Homework 3

Hanlin He[*], Lizhong Zhang[†]

December 3, 2018

Chapter 5

Exercise 54

Since we have `peek()` method for queue, we can simply let each thread enqueue its thread ID after proposing the value, and then return value by which the first thread in the queue (`peek()` from queue) had proposed. Code are shown as follow:

```
1  public class QueueConsensus<T> extends ConsensusProtocol<T> {
2    Queue queue;
3    public QueueConsensus() {
4      queue = new Queue();
5    }
6    // figure out which thread was first
7    public T decide(T value) {
8      propose(value);
9      int i = ThreadID.get();
10     // enq current id.
11     queue.enq(i)
12     // return value of the first thread in the queue
13     return proposed[queue.peek()];
14   }
15 }
```

By doing so, we can allow any arbitrary large number of threads competing but only one thread will be return from `peek()`, and thus all threads will decide on the same value. Therefore the consensus number would be infinite.

[*] hxh160630@utdallas.edu
[†] lxz160730@utdallas.edu

Exercise 57

If we announced the thread's value after dequeuing from the queue, it is possible that the thread which '`WIN`' propose its value after the `LOSE` thread trying to return the `proposed[1-i]`, which would lead to a null return value.

Exercise 58

We can implement the binary consensus protocol using the StickyBit as follow:

```
1  public class StickyBitConsensus {
2    StickyBit sb;
3    public StickyBitConsensus() {
4      sb = new StickyBit();
5    }
6    public int decide(int value) {
7      sb.write(value)
8      // return value of the first thread performed the sb.write()
9      return sb.read()
10   }
11 }
```

This solution works because for any number of threads, only the first thread successfuly performed `sb.write()` would be able to change the value of `sb`. Then all thread will return the same value in `sb`.

Given m possible input, we can label all possible input with an integer from 0 to $m-1$. We can encode the integer in $\log_2 m$ bit and then code with an array of `StickyBit` as follow:

```
1  public class MStickyBitConsensus {
2    StickyBit[] sbits;
3    public StickyBitConsensus(int m) {
4      // suppose Math.log return ceiling of log.
5      sbits = new StickyBit[Math.log(m)];
6      for (int i = 0; i < sbits.length; i++)
7        sbits[i] = new StickyBit();
8    }
9
10   public boolean decide(int value) {
11     int i = sbits.length - 1;
12     while (value != 0) {
13       sbits[i].write(value & 1);
14       // if the thread failed to write current StickyBit
15       // there is no need to write to following StickyBits,
16       // because some other thread must have succeeded me.
17       if (sbits[i].read != (value & 1))
18         break;
```

```
19        value >> 1;
20        i--;
21      }
22      int ret = 0;
23      for (StickyBit sb : sbits) {
24        // if current StickyBit haven't been written yet, wait.
25        while (sb.read() != 1 || sb.read() != 0);
26        ret |= sb.read();
27        ret << 1;
28      }
29      return ret;
30    }
31  }
```

Chapter 9

Exercise 104

Suppose there are three thread A, B and C, a possible execution would be:

1. A located the window $pred \to curr$.
2. B located the window $pred \to curr$, locked the window, added x between $pred$ and $curr$, and then unlocked the window.
3. A locked the window, validation failed because $pred \nrightarrow curr$, unlocked the window and retried.
4. C located the window $pred \to x$, locked the window, removed x, and then unlocked the window.
5. A located the window $pred \to curr$ and previous steps happened again.

From this execution, A located the same window every time, but failed to move forward because B and C added/removed an element between $pred$ and $curr$, invalidating the windows located by A.

Exercise 110

If setting the next field of the node to be deleted to `null`, the linked list would be broken. Any other threads trying to traverse the linked list to a latter node would fail. Thus, it won't work by setting the next field to `null`.

Exercise 112

The reasoning only conisdered the remove scenario. It is still possible that the window $pred \to curr$ was modified by `add(x)` to $pred \to x \to curr$. In this scenario, validation should still fail because $pred \nrightarrow curr$.

Chapter 10

Exercise 122

It is necessary to hold the lock when checking that the queue is not empty.

Consider a queue with only one element. If we didn't lock the queue, it is possible that two threads are both able to pass `if (head.next == null)` check and proceed.

After the one thread *lock-process-unlock* the queue, the other thread will encounter `NullPointerException` when it tried to dereference `head.next.value` in line 18 `result = head.next.value;`

Exercise 125

Based on the implementation:

- The `enq()` operation is wait-free/lock-free, because both operations in `enq()` wait-free.
- The `deq()` operation is not wait-free, but lock-free.

  A possible scenario would be, when a thread get the range by `tail.get()`, some other threads get the same value and deque the value by invoking `items[i].getAndSet(null);`. Then the prevois thread will fail because all element it encountered would be `null` and the thread would retry.

  A thread could be overtaken by other threads as described above arbitrary number of times. Hence the `deq()` is not wait-free.

  However, as long as there are elements in the queue, some threads will always succeed in dequeue operation, since there are always some not-null value returned from `items[i].getAndSet(null)`. Therefore, `deq()` is lock-free.

Furthermore, the linearization point for `deq()` method would be the first time that `items[i].getAndSet(null)` return a not-null value.

The linearization point for `enq()` depends on the execution order of `enq()` and `deq()`:

- If all `enq()` invocations happened before all `deq()`, then the linearization point would be line 6:

```
6  int i = tail.getAndIncrement();
```

  Because the order of the elements dequeued would be solely depend on the index i returned from `tail.getAndIncrement()`.

- If some `deq()` invocations happens concurrently with the `enq()`, actual linearization point could be line 7:

```
7  items[i].set(x);
```

Because, `deq()` invocation could happen before all `enq()` finished setting the element in the array. During such scenario, the `deq()` will return the first not-null element it encountered.

A concrete example would be:

1. Thread A tried to `enq(1)`, while thread B tried to `enq(2)`.
2. Thread A got `i = 10`, while thread B got `i = 11`.
3. Thread B successfuly invoked `item[12].set(2)` before A.
4. Before thread A invoke `item[10].set(1)`, another thread C invoked `deq()`.
5. The first element thread C encountered would be `item[12]`, which is 2.
6. `C.deq()` would return 2.

Therefore the linearization point should be line 7.