

50.003: Elements of Software Construction

Concurrency: Testing

Course Plan

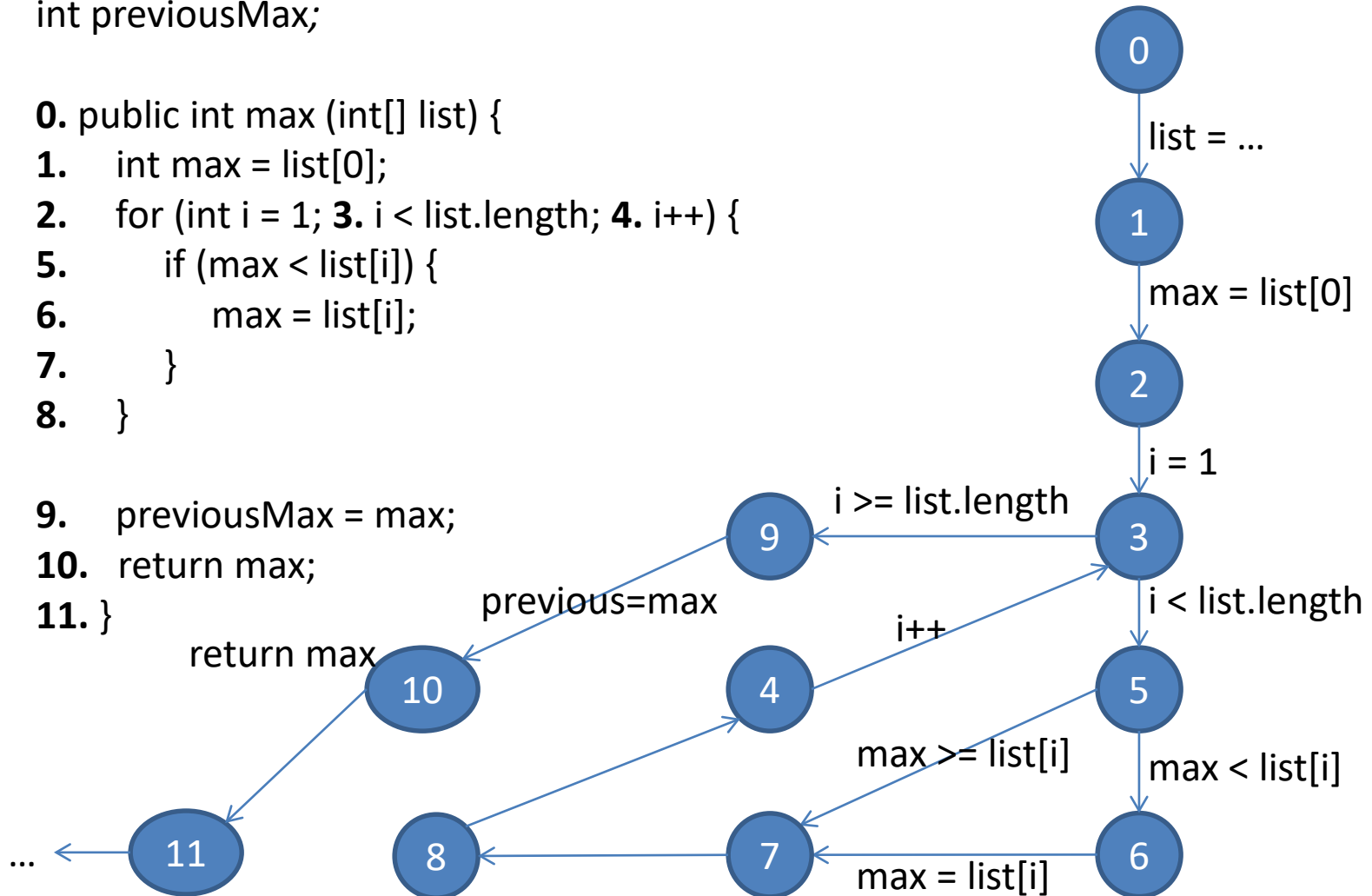
Week	Cohort Class 1	Cohort Class 2	Cohort Class 3	Remarks
1 (Jan 22)	Software Development Process			
2 (Jan 29)	Software Design and UML		Project Meeting I	Problem Set 1
3 (Feb 5)	Software Design and UML		Guest Lectures	
4 (Feb 12)	Design Patterns			Problem Set 2. Quiz 1
5 (Feb 19)	Software Testing		Project Meeting II	
6 (Feb 26)	Software Testing			Problem Set 3
7 (Mar 5)	Recess			Problem Set 4
8 (Mar 12)	Software Debugging, Code smells and Maintenance		Project Meeting III	Quiz 2
9 (Mar 19)	Concurrency: Requirement			Problem Set 5
10 (Mar 26)	Concurrency: Design and Implementation			Problem Set 6; Quiz 3
11 (Apr 2)	Concurrency: Testing		Project Meeting IV	Problem Set 7
12 (Apr 9)	Concurrency: Optimization			Problem Set 8; Quiz 4
13 (Apr 16)	Final Project Presentation (15 minutes for each group)			Project Report/Code Due
14 (Apr 27)	Final Exam			

