RACE CONDITIONS AND SYNCHRONIZATION

Lecture 25 – CS2110 – Fall 2014

Race Conditions

- A “race condition” arises if two or more threads access the same variables or objects concurrently and at least one does updates.
- Example: Suppose t1 and t2 simultaneously execute the statement $x = x + 1$; for some static global $x$.
  - Internally, this involves loading $x$, adding 1, storing $x$.
  - If t1 and t2 do this concurrently, we execute the statement twice, but $x$ may only be incremented once.
  - t1 and t2 “race” to do the update.

Example – A Lucky Scenario

- Suppose threads A and B want to call doSomething(), and there is one element on the stack.
  1. thread A tests stack.isEmpty() = false
  2. thread A pops ⇒ stack is now empty
  3. thread B tests stack.isEmpty() ⇒ true
  4. thread B just returns – nothing to do
Example – An Unlucky Scenario

private Stack<String> stack = new Stack<String>();
public void doSomething() {
    if (stack.isEmpty()) return;
    String s = stack.pop();
    // Do something with s...
}

Suppose threads A and B want to call doSomething(), and there is one element on the stack

1. thread A tests stack.isEmpty() => false
2. thread B tests stack.isEmpty() => false
3. thread A pops => stack is now empty
4. thread B pops => Exception!

Synchronization

Java has one “primary” tool for preventing these problems, and you must use it by carefully and explicitly – it isn’t automatic.

Called a “synchronization barrier”
We think of it as a kind of lock
- Even if several threads try to acquire the lock at once, only one can succeed at a time, while others wait
- When it releases the lock, the next thread can acquire it
- You can’t predict the order in which contending threads will get the lock but it should be “fair” if priorities are the same

Solution – with synchronization

private Stack<String> stack = new Stack<String>();
public synchronized void doSomething() {
    synchronized (stack) {
        if (stack.isEmpty()) return;
        String s = stack.pop();
        // Do something with s...
    }
}

• Put critical operations in a synchronized block
• The stack object acts as a lock
• Only one thread can own the lock at a time

Solution – Locking

• You can lock on any object, including this

public synchronized void doSomething() {
    ...
}

is equivalent to

public void doSomething() {
    synchronized (this) {
        ...
    }
}

How locking works

- Only one thread can “hold” a lock at a time
  - If several request the same lock, Java somehow decides which will get it
- The lock is released when the thread leaves the synchronization block
  - synchronized(someObject) { protected code }
- The protected code has a mutual exclusion guarantee: At most one thread can be in it
- When released, some other thread can acquire the lock

Locks are associated with objects

- Every Object has its own built-in lock
  - Just the same, some applications prefer to create special classes of objects to use just for locking
  - This is a stylistic decision and you should agree on it with your teammates or learn the company policy if you work at a company
- Code is “thread safe” if it can handle multiple threads using it… otherwise it is “unsafe”
Working Example: “SummationJob”

```java
public class SummationJob implements Runnable {
    public static int X = 0;
    public static int NDONE = 0;
    public static final int NTHREADS = 2;

    /** Increments X 1000 times. */
    public void run() {
        for(int k=0; k<1000; k++) {
            X = X + 1; // (WARNING: MAIN RACE CONDITION)
        }
        NDONE += 1; // (WARNING: ANOTHER RACE CONDITION)
    }

    /** Launches NTHREADS SummationJob objects that try to increment X to NTHREADS*1000 */
    public static void main(String[] args) {
        try {
            Thread[] threads = new Thread[NTHREADS];
            for(int k=0; k<NTHREADS; k++)
                threads[k] = new Thread(new SummationJob());
            for(int k=0; k<NTHREADS; k++)
                threads[k].start();
            while(NDONE < NTHREADS)
                Thread.sleep(100);
            System.out.println("X="+X);
        } catch(Exception e) {
            e.printStackTrace();
            System.out.println("OOPS"+e);
        }
    }
}
```

Synchronization+priorities

- Combining mundane features can get you in trouble
- Java has priorities... and synchronization
  - But they don’t “mix” nicely
  - High-priority runs before low priority
  - ... The lower priority thread “starves”
- Even worse...
  - With many threads, you could have a second high priority thread stuck waiting on that starving low priority thread! Now both are starving...

