Java Concurrency Framework

CSCI 5448: Graduate Presentation

Smitha Sunil Kamat and Krithika Parthan

Executive summary

- The Java Concurrency framework provides a platform to parallelize applications effectively. This allows the programmer to make full use of multiple cores and hence improve the performance of the application.
- The concurrency model allows performance improvement with a single core machine also since it tasks can be switched in and out of the processor giving the illusion of parallelism while hiding latencies effectively.
- In order to make the application concurrent, a set of “threads” are created for each task. These threads share the resources of the task and also have their own context which is smaller than the parent task. This can be done easily and effectively using classes and interfaces provided by the Java Concurrency framework.
- To write thread safe programs that allow multiple threads to work on shared resources without data corruption, the concurrency library provides a set of synchronization mechanisms that ensure security of data without costing performance.
- The object-oriented abstractions provided by the Java platform coupled with the parallelism offered by the concurrency framework provides a very powerful set of tools for programmers to develop applications that are modular and fast.

Contents

Introduction

- The JAVA concurrency framework provides a set of classes and services which helps JAVA programmers to design high performance, reliable and maintainable applications (design) with reduced programming effort.
- This presentation will introduce the various concepts that are integral parts of the JAVA concurrency framework along with examples of their usage in JAVA multithreaded programs.

Motivation

- In this age of multi-core processors, leveraging work on all the cores helps obtain a successful, high volume application.
- Threads are a mechanism that help tasks to run asynchronously.
- Sequential execution suffers from poor responsiveness and throughput.
- Multi-threaded programming is required to overcome the limitations of sequential execution.

History

- The JAVA concurrency package was developed by Doug Lea and it comprised Collection-relate classes.
- An updated version of these utilities was included in JDK 5.0 as of JSR 166.
- JAVA SE 6 and JAVA SE 7,both introduced updated versions of the JSR 166 APIs, inclusive of several new additions.
- This enabled the JAVA programming language and the JAVA Virtual Machine (JVM) to support concurrent programming where the program execution takes place in the context of threads.

Java Concurrency Packages

- Following are the JAVA packages that support concurrent programming
  - Java.util.concurrent
  - Java.util.concurrent.atomic
  - Java.util.concurrent.locks
  - Java.lang.Threads
- Multithreaded programs that are tedious to implement can be easily accomplished using the above packages.
- Each thread created in Java is an instance of class “Thread” provided in the Java framework as a part of java.lang.Threads.

Executor Framework

- Simple interface providing the primary abstraction for task execution, supporting launching of new tasks
- Describes tasks using “Runnable”, providing the means of decoupling “task submission” from “task execution”.
- Executor is based on the “producer-consumer” pattern, where activities submitting the tasks are producers and the threads that execute the tasks are consumers
- Following are the two implementations of the Executor framework:
  - ThreadPoolExecutor: Consists of worker threads which have minimized overhead of creation
  - ScheduledThreadPoolExecutor: Consists of worker threads that can be scheduled to run after a given delay or to execute periodically

Executor (II)

The subinterfaces of the “Executor” framework are:

- ExecutorService: Allows a task to return a value, accepts Callable objects, facilitates shutting down of the executor
- Runnable interface: Every thread created calls the run() method which is a part of the “Runnable” interface. The run() method contains the work that should be performed by the thread

Executor (III)

Example:

```
public void mainMethod() {
        HelperTask task = new HelperTask(); // Step 1: Create an object representing the task
        Thread t = new HelperThread(task);  // Step 2: Create a new thread for executing the task
        t.start();                          //  Step 3: Start the new thread
}

public class HelperTask implements Runnable {
        public void run() {
                doHelperTask();
        }
}
```

Executor (IV)

- Callable interface: It represents the asynchronous task to be executed. Its call() method returns a value (unlike Runnable's run() that doesn't return a value), i.e the task will be able to return a value once it is done executing. Callable can also throw an exception if it cannot calculate a result.
- The example below shows an example of Callable class returning a random integer

```
4  public class Calculation implements Callable<Integer> {
5      public Integer call() {
6          return new Random().nextInt(1000);
7      }
8  }
9
```