Testing Sequential Program

int previousMax;

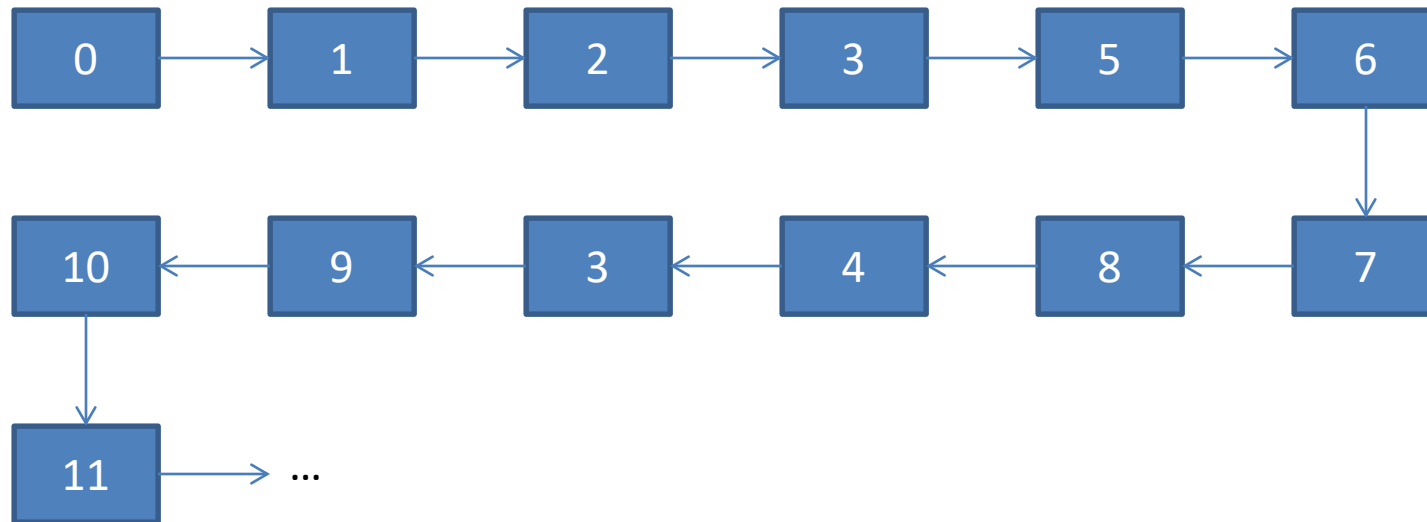
```
0. public int max (int[] list) {  
1.   int max = list[0];  
2.   for (int i = 1; 3. i < list.length; 4. i++) {  
5.     if (max < list[i]) {  
6.       max = list[i];  
7.     }  
8.   }
```


```
9.   previousMax = max;  
10.  return max;  
11. }
```



