WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Lecture 7

- First mid-term: Monday, Feb 21
- Assignment on Construction, Publishing and Visibility is posted (and we will discuss momentarily)

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Dekker's Algorithm Properties – mutual exclusion

```
1.  boolean enter1 = false;
2.  boolean enter2 = false;
3.  int turn = 1;
4.  { while(true) { /* Thread 1 */
5.      enter1 = true;
6.      while (enter2) {
7.          if (turn==2) {
8.              enter1 = false;
9.              while (turn==2) skip;
10.             enter1 = true;
11.         }
12.     }
13.     /* critical section */
14.     enter1 = false; turn = 2;
15.     /* non-critical section */
16. }}
```

```
/* Thread 2 */
{ while(true) {
    enter2 = true;
    while (enter1) {
        if (turn==1) {
            enter2 = false;
            while (turn==1) skip;
            enter2 = true;
        }
    }
    /* critical section */
    enter2 = false; turn = 1;
    /* non-critical section */
}}
```

2

Argument: at point where T1 enters its CS enter2 is false and enter1 is true.

Enter1 remains true while T1 is in the CS

Enter2 may become true while T1 is in the CS but T2 cannot enter its CS until enter1 becomes false.

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Dekker's Algorithm Properties – eventual entry

```
boolean enter1 = false;
boolean enter2 = false;
int turn = 1;
{ while(true) { /* Thread 1 */
  enter1 = true;
  while (enter2) {
    if (turn==2) {
      enter1 = false;
      while (turn==2) skip;
      enter1 = true;
    }
  }
  /* critical section */
  enter1 = false; turn = 2;
  /* non-critical section */
}}
```

```
/* Thread 2 */
{ while(true) {
  enter2 = true;
  while (enter1) {
    if (turn==1) {
      enter2 = false;
      while (turn==1) skip;
      enter2 = true;
    }
  }
  /* critical section */
  enter2 = false; turn = 1;
  /* non-critical section */
}}
```

3

Suppose T2 wants to enter but can't; then it's in the while(enter1) loop.

If turn==2 (it was set by T1 at the end of its CS) T1 won't change it back; also enter2 is continuously true.

Claim T1 cannot re-enter its CS. Furthermore T1 will set enter1 to false and not set it back to true as long as turn==2.

Thus, if T1 makes any progress at all eventually enter1 becomes false.

Similarly if T2 makes any progress at all, it detects enter1==false and enters the CS.

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Dekker's Algorithm Properties – non blocking

```
boolean enter1 = false;
boolean enter2 = false;
int turn = 1;
{ while(true) { /* Thread 1 */
  enter1 = true;
  while (enter2) {
    if (turn==2) {
      enter1 = false;
      while (turn==2) skip;
      enter1 = true;
    }
  }
  /* critical section */
  enter1 = false; turn = 2;
  /* non-critical section */
}}
```

```
/* Thread 2 */
{ while(true) {
  enter2 = true;
  while (enter1) {
    if (turn==1) {
      enter2 = false;
      while (turn==1) skip;
      enter2 = true;
    }
  }
  /* critical section */
  enter2 = false; turn = 1;
  /* non-critical section */
}}
```

4

This is the easy case when one thread wants to enter and the other doesn't. Suppose it's T1 that wants to enter.

Then enter2 is false so T1 enters the CS after testing enter2 exactly once.

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Dekker's Algorithm Properties – no deadlock

```
boolean enter1 = false;
boolean enter2 = false;
int turn = 1;
{ while(true) { /* Thread 1 */
  enter1 = true;
  while (enter2) {
    if (turn==2) {
      enter1 = false;
      while (turn==2) skip;
      enter1 = true;
    }
  }
  /* critical section */
  enter1 = false; turn = 2;
  /* non-critical section */
}}
```

```
/* Thread 2 */
{ while(true) {
  enter2 = true;
  while (enter1) {
    if (turn==1) {
      enter2 = false;
      while (turn==1) skip;
      enter2 = true;
    }
  }
  /* critical section */
  enter2 = false; turn = 1;
  /* non-critical section */
}}
```

5

Eventual entry implies no deadlock.

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Assignment: Construction, Visibility, and Publication

- Inspect real code for problems
- MARS MIPS simulator system
- Example

Example

```
class SimThread extends
  SwingWorker {
  private MIPSprogram p;
  private int pc, maxSteps;
  private int[] breakPoints;
  private boolean done;
  private ProcessingException
    pe;
  private volatile boolean
    stop = false;
  private volatile
    AbstractAction stopper;
  private AbstractAction
    starter;
  private int
    constructReturnReason;
```

At line 259 – in a SimThread

```
this.pe = new
  ProcessingException(el);
this.constructReturnReason =
  EXCEPTION;
this.done = true;
```

And at line 122 – in process that creates the SimThread

```
ProcessingException pe =
  simulatorThread.pe;
boolean done =
  simulatorThread.done;
```

Also, note the comments and code at lines 235-240. What's wrong with this?

Volatiles stop and stopper: both assigned to in setStop

this.done and this.pe are not volatile and yet are accessed from a different thread

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Advice

- Some uses in this code are suspicious but you can't really tell whether they are wrong without looking at other modules. For things that catch your eye as suspicious write down your suspicions and the questions they raise about other modules.
- It's a lot of code. What language constructs do you need to focus on?
  -
  -
  -
- When doing the homework, notice what makes it hard to determine whether the code is right or wrong
  - Learn from example code – both good and bad

Constructores – visibility and escape

Synchronized methods (and lack thereof)

Variables – accessed by multiple threads?

