Practical Concurrent and Parallel Programming XII

Testing concurrent programs

IT University of Copenhagen

Friday 2020-11-13 (not full moon Friday)

Kasper Østerbye

Starts at 8:00

Plan for today

Follow up on last week

Testing concurrent programs

- No deadlocks
- No race conditions
- Proper performance
    - Performance testing has been covered earlier

Form last week

- I am sorry many had problems in getting the libraries and the compiler to do what you wanted to!

If there is a kotlin question at the exam, it will be solveable using the kotlin playground.

Exercise 1.3 - From streams to channels

```
fun ex1Channels() = runBlocking { // Solution 1.3
  val tweetChannel = Channel<Tweet>()
  launch { tweetProducer( tweetChannel ) }
  launch { tweetConsumer( tweetChannel ) }
}
```

```
suspend fun tweetProducer(ch: Channel<Tweet>){
  readTweets().forEach{
    ch.send(it)
    delay(5)
  }
  ch.close()
}
```

```
suspend fun tweetConsumer(ch: Channel<Tweet>){
  var count = 200
  for( tw in ch){
    count--
    if (count == 0){
      println(tw.text)
      count = 200
    }
  }
}
```

Exercise 1.5 Filterning channel input

```
fun ex2Channels() = runBlocking {
  val tweetChannel = Channel<Tweet>()
  val swChannel = Channel<Tweet>()
  val amChannel = Channel<Tweet>()
  launch { tweetProducer( tweetChannel ) }
  launch { tweetSplitter( tweetChannel, swChannel, amChannel ) }
  launch { southwest( swChannel ) }
  launch { american( amChannel ) }
}
```

same basic structure as before

> It turns out that Southwest want to have all tweets printed where the negativereason field is non-empty, while American only wants the negative reason printed if the negativereason_confidence is above 0.75.

Who should do this filtering?

- the sender - to reduce amount of data moved
- the receiver - to avoid having to change the sender (service provider)
- receiver should send a filter to sender - so execution can happen at sender
  - security - accepting code to be executed is (potentially) a huge risk

Thread startup

```
public static void main(String[] args) throws InterruptedException{
  final int n = 1000;
  for (int i = 0; i < n; i++){
    final long start = System.nanoTime();
    Thread t = new Thread( ()->{
      long time = System.nanoTime() - start;
      System.out.printf("Time to start %,d\n", time);
    });
    t.start();
    t.join(); // Do not start the next until the previous have finished
  }
}
```

```
Time to start 47,374
Time to start 47,208
Time to start 48,170
Time to start 46,539
Time to start 47,056
Time to start 45,372
Time to start 48,361
```

Coroutine startup

```
runBlocking {
  val measurement = Channel<Long>()
  val reps = 1_000_000
  val warmUp = reps/10
  launch { measurementCollector(reps-warmUp, measurement) }
  for(i in 0..reps-1){
    val start = System.nanoTime()
    launch {
      val time = System.nanoTime() - start
      if (i >= warmUp) measurement.send( time )
    }.join() // coroutines can join too (I learned yesterday)
  }
}
```

```
400.0 ns +/-     2.30 900,000
```

Or about 100 times faster.

Section testing

What is testing

- A method to try to find errors in ones program
- Today (last 15 years at least) testing means: automated testing
- The *automated* is opposed to *manual*

A bit of terminology

- **Subject under test**. The class, method, invariant, property we are trying to test
- **Test program**. A program we use for testing the *subject under test*.
- **Test runner**. A program that runs a number of test programs

Sometimes the testprogram is a single method, other times we need to build more scafolding to perform a test.

Pitfall

Do not test the programming language or its libraries, but make sure subject

Plain unit testing

Test programs usually should have the letters "Test" in their name (e.g "BufferTest", "SpeedTest", "testInsert").

(In this course there have been several classes named "TestSomething" which should have had a different name, like SketchSomething, ProtoSomething, SimpleSomething or ExploreSometing)

Phases

A typical test have three phases:

- Setup - building the scafolding for the test
- Test - running the actual test, and verifying that the results was as expected
- Tear down - removing the scafolding to allow the next test to be run

Often one tries to do these steps in such a manner that several tests can reuse the same setup/tear down.

