Announcements

- HW8 due today
- HW9: Java Concurrency
  - Will post tonight
  - Due May 9th
  - Java infrastructure
    - Download the Eclipse IDE
- Rainbow grades
  - HW 1-6
  - Exam 1-2
  - Quiz 1-7
- Questions/concerns, contact us ASAP

Topics

- Intro to Concurrency
- Concurrency in Java
  - Threads
    - What can go wrong with threads?
  - Synchronized blocks
  - The Executor framework
- 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 main reasons for the (renewed) interest in concurrency in programming languages?

Last Class

- Object-oriented programming languages
  - Benefits of object orientation
  - Subtype polymorphism and dynamic method binding
  - Polymorphism
    - Subtype polymorphism
    - Parametric polymorphism
  - Ad-hoc polymorphism refers to function and operator overloading

All rights reserved. This material is protected under all copyright laws as they currently exist. It is not to be reproduced, copied, or otherwise used without written permission from the author. It is for your use in this course only. If you are in doubt, please consult your institution's guidelines for use of copyrighted material.
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:
  - Essentially, like Scheme’s 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.

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.

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 and necessary in shared memory models

Shared Memory Model

- **Programming language support for the shared memory model**
  - Explicit support for concurrency
    - E.g., Java, C#: 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

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 += x;
    }
}
class AccountTask implements Runnable {
    public void run() {
        Account.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
    }
}
```
What Can Go Wrong?

Thread A:  
\[ r_1 = \text{act.balance} \]
\[ r_1 += 10 \]
\[ \text{act.balance} = r_1 \]

Thread B:  
\[ r_2 = \text{act.balance} \]
\[ r_2 += 10 \]
\[ \text{act.balance} = r_2 \]

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**.

A common bug: Race Condition

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

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

The two callers (in thread A and thread B) could receive **different instances** although there should be only one 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.
Using Synchronized Blocks

How do we make the Account class “safe”?

- 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
}

this refers to global Account object

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 the code example below, does lock guarantee that no two threads ever execute the critical section “simultaneously”?

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

Sequential code:

List data = new ArrayList();
if (!data.contains(p)) {
    data.add(p);
}

Concurrent code, shared mutable state data:
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 the 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?

Concurrent Collections (since Java 1.5)

- E.g., ConcurrentHashMap
- Implement different, more efficient (concurrent) synchronization mechanism
- Provide additional atomic operations
  E.g., putIfAbsent(key, value)

Provide weak iterators – i.e., iterators that do not throw ConcurrentModificationException
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

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?

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();
        }
    }
}
```

Sequential Task Execution

```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);
        }
    }
}
```

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
      - 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);
        }
    }
}
```

Your Java Programming Assignment

- **Class WorkerTask becomes a Runnable**
- In `runServer`, create an **Executor** (a thread pool)
  - Send new runnable **WorkerTask** objects using `execute` to the thread pool
  - One task processes one transaction (i.e., one line from input file). I.e., the task is one transaction