Fancier forms of locking

- Java developers have created various synchronization ADTs
  - Semaphores: a kind of synchronized counter
  - Event-driven synchronization
- The Windows and Linux and Apple O/S all have kernel locking features, like file locking
- But for Java, synchronized is the core mechanism

Finer grained synchronization

- Java allows you to do fancier synchronization
  - But can only be used inside a synchronization block
  - Special primitives called wait/notify
    - In java.lang,Object

```
java.lang.Object

Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Modifier and Type</th>
<th>Method Description</th>
</tr>
</thead>
</table>
| wait()          | public             | Relinquishes lock on object.
| notify()        | public             | Notifies processes waiting for object lock.
| notifyAll()     | public             | Notifies all processes waiting for object lock.
```

Suppose we put this inside an object called animator:

```java
boolean isRunning = true;

public synchronized void run() {
    while (true) {
        // do one step of simulation
        try {
            wait();
        } catch (InterruptedException ie) {};
        isRunning = true;
    }
}

public synchronized void stopAnimation() {
    animator.isRunning = false;
}

public synchronized void restartAnimation() {
    animator.notify();
}
```

Constructor Summary

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructed with</td>
<td>java.lang.Object</td>
</tr>
</tbody>
</table>
Deadlock

- The downside of locking – deadlock
- A deadlock occurs when two or more competing threads are waiting for one-another... forever
- Example:
  - Thread t1 calls synchronized b inside synchronized a
  - But thread t2 calls synchronized a inside synchronized b
  - t1 waits for t2... and t2 waits for t1...

Deadlocks always involve cycles

- They can include 2 or more threads or processes in a waiting cycle
- Other properties:
  - The locks need to be mutually exclusive (no sharing of the objects being locked)
  - The application won’t give up and go away (no timer associated with the lock request)
  - There are no mechanisms for one thread to take locked resources away from another thread – no “preemption”

Higher level abstractions

- Locking is a very low-level way to deal with synchronization
  - Very nuts-and-bolts

- So many programmers work with higher level concepts. Sort of like ADTs for synchronization
  - We’ll just look at one example today
  - There are many others; take CS4410 “Operating Systems” to learn more

Visualizing deadlock

- A has a lock on X
  - wants a lock on Y
- Process A
- Process B
- B has a lock on Y
  - wants a lock on X

Dealing with deadlocks

- We recommend designing code to either
  - Acquire a lock, use it, then promptly release it, or
  - acquire locks in some “fixed” order

- Example, suppose that we have objects a, b, c, ...
- Now suppose that threads sometimes lock sets of objects but always do so in alphabetical order
  - Can a lock-wait cycle arise?
  - ... without cycles, no deadlocks can occur!

A producer/consumer example

- Thread A produces loaves of bread and puts them on a shelf with capacity K
  - For example, maybe K=10
- Thread B consumes the loaves by taking them off the shelf
  - Thread A doesn’t want to overload the shelf
  - Thread B doesn’t wait to leave with empty arms

... drop that mouse or you’ll be down to 8 lives
Producer/Consumer example

```java
class Bakery {
    int nLoaves = 0; // Current number of waiting loaves
    final int K = 10; // Shelf capacity
    public synchronized void produce() {
        while(nLoaves == K) this.wait(); // Wait until not full
        ++nLoaves;
        this.notifyAll(); // Signal: shelf not empty
    }
    public synchronized void consume() {
        while(nLoaves == 0) this.wait(); // Wait until not empty
        --nLoaves;
        this.notifyAll(); // Signal: shelf not full
    }
}
```

Things to notice

- Wait needs to wait on the same object that you used for synchronizing (in our example, “this”, which is this instance of the Bakery)
- Notify wakes up just one waiting thread, notifyAll wakes all of them up
- We used a while loop because we can’t predict exactly which thread will wake up “next”

Bounded Buffer example

```java
class BoundedBuffer<T> {
    int putPtr = 0, getPtr = 0; // Next slot to use
    int available = 0; // Items currently available
    final int K = 10; // buffer capacity
    T[] buffer = new T[K];
    public synchronized void produce(T item) {
        while(available == K) this.wait(); // Wait until not full
        buffer[putPtr++ % K] = item;
        ++available;
        this.notifyAll(); // Signal: not empty
    }
    public synchronized T consume() {
        while(available == 0) this.wait(); // Wait until not empty
        --available;
        T item = buffer[getPtr++ % K];
        this.notifyAll(); // Signal: not full
        return item;
    }
}
```

In an ideal world...

- Bounded buffer allows producer and consumer to both run concurrently, with neither blocking
  - This happens if they run at the same average rate
  - ... and if the buffer is big enough to mask any brief rate surges by either of the two
- But if one does get ahead of the other, it waits
  - This avoids the risk of producing so many items that we run out of computer memory for them. Or of accidentally trying to consume a non-existent item.

Bounded Buffer

- Here we take our producer/consumer and add a notion of passing something from the producer to the consumer
  - For example, producer generates strings
  - Consumer takes those and puts them into a file
- Question: why would we do this?
  - Keeps the computer more steadily busy
Trickier example

- Suppose we want to use locking in a BST
  - Goal: allow multiple threads to search the tree
  - But don't want an insertion to cause a search thread to throw an exception

```
class BST {
    Object name;      // Name of this node
    Object value;     // Value of associated with that name
    BST left, right;  // Children of this node

    // Constructor
    public BST(Object who, Object what) { name = who; value = what; }

    // Returns value if found, else null
    public Object get(Object goal) {
        if(name.equals(goal)) return value;
        if(name.compareTo(goal) < 0) return left==null? null: left.get(goal);
        return right==null? null: right.get(goal);
    }

    // Updates value if name is already in the tree, else adds new BST node
    public void put(Object goal, Object value) {
        if(name.equals(goal)) { this.value = value; return; }
        if(name.compareTo(goal) < 0) {
            if(left == null) { left = new BST(goal, value); return; }
            left.put(goal, value);
        } else {
            if(right == null) { right = new BST(goal, value); return; }
            right.put(goal, value);
        }
    }
}
```

