Java Concurrency Framework

Sidartha Gracias
Executive Summary

• This is a beginners introduction to the java concurrency framework
• Some familiarity with concurrent programs is assumed
  – However the presentation does go through a quick background on concurrency
  – So readers unfamiliar with concurrent programming should still get something out of this
• The structure of the presentation is as follows
  – A brief history into concurrent programming is provided
  – Issues in concurrent programming are explored
  – The framework structure with all its major components are covered in some detail
  – Examples are provided for each major section to reinforce some of the major ideas
Concurrent in Java - Overview

- Java like most other languages supports concurrency through thread
  - The JVM creates the Initial thread, which begins execution from main
  - The main method can then spawn additional threads

Thread Basics

- All modern OS support the idea of processes – independently running programs that are isolated from each other
- Thread can be thought of as light weight processes
  - Like processes they have independent program counters, call stacks etc
  - Unlike Processes they share main memory, file pointers and other process state
  - This means thread are easier for the OS to maintain and switch between
  - This also means we need to synchronize threads for access to shared resources
Threads Continued...

• So why use threads?
  – Multi CPU systems: Most modern systems host multiple CPU’s, by splitting execution between them we can greatly speed up execution
  – Handling Asynchronous Events: Servers handle multiple clients. Processing each client is best done through a separate thread, because the Server blocks until a new message is received
  – UI or event driven Processing: event handlers that respond to user input are best handled through separate threads, this makes code easier to write and understand
Synchronization Primitives in Java

- How does the java language handle synchronization
  - Concurrent execution is supported through the *Thread* class.
  - Access to shared resources is controlled through
    • Synchronized objects
    • Synchronized methods
  - Locking and Unlocking shared resources is handled automatically
  - However only one (duh.) thread can hold a shared object at one time
  - A thread that cannot acquire a lock will block.
Writing Concurrent Programs

• But it’s not so easy..
  – Writing non trivial concurrent programs is not so straightforward, here are a few of the issues you could run into

• Deadlock: Two or more threads waiting for each other to release a resource

```
Thread A
1: acquire A
3: acquire B

Thread B
1: acquire B
3: acquire A
```
Concurrency issues continued..

- Race Conditions: Non deterministic output, that depends on the order of thread execution.

```
Thread A
While(true)
Print 1
```
```
Thread B
While(true)
Print 2
```

- Output:
  - 11111111... or
  - 12121212... or etc
Concurrency issues continued..

• Starvation: A slow thread being starved of a resource by a fast thread

• This brings us to thread safe classes
  – A class that guarantees the internal state of the class as well as returned values from methods are correct while invoked concurrently from multiple threads
Where does this leave us?

• This shows us writing concurrent programs is hard and prone to bugs
• The synchronization primitives java provides are of too low a granularity and do not scale with program complexity
• It would be nice if we could abstract some of this complexity away
• Further many of the problems encountered writing parallel programs are common across a wide area
• We really should not have to reinvent the wheel each time we want to write a concurrent program
• This is where the Java Concurrency Framework comes in..
Framework Overview

- The Concurrency utilities includes the following:

1. **Task Scheduling Framework**: The Executor is a framework for handling the invocation, scheduling and execution of tasks.

2. **Concurrent Collection**: Concurrent implementations of commonly used classes like map, list and queue (no more reinventing the wheel 😊).

3. **Atomic Variables**: classes for atomic manipulation of single variables, provides higher performance than simply using synchronization primitives.

4. **Locks**: High performance implementation of locks with the same semantics as the `synchronized` keyword, with additional functionality like timeouts.

5. **Timers**: Provides access to a nanosecond timer for making fine grained timing measurements, useful for methods that accept timeouts.
Advantages of using the framework

• **Reusability and effort reduction**: Many commonly used concurrent classes already implemented

• **Superior performance**: Inbuilt implementation highly optimized and peer reviewed by experts

• **Higher reliability, less bugs**: Working from already developed building blocks lead to more reliable programs

• **Improved maintainability and scalability**: Reusable components leads to programs that are easier to maintain and scale much better

• **Increased productivity**: Developers no longer have to reinvent the wheel each time, programs easier to debug, more likely to understand standard library implementations
The Executor Framework

• The Executor framework provides a mechanism for invoking, scheduling, and executing tasks according to a set of policies
• Also provides an implementation of thread pools through the `ThreadPoolExecutor` class
• This provides a way to decouple task submission from task execution policy
• Makes it easy to change task execution policy
• Supports several different execution policies by default, and developers can create Executors supporting arbitrary execution policies
Interlude: Thread pools

• Consider writing a web server that handles an arbitrary number of clients
• One way of implementing this is to spawn a new thread for each client that makes a request
• However under high load this could crash the server, due to the large amount of resources used to create and maintain the threads
• Worse we are creating far more threads than can be handled by the server, probably most threads will sit around waiting for CPU time.
Thread pools continued...

