Distributed Computing
Spring 2009: Solutions to Assignment No. 3

Due Date: 3.31.09

April 21, 2009

Exercise 1: In Fig. 1 we present class *SafeBoolMRSWRegister* which implements a safe Boolean MRSW register, using an array of SRSW safe registers.

```
public class SafeBoolMRSWRegister implements Register < Boolean > {
  boolean[ ] s_table;  // array of safe SRSW registers
  public SafeBoolMRSWRegister(int capacity) {
    s_table = new boolean[capacity];
  }
  public Boolean read() {
    return s_table[ThreadId.get()];
  }
  public void write(Boolean x) {
    for (int i = 0; i < s_table.length; i++)
      s_table[i] = x;
  }
}
```

Figure 1: Class *SafeBoolMRSWRegister*: a safe Boolean MRSW register

True or false: if we replace the safe Boolean SRSW register array with an array of safe M-valued SRSW registers, then the construction yields a safe M-valued MRSW. Justify your answer.

Solution 1: *True:* If we replace the safe Boolean SRSW register array with an array of safe M-valued SRSW registers, then the construction does yield a safe M-valued MRSW register.

The proof is almost exactly the same as for the case of the safe Boolean MRSW register. For non-overlapping method calls, each $s\_table[i]$ holds the most recently written value, which is returned by the next *read*( ) call. For overlapping method calls, the reader may return any value, because the component registers are safe. ∎

Exercise 2: Consider again the class presented in Fig. 1. True or false: if we replace the safe Boolean SRSW register array with an array of regular Boolean SRSW registers, then the construction yields a regular Boolean MRSW register. Justify your answer.

Solution 2: *True:* If we replace the safe Boolean SRSW register array with an array of regular Boolean SRSW registers, then the construction does yield a regular Boolean MRSW register.

The proof is almost exactly the same as for the case of the safe Boolean MRSW register. For non-overlapping method calls, each $s\_table[i]$ holds the most recently written value, which is returned by the next $read(\ )$ call. For overlapping method calls, the reader may return either the new value or the old value, because the component registers are regular. ∎

Exercise 3: Consider the atomic MRSW construction presented in Fig. 2. This class implements an atomic MRSW register by an array of atomic SRSW registers. You may consult Subsection 4.2.5 in the textbook for a detailed explanation of the construction.

True or false: if we replace the atomic SRSW registers with regular SRSW registers, then the construction still yields an atomic MRSW register. Justify your answer.

Solution 3: *True:* If we replace the atomic SRSW registers with regular SRSW registers, then the construction still yields an atomic MRSW register.

We follow the textbook and present three conditions that characterize regular and atomic registers.

4.1.1 : It is never the case that $R^i \to W^i$.
4.1.2 : It is never the case that $W^i \to W^j \to R^i$.
4.1.3 : If $R^i \to R^j$ then $i \leq j$.

According to the book, a register is regular iff all of its behaviors satisfy conditions 4.1.1 and 4.1.2. It is atomic iff, in addition, it also satisfies Condition 4.1.3. In Claim 1 of the appendix, we prove that these three conditions imply atomicity (linearizability).

We show that the version of the algorithm of Fig. 2 in which the SRSW registers are all regular (instead of atomic) satisfies the three conditions listed above.

First, we observe that no reader can return a value from the future, so Condition 4.1.1 is clearly satisfied.

Consider a situation in which $W_a^i \to W_b^j \to R_c^i$. By the construction $i < j$. Since W_b^j terminates before Thread c attempts to execute R_C^i, we have that $a\_table[c, c] = j > i$. When Thread c reads $a\_table[c, c]$, this reading does not overlap with the writing. Hence, Thread c will read the time stamp $j > i$. It follows that Condition 4.1.2 also holds.

