CS11 – Java

Spring 2011-2012
Lecture 7
Today’s Topics

- All about Java Threads
- Some Lab 7 tips
Java Threading Recap

- A program can use multiple threads to do several things at once
  - A thread can have local (non-shared) resources
  - Threads can share resources, too!
- Interactions with shared resources must be performed atomically
  - Not doing this produces spurious results
  - Shared resources must be locked carefully to avoid deadlock and other similar problems
Why Multithreading?

- Sometimes threads perform "slow" operations
  - e.g. communication over a network
  - Can perform other tasks, while slow operation takes place in a separate thread

- Threads also provide a powerful conceptual model
  - Some programs are simply easier to understand, when implemented with several threads to perform various tasks

- Threads impose a (usually small) performance cost
  - Single processor has to switch between several threads, to give each one a time-slice to run in
  - Even with multiple processors, have synchronization costs
This Week’s Lab

- Make last week’s web-crawler faster!
  - Lots of time spent sending HTTP request and waiting for response

- Create multiple crawler threads
  - Each will analyze one web page at a time
  - Provides dramatic improvement in performance
    - ...as long as there aren’t too many crawler threads!

- Need a “URL Pool”
  - Crawlers get “next URL to crawl” from the pool
  - Each crawler thread puts new URLs into the pool
The URL Pool

- URL Pool is a shared resource
  - Crawler threads must interact atomically with it
  - Sometimes, no “next URL” will be available!

- How can a thread perform atomic interactions with an object?
- How can a thread passively wait for a condition to become true?
In Java, every object has a **monitor**
- A monitor is a simple mutex ("mutual exclusion") lock
- An object can be locked by *at most* one thread at a time

**Use** `synchronized` **block** to lock an object

```java
synchronized (sharedObj) {
    ... // Perform atomic operations on shared object
}
```
- Thread **blocks** (suspends) until it acquires `sharedObj`'s monitor
- Thread resumes when it acquires `sharedObj`'s monitor
- At end of `synchronized` block, thread automatically releases `sharedObj`'s monitor
Producer-consumer problem:
- One thread is producing data
- Another thread is consuming the data
- How to interface the two threads?

A simple solution: build a thread-safe FIFO
- “First In, First Out” queue
- Both producer and consumer use the FIFO
  - Producer puts data into the FIFO
  - Consumer gets data out of the FIFO
- Interaction with FIFO must be synchronized!
A Simple FIFO

- Build a FIFO that uses a LinkedList for storage
- Give our FIFO a maximum size.
  - If producer is faster than consumer, don’t want FIFO to grow out of control!
- Our FIFO class:
  ```java
  public class FIFO {
      private int maxSize;
      private LinkedList items;

      public FIFO(int size) {
          maxSize = size;
          items = new LinkedList();
      }
      ...
  ```
If there is space, add object to end of FIFO and return true.

Otherwise, do nothing and return false.

FIFO Code:

```java
public boolean put(Object obj) {
    boolean added = false;
    if (items.size() < maxSize) {
        items.addLast(obj);
        added = true;
    }
    return added;
}
```
Getting Items from the FIFO

- If an item is available, remove it and return it
- If no item is available, return `null`

FIFO Code:

```java
public Object get() {
    Object item = null;
    if (items.size() > 0)
        item = items.removeFirst();

    return item;
}
```

Removing an item from an empty list causes an exception to be thrown.
This FIFO code isn’t thread-safe!

- **LinkedList** isn’t thread-safe, so getting and putting at same time can produce spurious results.
- Bigger issues arise with multiple producers, or multiple consumers.
- Example: two consumer threads, one item in queue

```java
public Object get() {
    Object item = null;
    if (items.size() > 0)
        item = items.removeFirst();

    return item;
}
```

Both consumers might see `items.size()` return 1, then try to grab the one item. The FIFO would throw an exception!
Synchronizing FIFO Operations

- FIFO can use `synchronized` blocks to ensure thread-safety

```java
public Object get() {
    Object item = null;
    synchronized (items) {
        // This thread has exclusive
        // access to items now.
        if (items.size() > 0)
            item = items.removeFirst();
    }
    return item;
}
```

- Must also make `put(Object)` method thread-safe!
  - Enclose operations on `items` within a `synchronized` block
Another FIFO Issue

- What about when there’s nothing to get?
  - Could write a loop that checks regularly ("polls" or "spins")
    ```java
    // Keep trying until we get something!
    do {
      item = myFifo.get();
    } while (item == null);
    ```

- Polling in a tight loop is very costly!
  - Polling operations almost invariably use way too many CPU resources to be a good idea
  - *Always* try to find another solution to polling
Passive Waiting

- Would like threads to wait **passively**
  - Put a thread to sleep, then wake it up later
  - Accomplished with `wait()` and `notify()` methods
  - Defined on `java.lang.Object` (see API docs)

- Once a thread has synchronized on an object:
  - (i.e. the thread holds that object’s monitor)
  - The thread can call `wait()` on that object to suspend itself
  - The thread *releases* that object’s monitor, then suspends.

