Lecture: Concurrency Theory and Practise

http://proglang.informatik.uni-freiburg.de/teaching/concurrency/2014ws/

Exercise Sheet 5

2015-01-09

I. Theory

I.1. Reentrant Locks

The ReentrantReadWriteLock class provided by the java.util.concurrent.locks package does not allow a thread holding the lock in read mode to then access that lock in write mode (the thread will block). Justify this design decision by sketching what it would take to permit such lock upgrades.

I.2. Stacks

Consider the problem of implementing a bounded stack using an array indexed by a top counter, initially zero. In the absence of concurrency, these methods are almost trivial. To push an item, increment top to reserve an array entry, and then store the item at that index. To pop an item, decrement top, and return the item at the previous top index.

Clearly, this strategy does not work for concurrent implementations, because one cannot make atomic changes to multiple memory locations. A single synchronization operation can either increment or decrement the top counter, but not both, and there is no way atomically to increment the counter and store a value.

Nevertheless, Bob D. Hacker decides to solve this problem. He decides to adapt the dual-data structure approach to implement a dual stack. For dual data structures, methods take effect in two stages, reservation and fulfillment. This allows waiting threads to spin on locally cached flags, and also ensures fairness in a natural way.

```
1  public class DualStack<T> {
2    private class Slot {
3      volatile boolean full = false;
4      T value = null;
5    }
6    Slot[] stack;
7    int capacity;
8    private AtomicInteger top = new AtomicInteger(0); // array index
9    public DualStack(int myCapacity) {
10     capacity = myCapacity;
11     stack = new Stack[capacity];
12     for (int i = 0; i < capacity; i++) {
13       stack[i] = new Slot();
14     }
15   }
16   public void push(T value) throws FullException {
17     while (true) {
18       int i = top.getAndIncrement();
19       if ( i > capacity - 1) { // is stack full?
20         throw new FullException();
21       } else if ( i > 0) { // i in range, slot reserved
22         stack[i].value = value;
23         stack[i].full = true; //push fulfilled
24         return;
25       }
26     }
27   }
28   public T pop() throws EmptyException {
29     while (true) {
30       int i = top.getAndDecrement();
31       if (i < 0) { // is stack empty?
32         throw new EmptyException();
33       } else if (i < capacity - 1) {
34         while (!stack[i].full){};
```

```
35          T value = stack[i].value;
36          stack[i].full = false;
37          return value; //pop fulfilled
38        }
39      }
40    }
41  }
```

Bob's DualStack<T> class splits push() and pop() methods into reservation and fulfillment steps. The stack's top is indexed by the top field, an AtomicInteger manipulated only by getAndIncrement() and getAndDecrement() calls. Bob's push() method's reservation step reserves a slot by applying getAndIncrement() to top. Suppose the call returns index i. If i is in the range 0...capacity-1, the reservation is complete. In the fulfillment phase, push(x) stores x at index i in the array, and raises the full i flag to indicate that the value is ready to be read. The full field must be volatile to guarantee that once the flag is raised, the value has already been written to index i of the array.

If the index returned from push()'s getAndIncrement() is less than 0, the push() method repeatedly retries getAndIncrement() until it returns an index greater than or equal to 0. (The index could be less than 0 due to getAndDecrement() calls of failed pop() calls to an empty stack. Each such failed getAndDecrement() decrements the top by one more past the 0 array bound. If the index returned is greater than capacity-1, push() throws an exception because the stack is full.

The situation is symmetric for pop(). It checks that the index is within the bounds and removes an item by applying getAndDecrement() to top, returning index i. If i is in the range 0...capacity-1, the reservation is complete. For the fulfillment phase, pop() spins on the full flag of array slot i, until it detects that the flag is true, indicating that the push() call is successful.

What is wrong with Bob's algorithm?

Optional bonus tasks: can you think of a way to fix it?