Finally, if we have the situation $R_a^i \to R_b^j$, then Thread b will read from $a\_table[a, b]$ a time stamp which is at least i. It follows that $i \leq j$. ∎

```
public class AtomicMRSWRegister <T> implements Register <T> {
  ThreadLocal <Long> lastStamp;
  private StampedValue <T> [ ][ ] a_table;  // each entry is SRSW atomic
  public AtomicMRSWRegister(T init, int readers) {
    lastStamp = new ThreadLocal <Long> () {
      protected Long initialValue() { return 0; };
    };
    a_table = (StampedValue <T> [ ][ ]) new StampedValue[readers][readers];
    StampedValue <T> value = new StampedValue <T> (init);
    for (int i = 0; i < readers; i++) {
      for (int j = 0; j < readers; j++) {
        a_table[i][j] = value;
  } } }
  public T read() {
    int me = ThreadId.get();
    StampedValue <T> value = a_table[me][me];
    for (int i = 0; i < a_table.length; i++) {
      value = StampedValue.max(value, a_table[i][me]);
    }
    for (int i = 0; i < a_table.length; i++) {
      if (i ≠ me) a_table[me][i] = value;
    }
    return value;
  }
  public void write(T v) {
    long stamp = lastStamp.get() + 1;
    lastStamp.set(stamp);
    StampedValue <T> value = new StampedValue <T> (stamp, v);
    for (int i = 0; i < a_table.length; i++) {
      a_table[i][i] = value;
} } }
```

Figure 2: Class *AtomicMRSWRegister*: an atomic MRSW register constructed from atomic SRSW registers

Exercise 4: Recall that a quiescently-consistent register is such that every complete execution can be transformed into a sequential execution by a permutation of the invocations that does not reorder two invocations that are separated by a quiescent period. Give an example of a quiescently-consistent register that is not regular.

Solution 4: In Fig. 3 we present a quiescently-consistent behavior that cannot be generated by a regular register.

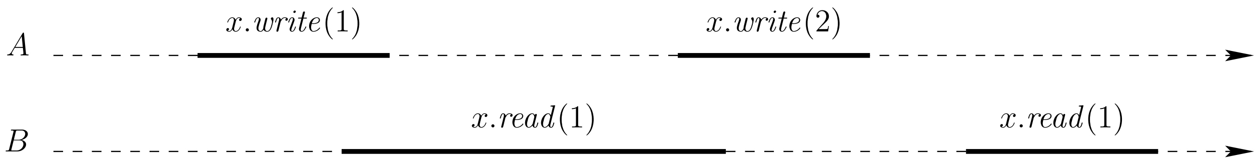

Figure 3: A quiescently consistent but not regular execution

Exercise 5: Reconsider the class presented in Fig. 1. True or false: if we replace the safe Boolean SRSW register array with an array of *regular* M-valued SRSW registers, then the construction yields a regular M-valued MRSW register. Justify your answer.

Solution 5: *True:* If a *read*( ) call by Thread j does not overlap with a *write*( ) call, then the value retrieved would be the value most recently written. On the other hand, assume that the previously written value was v_0 and a *read*( ) call by Thread j overlaps a *write*(v_1) call. If the execution of Line 7 does not overlap with the execution of Line 11 with $i = j$, then if $i < j$ the "read" will return v_0, and if $i > j$, the "read" will return v_1. If Line 7 overlaps with the execution of Line 11 with $i = j$, then the value retrieved will be v_0 or v_1, according to the properties of regular registers.

Exercise 6: In Fig. 4 we present a class that implements a regular Boolean MRSW register using a safe Boolean MRSW register.

True or false: if we replace the safe Boolean MRSW register with a safe M-valued MRSW register, then the construction yields a regular M-valued MRSW register. Justify your answer.

Solution 6: *False:* If we replace the safe Boolean MRSW register of Fig. 4 with a safe M-valued MRSW register, then the construction does not yield a regular M-valued MRSW register.

If the execution of Line 16 overlaps the execution of Line 12, we may get an arbitrary value in the range $0..M-1$, not necessarily the new or old value.

