RACE CONDITIONS AND SYNCHRONIZATION

Lecture 21 – CS2110 – Fall 2009

Reminder

2

- A “race condition” arises if two threads try and share some data
- One updates it and the other reads it, or both update the data
- In such cases it is possible that we could see the data “in the middle” of being updated
  - A “race condition”: correctness depends on the update racing to completion without the reader managing to glimpse the in-progress update
  - Synchronization (aka mutual exclusion) solves this

Java Synchronization (Locking)

3

```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
   synchronized (stack) {
      if (stack.isEmpty()) return;
      String s = stack.pop();
   }
   //do something with s...
}
```

`synchronized` block

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

Java Synchronization (Locking)

4

- You can lock on any object, including **`this`**

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

is equivalent to

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

How locking works

5

- 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

6

- 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”

File Locking: Same idea

7

- In file systems, if two or more processes could modify a file simultaneously, this could result in data corruption
  - A process must *open* a file to modify it – gives exclusive access until it is *closed*
  - Multiple processes can open the same file to read it
- This *file locking* synchronization rule is enforced by the operating system

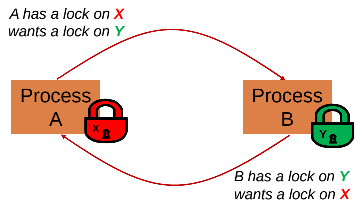
Deadlock

8

- The downside of locking – *deadlock*
- A *deadlock* occurs when two or more competing threads each hold a lock, and each are waiting for the other to relinquish a lock, so neither ever does
- Example:
  - thread A tries to open file X, then file Y
  - thread B tries to open file Y, then file X
  - A gets X, B gets Y
  - Each is waiting for the other forever

Visualizing deadlock

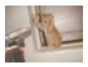
9



Deadlocks always involve cycles

10

- 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"



`wait/notify`

11

- A mechanism for event-driven activation of threads
- Animation threads and the **`GUI`** event-dispatching thread in can interact via **`wait/notify`**

`wait/notify`

12

`animator:`

```
boolean isRunning = true;

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

relinquishes lock on **`animator`** – awaits notification

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

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

notifies processes waiting for **`animator`** lock

A producer/consumer example

13

- 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

producer

shelves

consumer

Producer/Consumer example

14

```
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

15

- 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"

Trickier example

16

- 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

Code we're given is unsafe

```
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 void 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

18

- Just make both put and get synchronized:
  - public synchronized Object get(…) { … }
  - public synchronized void put(…) { … }
- Let's have a look….

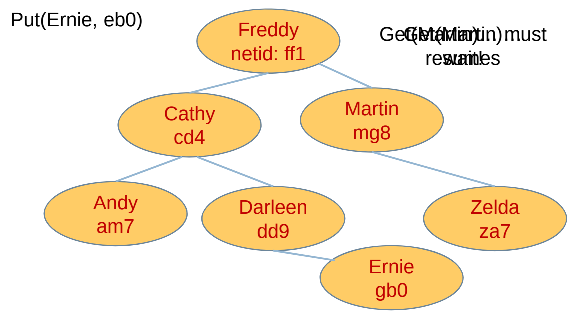
Safe version: Attempt #1

```
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 void 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);
        }
    }
}
```

Attempt #1

- Just make both put and get synchronized:
  - public synchronized Object get(…) { … }
  - public synchronized void put(…) { … }
- This works but it kills ALL concurrency
  - Only one thread can look at the tree at a time
  - Even if all the threads were doing “get”!

Visualizing attempt #1



Attempt #2

- put uses synchronized in method declaration
  - So it locks every node it visits
- get tries to be fancy:

```
    // 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! It only looks different but in fact is doing exactly the same thing

Attempt #3

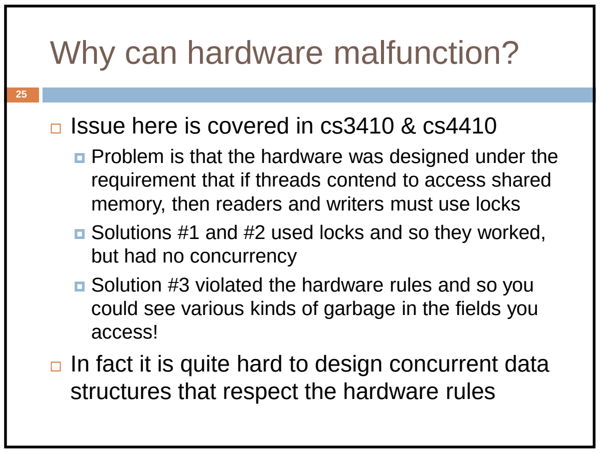
```
    // Returns value if found, else null
    public Object get(Object goal) {
        boolean checkLeft = false, checkRight = false;
        synchronized(this) {
          if(name.equals(goal)) return value;
          if(name.compareTo(goal) < 0) {
                  if (left==null) return null; else checkLeft = true;
          } else {
                  if                                          true;
          }
        }
        if (checkLeft) return left.get(goal);
        if (checkRight) return right.get(goal);

        /* Never executed but keeps Java happy */ return null;
    }
```

relinquishes lock on **this** – next lines are “unprotected”

- Risk: “get” (read-only) threads sometimes look at nodes without locks, but “put” always updates those same nodes.
- According to JDK rules this is unsafe

Attempt #3 illustrates risks

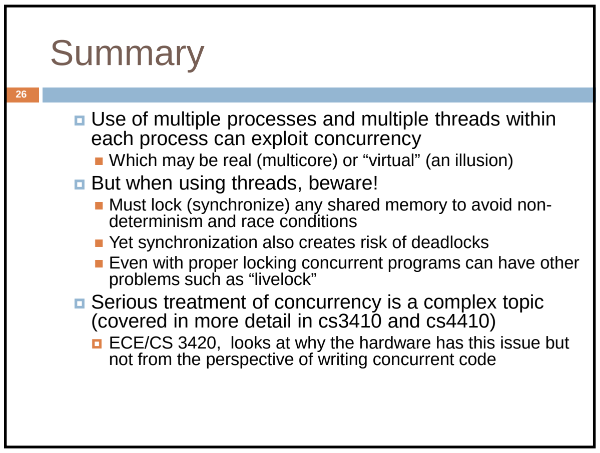
- The hardware itself actually needs us to use locking and attempt 3, although it looks right in Java, could actually malfunction in various ways
  - Issue: put updates several fields:
    - parent.left (or parent.right) for its parent node
    - this.left and this.right and this.name and this.value
  - When locking is used correctly, multicore hardware will correctly implement the updates
  - But if you look at values without locking, as we did in Attempt #3, hardware can malfunction!



Why can hardware malfunction?

25

- Issue here is covered in cs3410 & cs4410
  - Problem is that the hardware was designed under the requirement that if threads contend to access shared memory, then readers and writers must use locks
  - Solutions #1 and #2 used locks and so they worked, but had no concurrency
  - Solution #3 violated the hardware rules and so you could see various kinds of garbage in the fields you access!
- In fact it is quite hard to design concurrent data structures that respect the hardware rules



Summary

26

- 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!
  - 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"
- Serious treatment of concurrency is a complex topic (covered in more detail in cs3410 and cs4410)
  - ECE/CS 3420, looks at why the hardware has this issue but not from the perspective of writing concurrent code