Announcements

- Rainbow grades: HW1-9, Quiz1-6, Exam1-2
- Now grading: HW10-11, Quizzes 7-8
- HW11 due today
- HW12 out today, due December 11th
- Final exam, December 17th

Last Class

- Scala: a functional and object-oriented programming language
- Key ideas
  - Lists: cons, map/reduce, immutable types
  - Scala is functional: anonymous functions, higher-order functions, call-by-name parameters
  - Scala is object-oriented: classes, singleton objects, case classes and pattern matching, traits and mixins
- (We’ll get back to Scala next week)

Topics

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

Concurrency

- Concurrent program
  - Any program is concurrent if it may have more than one active execution context --- more than one “thread of control”
- Concurrency is everywhere
  - A multithreaded web browser
  - An IDE which compiles while we edit
- What are the reasons for the (renewed) interest in concurrency in programming languages?

Concurrency and Parallelism

- Concurrent characterizes a system in which two or more tasks may be underway (at any point of their execution) at the same time
- A concurrent system is parallel if more than one task can be physically active at once
  - This requires more than one processor
Parallelism in Software Systems

- Arises at different granularity
  - From simple and small tasks, to large and complex tasks
- Instruction-level parallelism (ILP)
- Vector parallelism
  - Similar to map
- Thread-level parallelism
  - Tasks are now arbitrarily complex; concurrency is no longer hidden from programmer

Multiprocessor Machines

- Two broad categories of parallel architectures
- Shared-memory machines
  - Those in which processors share common memory
- Non-shared-memory machines
  - Those in which processors must communicate with messages

Aside: What Exactly is a Processor?

- For 30+ years, it used to be the single chip with a CPU, cache and other components
- Now, it can mean a single “device” with multiple chips; each chip can have multiple cores; each core can have multiple hardware threads. Also, subsets of the cores can share different levels of cache

Aside: What Exactly is a Processor?

- Usually OS and programming languages abstract away hardware complexity
- For us, programmers, “processor” means a task/thread of computation
  - Or the hardware that runs thread of computation
- But as we saw many times in this class, abstraction (i.e., improved programmability) comes at a price

Fundamentals of Concurrent Programming

- Two programming models for concurrency
- Shared memory
  - Some program variables are accessible to multiple threads --- threads have shared state
  - Threads communicate (interact) through shared state
- Message passing
  - Threads have no shared state
  - One thread performs explicit send to transmit data to another

Fundamentals of Concurrent Programming

- Communication
  - More formally, refers to any mechanism that allows one thread to obtain information produced by another thread
  - Explicit in message passing models
  - Implicit in shared memory models
- Synchronization
  - Refers to any mechanism that allows the programmer to control the relative order of operations that occur
  - Implicit in message passing models
  - Explicit in shared memory models
Shared Memory Model

- Programming language support for the shared memory model
  - Explicit support for concurrency
    - E.g., Java, C#, Rust: explicit threads, locks, synchronization, etc.
  - Libraries
    - C/C++: The POSIX #include <threads.h>
    - Many types, macros and routines for threads, locks, other synchronization mechanisms
- We will take a closer look at 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

Terminology

- Concurrent programming with shared memory is about managing shared mutable state
  - Shared state – memory locations that can be accessed by multiple threads
  - Mutable state – the value of a location could change during its lifetime
- Atomic action – action that executes on the machine as a single indivisible operation
  - E.g., read the value of variable i is atomic
  - E.g., write the value of variable i is atomic
  - E.g., i++ is not atomic

Java Programming Assignment

- Atomic transactions execute “at once”
  - For example, transaction A=B+C; B=A+B must be executed “at once” (informally, it cannot be interrupted in the middle by another transaction)
- Accounts A: 0, B: 1, C: 2
- Transaction 1: A = B + C; B = A + B
- Transaction 2: B = C + B
- What are the acceptable outcomes?

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() {
        Main.act.deposit(10); // Account object is shared mutable state.
    }
}

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
    }
}
```

```java
Thread A:
act.balance = r1
r1 += 10
act.balance = r1
```

```java
Thread B:
act.balance = r2
r2 += 10
act.balance = r2
```
A common bug: Race Condition

- New types of bugs occur in concurrent programs; race conditions are the most common.
- A data race (a type of race condition) occurs when two threads can access the same memory location “simultaneously” and at least one access is a write.

```java
Thread A: Thread B:
x = 0 if (x != 0) then...
```

The two callers (in thread A and thread B) could receive distinct instances although there should be only one instance.

A common bug: Race Condition

- Check-and-act data race (common data race)

```java
public class LazyInitRace {
    private ExpensiveObject instance = null;
    public ExpensiveObject get Instance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}
```

synchronized Block

- One mechanism to control the relative order of thread operations and avoid race conditions is the synchronized block.
- Use of synchronized:

```java
synchronized (lock) {
    // Read and write of shared state
}
```

synchronized Method

- One can also declare a method as synchronized:

```java
synchronized int m(String x) {
    // blah blah blah
}
```

equivalent to:

```java
int m(String x) {
    synchronized (this) {
        // blah blah blah
    }
}
```

synchronized Blocks

- Every Java object has a built-in intrinsic lock.
- A synchronized block has two parts:
  - A reference to an object that serves as the lock
  - Block of code to be guarded by this lock
- The lock serves as a mutex (or mutual exclusion lock):
  - Only one thread can hold the lock.
  - If thread B attempts to acquire a lock held by thread A, thread B must wait (or block) until thread A releases the lock.

How Do We Make Account “Safe”?