Case 1 - The increment problem

We saw at the beginning of our course that even this one line is not thread-safe:

```
counter++:
```

Lets go all in and test the different solutions to the increment problem.

Test strategy

I suggest you always follow this plan for testing concurrency:

- Write a test which will fail on the not thread-safe solution (`counter++` in our case)
- Introduce an interface to allow experimenting with different solutions
- Try the simple `synchronized` solution (if such solution exist)
- Try to get a reasonable timing for the `synchronized` solution
- Try an other way to solve the problem - stripes, atomic, immuatable, looser invariants, library wrapping
- Run the other solutions against the same test
- Compare the timings (the not thread-safe is often the fastest (but wrong), and the synchronized the worst)

Incrementer test

```
static void testIncrementer(Incrementer incr) throws Exception {
  final int noCores = 12; // including the hyperthreaded
  final int noThreads = noCores * 10; // make sure each core runs many threads
  final int noReps = 100_000;
  // Body for the threads
  Runnable incrBody = () -> {
    try {
      for(int i=0; i < noReps; i++) incr.increment(); // Critical operation
    } catch (Exception ie) {}
  };
  // Start them all
  for (int i=0; i < noThreads; i++)
    new Thread( incrBody ).start();
  // Wait for them to finish
  Thread.sleep(2000); // We will do something better next slides
  String res = incr.get() == noThreads*noReps ? "SUCCESS" : "FAIL";
  System.out.println(incr.getClass().getName() + " test " + res);
}
```

```
interface Incrementer {
  void increment();
  int get();
}
```

Test driver and interfaces

```
public static void main(String[] args) throws Exception{
  testIncrementer( new IncrementerBasic() );
}
```

```
class IncrementerBasic implements Incrementer{
  int counter = 0;
  public void increment(){ counter++; }
  public int get(){ return counter; }
}
```

Maximizing contention

We have seen earlier that it takes a long time to start a thread.
Ensure no thread finish before an other is even started - use `CyclicBarrier`:

```
final CyclicBarrier barrier = new CyclicBarrier(noThreads+1);
Runnable incrBody = () -> {
  try {
    barrier.await(); // wait for all to be ready
    for(int i=0; i < noReps; i++) incr.increment(); // Critical operation
    barrier.await(); // release main thread
  } catch (Exception ie) {}
};
// Start them all
for (int i=0; i < noThreads; i++)
  new Thread( incrBody ).start();
barrier.await(); // release all threads
barrier.await(); // wait for them all to finish

String res = incr.get() == noThreads*noReps ? "SUCCESS" : "FAIL";
System.out.println(incr.getClass().getName() + " test " + res);
```

Consistent results?

Are we sure we find failures? Run the test 10 times (or 100)

```
public static void main(String[] args) throws Exception{
  for(int i=0; i < 10; i++) testIncrementer( new IncrementerBasic() );
}
```

```
final int noCores = 12;
final int noThreads = noCores * 10;
final int noReps = 10_000;
```

```
IncrementerBasic test FAIL
IncrementerBasic test FAIL
IncrementerBasic test SUCCESS
IncrementerBasic test FAIL
IncrementerBasic test SUCCESS
IncrementerBasic test SUCCESS
IncrementerBasic test SUCCESS
IncrementerBasic test SUCCESS
IncrementerBasic test SUCCESS
IncrementerBasic test SUCCESS
```

```
final int noCores = 12;
final int noThreads = noCores * 10;
final int noReps = 1_000_000;
```

```
IncrementerBasic test FAIL
IncrementerBasic test FAIL
IncrementerBasic test FAIL
IncrementerBasic test FAIL
IncrementerBasic test FAIL
IncrementerBasic test FAIL
IncrementerBasic test FAIL
IncrementerBasic test FAIL
IncrementerBasic test FAIL
IncrementerBasic test FAIL
```

At 100_000 reps I still got an occasional SUCCESS.

