Concurrent Programming with Java Threads

Tutorial for

Programming Languages Laboratory (CS 431)

July to November -2016

Instructor: Dr. Samit Bhattacharya

Indian Institute of Technology Guwahati

Before we start

- Lab has three components (concurrent, functional, logic prog)
  - One tutorial for each of the components
  - Assignments for each component carries 25% weightage
- 25% on written test and/or viva at the end
- Assignments to be done in groups (hope already formed) – will be checked by TAs
- Copying is strictly prohibited (if caught at stage will lead to F for the whole course)

Before we start

- Head TA
  - Subrata Tikadar (t.subrata)
  - Sandeep Vidyapu (s.vidyapu)
  - Sheel Sindhu Manohar (sheel.manohar)

- Doubt clearing
  - On Moodle discussion forum
  - TAs (mutually-agreed time)
  - Instructor (Monday 4-5 PM at my office)

Java

- Hope you know
  - If not, follow the link (or any other of the numerous online tutorials) to learn the basics

    https://docs.oracle.com/javase/tutorial/java/

- We'll discuss only the concurrency concepts in Java

Basics revisited

- Concurrency – doing things simultaneously

- Concurrent programming – doing things (tasks) simultaneously (mainly at the application/user level)
  - Accessing slow I/O devices
  - Interacting with humans
  - Reducing latency by deferring work
  - Servicing multiple network clients
  - Computing in parallel on multi-core machines

Basics revisited

- Process vs threads
  - Concurrency using multi-threading
- Why threads (example – interactive system response time)
  - Important requirement – synchronization
- Synchronization: Methods to manage and control concurrent access to shared data by multiple-threads

Multi-threading in Java

- Prior to Java 5
  - Main focus: multithreading through time-slicing (pseudo-parallelism)

- Java 5 and afterwards
  - Many more exclusive constructs
  - Targeted to utilize multi-core architecture (parallelism)

Threads in Java (Prior to Java 5)

➢There are two ways to create a thread in java (need to import java.lang.Thread)

1. By extending Thread class

```
class Multi extends Thread{
    public void run(){
    System.out.println("thread is running...");
}
public static void main(String args[]){
     Multi t1=new Multi();
     t1.start();
    }
}
```

Output: thread is running...

2. By implementing Runnable interface

```
class Multi3 implements Runnable{
    public void run(){
    System.out.println("thread is running...");
}

public static void main(String args[]){
    Multi3 m1=new Multi3();
    Thread t1 =new Thread(m1);
    t1.start();
     }
}
```

Output: thread is running...

Thread class (contd..)

- Class Thread: it's method run() does its business when that thread is run
- But you never call run(). Instead, you call start() which lets Java start it and call run()
- To use Thread class directly
  - define a subclass of Thread and override run()
  - Create a task as a Runnable, link it with a Thread, and then call start() on the Thread.
    - The Thread will run the Runnable's run() method.

Common Constructors of Thread class

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Common methods of Thread class

- **public void run():** is used to perform action for a thread.
- **public void start():** starts the execution of the thread. JVM calls the run() method on the thread.
- **public void sleep (long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **public void join():** waits for a thread to die.
- **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
- **public int getPriority():** returns the priority of the thread.
- **public int setPriority(int priority):** changes the priority of the thread.
- **public String getName():** returns the name of the thread.
- **public void setName(String name):** changes the name of the thread.
- **public Thread currentThread():** returns the reference of currently executing thread.
- **public int getId():** returns the id of the thread.
- **public Thread.State getState():** returns the state of the thread.
- **public boolean isAlive():** tests if the thread is alive.
- **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **public void suspend():** is used to suspend the thread(depricated).
- **public void resume():** is used to resume the suspended thread(depricated).
- **public void stop():** is used to stop the thread(depricated).
- **public boolean isDaemon():** tests if the thread is a daemon thread.
- **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
- **public void interrupt():** interrupts the thread.
- **public boolean isInterrupted():** tests if the thread has been interrupted.
- **public static boolean interrupted():** tests if the current thread has been interrupted

Runnable interface

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread
- Runnable interface have only one method named **run()**
- **public void run():** is used to perform action for a thread

Starting a thread

- **start() method** of Thread class is used to start a newly created thread. It performs following tasks:
  - A new thread starts(with new callstack).
  - The thread moves from New state to the Runnable state.
  - When the thread gets a chance to execute, its target run() method will run.

