COMP 322: Fundamentals of Parallel Programming

Lecture 25: Java Threads (contd), Java synchronized statement

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Acknowledgments for Today’s Lecture

- “Introduction to Concurrent Programming in Java”, Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  — Contributing authors: Doug Lea, Brian Goetz
/** Atomically adds delta to the current value.

1. *
2. * @param delta the value to add
3. * @return the previous value
4. */

5. public final int getAndAdd(int delta) {
6.     for (; ;) {
7.         int current = get();
8.         int next = current + delta;
9.         if (compareAndSet(current, next))
10.             // commit
11.             return current;
12.     }
13. }

Assume that multiple tasks call getAndAdd() repeatedly in parallel. Can this implementation of getAndAdd() lead to a) deadlock, b) livelock, c) starvation, or d) unbounded wait? Write and explain your answer below.

c) starvation and d) unbounded wait are both possible
Thread.start() and Thread.join() provide rudimentary support for async and finish. What about monitors, critical sections, isolated?

Interesting.
Let's go straight to Worksheet 24!
Monitors --- an object-oriented approach to isolation (Recap from Lecture 20)

• A monitor is an object containing
  • some local variables (private data)
  • some methods that operate on local data (monitor regions)

• Only one task can be active in a monitor at a time, executing some monitor region
  • Analogous to a critical section for a single object

• Monitors can also be used for
  • Mutual exclusion
  • Cooperation among parallel method invocations
How to convert a sequential library to a monitor in HJ vs. Java?

**HJ approach:**

- Use object-based isolation to ensure that each call to a public method is isolated on “this” e.g.,
  
  ```java
  public void add(...) { isolated(this) { .... } }
  ```

- Can also use general isolated statement, but that is overkill e.g.,
  
  ```java
  public void add(...) { isolated { .... } }
  ```

**Java approach:**

- Use Java’s synchronized statement instead of object-based isolation e.g.,
  
  ```java
  public void add(...) { synchronized(this) { .... } }
  ```
  or equivalently
  
  ```java
  public synchronized void add(...) { .... }
  ```

- Both HJ and Java programs can use specialized implementations of monitors available in java.util.concurrent
  - ConcurrentHashMap, ConcurrentLinkedQueue, CopyOnWriteArraySet
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 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 desirable to use java.util.concurrent.atomic and other standard monitor classes when possible.

- Locks are needed for more general cases. Basic idea is 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 monitors/isolated 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 NoDeadlock1 {
    ...
    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
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()");
            super.doSomething();  // Doesn't deadlock!
        }
    }
    ```
Object-based isolation in HJ does not deadlock

```java
public class NoDeadlock2 {
    public void transferFunds(Account from,
                              Account to,
                              int amount) {
        isolated (from, to) {
            from.subtractFromBalance(amount);
            to.addToBalance(amount);
        }
    }
}
```

- HJ’s implementation guarantees that object-based isolation is deadlock-free
- However, HJ does not permit an inner isolated statement to add a new object e.g., the following code is not permitted in HJ, but the equivalent synchronized version is permitted in Java

```java
Not permitted in HJ (if from != to) Permitted in Java
isolated (from) {
    ...
    isolated (to) { . . . }
}                     synchronized (from) {
    ...
    synchronized(to) { . . . }
}
```

14
Implementation of Java synchronized statements/methods

- Every object has an associated lock
- “synchronized” is translated to matching monitorenter and monitorexit bytecode instructions for the Java virtual machine
  - monitorenter requests “ownership” of the object’s lock
  - monitorexit releases “ownership” of the object’s lock
- If a thread performing monitorenter does not own the lock (because another thread already owns it), it is placed in an unordered “entry set” for the object’s lock
Monitors – a Diagrammatic summary

Figure source: http://www.artima.com/insidejvm/ed2/images/fig20-1.gif
What if you want to wait for shared state to satisfy a desired property?

```java
public synchronized void insert(Object item) { // producer
    // TODO: wait till count < BUFFER SIZE
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
    // TODO: notify consumers that an insert has been performed
}

public synchronized Object remove() { // consumer
    Object item;
    // TODO: wait till count > 0
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    // TODO: notify producers that a remove() has been performed
    return item;
}
```
The Java wait() Method

- A thread can perform a `wait()` method on an object that it owns:
  1. the thread releases the object lock
  2. thread state is set to blocked
  3. thread is placed in the wait set

- Causes thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.

- Since interrupts and spurious wake-ups are possible, this method should always be used in a loop e.g.,

  ```java
  synchronized (obj) {
    while (<condition does not hold>)
      obj.wait();
    ... // Perform action appropriate to condition
  }
  ``

- Java’s wait-notify is related to “condition variables” in POSIX threads
Entry and Wait Sets

acquire lock

object lock

wait

entry set

wait set
The notify() Method

When a thread calls `notify()`, the following occurs:

1. selects an arbitrary thread $T$ from the wait set
2. moves $T$ to the entry set
3. sets $T$ to Runnable

$T$ can now compete for the object’s lock again
Multiple Notifications

- `notify()` selects an arbitrary thread from the wait set.
  - This may not be the thread that you want to be selected.
  - Java does not allow you to specify the thread to be selected.

- `notifyAll()` removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.

- `notifyAll()` is a conservative strategy that works best when multiple threads may be in the wait set.
public synchronized void insert(Object item) {
    while (count == BUFFER SIZE) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
    notify();
}
public synchronized Object remove() {
    Object item;
    while (count == 0) {
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    notify();
    return item;
}
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private int count, in, out;
    private Object[] buffer;
    public BoundedBuffer() { // buffer is initially empty
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];
    }
    public synchronized void insert(Object item) { // See previous slides
    }
    public synchronized Object remove() { // See previous slides
    }
}
Write a sketch of the pseudocode for a Java threads program that exhibits a data race using start() and join() operations.