Timing

How long does it take from when all threads start until they have all finished?

```
final CyclicBarrier barrier = new CyclicBarrier(noThreads+1);
Runnable incrBody = () -> {
  try {
    barrier.await(); // wait for all to be ready
    for(int i=0; i < noReps; i++) incr.increment(); // Critical operation
    barrier.await(); // release main thread
  } catch (Exception ie) {}
};
// Start them all
for (int i=0; i < noThreads; i++)
  new Thread( incrBody ).start();
barrier.await(); // release all threads
barrier.await(); // wait for them all to finish
```

Where do we start timin, and where do we end the stop timing?

- We do not control when a thread is `runnable` or `running`
- We could
  - measure from when the first thread started working
  - measure from when the last thread started working
  - when the first thread got the chance to start

Timing 2

I tried the different options and could see no difference due to *timing noice*.

```
barrier.await(); // release all threads
long start = System.nanoTime();
barrier.await(); // wait for them all to finish
long time = System.nanoTime()-start;
```

Final test program

```
static void testIncrementer(Incrementer incr) throws Exception {
  final int noCores = 12;
  final int noThreads = noCores * 10;
  final int noReps = 1_000_000;
  final CyclicBarrier barrier = new CyclicBarrier(noThreads+1);
  Runnable incrBody = () -> {
    try {
      barrier.await(); // wait for all to be ready
      //if ( start.get() == 0) start.set( System.nanoTime() );
      for(int i=0; i < noReps; i++) incr.increment();
      barrier.await(); // release main thread
    } catch (Exception ie) {}
  };
  for (int i=0; i < noThreads; i++)
    new Thread( incrBody ).start();

  barrier.await(); // release all threads
  long start = System.nanoTime();
  barrier.await(); // wait for them all to finish

  long time = System.nanoTime()-start;
  String res = incr.get() == noThreads*noReps ? "SUCCESS" : "FAIL";
  System.out.print(incr.getClass().getName() + " test " + res);
  System.out.printf(" in %,dµs \n", time/1_000);
}
```

Synchronized

```
class IncrementerSync implements Incrementer{
  int counter = 0;
  public synchronized void increment(){ counter++; }
  public int get(){ return counter; }
}
```

```
IncrementerBasic test FAIL in 2,195µs
IncrementerBasic test FAIL in 1,491µs
IncrementerBasic test FAIL in 1,264µs
IncrementerBasic test FAIL in 1,163µs
IncrementerBasic test FAIL in 835µs
IncrementerBasic test FAIL in 868µs
IncrementerBasic test FAIL in 856µs
IncrementerBasic test FAIL in 900µs
IncrementerBasic test FAIL in 797µs
IncrementerBasic test FAIL in 926µs
IncrementerSync test SUCCESS in 4,566,066µs
IncrementerSync test SUCCESS in 4,587,900µs
IncrementerSync test SUCCESS in 4,624,848µs
IncrementerSync test SUCCESS in 7,427,304µs
IncrementerSync test SUCCESS in 6,371,031µs
IncrementerSync test SUCCESS in 7,197,069µs
IncrementerSync test SUCCESS in 6,444,359µs
IncrementerSync test SUCCESS in 7,359,687µs
IncrementerSync test SUCCESS in 7,140,521µs
IncrementerSync test SUCCESS in 7,344,686µs
```

