ISCTE IUL
Instituto Universitário de Lisboa

High Level Concurrency Objects in JAVA

Programação Concorrente e Distribuída
Parallel and Distributed Programming

2012-2013

After this class you will be able to…

- Understand the Lock interface
- Know how to use Semaphore
- Understand the use of executors
- Know how to use Executor, ExecutorService and ScheduledExecutorService
- Concurrent collections
- Use atomic variables

Low-level concurrency constructs

- We have already seen:
  - Threads
  - Synchronized methods and blocks
  - wait()/notify() coordination
- And we have programmed:
  - Condition variables
  - Semaphores
  - Barriers

High-level concurrency constructs

The low-level are adequate for very basic tasks, but higher-level building blocks are needed for more advanced tasks. In version 5.0 of the JAVA platform, a number higher-level building blocks were introduced:

- Advanced lock objects
- Semaphores
- Executors and thread pool management
- Concurrent collections
- Atomic variables

The Lock interface 1/2

Works as a standard monitor lock:

```
class X {
  private final ReentrantLock lock = new ReentrantLock();
  public void m() {
    lock.lock(); // block until condition holds
    try {
      // ... method body
    } finally {
      lock.unlock()
    }
  }
}
```

The Lock interface 2/2

- `void lockInterruptibly()`
  Acquires the lock unless the current thread is interrupted.
- `Condition newCondition()`
  Returns a new Condition instance that is bound to this Lock instance.
- `boolean tryLock()`
  Acquires the lock only if it is free at the time of invocation.
- `boolean tryLock(long time, TimeUnit unit)`
  Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.

Example

Suppose a bounded buffer which supports put and take methods.

- If a **take** is attempted on an empty buffer, the thread will block until an item becomes available;
- if a **put** is attempted on a full buffer, then the thread will block until a space becomes available.

put

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();
   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
          while (count == items.length)
               notFull.await();
         items[putptr] = x;
         putptr++;
         if (putptr == items.length)
              putptr = 0;
         ++count;
         notEmpty.signal();
     } finally {
         lock.unlock();
     }
   }
```

take

```
 public Object take() throws InterruptedException {
     lock.lock();
     try {
         while (count == 0)
         notEmpty.await();
         Object x = items[takeptr];
         takeptr++;
         if (takeptr == items.length)
             takeptr = 0;
         --count;
         notFull.signal();
         return x;
     } finally {
         lock.unlock();
     }
}
```

Exercise 1

Create an application that simulates 20 telephone callers and 10 phones. Each caller should be a thread and each phone should be guarded by a ReentrantLock.

Each caller should first try to get access to a phone by going through the phones one after the other and tryLock() until the caller manages to lock a phone. Then the caller should make a call (sleep(100..1000)) and free the phone. Each caller should make a total of 5 calls.

Use `System.out.println`’s to verify your application does the right thing.

The `Semaphore` class 1/3

- A counting semaphore. Conceptually, a semaphore maintains a set of permits. Each call to **acquire()** blocks if necessary until a permit is available, and then takes it. Each call to **release()** adds a permit, potentially releasing a blocking acquirer.
- JAVA's Semaphore class allows for fair scheduling and enables a thread to discover how many threads currently are waiting for permits
- When fairness is set true, the semaphore guarantees that threads invoking any of the acquire methods are selected to obtain permits in the order in which their invocation of those methods was processed (first-in-first-out; FIFO)

The `Semaphore` class 2/3

- A semaphore initialized to one, and which is used such that it only has at most one permit available, can serve as a mutual exclusion lock.

- When used in this way, the binary semaphore has the property (unlike many Lock implementations), that the "lock" can be released by a thread other than the owner (as semaphores have no notion of ownership).

The `Semaphore` class 3/3

`Semaphore(int permits, boolean fair)`

Creates a Semaphore with the given number of permits and the given fairness setting.

`void acquire()`

Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted.

`void acquireUninterruptibly()`

Acquires a permit from this semaphore, blocking until one is available.

`int getQueueLength()`

Returns an estimate of the number of threads waiting to acquire.

A semaphore to control access to a pool of items (1/3)

```
class Pool {
   private static final MAX_AVAILABLE = 100;
   private final Semaphore available = new
                  Semaphore(MAX_AVAILABLE, true);

   public Object getItem() throws
                           InterruptedException {
     available.acquire();
     return getNextAvailableItem();
   }

   public void putItem(Object x) {
     if (markAsUnused(x))
       available.release();
   }
}
```

A semaphore to control access to a pool of items (2/3)

```
protected synchronized Object getNextAvailableItem(){
     for (int i = 0; i < MAX_AVAILABLE; ++i) {
       if (!used[i]) {
          used[i] = true;
          return items[i];
       }
     }
     return null; // not reached
}
```

A semaphore to control access to a pool of items (3/3)

