead, we use code like the following to create transactions for threads
  - aMonitor.lock();
  - try {
    - //...
  - } finally {
  - aMonitor.unlock();
  - }

Each thread that requires a transaction to access to a resource, calls lock(), performs **multiple calls** on the resource and then calls unlock() in a finally block

Use of finally ensures the lock is released no matter what happens during the transaction

with the exception of deadlock, of course

Example

- The book provides an example of using the Lock interface
  - Two threads performing deposits and withdrawals on bank accounts, and transfers between bank accounts
    - Transfers require “transaction semantics”
  - We must also make sure that Thread A doesn’t acquire Account A at the same time that Thread B acquires Thread B because deadlock can occur if they then both need the other account
    - To prevent that, it ensures that threads acquire locks on the accounts in the same order
  - Individual methods acquire the lock

DEMO

    - this does not block a transaction since the lock is reentrant

Summary

- We've examined some of the problems and the inflexibility with the old approach to concurrency that characterized JDKs prior to version 1.5
  - Low level Thread, Runnable, run(), wait(), notify(), synchronized
  - Poor performance of synchronized data structures
- And then examined the benefits of the new approach to concurrency embodied in java.util.concurrent and its related packages
  - Thread pools and multiple ways to coordinate threads
  - Concurrent data structures
  - fine grain, flexible locking with the new Lock interface and its reentrant implementation

Coming Up Next

- Lecture 12: Taming Shared Mutability, Part 2
- Lecture 13: More on Cucumber: Steps, Scenarios, & Debugging
- Lecture 14: Review for Midterm