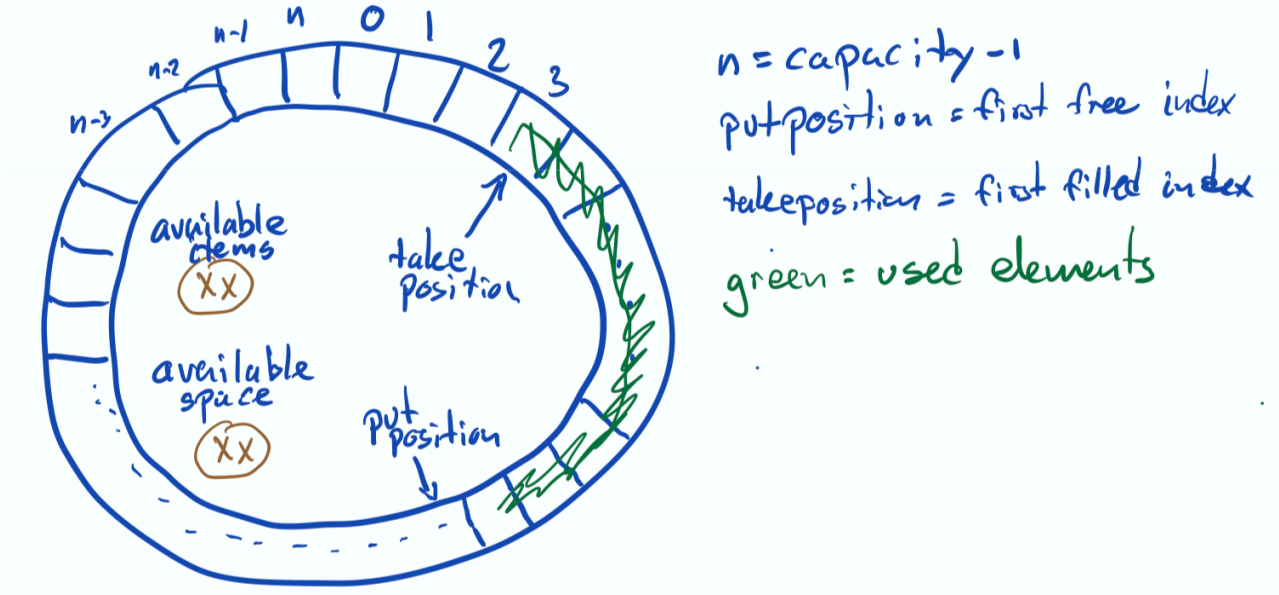
Atomic

```
class IncrementerAtomic implements Incrementer{
  AtomicInteger counter = new AtomicInteger(0);
  public void increment(){ counter.incrementAndGet(); }
  public int get(){ return counter.get(); }
}
```

```
IncrementerSync test SUCCESS in 240,551µs
IncrementerSync test SUCCESS in 258,151µs
IncrementerSync test SUCCESS in 15,317µs
IncrementerSync test SUCCESS in 261,688µs
IncrementerSync test SUCCESS in 11,587µs
IncrementerSync test SUCCESS in 12,130µs
IncrementerSync test SUCCESS in 11,409µs
IncrementerSync test SUCCESS in 10,617µs
IncrementerSync test SUCCESS in 14,162µs
IncrementerSync test SUCCESS in 12,182µs
IncrementerAtomic test SUCCESS in 3,752µs
IncrementerAtomic test SUCCESS in 4,143µs
IncrementerAtomic test SUCCESS in 3,567µs
IncrementerAtomic test SUCCESS in 3,051µs
IncrementerAtomic test SUCCESS in 3,374µs
IncrementerAtomic test SUCCESS in 2,003µs
IncrementerAtomic test SUCCESS in 2,862µs
IncrementerAtomic test SUCCESS in 2,165µs
IncrementerAtomic test SUCCESS in 2,245µs
IncrementerAtomic test SUCCESS in 3,040µs
```

Next case

Case 2 - Circular buffer (or bounded queue)




```
public class SemaphoreBoundedBuffer <E> {
    private final Semaphore availableItems, availableSpaces;
    private final E[] items;
    private int putPosition = 0, takePosition = 0;

    public SemaphoreBoundedBuffer(int capacity) {
        availableItems = new Semaphore(0);
        availableSpaces = new Semaphore(capacity);
        items = (E[]) new Object[capacity];
    }
    ...
```

