Announcements

- HW9: Java Concurrency
  - Due Monday May 9th
  - Tests in HW Server coming up

- Check your grades in Rainbow Grades!

- Homework Server survey:
  - [http://tinyurl.com/gqvbyv9](http://tinyurl.com/gqvbyv9)
  - Link off of Announcements page

Final Exam is Cumulative

- Programming Language Syntax (Ch. 2.1-2.3.3)
- Logic Programming and Prolog (Ch. 11)
- Scoping (Ch. 3.1-3.3)
- Programming Language Semantics: Attribute Grammars (Sc. Ch. 4.1-4.3)
- Functional programming and Scheme (Ch. 10)
  - map/fold problems as in Exam 2
- Lambda calculus (notes + Ch. 10.6 on CD)
- Data abstraction: Types (Ch. 7.1 – 7.7.1)
- Control abstraction: Parameter Passing (Ch. 8.1-8.3)
- Object-oriented languages (9.1-9.2)
- Concurrency (12.1-12.2). What can go wrong questions
- Dynamic languages and Python (Ch. 13 optional)
- Comparative Programming Languages

Last Class

- Intro to Concurrency
- Concurrency in Java
  - Threads
    - What can go wrong with threads?
    - Synchronized blocks
    - The Executor framework

Today's Lecture Outline

- Java Programming Assignment
  - Consistency with two-phase locking
  - Avoiding deadlock

- Concurrency in Java
  - Threads and locks
  - What can go wrong? Data races, atomicity errors, deadlocks
  - More on synchronization

Java Programming Assignment

- Executing transactions
  - In parallel
  - Transactions are atomic --- they execute at once, as a single indivisible operation

- Any outcome that corresponds to a serialized execution is considered consistent/correct

A: 0, B: 1, C: 2, D: 3
Transaction 1  Transaction 2  Transaction 3
B=C; A=A+B  B=D+A  D=C
Transactions execute under the optimistic (i.e., speculative) concurrency model. Each transaction proceeds optimistically, i.e., expecting that no conflicts will occur. If a transaction detects a conflict with another transaction, it aborts (or rolls back) and retries later.

Transaction 1: $B = C$; $A = A + B$
Transaction 2: $B = D + A$
Transaction 3: $D = C$

What conflicts do we have here?

A conflict occurs if two transactions access the same account “simultaneously” and at least one access is a write.

Transaction 1: $B = C$; $A = A + B$
Transaction 2: $B = D + A$
Transaction 3: $D = C$

A write-write conflict between T1 and T2 (on B)
A read-write conflict between T1 and T2 (on A and B)
A read-write conflict between T2 and T3 (on D)

**Account class**

- **value**
  - // Only ONE writer thread is allowed
- **writer**
  - // MANY reader threads are allowed
- **readers**
- **peek()**
  - // peek the value of Account object
  - // Can be called before OPEN
- **update()**
  - // Must be called after account has
  - // been OPENED (i.e., locked) for
  - // writing by current transaction

**open(writing/reading)**

// tries to “lock” account:
// if it detects conflict then abort else “lock”
// open checks for conflicts:
  - If opening for writing and another transaction holds the “write lock” (i.e., writer != null) abort.
  - Detected write-write conflict!
  - If opening for writing and another transaction holds a “read lock”, abort transaction.
  - Detected read-write conflict!
  - If opening for reading and another transaction holds the “write lock”, abort transaction.

**Example**

Global shared accounts: $A$: 0, $B$: 1, $C$: 2
Task runs transaction $A = B + C$; $B = A + B$
Task works in local cache:
Original values: $A$: 0, $B$: 1, $C$: 2
Local cache: $A$: 3, $B$: 4, $C$: 2
written: $A$: yes, $B$: yes, $C$: no
read: $A$: yes, $B$: yes, $C$: yes
Write back

- Now, task needs to write local cache to global shared accounts
- Here is an outline of the process (but there are details you need to figure out yourselves!)

