1. Short answer (15 points total)

(a) If you ask your web browser to display the web page at http://www.cs.umd.edu/users/pugh/index.html, what might be the first line of the HTTP request sent by the web browser to the web server?

   **Answer:** GET /users/pugh/index.html HTTP/1.1

(b) Briefly describe Cross site scripting. What can it do if you have Javascript and other active content (e.g., Flash and Java) turned off in your browser? What can it do if they are turned on?

   **Answer:** Cross site scripting allows an attacker to insert their own unexpected content onto a web page generated by another site (generally, some trusted or authoritative web site). Many XSS vulnerabilities allow the insertion of arbitrary content including any HTML tags. But even arbitrary text is prohibited, the attacker might be able to insert unexpected content with undesired consequences (e.g., a link or image URL starting with javascript:)

   With active content turned off, the damage from cross site scripting is primarily limited to web site defacement. With active content turned on, an attacker can do things like submit forms and steal cookie data (e.g., click the "buy now" button on amazon).

(c) Discuss the interaction between cookies and cross site request forgery. If a web site doesn’t use cookies, how does that change the impact of cross site request forgery?

   **Answer:** In general, a CSRF attacker only knows public information, like the URL used to request that money be transferred between accounts. Without cookies, an attacker can get your browser to send an unexpected request to a server, but without any authentication information, it is unlikely to cause anything significantly bad to happen.

   With cookies enabled, if a user is already signed in, CSRF can get a browser to combine the request generated by the attacker with a cookie that piggybacks on the authentication previously supplied by the user. In the only thing in a request that identifies a user as authenticated is a cookie, CSRF allows an attacker to forge/appropriate that authentication.

2. Thread jukebox (15 points)

   Say we had a design where any of several threads could request that a particular song should be played, but the threads shouldn’t be blocked while until the requested song is played. We can also only play one song at a time; each song is played to completion before the next song starts. Sketch a design for implementing this. You can provide code below if you find that easier than describing a design. But a detailed design is sufficient (make sure you provide all the details needed to allow the code to be written).

   Don’t worry about the exact name of any classes or methods. Tell me what the classes and methods do, and you can name them whatever you want.

   Note: there is more than one good design (and probably several bad designs). Any good design is fine.
Answer:

I thought of two designs, both of which I thought were good:

- Use a blocking concurrent queue of songs. One jukebox thread is created when the ThreadJukebox is create. The jukebox thread runs in a loop, trying to removing songs from the queue, and playing each song. Since the queue is blocking, if no songs are requested it blocks until a song is added to the queue. Once a song is removed from the queue, the jukebox thread plays the song to completion, and then loops back to try to remove another song from the queue.
- Use a thread pool executor with a single thread. Each invocation of the play method creates a Runnable to play that song, and submits it to the executor. Since the thread pool contains a single thread, only one song will be played at a time.

3. HTML forms (20 points)

Looking at the HTML for the page [http://store.apple.com/ipad](http://store.apple.com/ipad), you see the following as part of the HTML.

```html
<li>
<form action="/buyNow" method="GET">
  iPad, 16GB, Wi-Fi, $499
  <input type="hidden" name="model" value="16gb-wifi" />
  <input type="hidden" name="price" value="499" />
  <input type="submit" value="Buy with 1-click" />
</form>
</li>
<li>
<form action="/buyNow" method="GET">
  iPad, 16GB, Wi-Fi + 3G, $629
  <input type="hidden" name="model" value="16gb-wifi-3g" />
  <input type="hidden" name="price" value="629" />
  <input type="submit" value="Buy with 1-click" />
</form>
</li>
```

- What URL would the browser load if you clicked the “Buy with 1-click” button for the iPad without 3G.

- What serious problem(s) do you see? Explain why they are problems.
  **Answer:** The biggest problem is the hidden form field with the price. If the web server actually trusts and uses that data, then you might be able to buy an iPad for $1 (or maybe even zero or negative dollars, but that would be more likely to cause a sanity check to blow up somewhere in the server. Don’t be greedy, just take it for $1).

  Whether this is a problem or an opportunity depends on whether you are Apple or someone who wants an cheap iPad.

  Instead of trusting a price provied in a parameter, the web server should look it up (depending on the model). If the price it is being offered at depends on other information (e.g., a discount for students), that discount should be applied within the web server, based on information the web server can verify.

  Another problem is that it is a GET request. For many reasons, it is strongly recommended that HTTP requests that change state be POST requests, rather than GET. Among other things, it browsers don’t resend POST requests if you hit reload, unless you specifically OK it.
Finally, the URL should be https: rather than http: With http: it is too easy for someone to snoop on the connection and steal your session identifier. Technically, this isn’t really a problem with the HTML, but with the URL used to reach it.

