

# Practical Concurrent and Parallel Programming

Martin Rønning Bech (mrob@itu.dk)

PRCPP E2014  
January 7, 2015

## 1 Question 1:

To make the *SimpleDeque* thread safe we can apply the Java Monitor Pattern, that is that any mutable state is put in private fields and are guarded by the objects own lock.

So in our *SimpleDeque* we ensure that *items*, *bottom* and *top* are private and are *GuardedBy*("this"). We then make the public method's *push*, *pop* and *steal* *synchronized*. These changes makes the class threadsafe by ensuring that only a single thread will ever be allowed to make updates to its internal mutable state.

The *GuardedBy*("this") annotations are added to all internal state that needs to be guarded by object lock, this is to communicate to others that any updates made by these fields must only be done when the *this* lock is taken, it is also useful in combination with tools that can be used to detect if the correct lock is always taken when access are made to the field.

```
@GuardedBy("this")
private long bottom = 0, top = 0;
@GuardedBy("this")
private final T[] items;

public synchronized void push(T item) {...}
public synchronized T pop() {...}
public synchronized T steal() {...}
```

## 2 Question 2

The *SortTask* is a immutable object, this is done by making the public fields *final*. By being immutable we ensure that after its creation no one can update it, so we can safely pass it around threads. Any thread that needs to do an

“update” of the *SortTask* will instead have to create a new instance with the updated values.

### 3 Question 3

```
private static void sqmtWorkers(Deque<SortTask> queue, int threadCount) {
    //Initialize ongoing counter with the size of the queue
    //We assume the queue only has a single task
    LongAdder ongoing = new LongAdder();
    ongoing.increment();

    //Creating threads:
    Thread[] threads = new Thread[threadCount];
    for(int t = 0; t < threadCount; t++){
        threads[t] = new Thread(()->{
            SortTask task;
            while (null != (task = getTask(queue, ongoing))) {
                //We have a task now partition!
                final int[] arr = task.arr;
                final int a = task.a, b = task.b;
                if (a < b) {
                    int i = a, j = b;
                    int x = arr[(i+j) / 2];
                    do {
                        while (arr[i] < x) i++;
                        while (arr[j] > x) j--;
                        if (i <= j) {
                            swap(arr, i, j);
                            i++; j--;
                        }
                    } while (i <= j);

                    //Increment the counter when pushing
                    queue.push(new SortTask(arr, a, j));
                    ongoing.increment();
                    queue.push(new SortTask(arr, i, b));
                    ongoing.increment();
                }
                //We have sorted something, time to decrement
                ongoing.decrement();
            }
        });
        //Start the thread
        threads[t].start();
    }
}
```

```

    }

    //Wait for the threads to finish
    for(int t = 0; t < threadCount; t++){
        try{
            threads[t].join();
        }catch(InterruptedException e){}
    }
}

```

Output:

Before:

34 13 35 6 35 24 21 38 17 27 28 3 28 7 19 29 37 24 18 20

After:

3 6 7 13 17 18 19 20 21 24 24 27 28 28 29 34 35 35 37 38

## 4 Question 4

I have used a few helper functions: From the course: `assertEquals`, `assertTrue`.  
Own: `assertNull`, `awaitBarrier`. Can be found in appendix [A](#)

```

static void sequentialDequeTest(Deque<Integer> queue) throws Exception{
    //Check that it only returns null
    assertNull(queue.pop());
    assertNull(queue.steal());

    //Check that pop/push works on single insert
    queue.push(42);
    assertEquals(42, queue.pop());
    assertNull(queue.pop());

    //Check that steal work on single insert
    queue.push(43);
    assertEquals(43, queue.steal());
    assertNull(queue.pop());

    queue.push(44);
    queue.push(45);
    queue.push(46);

    //Check that steal takes from the back