**Phase 1. Open (i.e., lock) all accounts**

Account.open checks for conflicts
Wrap Account.open. Catch AbortException

**Phase 2. Once all accounts are successfully opened (i.e. locked), write back (commit) to accounts and close (i.e., unlock) all accounts**

One more thing...

- I didn’t specify in what order transaction opens all accounts
- What can happen if transaction opens the accounts in the order it sees them
  - E.g., A = B opens A then B, and B = A opens B then A
  - DEADLOCK!
- Solution: each transaction opens accounts in the same well-defined order, from A to Z

Today’s Lecture Outline

- Java Programming Assignment
  - Consistency with two-phase locking
  - Avoiding deadlock

- Concurrency in Java
  - Threads and Locks
  - What can go wrong? Data races, atomicity errors, deadlocks
  - More on synchronization
  - Memory consistency models

Back to Java. Threads ...

- Java has explicit support for multiple threads
- Two ways to create new threads:
  - Extend java.lang.Thread
    - Override ‘run()’ method
  - Implement Runnable interface
    - Include a ‘run()’ method in your class

- Starting a thread
  - new MyThread().start();
  - new Thread(runnable).start();

- Abstracted away by Executor framework

What Can Go Wrong?

```java
class Account {
    int balance = 0;
    void deposit (int x) {
        this.balance = this.balance + x;
    }
}
class AccountTask implements Runnable {
    public void run() {
        Account act = new Account();
        Main.act.deposit(10); // Account object is shared mutable state.
    }
}
public class Main {
    static Account act = new Account();
    public static void main(String[] args) {
        new Thread(new AccountTask()).start(); // Thread A
        new Thread(new AccountTask()).start(); // Thread B
    }
}
```

Spring 16 CSCI 4430, A Milanova

What Can Go Wrong?

```
Thread A: 
\( r_1 = \text{act.balance} \)
\( r_1 += 10 \)
\( \text{act.balance} = r_1 \)

Thread B:
\( r_2 = \text{act.balance} \)
\( r_2 += 10 \)
\( \text{act.balance} = r_2 \)
```

13
14
15
16
17
18
Using Synchronized Blocks

- How do we make the Account class "safe"?
  - synchronized void deposit(int x) { ... }

Thread A:
```java
synchronized (this) {
  r1 = balance
  r1 += 10
  balance = r1
}
```

Thread B:
```java
synchronized (this) {
  r2 = balance
  r2 += 10
  balance = r2
}
```

This refers to global Account object.

What Can Go Wrong?

Real-world Example (Java 1.1)

```java
class Vector {
  private Object elementData[];
  private int elementCount;

  synchronized void trimToSize() { ... }
  synchronized boolean removeAllElements() {
    elementCount = 0;
    trimToSize();
    return true;
  }
  synchronized int lastIndexOf(Object elem, int n) {
    for (int i = n-1; i >= 0; i--)
      if (elem.equals(elementData[i])) return i;
    return -1;
  }
  int lastIndexOf(Object elem) {
    n = elementCount;
    return lastIndexOf(elem, n);
  }
  ...
}
```

What Can Go Wrong?

A Real-world Example (Java 1.1)

There is a data race on `elementCount`:

Thread A:
```java
removeAllElements
```

Thread B:
```java
lastIndexOf(elem)
```

Will raise an exception because `elementData` has been reset by thread A.

What Can Go Wrong?

`java.lang.StringBuffer (1.4) Has a Bug`

```java
public class StringBuffer {
  private int count;
  private char[] value;

  public synchronized StringBuffer append(StringBuffer sb) {
    if (sb == null) sb = NULL;
    int len = sb.length();
    int newcount = count + len;
    if (newcount > value.length) expandCapacity(newcount);
    sb.getChars(0, len, value, count);
    count = newcount;
    return this;
  }

  public synchronized int length() { return count; }
  public synchronized void getChars(int i, int) {
    ...
}
```

What Can Go Wrong?