Exercise 7: Does Peterson's two-thread mutual exclusion algorithm work if we replace shared atomic registers with regular registers?

```
public class RegBoolMRSWRegister implements Register<Boolean> {
  ThreadLocal<Boolean> last;
  boolean s_value;  // safe Boolean MRSW register
  RegBoolMRSWRegister(int capacity) {
    last = new ThreadLocal<Boolean>() {
      protected Boolean initialValue() { return false; };
    };
  }
  public void write(Boolean x) {
    if (x ≠ last.get()) {
      last.set(x);
      s_value = x;
    };
  }
  public Boolean read() {
    return s_value;
  }
}
```

Figure 4: Class *RegBoolMRSWRegister*: a regular Boolean MRSW register constructed from a safe Boolean SRSW register

Solution 7: Yes, Peterson's algorithm will work with regular registers. Where could problems arise? One possibility is if Thread i tests the value of *flag*$[j]$ precisely when Thread j sets *flag*$[j]$ to **true**. Due to regularity, Thread i may get both **true** and **false** as possible results. However, this is similar to the situation that Thread i checks *flag*$[j]$ when Thread j is at Lines 9 or 7, respectively. Since both of these situations lead to correct behavior, this particular overlap does not cause any undesired behavior.

A similar situation is if the testing of the condition $victim == i$ by Thread i overlaps the execution of Line 9 by Thread j. However, this corresponds to Thread i testing $victim == i$ while Thread j is at lines 8 or 10, respectively.

It follows that both of these situations correspond to states that arise in the behavior of Peterson's algorithm in the case of atomic registers. ▪

Exercise 8: Consider the following implementation of a Register in a distributed message-passing system. There are n processors $P_0, \ldots, P_{n-1}$ arranged in a ring, where P_i can send message only to $P_{i+1 \bmod n}$. Messages are delivered in FIFO order along each link. Each processor keeps a copy of the shared register.

- To read a register, the processor reads the copy in its local memory.
- A processor P_i starts a *write*() call of value v to register x, by sending the message "P_i : write v to x" to $P_{i+1 \bmod n}$.

- If P_i receives a message "P_j : write v to x" for $j \neq i$, then it writes v to its local copy of x, and forwards the message to $P_{i+1 \, mod \, n}$.
- If P_i receives a message "P_i : write v to x", then it writes v to its local copy of x, and discards the message. The *write*() call is now complete.

Give a short justification or counterexample.

A. If *write*() calls never overlap, then

   (a) Is this register implementation regular?

   (b) Is it atomic?

B. If multiple processors call *write*(), then

   (a) Is this register implementation atomic?

Solution 8.A.a: When only a single writer exists, the ring register is regular. If one processor reads its local register while another processor's write is in progress, then it will see either the new or the old value, depending on how far around the ring the write message has propagated.

Solution 8.A.b: The ring register is not atomic even under the assumption of a single writer. Processor P_0 starts writing, and processor P_1 updates its local register. The next two method calls overlap the write but do not overlap one another.

1. P_1 reads the new value.
2. P_2 reads the old value.

This history is not linearizable because P_2's call must be linearized after P_1's, but P_2 saw the old value.

Solution 8.B.a: Adding more writers will not cause the algorithm to become atomic. ■

Exercise 9: We considered safe and regular registers in these lectures. Define a *wraparound* register that has the property that there is a value v such that adding 1 to v yields 0, not $v + 1$.

If we replace the shared variables *label*[i] in the Bakery algorithm with either (a) safe registers, (b) regular registers, or (c) wraparound registers, then does it still satisfy (1) Mutual Exclusion, (2) Starvation Freedom?

You should provide six answers (some may imply others). Justify each claim.

	Mutual Exclusion	Starvation Freedom (FCFS)
Safe Registers	No	Yes
Regular Registers	Yes	Yes
Wraparound	No	No

Figure 5: Properties satisfied by various versions of the Bakery Algorithm

Solution 9: The summary of the various cases is presented in the table of Fig. 5. FCFS here means "First Come First Serve. If A finishes its doorway before B enters its doorway, then B cannot overtake A and enter the critical section ahead of A.

