COMP 322: Fundamentals of Parallel Programming

Lecture 26: Java synchronized statement (contd), Advanced Locking

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Solution to Worksheet #25: Java Threads

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

1. // Start of thread t0 (main program)
2. sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields
3. // Compute sum1 (lower half) and sum2 (upper half) in parallel
4. final int len = X.length;
5. Runnable r1 = new Runnable() {
6.   public void run(){ for(int i=0 ; i < len/2 ; i++) sum1+=X[i];}
7. };
8. Thread t1 = new Thread(r1);
9. t1.start();
10. Runnable r2 = new Runnable() {
11.   public void run(){ for(int i=len/2 ; i < len ; i++) sum2+=X[i];}
12. };
13. Thread t2 = new Thread(r2);
14. t2.start();
15. int sum = sum1 + sum2; // data race between t0 & t1, and t0 & t2
16. t1.join(); t2.join;
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

- “Java Concurrency Utilities in Practice”, Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  —Contributing authors: Doug Lea, Tim Peierls, Brian Goetz

- ECE 3005 course slides from Georgia Tech
  —http://users.ece.gatech.edu/~copeland/jac/3055-05/ppt/ch07-sync-b.ppt

- A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency, Lecture 6, Dan Grossman, U. Washington
Complete Bounded Buffer using Java Synchronization (Recap)

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
    }
}

insert() with wait/notify Methods

1. public synchronized void insert(Object item) {
2.     while (count == BUFFER SIZE) {
3.         try {
4.             wait();
5.         } 
6.         catch (InterruptedException e) { } 
7.     }
8.     ++count;
9.     buffer[in] = item;
10.    in = (in + 1) % BUFFER SIZE;
11.    notify(); // Should we use notifyAll() instead?
12.}
public synchronized Object remove() {
    Object item;
    while (count == 0) {
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    notify(); // Should we use notifyAll() instead?
    return item;
}
Entry and Wait Sets

Scenario for BUFFER_SIZE = 1 with multiple producers (P0, P1, ...) and multiple consumers (C0, C1, ...)

<table>
<thead>
<tr>
<th>Time-step</th>
<th>Entry set</th>
<th>Buffer state</th>
<th>Wait set</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>P0</td>
<td>EMPTY</td>
<td>C0, C1</td>
</tr>
<tr>
<td>t+1</td>
<td>C0, P1</td>
<td>FULL</td>
<td>C1</td>
</tr>
<tr>
<td>t+2</td>
<td>C0</td>
<td>FULL</td>
<td>P1, C1</td>
</tr>
</tbody>
</table>

Problem: notify() may select the “wrong” thread each time, leading to livelock ⇒ use notifyAll() instead.
notify() vs. notifyAll() --- Recap

- **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.
Two Tips for working with Java Threads

• Any variable from an outer scope that is accessed in an anonymous inner class (e.g., in the run() method) must be declared final.

```java
final int len = X.length;
Runnable r = new Runnable() {
    public void run() {
        for(int i=0 ; i < len/2 ; i++) sum1 += X[i];
    }
};
```

• Remember to call the start() method on any thread that you create. Otherwise, the thread’s computation does not get executed.

```java
Thread t = new Thread(r); t.start();
```
Cancelling Threads: Interruption

- Problem: how do we shut down a thread like a web server?
- Need to communicate that shutdown has been requested
  - Could set a flag that is polled in the main loop
    - But main loop could be blocked in `accept()`
- Interruption provides a means of signalling a request to another thread
- Each Thread has an “interrupted status” which is
  - Set when `interrupt()` method is invoked on it
  - Queried by `isInterrupted()` method
- Many blocking methods respect interruption requests and return early by throwing checked `InterruptedException`
  - `Object.wait()`
  - Throwing IE usually clears interrupted status
Calling methods that may throw InterruptedException

- Many methods in Java thread libraries may throw an InterruptedException e.g., <thread>.join(), <object>.wait(),
- When calling any such method, you will either need to include each call to join() in a try-catch block, or add a “throws InterruptedException” clause to the definition of the method that includes the call to join()

