CMSC 433 Fall 2013
Michael Hicks
(Some slides due to Rance Cleaveland)

UNIVERSITY OF MARYLAND

Lecture 9
Synchronizers

Producer-Consumer

- An common pattern in concurrent programs
  - A **producer** is a thread that creates something
    - A job to complete, the result from a completed job, etc.
  - A **consumer** is a thread that does something with what is produced
    - Executes the job, uses the result of an execute, etc.
- To implement this pattern, we need a way for producers to communicate with consumers
  - Key idea: decouple the identification of the work from the doing of the work

Programming Producer-Consumer Applications

- General strategy
  - Create classes for producers, consumers
  - Ensure constructors take a `BlockingQueue` argument (this is the work queue)
  - In main method class:
    - Create work queue
    - Create producers / consumers using this queue
    - Start threads
- This establishes that producers, consumers access same queue

Example

- ProducerThread.java

```
public class ProducerThread extends Thread {
  private final BlockingQueue<Integer> queue;  // Work queue
  …
  public ProducerThread (BlockingQueue<Integer> queue) { this.queue = queue; }
  …
}
```

- ConsumerThread.java

```
public class ConsumerThread extends Thread{
  private final BlockingQueue<Integer> queue;  // Work queue
  …
  public ConsumerThread (BlockingQueue<Integer> queue) { this.queue = queue; }
  …
}
```

- ProducerConsumerRandomizeTester.java

```
public static void main(String[] args) {
  BlockingQueue<Integer> workQueue = new ArrayBlockingQueue<Integer>(10);
  …
  for (int i=0; i < numConsumers; i++) {
    new ConsumerThread(workQueue).start();
  }

  for (int i=0; i < numProducers; i++) {
    new ProducerThread(workQueue).start();
  }
}
```

Kinds of BlockingQueue

- All extend the **Queue** interface. Four kinds:
- **LinkedBlockingQueue**
  – Ordered FIFO, may be bounded, two-lock algorithm
- **PriorityBlockingQueue**
  – Unordered but retrieves least element, unbounded, lock-based
- **ArrayBlockingQueue**
  – Ordered FIFO, bounded, lock-based
- **SynchronousQueue**
  – Rendezvous channel, lock-free in Java 6

Blocking Queues and `InterruptedException`

- Consider following in ProducerThread.java

```
private void enqueue (int i) {
  try {
    queue.put(i);
  }
  catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    throw new RuntimeException ("Interrupted Producer");
  }
}
```

  - This method is used for putting elements into the blocking queue
  - It calls `queue.put()`, which can *block*
    - If the queue is full, then thread executing `queue.put()` is suspended
    - When the queue has an empty slot, the thread may be reawakened
  - This means that `enqueue()` is also a blocking method!
- Blocking methods can throw `InterruptedException` when they are interrupted
  - Threads can interrupt each other, i.e. request each other to stop!
    - If thread T1 executed `T2.interrupt()`, it is requesting that T2 cease executing
    - T2 is not required to oblige
    - If T2 is executing normally a status flag is set
  - If a thread is blocking (i.e. its thread-state is BLOCKED, WAITING, TIMED_WAITING) then this exception is generated for T2
    The status flag is not set in this case
  - T2 then has the opportunity to decide what to do re: interruption (usually: clean-up and halt)

What To Do about `InterruptedException`?

- Propagate it
- Catch it and raise another exception
- Catch it and do some other actions
  - In real applications it is a good idea to set the interrupt status to reflect fact that thread has been interrupted
  - This can be done by invoking the static method `Thread.currentThread().interrupt();`
    - This sets the interrupt status of the current thread
    - Other threads can now see that this thread has indeed been interrupted

Synchronizers

- Blocking queues play two roles in Producer / Consumer applications
  - They store data that has been produced but not yet consumed
  - If they are bounded, they also “slow down” producers by forcing them to block when the buffer is full
- *Synchronizers*
  - *Objects that coordinates the control flow of threads based on the synchronizer’s state*
  - Blocking queues act as synchronizers
    - They cause producers to block when the queue is full
    - They cause consumers to block when the queue is empty
  - There are other types of synchronizers also

Locks

- Locks are synchronizers
  - When their state indicates they are free, they may be acquired
  - When their state indicates they are currently held, then any thread trying to acquire them must block
- So far we have seen only intrinsic (aka “monitor”) locks
  - Every object has such a lock
  - They are manipulated using synchronized blocks, synchronized methods, etc.
- Beginning in Java 1.5, explicit locks were also introduced
  - Richer interface (e.g., you can interrupt a lock holder)

