COMP 322: Fundamentals of Parallel Programming

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

Lecture 29: Java synchronized statement with wait/notify

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

Acknowledgments for Today's Lecture

- Combined handout for Lectures 27-29 (to be updated)
- "Introduction to Concurrent Programming in Java", Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - Contributing authors: Doug Lea, Brian Goetz
- ECE 3005 course slides from Georgia Tech
  - http://users.ece.gatech.edu/~copeland/jac/3055-05/ppt/ch07-sync-b.ppt

Announcements

- **Homework 6 due by 5pm on Monday, April 4th**
- **Homework 7 will be assigned on April 4th**
    - **Programming assignment using pure Java (no HJ)**
    - **Choice of projects (per survey feedback)**

Recap of Java synchronized statement/method

- 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

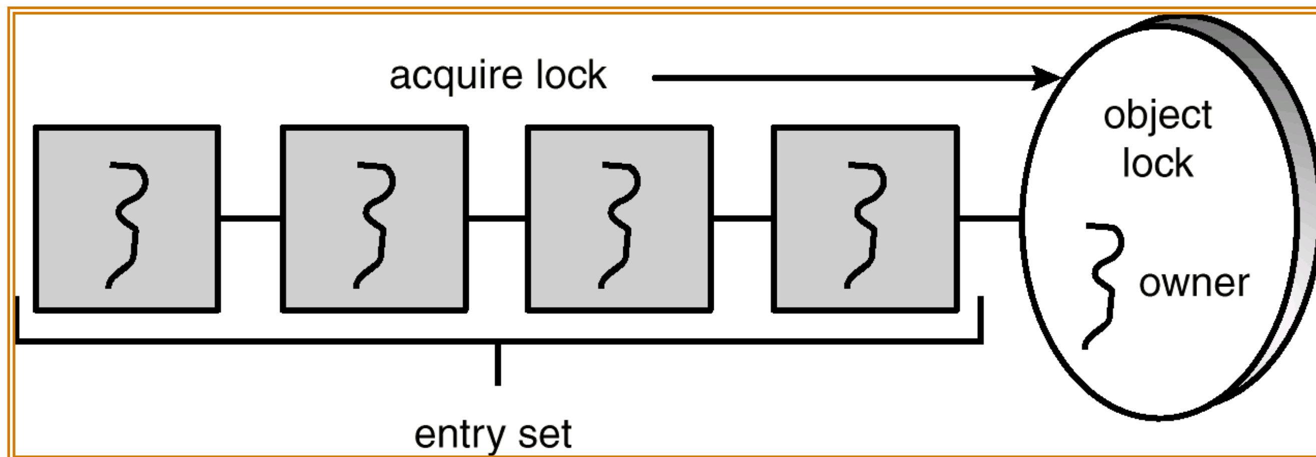
Use of class objects in synchronized statements/methods

- 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

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



What if you want to wait for shared state to satisfy a desired property?

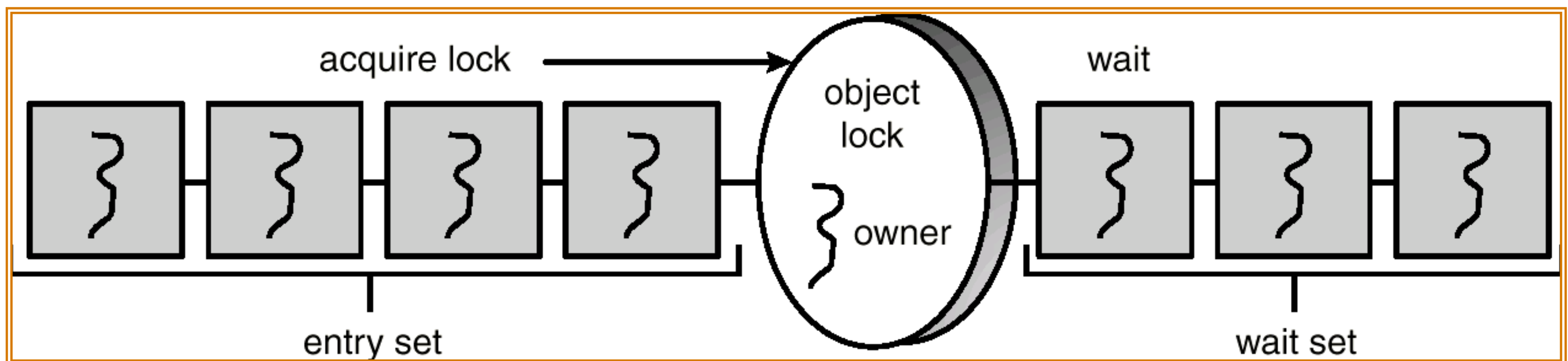
```
public synchronized void insert(Object item) { // producer
   // TODO: wait till count < BUFFER SIZE
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
   // TODO: notify consumers that an insert has been performed
}

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 wakeups are possible, this method should always be used in a loop e.g.,

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

Entry and Wait Sets



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

insert() with wait/notify Methods

```
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();
}
```

remove() with wait/notify Methods

```
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;
}
```

Complete Bounded Buffer using Java Synchronization

```
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
   }
}
```

TrafficSignal example

- 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

```
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();
    }
}
```

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

Dealing with Interruption

- **Golden rule for library and general-purpose task code:**
  - *Never hide the fact that a thread was interrupted!*
  - **Either deal with the exception, or leave evidence of the interruption for your caller**
    - Throw `InterruptedException` yourself
    - Re-assert interrupted status with `interrupt()`

```
public class Foo implements Runnable {
    public void run() {
        try {
            blockingMethod();
        }
        catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } } }
```

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