- Can only call `wait()` on an object if you have actually synchronized on it.
  - If not, `IllegalMonitorStateException` is thrown!
Another thread can wake up the suspended thread
- First, the thread must lock the same object as before
  - (It synchronizes on the object.)
- Then the thread can call `notify()` or `notifyAll()` to wake up any threads that are suspended on that object.
  - `notify()` wakes up one thread waiting on that object
  - `notifyAll()` wakes all threads waiting on that object
- If no thread is waiting when `notify()` or `notifyAll()` is called, `nothing` happens. (It’s a no-op.)

Can only call `notify()`/`notifyAll()` on objects that the thread has already locked...
Thread Notification

- When a thread is notified, it immediately tries to relock the object it called `wait()` on:
  - It called `wait()` inside a synchronized block…
  - But the thread that called `notify()` still holds the lock.

- When the notifying thread releases the lock, one of the notified threads gets the lock next:
  - The JVM arbitrarily picks one!
  - The notified thread gets to resume execution with exclusive access to the locked object.
How To Use `wait()` and `notify()`

- **Common scenario:**
  - One thread can’t proceed until some condition is true.
  - The thread can call `wait()` to go to sleep.
  - FIFO: `get()` method can `wait()` if no items

- **Another thread changes the state:**
  - It *knows* that the condition is now true!
  - It calls `notify()` or `notifyAll()` to wake up any suspended threads
  - FIFO: `put()` method can `notify()` when it adds something
How to `wait()`

- A waiting thread shouldn’t *assume* that the condition is true when it wakes up.
  - If multiple threads are waiting on the same object, and `notifyAll()` was called, another thread may have gotten to the object first.
  - Can also use `wait()` with a timeout
    - “Wait to be notified, or until this amount of time passes.”
  - Also, *spurious wakeups* can occur
    - A thread resumes without being notified (!!!)
    - Can occur depending on how JVM was implemented

- **Always** use `wait()` in a loop that checks the condition
Now we can use `wait()` in our FIFO:

```java
public Object get() {
    Object item = null;
    synchronized (items) {
        // This thread has exclusive access to items

        // Keep waiting until an item is available
        while (items.size() == 0)
            items.wait();

        item = items.removeFirst()
    }
    return item;
}
```

Always wait inside of a loop that checks the condition!
Now `put()` must notify waiting consumers

```java
public boolean put(Object obj) {
    boolean added = false;
    synchronized (items) {
        if (items.size() < maxSize) {
            items.addLast(obj);
            added = true;
            // Added something, so wake up a consumer.
            items.notify();
        }
    }
    return added;
}
```

Call `notify()` on same object that other threads are waiting on.
One More Issue…

- If producer is faster than consumer, it has no way to wait until there’s room in the FIFO!
  - The consumer can passively wait, but…
  - Producer has to *poll* if there’s no room in the FIFO

- This is a **simple** FIFO. 😊
  - In fact, it’s *really* simple – it has other issues too!
  - Example: using a single lock for both gets & puts

- See `java.util.concurrent` classes for really sophisticated queues, pools, etc.
  - New in Java 1.5! Written by Doug Lea.
Synchronizing on **this**

- An object can synchronize on *itself*
  - Particularly useful when an object manages several shared resources
  - Manually locking multiple resources can lead to deadlock, if you aren’t careful…
- FIFO could do this instead of locking **items**:
  ```java
  public Object get() {
      Object item = null;
      // Lock my own monitor.
      synchronized (this) {
          while (items.size() == 0)
              wait();  // Call wait() on myself.

          item = items.removeFirst();
      }

      return item;
  }
  ```
Synchronized Methods

- Synchronizing on this is very common...
- Java provides an alternate syntax:
  ```java
  public synchronized Object get() {
      while (items.size() == 0)
          wait();
      return items.removeFirst();
  }
  ```
  - this is locked at beginning of method body
  - this is unlocked at end of method body
  - Can call `wait()` or `notify()` inside method

- Putting **synchronized** on all methods is an easy way to make a class thread-safe
  - (Don’t need to put **synchronized** on constructors)
Threads and Performance

- Synchronization incurs a cost
  - Locking and unlocking the mutex takes time
  - Don’t use synchronization unless it’s necessary
  - Bad examples:
    - `java.util.Vector`, `java.util.Hashtable`
      - Both classes synchronize every single method!
      - Don’t use them in single-threaded programs (or at all?)

