50.003: Elements of Software Construction

Concurrency: Testing

Testing

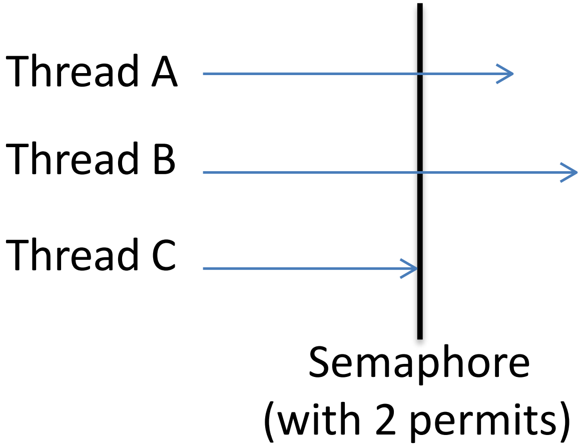
- For sequential programs,
  - Finding the right inputs
- For concurrent programs,
  - Finding the right inputs and scheduling
  - To be able to generate more scheduling, we could use Thread.sleep(), and synchronizers.

Synchronizers

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

Semaphores

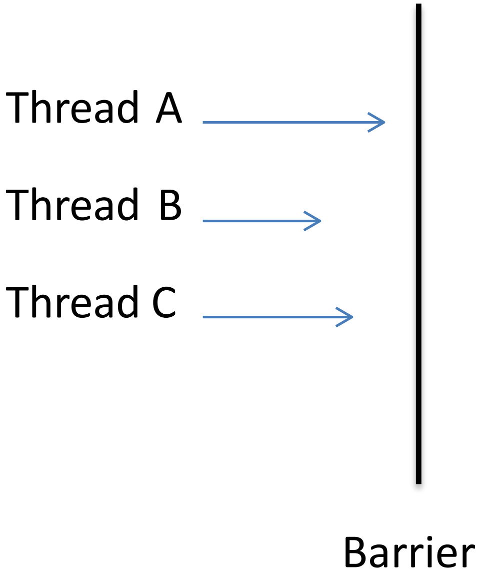
A semaphore maintains a set of permits. Each acquire() blocks if necessary until a permit is available, and then takes it. Each release() adds a permit, potentially releasing a blocked acquirer.



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.



Thread A

Thread B

Thread C

Barrier

Click here for a sample program: BarrierExample.java

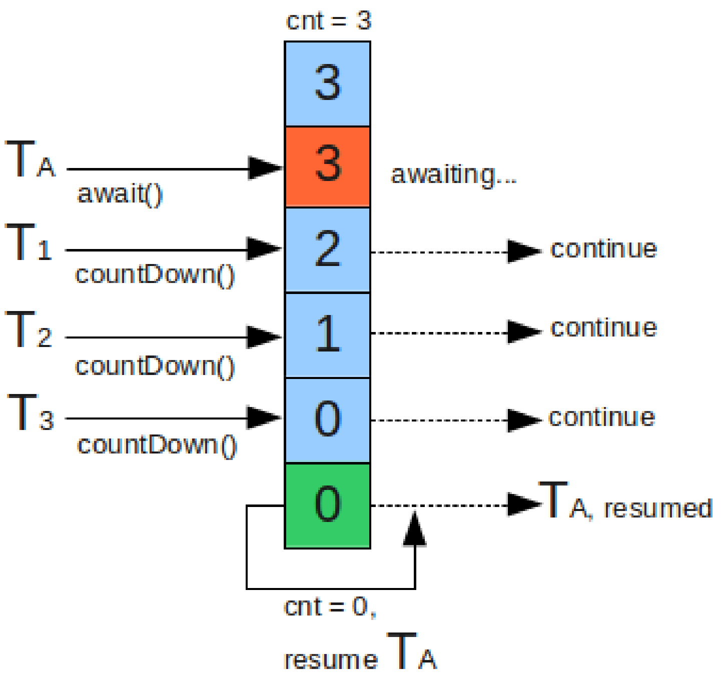
Cohort Exercise 1

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

```
public synchronized void await() throws Exception {
        count--;
        if (count > 0) {
                wait();
        } else {
                notifyAll();
                if (torun != null) {
                        torun.run();
                }
                count = initialcount; // reset back
        }
}
```

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.

```
final CountDownLatch latch = new CountDownLatch(7);
final CountDownLatch finish = new CountDownLatch(7);
// start each searcherThread
// searcherThread latch.countDown() if 'F' found
// if isInterrupted(), break;
// finish.countDown() when it finishes its assignment/interrupted
// main thread's run: latch.await(), interrupt all searchers,
// finish.countDown() to 0
// mainThread.start()
// finish.await(), latch.countDown() till 0
```

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: testIsEmptyWhenConstructued()

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()

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;
}
```

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)?

```
@Test
public void testTakeBlocksWhenEmpty () {
   final BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
   Thread taker = new Thread() {
      public void run() {
                try { int unused = bb.take(); assertTrue(false);
                } catch (InterruptedException success) {//catch}
        }
};
   try {taker.start();
          Thread.sleep(LOCKUP_DETECT_TIMEOUT);
          taker.interrupt();
          taker.join(LOCKUP_DETECT_TIMEOUT);
          assertFalse(taker.isAlive()); //the taker should not be
       } catch (Exception unexpected) {  alive for some time
               assertTrue(false);
       }
}
```

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 ConcurrrentHashMap.
- // look at the codes print

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)