COMP 322: Fundamentals of Parallel Programming

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Lecture 28: Java Threads (contd), synchronized statement

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu
Acknowledgments for Today’s Lecture

- Handout for Lectures 27 and 28
- “Introduction to Concurrent Programming in Java”, Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - Contributing authors: Doug Lea, Brian Goetz
Example of creating Java threads by subclassing Thread (not recommended for wide use)

- This program uses two threads: the main thread and a HelloThread
  - Each prints a greeting – the order of which is nondeterministic

```java
public static void main(String[] args) {
    class HelloThread extends Thread {
        public void run() {
            System.out.println("Hello from thread "+Thread.currentThread().getName());
        }
    }
    Thread t = new HelloThread(); // create HelloThread
    t.start(); // start HelloThread
    System.out.println("Hello from main thread");
}
```

- Program execution ends when both user threads have completed
Example of creating Java threads with Runnable objects (recap)

```
// Start of Task T1 (main program)
sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
// Compute sum1 (lower half) and sum2 (upper half) in parallel
final int len = X.length;
Runnable r1 = new Runnable() {
    public void run(){ for(int i=0 ; i < len/2 ; i++) sum1 += X[i];}
};
Thread t1 = new Thread(r1);
t1.start();
Runnable r2 = new Runnable() {
    public void run(){ for(int i=len/2 ; i < len ; i++) sum2 += X[i];}
};
Thread t2 = new Thread(r2);
t2.start();
// Wait for threads t1 and t2 to complete
t1.join(); t2.join();
int sum = sum1 + sum2;
```

Listing 4: Two-way Parallel ArraySum using Java threads
public class SequentialWebServer {
    public static final int PORT = 8080;
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(PORT);
        while (true) {
            Socket sock = server.accept(); // get next connection
            try {
                processRequest(sock); // do the real work
            } catch (IOException ex) {
                System.err.println("An error occurred ...");
                ex.printStackTrace();
            }
        }
    }
    // ... rest of class definition
Parallelization of Web Server Example using Runnable Tasks

```java
public class ThreadPerTaskWebServer {
    
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(PORT);
        while (true) {
            final Socket sock = server.accept();
            Runnable r = new Runnable() { // anonymous implementation
                public void run() {
                    try {
                        processRequest(sock);
                    } catch (IOException ex) {
                        System.err.println("An error occurred ...");
                    }
                }
            }
            new Thread(r).start();
        }
    }
}
```
Callable Objects can be used to create Future Tasks in Java

- Any class that implements java.lang.Callable<V> must provide a call() method with return type V
- Sequential example with Callable interface

```java
dataclass imageData = imageInfo.downloadImage(1);
dataclass imageData = imageInfo.downloadImage(2);
... renderImage(image1);
renderImage(image2);
```

Listing 5: HTML renderer in Java before decomposition into Callable tasks

```java
callable<ImageData> c1 = new Callable<ImageData>() {
  public ImageData call() { return imageInfo.downloadImage(1); }
};
callable<ImageData> c2 = new Callable<ImageData>() {
  public ImageData call() { return imageInfo.downloadImage(2); }
};
... renderImage(c1.call());
renderImage(c2.call());
```

Listing 6: HTML renderer in Java after decomposition into Callable tasks
4 steps to create future tasks using Callable objects

1. Create a parameter-less callable closure using a statement like “Callable<Object> c = new Callable<Object>() {public Object call()
 { return ...; }}; ”

2. Encapsulate the closure as a task using a statement like “FutureTask<Object> ft = new FutureTask<Object>(c);”

3. Start executing the task in a new thread by issuing the statement, “new Thread(ft).start();”

4. Wait for the task to complete, and get its result by issuing the statement, “Object o = ft.get();”.
Listings 7 and 8: parallelization of HTML renderer example

Listing 7: HTML renderer in Java after parallelization of Callable tasks

```java
Callable<ImagesData> c1 = new Callable<ImagesData>() {
    public ImagesData call() { return imageInfo.downloadImage(1); };
};
FutureTask<ImagesData> ft1 = new FutureTask<ImagesData>(c1);
new Thread(ft1).start();
Callable<ImagesData> c2 = new Callable<ImagesData>() {
    public ImagesData call() { return imageInfo.downloadImage(2); };
};
FutureTask<ImagesData> ft2 = new FutureTask<ImagesData>(c2);
new Thread(ft2).start();
...
renderImage(ft1.get());
renderImage(ft2.get());
```

Listing 8: Equivalent HJ code for the parallel Java code in Listing 7

```java
future<ImagesData> ft1 = async<ImagesData>{return imageInfo.downloadImage(1);};
future<ImagesData> ft2 = async<ImagesData>{return imageInfo.downloadImage(2);};
...
renderImage(ft1.get());
renderImage(ft2.get());
```
Possible states for a Java thread (java.lang.Thread.State)

• **NEW**
  - A thread that has not yet started is in this state.

• **RUNNABLE**
  - A thread executing in the Java virtual machine is in this state.

• **BLOCKED**
  - A thread that is blocked waiting for a monitor lock is in this state.

