



RACE CONDITIONS AND SYNCHRONIZATION

Lecture 21 – CS2110 – Fall 2013

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

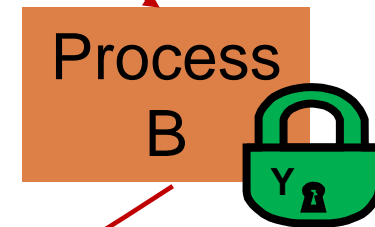
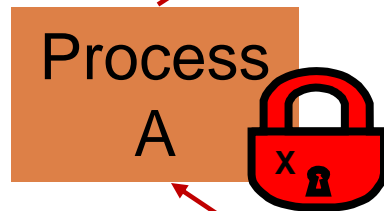
6

- 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”

Visualizing deadlock

7

*A has a lock on **X**
wants a lock on **Y***



*B has a lock on **Y**
wants a lock on **X***

Deadlocks always involve cycles

8

- 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”

“... drop that mouse or you'll be down to 8 lives”



Dealing with deadlocks

9

- 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!

Higher level abstractions

10

- 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 to learn more

A producer/consumer example

11

- 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

12

```
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

13

- 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

14

- 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

Producer/Consumer example

15

```
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
    }
}
```

Bounded Buffer example

16

```
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...

17

- 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.

Trickier example

18

- 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

20

- 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

22

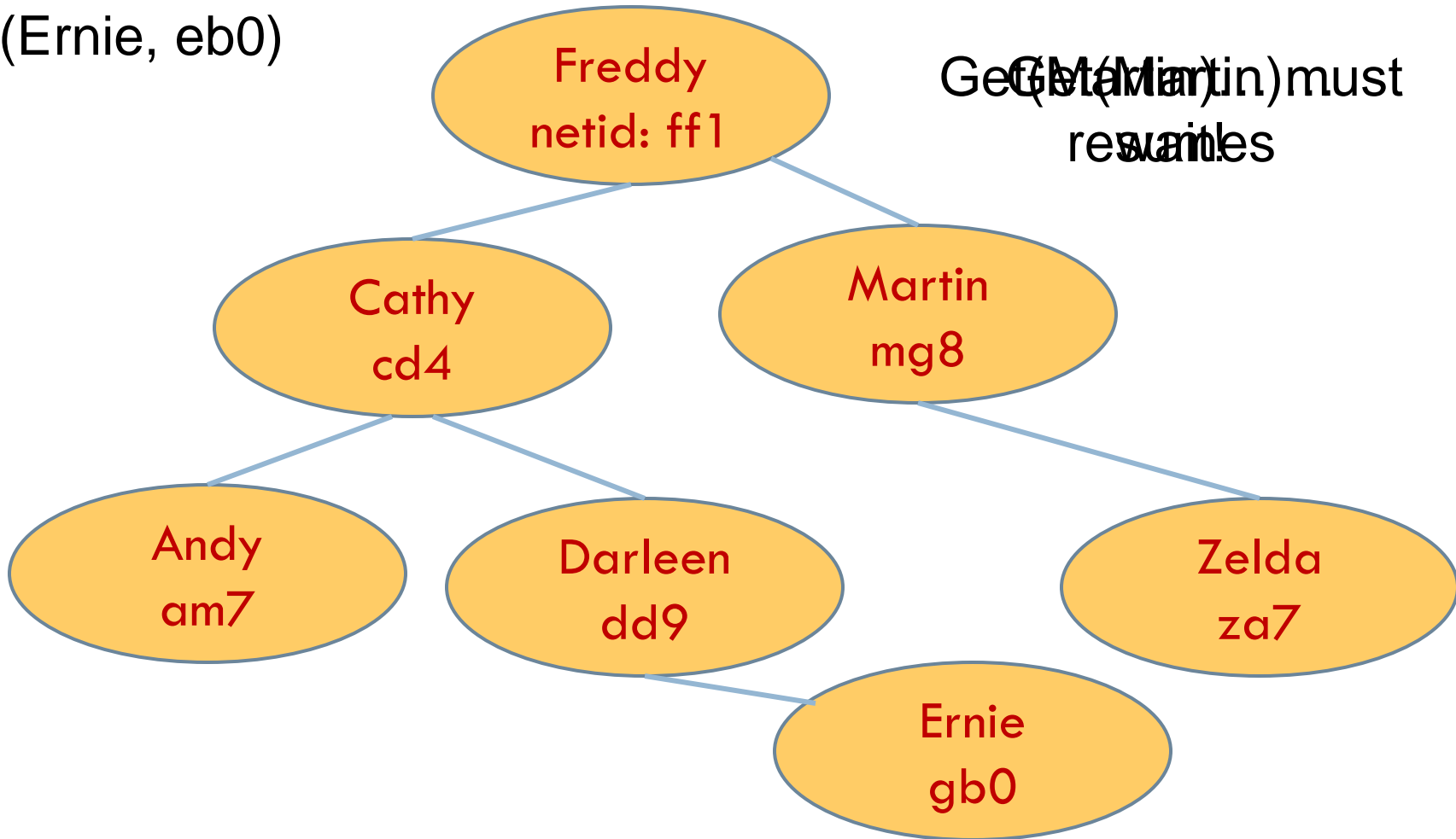
- Just make both put and get synchronized:
 - ▣ `public synchronized Object get(...) { ... }`
 - ▣ `public synchronized void put(...) { ... }`
- This works but it kills ALL concurrency
 - ▣ Only one thread can look at the tree at a time
 - ▣ Even if all the threads were doing “get”!

