CS131, Spring19 – Discussion 1B

Week 5 (05/02/19)

Administrations

1. TA: Wenhao Zhang (wenhaoz@cs.ucla.edu)
2. Office hour:
    - Time: 9:30am ~ 11:30am, Monday
    - Location: 3rd floor, common area in Eng VI
3. Class Announcements
    - All TAs slides are under **resources tab** on Piazza
    - HW3 dues on **05/06** (This coming Monday)
    - Please dont copy & paste code from stackoverflow or other sources.
4. Today's agenda
    - Java intro
    - JMM
    - HW3

Object-oriented programming (OOP)

Overview

- Objects are the first-class citizens.
- Objects encapsulate related methods and fields
- Example languages e.g. Java, C++, C#, Python, PHP, JavaScript, Ruby, etc

Class

Class is a template for an object. Object is an instance of a class. **All objects created using the same class will have the same methods/fields.**

What are the benefits of OOP?

- Modularity
    - Splitting code into objects can help keep different parts of code separated
- Information-hiding
    - Objects should only interact by using each other's public methods
- Code reuse
    - Objects easy to re-use in other programs
- Pluggability and debugging ease
    - You can easily replace a buggy object with a working one if necessary

Alan Kay's definition of OOP

- Everything is an object
- Objects communicate by sending/receiving messages
- Objects have their own memory
- Every object is an instance of some class
- All objects of a specific type can receive the same messages

Note: Some of these do not apply to all of the modern OOP languages!

Java intro

- We will be using Java 11 for this class.
- We recommend you to use an IDE like Eclipse, Netbeans, IntelliJ.
  - This will give you autocomplete, debugging, syntax highlighting and other features to make your life easier
- Other option would be a text editor (e.g. Emacs, sublime, Vim) + the terminal
  - Compile with: javac fileName.java
  - Run with: java fileName

Hello World

```java
/*HelloWorld.java*/
/*`public` is access modifier, controlling who can access obj's fields*/
/*subclass only inherits public methods/fields*/

public class HelloWorld {
        public static void main(String[] args) {
                System.out.println("Hello, World");
        }
}
```

Java files

- MyClass.java
  - Code for MyClass
- MyClass.class
  - Bytecode for MyClass (Compiled using `javac MyClass.java`)
- Foo.jar
  - Java Archive file; ZIP archive
  - Could contain dependent code files or other resources
  - In HW3, you are provided a jar file containing the necessary code files

Inheritance

```
class Shape {
      void draw() { /* do nothing */ }
}
class Rectangle extends Shape {
      void draw() { /* draw a rectangle */ }
}
class Circle extends Shape {
      void draw() { /* draw a circle */ }
}
class Triangle extends Shape {
      void draw() { /* draw a triangle */ }
}
```

```
Triangle a = new Triangle(); a.draw(); /* draws a triangle */
Shape b = a;
b.draw(); /* draws a triangle */
b = new Circle();
b.draw(); /* draws a circle */

/*Late binding, polymorphism*/
```

Interface

- Defines what a class must be able to do, not how to do it
- Interface can not be instantiated, must create a class that implements that interface
- One class can implement multiple interfaces

```
interface Vehicle {
      public int currentSpeed;
      public void increaseSpeed();
      public void decreaseSpeed();
      public void turnLeft();
      public void turnRight();
}

class Car implements Vehicle {
      public void increaseSpeed() {
             pressGasPedal();
      }
      public void decreaseSpeed() {
             pressBrakePedal();
      }
... rest of the implementations ...
}
```

Abstract Classes

- Abstract classes are a combination of a class and an interface
  - Can't create an object using an abstract class
  - Can define some parts of the class, while leaving other implementations for children
- Classes can extend only one abstract or normal class

```
abstract class Shape {
     abstract void draw();
     void setColor() { /* set color */ }
```

Polymorphism and Method Binding

Binding means connecting a method call to a method body.

