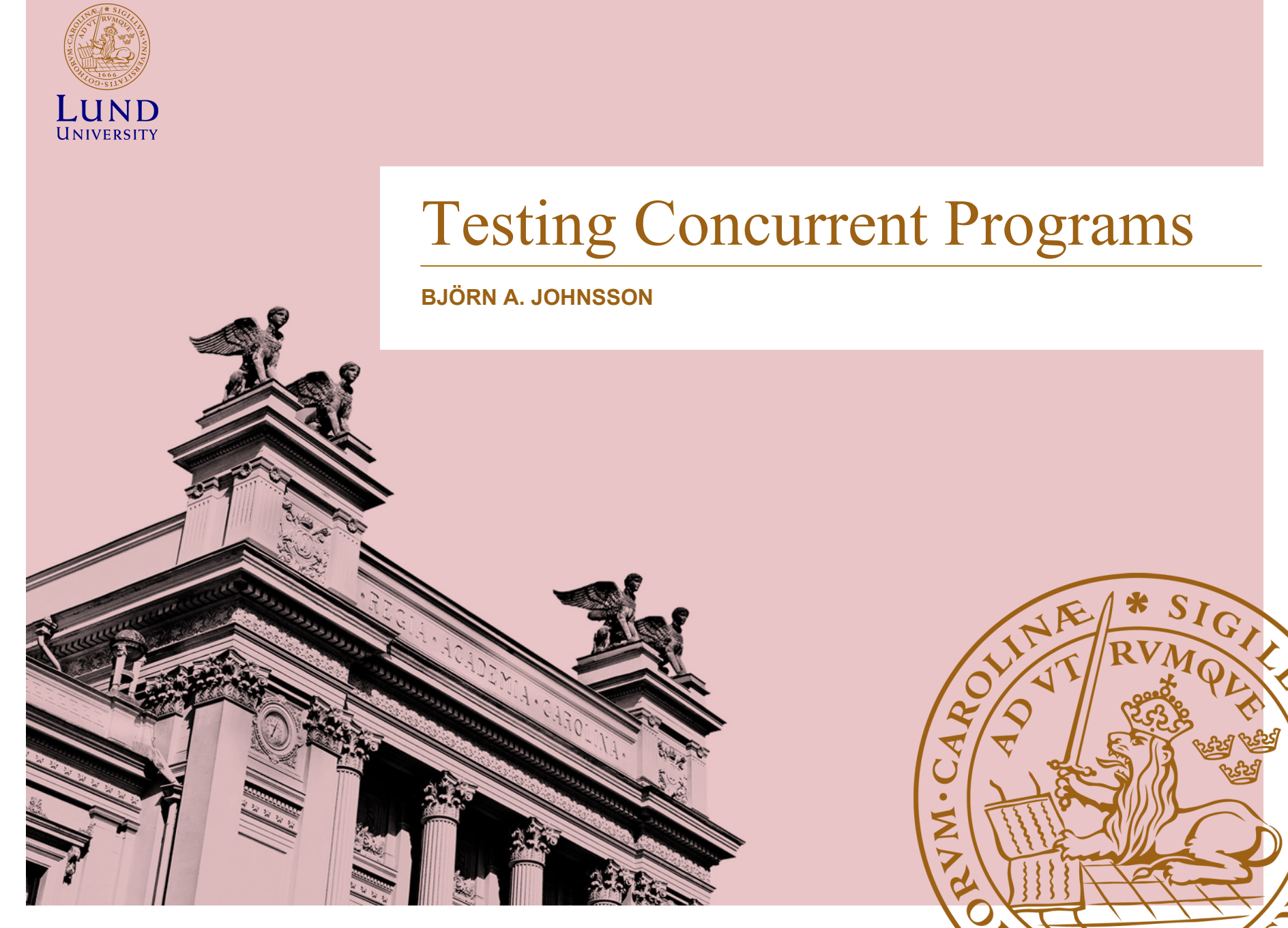

Testing Concurrent Programs

BJÖRN A. JOHNSSON

Introduction

- Concurrency introduces degree of non-determinism
- Similar techniques/patterns, larger space of errors
- Errors are rare probabalistic occurences, not deterministic ones.
- Tests: for *safety*, or *liveness*

- Chapter overview
  - Testing for correctness
  - (Testing for performance)
  - (Pitfalls, performance testing)
  - (Complementary testing approaches)

LUND UNIVERSITY

Testing for correctness

```
@ThreadSafe
public class BoundedBuffer<E> {
  private final Semaphore availableItems, availableSpaces;
  @GuardedBy("this") private final E[] items;
  @GuardedBy("this") private int putPosition = 0,
takePosition = 0;

  public BoundedBuffer(int capacity) {
    availableItems = new Semaphore(0);
    availableSpaces = new Semaphore(capacity);
    items = (E[]) new Object[capacity];
  }
  public boolean isEmpty() {
    return availableItems.availablePermits() == 0;
  }
  public boolean isFull() {
    return availableSpaces.availablePermits() == 0;
  }
  public void put(E x) throws InterruptedException {
    availableSpaces.acquire();
    doInsert(x);
    availableItems.release();
  }
  public E take() throws InterruptedException {
    availableItems.acquire();
    E item = doExtract();
    availableSpaces.release();
    return item;
  }
  private synchronized void doInsert(E x) {
    int i = putPosition;
    items[i] = x;
    putPosition = (++i == items.length)? 0 : i;
  }
  private synchronized E doExtract() {
    int i = takePosition;
    E x = items[i];
    items[i] = null;
    takePosition = (++i == items.length)? 0 : i;
    return x;
  }
}
```

