Java Concurrency Framework

-Aditya Bhave
CSCI-5448 Graduate Presentation
04-01-2011

About Me…

- MSEE at CU finishing up in Summer 11
- Work full time at Alticast Inc
- Background in Embedded Systems
  - Media Processing
  - Middleware
  - Cable TV industry

Agenda

- Concurrency in C and Java
- Need for a framework
- Threads
  - Creation and Starting
- Synchronization
- Executor Interfaces
- Thread Pools
- Future Interface
- More Synchronizers
- AbstractQueueSynchronizer framework
- BlockingQueueInterfaces
- References

Concurrency in general

- Sequential programs: execute a single stream of operations
- Concurrent program: several streams of operations may execute concurrently
  - Streams can communicate and interfere with one another
  - Each such sequence of instructions is called a **thread**
  - Operations in threads are interleaved in an unpredictable order. Operations within a thread are strictly ordered
  - Different than parallel execution
  - Difficult to design, test, write, reason about, debug and tune

Concurrency contd …

- Things to worry about:
  - Shared data
    - Locking
    - Visibility
    - Atomicity
  - Coordination
    - Communication between Threads
  - Performance
    - Deadlock
    - Spin wait
    - Lock Contention

Concurrency in C

- Locking in C:
  - Initialize mutex
  - Explicitly guard shared data “shared”:
    - pthread_mutex_lock(m1);
      shared++;   // This operation should be atomic
      pthread_mutex_unlock(m1);
- Increased development overhead
  - Explicitly create threads, remember thread id’s
  - Explicit locking and unlocking
  - Remembering which thread holds which locks
  - Can break modularity completely
- Need for abstracting the internals of synchronization and atomicity from developers

Java Concurrency framework

- Package java.util.concurrent to the rescue
- Important aspects
  - Defining and starting threads
  - Synchronization
  - Liveness
  - Immutable objects
  - High Level Concurrency
    - Lock Objects
    - Executors
    - Thread Pools
    - Atomic Variables

Threads

- Method I: Imlementing Runnable
  - public class PrimeRun **implements Runnable** {

```
    long minPrime;
    public PrimeRun(long minPrime) {
            this.minPrime= minPrime;
    }
    public void run() {
            // Compute prime larger than minPrime
    }
    public static void main(String[] args) {
            PrimeRun pr= new PrimeRun(7);
            new Thread(pr).start();
    }
}
```

Threads

- Method II: Extending Thread
  - public class PrimeThread **extends Thread** {

```
    long minPrime;
    public PrimeThread(long minPrime) {
            this.minPrime= minPrime;
    }
    public void run() {
            // Compute prime larger than minPrime
    }
    public static void main(String[] args) {
            PrimeThread pt= new PrimeThread(7);
            pt.start();
    }
}
```

  - Thread class itself implements **Runnable**

Threads

- Method I better:
  - more general, because the Runnable object can subclass a class other than Thread
  - more flexible
  - applicable to the high-level thread management APIs
- Invoke start method to start the thread
- Run method of thread object can be invoked by current thread without starting new thread. (Error Prone)

Synchronization

- Memory Consistency Errors
- Ex: Threads A and B increment shared variable "ct"
  - Race condition !!!
    - A fetches ct = 0
    - B fetches ct = 0
    - A computes the value ct++ = 1
    - A stores the value 1 in ct
    - B computes new value ct++ = 1
    - B stores the value 1 in ct
  - Make "ct" atomic. i.e. All threads should have same view of "ct" and access it in a synchronized manner

Synchronization

- Thread Interference
  - when two operations, running in different threads, but acting on the same data, interleave
  - Ex: Thread A and B share variable int c;
    - Thread A: Retrieve c.
    - Thread B: Retrieve c.
    - Thread A: Increment retrieved value; result is 1.
    - Thread B: Decrement retrieved value; result is -1.
    - Thread A: Store result in c; c is now 1.
    - Thread B: Store result in c; c is now -1.

