Theory Assignment 3: Atomic Registers & Consensus
CO21BTECH11004

Que 4.5)
Sol:

True, If we replace the atomic SRSW registers with regular SRSW registers, then the construction still yields an atomic MRSW register.

As underlying registers are regular SRSW registers, the reader may return either the new value (just written different from the old) or the old value, i.e., no reader returns a value from the future, so condition (4.1.1) is satisfied.

Suppose writer thread T0 writes v with timestamp t. In that case, any subsequent read call by thread T1 (where T0's call completely precedes T1's) reads a maximum timestamp greater than equal to t, i.e., no read call returns a value from the distant past, that is, one that precedes the most recently written nonoverlapping value. Condition (4.1.2) is satisfied.

If a read call by thread T0 completely precedes a read call by thread T1, then T0 would update the column with timestamp t, that it is reading, and T1 would check his row and choose a timestamp greater than equal to t, thus satisfying the condition (4.1.3).

Since conditions 4.1.1, 4.1.2, and 4.1.3 are satisfied, this construction still yields an atomic MRSW register if we replace the atomic SRSW register with regular SRSW.

Que 4.8)
Sol:

Safe register is the strongest property that these 64-bit registers satisfy.

Here, writing on 32-bit memory is atomic, but we can't guarantee that for write operation on 64-bit register (containing two 32-bits).

Consider the case write overlaps with read, in that if the writer thread writes on the first block, and the reader thread now reads both blocks, then the writer thread writes on the second block. The reader thread reads a different value, which was then on the 64-bit register after and before writing. So, the reader thread can read any value in the range. Hence safe register.

Thus, none of the properties (4.1.1-4.1.3) are satisfied so, this construction can't be regular and atomic.

Que 5.12)
Sol a):
Binary consensus, for any number of threads, can be achieved in the following way:

```
// consider StickyBit object as per question
int decide(int value) {
     StickyBit.write(value);     // propose(value)
     return StickyBit.read();
}
```

Each thread calls write(value) and decides the value that it reads, which means return read(). This solves the problem because once a write occurs to the sticky bit, its value can't be changed (as per question), so all other subsequent writes would do nothing, so all threads will agree on the value written by the first thread.

This construction is wait-free, as every thread completes it's decision by calling a write and read, and these instructions are independent of other threads.

Sol b):

```
// consider StickyBits :- an array of log2m StickyBit objects
// StickyBits = new int[(log2m)], where log2m is  no of bits for m
// given each thread has a MRSW_atomic register
// threadRegister = new MRSWRegister[2*n], n is no of threaed for
storing its number and bit
// StickyBits & threadRegister array initialized to 0.
int decide(int value) {
      int v = value;  // propose(value)
      int id = ThreadId.get();
      threadRegister[n+id] = 1;
      threadRegister[id] = 1;
      int up = (int)log2(m)+1; // log2m bits for representing m
      for (int i=0;i<m;i++) {
          int bi = (v>>i)&1;
          StickyBits[i].write(bi);
          if (StickyBits[i].read() != bi) {
              // find the thread which has written
              for (int j=0;j<n;j++) {
                  if (threadRegister[i]) {
                     bool flag = true;
                     for (int k=0;k<=i;k++) {
                           if ((threadRegister[n+j]>>k)&1 !=
StickyBits[k]) {
                               flag = false;
                           }
                     }
                     if (flag) {
                        // set current thread value to the thread value
                        // that has written StickBits[i]
                        v = threadRegister[t+j];
                     }
                  }
              }
          }
      }
      // return integer that is formed by converting StickyBits from
      // binary to decimal
}
```

Let v be the binary representation of the value the thread wants to write. In a loop, each thread starts to set the StickyBits array of size (log2m). In the ith turn of loop, thread call StickyBits[i].write(ith bit of v). Now the thread reads, i.e., StickyBits[i].read. If this read is not equal to write, some other threads have written different values for that bit before the current thread. The current thread looks for the thread for which the ith bit of v gives the same value as read, sets its value to that thread value, and continues. When bits are set, return the number with binary representation, the same as the StickyBits array.

Validity: -
Proof by induction: -
StickyBits array of size 1, i.e., log2m = 1, implies m=2, binary consensus. A StickBit object would be used, and in part(a), we have shown that using one StickyBit object, we can solve binary consensus.
True of log2m = 1.

Assume using an array of StickBits objects of size k (log2m = k), we can solve the consensus problem for 2^k possible inputs.

To prove, using an array of StickyBits objects of size k+1 (log2m = k+1), we can solve the consensus problem for 2^(k+1) possible inputs.

Say i-th thread is at (k+1) bit, It would write the ((k+1)bit in v) and then read.
Now, two cases arrive:-

- read() returns the same value it was writing. As for k bits, the value was valid, and for (k+1)th bit it is the same as the value written. Then, v would be the value that all threads would decide.
- read() returns a different value from what it has written. Find the thread for which (k+1)th bit has the same value as read() and set v to that value. As for k bits value is valid, and for (k+1)th bit, we have made it valid. So, all threads will return the new set value of v.

This implies that using (k+1) bits, we can solve the consensus problem for 2^(k+1) inputs and give valid output.

So, by induction, an array of log2 m StickyBit objects can solve the consensus problem for m possible inputs and give a valid result.

Wait-free: -
Here, StickyBit methods write() and read(), and atomic MRSW are wait-free.
All loops used in the construction are bounded. Also, if any thread crashed, it would not affect other threads. So, this is wait-free construction.

Que 5.22)
Sol:
The second step is faulty step in the chain of reasoning.

- We can add a peek() simply by taking a snapshot of the queue (using the methods studied earlier) and returning the item at the head of the queue.

For implementing the peek() method, by taking a snapshot of the queue and returning the head which is a whole different problem of taking an atomic snapshot of a single MRMW object.

Also, for this queue implementation, the snapshot doesn't tell the actual state of the queue. Here, the enqueue method is not atomic. Consider the case when thread T0 calls enqueue and increment head by calling getAndIncreamnet(). Now thread T1 comes, increments head, and writes in the item array and peak is called. Then T0 completes writing and peak is called. So, there would be different values returned in peak calls. So, the consensus problem is not solved.