The Java `Lock` Inteface

- Package: `java.util.concurrent.locks`
- Methods (from docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html)
  - **`void lock()`**
    Acquires the lock
  - **`void lockInterruptibly()`**
    Acquires the lock unless the current thread is interrupted
  - **`boolean tryLock()`**
    Acquires the lock only if it is free at the time of invocation, returning true if lock acquired and false otherwise (does not block)
  - **`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
  - **`void unlock()`**
    Releases the lock
  - **`Condition newCondition()`**
    Returns a new `Condition` instance that is bound to this `Lock` instance

Lock Classes in Java 7

- The following classes implement the `Lock` interface
  - `ReentrantLock`
  - `ReentrantReadWriteLock.ReadLock`
  - `ReentrantReadWriteLock.WriteLock`
- `ReentrantLock` objects are like intrinsic locks
  - Same effects on visibility, are reentrant, etc.
  - However
    - You have to issue `lock()` / `unlock()` operations explicitly
      - May require use of **finally** to cover all circumstances
    - There are lots more operations besides the basic ones!
      - **If you don't need these operations, stick with intrinsic locks**
- We will discuss read/write locks later

Latches

- Synchronizer objects that:
  - Block threads until a terminal condition is met
  - Subsequently release the blocked threads
  - Threads participate in synchronization by executing operations to wait on / modify latch state
- `CountdownLatch`
  - Latch based on counting
    - Terminal condition is that latch has value 0
    - Constructor accepts number to use as initial value
  - Methods
    - `await()`
      Block until latch has value 0
    - `countDown()`
      Decrement latch value by 1

Uses for Latches

- To delay starting of threads until an initial condition is satisfied
  - For example: timing a collection of threads
    - Don't want threads to start until all are created
    - In each thread, use a latch to wait for a "starting signal"
  - In this case, programming would consist of
    - Creation of latch with value 1
    - Creation, starting of threads
    - Decrement of latch using countDown(), which releases threads
- To do a "multi-way join" on thread termination
  - Idea: Initialize latch to number of threads
  - When each thread terminates, have it decrement latch
  - When latch is 0, all threads have terminated

Example: `LatchExample.java`

```
public class LatchExample {

  public static void main(String[] args) {
    final int numThreads = 25;
    final int numIterations = 1000000;
     final CountDownLatch startGate = new CountDownLatch(1);
     final CountDownLatch endGate = new CountDownLatch(numThreads);

    for (int i = 0; i < numThreads; i++) {
      Thread t = new Thread() {
        public void run () {
          try { startGate.await(); }
          catch (InterruptedException e) {}
          for (int j = 0; j < numIterations; j++) {}
          System.out.println ("Thread " + getName() + " finishes.");
          endGate.countDown();
        }
      };
      t.start();
    }

    long start = System.nanoTime();
    startGate.countDown();
    try { endGate.await(); }
    catch (InterruptedException e) {}
    long end = System.nanoTime();
    System.out.println ("The whole race took " + (end-start) + " ns.");
  }
}
```

FutureTask<T>

- A synchronization construct for starting computations now, getting the results later
  - A FutureTask<T> object is like a method call
    - It is invoked
    - It returns a value of type T
  - Unlike a method call, the invocation and return are separate events
    - A thread can start a FutureTask …
    - … do other work …
    - … then reconnect with the FutureTask when it needs the results
- The FutureTask<T> constructor requires an object matching the Callable<T> interface
  - Callable<T> like Runnable
  - Main method to implement is public T call() (as opposed to void run ())
- A FutureTask must be embedded in a thread in order to be invoked
  - Thread class includes constructor taking a FutureTask object, which also implements Runnable
  - Starting this thread amounts to "invoking" the FutureTask
- To get result of FutureTask object future, execute future.get()
  - Thread executing this will block until call is complete
  - future.get() can throw several exceptions

Example: FutureTaskTest.java

```
public class FutureTaskTest {

  public static void main(String[] args) {
    Callable<String> c = new Callable<String> () {
      public String call() {
        return ("Foo");
      }
    };

    FutureTask<String> future = new FutureTask<String>(c);

    new Thread(future).start();      // "Invokes" future
    /* Can do something else here */
    try { System.out.println(future.get()); }
    catch (InterruptedException e) {}
    catch (ExecutionException e) {}
    finally {
      System.out.println("Done");
    }
  }
}
```

FutureTasks and Exceptions

- FutureTask invocations, like method calls, can generate checked, unchecked exceptions
  - Executions thrown when `get()` is called
  - If call generates an exception, it is "wrapped" inside a special `ExecutionException` object
  - To recover original exception, you must analyze this object
- Because `get()` blocks, it can also throw `InterruptedException`