Synchronization

- Synchronized Methods
  - public *synchronized* void increment() { c++; }
  - public *synchronized* void decrement() { c--; }

  - It is not possible for two invocations of synchronized methods on the same object to interleave

  - Automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object

  - Thread invoking synchronized method automatically acquires the *intrinsic lock* for that method's object and releases it when the method returns

Synchronization

- Synchronized Statements
  - public class MsLunch {

```
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc1()
    { synchronized(lock1) { c1++; } }
    public void inc2()
    { synchronized(lock2) { c2++; } }
}
```

- To make update of c1 independent of update of c2, but still keep both updates synchronized.
- Fine grained synchronization

Atomic Variables

- java.util.concurrent.atomic provides classes methods to have atomic variables
- Ex: in previous example, replace int c with atomic integer c:
  - private AtomicInteger c = new AtomicInteger(0);
  - Replace “c++” by c. incrementAndGet()
  - Replace “c--” by c.decrementAndGet();
  - You can obtain value of c by c.get();
- Has performance advantages.
- Neatly *encapsulate* operations; prevent inadvertent access to data from unsynchronized code
- Implemented using the fastest native construct available on the platform (compare-and-swap etc.)

Executors

- Separate the thread management and creation from the rest of the application
- Objects that **encapsulate** these functions are known as executors
- Three interfaces:
  - Executor
  - ExecutorService
  - ScheduledExecutorSercice

Executor Interface

- Simple interface that supports launching new tasks
- Provides a single method, execute
- Runnable object "r" and Executor object "e" then,
  - (new Thread(r)).start(); ➔ e.execute(r);
- Can create a new thread and launch it immediately
- More likely to use an existing worker thread to run "r"

ExecutorService

- Supplements executor with more versatile **submit** method
  - **submit** accepts *Runnable* objects, but also accepts *Callable* objects, which are similar to Runnable but allow the task to return a value
  - **submit** returns a *Future* object
    - used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks
- ExecutorService provides methods for submitting large collections of Callable objects
- It provides methods for managing the shutdown of the executor

ScheduledExecutorService

- Supplements the methods of its parent ExecutorService with **schedule** method
  - **schedule** executes a Runnable or Callable task after a specified delay
- Defines **scheduleAtFixedRate** and **scheduleWithFixedDelay**
  - To execute specified tasks repeatedly, at defined intervals

Thread Pools

- Thread pools consist of *worker threads*
- Minimize the overhead due to thread creation
- Avoid allocating and de-allocating many thread objects. Reduce significant memory management overhead
- *Fixed thread pool*
  - Always has a specified number of threads running
  - If a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread.
  - Tasks are submitted to the pool via an internal queue
  - Queue holds extra tasks whenever there are more active tasks than threads

Thread Pools

- Create an **executor** that uses a fixed thread pool
  - invoke the **newFixedThreadPool** factory method in java.util.concurrent.Executors
- Additional Methods:
  - **newCachedThreadPool** method creates an executor with an expandable thread pool, suitable for applications that launch many short-lived tasks
  - **newSingleThreadExecutor** method creates an executor that executes a single task at a time
- FactoryMethod design pattern in practice

Putting it all together … in code

- public class RunnableTester {

```
        public static void main(String[] args) {
        // create and name each runnable
        SomeTask task1 = new SomeTask("thread1");
        SomeTask task2 = new SomeTask("thread2");
        // create ExecutorService to manage threads
        ExecutorService    threadExecutor = Executors.newFixedThreadPool(2);
        // start threads and place in runnable state
        threadExecutor.execute(task1); // start task1
        threadExecutor.execute(task2); // start task2
        // shutdown worker threads
        threadExecutor.shutdown();
        System.out.println("Threads started, main ends\n"); }
} // end class RunnableTester
```

- public SomeTask implements Runnable{

```
        // do something here
}
```

Putting it all together … in code

```
public class MyScheduledExecutorService {
ScheduledExecutorService scheduler= Executors.newScheduledThreadPool(1);
public void beepForAnHour() {
        final Runnable beeper = new Runnable() {
                        public void run() { System.out.println("beep");
                }
};
ScheduledFuture<?> beeperHandle = scheduler.scheduleAtFixedRate( beeper, 1, 5,
SECONDS);
// Schedule to beep every five seconds
scheduler.schedule(new Runnable() {
                public void run() { beeperHandle.cancel(true); }
                }, 60 * 60, SECONDS);
}
public static void main(String args[]) {
        MyScheduledExecutorService mses = new MyScheduledExecutorService();
        mses.beepForAnHour();
}
}
```