Attempt #1

- Just make both put and get synchronized:
  - public synchronized Object get(...) { ... }
  - public synchronized void put(...) { ... }

- Let's have a look....

```
public synchronized Object get(...) {
    if(name.equals(goal)) return value;
    if(name.compareTo(goal) < 0) return left==null? null: left.get(goal);
    return right==null? null: right.get(goal);
}

public synchronized void put(...) {
    if(name.equals(goal)) { this.value = value; return; }
    if(name.compareTo(goal) < 0) {
        if(left == null) { left = new BST(goal, value); return; }
        left.put(goal, value);
    } else {
        if(right == null) { right = new BST(goal, value); return; }
        right.put(goal, value);
    }
}
```

Safe version: Attempt #1

```
class BST {
    Object name;      // Name of this node
    Object value;     // Value of associated with that name
    BST left, right;  // Children of this node

    // Constructor
    public BST(Object who, Object what) { name = who; value = what; }

    // Returns value if found, else null
    public synchronized Object get(Object goal) {
        if(name.equals(goal)) return value;
        if(name.compareTo(goal) < 0) return left==null? null: left.get(goal);
        return right==null? null: right.get(goal);
    }

    // Updates value if name is already in the tree, else adds new BST node
    public synchronized void put(Object goal, Object value) {
        if(name.equals(goal)) { this.value = value; return; }
        if(name.compareTo(goal) < 0) {
            if(left == null) { left = new BST(goal, value); return; }
            left.put(goal, value);
        } else {
            if(right == null) { right = new BST(goal, value); return; }
            right.put(goal, value);
        }
    }
}
```

Visualizing attempt #1

- put(Ernie, eb0)  
- get(Martin).... must wait!
- get(Martin).... resumes

```
public synchronized Object get(Object goal) {
    if(name.equals(goal)) return value;
    if(name.compareTo(goal) < 0) return left==null? null: left.get(goal);
    return right==null? null: right.get(goal);
}

public synchronized void put(Object goal, Object value) {
    if(name.equals(goal)) { this.value = value; return; }
    if(name.compareTo(goal) < 0) {
        if(left == null) { left = new BST(goal, value); return; }
        left.put(goal, value);
    } else {
        if(right == null) { right = new BST(goal, value); return; }
        right.put(goal, value);
    }
}
```
11/25/14

Attempt #2: Improving “get”

- **put** uses synchronized in method declaration
- So it locks every node it visits
- **get** tries to be fancy:
  ```java
  // Returns value if found, else null
  public Object get(Object goal) {
    synchronized(this) {
      if(name.equals(goal)) return value;
      if(name.compareTo(goal) < 0) return left==null? null: left.get(goal);
      return right==null? null: right.get(goal);
    }
  }
  ```
- Actually this is identical to attempt 1!
  - Looks different but does exactly the same thing
  - Still locks during recursive tree traversal

Attempt #3: An improved “get”

- **public synchronized Object get(Object goal)**
- Locks node when accessing fields, but not during subsequent traversal
  ```java
  // Returns value if found, else null
  public Object get(Object goal) {
    BST checkLeft = null, checkRight = null;
    synchronized(this) {
      if(name.equals(goal)) return value;
      if(name.compareTo(goal) < 0) {
        if(left==null) return null; else checkLeft = left;
      } else {
        if(right==null) return null; else checkRight = right;
      }
    }
    if (checkLeft  != null) return checkleft.getValue(goal);
    if (checkRight != null) return checkright.getValue(goal);
    // Never executed but keeps Java happy */
    return null;
  }
  ```

More tricky things to know about

- With thread priorities Java can be very annoying
  - ALWAYS runs higher priority threads before lower priority threads if scheduler must pick
  - The lower priority ones might never run at all

- Consequence: risk of a “priority inversion”
  - High-priority thread t1 is waiting for a lock, t2 has it
  - Thread t2 is runnable, but never gets scheduled because t3 is higher priority and “busy”

Teaser: Threads super important for GPUs


Summary

- Use of multiple processes and multiple threads within each process can exploit concurrency
  - Which may be real (multicore) or “virtual” (an illusion)
  - But when using threads, beware!
  - A “race condition” can arise if two threads try and share data
  - Must lock (synchronize) any shared memory to avoid non-determinism and race conditions
  - Yet synchronization also creates risk of deadlocks
  - Even with proper locking concurrent programs can have other problems such as “livelock”
  - Nice tutorial at http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

- Serious treatment of concurrency is a complex topic (covered in more detail in cs3410 “systems” and cs4410 “OS”)