```
protected synchronized boolean markAsUnused(
                                 Object item) {
     for (int i = 0; i < MAX_AVAILABLE; ++i) {
       if (item == items[i]) {
          if (used[i]) {
            used[i] = false;
            return true;
          } else
            return false;
       }
     }
     return false;
}
```

Executors

- In large-scale applications, it makes sense to separate thread management and creation from the rest of the application.
- Executors are specified to do the following:
  - Create and start threads
  - Schedule tasks – start tasks with a delay or schedule tasks to run periodically
  - Create and manage thread pools

Executor interfaces

- **Executor** – Provides a single method, execute. If r is a Runnable object, and e is an Executor object one can replace (new Thread(r)).start() with e.execute(r)
- **ExecutorService** – more extensive, provides methods to manage termination and methods that can produce a result of a computation (through the interface Future) for tracking progress of one or more asynchronous tasks.
- **ShedulledExecutorService** – for scheduling commands to run after a given delay, or to execute periodically.

Thread pools

- Most of the executor implementations in java.util.concurrent use *thread pools*, which consist of *worker threads*.
- Worker threads exist separately from the Runnable tasks it executes and is often used to execute multiple tasks.
- Worker threads minimizes the overhead due to thread creation.
- One common type is the *fixed thread pool, which* 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, which holds extra tasks whenever there are more active tasks than threads.
  - An important advantage of the fixed thread pool is that applications using it *degrade gracefully*

Factory and utility methods: the class Executors

- A simple way to create an executor that uses a fixed thread pool is to invoke the newFixedThreadPool factory method in java.util.concurrent.Executors
- Other more sophisticated needs can use from java.util.concurrent:
  - ThreadPoolExecutor
  - ScheduledThreadPoolExecutor

Executor/thread pool – server example 1/2

```
class NetworkService {
    private final ServerSocket serverSocket;
    private final ExecutorService pool;

    public NetworkService(int port, int poolSize) throws…{
      serverSocket = new ServerSocket(port);
      pool = Executors.newFixedThreadPool(poolSize);
    }

    public void serve() {
      try {
        for (;;) {
          pool.execute(new Handler(serverSocket.accept()));
        }
      } catch (IOException ex) {
        pool.shutdown();
      }
    }
  }
```

Executor/thread pool – server example 2/2

```
class Handler implements Runnable {
    private final Socket socket;
    Handler(Socket socket) {
       this.socket = socket;
    }

    public void run() {
      // read and service request
    }
 }
```

Concurrent collections

Concurrent collections help avoid memory consistency errors by defining a *happens-before* relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

Example: `BlockingQueue`

BlockingQueue producer/ consumer example

```
static void main(String[] args) {
  BlockingQueue q = new SomeQueueImplementation();
  Producer p = new Producer(q);
  Consumer c1 = new Consumer(q);
  Consumer c2 = new Consumer(q);
  new Thread(p).start();
  new Thread(c1).start();
  new Thread(c2).start();
}
```

`BlockingQueue` is an interface implememted by:

`ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue`

BlockingQueue - Producer

```
class Producer implements Runnable {
   private final BlockingQueue queue;
   Producer(BlockingQueue q) { queue = q; }
   public void run() {
     try {
       while (true) {
         queue.put(produce());
       }
     } catch (InterruptedException ex) {.handle.}
   }
   Object produce() { ... }
 }
```

BlockingQueue - Consumer

```
class Consumer implements Runnable {
   private final BlockingQueue queue;
   Consumer(BlockingQueue q) { queue = q; }
   public void run() {
     try {
       while (true) {
         consume(queue.take());
       }
     } catch (InterruptedException ex) {.handle.}
   }
   void consume(Object x) { ... }
 }
```

Atomic variables

The `java.util.concurrent.atomic` package defines classes that support atomic operations on single variables.

Examples:

- `AtomicInteger`
- `AtomicBoolean`
- `AtomicIntegerArray`

Counter using `synchronized`:

```
class SynchronizedCounter {
    private int c = 0;


    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Counter using `AtomicInteger`:

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }
}
```

Advantages of Atomic variables

- The atomic variables classes support lock-free thread-safe programming on single variables.
- The specifications of these classes enable implementations to employ efficient *machine-level* atomic instructions that are available on contemporary processors.
- They are safe, because the programmer need not to worry about locking, and they are fast because locking is not needed on most platforms.

Bibliography

The Java tutorials:

http://download.oracle.com/javase/tutorial/essential/concurrency/highlevel.html

Exercise 2

Create an application that displays a window with a start button and an exit button. Initially, 100 balls are displayed on the left-hand side of the window. When the start button is pressed, a fixed thread pool of size 25 is created. One runnable object for each ball should be submitted to the thread pool. Each runnable object should move its ball from the left-hand side to the right-hand side of the window. When all the balls have reached the right-hand side of the window, the application should terminate.

[*See video on the e-learning system*]

Summary

- High-level concurreny objects in JAVA:
  - Review of low-level constructs
  - The Lock interface
  - Semaphores
  - Executors and thread pools
  - Concurrent collections
  - Atomic variables