Future Interface

- Future represents the result of an asynchronous computation
- Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation
- The result can only be retrieved using method get when the computation has completed
- Cancellation is performed by the cancel method

Future Interface

- interface ArchiveSearcher {
  String search(String target); }
  class App {
  ExecutorService executor = ...
  ArchiveSearcher searcher = ...
  void showSearch(final String target) throws
  InterruptedException { Future<String>
  future = executor.**submit**(new
  Callable<String>() {
  public String call() {
  return searcher.search(target); }
  }
  );
  displayOtherThings(); **// do other things while searching** try {
  displayText(future.get()); // use future }
  catch (ExecutionException ex) { cleanup(); return; } }
  }
- The **ScheduledFuture**<?> in the previous code states that you would like to use a ScheduledFuture for the sake of cancellability but not provide a usable result. It is return value of ScheduledExecutorService methods.

More Synchronizers…

- Semaphores
  - Semaphore maintains a set of permits.
  - acquire() blocks if necessary until a permit is available, and then takes it
  - release() adds a permit, potentially releasing a blocking acquirer
  - Used to restrict the number of threads than can access some (physical or logical) resource
  - Semaphore ***encapsulates*** the synchronization needed to restrict access to the resource pool, separately from any synchronization needed to maintain the consistency of the pool itself
  - Methods provided to ensure fairness, checking for permits, acquire in non-blocking manner and get number of threads queued to acquire that semaphore object

More Synchronizers…

- Cyclic Barrier
  - A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point
  - barrier is called *cyclic* because it can be re-used after the waiting threads are released
  - await()
    - Waits until all parties have invoked await on this barrier
    - If the current thread is not the last to arrive then it is disabled for thread scheduling purposes and lies dormant until:
      - The last thread arrives
      - Some other thread interrupts the current thread
      - Some other thread interrupts one of the other waiting threads
      - Some other thread times out while waiting for barrier
      - Some other thread invokes reset() on this barrier

More Synchronizers…

- Thread starting code in some class:
  - barrier = new **CyclicBarrier** (**N**, new Runnable()

```
barrier = new CyclicBarrier (N, new Runnable()
                                  { public void run()
                                        { some_random_function(...); }
                                  });
for (int i = 0; i < N; ++i) {
    new Thread(new some_random_class(i)).start();
}
```

- Code for threads to wait on cyclic barrier
  - class some_random_class implements Runnable {

```
class some_random_class implements Runnable {
    public void run() {
        while (!done()) {
            function1(arg1);
            try { barrier.await();
            } catch (InterruptedException ex) { return; }
              catch (BrokenBarrierException ex) { return; }
        }
    }
}
```

- **some_random_function()** is executed each time a barrier is encountered and tripped by a thread

AbstractQueueSynchronizer class

- Provides a framework for implementing blocking locks and related synchronizers (semaphores etc. that rely on first-in-first-out (FIFO) wait queues

- Nearly any synchronizer can be used to implement nearly any other
  - it is possible to build semaphores from reentrant locks, and vice versa

- But…
  - Doing so often entails enough complexity, overhead, and inflexibility
  - It is conceptually unattractive. If none of these constructs are intrinsically more primitive than the others, developers should not be compelled to arbitrarily choose one of them as a basis for building others.