4. Decorator (15 points)

The Iterator interface is given below:

```java
interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Write/implement the following two Iterator decorators:

(a) **Non-removing iterator** This decorator is used to create a decorator that prevents remove from being invoked on the decorated Iterator by throwing an UnsupportedOperationException if a call to remove is made.

**Answer:**

```java
public class NonRemovingIterator<E> implements Iterator<E> {
    final Iterator<E> i;
    public NonRemovingIterator(Iterator<E> i) {
        this.i = i;
    }
    public boolean hasNext() {
        return i.hasNext();
    }
    public E next() {
        return i.next();
    }
    public void remove() {
        throw new UnsupportedOperationException("remove not supported");
    }
}
```
(b) **enhanced iterator** This decorator is used to support an additional capability: a method \( \text{E curr()} \) that returns the value most recently returned by \( \text{next()} \). Calling \( \text{curr()} \) is idempotent: calling it multiple times has in a row just returns the same value. Calling \( \text{curr()} \) before \( \text{next()} \) has been called results in a \text{IllegalStateException} being thrown.

**Answer:**

```java
class EnhancedIterator<E> implements Iterator<E> {
    boolean canCallCurrent;
    E current;
    final Iterator<E> i;

    public EnhancedIterator(Iterator<E> i) {
        this.i = i;
    }

    public boolean hasNext() {
        canCallCurrent = false;
        return i.hasNext();
    }

    public E next() {
        current = i.next();
        canCallCurrent = true;
        return current;
    }

    public void remove() {
        i.remove();
        canCallCurrent = false;
    }

    public E curr() {
        if (!canCallCurrent) throw new IllegalStateException();
        return current;
    }
}
```

5. Testing a Semaphore (15 points)

Consider a Semaphore initialized to hold two permits. We want to test the following situation. One thread acquires two permits, which is granted immediately. It then tries to acquire another permit, and this request blocks. Then, another thread releases a permit, and this allows the first thread to proceed.

Write a multithreaded test case, using the metronome timer and testing framework discussed in class and used in project 5, that tests this scenario.

As a reminder of how the metronome timer and testing framework works, here is the public test from project 5 that checks that two locks provide mutual exclusion.

```java
class TwoLocksProvideMutualExclusion extends MultithreadedTestCase {
    final Lock lock0, lock1;

    public TwoLocksProvideMutualExclusion(Lock lock0, Lock lock1) {
        this.lock0 = lock0;
    }
}
```
```java
this.lock1 = lock1;
}

public void thread0() {
    lock0.lock();
    assertEquals(0, getTick());
    waitForTick(2);
    lock0.unlock();
    assertEquals(2, getTick());
}

public void thread1() {
    waitForTick(1);
    lock1.lock(); // should block here
    assertEquals(2, getTick());
    waitForTick(3);
    lock1.unlock();
    assertEquals(3, getTick());
}
}

For your case, you should only worry about testing the methods shown below. Note: for the unlike
the standard Semaphore class, the methods don’t throw InterruptedException, and you don’t have to worry about interrupts.

public class Semaphore {
    ...
    // the only methods you should invoke in your test cases.
    public void acquire(int permits) { ... }
    public void release(int permits) { ... }
}

Your MultithreadedTestCase for the scenario described:

static class CheckSemaphoreWithTwoPermits extends MultithreadedTestCase {

    final Semaphore semaphore = Semaphore(2);

    public void thread0() {
        semaphore.acquire(2);
        assertEquals(0, getTick());
        semaphore.acquire(1);
        assertEquals(1, getTick());
    }

    public void thread1() {
        waitForTick(1);
        semaphore.release(1);
        assertEquals(1, getTick()); // optional sanity check
    }
}
```
6. A buggy semaphore (20 points)

We have a semaphore implementation, part of which is provided below. It seems to work in all of the test cases with one or two threads. But in fails in one test case with three threads. If you have a semaphore with no remaining permits, and two threads each try to acquire a permit, both requests are blocked, as they should be. But then, if another thread releases one permit, both of the blocked acquire requests become unblocked, but only one of them should. Based on this bug report, provide your best guess as to what the problem is, and what the code for acquire looks like.

(a) Description of likely problem

**Answer:** The most likely cause is that the acquire method doesn’t call `wait()` in a while loop. Both threads tested the number of permits, saw that it was zero, and called wait(). The call to notifyAll woke up both threads. One got the lock first, and decremented the number of permits to zero, and then the other thread got the lock, didn’t recheck the number of permits, and changed the number of permits from 0 to -1.

(b) Example of code that would have this problem (you are free to ignore the fact that wait is declared to throw InterruptedException).

```java
public class Semaphore {
    int permits;
    public Semaphore(int permits) { this.permits = permits; }
    public synchronized void release(int p) {
        // in original midterm, this notifyAll was done only if permits was zero
        // that was incorrect, but wasn't involved in the bug described above
        this.notifyAll();
        this.permits += p;
    }
    public synchronized void acquire(int p) {
        if (this.permits < p)
            this.wait();
        this.permits -= p;
    }
}
```

(c) Correct implementation of acquire (consistent with the design above):

```java
public synchronized void acquire(int p) {
    while (this.permits < p)
        this.wait();
    this.permits -= p;
}
```