- Try-catch example

```java
class Foo implements Runnable {
    public void run() {
        try {
            t1.join();
        }
        catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```
TrafficSignal example (throws clause)

- The `wait` methods will
  - *Atomically* release the lock and block the current thread
  - Reacquire lock before returning
- `notify()` means wake up one waiting thread
- `notifyAll()` means wake up all waiting threads

```java
public class TrafficSignal {
    public enum Color { GREEN, YELLOW, RED }
    private Color color;
    public synchronized void setColor(Color color) {
        this.color = color;
        notifyAll();
    }
    public synchronized void awaitGreen() throws InterruptedException {
        while (color != Color.GREEN) wait(); // waits on “this” object
    }
}
```
Responses to Interruption

- Re-throw IE
  - So caller can handle interruption request

- Cancel and return early
  - Clean up and exit without signalling an error
  - May require rollback or recovery

- Ignore interruption
  - When it is too dangerous to stop
  - Should re-assert interrupted status before returning

- Postpone interruption
  - Remember that interrupt occurred
  - Finish what you are doing and then throw IE

- Throw a general failure exception
  - When interruption is one of many reasons method can fail
Example: Shutting Down the Web Server

```java
public class WebServerWithShutdown {
    private final ServerSocket server;
    private Thread serverThread;

    public WebServerWithShutdown(int port) throws IOException {
        server = new ServerSocket(port);
        server.setSoTimeout(5000); // so we can check for interruption
    }

    public synchronized void shutdownServer() throws IOException {
        if (serverThread == null) throw new IllegalStateException();
        serverThread.interrupt();
        serverThread.join(5000); // wait 5s before closing socket
        server.close(); // to give thread a chance to cleanup
    }

    public synchronized void startServer() {
        if (serverThread == null) {
            serverThread = new Thread() {
                public void run() {
                    while (!Thread.interrupted()) {
                        try { processRequest(server.accept()); } catch (SocketTimeoutException e) { continue; } catch (IOException ex) { /* log it */ }
                    }
                }
            }).start();
        }
    }
}
```

Note: shutdownServer can be harmlessly called more than once
Use of class objects in synchronized statements/methods

- A **class** object exists for every class
- **static synchronized** methods lock the **class** object
- **class** object can be locked explicitly:
  - `synchronized(Foo.class) { /* ... */ }`

- No connection between locking the **Class** object and locking an instance of the class
  - Locking the **Class** object does not lock any instance
  - Instance methods that use static variables must synchronize access to them explicitly by locking the **Class** object

Always use the class literal to get reference to **Class** object—not `this.getClass()` as you may access a subclass object
java.util.concurrent

- General purpose toolkit for developing concurrent applications
  - import java.util.concurrent.*

- Goals: “Something for Everyone!”
  - Make some problems trivial to solve by everyone
    - Develop thread-safe classes, such as servlets, built on concurrent building blocks like ConcurrentHashMap
  - Make some problems easier to solve by concurrent programmers
    - Develop concurrent applications using thread pools, barriers, latches, and blocking queues
  - Make some problems possible to solve by concurrency experts
    - Develop custom locking classes, lock-free algorithms

- HJ approach
  - Build HJ runtime on top of java.util.concurrent library
Key Functional Groups in j.u.c.

- Atomic variables
  - The key to writing lock-free algorithms
- Concurrent Collections:
  - Queues, blocking queues, concurrent hash map, ...
  - Data structures designed for concurrent environments
- Locks and Conditions
  - More flexible synchronization control
  - Read/write locks
- Executors, Thread pools and Futures
  - Execution frameworks for asynchronous tasking
- Synchronizers: Semaphore, Latch, Barrier, Exchanger
  - Ready made tools for thread coordination
Locks

- Use of monitor synchronization is just fine for most applications, but it has some shortcomings
  - Single wait-set per lock
  - No way to interrupt or time-out when waiting for a lock
  - Locking must be block-structured
    - Inconvenient to acquire a variable number of locks at once
    - Advanced techniques, such as hand-over-hand locking, are not possible

- Lock objects address these limitations
  - But harder to use: Need finally block to ensure release
  - So if you don’t need them, stick with synchronized

Example of hand-over-hand locking:
- L1.lock() ... L2.lock() ... L1.unlock() ... L3.lock() ... L2.unlock() ....
interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock(); // return false if lock is not obtained
    boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException;
    void unlock();
    Condition newCondition();
    // can associate multiple condition vars with lock
}

• java.util.concurrent.locks.Lock interface is implemented by java.util.concurrent.locks.ReentrantLock class
Simple ReentrantLock() example

- Used extensively within java.util.concurrent