Basic unit tests

- Start simple (non-concurrent)!
- Include sequential test
- Excludes problems *not* related concurrency

```
class BoundedBufferTest extends TestCase {
  void testIsEmptyWhenConstructed() {
    BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
    assertTrue(bb.isEmpty());
    assertFalse(bb.isFull());
  }
  void testIsFullAfterPuts() throws InterruptedException {
    BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
    for (int i = 0; i < 10; i++)
      bb.put(i);
    assertTrue(bb.isFull());
    assertFalse(bb.isEmpty());
  }
}
```

Testing blocking operations

- Lacking support in most testing frameworks
  - Helper threads – which test failed?
- Success if thread does *not* proceed
- Unblock via interruption
  - Requires interruption responsiveness
- How long to wait? Slow or blocked?

- Don't use `Thread.getState`[1]!

```
void testTakeBlocksWhenEmpty() {
  final BoundedBuffer<Integer> bb = new
BoundedBuffer<Integer>(10);

  Thread taker = new Thread() {
    public void run() {
      try {
        int unused = bb.take();
        fail(); // if we get here, it's an error
      } catch (InterruptedException success) { }
  }};
  try {
    taker.start();
    Thread.sleep(LOCKUP_DETECT_TIMEOUT);
    taker.interrupt();
    taker.join(LOCKUP_DETECT_TIMEOUT);
    assertFalse(taker.isAlive());
  } catch (Exception unexpected) {
    fail();
  }
}
```

[1] Exercise!

Testing safety

- Test for data races
  - Multiple threads doing `put` and `take` for *how* long
  - Test nothing went wrong
- ”Chicken-and-egg” problem…
- For classes used in producer-consumer
  - Everything put into queue comes out, and nothing else
  - *Naïve:* ”shadow” list – distorts scheduling due to syncronization and blocking
  - *Better:* Use checksums for enqueued items
    - » *Don’t use compiler guessable checksums!*

Testing safety (cont'd)

```
public class PutTakeTest {
  private static final ExecutorService pool =
 Executors.newCachedThreadPool();

  private final AtomicInteger putSum = new AtomicInteger(0);
  private final AtomicInteger takeSum = new
AtomicInteger(0);

  private final CyclicBarrier barrier;
  private final BoundedBuffer<Integer> bb;
  private final int nTrials, nPairs;

  public static void main(String[] args) {
    new PutTakeTest(10, 10, 100000).test();
    pool.shutdown();
  }
  PutTakeTest(int capacity, int npairs, int ntrials) {
    this.bb = new BoundedBuffer<Integer>(capacity);
    this.nTrials = ntrials;
    this.nPairs = npairs;
    this.barrier = new CyclicBarrier(npairs * 2 + 1);
  }
  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
      assertEquals(putSum.get(), takeSum.get());
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
  class Producer implements Runnable { /* next slide */ }
  class Consumer implements Runnable { /* next slide */ }
}
```

Testing safety (cont'd)

```
class Producer implements Runnable {
  public void run() {
    try {
      int seed = (this.hashCode() ^ (int)System.nanoTime());
      int sum = 0;
      barrier.await();
      for (int i = nTrials; i > 0; --i) {
        bb.put(seed);
        sum += seed;
        seed = xorShift(seed);
      }
      putSum.getAndAdd(sum);
      barrier.await();
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

```
class Consumer implements Runnable {
  public void run() {
    try {
      barrier.await();
      int sum = 0;
      for (int i = nTrials; i > 0; --i) {
        sum += bb.take();
      }
      takeSum.getAndAdd(sum);
      barrier.await();
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

Testing resource management

- Test e.g. r*esource leaks*
  - Objects that hold other objects should not hold them when unnecessary
- Important for bounded classes
  - The point is to not have uncontrollably growing objects (needs tests)

```
class Big { double[] data = new double[100000]; }

void testLeak() throws InterruptedException {
  BoundedBuffer<Big> bb = new BoundedBuffer<Big>(CAPACITY);
  int heapSize1 = /* snapshot heap, “forces” GC */;
  for (int i = 0; i < CAPACITY; i++)
    bb.put(new Big());
  for (int i = 0; i < CAPACITY; i++)
    bb.take();
  int heapSize2 = /* snapshot heap, “forces” GC */;
  assertTrue(Math.abs(heapSize1-heapSize2) < THRESHOLD);
}
```

More tricks for interleavings

- Number of threads > number of CPUs
- Test on various systems with different:
  - Number of CPUs, processor clock frequencies, operating systems, processor architectures, …
- Use `Thread.yield` – more context switches
  - Code ”messiness” could be fixed with *AOP*

```
public synchronized void transferCredits(Account from, Account to, int amount) {
  from.setBalance(from.getBalance() - amount);
  if (random.nextInt(1000) > THRESHOLD)
    Thread.yield();
  to.setBalance(to.getBalance() + amount);
}
```

Summary

- Testing correctness in concurrent programs is challenging
  - Low-probability failure modes
  - Sensitive to timing, load, and other hard-to-reproduce conditions
- Risk of tests introducing additional synchronization and timing constraints
  - Might ”hide” concurrency problems being tested

LUND UNIVERSITY

LUND
UNIVERSITY