Frequently asked questions from the previous class survey

- Race conditions in `suspend()`, `resume()`, `stop()`
- What happens if a main thread dies before child threads?
- When would it be good to `interrupt()`?
- Calling `interrupt` generates an OS/hardware interrupt?
- How does one thread execute another thread's `run()` method, should it not call `start()`?
- How does OS share between threads?
- Thread context switching: Why is it faster than context switching between processes?

Topics covered in this lecture

- Data synchronization
- Synchronized blocks
- Lock fairness
- Wait-notify

Heisenbugs

- Term coined by ACM Turing Award winner Jim Gray
- Pun on the name of Werner Heisenberg
- Act of observing a system, alters its state!
- Describes a particular class of bugs
  - Those that disappear or change behavior when you try to examine them
- Multithreaded programs are a common source of Heisenbugs

What about regular bugs?

- Sometimes referred to as Bohr bugs
  - Deterministic
  - Generally much easier to diagnose

Reasoning about interleaved access to shared state: Too much milk!

<table>
<thead>
<tr>
<th>Time</th>
<th>Roommate 1's actions</th>
<th>Roommate 2's actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in fridge; out of milk</td>
<td>Look in fridge; out of milk</td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store</td>
<td>Leave for store</td>
</tr>
<tr>
<td>3:10</td>
<td>Arrive at store</td>
<td>Arrive at store</td>
</tr>
<tr>
<td>3:15</td>
<td>Buy milk</td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive home; put milk away</td>
<td>Arrive home; put milk away</td>
</tr>
<tr>
<td>3:25</td>
<td></td>
<td>Oh no!</td>
</tr>
<tr>
<td>3:30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why sharing data between threads is problematic

- **Race conditions**
- Threads attempt to access data more or less simultaneously
  - A thread may change the value of data that some other thread is operating on

Example code with race condition

```java
public class MyThread extends Thread {
    private byte[] values;
    private int position;

    public void modifyData(byte[] newValues, int newPosition) {
        // Modify values and position
    }

    public void utilizeDataAndPerformFunction() {
        // Use values and position
    }

    public void run() {
        // Main logic
    }
}
```

In the previous snippet a race condition exists because ...

- The thread that calls `modifyData()` is accessing the same data as the thread that calls `utilizeDataAndPerformFunction()`
- `modifyData()` and `utilizeDataAndPerformFunction()` are not atomic
  - It is possible that `values` and `position` are changed while they are being used

What is atomic?

- The code cannot be interrupted during its execution
  - Accomplished in hardware or simulated in software
- Code that cannot be found in an intermediate state

Eliminating the race condition using the `synchronized` keyword

- If we declared both `modifyData()` and `utilizeDataAndPerformFunction()` as synchronized?
  - Only one thread gets to call either method at a time
    - Only one thread accesses data at a time
  - When one thread calls one of these methods, while another is executing one of them?
    - The second thread must wait
Example code with no race conditions by using the synchronized keyword:

```java
public class MyThread extends Thread {
    private byte[] values;
    private int position;

    public synchronized modifyData(byte[] newValues, int newPosition) {
        ... Modify values and position
    }

    public synchronized utilizeDataAndPerformFunction() {
        ... Use values and position
    }

    public void run() {
        ... Main logic
    }
}
```

Revisiting the mutex lock:
- **Mutually exclusive lock**
- If two threads try to grab a mutex?
  - Only one succeeds
- In Java every object has an associated lock

When a method is declared synchronized...
- The thread that wants to execute the method must **acquire** a lock
- Once the thread has acquired the lock?
  - It executes method and **releases** the lock
- When a method returns, the lock is released
  - Even if the return is because of an exception

Locks and objects:
- There is only **one lock per object**
- If two threads call synchronized methods of the same object?
  - Only one can execute immediately
    - The other has to wait until the lock is released

Another code snippet to look at...

```java
public class MyThread extends Thread {
    private boolean done = false;

    public void run() {
        while (!done) {
            ... Main logic
        }
    }

    public void setDone(boolean isDone) {
        done = isDone;
    }
}
```

Can’t we just synchronize the two methods as we did previously?
- If we synchronized both `run()` and `setDone()`?
  - `setDone()` would never execute!
- The `run()` method does not exit until the done flag is set
  - But the done flag cannot be set because `setDone()` cannot execute till `run()` completes
  - Uh oh ...