`java.lang.StringBuffer (1.4) Has a Bug`

- Method `append` is not "atomic":

Thread A:
```java
sb.length()
```

Thread B:
```java
sb.delete()
```

Will raise an exception because `sb`'s value array has been updated by thread B.
### Atomicity Violation

- Method `StringBuffer.append` is not "atomic"
- Informally, a method is said to be atomic if its “sequential behavior” (i.e., behavior when method is executed in one step), is the same as its “concurrent behavior” (i.e., behavior when method is interrupted by other threads)
  - A method is atomic if it appears to execute in “one step” even in the presence of multiple threads
- Atomicity is a stronger correctness property than race freedom

### Using Synchronization

- Lock-based synchronization helps avoid race conditions and atomicity violations
  - But synchronization can cause deadlocks!
- Lock granularity
  - Synchronized blocks that are too long (i.e., coarse grained locking) sacrifice concurrency and may lead to slow down
    - Force sequential execution as threads wait for locks
  - Synchronized blocks that are too short (i.e., fine grained locking) may miss race conditions!

### Concurrent Programming is Difficult

- Concurrent programming is about managing shared mutable state
  - Exponential number of interleavings of thread operations
  - OO concurrency: complex shared mutable state
    - Defense: immutable classes, objects, or references
    - Defense: avoid representation exposure

### What Can Go Wrong?

```java
public abstract class Writer {
    protected Object lock;
    Writer() { this.lock = this; }
    public void write(char ch) { ... }
    public void write(String str) { ... }
}
```

### What Can Go Wrong? JDK 1.4.2

```java
public abstract class Writer {
    protected Object lock;
    Writer() { this.lock = this; }
    public void write(char ch) { ... }
    public void write(String str) { ... }
}
```

### What Can Go Wrong? (cont.)

```java
public class PrintWriter extends Writer {
    protected Writer out;
    public PrintWriter(Writer out) {
        super(); this.out = out;
    }
}
```
public void print(int x) {
    synchronized (lock) {
        out.write(Integer.toString(x));
    }
}

public void println() {
    synchronized (lock) {
        out.write(lineSeparator);
    }
}

public void println(int x) {
    synchronized (lock) {
        print(x);
        println();
    }
}

Today's Lecture Outline

- Java Programming Assignment
  - Consistency with two-phase locking
  - Avoiding deadlock
- Concurrency in Java
  - Threads and Locks
  - What can go wrong? Data races, atomicity errors, deadlocks
- More on synchronization

Synchronization

Synchronization refers to any mechanism that allows the programmer to control the relative order of thread operations. Synchronization serves two purposes:

- Atomicity
  - Require that thread executes operation “at once”
  - In Java, we use mutex locks and synchronized blocks
- Condition synchronization
  - Require that thread waits on a condition (expressed as a predicate on one or more shared variables)
  - Until Java 5, Java had poor support for condition synchronization (essentially just `wait().notify()`)

Busy-wait Condition Synchronization

- Busy-wait: thread remains active until a condition becomes true; thread isn’t performing anything useful
- Two forms of busy-wait synchronization:
  - Spin lock: intuitively, the thread “spins” in a loop repeatedly, checking if the lock has become available (or the condition has become true)
  - Barrier: a group of threads must all reach a common point; a thread cannot proceed until all other threads have reached the point

Implementing Busy-wait

Implementing busy-wait condition synchronization relies on hardware support for atomic read-modify-write instructions

- Simplest atomic `test_and_set`: sets a boolean variable to true and returns an indication of whether the variable was previously false
- More advanced versions: atomic `compare_and_swap` (CAS): takes 3 arguments, a location, an expected value and a new value; if location’s value is the expected value, CAS swaps value with new value

... in Java

- Java 5 has the `java.util.concurrent` library:
  - `Executor` thread pools
  - `AtomicInteger` etc., which provide atomic read-modify-write methods
  - `Synchronizers` which provide efficient barriers, semaphores, locks, etc.
- Huge improvement of support for condition synchronization