- AbstractExecutorService: Provides default implementations of the ExecutorService execution method
- ScheduledExecutorService: Supports both the Callable and Runnable objects with delays

Executor (V)

Example:

```
4  public class ExecutorExample implements Executor {
5      private static ArrayBlockingQueue<Thread> threads = new ArrayBlockingQueue<Thread>(2);
6      private static int id = 0;
7
8      public void execute(Runnable r, String name, int priority) {
9          ThreadFactoryExample tFactory = new ThreadFactoryExample();
10         Thread t = tFactory.newThread(r,name,priority);
11         if (threads.offer(t)) {
12             System.out.println("Thread "+t.getName()+" added to execution queue");
13         }
14         try {
15             t = threads.take();
16         } catch (InterruptedException e) {
17             e.printStackTrace();
18         }
19         System.out.println("Thread "+t.getName()+" started with priority "+t.getPriority());
20         t.start();
21     }
22
23     public void execute(Runnable c) {
24         this.execute(c,"Task "+id++,Thread.NORM_PRIORITY);
25     }
26
27 }
```

Executor (VI)

- The code snippet in the previous slide shows that the ExecutorExample class implements the Executor interface.
- The execute() method is overloaded to allow for a customized version of Executor. This allows the name and priority of a thread to be controlled or set to a default value if it is not used in the execute() call.
- The customized version contains a ThreadFactory to control the creation of new threads used in the executor.
- It also contains an ArrayBlockingQueue used to store the threads so that if multiple runnable tasks are submitted to the ExecutorExample, they will by safely handled and run in the order submitted to the queue.

Thread Factory

- The Executor interface provides the ThreadFactory utility method.
- The default ThreadFactory method creates a ThreadFactory class instance that can be used to spawn new threads.
- The Threadfactory allows creation of threads without any client (producer) intervention.
- The next code snippet shows a ThreadFactory example.
- Here a ThreadFactory instance is created using the Executor interface's default ThreadFactory.

Thread Factory (I)

- The Runnable instance's new thread to be spawned is passed to the ThreadFactory instance's newThread method.
- A thread can be created in two ways:
  - Providing a Runnable object.
  - Creating a subclass of Thread class.
- Advantages of using the Executor interface and the ThreadFactory class:
  - Automatic assignment of the thread pool name, thread group name and the thread name to the newly created thread.
  - Threads can take advantage of the debugging features.

Thread Factory (II)

Example:

```
package com.concurrency.tf.test;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadFactory;
import com.concurrency.Activity;
public class ThreadFactoryTest {
    public static void main(String[] args) {
        ThreadFactory tf = Executors.defaultThreadFactory();
        Thread t = tf.newThread(new Activity());
        t.start();
    }
}
```

Futures

- This interface represents the result of an asynchronous task
- The ExecutorService which can execute a Callable task returns a Future object to return the result of the Callable task
- The result can be obtained using get() that remains blocked until the result is computed
- The completion of the task can be checked via isDone()
- Computations can be cancelled via cancel(), if the result has already been calculated

Futures (I)

Example:

```
public class CallableFutures {
  private static final int NTHREDS = 10;

  public static void main(String[] args) {

    ExecutorService executor =
Executors.newFixedThreadPool(NTHREDS);
    List<Future<Long>> list = new ArrayList<Future<Long>>();
    for (int i = 0; i < 20000; i++) {
      Callable<Long> worker = new MyCallable();
      Future<Long> submit = executor.submit(worker);
      list.add(submit);
    }
```

Futures (II)

Example continued:

```
long sum = 0;
    System.out.println(list.size());
    // Now retrieve the result
    for (Future<Long> future : list) {
      try {
        sum += future.get();
      } catch (InterruptedException e) {
        e.printStackTrace();
      } catch (ExecutionException e) {
        e.printStackTrace();
      }
    }
    System.out.println(sum);
    executor.shutdown();
  }
```