• A smarter way to handle this is to use the idea of a thread pool
• We create a fixed number of threads called a thread pool
• We use a queue data structure into which tasks are submitted
• Each free thread picks a task of the queue and processes it
• If all threads are busy, tasks wait in queue for threads to free up
• This way we never overload the server, and get the most efficient implementation
Executor Framework Continued...

- The Executor interface is fairly simple
  - it describes an object, which executes Runnables

```java
Public interface Executor {
    void execute(Runnable task)
}
```

- A class that wishes to use the Executor framework, must implement the Executor interface and provide an implementation for execute

```java
Class ImpExecutor implements Executor {
    void execute(Runnable t)
    {
        new Thread(t).start  //This executor creates a new thread for each task submitted
    }
}
```
The Executor framework comes with several implementations of Executor that implement different execution policies

- execution policy determines when a task will run, its priority and other associated parameters

1. **Executor.newCachedThreadPool**: Creates a thread pool of unlimited size, but if threads get freed up, they are reused

2. **Executor.newFixedThreadPool**: Create a thread pool of fixed size, if pool is exhausted, tasks must wait till a thread becomes free

3. **Executor.newSingleThreadExecutor**: Creates only a single thread, tasks are executed sequentially from the queue
```java
• class WebServer {
•   Executor execs = Executors.newFixedThreadPool(7);
•   public static void main(String[] args) {
•       ServerSocket soc = new ServerSocket(80);
•       while (true) {
•           Socket conn = soc.accept();
•           Runnable r = new Runnable() {
•               public void run() {
•                  handleRequest(conn);
•               }
•           };
•           pool.execute(r);
•       }
•   }
• }
```

Example code modified from the book “Java Concurrency in Practice”
Code Walkthrough

• That was a quick example of how to use the executor framework and thread pools
• We have used an inbuilt executor “Fixed ThreadPool” to create a pool of 7 threads
• We create a runnable to handle new connections
• We then hand the runnable to the executor for execution
• Thus task execution is decoupled from task submission
Concurrent Collections

• The Concurrency framework provides implementation of several commonly used collections classes optimized for concurrent operations

  1. public interface Queue<E> extends Collection<E>: A collection class that hold elements prior to processing, generally orders element in a FIFO manner (however can also be LIFO etc). Supports the following functionality
     • Offer method inserts an element if possible else return false
     • remove() and poll() return and remove the head of the queue
     • element() and peek() return but do not remove the head of the queue
1. The **BlockingQueueInterface** extends this interface
   - Supports all queue functionality, and additionally waits for a queue to become non-empty before retrieving an element and waits for space to become available when storing an element
   - Intended primarily for use as a producer-consumer queues
   - BlockingQueue implementations are thread safe, with all queue operations achieved atomically
   - Some bulk collection operations are not atomic, but will fail if other threads interfere before the operation is completed
Concurrent Collections Continued..

1. public interface ConcurrentMap<K,V> extends Map<K,V>: an extension to the Map interface that provides support for concurrency
   – Supports concurrent putIfAbsent, remove and replace methods
   – putIfAbsent(key, value): If the specified key is not associated with a value associate it with a value. Performed atomically
   – remove(key, value): Removes entry for a key if associated with value. Performed atomically
   – remove(key, oldValue, newValue): Replaces entry for a key if associated with oldValue with newValue. Performed atomically
public class consumerthread implements Runnable {
    private static int capacity;
    private BlockingQueue<Integer> intqueue;

    public consumerthread(BlockingQueue<Integer> queue, int cap) {
        capacity = cap;
        this.intqueue = queue;
    }

    public void run() {
        int num;
        for (int i = 0; i < capacity; i++) {
            try {
                num = intqueue.take();
                if (num == -1)
                    break;
                System.out.println("The Square of " + num + " is " + num * num);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class producerthread implements Runnable {
    private static int capacity;
    private BlockingQueue<Integer> intqueue;

    public producerthread(BlockingQueue<Integer> queue, int cap) {
        capacity = cap;
        this.intqueue = queue;
    }

    public void run() {
        for(int i = 0; i<capacity-1; i++) {
            try {
                intqueue.put(i);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        try {
            intqueue.put(-1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
public class test {

    public static void main(String [] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>(100);
        consumerthread consumer = new consumerthread(queue, 100);
        producerthread producer = new producerthread(queue, 100);
        new Thread(consumer).start();
        new Thread(producer).start();
    }

}
Code Walkthrough

• That was an example of using blocking queues to implement producer consumer relationships
• The producer class fills in a queue of integers
• The consumer class pulls integers of this queue and finds the square
• If the queue is empty a take() blocks, if full a put() blocks
• We could extend this to use a thread pool for the producers
Synchronizer Classes

- Includes a set of classes that aid synchronization between threads
- Includes semaphores, mutexes, barriers, latches and exchangers

Semaphores

- Implements a counting semaphore (Djikstra Counting Semaphore)
  - you can think of a counting semaphore as holding a certain number of permits
  - If the permits are all used up, succeeding threads must wait for one to become available.
  - The same thread can hold multiple permits
  - Useful for imposing a resource limit
  - A thread invokes the acquire() method to obtain a permit, acquire() blocks if no permits available
  - A thread invokes release() to release a permit back to the pool
Synchronizer Classes Continued...