```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.
    }
}

class Main {
    public static void main(String args[]) {
        new Thread(new AccountTask()).start(); // Thread A
        new Thread(new AccountTask()).start(); // Thread B
    }
}
```
Use Synchronized

- To make Account "safe", make deposit synchronized
  - synchronized void deposit(int x) { ... }

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

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

Using Synchronized Blocks

- Synchronized blocks help avoid data races
- Granularity of synchronized blocks
  - Synchronized blocks that are too long (i.e., coarse grained locking) sacrifice concurrency and may lead to slowdown
  - Force sequential execution as threads wait for locks
  - Synchronized blocks that are too short (i.e., fine grained locking) may miss data races!
  - Synchronization can cause deadlock!

Question

- In this code example, does lock guarantee that no two threads ever execute the critical section "simultaneously"?

  synchronized (lock) {
    // Read and write of shared state
  }

Question

- Sequential code:
  ```java
  List data = new ArrayList();
  if (!data.contains(p)) {
    data.add(p);
  }
  ```
- Concurrent code, shared mutable state data:
  ```java
  List data = new ArrayList() created in main thread
  if (!data.contains(p)) {
    data.add(p);
  }
  ```
  is executed by multiple threads

Implementing data Safely

- One attempt is to use Synchronized Collections (since Java 1.2)
  - Created by Collections.synchronizedXYZ methods
    - E.g., List data = Collections.synchronizedList(new ArrayList());
  - All public methods are synchronized on this
  - Even if data is a synchronized List, code still not right. What can go wrong?

Implementing data Safely

- Concurrent Collections (since Java 1.5)
  - E.g., ConcurrentHashMap
  - Provide additional atomic operations
    - E.g., putIfAbsent(key, value)
  - Implement different, more efficient (concurrent) synchronization mechanisms
Organizing Concurrent Applications

- One way to organize concurrent programs:
  - Organize program into tasks
  - Identify tasks and task boundaries
  - Tasks should be as independent of other tasks as possible
    - Ideally, tasks do not depend on mutable shared state and do not write mutable shared state
    - If there is mutable shared state, tasks should be synchronized appropriately!
  - Each task should be a relatively small portion of the total work
- In your Java assignment, what is the task?

Sequential Task Execution

- A Web Server
  ```java
  public class SingleThreadedWebServer {
      public static void main(String[] args)
      throws IOException {
          ServerSocket socket = new ServerSocket(80);
          while (true) {
              Socket connection = socket.accept();
              handleRequest(connection);
          }
      }
  }
  ```
  What problems do you see here?

Explicit Threads for Task Execution

```java
public class ThreadPerTaskWebServer {
    public static void main(String[] args)
    throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }
}
```

The Executor Framework

- Part of java.util.concurrent (Java 1.5)
- Flexible thread pool implementation
- High-level abstraction: Executor, not Thread
- Decouples task submission from task execution
- E.g., Executor e manages a thread pool of 3 threads
  ```java
e = Executors.newFixedThreadPool(3);
e.execute(t1);
e.execute(t2);
e.execute(t3);
e.execute(t4);
e.execute(t5);
e.execute(t6);
```

Using Executor for Task Execution

```java
public class TaskExecutorWebServer {
    private Executor e = Executors.newFixedThreadPool(3);
    public static void main(String[] args)
    throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            e.execute(task); // Task submission,
            // Decoupled from task execution
        }
    }
}
```
Transaction Server (HW12)

- Class Task becomes a Runnable
- In runServer, create an Executor (a thread pool)
  - Send new runnable Task objects using execute to the thread pool
  - Each task processes a single transaction (i.e., one line from input file)

So... What Can Go Wrong?

- New types of bugs occur in concurrent programs
  - Race conditions
  - Atomicity violations
  - Deadlocks
- There is nondeterminism in concurrency, which makes reasoning about program behavior extremely difficult

What Can Go Wrong?
Class Vector (Java 1.1’s ArrayList)

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

    synchronized void trimToSize() { .. }
    synchronized boolean removeAllElements() {
        elementCount = 0; trimToSize();
    }
    synchronized int lastIndexOf(Object elem, int n) {
        for (int i = n; --i > 0)
            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
lastIndexOf(elem)
trimToSize
.. elementData=...
```

Thread B:
```java
lastIndexOf(elem, n) elementCount=n
.. elementData[n-1]...
```

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

What Can Go Wrong?
ArrayList seen = new ArrayList(); // seen is shared state

```java
void search(Node node) {
    Runnable task = new Runnable() {
        public void run() {
            synchronized (this) { // synchronize access to seen
                if (!seen.contains(node.pos))
                    seen.add(node.pos);
                else return;
            }
            // check if current node is a solution
            ..
            // compute legal moves, call search(child)
            ..
        }
    }
    e.execute(task);
}
```

What Can Go Wrong?
java.lang.StringBuffer (Java 1.4)

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

```java
public synchronized int length() { return count; }
```

```java
public synchronized void getChars .. { .. }
```
What Can Go Wrong?
java.lang.StringBuffer (Java 1.4)

- Method `append` is not “atomic”:
  
  Thread A:                      Thread B:
  
  `sb.length()`  
  `sb.delete(...)`  
  `sb.getChars()`

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