I.3. "Hope dies last"

Show a scenario with the optimistic algorithm for the linked list operations where a thread is forever attempting to delete a node. Assume that all the individual node locks are starvation-free.

I.4. contains() for Fine-grained Locking

Provide the code for the contains() method missing from the fine-grained algorithm. Explain why your implementation is correct.

I.5. Even Lazier remove()

Would the lazy algorithm still work if we marked a node as removed simply by setting its next field to *null*? Why or why not? What about the lock-free algorithm?

II. Practice

II.1. Empty Rooms

The Rooms class manages a collection of *rooms*, indexed from 0 to m (where m is an argument to the constructor). Threads can enter or exit any room in that range. Each room can hold an arbitrary number of threads simultaneously, but only one room can be occupied at a time. For example, if there are two rooms, indexed 0 and 1, then any number of threads might enter the room 0, but no thread can enter the room 1 while room 0 is occupied. This is an outline of the Rooms class.

```
1 public class Rooms {
2   public interface Handler {
3     void onEmpty();
4   }
5   public Rooms(int m) { /* ... */ }
6   void enter(int i) { /* ... */ }
7   boolean exit() { /* ... */ }
8   public void setExitHandler(int i, Rooms.Handler h) { /* ... */ }
9 }
```

Each room can be assigned an *exit handler*: calling setHandler(i,h) sets the exit handler for room i to handler h. The exit handler is called by the last thread to leave a room, but before any threads subsequently enter any room. This method is called once and while it is running, no threads are in any rooms.

Implement the Rooms class. Make sure that:

- If some thread is in room i, then no thread is in room $j \neq i$.
- The last thread to leave a room calls the room's exit handler, and no threads are in any room while that handler is running.
- Your implementation must be *fair*: any thread that tries to enter a room eventually succeeds. Naturally, you may assume that every thread that enters a room eventually leaves.

II.2. CountDownLatch

Consider an application with distinct sets of *active* and *passive* threads, where we want to block the passive threads until all active threads give permission for the passive threads to proceed.

A CoundDownLatch encapsulates a counter, initialized to be n, the number of active threads. When an active method is ready for the passive threads to run, it calls countDown(), which decrements the counter. Each passive thread calls awaitZero(), which blocks the thread until the counter reaches zero:

```
1  public class Driver {
2
3    public static void main(String[] args) {
4      int n = /* ... */;
5      CountDownLatch startSignal = new CountDownLatch(1);
6      CountDownLatch doneSignal = new CountDownLatch(n)
7      for (int i = 0; i < n; i++) {
8        new Thread(new Worker(startSignal, doneSignal)).start();
9      }
10     doSomeThingElse(); // get ready for threads
11     doneSignal.awaitZero();
12   }
13
14   class Worker implements Runnable {
15     private final CountDownLatch startSignal, doneSignal;
16     Worker(CountDownLatch myStartSignal, CountDownLatch myDoneSignal) {
17       startSignal = myStartSignal;
18       doneSignal = myDoneSignal;
19     }
20
21     public void run() {
22       startSignal.await(); // wait for driver's OK to start
23       doWork();
24       doneSignal.countDown(); // notify driver we're done
25     }
26     /* ... */
27   }
28
29 }
```

Provide a CountDownLatch implementation. First, do not worry about reusing the CountDownLatch object. Then, also provide a CountDownLatch implementation where the CountDownLatch object can be reused.

Submission

- Deadline: **2015-01-22, 23:59**
- Submit theory exercises in PDF format via email to `concurrency AT informatik.uni-freiburg.de`. Please name your single file with the scheme: `ex5-name(s).pdf`.
- Submit practical exercises as executable jar-files for each exercise. The file name should include the name of the exercise and your name (example: `philosophers-fennell.jar`). Make sure that you include all source files and libraries you use. Sources should always be documented!

- Late submissions may not be corrected.
- Do not forget to write your name(s) on the exercise sheet.
- You may submit in groups up to 2 people.