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* *syncronized*. 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 singleQueueMultiThread(final int threadCount) {
    System.out.println("Running singleQueueMultiThread");
    SimpleDeque<SortTask> queue = new SimpleDeque<SortTask>(100000);
    //int[] arr = IntArrayUtil.randomIntArray(size);
    System.out.println("Before:");
    int[] arr = IntArrayUtil.randomIntArray(20);

    IntArrayUtil.printout(arr, 20);

    queue.push(new SortTask(arr, 0, arr.length-1));
    sqmtWorkers(queue, threadCount);

    System.out.println("After:");
    IntArrayUtil.printout(arr, 20);

    System.out.println("Sorted:");
    System.out.println(IntArrayUtil.isSorted(arr));
}

/**
 * Function for starting singleQueueMultiThread workers
 */
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++){
        //Start the thread
        threads[t] = new Thread(()-> sqmtWorker(queue, ongoing));
        threads[t].start();
    }

    //Wait for the threads to finish
    for(int t = 0; t < threadCount; t++){
        try{
```

```
            threads[t].join();
        }catch(InterruptedException e){}
    }
}

/**
 * Function for a singleQueueMultiThread worker
 */
private static void sqmtWorker(Deque<SortTask> queue, LongAdder ongoing){
    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();
    }
}
```

Output:

```
Running singleQueueMultiThread
Before:
38 1 12 30 27 19 15 18 33 6 13 11 28 25 12 4 38 33 29 7
After:
1 4 6 7 11 12 12 13 15 18 19 25 27 28 29 30 33 33 38 38
Sorted:
true
```

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

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);
            sqmtWorker(queue, ongoing);
            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 multiQueueMultiThread(final int threadCount) {
    System.out.println("Running multiQueueMultiThread");
    SimpleDeque<SortTask>[] queues = new SimpleDeque[threadCount];
    for(int i = 0; i < threadCount; i++){
        queues[i] = new SimpleDeque<SortTask>(100000);
    }
    //int[] arr = IntArrayUtil.randomIntArray(size);
    System.out.println("Before:");
    int[] arr = IntArrayUtil.randomIntArray(20);
    IntArrayUtil.printout(arr, 20);
```

```
    queues[0].push(new SortTask(arr, 0, arr.length-1));
    mqmtWorkers(queues, threadCount);

    System.out.println("After:");
    IntArrayUtil.printout(arr, 20);

    System.out.println("Sorted:");
    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++){
        //Start worker thread
        final int myNumber = t;
        threads[t] = new Thread(()-> mqmtWorker(queues, myNumber, ongoing));
        threads[t].start();
    }

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

private static void mqmtWorker(Deque<SortTask>[] queues, int myNumber,
        LongAdder ongoing){
        SortTask task;
        while (null != (task = getTask(myNumber, 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[myNumber].push(new SortTask(arr, a, j));
                ongoing.increment();
                queues[myNumber].push(new SortTask(arr, i, b));
                ongoing.increment();
            }
            //We have sorted something, time to decrement
            ongoing.decrement();
        }
}

// 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++){
        SimpleDeque<SortTask>[] queues = new SimpleDeque[i];
        for(int t = 0; t < i; t++){
            queues[t] = new SimpleDeque<SortTask>(100000);
        }
        double time = mqmtBenchMarkVersion(i, queues);
        System.out.println(i + "\t" + time);
    }
}

private static double mqmtBenchMarkVersion(int threadCount,
    Deque<SortTask>[] queues) {
    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 myNumber = t;
        threads[t] = new Thread(()->{
            awaitBarrier(barrier);
            mqmtWorker(queues, myNumber, ongoing);
            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> cl = new ChaseLevDeque<Integer>(100_000_000);
    sequentialDequeTest(cl);
    ChaseLevDeque<Integer> cl2 = new ChaseLevDeque<Integer>(100_000_000);
    parallelCLDequeTest(cl2, 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

```
private static void multiQueueMultiThreadCL(final int threadCount) {
    System.out.println("Running multiQueueMultiThreadCL");
    ChaseLevDeque<SortTask>[] queues = new ChaseLevDeque[threadCount];
    for(int i = 0; i < threadCount; i++){
        queues[i] = new ChaseLevDeque<SortTask>(100000);
    }
    //int[] arr = IntArrayUtil.randomIntArray(size);
    System.out.println("Before:");
    int[] arr = IntArrayUtil.randomIntArray(20);
    IntArrayUtil.printout(arr, 20);


    queues[0].push(new SortTask(arr, 0, arr.length-1));
    mqmtWorkers(queues, threadCount);

    System.out.println("After:");
    IntArrayUtil.printout(arr, 20);

    System.out.println("Sorted:");
    System.out.println(IntArrayUtil.isSorted(arr));
```

```
}

Running multiQueueMultiThreadCL
Before:
15 0 15 31 6 26 38 29 12 38 38 16 27 20 7 20 23 13 10 24
After:
0 6 7 10 12 13 15 15 16 20 20 23 24 26 27 29 31 38 38 38
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

```
private static void benchMarkMultiCLQueueMultiThread() {
    System.out.println("Threads\tTime");
    for(int i = 1; i<9; i++){
        ChaseLevDeque<SortTask>[] queues = new ChaseLevDeque[i];
        for(int t = 0; t < i; t++){
            queues[t] = new ChaseLevDeque<SortTask>(100000);
        }
        double time = mqmtBenchMarkVersion(i, queues);
        System.out.println(i + "\t" + time);
    }
}
```

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);
    }
}
```