- Early binding: binding is performed before the program is run(by compiler or linker if there is one)
- Dynamic/late binding: binding occurs at run time, based on the type of object(Polymorphism is usually involved.)
  - All method binding in Java uses late binding unless the method is static or final.

```
class A{
    void doSomething(){
        print("Class A")
    }
}

class B extends A{
    void doSomething(){
        print("Class B")
    }
}

A x = new A();
B y = new B();
x = y;
x.doSomething();// on Class A or Class B?
```

Pass by Value

Java always passes parameters to methods by value e.g.

```
public void badSwap(int var1, int var2){
    int tmp = var1;
    var1 = var2;
    var2 = tmp;
}
```

Java Memory Model

Java memory model(JMM) describes how threads in the Java programming language interact through memory. Specifically, JMM defines the allowable behavior of multithreaded programs.

Q: where does these vars live in?

```
Set s = new ListSet();
// 's' is on the stack
// the new object is on the heap, call it 'o'
// 's' is a reference to the new object,

s.add("hi");
// dereference 's' and update the object 'o'
// to obtain "hi" value

Set s2 = s;
// s2 now refers to 'o'

s2.remove("hi");
// now 'o' no longer contains "hi"
```

as-if semantics

On moder platform, code is frequently not executed in the order it was written. Processors may execute instructions out of order. Data may be moved between registers, processor caches, main memory in different order than specified by the program. Compiler will reorder the instructions.

As-if semantics: In a single-threaded program, the program should not be able to observe the effects of reorderings. Reordering might be an issue in a incorrectly synchronized multithreaded programs.

```
/*Assuming we are running this code in two threads, concurrently*/
/*What problems could take place?*/

Class Reordering{
    int x = 0, y=0;
    public void writer(){
        x = 1;
        y = 2;
    }

    public void reader(){
        int r1 = x;
        int r2 = y;
    }
}
```

Race condiction

```
/*what could go wrong if T1 and T2 run simultaneously*/
T1: counter++;
T2: counter++;
```

CPU Cache

Multiple levels of caches: each CPU/core can have their own cached values.

Notes on JMM

JMM specification was updated in 2001

- Many online resources are still missing some of the updates…
- Check official documentation on Oracle website

Synchronized keyword

If one thread is executing a synchronized method, all other threads have to wait before entering synchronized methods. Lock within one object - applies to all synchronized methods in that object.

Java guarantees that all other threads will see the changes after a thread leaves a synchronized method. Once a thread is in a synchronized method, it can call other ones within that object.

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Synchronized can also be used for smaller blocks of code. Avoid blocking other threads when it is not necessary.

```
public class SynchronizedCounter {
    private int c = 0;
    public void incrementAndWork() {
        ... computation here ....
        synchronized(this) {
            c++;
        }
        ... computation here ....
    }
}
```

Volatile

Defining variables volatile guarantees that other threads will see the changes immediately. Without volatile, there is no guarantee that threads are not using their locally cached versions of variables.

```
public class SharedObject {
    public volatile int counter = 0;
}
```

Additional guarantees: - Volatile access can not be reordered relative to other reads/writes. - If two threads access the same volatile variable, the second thread is guaranteed to see the same state as the first thread.

Note: Volatile does not prevent our earlier problem where two threads tried to perform counter++ simultaneously! No locks used -> Reads and writes can still happen simultaneously

java.util.concurrent.atomic

Atomic package provides data types with atomic operations. Atomic = All other threads see an operation as if it all happened at once (No intermediate states visible). For example, AtomicInteger can be used to perform cnt++ as an atomic operation:

```
AtomicInteger cnt = new AtomicInteger(5);
cnt.incrementAndGet();
```

AtomicIntegerArray provides an array with atomic/volatile operations. Calling get/set on individual elements is volatile. Calling incrementAndGet and similar methods is atomic.

HW3

Thread safety vs. Performance

In HW #3, you'll compare different synchronization techniques. What's the best compromise between reliability and performance? Some techniques are 100% safe but slow, while others are faster but not safe.