- Threads should lock shared resources for as little time as possible
  - Keep thread-contention to a minimum
Lab 7 Tips

- Need a pool of `URLDepthPair` objects
  - This pool is shared among all web-crawler threads
  - Crawler threads get URLs from pool, add new ones to pool

- Internals:
  - One `LinkedList` to keep track of URLs to crawl
  - Another `LinkedList` for URLs you have seen

- Methods:
  - Get the next `URLDepthPair` to process
    - Suspend the thread if nothing is immediately available
  - Add a `URLDepthPair` to the pool
    - Always add the URL to “seen” list
    - Only add to “pending” list if depth is less than max depth
    - If added to “pending” list, notify any suspended threads
Most Challenging Problem

- When are we done crawling? How do we know?
  - When all crawler threads are waiting, we’re done!
  - (Pending queue had better be empty, too!)
- URL Pool should keep a count of waiting threads
  - Easy to implement:
    - In constructor, initialize count of waiting threads to 0
    - Increment count before calling `wait()`
    - Decrement count after `wait()` returns
- Main thread can periodically check this count
  - It knows how many crawler threads were requested
  - It needs to print out the results at the end, anyways.
  - Make sure to synchronize access to this shared state!
Crawler Threads

- Create a **CrawlerTask** that implements **Runnable**
  - **CrawlerTask** needs a reference to the **URLPool**
    - Hint: pass **URLPool** to the **CrawlerTask** constructor
  - **run()** method contains a loop:
    - Get a URL from the pool.
    - Download the web page, looking for new URLs.
    - Stick new URLs back into the pool.
    - Go back to the beginning!
  - Process each URL in a helper method (or several helpers)
    - Hint: reuse your code from last week’s crawler.
  - Handle exceptions gracefully!
    - If a problem occurs with one URL, go on to the next one!
Web-Crawler Main Method

- **main()** drives everything from start to finish
  - Get initial URL, max depth, number of threads from command-line parameters
  - Create a URL pool, add the initial URL to pool
  - Create and start the requested number of threads
    - Could put them into an array, to clean them up later, but really not necessary for this lab
  - Check pool every 0.1 to 1 second for completion
  - When finished, print URLs in the pool’s “seen” list
  - `System.exit(0);`
Using Threads

- Create a class that implements **Runnable**
  - Implement `run()` method to do your work
- Pass an instance of your class to **Thread** constructor
  ```java
  CrawlerTask c = new CrawlerTask(pool);
  Thread t = new Thread(c);
  ```
- Call `start()` on the new thread object
  ```java
  t.start();
  ```
- The thread will automatically call your object’s `run()` method
- Thread terminates when your `run()` method ends
Gentle Polling

- Use `Thread.sleep()` to pause between checks
  - `sleep()` is a static method
  - Can throw `InterruptedException`!
  - (About the nicest way one can poll...)

- Something like this:
  ```java
  while (pool.getWaitCount() != numThreads) {
      try {
          Thread.sleep(100); // 0.1 second
      } catch (InterruptedException ie) {
          System.out.println("Caught unexpected " +
                          "InterruptedException, ignoring...");
      }
  }
  ```
The Big Picture

Lab7 class
- main() method
  - Set up pool
  - Start threads
  - Monitor pool
  - Print results
  - Shut down

Threads
- CrawlerTask
- CrawlerTask
- CrawlerTask

URLPool
- pending
- seen
- waitCount
Pool Synchronization

- **URLPool** contains several shared resources!
  - Pending list, seen list, count of waiting threads, …

- **URLPool** object can synchronize on itself.
  - Avoids thread-safety/deadlock issues, etc.

- **URLPool** should take care of threading operations internally.
  - Crawler tasks shouldn’t have to manually synchronize/wait/notify on pool to use it.
  - Want to encapsulate threading behavior, too!
Java Threading References

- **Concurrent Programming in Java** (2nd ed.)
  - Doug Lea

- **Effective Java**
  - Joshua Bloch