Futures (III)

- 10 Callable threads are created
- A linked list of future objects are created to store the results computed by the callable threads
- The result is retrieved from the Future objects using the get() method

Queues

- Queues in the JAVA concurrency framework are data structures used to hold tasks before they are executed.
- Offer(), poll(), remove() are standard queue methods
- The AbstractQueue provides the basic queue features
- Blocking queues: Can be used to implement producer/consumer pattern dependent problems
- Non-blocking queues: Queues that don’t require synchronization, are based on low level hardware atomic operations such as compare and swap (CAS)

Queues (I)

Types of blocking queues:

- PriorityBlockingQueue: Priority queue based on priority heap, supplying blocking retrieval operations. This class implements the Collection and the Iterator interfaces. This class doesn't permit NULL elements and is thread safe.
- LinkedBlockingQueue: Queue based on linked nodes, ordering of elements is FIFO, dynamic in nature.
- DelayQueue: Element(task) at the head of the queue will be executed if its delay has expired
- ArrayBlockingQueue: Queue based on a fixed size array of elements, ordering of elements is FIFO, static in nature. Can be used for Producer/Consumer problems.

Queues (II)

Types of non-blocking queues

- ConcurrentLinkedQueue: Thread safe queue based on linked nodes. Ordering of elements is FIFO.
- PrirorityQueue: Unbounded queue based on priority heap, doesn’t accept null elements in the queue.

Synchronous Queues

- The synchronous queue is a blocking queue where the insert operation by one thread (producer) must wait for the corresponding remove operation by another thread (consumer)
- Until there exists a consumer and a corresponding producer, the capacity of the queue is zero

Synchronous Queues (I)

Example:

```
3  public class Consumer implements Runnable{
4     SynchronousQueue<Integer> sq;
5     public Consumer(SynchronousQueue<Integer> sq){
6        super();
7        this.sq = sq;
8     }
9     public void run(){
10       try{
11          System.out.println(sq.take().toString() + "removed from the Synchronous Queue\n");
12       }
13       catch(InterruptedException e){
14          e.printStackTrace();
15       }
16    }
17 }
```

```
4  public class Main {
5     public static void main(String args[]) {
6        SynchronousQueue<Integer> sq = new SynchronousQueue<Integer> ();
7        Consumer c = new Consumer(sq);
8        Producer p = new Producer(sq);
9        for(int i = 0; i<10; i++){
10          new Thread(c).start();
11          new Thread(p).start();
12       }
13    }
14 }
```