Test Execution

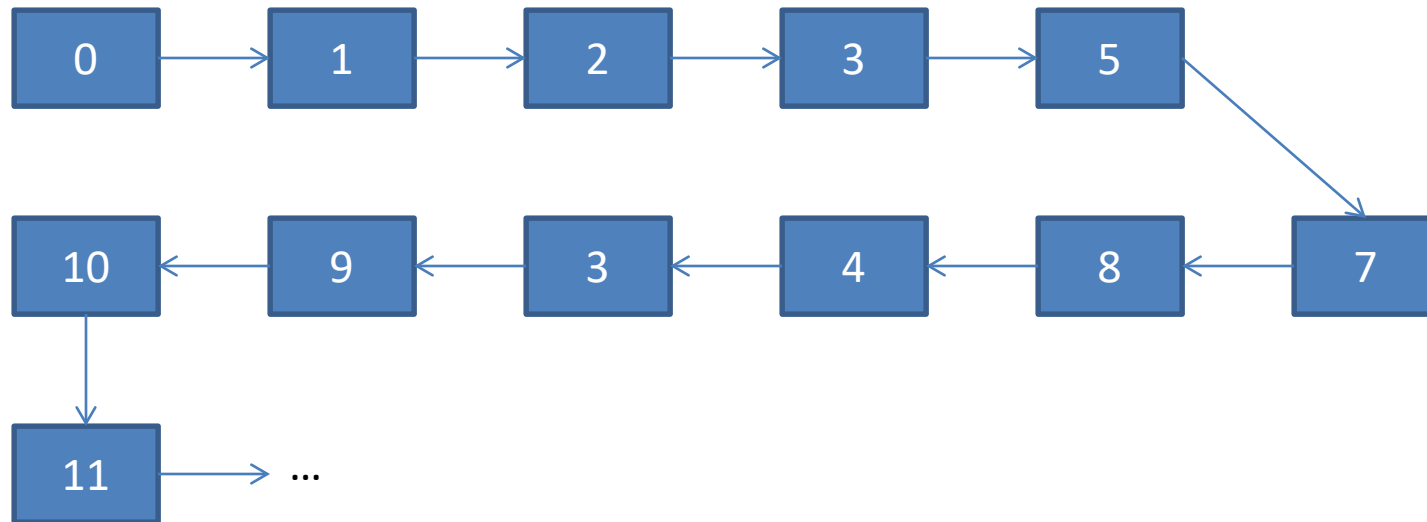
- With input = [2,4]




 : a configuration of the program with control at line i

Test Execution

- With input = [4,2]



 : a configuration of the program with control at line i

Testing

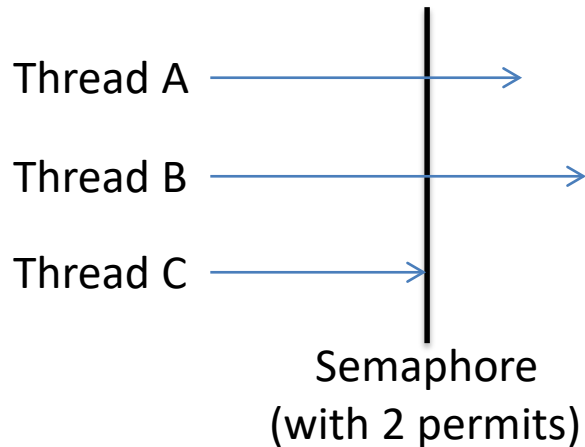
- For sequential programs,
 - Finding the right inputs
- For concurrent programs,
 - Finding the right inputs and scheduling
 - To be able to generate more scheduling, we could use `Thread.sleep()`, and synchronizers.