Example

```
public class Worker implements Runnable
{
  public static void main (String[] args)
  {
    System.out.println("This is currently running on the main thread, " +
            "the id is: " + Thread.currentThread().getId());
    Worker worker = new Worker();
    Thread thread = new Thread(worker);
    thread.start();
  }

  @Override
  public void run()
  {
    System.out.println("This is currently running on a separate thread, " +
            "the id is: " + Thread.currentThread().getId());
  }
}
```

Output:

This is currently running on the main thread, the id is: 1
This is currently running on a separate thread, the id is: 9

Synchronization is Important

- Every Java object with a *critical section* of code gets a lock associated with the object
- To enter critical section a thread need to obtain the corresponding object's lock

- **General Syntax :**

```
synchronized (object)
{
//statement to be synchronized
}
```

Example with no-Synchronization

```
class First{
 public void display(String msg) {
  System.out.print ("["+msg);
  try  {
   Thread.sleep(1000);
  }
  catch(InterruptedException e)  {
   e.printStackTrace();
  }
  System.out.println ("]");
  }
 }
```

```
class Second extends Thread{
 String msg;
 First fobj;
 Second (First fp,String str) {
  fobj = fp;
  msg = str;
  start();
 }
 public void run() {
  fobj.display(msg);
 }
}
```

Example with no-Synchronization (contd..)

```
public class Syncro
{
 public static void main (String[] args)  {
  First fnew = new First();
  Second ss = new Second(fnew, "welcome");
  Second ss1= new Second (fnew,"new");
  Second ss2 = new Second(fnew,
"programmer");
 }
}
```

Output:

[welcome [new [programmer]

]

]

In this program, object **fnew** of class First is shared by all the three running threads (ss, ss1 and ss2) to call the shared method(*void* **display**). Hence the result is unsynchronized and such situation is called **Race condition**

Synchronized Keyword

- To synchronize the program, we must *serialize* access to the shared **display()** method, making it available to only one thread at a time

- This is done by using keyword **synchronized** with display() method

Example with Synchronization

```
class First{
 public void display(String msg) {
  System.out.print ("["+msg);
  try  {
   Thread.sleep(1000);
  }
  catch(InterruptedException e)
  {
   e.printStackTrace();
  }
  System.out.println ("]"); }}
```

```
class Second extends Thread{
 String msg;
 First fobj;
 Second (First fp,String str) {
  fobj = fp;
  msg = str;
  start();
 }
 public void run() {
  synchronized(fobj)      //Synchronized block
  {    fobj.display(msg);   } }}
```

Example with Synchronization (contd..)

```
public class Syncro1{
 public static void main (String[] args) {
  First fnew = new First();
  Second ss = new Second(fnew,
"welcome");
  Second ss1= new Second
(fnew,"new");
  Second ss2 = new Second(fnew,
"programmer");
 }
}
```

Output:

[welcome]

[new]

[programmer]

Because of synchronized block this program gives the expected output

Synchronization (prior to Java 5)

- Synchronized methods

```
public class SynchronizedCounter {
    public synchronized void update(int x) {
    count += x;
    }
    public synchronized void reset {
    count = 0;
    }
}
```

Synchronization (prior to Java 5)

- Synchronized statements

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();

