Java Synchronization Mechanisms

Java provides a number of mechanisms for synchronizing threads. These mechanisms are found in the `java.util.concurrent` package.

CountDownLatch

The `CountDownLatch` is a simple means of having a thread wait until a number of tasks have been completed by other threads. We begin by creating the latch and placing it in a location where it will be visible to the other threads we are working with. The latch starts with a counter, representing the number of actions which we want to wait for. Our main thread starts the other threads and then calls `await`. This halts the main thread until all the other threads have completed their tasks.

Here's an example from our Synchronized Collections lecture of a `CountDownLatch` in actual use.

```
static CountDownLatch latch;

public static void testThreadingSupport(int numThreads) {
   latch = new CountDownLatch(numThreads);

   for(int i=0; i< numThreads; i++) {
        new TestListThread().start();
   }

   try {
      latch.await();
   } catch (InterruptedException e) {
        …
   }
}
```

In each of the test threads, I simply call `countDown` on the latch when they have completed their task. Here's the code:

```
public void run() {
   … // do stuff
   latch.countDown();
}
```

Each call to `countDown` reduces the value of the latch. When the value drops to zero, any threads waiting on the latch are allowed to continue.

A few additional points on the latch:

- countDown is not blocking. A thread can signal using countDown and then continue to carry out tasks.
- I've only shown one thread calling await. You can actually have as many threads as you want calling await. All will block until the counter reaches zero, then they will all be released.
- As shown, await calls can be interrupted. In addition, a version of await supports a timeout.

Cyclic Barrier

The CyclicBarrier can be used to halt a number of threads until a given number have reached a particular point in their code. It is somewhat similar to the CountDownLatch, except the CountDownLatch involves two sets of threads, one set signals with countdown but does not block, the other set blocks until the counter hits zero. With a CyclicBarrier there is only an await call and all threads block until the counter hits zero.

Here is an example where a program initializes a CyclicBarrier and starts up a number of threads:

```
static CyclicBarrier barrier;

public static void main(String[] args) {
  barrier = new CyclicBarrier(NUM_THREADS);

  for(int i=0; i< NUM_THREADS; i++) {
    new TestListThread().start();
  }

  System.out.println("Main Thread Done");
}
```

Here's the basic structure for the run method on the actual threads:

```
// WARNING not quite right yet
public void run() {
  // get work done
  System.out.println(getName() + " is working");

  barrier.await();

  // do whatever needs to get done after synching up
  System.out.println(getName() + " is done");
}
```

The threads essentially do some work, then wait to synch up with the other threads. When all the other threads have completed their work, they all continue on.

We do need to add a few things to our code here. A thread waiting may be interrupted. The thread that is interrupted receives an `InterruptedException`. In addition if one thread receives `InterruptedException`, every other thread waiting on the same barrier will receive a `BrokenBarrierException`. Here's our revised code properly taking the two possible exceptions into account:

```
public void run() {
  // get work done
  System.out.println(getName() + " is working");

  try {
    barrier.await();
  } catch (InterruptedException e) {
    // handle interrupt
  } catch (BrokenBarrierException e) {
    // handle broken barrier
  }

  // do whatever needs to get done after synching up
  System.out.println(getName() + " is done");
}
```

Semaphores

Semaphores maintain a number of permits.[1] A thread can ask the semaphore for a given number of permits, if the permits are available, the thread will continue on (and the number of available permits is reduced). If the permits are not available the thread will block. When a thread is done using the permits, it returns them to the pool.

Semaphores are a natural means of managing access to a limited pool of resources. Suppose, for example, we had three printers available for use. We would create a Semaphore with give permits like this:

---

[1] The permits are not represented as actual objects. Instead the Semaphore internally stores an integer representing the number of permits currently available.

```
static final int NUM_PRINTERS = 3;
static Semaphore printerSemaphore;

public static void main(String[] args) {
  printerSemaphore = new Semaphore(NUM_PRINTERS);

  …
}
```

Worker threads can request a printer using acquire. Once they're done with the printer they call release. Here's what their code might look like:

```
public void run() {
  // do some stuff

  // get a hold of a printer, do printing,
  // then release printer to pool
  try {
    printerSemaphore.acquire();

    // actual printing here

    printerSemaphore.release();

  } catch (InterruptedException e) {
    e.printStackTrace();
  }

  // continue doing work
}
```

In this case, since each thread is only assumed to need a single printer, they call the default versions of acquire and release which request and return one permit each. Both acquire and release have alternate versions which include the number of permits required or released as a parameter.

Here is some additional information on using semaphores:

- A number of different versions of acquire are available. These include tryAcquire which will not block if a permit is not available (it returns a boolean, true if a permit is acquired, false otherwise) and acquireUninterruptibly which as its name implies cannot be interrupted.
- Typically a semaphore does not guarantee fair behavior. A thread which has recently called acquire may receive a permit before a thread which has been waiting for a long time. This behavior is referred to as barging. The semaphore can be setup for fair

behavior, guaranteeing first-in-first-out (FIFO), using a special constructor. However, fair semaphores are less efficient.