COMP 322: Fundamentals of Parallel Programming

Lecture 27: InterruptedException, Advanced Locking

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

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

Solution to Worksheet #26: Java Threads

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

```
// Start of thread t0 (main program)
sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields
// Compute sum1 (lower half) and sum2 (upper half) in parallel
final int len = X.length;
Thread t1 = new Thread(() -> {
                   for(int i=0 ; i < len/2 ; i++) sum1+=X[i];});
t1.start();
Thread t2 = new Thread(() -> {
                   for(int i=len/2 ; i < len ; i++) sum2+=X[i];});
t2.start();
int sum = sum1 + sum2; // data race between t0 & t1, and t0 & t2
t1.join(); t2.join();
```

Solution to Worksheet #26: Java Threads (contd)

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

```
// Start of thread t0 (main program)
sum = 0; // static int field
Object a = new ... ;
Object b = new ... ;
Thread t1 = new Thread(() -> { synchronized(a) { sum++; } });
Thread t2 = new Thread(() -> { synchronized(b) { sum++; } });
t1.start();
t2.start(); // data race between t1 & t2
t1.join(); t2.join();
```

Objects and Locks in Java --- synchronized statements and methods (Recap)

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

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

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 waiting for a request
- 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 code for InterruptedException in Bounded Buffer example:

```
while (count == BUFFER SIZE) {
            try {
                    wait();
            }
            catch (InterruptedException e) { }
}
```

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

```
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 IE..,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

Locks and Conditions in java.util.concurrent library

- 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() ….

java.util.concurrent.locks.Lock interface

```
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`

```
final Lock lock = new ReentrantLock();
...
lock.lock();
try {
  // 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();
  }
}
```

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

```
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

java.util.concurrent.locks.ReadWriteLock interface

```
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

Example code

```
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();
  }
}
```

Worksheet #27: use of tryLock()

Name: ______________________ Netid: ______________________

Extend the transferFunds() method from Lecture 26 (shown below) to use j.u.c. 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. Assume that each Account object contains a reference to a dedicated ReentrantLock object. Sketch your answer below using pseudocode. Can you create a deadlock with multiple calls to transferFunds() in parallel?

```
public void transferFunds(Account from, Account to, int amount) {
        synchronized (from) {
            synchronized (to) {
                from.subtractFromBalance(amount);
                to.addToBalance(amount);
            }
        }
    }
```