The problem stems from the scope of the lock

- **Scope of a lock**
  - Period between grabbing and releasing a lock
  - Scope of the `run()` method is too large!
    - Lock is grabbed and never released
  - We will look at techniques to *shrink the scope* of the lock
  - But let's look at another solution for now

Let's look at operations performed on the data item (done)

- The `setDone()` method stores a value into the flag
- The `run()` method reads the value
- In our previous example:
  - Threads were accessing *multiple* pieces of data
  - No way to update *multiple* data items *atomically* without the `synchronized` keyword

But Java specifies that the loading and storing of variables is atomic

- Except for `long` and `double` variables
- The `setDone()` should be atomic
  - The `run()` method has only one read operation of the data item
- The race condition should *not* exist
  - But why is it there?

Threads are allowed to hold values of variables in registers

- When one thread changes the value of the variable?
  - Another thread *may not see* the changed variable
- This is particularly true in loops controlled by a variable
  - Looping thread *loads value of variable in register* and *does not notice* when value is changed by another thread

Two approaches to solving this

- Providing setter and getter methods for variable and using the `synchronized` keyword
  - When lock is acquired temporary values stored in registers are flushed to main memory
- The `volatile` keyword
  - Much cleaner solution

If a variable is marked as `volatile`

- Every time it is used?
  - Must be read from main memory
- Every time it is written?
  - Must be written to main memory
- Load and store operations are *atomic*
  - Even for `long` and `double` variables
Some more about volatile variables

- Prior to JDK 1.2 variables were always read from main memory
  - Using volatile variables was moot
- Subsequent versions introduced memory models and optimizations

Synchronization and the volatile keyword

- Can be used only when operations use a single load and store
  - Operations like ++, --?
  - Load-change-store ...
- The volatile keyword forces the JVM to not make temporary copies of a variable
- Declaring an array volatile?
  - The reference becomes volatile
  - The individual elements are not volatile

Synchronizing methods

- Not possible to execute the same method in one thread while ...
  - Method is running in another thread
- If two different synchronized methods in an object are called?
  - They both require the lock of the same object
- Two or more synchronized methods of the same object can never run in parallel in separate threads

A lock is based on a specific instance of an object

- Not on a particular method or class
- Suppose we have 2 objects: objectA and objectB
- One thread can execute objectA.modifyData() while another executes objectB.utilizeData() in parallel
- Two different locks are grabbed by two different threads
- No need for threads to wait for each other

How does a synchronized method behave in conjunction with an unsynchronized one?

- Synchronized methods try to grab the object lock
  - Only 1 synchronized method in a object can run at a time ... provides data protection
- Unsynchronized methods
  - Don’t grab the object lock
  - Can execute at any time ... by any thread
  - Regardless of whether a synchronized method is running
For a given object, at any time ...

- Any number of unsynchronized methods may be executing.
- But only 1 synchronized method can execute.

Synchronizing static methods

- A lock can be obtained for each class.
  - The class lock
- The class lock is the object lock of the Class object that models the class.
  - There is only 1 Class object per class.
  - Allows us to achieve synchronization for static methods.

Object locks and class locks

- Are not operationally related.
- The class lock can be grabbed and released independently of the object lock.
- If a non-static synchronized method calls a static synchronized method?
  - It acquires both locks.

Explicit locking

Many synchronization schemes in J2SE 5.0 onwards implement the Lock interface.

- Two important methods:
  - lock() and unlock()
- Similar to using the synchronized keyword.
  - Call lock() at the start of the method.
  - Call unlock() at the end of the method.

Difference: we have an actual object that represents the lock.
- Store, pass around, or discard.
Semantics of the using Lock

- If another thread owns the lock
  - Thread that attempts to acquire the lock must wait until
    the other thread calls unlock()
- Once the waiting thread acquires the lock, it returns
  from the lock() method

Advantages of using the Lock interface

- Grab and release locks whenever we want
- Now possible for two objects to share the same lock
  - Lock is no longer attached to the object whose method
    is being called
- Can be attached to data, groups of data, etc.
  - Not objects containing the executing methods

Advantages of explicit locking

- We can move them anywhere to adjust lock scope
  - Can span from a line of code to a scope that
    encompasses multiple methods and objects
- Lock at scope specific to problem
  - Not just the object