Mutexes

• Special case of the semaphore
• Stands for mutual exclusion semaphore
• A Semaphore with a single permit
• Used to gain exclusive access to a resource
• Similar to a lock with one key difference: can be released by a thread other than the one holding the mutex
Synchronizer Classes Continued...

**Barriers (Cyclic Barrier)**

- A synchronization aid that allows a set of threads to all reach a common point before proceeding.
- Useful in programs that involve a fixed number of threads that must wait for each other at some point.
- Called cyclic because barrier can be reused.
- A thread signals it has reached the barrier by calling `CyclicBarrier.await()`.
- The threads blocks until all other threads have reached the barrier (which signal this the same way).
- Can specify a timeout at which time the thread will stop blocking.
Latches (CountDown Latches)

- Similar to a Cyclic Barrier, used to synchronize a set of threads that have a task divided amongst themselves.
- The CountDownLatch is initialized with a given count.
- On reaching the synchronization point a thread can invoke the countdown() method which decrements the internal count.
- Threads that call the await() method will block until the count has reached zero.
- At this point all waiting threads are released.
- Calling await() after count has reached zero, has no effect (the thread will continue executing).
- Threads that call countdown() are not required to call await() (and can proceed executing).
- This is useful when the main thread has split execution between a number of worker threads.
- The worker threads call countdown() on completing their task, the main thread blocks on await() until count has reached zero.
Exchangers

- A synchronization point at which two threads can exchange objects
- Can be thought of as a CyclicBarrier with a count of two plus allowing an exchange of state at the barrier
- On calling the exchange() method the thread provides some object as input
- The exchange() method returns the object entered as input by the second thread to the calling thread
- Useful for example in the case where one thread is filling a buffer (filler thread) and the other emptying (emptying thread)
- On calling exchange() the filler thread is passed an empty buffer
- The emptying thread obtains the newly full buffer from the filler thread
class cyclicbarriertest {
  void compute(Task t, int num) {
    final CyclicBarrier barrier = new CyclicBarrier(num, //two inputs,num_threads
      new Runnable() { //runnable tasks
      for (int i = 0; i < nThreads; ++i) //split computation between threads
        {
          final int id = i;
          Runnable worker = new Runnable() { //new worker
            final Segment segment = t.createSegment(id);
            public void run() {
              try {
                segment.update();
                barrier.await();
              }
              catch (Exception e) { return; }
            }
          };
          new Thread(worker).start();
        }
      }
    }
  }
}

Example code modified from the book “java concurrency in practice”
Code Walkthrough

• Example program that shows the usage of cyclic barriers
• Our function takes as input as task t and number of threads num
• It creates a cyclic barrier that takes two inputs num, and task
• Task execution is split among a group of worker threads
• Each worker computes a portion of the problem and waits to see if all other threads have finished (using a cyclic barrier to synchronize)
Atomic Variables

• A set of classes that provide lock free thread safe programming on single variables
• Provides for atomic conditional updating of single variables
  – Boolean compareAndSet(expected, update): atomically set value of variable to update value provided it holds expected value
  – WeakCompareAndSet is more efficient but any invocation of this method may fail, with the guarantee that repeated invocations will eventually succeed
  – Also provides methods for getting and unconditionally setting values
• Instance of classes AtomicInteger, AtomicBoolean, AtomicLong, AtomicReference provide access and update to single variables of that type
Locks

• A generalization of built-in locking behavior, but with several different types of locks with superior functionality

• Reentrant lock: This has the same overall semantics as the “synchronized” keyword, with the exception that locks must be explicitly obtained and released
  – Provides better throughput than synchronized when multiple threads are vying for the same lock

• ReadWrite lock: A pair of locks, one for reading and one for writing. The read lock can be simultaneously held by multiple threads. The write lock is exclusive

• AbstractQueuedSynchronizer: provides a framework for implementing locks and related synchronizers
• Lock syntax

• `lock.lock();`

• `try {`
  • `// operations protected by lock`
  • `}`

• `catch(Exception ex) {`
  • `}`

• `finally {`
  • `lock.unlock();`
  • `}`
Summary

• So What did we cover
  – Thread basics
  – Java language support for concurrency
  – Examples of why concurrency is hard
  – Overview of the Java Concurrency framework
  – A look at the principle classes in the framework and the functionality offered
  – A few examples covering some of the major concepts

• For a more in depth look at the Concurrency Framework have a look at the excellent “Java Concurrency in Practice” book by Brian Goetz et al.