```java
final Lock lock = new ReentrantLock();
...
lock.lock();
try {
    // perform operations protected by lock
} // perform operations protected by lock
  catch(Exception ex) {
    // restore invariants & rethrow
  }
finally {
    lock.unlock();
}
```

- Must manually ensure lock is released
java.util.concurrent.locks.Condition interface

- Can be allocated by calling ReentrantLock.newCondition()
- Supports multiple condition variables per lock
- Methods supported by an instance of condition
  - void await()  // NOTE: not wait
    - Causes current thread to wait until it is signaled or interrupted
    - Variants available with support for interruption and timeout
  - void signal()  // NOTE: not notify
    - Wakes up one thread waiting on this condition
  - void signalAll()  // NOTE: not notifyAll()
    - Wakes up all threads waiting on this condition
- For additional details see
  - http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Condition.html
BoundedBuffer implementation using two conditions, notFull and notEmpty

class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    ...
BoundedBuffer implementation using two conditions, notFull and notEmpty (contd)

public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
        while (count == items.length) notFull.await();
        items[putptr] = x;
        if (++putptr == items.length) putptr = 0;
        ++count;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
public Object take() throws InterruptedException {
    lock.lock();
    try {
        while (count == 0) notEmpty.await();
        Object x = items[takeptr];
        if (++takeptr == items.length) takeptr = 0;
        --count;
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}
Reading vs. writing

- Recall that the use of synchronization is to protect interfering accesses
  - Multiple concurrent reads of same memory: Not a problem
  - Multiple concurrent writes of same memory: Problem
  - Multiple concurrent read & write of same memory: Problem

So far:
- If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time

But:
- This is unnecessarily conservative: we could still allow multiple simultaneous readers

Consider a hashtable with one coarse-grained lock
- So only one thread can perform operations at a time

But suppose:
- There are many simultaneous lookup operations
- insert operations are very rare
interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}

• Even though the interface appears to just define a pair of locks, the semantics of the pair of locks is coupled as follows
  — Case 1: a thread has successfully acquired writeLock().lock()
    - No other thread can acquire readLock() or writeLock()
  — Case 2: no thread has acquired writeLock().lock()
    - Multiple threads can acquire readLock()
    - No other thread can acquire writeLock()

• java.util.concurrent.locks.ReadWriteLock interface is implemented by java.util.concurrent.locks.ReadWriteReentrantLock class
class Hashtable<K,V> {
    
    // coarse-grained, one lock for table
    ReadWriteLock lk = new ReentrantReadWriteLock();
    V lookup(K key) {
        int bucket = hasher(key);
        lk.readLock().lock(); // only blocks writers
        ... read array[bucket] ...
        lk.readLock().unlock();
    }

    void insert(K key, V val) {
        int bucket = hasher(key);
        lk.writeLock().lock(); // blocks readers and writers
        ... write array[bucket] ...
        lk.writeLock().unlock();
    }
}

Announcements

• Homework 4 due on Friday, March 22nd

• Week 8 Lecture Quiz due on Friday, March 22nd

• Week 9 lab work due via turn-in as usual, but there is no Week 9 Lab Quiz

• Guest lecture on Friday (March 22nd) by Prof. Swarat Chaudhuri on “Speculative parallelization of isolated blocks”
Worksheet #26: use of tryLock()

Extend the transferFunds() method from Lecture 25 to use library locks with tryLock() instead of synchronized, and to return a boolean value --- true if it succeeds in obtaining in obtaining both locks and performing the transfer, and false otherwise. Sketch your answer below using pseudocode. Can you create a deadlock with multiple calls to transferFunds() in parallel?

1. public void transferFunds(Account from, Account to, int amount) {
2.     synchronized (from) {
3.         synchronized (to) {
4.             from.subtractFromBalance(amount);
5.             to.addToBalance(amount);
6.         }
7.     }
8. }