Lecture 24: Java Threads, Java synchronized statement

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
Worksheet #23: Synchronized Reply using Pause/Resume

Actors don’t normally require synchronization with other actors. However, sometimes we might want actors to be in sync with one another. Using a DDF and pause/resume, ensure that the SynchSenderActor doesn’t process the next message until notified by the SyncReplyActor that the message was received and processed.

```java
1. class SynchSenderActor
2.   extends Actor<Message> {
3.     private Actor otherActor = ...
4.     void process(Msg msg) {
5.         ...
6.         DDF<T> ddf = newDDF();
7.         otherActor.send(ddf);
8.         pause(); // non-blocking
9.         asyncAwait(ddf, () -> {
10.            T synchronousReply = ddf.get();
11.            println("Response received");
12.            resume(); // non-blocking
13.        });
14.     }
15. }

1. class SynchReplyActor
2.   extends Actor<DDF> {
3.     void process(DDF msg) {
4.         ...
5.         println("Message received");
6.         // process message
7.         T responseResult = ...;
8.         msg.put(responseResult);
9.         ...
10. } }
```
Introduction to Java Threads and the \texttt{java.lang.Thread} class

- Execution of a Java program begins with an instance of Thread created by the Java Virtual Machine (JVM) that executes the program's main() method.
- Parallelism can be introduced by creating additional instances of class Thread that execute as parallel threads.

A lambda can be passed as a Runnable
start() and join() methods

- A Thread instance starts executing when its start() method is invoked
  - start() can be invoked at most once per Thread instance
  - As with async, the parent thread can immediately move to the next statement after invoking t.start()

- A t.join() call forces the invoking thread to wait till thread t completes.
  - Lower-level primitive than finish since it only waits for a single thread rather than a collection of threads
  - No restriction on which thread performs a join on which thread, so it is possible to create a deadlock cycle using join() even when there are no data races
1. // Start of main thread
2. sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
3. Thread t1 = new Thread(() -> {
4.     // Child task computes sum of lower half of array
5.     for(int i=0; i < X.length/2; i++) sum1 += X[i];
6. })
7. t1.start();
8. // Parent task computes sum of upper half of array
9. for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
10. // Parent task waits for child task to complete (join)
11. t1.join();
12. return sum1 + sum2;
1. // Start of Task T0 (main program) 
2. sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields 
3. finish(() -> {
4.     async(() -> {
5.         // Child task computes sum of lower half of array
6.         for(int i=0; i < X.length/2; i++) sum1 += X[i];
7.     });
8.     // Parent task computes sum of upper half of array
9.     for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
10. });
11. // Parent task waits for child task to complete (join)
12. return sum1 + sum2;
HJlib runtime uses Java threads as workers

- HJlib runtime creates a small number of worker threads in a *thread pool*, typically one per core.
- Workers push async’s/continuations into a logical work queue:
  - when an async operation is performed
  - when an end-finish operation is reached
- Workers pull task/continuation work item when they are idle.

Local variables are *private* to each task.

Static & instance fields are *shared* among tasks.
Objects and Locks in Java — synchronized statements and methods

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

- Java language does not enforce any relationship between the 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 a monitor

- 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
Locking guarantees in Java

• It is preferable to use java.util.concurrent.atomic or HJlib isolated constructs, since they cannot deadlock.

• Locks are needed for more general cases. Basic idea is for JVM to implement synchronized(a) <stmt> as follows:
  1. Acquire lock for object a
  2. Execute <stmt>
  3. Release lock for object a

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

• The main guarantee provided by locks is that only one thread can hold a given lock at a time, and the thread is blocked when acquiring a lock if the lock is unavailable.
Deadlock example with Java synchronized statement

- The code below 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);
            }
        }
    }
}
```
Deadlock avoidance in HJ with object-based isolation

- HJ implementation ensures that all locks are acquired in the same order
- \(\Rightarrow\) no deadlock

```java
public class ObviousDeadlock {
    . . .

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

    public void rightHand() {
        isolated(lock2, 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, acctId, to establish an order

```java
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’s Object 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()");
        ...
        doSomething(); // Doesn't deadlock!
    }
}
```
Worksheet #24: Java Threads and Data Races

1) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using start() and join() operations.

2) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using synchronized statements