```

```

        assertEquals(44, queue.steal());

        //Check that pop takes from the front
        assertEquals(46, queue.pop());
    }

    static void parallelDequeTest(Deque<Integer> queue, int threadCount) throws Exception {
        CyclicBarrier barrier = new CyclicBarrier((threadCount*3)+1);
        int pushedSum = 0;

        //Start pushing threads
        LongAdder pushed = new LongAdder();
        for(int t = 0; t < threadCount; t++){
            final int lt = t;
            new Thread(()->{
                awaitBarrier(barrier);
                long p = 0;
                for(int i = 0; i < 1_000_000; i++){
                    Random random = new Random();
                    int r = random.nextInt() % 1000;
                    p += r;
                    queue.push(r);
                }
                pushed.add(p);
                awaitBarrier(barrier);
            }).start();
        }

        //Start pop threads
        LongAdder popped = new LongAdder();
        for(int t = 0; t < threadCount; t++){
            final int lt = t;
            new Thread(()->{
                awaitBarrier(barrier);
                long pop = 0;
                for(int i = 0; i < 1_000_000; i++){
                    Integer p = queue.pop();
                    if(p != null){
                        pop += p;
                    }
                }
                popped.add(pop);
                awaitBarrier(barrier);
            }).start();
        }
    }

```

```
//Start stealing threads
LongAdder stolen = new LongAdder();
for(int t = 0; t < threadCount; t++){
    final int lt = t;
    new Thread()->{
        awaitBarrier(barrier);
        long s = 0;
        for(int i = 0; i < 1_000_000; i++){
            Integer p = queue.steal();
            if(p != null){
                s += p;
            }
        }
        stolen.add(s);
        awaitBarrier(barrier);
    }).start();;
}

//Start test
awaitBarrier(barrier);
//Wait for the test to stop
awaitBarrier(barrier);

//Get the remaining sum
long remaining = 0;
Integer p = queue.pop();
while(p != null){
    remaining += p;
    p = queue.pop();
}

//Get the sum of the threads
long pushedsum = pushed.sum();
long retrievedsum = remaining + popped.sum() + stolen.sum();

//Check that sum matches
assertEquals(retrievedsum, pushedsum);
}
```

## 5 Question 5

With 20 million integers:

Threads	Time (Seconds)
1	5.163053139
2	7.496664894
3	6.961242125
4	7.567593447
5	5.036839369
6	4.811132867
7	4.600989057
8	4.53502937

```

public static double sqmtBenchMarkVersion(int threadCount){
    SimpleDeque<SortTask> queue = new SimpleDeque<SortTask>(100000);
    int[] array = IntArrayUtil.randomIntArray(20_000_000);
    queue.push(new SortTask(array, 0, array.length-1));
    CyclicBarrier barrier = new CyclicBarrier(threadCount+1);

    //Initialize ongoing counter with the size of the queue
    //We assume the queue only has a single task
    LongAdder ongoing = new LongAdder();
    ongoing.increment();

    //Creating threads:
    Thread[] threads = new Thread[threadCount];
    for(int t = 0; t < threadCount; t++){
        threads[t] = new Thread(()->{
            awaitBarrier(barrier);
            SortTask task;
            while (null != (task = getTask(queue, ongoing))) {
                //We have a task now partition!
                final int[] arr = task.arr;
                final int a = task.a, b = task.b;
                if (a < b) {
                    int i = a, j = b;
                    int x = arr[(i+j) / 2];
                    do {
                        while (arr[i] < x) i++;
                        while (arr[j] > x) j--;
                        if (i <= j) {
                            swap(arr, i, j);
                            i++; j--;
                        }
                    } while (i <= j);
                }
                //Increment the counter when pushing
                queue.push(new SortTask(arr, a, j));
            }
        });
    }
}

```

```

        ongoing.increment();
        queue.push(new SortTask(arr, i, b));
        ongoing.increment();
    }
    //We have sorted something, time to decrement
    ongoing.decrement();
}
    awaitBarrier(barrier);
});
//Start the thread
    threads[t].start();
}

//Waiting for threads
awaitBarrier(barrier);
//Threads started
Timer t = new Timer();
awaitBarrier(barrier);
//Threads done
return t.check();
}

```

## 6 Question 6

```

private static void multiQueueMultiThreadCL(final int threadCount) {
    int[] arr = IntArrayUtil.randomIntArray(size);
    // To do: ... create queues and so on, then call mqmtWorkers(queues, threadCount)
    System.out.println(IntArrayUtil.isSorted(arr));
}

private static void mqmtWorkers(Deque<SortTask>[] queues, int threadCount) {
    //Initialize ongoing counter with the size of the queue
    //We assume the queue only has a single task
    LongAdder ongoing = new LongAdder();
    ongoing.increment();

    //Creating threads:
    Thread[] threads = new Thread[threadCount];
    for(int t = 0; t < threadCount; t++){
        final int tl = t;
        threads[t] = new Thread(()->{
            SortTask task;
            while (null != (task = getTask(tl, queues, ongoing))) {