Insert

```
// Goetz version from book
private synchronized void doInsert(E x) {
    int i = putPosition;
    items[i] = x;
    putPosition = (++i == items.length) ? 0 : i;
}
```

synchronized to protect putPosition. (Unclear to me why `i` is introduced)

```
public void put(E x) throws InterruptedException {
    availableSpaces.acquire();
    doInsert(x);
    availableItems.release();
}
```

if there is no space, wait (using `acquire`)
The take() operation will release the waiting thread

```
public E take() throws InterruptedException {
    availableItems.acquire();
    E item = doExtract();
    availableSpaces.release();
    return item;
}
```

Take

```
// As on previous slide
public E take() throws InterruptedException {
    availableItems.acquire();
    E item = doExtract();
    availableSpaces.release();
    return item;
}
```

```
private synchronized E doExtract() {
    int i = takePosition;
    E x = items[i];
    items[i] = null;
    takePosition = (++i == items.length) ? 0 : i;
    return x;
}
```

Circular counting

I know of three ways to do this:

Goetz (Kasper really do not like this one)

```
putPosition = (++i == items.length) ? 0 : i;
```

Slightly more expensive as it uses modulo:

```
putPosition = (putPosition+1) % items.length
```

Easier to read - let the jit compiler optimize if needed.

```
putPosition++
if (putPosition == items.length) putPosition = 0
```

Testing - Do not forget the regular tests

Setup:

```
bq = BoundedQueue(3)
```

Tests:

```
assertTrue(bq.isEmpty());
assertTrue(!bq.isFull());

bq.put(7); bq.put(9); bq.put(13);
assertTrue(!bq.isEmpty());
assertTrue(bq.isFull());

assertEquals(bq.take(), 7);
assertEquals(bq.take(), 9);
assertEquals(bq.take(), 13);

assertTrue(bq.isEmpty());
assertTrue(!bq.isFull());
```

(Slidetesting - each of these tests should have been made in seperate test methods to fit into the overall quality assurance and continious integration...)

Testing concurrency properties

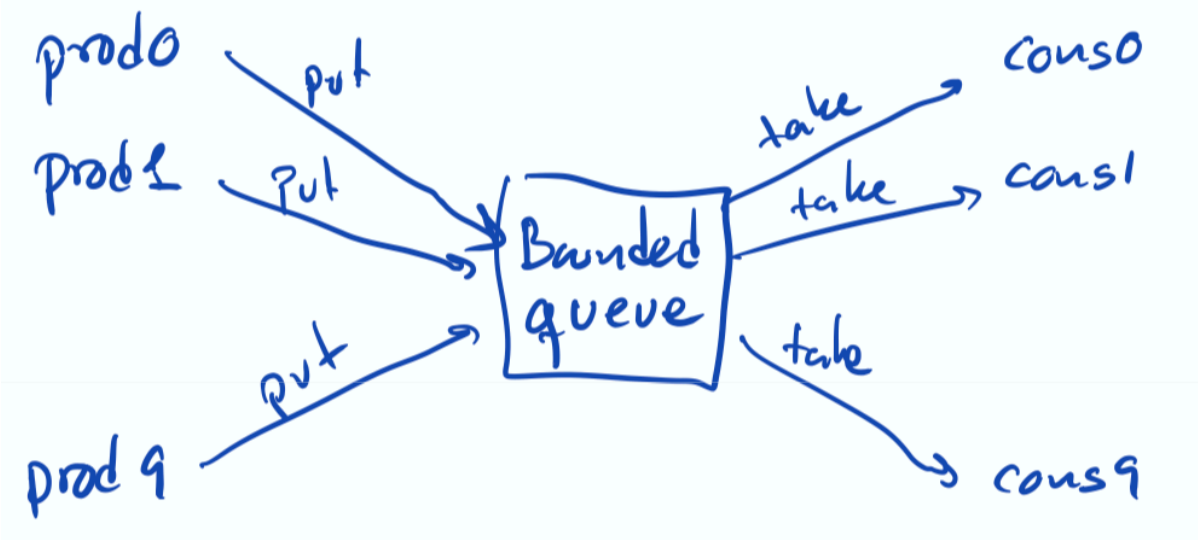
Sequential 1-thread test with precise results

