WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

# Videostreams of Lectures

- http://ams.wsu.edu/Videostreaming/ArchiveStreams.aspx?sem=Spring&y=2010&class=CPTS580
- **The first few are not there – they start with 1/25/10.**
- **Existence of the videostreams is not a reason to skip class**

1

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

# Dekker's Algorithm Properties – mutual exclusion

```
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 */
}}
```

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.

## Two key points from last time

- Proper Construction – this reference does not escape (to another thread) during construction
  - Remember – an object is not fully constructed until its constructor *returns*
- Thread-confined objects are safe
  - References are only on stack or in ThreadLocal objects
  - Ad-hoc thread-confinement – only one thread accesses an object by agreement (but no language enforcement)

6

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

# Immutable objects are thread-safe

- Object is immutable if
  - It's state cannot be modified after construction
    - "Cannot be modified" vs "is not modified"
  - Its data fields are declared final: cannot be changed after construction (must be declared final and not just treated as final)
  - It's *properly constructed*
- Volatile references + immutable objects => simple thread safety w/o locks
  - Beware inadvertant publishing of private values as in Fig. 3.6

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

# Example: atomic update using immutable object

```
Class OVCache {
  private final BigInteger last;
  private final BigInteger[] lastFactors;
  public OVCache( … i, … factors) {
        last = i;
        lastFactors = Arrays.copyOf(factors…);
  }
  in using code
  private volatile OVCache cache = new OVCache(null,null);
  …
  cache = new OVCache(i, factors);
```

Listings 3.12 and 3.13

8

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

## Unsafe Publication

```
public Holder holder;
public void init() { holder = new Holder(42); }
public class Holder {
  private int n;
  public Holder(int n) { this.n = n; }
  public void sanityCheck() { if (n != n) … }
}
```

**Holder is properly constructed but not properly published.**
**sanityCheck() can see two different values for holder.n!**

9

Publishing the new(Holder(42)).

Making the private field n final makes the Holder class immune to improper publication – because a special rule of the JVM memory model says that an immutable object is safely published even if no synchronization is used. (And this is why final is essential for an object to be considered immutable.)

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

# Safe Publication

- Recall: publication is the action that makes an object visible outside the current scope – typically an assignment of the reference
- Safe publication is all about making sure that the reference and the properly initialized fields in the object that it refers to become visible at the same time

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

# Safe publication idioms

- Initializing an object reference in a static initializer
  - i.e., static class-level data fields
- Assigning to a volatile field or AtomicReference object
- Assigning to a final field of a properly constructed object
- Assigning to a field that is properly guarded by a lock

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

# Safe Publication Summary

- Immutable objects can be published by any mechanism
- Effectively immutable objects (ones that in fact are not changed after construction though they do not meet the strict test) must be safely published – but then can be accessed without further synchronization
- Mutable thread-safe objects must be safely published (necessary synchronization is invoked internally by the object)
- Mutable thread-unsafe objects must be safely published then accessed using proper synchronization

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

# Chapter 3 Summary

- Visibility of changes is the main issue
- Correct synchronization, proper construction and safe publication are three requirements for programs to have well-defined behavior
- Rules are simpler for immutable objects
- There are no concurrency concerns for thread-confined objects, mutable or immutable

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

# Chapter 4

- **50+ years of CS and SE devoted to issues of composability: how to create solutions to big problems by combining (composing) solutions to smaller problems**
  - **Mechanisms – e.g. intrinsic locks; GC**
  - **Techniques – Invariants, pre- and post-conditions; confinement; delegation; etc. (Ch4)**
  - **Libraries – working code embodying the techniques for specific domains (Ch 5)**
- **This chapter mainly about techniques**

14

Chapter 4: Composing Objects

Mechanisms – built in to languages, OS, etc.

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

## Invariants – the fundamental technique

- What property is always true of an object when it is in a correct state?
- For sequential programming, the class invariant gives you critical information about what each public method needs to achieve
- For concurrent programming, the class invariant tells you which fields are related and therefore have to be protected by a single lock

15

Invariants again: essential for both sequential and concurrent programming.

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

# Post-conditions and Pre-conditions

- The post-condition of a method tells you what that method is supposed to accomplish
  - (A no-op will preserve the invariant but that's not very interesting or useful!)
- The pre-condition tells you what (beyond the invariant) is supposed to be true for a method to successfully reach the post-condition
  - In sequential programming calling a method when its precondition is false is an error
  - In concurrent programming we can wait for the precondition to become true

16