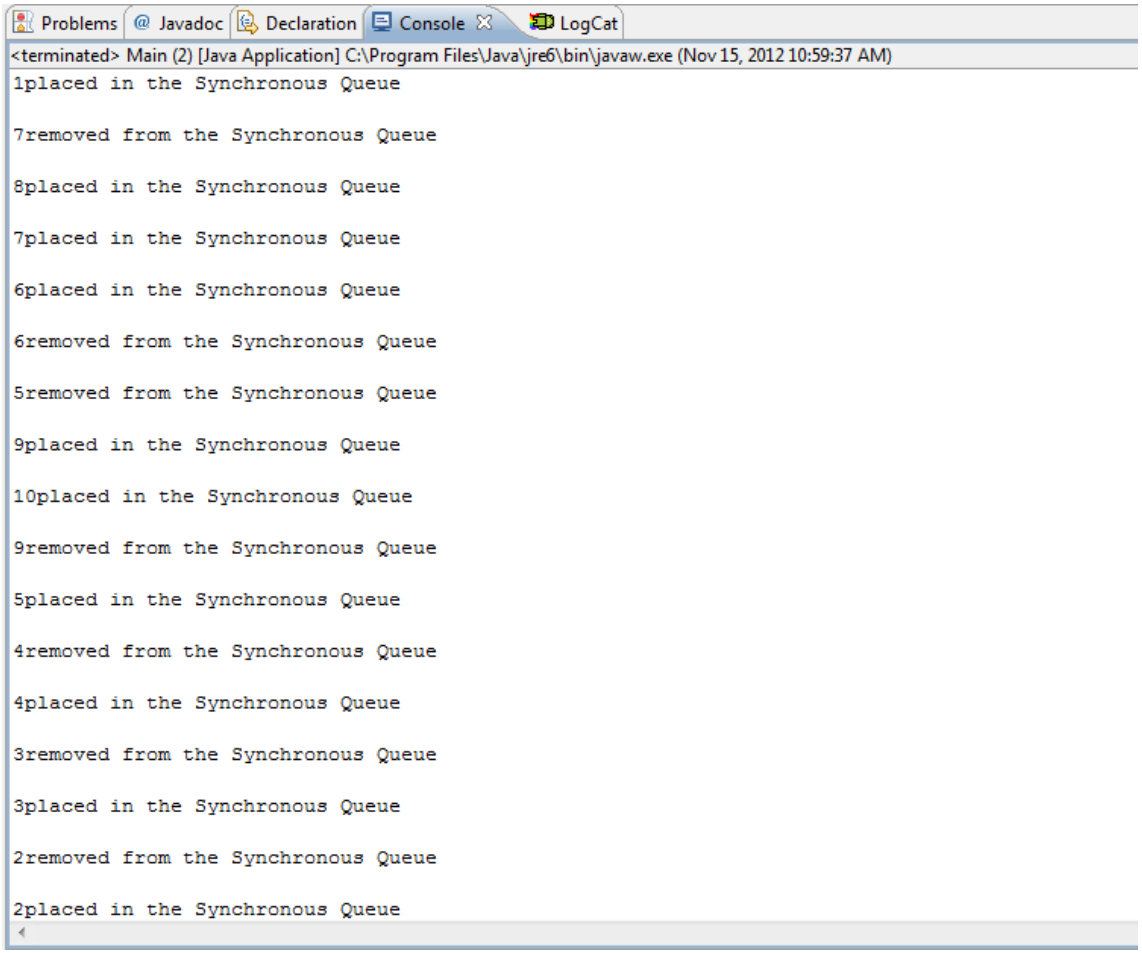
Synchronous Queues (II)

Example continued:

```
4  public class Producer implements Runnable{
5     SynchronousQueue<Integer> sq;
6     private static AtomicInteger  value = new AtomicInteger(1);
7
8     public Producer(SynchronousQueue<Integer> sq) {
9        super();
10       this.sq = sq;
11    }
12
13    public void run(){
14       try{
15          Integer i = value.getAndIncrement();
16          sq.put(i);
17          System.out.println(i.toString() + "placed in the Synchronous Queue\n");
18       }
19       catch(InterruptedException e){
20          e.printStackTrace();
21       }
22    }
23 }
24
```

Synchronous Queues (III)

Output:



Synchronous Queues (IV)

- In the given example, the consumer producer is illustrated using synchronous queues.
- The fundamental concept is that the synchronous queue has no in-built capacity to store elements. Hence, a consumer and producer construct is required to enable extraction and insertion of elements into the queue respectively.
- From the output of the example code, we can see the threads execute in a random order with no deterministic behavior.

Synchronizers

- With multiple threads, it is important to write thread safe code.
- This means that multiple threads should be able to work on a set of data without any corruption.
- Synchronization mechanisms are used to implement thread safe practices.
- The Java concurrency framework offers a number of synchronization methods such as semaphores, mutexes and barriers.

Synchronizers (I)

- Semaphores are binary flags that can be used to indicate if a resource is available for modification or not.
  - A semaphore object can be created using the Semaphore class. The available or busy state can be set using acquire() which blocks until a permit is available. The release() method releases the acquired permit.
  - A semaphore that is defined as binary can take the values 0 (on acquire()) and 1 (on release()). In general, semaphores in Java are said to be counting semaphores.
- Mutex is implemented as a lock in JAVA. They can be exclusive locks or locks that allow concurrent access to a shared resource (ReadWriteLocks).

