Announcements

- A7 due on Sunday
- See Piazza for corrections to the starter code and our solution to A6
- Lots of office hours, including over the weekend
  - Gries 1-3pm
  - Foster 3-4pm
- A8 out next week
- Prelim #2 next Thursday
- Please fill out P2Conflict today!

Reminder

- A "race condition" arises if two threads try and read and write the same data
- In such cases it is possible that we could see the data "in the middle" of being updated
  - A "race condition": correctness depends on the update racing to completion without the reader managing to glimpse the in-progress update
  - Synchronization (also known as mutual exclusion) solves this

Java Synchronization (Locking)

- You can lock on any object, including this

```java
public synchronized void doSomething() {
    //do something...
}
```

behaves the same as...

```java
public void doSomething() {
    synchronized (this) {
        //do something...
    }
}
```

How locking works

- Only one thread can "hold" a lock at a time
  - If several request the same lock, the Java runtime decides which one gets it
- The lock is released when the thread leaves the synchronization block
  - `synchronized(someObject) { protected code }
  - The protected code has a mutual exclusion guarantee: At most one thread can be in it
- When released, another thread can acquire the lock
Locks are associated with objects

- Every Object has its own built-in lock
  - Just the same, some applications prefer to create special classes of objects to use just for locking
  - This is a stylistic decision and you should agree on it with your teammates, or learn the company policy if you work at a company
  - A piece of code is said to be “thread safe” if it can handle multiple threads using it… otherwise it is “unsafe”

Visualizing deadlock

Deadlocks always involve cycles

- They can include two or more threads or processes in a waiting cycle
- Other properties:
  - The locks need to be mutually exclusive (no sharing of the objects being locked)
  - The application won’t give up and go away (no timer associated with the lock request)
  - There are no mechanisms for one thread to take locked resources away from another thread – no “preemption”

Dealing with deadlocks

- We recommend designing code to either
  - Acquire a lock, use it, then promptly release it, or
  - … acquire locks in some “fixed” order
- Example, suppose that we have objects a, b, c, …
- Now suppose that threads sometimes lock sets of objects but always do so in alphabetical order
  - Can a lock-wait cycle arise?
  - … without cycles, no deadlocks can occur!

Higher-level abstractions

- Locking is a low-level way to deal with synchronization
  - A specific mechanism… very nuts-and-bolts
  - So many programmers work with higher level concepts. Sort of like ADTs for synchronization
    - We’ll just look at one example today
    - There are many other alternatives
    - Take CS 4410 to learn more…

A producer/consumer example

- Thread A produces loaves of bread and puts them on a shelf with capacity K
  - For example, maybe K=10
- Thread B consumes loaves by taking them off the shelf
  - Thread A doesn’t want to overload the shelf
  - Thread B doesn’t want to leave with empty arms

... drop that mouse or you’ll be down to 8 lives
Producer/Consumer example

```java
class Bakery {
    int nLoaves = 0; // Current number of waiting loaves
    final int K = 10; // Shelf capacity

    public synchronized void produce() {
        while(nLoaves == K) this.wait(); // Wait until not full
        ++nLoaves;
        this.notifyAll(); // Signal: shelf not empty
    }

    public synchronized void consume() {
        while(nLoaves == 0) this.wait(); // Wait until not empty
        --nLoaves;
        this.notifyAll(); // Signal: shelf not full
    }
}
```

Things to notice

- Wait needs to wait on the same object that you used for synchronizing (in our example, this, which is this instance of the Bakery)
- Method notify wakes up just one waiting thread, notifyAll wakes all of them up
- We used a while loop because we can’t predict exactly which thread will wake up “next”

Bounded Buffer

- Here we take our producer/consumer and add a notion of passing something from the producer to the consumer
  - For example, producer generates strings
  - Consumer takes those and puts them into a file
- Why would we do this?
  - Keeps the computer more steadily busy

```java
class BoundedBuffer<T> {
    int putPtr = 0, getPtr = 0; // Next slot to use
    final int K = 10; // Buffer capacity
    T[] buffer = new T[K];

    public synchronized void produce(T item) {
        while(available == K) this.wait(); // Wait until not full
        buffer[putPtr++ % K] = item;
        ++available;
        this.notifyAll(); // Signal: not empty
    }

    public synchronized T consume() {
        while(available == 0) this.wait(); // Wait until not empty
        --available;
        T item = buffer[getPtr++ % K];
        this.notifyAll(); // Signal: not full
        return item;
    }
}
```

In an ideal world...

- Bounded buffer allows producer and consumer to both run concurrently, with neither blocking
  - This happens if they run at the same average rate
  - ... and if the buffer is big enough to mask any brief rate surges by either of the two
- But if one does get ahead of the other, it waits
  - This avoids the risk of producing so many items that we run out of computer memory for them. Or of accidentally trying to consume a non-existent item.