Variables – accessed in ways that suggest an implicit invariant?

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Two common problems

- Composing a thread-safe class from unsafe building blocks
- Composing a thread-safe class when the building blocks are already thread-safe

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Confinement Technique (4.2) – thread-safe object built from unsafe objects

- Allow access to a thread-unsafe object only through another object that is thread-safe

```
public class PersonSet {
   private final Set<Person> mySet = new
  HashSet<Person>();
   public synchronized void add(Person p) {
      mySet.add(p); }
   public synchronized boolean contains(Person p) {
      return myset.contains(p); }
}
```

- hashSet is not ThreadSafe, PersonSet is
- Idea of ownership: PersonSet owns mySet but probably not the Persons contained in it

mySet confined to PersonSet

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Monitors (4.2.1) (aside)

- Idea: a syntactic construct that pairs lock acquisition and release around a block of code
  - synchronized(foo) { … }
  - compared to explicit lock acquisition and release, encourages structured programs and avoids bugs due to failure to release locks
- Along with a mechanism for releasing a lock, suspending, and reacquiring the lock
  - wait()
- And a mechanism to wake up suspended threads
  - notify(), notifyAll()
- Monitors were independently invented by Sir Anthony Hoare and Per Brinch-Hansen in the mid-1970's.

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Danger in Confinement Technique

- Inadvertant publication of what is supposed to be private (confined) mutable state

```
public synchronized MutablePoint getLocation(String id) {
   MutablePoint loc = locations.get(id);
   return loc == null ? Null : new MutablePoint(loc);
}
Public synchronized setLocation(String id, int x, int y) {
   MutablePoint loc = locations.get(id);
   if (loc == null) { … exception …}
   loc.x = x; loc.y = y
}
```

- My preference would be to express this interface using ImmutablePoints.

Listing 4.4 Page 63

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Thread-safe objects built from thread-safe components – Delegating safety (4.3)

- Delegation: giving responsibility for thread safety to the object(s) containing this object's state
  - ConcurrentMap (TS) instead of Map (not TS)
  - Atomic<foo>
- If this object's state involves multiple other objects delegation may or may not work
  - If the sub-objects are independent, ok
  - If the sub-objects are related, this object must provide its own synchronization – even if all the sub-objects are themselves thread-safe

Example

```
class PongPaddle {
   private final AtomicInteger left = new AtomicInteger(0);
   private final AtomicInteger right = new AtomicInteger(1);
   public void move(int dx) { left.getAndAdd(dx);
  right.getAndAdd(dx); }
   public void changeWidth(int dw) { right.getAndAdd(dw); }
   public boolean hit(int pos) {
      return left.get()<=pos && pos <= right.get();
   }
}
```

- No visibility concerns
- What is the invariant that relates left and right?
- What should we do to fix it?

Reduce, Reuse, Recycle

- I am very *anti* cut-and-paste coding
  - Hard on the reader
  - Hard on the maintainer
  - Instead of 1 change, n changes
  - Instead of 1 bug, n bugs
- Design code so there only needs to be one copy (use parameterization, polymorphic parameterization)
- Even better, reuse existing code that does almost the right thing
- How does this interact with synchronization?

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Adding functionality (4.4)

- Example: add putIfAbsent to a collection that already supports atomic contains() and add() methods
- Four approaches
  - Modify existing class
  - Extend existing class or
  - Wrap existing class – "client-side" locking
  - Composition -
- Add to existing class
  - Best way but
  - Assumes you have control over the existing class

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Extend the existing class

```
public class BetterVector<E> extends Vector<E> {
   public synchronized boolean putIfAbsent(E x) {
      boolean absent = !contains(x);
      if (absent) { add(x); }
      return absent;
   }
}
```

- Note the benefit of re-entrant locks!
- Note the implicit assumption that we understand the way that Vector does synchronization – using intrinsic locks, in this case
- Note that we don't have any dependency on Vector's implementation
- Vector is thread-safe
- Vector provides enough primitive building blocks to allow construction of putIfAbsent

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Client-side locking

- Assume v is a thread-safe list obtained from `Collections.synchronizedList(new ArrayList<E>());`
- Type of this object is List<E> -- not extendable
- Any code that wants to do putIfAbsent item x to such a list, v, can write

```
synchronized (v) {
   if (!v.contains(x)) v.add(x);
}
```

- Could be placed in a helper class – beware you have to synchronize on the *list* and not on the helper object
- Now spreading the synchronization far and wide
- Still depending on knowing the synch policy for the wrapped object

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Composition

- Mimic the idea of Collections.synchronizedList
  - Provide all the synchronization in a new object that extends the functionality of an existing object instance (not class)
  - Delegates most operations to the existing object

```
Public class ImprovedList<T> implements List<T> {
   private final List<T> list;
   public ImprovedList(List<T> list) {
      this.list = list; }
   public synchronized boolean putIfAbsent(T x) …
   public synchronized boolean contains(T x) {
      return list.contains(x); }
   …
```

Contains: example of delegation of functionality but not synchronization.

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Intro to Chapter 5 – Building Blocks

- Chapter 4 was about low-level techniques
- This chapter is about libraries – embodiments of the techniques
- Section 5.1 Synchronized collections – read to see why you want to use Concurrent collections instead (we will not cover)
  - The idioms described are even more unsafe than asserted in the book because of visibility problems

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Chapter 5 topics

- Concurrent collections (5.2)
- The ubiquitous producer-consumer pattern (5.3)
- Interruptable methods (5.4)
- Primitive synchronizers (5.5)