```

```

        //We have a task now partition!
        final int[] arr = task.arr;
        final int a = task.a, b = task.b;
        if (a < b) {
            int i = a, j = b;
            int x = arr[(i+j) / 2];
            do {
                while (arr[i] < x) i++;
                while (arr[j] > x) j--;
                if (i <= j) {
                    swap(arr, i, j);
                    i++; j--;
                }
            } while (i <= j);

            //Increment the counter when pushing
            queues[t1].push(new SortTask(arr, a, j));
            ongoing.increment();
            queues[t1].push(new SortTask(arr, i, b));
            ongoing.increment();
        }
        //We have sorted something, time to decrement
        ongoing.decrement();
    }
});
//Start the thread
threads[t].start();
}

//Wait for the threads to finish
for(int t = 0; t < threadCount; t++){
    try{
        threads[t].join();
    }catch(InterruptedException e){}
}
}

// Tries to get a sorting task. If task queue is empty, repeatedly
// try to steal, cyclically, from other threads and if that fails,
// yield and then try again, while some sort tasks are not processed.

private static SortTask getTask(final int myNumber, final Deque<SortTask>[] queues,
    LongAdder ongoing) {
    final int threadCount = queues.length;
    SortTask task = queues[myNumber].pop();
    if (null != task)

```



```

        return task;
    else {
        do {
            //Lets try to steal a task from someone...
            for(int i = 0; i < queues.length; i++){
                if(i != myNumber){
                    task = queues[i].steal();
                    if(task != null){
                        return task;
                    }
                }
            }
            Thread.yield();
        } while (ongoing.longValue() > 0);
        return null;
    }
}

```

## 7 Question 7

Threads	Time
1	4.673261823
2	3.164645371
3	2.573688754
4	1.94583811
5	2.272075977
6	2.179852209
7	1.950004463
8	2.179962992

```

private static void benchMarkMultiQueueMultiThread() {
    System.out.println("Threads\tTime");
    for(int i = 1; i<9; i++){
        double time = mqmtBenchMarkVersion(i);
        System.out.println(i + "\t" + time);
    }
}

private static double mqmtBenchMarkVersion(int threadCount) {
    SimpleDeque<SortTask>[] queues = new SimpleDeque[threadCount];
    for(int i = 0; i < threadCount; i++){
        queues[i] = new SimpleDeque<SortTask>(100000);
    }
}

```

```

}
int[] array = IntArrayUtil.randomIntArray(20_000_000);
queues[0].push(new SortTask(array, 0, array.length-1));
CyclicBarrier barrier = new CyclicBarrier(threadCount+1);

//Initialize ongoing counter with the size of the queue
//We assume the queue only has a single task
LongAdder ongoing = new LongAdder();
ongoing.increment();

//Creating threads:
Thread[] threads = new Thread[threadCount];
for(int t = 0; t < threadCount; t++){
    final int tl = t;
    threads[t] = new Thread(()->{
        awaitBarrier(barrier);
        SortTask task;
        while (null != (task = getTask(tl, queues, ongoing))) {
            //We have a task now partition!
            final int[] arr = task.arr;
            final int a = task.a, b = task.b;
            if (a < b) {
                int i = a, j = b;
                int x = arr[(i+j) / 2];
                do {
                    while (arr[i] < x) i++;
                    while (arr[j] > x) j--;
                    if (i <= j) {
                        swap(arr, i, j);
                        i++; j--;
                    }
                } while (i <= j);

                //Increment the counter when pushing
                queues[tl].push(new SortTask(arr, a, j));
                ongoing.increment();
                queues[tl].push(new SortTask(arr, i, b));
                ongoing.increment();
            }
            //We have sorted something, time to decrement
            ongoing.decrement();
        }
        awaitBarrier(barrier);
    });
    //Start the thread
    threads[t].start();
}

```

```

    }

    //Waiting for threads
    awaitBarrier(barrier);
    //Threads started
    Timer t = new Timer();
    awaitBarrier(barrier);
    //Threads done
    return t.check();
}