Background

We have an array containing integer values between 0..maxval:

Pos	0	1	2	...	n-1	n
Value	5	98	4	...	3	12

Only one operation allowed: swap(i,j):

- This operation decreases ith value by 1 and increases jth value by 1
- E.g. swap(0,1) would update the first two values to 4 and 99 respectively.

The question is:

- We want to call swap(i,j) millions of times efficiently
- How can we make it fast and reliable?

Checking for synchronization problems

The only efficient way to check the correctness is to check:

- Sum of all the values should be the same as in the beginning
- Value at each location is between 0..maxval -These checks can only show that there was a synchronization problem!

Data Structure - State.java

Your solutions will implement interface State:

```java
interface State {
    int size();
    byte[] current();
    boolean swap(int i, int j);
}
```

NullState.java - Dummy implementation

Note that this implementation passes our sanity checks!

```
class NullState implements State {
    private byte[] value;
    NullState(byte[] v, byte maxval) { value = v; }
    public int size() { return value.length; }
    public byte[] current() { return value; }
    public boolean swap(int i, int j) { return true; }
}
```

SynchronizedState.java - Safe but inefficient

```
class SynchronizedState implements State {
    private byte[] value;
    private byte maxval;
    SynchronizedState(byte[] v) { value = v; maxval = 127; }
    SynchronizedState(byte[] v, byte m) { value = v; maxval = m; }
    public int size() { return value.length; }
    public byte[] current() { return value; }
    public synchronized boolean swap(int i, int j) {
        if (value[i] <= 0 || value[j] >= maxval) {
            return false;
        }
        value[i]--; value[j]++; return true;
    }
}
```

SwapTest.java

Contains test code for one thread: Runs state.swap(a,b) with random values a and b as many times as specified

```
class SwapTest implements Runnable {
    private int nTransitions;
    private State state;
    SwapTest(int n, State s) { ... }
    public void run() { ... }
}
```

Runnable interface defines that a Thread object can run this code. Must have run() method.

UnsafeMemory.java

Contains the main method of the code:

1. Parse command line parameters
   - (model name, # threads, # swaps, initial values)
2. Initialize the state object
3. Create and start threads (by running SwapTest objects in multiple threads)
4. Wait for threads to finish (keeping track of time)
5. Verify that the state is consistent (sum hasn't changed, values within bounds)

Task 1

Implement an UnsynchronizedState class

- Similar to SynchronizedState.java, except without the synchronized keyword

Should be a faster solution, but does not guarantee safety

- Might sometimes run into other problems too, depending on parameters…

Task 2

Implement GetNSet, which is a compromise between synchronized and unsynchronized state classes. You should use AtomicIntegerArray class, which provides volatile access to array elements. Use only get/set methods.

Task 3

Design and implement class BetterSafe, which is faster than the synchronized class while providing perfect thread safety.

- Performance difference might be very insignificant on latest Java versions

Also, check packages:

- java.util.concurrency
- java.util.concurrent.atomic
- java.util.concurrent.locks
- java.lang.invoke.VarHandle

Task 4

Integrate all the state classes into one program UnsafeMemory

```
if (args[0].equals("Null"))
    s = new NullState(stateArg, maxval);
else if (args[0].equals("Synchronized"))
    s = new SynchronizedState(stateArg, maxval);
/* Add your object initializations here */
else
    throw new Exception(args[0]);\
```

Task 5 & 6

- Measure and characterize the performance and reliability of each class
- Compare these measurements
- Use OpenJDK 9 and 11.0.2
- Make sure you test on SEASnet!

Reprot

Write a 2-3 page report discussing:

- Pros/cons of the four packages that you were given for BetterSafe implementation
- Why your solution is faster than Synchronized and why it is still 100% reliable
- Discuss any problems you had to overcome to do your measurements properly
- Explain why your class is free of data races, or if it isn't (due to a bug), show how to reproduce the problem (i.e. how to run the program in a way that it is likely to fail)