Synchronizers (II)

- Barriers aids synchronization by allowing a set of threads to wait for each other to reach a common barrier point, barriers are cyclic since they can be reused after the waiting threads are released.
- Java provides a cyclic barrier for this purpose. This allows the same barrier to be reused once all the threads are released.
- This can be created using CyclicBarrier class. The method await() is used to act as the actual barrier point. All threads associated with the Barrier need to invoke this for the program to move forward.

Synchronizers (III)

Example:

```
Runnable barrier1Action = new Runnable() {
    public void run() {
        System.out.println("BarrierAction 1 executed ");
    }
}          //Defines action to be performed after crossing barrier
//Creates a CyclicBarrier object
CyclicBarrier barrier1 = new CyclicBarrier(2, barrier1Action);
CyclicBarrierRunnable barrierRunnable1 =
        new CyclicBarrierRunnable(barrier1);
//Passes barrier to threads
new Thread(barrierRunnable1).start();
new Thread(barrierRunnable1).start();
```

Synchronizers (IV)

- As an alternative to CyclicBarrier, a CountDownLatch can also be used.
- The main difference is that CyclicBarrier can be passed only when specified number of threads call the await() method whereas CountDownLatch can be released based on the absolute number of calls to the function, independent of the threads calling it.
- CountDownLatch will also need to be reset before being used again.
- CyclicBarrier also takes an additional parameter “runnable()” that specifies the action to be carried out once the barrier is crossed.

Atomics

- Atomic statements or code blocks are used when certain operations are required to execute uninterrupted. For example, writes are atomic for most variables.
- This is a simple synchronization mechanism: specifying atomic code ensures that the thread executing the statement cannot be interrupted.
- The programmer can use this to ensure memory consistency.
- The java.util.concurrent.atomic package provides a number of classes that can be used to make single variable accesses atomic.
- For example, consider an atomic integer called value initialized to 1. This can be accessed atomically.
  - private static AtomicInteger *value = new AtomicInteger(1);*

Locks

- At the most basic level, read and write operations need to be guarded for thread safety.
- For this purpose, Java provided a mechanisms under java.concurrent.locks package.
- The package provides a simple Lock class and a ReentrantLock class as well. ReentrantLock takes an additional optional fairness parameter. Fairness indicates that when a lock is released, all threads waiting for the resource get a fair chance of acquiring it.
- A Condition can also be associated with a lock to define the action that triggers its release.

Applications

The Java Concurrency library has several applications:

- One of the classic applications of these Concurrent libraries is with Web services which match the consumer-producer model.
- The queue implementation can be integrated into other applications for the purpose of message-passing or inter-process communication.
- The combination of multiple abstractions offered by the Java Concurrency framework coupled with the security inherent in Java applications makes this well suited for real time systems. For example, VOIP can be written as a Java application.

Concurrent Hash Maps

- One other application is to improve the performance of standard data structures that can be parallelized.
- Java provides a Hashtable implementation that maps keys to values. however the sequential version of this allows only single access to the entire object at any point in time.
- ConcurrentHashMaps can improve this performance significantly. This works on the principle that consistency can be maintained by blocking access to the single bucket in use, known as lock striping.
- ConcurrentHashMaps provide 16 such locks for concurrent access by threads in a program. This means that a maximum of 16 threads can access the HashMap object simultaneously.
- This concurrent accesses help performance significantly since reads and lock-free and can proceed simultaneously.

References

- http://www.vogella.com/articles/JavaConcurrency/article.html
- http://multiverse.codehaus.org/overview.html
- http://stackoverflow.com/questions/3900941/open-source-software-transactional-memory
- http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Exchanger.html
- http://www.cs.rice.edu/~cork/book/node97.html
- http://www.javapractices.com/topic/TopicAction.do?Id=118
- http://www.e-zest.net/blog/writing-thread-safe-programs-using-concurrenthashmap/
- http://www.baptiste-wicht.com/2010/08/java-concurrency-part-4-semaphores/
- http://tutorials.jenkov.com/java-util-concurrent/cyclicbarrier.html