Synchronizers

- A synchronizer is an object that coordinates the control flow of threads based on its state.
 - Semaphore
 - CyclicBarrier
 - CountdownLatch
 - Phaser

Semaphores

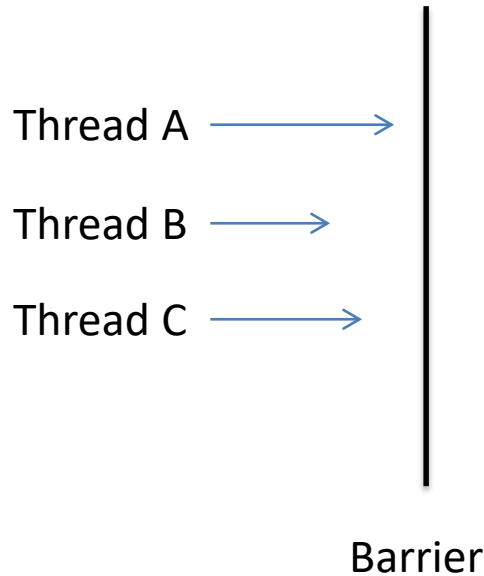
A semaphore maintains a set of permits. Each `acquire()` blocks if necessary until a permit is available, and then takes it. Each `release()` adds a permit, potentially releasing a blocked acquirer.



Example: `SemaphoreExample.java`

Cyclic Barriers

A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. The barrier is often called cyclic because it can be re-used after the waiting threads are released.



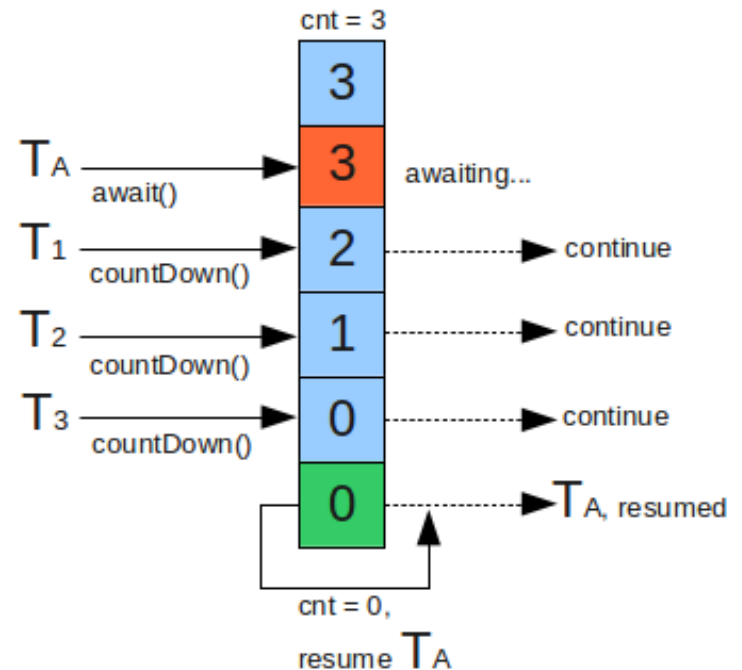
[Click here for a sample program: BarrierExample.java](#)

Cohort Exercise 1

- Given `MyCyclicBarrier.java`, complete method `await()` such that you can replace `CyclicBarrier` in `BarrierExample.java` with `MyCyclicBarrier` without changing its behaviour.

CountDownLatch

A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.



[Click here for a sample program: CountDownLatchExample.java](#)

Cohort Exercise 2

- Given an (large) array of strings (of grades), write a multi-threaded program, using `CountDownLatch`, to check whether the array contains 7 “F”. Stop all threads as soon as possible.

Phaser

- Phaser (introduced in Java 7)
 - A reusable synchronization barrier, similar in functionality to `CyclicBarrier` and `CountDownLatch` but supporting more flexible usage.

Sample program: `PhaserExample.java`

Cohort Exercise 3

Draw the state machine diagram for a Phaser object.

- A state should be identified by three numbers: the phase number, the number of registrations, the number of arrivals. For simplicity, limit the numbers to maximum 2.
- The transitions should be labelled with methods in the class.