Synchronized Blocks

Much of what we accomplish with the Lock we can do so with the synchronized keyword

```
public class DataOperator {
  private Lock dataLock = new ReentrantLock();
  public void modifyData(byte[] newValues, int newPosition) {
    try {
      dataLock.lock();
      ... Modify values and position
    } finally {
      dataLock.unlock();
    }
  }
  public void utilizeDataAndPerformFunction() {
    try {
      dataLock.lock();
      ... Use values and position
    } finally {
      dataLock.unlock();
    }
  }
}
```
Synchronized methods vs. Synchronized Blocks

- Possible to use only the synchronized block mechanism to synchronize whole method
- You decide when it's best to synchronize a block of code or the whole method
- **RULE:** Establish as small a lock scope as possible

The Lock interface [java.util.concurrent.locks]

```java
public interface Lock {
    public void lock();
    public void lockInterruptibly()
        throws InterruptedException;
    public boolean tryLock();
    public boolean tryLock(long time, TimeUnit unit)
        throws InterruptedException;
    public void unlock();
    public Condition newCondition();
}
```

Lock Fairness

- ReentrantLock allows locks to be granted fairly
  - Locks are granted as close to arrival order as possible
  - Prevents lock starvation from happening
- Possibilities for granting locks
  1. First-come-first-served
  2. Allows servicing the maximum number of requests
  3. Do what's best for the platform

Threads Notifications

Objects and communications

- Every object has a lock
- Every object also includes mechanisms that allow it to be a waiting area
  - Allows communication between threads

Conditions

- One thread needs a condition to exist
  - Assumes another thread will create that condition
- When another thread creates the condition?
  - It notifies the first thread that has been waiting for that condition
wait(), notify() and the Object class

public class Object {
    public void wait();
    public void wait(long timeout);
    public void notify();
}

Wait-and-notify relate to synchronization, but ...
- It is more of a communications mechanism
- Allows one thread to communicate to another that a condition has occurred
  - Does not specify what that specific condition is

Can wait-and-notify replace the synchronized mechanism?
- No
- Does not solve the race condition that the synchronized mechanism solves
  - Must be used in conjunction with the synchronized lock
    - Prevents race condition that exists in the wait-notify mechanism itself

About the wait() method
- When wait() executes, the synchronization lock is released
  - By the JVM
- When a notification is received?
  - The thread needs to reacquire the synchronization lock before returning from wait()
Integration of wait-notify and synchronization

- **Tightly integrated** with the synchronization lock
- Feature not directly available to us
- Not possible to implement this: native method
- This is typical of approach in other libraries
  - Condition variables for Solaris and POSIX threads require that a mutex lock be held

Details of the race condition in the wait-notify mechanism

- The first thread tests the condition and confirms that it must wait
- The second thread sets the condition
- The second thread calls notify()
  - This goes unheard because the first thread is not yet waiting
- The first thread calls wait()

How does the potential race condition get resolved?

- To call wait() or notify()  
  - Obtain lock for the object on which this is being invoked  
  - It seems as if the lock has been held for the entire wait() invocation, but ...  
    1. wait() releases lock prior to waiting  
    2. Reacquires the lock just before returning from wait()

Is there a race condition during the time wait() releases and reacquires the lock?

- wait() is tightly integrated with the lock mechanism  
  - Object lock is not freed until the waiting thread is in a state in which it can receive notifications  
  - System prevents race conditions from occurring here

If a thread receives a notification is it guaranteed that condition is set?

- No
  - Prior to calling wait(), test condition while holding lock  
  - Upon returning from wait() retest condition to see if you should wait() again

What if notify() is called and no thread is waiting?

- Wait-and-notify mechanism has no knowledge about the condition about which it notifies  
  - If notify() is called when no other thread is waiting?  
    - The notification is lost
What happens when more than 1 thread is waiting for a notification?

- Language specification does not define which thread gets the notification
  - Based on JVM implementation, scheduling and timing issues
- No way to determine which thread will get the notification

notifyAll()

- All threads that are waiting on an object are notified
- When threads receive this, they must work out
  1. Which thread should continue
  2. Which thread(s) should call wait() again
     - All threads wake up, but they still have to reacquire the object lock
     - Must wait for the lock to be freed

The contents of this slide-set are based on the following references