- Instead, JSR166 establishes a small framework
  - centered on class AbstractQueuedSynchronizer
  - Provides common mechanics that are used by most of the provided synchronizers in the package

AbstractQueueSynchronizer class

- Generic Synchronizers have acquire() and release() methods in some form
  - Eg: methods Lock.lock, Semaphore.acquire, CountDownLatch.await, and FutureTask.get all map to *acquire* operations in the framework
- Support for these operations requires the coordination of three basic components:
  - Atomically managing synchronization state
  - Blocking and unblocking threads
  - Maintaining queues
- Synchronizer framework has a concrete implementation of each of these three components, while still permitting a wide range of options in how they are used.
- This intentionally limits the range of applicability, but provides efficient enough support that there is practically never a reason not to use the framework (and instead build synchronizers from scratch)

AbstractQueueSynchronizer code

- Implementation of Mutex class using the framework

- Mutex class, that uses synchronization state zero to mean unlocked, and one to mean locked

- ```
  class Mutex {
          class Sync extends AbstractQueuedSynchronizer {
                  public boolean tryAcquire(int ignore) {
                  return compareAndSetState(0, 1);
                  }
                  public boolean tryRelease(int ignore) {
                  setState(0); return true;
                  }
          }
          private final Sync sync = new Sync();
          public void lock() { sync.acquire(0); }
          public void unlock() { sync.release(0); }
  };
  ```
  ```

BlockingQueueInterface

- ArrayBlockingQueue
  - A bounded blocking queue backed by an array
  - This queue orders elements FIFO (first-in-first-out)
  - **Head:** element that has been on the queue the longest time
  - **Tail**: element that has been on the queue the shortest time
  - New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue
  - It is classic Bounded Buffer: Fixed-sized array holds elements inserted by producers and extracted by consumers
  - Attempts to put an element to a full queue will result in the put operation blocking; attempts to retrieve an element from an empty queue will similarly block

BlockingQueueInterface

- ArrayBlockingQueue code
  - private ArrayBlockingQueue messageQ = new ArrayBlockingQueue <String> (10);

```
Logger logger = new Logger(messageQ);
public void run () {
    String someMsg;
    try {
            while (true){
                    // do something
                    // blocks if no space available
                    messageQ.put(someMsg);
            }
    } catch (InterruptedException IE) { … }
}
```

BlockingQueueInterface

- LinkedBlockingQueues
  - Not hard bounded as ArrayBlockingQueues
  - Same features as ArrayBlockingQueues, but based on linked nodes
  - Linked queues typically have higher throughput than array-based queues but less predictable performance in most concurrent applications
  - The optional capacity bound constructor argument serves as a way to prevent excessive queue expansion
  - Linked nodes are dynamically created upon each insertion unless this would bring the queue above capacity

BlockingQueueInterface

- PriorityBlockingQueue
  - An unbounded blocking queue that uses the same ordering rules as class PriorityQueue and supplies blocking retrieval operations
  - While this queue is logically unbounded, attempted additions may fail due to resource exhaustion (causing OutOfMemoryError)
  - A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so results in ClassCastException)
  - This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces
  - The Iterator provided in method iterator() is not guaranteed to traverse the elements of the PriorityBlockingQueue in any particular order
  - If you need ordered traversal, consider using Arrays.sort(pq.toArray())

References

- http://gee.cs.oswego.edu/dl/papers/aqs.pdf
- http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/package-summary.html
- http://www.wiziq.com/tutorial/183-Concurrency-in-Java
- http://www.slideshare.net/alexmiller/java-concurrency-gotchas
- http://book.javanb.com/java-threads-3rd/jthreads3-CHP-14-SECT-3.html

Thank You!

Executive Summary

- The presentation talks about the following things:
  - General concurrency: what it means, how is it important
  - Concurrency in C: with some code thrown in, we explore how it is to write concurrent programs in C, and how it is complicated.
  - Concurrency in Java: Based off previous, point describe the need for abstracting all the concurrency related constructs from application developers, thus a need for Java framework.
  - Threads: How to create and start threads in efficient and scalable manner
  - Synchronization:

    Explore the need for Thread synchronization and different methods to do so like synchronized methods, synchronized statements, atomic variables etc.
  - Explore abstractions for thread creation and starting like Executor, ExecutorService and ScheduledExecutorService
  - Introduction to Thread Pools and usage
  - Usage of ExecutorService and Thread Pools together explained using code examples
  - Explanation of Future interface with code
  - More on Synchronizers like Semaphores and CyclicBarriers with code examples
  - Explanation of AbstractQueueSynchronizer class and the abstract framework it provides based on which various Synchronizers mentioned before are implemented, including code example
  - Introduction to BlockingQueueInterfaces