• **WAITING**
  - A thread that is waiting indefinitely for another thread to perform a particular action is in this state e.g., join()

• **TIMED_WAITING**
  - A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state e.g., join() with timeout

• **TERMINATED**
  - A thread that has exited is in this state.
Thread Lifecycle

- A thread is created by instantiating a `Thread` object
- A thread is started by calling `Thread.start()` on that object
  - Causes execution of its `run()` method in a new thread of execution
- A thread’s state can be inspected by calling `Thread.getState()`
- A thread terminates by:
  - Returning normally from its `run()` method
  - Throwing an exception that isn't caught by any catch block
  - The VM being shut down
- The JVM shuts down when all user (non-daemon) threads terminate
  - Or when shutdown is requested by `System.exit`, CTRL/C, signal, or other process termination triggers
- **Daemon threads** are terminated when JVM shuts down
  - Child thread inherits daemon status from parent thread
  - Override by calling `Thread.setDaemon(boolean)` before starting thread
  - Main thread is started as user thread
HJ isolated statement  
(recap from Lecture 10)

**isolated <body>**

- Two tasks executing isolated statements with interfering accesses must perform the isolated statement in mutual exclusion
  - Two instances of isolated statements, \( \langle \text{stmt1} \rangle \) and \( \langle \text{stmt2} \rangle \), are said to interfere with each other if both access a shared location, such that at least one of the accesses is a write.

  ➔ **Weak isolation guarantee:** no mutual exclusion applies to non-isolated statements i.e., to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances

- Isolated statements may be nested (redundant)

- Isolated statements must not contain any other parallel statement: `async`, `finish`, `get`, `forall`

- In case of exception, all updates performed by <body> before throwing the exception will be observable after exiting <body>
How to implement critical sections and isolated statements in Java?

• Atomic variables can be used to handle special cases of isolated operations on single variable of primitive or reference type
  — Highly recommended that you use java.util.concurrent.atomic whenever it fits your needs

• Need locks for more general cases. Basic idea is to implement isolated <stmt> as follows:
  1. Acquire lock $L_i$
  2. Execute <stmt>
  3. Release lock $L_i$

• The responsibility for ensuring that the choice of locks correctly implements the semantics of isolated lies with the programmer.

• The main guarantee provided by locks is that only one thread can hold a lock at a time, and the thread is blocked when acquiring the lock if the lock is unavailable.
Objects and Locks in Java --- synchronized statements and methods

• Every Java object has an associated lock acquired via:
  – synchronized statements
    - `synchronized( foo ){`  
      // execute code while holding foo’s lock
    }
  – synchronized methods
    - `public synchronized void op1(){`  
      // execute op1 while holding ‘this’ lock
    }

• Language does not enforce any relationship between object used for locking and objects accessed in isolated code
  – If same object is used for locking and data access, then the object behaves like monitors

• Locking and unlocking are automatic
  – Locks are released when a synchronized block exits
    By normal means: end of block reached, return, break
    When an exception is thrown and not caught
Example: Obvious Deadlock

• This code can deadlock if `leftHand()` and `rightHand()` are called concurrently from different threads
  — Because the locks are not acquired in the same order

```java
public class ObviousDeadlock {

  ...
  public void leftHand() {
    synchronized(lock1) {
      synchronized(lock2) {
        for (int i=0; i<10000; i++)
          sum += random.nextInt(100);
      }
    }
  }

  public void rightHand() {
    synchronized(lock2) {
      synchronized(lock1) {
        for (int i=0; i<10000; i++)
          sum += random.nextInt(100);
      }
    }
  }
```

```
Dynamic Order Deadlocks

- There are even more subtle ways for threads to deadlock due to inconsistent lock ordering
  - Consider a method to transfer a balance from one account to another:
    ```java
    public class SubtleDeadlock {
      public void transferFunds(Account from, Account to, int amount) {
        synchronized (from) {
          synchronized (to) {
            from.subtractFromBalance(amount);
            to.addToBalance(amount);
          }
        }
      }
    }
    
    - What if one thread tries to transfer from A to B while another tries to transfer from B to A?
      Inconsistent lock order again - Deadlock!
Avoiding Dynamic Order Deadlocks

- The solution is to *induce* a lock ordering
  - Here, uses an existing unique numeric key
  - public class SafeTransfer {
    public void transferFunds(Account from, Account to, int amount) {
      Account firstLock, secondLock;
      if (fromAccount.acctId == toAccount.acctId)
        throw new Exception("Cannot self-transfer");
      else if (fromAccount.acctId < toAccount.acctId) {
        firstLock = fromAccount;
        secondLock = toAccount;
      } else {
        firstLock = toAccount;
        secondLock = fromAccount;
      }
      synchronized (firstLock) {
        synchronized (secondLock) {
          from.subtractFromBalance(amount);
          to.addToBalance(amount);
        }
      }
    }
  }
Java Locks are Reentrant

- Locks are granted on a per-thread basis
  - Called reentrant or recursive locks
  - Promotes object-oriented concurrent code

- A synchronized block means execution of this code requires the current thread to hold this lock
  - If it does — fine
  - If it doesn't — then acquire the lock

- Reentrancy means that recursive methods, invocation of super methods, or local callbacks, don't deadlock

```java
public class Widget {
    public synchronized void doSomething() { ... }
}

public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        Logger.log(this + ": calling doSomething()");
        super.doSomething(); // Doesn't deadlock!
    }
}
```