```

## 8 Question 8

```

class ChaseLevDeque<T> implements Deque<T> {
    volatile long bottom = 0;
    AtomicLong top = new AtomicLong(0);
    T[] items;

    public ChaseLevDeque(int size) {
        this.items = makeArray(size);
    }

    @SuppressWarnings("unchecked")
    private static <T> T[] makeArray(int size) {
        // Java's @$#@?! type system requires this unsafe cast
        return (T[])new Object[size];
    }

    private static int index(long i, int n) {
        return (int)(i % (long)n);
    }

    public void push(T item) { // at bottom
        final long b = bottom, t = top.get(), size = b - t;
        if (size == items.length)
            throw new RuntimeException("queue overflow");
        items[index(b, items.length)] = item;
        bottom = b+1;
    }

    public T pop() { // from bottom
        final long b = bottom - 1, t = top.get(), afterSize = b - t;
        bottom = b;
        if (afterSize < 0) { // empty before call

```

```

        bottom = t;
        return null;
    } else {
        T result = items[index(b, items.length)];
        if (afterSize > 0) // non-empty after call
            return result;
        else { // became empty, update both top and bottom
            if (!top.compareAndSet(t, t+1)) // somebody stole result
                result = null;
            bottom = t+1;
            return result;
        }
    }
}

public T steal() { // from top
    final long b = bottom, t = top.get(), size = b - t;
    if (size <= 0)
        return null;
    else {
        T result = items[index(t, items.length)];
        if (top.compareAndSet(t, t+1))
            return result;
        else
            return null;
    }
}
}

```

## 9 Question 9

```

static void runTestChaseLevDeque() throws Exception {
    System.out.println("Running ChaseLevDeque Tests");
    ChaseLevDeque<Integer> c1 = new ChaseLevDeque<Integer>(100_000_000);
    sequentialDequeTest(c1);
    ChaseLevDeque<Integer> c12 = new ChaseLevDeque<Integer>(100_000_000);
    parallelCLDequeTest(c12, 10);
    System.out.println("ChaseLevDeque Tests Completed");
}

```

```

static void parallelCLDequeTest(Deque<Integer> queue, int threadCount) throws Exception {
    CyclicBarrier barrier = new CyclicBarrier(threadCount+2);
    int pushedSum = 0;
}

```

```

//Start pushing and popping thread
LongAdder pushed = new LongAdder();
LongAdder popped = new LongAdder();
new Thread()->{
    awaitBarrier(barrier);
    long p = 0;
    long pop = 0;
    for(int i = 0; i < 1_000_000; i++){
        Random random = new Random();
        if((random.nextInt() % 2) == 0){
            int r = random.nextInt() % 100;
            p += r;
            queue.push(r);
        }else{
            Integer pp = queue.pop();
            if(pp != null){
                pop += pp;
            }
        }
    }
    pushed.add(p);
    popped.add(pop);
    awaitBarrier(barrier);
}).start();

//Start stealing threads
LongAdder stolen = new LongAdder();
for(int t = 0; t < threadCount; t++){
    final int lt = t;
    new Thread()->{
        awaitBarrier(barrier);
        long s = 0;
        for(int i = 0; i < 1_000_000; i++){
            Integer p = queue.steal();
            if(p != null){
                s += p;
            }
        }
        stolen.add(s);
        awaitBarrier(barrier);
    }).start();
}

//Start test
awaitBarrier(barrier);

```

```

    //Wait for the test to stop
    awaitBarrier(barrier);

    //Get the remaining sum
    long remaining = 0;
    Integer p = queue.pop();
    while(p != null){
        remaining += p;
        p = queue.pop();
    }

    //Get the sum of the threads
    long pushedsum = pushed.sum();
    long retrievedsum = remaining + popped.sum() + stolen.sum();

    //Check that sum matches
    assertEquals(retrievedsum, pushedsum);
}

```

## 10 Question 10

Before:

37 32 25 34 0 14 10 39 12 25 30 2 36 1 37 1 12 1 11 33

After:

0 1 1 1 2 10 11 12 12 14 25 25 30 32 33 34 36 37 37 39

Sorted:

true

## 11 Question 11

Threads	Time
1	4.304853945
2	2.73936218
3	2.046592741
4	1.850278707
5	1.942862235
6	2.034372296
7	2.07867276
8	1.821243739

## References

## A Helper methods

```
public static void assertNull(Object o) throws Exception {
    if(o != null)
        throw new Exception(String.format("ERROR: assertNull"));
}

/**
 * What is up with this checked exception madness
 */
public static void awaitBarrier(CyclicBarrier c){
    try{
        c.await();
    }catch(Exception e){
        throw new RuntimeException(e);
    }
}
```