Visualizing attempt #1

23

Put(Ernie, eb0)

Get(Martin) must
return



Attempt #2

24

- 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);
    }
}
```

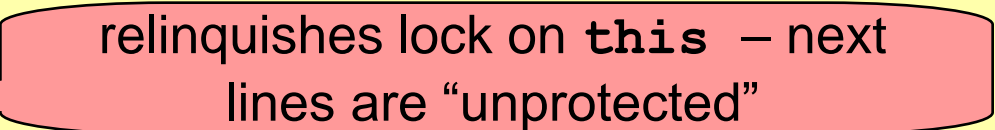
- Actually this is identical to attempt 1! It only looks different but in fact is doing exactly the same thing

Attempt #3

25

```
// 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 (right==null) return null; else checkRight = true;
        }
    }
    if (checkLeft) return left.get(goal);
    if (checkRight) return right.get(goal);

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



- 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 #4

26

```
// 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.get(goal);
    if (checkRight != null) return checkright.get(goal);

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

- This version is safe: only accesses the shared variables left and right while holding locks
- In fact it should work (I think)

Attempt #3 illustrates risks

27

- 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 behave oddly!

Why can hardware cause bugs?

28

- 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!
 - Solution #4 should be correct, but perhaps not optimally concurrent (doesn't allow concurrency while even one "put" is active)
- It's hard to design concurrent data structures!

More tricky things to know about

29

- Java has actual “lock” objects
 - ▣ They support lock/unlock operations
- But it isn't easy to use them correctly
 - ▣ Always need a try/finally structure

```
Lock someLock = new Lock();

try {
    someLock.lock();
    do-stuff-that-needs-a-lock();
}
finally {
    someLock.unlock();
}
```

More tricky things to know about

30

□ Needs try/finally

```
Lock someLock = new Lock();

try {
    someLock.lock();
    do-stuff-that-needs-a-lock();
}
finally {
    someLock.unlock();
}
```

- **Complication:** `someLock.unlock()` can only be called by same thread that called `lock`.
- **Advanced issue:** *If your code catches exceptions and the thread that called `lock()` might terminate, the lock can't be released! It remains locked forever... bad news...*

Semaphores

31

- Yet another option, mentioned Tuesday
 - ▣ But avoids this issue seen with locks

- A Semaphore has an associated counter
 - ▣ When created you specify an initial value
 - ▣ Then each time the Semaphore is acquired the counter counts down. And each time the Semaphore is released, it counts up.
 - ▣ If zero, `s.acquire()` waits for a release

More tricky things to know about

32

- With 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”

Debugging concurrent code

33

- There are Eclipse features to help you debug concurrent code that uses locking
 - ▣ These include packages to detect race conditions or non-deterministic code paths
 - ▣ Packages that will track locks in use and print nice summaries if needed
 - ▣ Packages for analyzing performance issues
 - Heavy locking can kill performance on multicore machines
 - Basically, any sharing between threads on different cores is a performance disaster

Summary

34

- ▣ 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)
- ▣ Nice tutorial at <http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>