Counting Semaphores

- Counting semaphores act like bounded counters
  - Initially, a positive value is given to semaphore
  - Operations can atomically decrement (`acquire()`) or increment (`release()`) this value
  - If the semaphore value is 0, then `acquire()` *blocks*
- Why "`acquire()` / `release()`"?
  - Intuition: semaphores dispense "permits"
    Count reflect number of permits available
  - Acquisition of a permit reduces available permits by 1
  - Release increments number of permits by 1
    - Note: you can release even if you have not acquired!
    - So release really means: generate a new permit and add it into pool
  - The permit idea is only for intuition! There are no explicit permit objects
- What are semaphores used for?
  - Resource allocation
    - You have n copies of a resource
    - You can use a semaphore to ensure that when more than n threads need the resource, some of them block
  - Size restrictions for data structures
    - Semaphore records maximum size
    - When you add an element, you need to acquire a permit first
    - When an element is deleted, you release a permit

Example: BoundedHashSet.java

```
public class BoundedHashSet<T> {

  private final Set<T> set;
  private final Semaphore spaceAvailable;

  public BoundedHashSet (int capacity) {
    this.set = Collections.synchronizedSet(new HashSet<T>());
    spaceAvailable = new Semaphore(capacity);
  }

  public boolean add(T o) throws InterruptedException {
    spaceAvailable.acquire();
    boolean wasAdded = false;
    try {
      wasAdded = set.add(o);
      return(wasAdded);
    }
    finally { if (!wasAdded) spaceAvailable.release(); }
  }

  public boolean remove(T o) {
    boolean wasRemoved = set.remove(o);
    if (wasRemoved) spaceAvailable.release();
    return wasRemoved;
  }
}
```

Barriers

- A synchronizer for blocking a collection of threads until they all are at "the barrier point"
  - Threads wait at the barrier by invoking `barrier.await()`
  - When the number of threads indicated in the barrier object have arrived, all are released
  - Barriers can optionally have a `Runnable` object that is executed right before threads are released
- Uses: simulations
  - Simulations are often "step-by-step"
  - Computation at each step can be done in parallel using threads
  - Don't want to start next step until current step is complete
- Key class: `CyclicBarrier`
  - Cyclic: same barrier object can be reused after it releases threads
  - Methods
    - `public int await()`
      Blocks until the number of threads needed are blocking; then releases. Returns arrival index of party: 1 is first, 0 is last
    - `void reset()`
      Resets barrier to its initial state. Any currently waiting threads throw a BrokenBarrierException
    - `public boolean isBroken()`
      Returns true if barrier is broken (i.e. a waiting thread is interrupted or times out, or a barrier action causes an exception), false otherwise

Conditions

- Condition variables make a thread wait (e.g., at the start of a method) until a precondition holds
  - The scenario: if the precondition fails now, it will only succeed when another thread does something
    - E.g., condition is that to put something in a buffer, the buffer must not be full. If it is full, the putter thread must wait until another thread removes something
- Condition variables make waiting efficient compared to constantly testing the precondition over and over
  - Intrinsic locks have an implicit condition variable, used via calls to wait(), notify(), and notifyAll()
  - Non-intrinsic locks generalize this support
- We'll discuss usage patterns in more detail next week

Conditions for Locks

- Another difference between intrinsic, explicit locks
  - No `wait()` / `notify()` / `notifyAll()`
  - There is a "`newCondition()`" method
- Conditions are used to implement suspension / resumption
  - Any lock can have several conditions associated with it
  - A thread can wait on a condition using method `await()`
    - Very similar to `wait()`
    - Thread suspends, surrenders lock
    - When a notification occurs thread awakens and tries to reacquire lock
    - When lock is successfully reacquired `await()` terminates
  - A thread can awaken processes that are suspended on a condition using `signal()` / `signalAll()`

    Like `notify()` / `notifyAll()`

Example: ArrayBoundedBuffer.java

```
// Adapted from http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Condition.html
public class ArrayBoundedBuffer {
  private final ArrayList<Object> items;
  private final int capacity;
  private final Lock lock = new ReentrantLock();
  private final Condition notFull  = lock.newCondition(); // Waiting for not full
  private final Condition notEmpty = lock.newCondition(); // Waiting for not empty

  public ArrayBoundedBuffer (int capacity){ … }

  public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
      while (items.size() == capacity) notFull.await();
      items.add(x);
      notEmpty.signal();
    } finally { lock.unlock(); }
  }

  public Object take() throws InterruptedException {
    lock.lock();
    try {
      while (items.size() == 0) notEmpty.await();
      Object x = items.get(0);
      notFull.signal();
      return x;
    } finally { lock.unlock(); }
  }
}
```