Violation of mutual exclusion for safe registers may occur if B reads $label[A]$ at the same time that A write to it. B decides that $label[A] > label[B]$ and enters the critical section. In fact, $label[B] > label[A]$, and B also enters the critical section.

FCFS holds. If A finishes its doorway before B enters its own doorway, then $(label[A], A) \prec (label[B], B)$. B's read of $label[A]$ does not overlap A's write, so B reads the correct value, take a larger label and the is blocked until A leaves the critical section.

We show that Bakery with regular registers satisfies mutual exclusion. Assume that B entered the critical section ahead of A and that $(label[A], A) \prec (label[B], B)$. Thread B must have concluded that $flag[A] = 0$ or that $(label[B], B) \prec (label[A], A)$.

If B read that $flag[A] = 0$, then the read preceded or overlapped A's write to $flag[A]$, which preceded A's write to $label[A]$, implying that $label[A] > label[B]$ – a contradiction.

So B must have (falsely) observed that $(label[B], B) \prec (label[A], A)$. Since A never wrote such a value, B must have read $label[A]$ at the time A was updating it. But B must have seen either the value being written, or the previous value, both of which are less than or equal to $label[B]$ – a contradiction.

Wraparound registers do not provide either FCFS or mutual exclusion. It fails to be FCFS because the label written by a later doorway is not necessarily larger than the label it read. Essentially, the same counterexample as the safe register case shows that the lock fails to provide mutual exclusion. ∎

Appendix: Sufficient Conditions for Realizability

Reconsider the three conditions characterizing regular and atomic registers:

4.1.1 : It is never the case that $R^i \to W^i$.
4.1.2 : It is never the case that $W^i \to W^j \to R^i$.
4.1.3 : If $R^i \to R^j$ then $i \leq j$.

The following claim states that these three conditions guarantee that an implementation of a register is atomic.

Claim 1 *If all behaviors of a concurrent register satisfy Conditions* 4.1.1–4.1.3*, then this implementation is atomic.*

Proof:

Assume that we have a history α that satisfies Conditions 4.1.1–4.1.3. We show that we can construct a permutation π that respect the ordering of events in α and forms a legal sequential computation of a register.

The permutation π is constructed as follows:

First, place in π all the "write" events according to the order of their occurrence in α. Then, we place in α the "read" actions, proceeding according to the order of appearance of their response events in α. Place each R^i immediately before W^{i+1}. If there is no W^{i+1}, place R^i at the end of of π. In Lemma 2 we show that the permutation π respects the ordering determined by α.

Q.E.D.

Lemma 2 *Let op_1 and op_2 be two invocations in α, such that $op_1 \to op_2$ in α. Then $op_1 \prec op_2$ in π.*

Proof:

By construction, the real-time order of write operations is preserved.

- Consider $R^i \to W^j$ in α. By Condition 4.1.1 and Condition 4.1.2 it follows that $i < j$. Because otherwise $i > j$, and there would be $W^j \to W^i$ leading (by transitivity) to $R^i \to W^i$ contradicting Condition 4.1.1. We claim that $R^i \prec W^j$ in π, because by the construction R^i was placed before $W^{i+1} \preceq W^j$.

- Next, consider $W^j \to R^i$ in α. By Condition 4.1.2 we cannot have $i < j$ because this would lead to $W^i \to W^j \to R^i$. It follows that $i \geq j$. By the construction, R^i was placed in π in the interval between W^i and W^{i+1}. Consequently, $W^j \preceq W^i \prec R^i$ in π, from which we can infer $W^j \prec R^i$.

- Finally, consider $R^i \to R^j$. By Condition 4.1.3, $i \leq j$. Consider first the case that $i < j$. By the construction it follows that $R^i \prec W^{i+1} \preceq W^j \prec R^j \prec W^{j+1}$, from which we can conclude that $R^i \prec R^j$. In the case that $i = j$, we have $R^i_a \to R^i_b$. Since read operations with the same index are placed in π in their order of appearance in α, it follows that $R^i_a \prec R^i_b$ in π

Q.E.D.