Barrier vs Latch vs Phaser

CountDownLatch

- Created with a fixed number of threads
- Cannot be reset
- Allow threads to wait (method `await`) or continue with its execution (method `countdown()`)

Cyclic Barrier

- Can be reset.
- Does not provide a method for the threads to advance. The threads have to wait till all the threads arrive.
- Created with fixed number of threads.

Phaser

- Number of threads need not be known at Phaser creation time. They can be added dynamically.
- Can be reset and hence is, reusable.
- Allows threads to wait (method `arriveAndAwaitAdvance()`) or continue with its execution (method `arrive()`).
- Supports multiple Phases.

Testing for Concurrency

- Testing for correctness
 - Safety: nothing bad ever happens
 - Liveness: something good eventually happens (e.g., no deadlock)
- Testing for performance
 - Throughput: the rate at which a set of concurrent tasks is completed
 - Responsiveness: the delay between a request and completion of some action

Step 1: Identifying Specification

- You must know what is correct.
- Identify
 - class invariants which specify relationships among the variables;
 - pre/post-conditions for each method;
 - whether the class is thread-safe and how its states guarded

Sample program: BoundedBufferWithSpec.java

Step 2: Basic Unit Tests

- Create an object of the class, call its methods (in different sequences with different inputs) and assert post-conditions and invariants.

Sample program: BoundedBufferTest.java
Test: testIsEmptyWhenConstructed()

Step 3: Test for Concurrency

- Set up multiple threads performing operations over some amount of time and then somehow test that nothing went wrong
 - Mind that the test programs are concurrent programs too!
- It's best if checking the test property does not require any synchronization

Sample program: `BoundedBufferTest.java`
Test: `testIsFullAfterPuts()`

Cohort Exercise 4

Given BoundedBufferTest.java,

- write two more test cases (with threads)
- document what you are testing for.

Additional Synchronization

- Example: how do we test that everything put into the buffer comes out of it and that nothing else does, assuming there are multiple producers and consumers?
 - A naïve approach: maintain a “shadow” list and assert that the buffer is consistent with the “shadow” list
 - Use a check sum function would be better (see example later)

Example

- Some test data should be generated randomly
- Random number generator can create couplings between classes and timing artifacts because most random number generator classes are thread-safe and therefore introduce additional synchronization.
 - Use pseudo-random number generator

```
static int xorShift (int y) {  
    y ^= (y << 6);  
    y ^= (y >>> 21);  
    y ^= (y << 7);  
    return y;  
}
```

Example

[Click here for a sample program: PutTakeTest.java](#)

Generating More Scheduling

- Test with more active threads than CPUs
- Testing with different processor counts, operating systems, and processor architectures
- Encourage context switching using `Thread.yield()` or `Thread.sleep(10)`

```
Public synchronized void transfer (Account from, Account to, int amount) {  
    from.debit(amount);  
    if (random.nextInt(1000) > THREADHOLD) {  
        Thread.yield();  
    }  
    to.credit(amount);  
}
```


Testing Blocking Operations

- How do we test that an operation has been blocked (in a concurrent context)?

Click here for a sample program: `BoundedBufferTest.java`
Test: `testTakeBlocksWhenEmpty()`

Step 4: Testing for Performance

- Identify appropriate test scenarios – how the class is used
- Sizing empirically for various bounds, e.g., number of threads, buffer capabilities, etc.

Click here for a sample program: `TimedPutTakeTest.java`
`TimedPutTakeTestABQ`; `TimedPutTakeTestLBQ.java`

Cohort Exercise 5

- Design a test to compare the performance difference between `Collections.synchronizedMap` and `ConcurrentHashMap`.

Beyond Testing

- Code review (e.g. Team Explorer)
- Static analysis (e.g. Coverity Scan, and Facebook Infer)
- Symbolic execution (e.g. KLEE, and Microsoft SAGE)
- Model checking (e.g. SPIN, Uppaal, and Microsoft Static Analyzer)
- Theorem proving (e.g. Coq and PVS)