    public void inc1() {
        synchronized(lock1) { c1++; }
}
```

Join

- The **Thread** class defines various primitive methods you could not implement on your own
  - For example: **start**, which calls **run** in a new thread

- The join() method is one such method, essential for coordination in this kind of computation
  - Caller blocks until/unless the receiver is done executing (meaning its **run** returns)
  - E.g. in method foo() running in "main" thread, we call:
    myThread.start(); myThread.join();
  - Then this code waits ("blocks") until myThread's run() completes

- Fork-Join framework

Java 5 and later

- Improved concurrent programming support
- Try to exploit multi-core architecture
- Dedicated concurrency support package - *java.util.concurrent*

Atomic and lock objects

- *java.util.concurrent.atomic* contains constructs to work with atomic objects
  - When an atomic object is accessed. The operation either completes or does not take place at all
  - Atomic objects provide a way to implement synchronization without using locks

- *java.util.concurrent.lock* contains constructs to manage locks
  - Defines interfaces and classes for locking and waiting for certain condition
  - Allows creation of own synchronization frameworks different than built-in locking and monitors

Read yourself the package details

Synchronization

- Java.util.concurrent contains several synchronization constructs
  - Semaphor
  - CountDownLatch
  - Barriers
  - Executor

Read yourself the details

Java Volatile Keyword

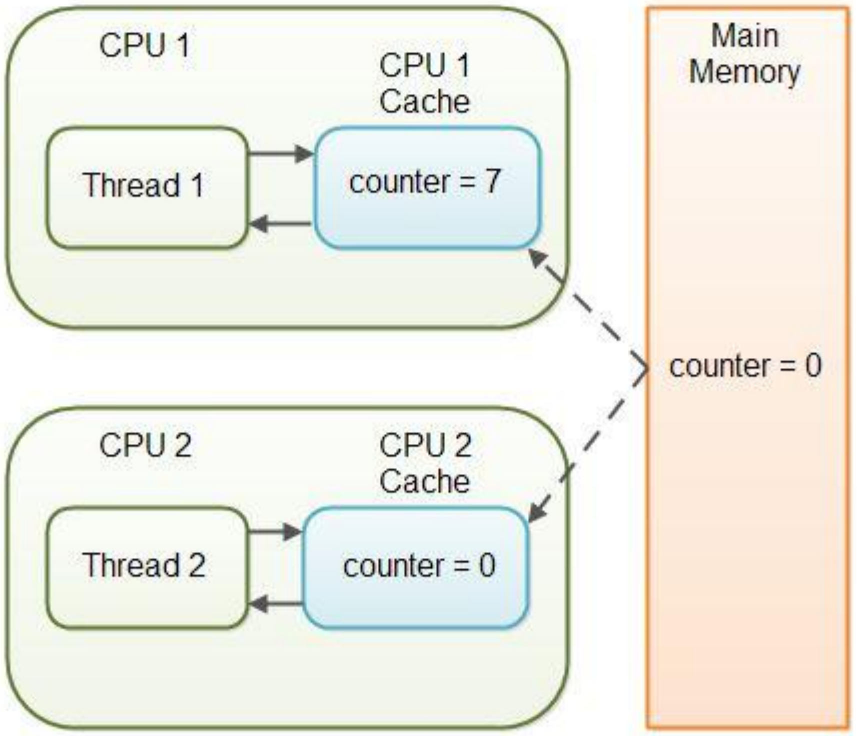
- The Java volatile keyword is used to mark a Java variable as "being stored in main memory"
  - Every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache
  - Every write to a volatile variable will be written to main memory, and not just to the CPU cache
  - From Java 5 onwards the volatile keyword guarantees
    - Visibility of changes to variables across threads
    - Happens-Before relations

The Java volatile Visibility Guarantee (contd..)

- Suppose two threads access a shared object which contains a counter variable declared like this

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

- Only Thread 1 increments counter, but both Threads may read
- If counter not declared volatile, there is no guarantee when the value of the counter variable is written from the CPU cache back to main memory
  - The counter value in CPU cache may not be the same as in main memory



Threads not seeing the latest value of a variable because it has not yet been written back to main memory is called "visibility" problem. The updates of one thread are not visible to other threads.

Image Source: http://tutorials.jenkov.com/java-concurrency/volatile.html

The Java volatile Happens-Before Guarantee

- If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable, will also be visible to Thread B after it has read the volatile variable.

- The reading and writing instructions of volatile variables cannot be *reordered* by the JVM (the JVM may reorder instructions for performance reasons as long as the JVM detects no change in program behaviour from the reordering)
  - Instructions before and after can be reordered, but the volatile read or write cannot be mixed with these instructions
  - Whatever instructions follow a read or write of a volatile variable are guaranteed to happen after the read or write

The Java volatile Happens-Before Guarantee (contd..)

- Look at this:

```
Thread A:
    sharedObject.nonVolatile = 123;
    sharedObject.counter     = sharedObject.counter + 1;
Thread B:
    int counter     = sharedObject.counter;
    int nonVolatile = sharedObject.nonVolatile;
```

- Since Thread A writes the non-volatile variable sharedObject.nonVolatile before writing to the volatile sharedObject.counter, then both sharedObject.nonVolatile and sharedObject.counter are written to main memory when Thread A writes to sharedObject.counter (the volatile variable).
- Since Thread B starts by reading the volatile sharedObject.counter, then both the sharedObject.counter and sharedObject.nonVolatile are read from main memory into the CPU cache used by Thread B. By the time Thread B reads sharedObject.nonVolatile it will see the value written by Thread A

Threading in Swing

- Swing: the package in Java for GUI programming

- Threading matters a lot in Swing GUIs
  - main's thread ends "early"
  - JFrame.setvisible(true) starts the "GUI thread"

- Swing methods run in a separate thread called the **Event-Dispatching Thread** (EDT)
  - Why? GUIs need to be responsive quickly (important for good user interaction)

Threading in Swing

- All operations that update GUI components <u>must</u> happen in the EDT
  - SwingUtilities.invokeLater(Runnable r) is a method that runs a task in the EDT when appropriate

- But execute slow tasks in separate *worker threads*

- To make common tasks easier, use a SwingWorker task

SwingWorker

- A class designed to be extended to define a task for a worker thread

  - Override method **doInBackground()**
    This is like run() – it's what you want to do

  - Override method **done()**
    This method is for updating the GUI afterwards
    - It will be run in the EDT

Assignments - explanation

- Assignment 1 (sensor data fusion) – use different constructs (thread, runnable, lock, atomic, synchronized etc)
- Assignment 2 (sensor data fusion) – use of fork-join framework
- Assignment 3 – calculator with alternate input mechanism (requires Swing)

END