Trickier example

- Suppose we want to use locking in a BST
  - Goal: allow multiple threads to search the tree
  - But don’t want an insertion to cause a search thread to throw an exception
Code we’re given is thread unsafe

```
class BST<T> {
  String name;  // Name of this node
  T value;     // Value of associated with that name
  BST<T> left, right;  // Children of this node

  // Constructor
  public BST(String who, T what) { name = who; value = what; }

  // Returns value if found, else null
  public T get(String goal) {
    if(name.equals(goal)) return value;
    if(name.compareTo(goal) < 0) return left==null? null: left.get(goal);
    return right==null? null: right.get(goal);
  }

  // Updates value if name is already in the tree, else adds new BST node
  public void put(String goal, T value) {
    if(name.equals(goal)) { this.value = value; return; }
    if(name.compareTo(goal) < 0) {
      if(left == null) { left = new BST<T>(goal, value); return; }
      left.put(goal, value);
    } else {
      if(right == null) { right = new BST<T>(goal, value); return; }
      right.put(goal, value);
    }
  }
}
```

**Attempt #1**
- Just make both `put` and `get` synchronized:
  - `public synchronized Object get(…){ … }`
  - `public synchronized void put(…){ … }

- Let’s have a look….

**Safe version: Attempt #1**

```
class BST<T> {
  String name;  // Name of this node
  T value;     // Value of associated with that name
  BST<T> left, right;  // Children of this node

  // Constructor
  public BST(String who, T what) { name = who; value = what; }

  // Returns value if found, else null
  public synchronized T get(String goal) {
    if(name.equals(goal)) return value;
    if(name.compareTo(goal) < 0) return left==null? null: left.get(goal);
    return right==null? null: right.get(goal);
  }

  // Updates value if name is already in the tree, else adds new BST node
  public synchronized void put(String goal, T value) {
    if(name.equals(goal)) { this.value = value; return; }
    if(name.compareTo(goal) < 0) {
      if(left == null) { left = new BST<T>(goal, value); return; }
      left.put(goal, value);
    } else {
      if(right == null) { right = new BST<T>(goal, value); return; }
      right.put(goal, value);
    }
  }
}
```

**Attempt #1**
- Just make both `put` and `get` synchronized:
  - `public synchronized Object get(…){ … }`
  - `public synchronized void put(…){ … }

- This works but it kills ALL concurrency
  - Only one thread can look at the tree at a time
  - Even if all the threads were doing "get"!

**Visualizing attempt #1**

```
Put(Ernie, eb0)  
Get(Martin)… resumes
Get(Martin)… must wait!
```

- `public synchronized void put(String goal, T value) { … }

**Attempt #2**
- `put` uses synchronized in method declaration
  - So it locks every node it visits
- `get` tries to be fancy:

```
// Returns value if found, else null
public synchronized T get(String goal) {
    synchronized(this) {
      if(name.equals(goal)) return value;
      if(name.compareTo(goal) < 0) return left==null? null: left.get(goal);
      return right==null? null: right.get(goal);
    }
}
```

- Actually this is identical to attempt 1! It only looks different but in fact is doing exactly the same thing
Attempt #3 illustrates risks

- The hardware itself actually needs us to use locking and attempt 3, although it looks right in Java, could actually malfunction in various ways
- Issue: put updates several fields:
  - parent.left (or parent.right) for its parent node
  - this.left and this.right and this.name and this.value
- When locking is used correctly, multicore hardware will correctly implement the updates
- But if you look at values without locking, as we did in Attempt #3, hardware can behave oddly!

Using Locks...

```java
class Counter {
    private static int counter = 0;
    public static synchronized int getCount() {
        return counter++;
    }
}
```

Using Concurrent Collections...

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

class Counter {
    private static AtomicInteger counter;
    public Counter() {
        counter = new AtomicInteger(0);
    }
    public static int getCount() {
        return counter.getAndIncrement();
    }
}
```
More tricky things to know about

- With priorities Java can be very annoying
  - ALWAYS runs higher priority threads before lower priority threads if scheduler must pick
  - The lower priority ones might never run at all

- Consequence: risk of a “priority inversion”
  - High priority thread t1 is waiting for a lock, t2 has it
  - Thread t2 is runnable, but never gets scheduled because t3 is higher priority and “busy”

Summary

- Use of multiple processes and multiple threads within each process can exploit concurrency
  - Which may be real (multicore) or “virtual” (an illusion)

- But when using threads, beware!
  - Must lock (synchronize) any shared memory to avoid nondeterminism and race conditions
  - Yet synchronization also creates risk of deadlocks
  - Even with proper locking concurrent programs can have other problems such as “livelock”

- Serious treatment of concurrency is a complex topic (covered in more detail in cs3410 and cs4410)
- Nice tutorial at http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html