- (previous slide)

Concurrent n-thread test with aggregate results

- that make it *plausible* that invariants hold

Make it *plausible* that an error would occour



We can try with 10 producers and 10 consumers to try and put some contension on the buffer.
Hint: as before, use more threads than you have cores to make sure the threads are suspended

Test program

We need the test program to be able to:

- have a producer thread we can start a number of (10 in the previous figure)
- have a consumer thread ….
- Have a way to find out if what comes in is what comes out
- Start all producers and consumers at the same time
- Ideally you should run the program on different machines with different set of cores

Ready - go

In order to maximize contention, it is usefull to make sure all producers and consumers start to access the buffer at the same time. (as with incrementer case)

A `CyclicBarrier` initilized to `N` will pause `N` threads calling `await()`, and release them all when all `N` have called `await()`. The `CyclicBarrier` will then reset to waiting for `N` threads again.

```
// Main thread:...
for (int i = 0; i < nPairs; i++) {
    pool.execute(new Producer());
    pool.execute(new Consumer());
}
barrier.await(); // wait for all threads to be ready
barrier.await(); // wait for all threads to finish
```

```
class Producer implements Runnable {
    public void run() {
      barrier.await(); // await all to be ready
      for (int i = nTrials; i > 0; --i) {
          bb.put(...);
      }
      barrier.await(); // Signal I am done
    }
}
```

what comes in is what comes out

One synchronization/race condition could be that elements are overwritten or the same is removed twice.

- Maintain lists of items inserted, and lists received
  - Take care you are not creating contention elsewhere (dealing with the lists)
- Sum of inserted hashkeys equals sum of removed removed hashkeys
  - If we insert numbers, just use the number directly
  - Each thread sums its own insertion/removals,
  - and only reports its total sum to a shared sum
- Which numbers should we insert? Something which:
  - Is not discovered by the jit compiler and removed
  - Is efficient to generate (not prime numbers)
  - Do not give overflow (not fibonacci numbers)
  - cheap random numbers are a good candidate

Full producer

```
class Producer implements Runnable {
    public void run() {
        try {
            int seed = (this.hashCode() ^ (int) System.nanoTime()); // "random"
            int sum = 0; // local sum, not shared with other threads
            barrier.await(); // Wait for all producers and consumers to get ready
            for (int i = nTrials; i > 0; --i) {
                bb.put(seed);
                sum += seed;
                seed = xorShift(seed); // discount random next()
            }
            putSum.getAndAdd(sum); // putSum is an atomic integer
            barrier.await();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

The test

```
void test() {
    try {
        for (int i = 0; i < nPairs; i++) {
            pool.execute(new Producer());
            pool.execute(new Consumer());
        }
        barrier.await(); // wait for all threads to be ready
        barrier.await(); // wait for all threads to finish
        // Test that the sum of inserted elements are the same as those taken out
        assertEquals(putSum.get(), takeSum.get());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Test strategy

- Write a test which will fail on the not thread-safe solution
  - Figuring out how to check for fail is the creative part - sums and checksums are good candidates
- Introduce an interface to allow experimenting with different solutions
- Try the simple `synchronized` solution (if such solution exist)
- Try to get a reasonable timing for the `synchronized` solution
- Try an other way to solve the problem - stripes, atomic, immuatable, looser invariants, library wrapping
- Run the other solutions against the same test
- Compare the timings (the not thread-safe is often the fastest (but wrong), and the synchronized the worst)

AtomicInteger - compare and set/swap/exchange

On the intel (and all other multicore CPU's) there is an machine instruction `CMPXCHG`, which is similar to:

```
myBool = myAtomicInteger.compareAndSet(expect, update)
```

This is the fundamental atomic operation. 'addAndGet(delta)' can be implemented as

```
int AtomicInteger.addAndGet(int delta){
  int current = this.get();
  while (!compareAndSet(current, current+delta) )
    current = this.get();
  return current+delta;
}
```

You need to use this idea in one of the exercises