Registers

Introduction

A register is one of the simplest objects available to us. It contains a single value and can perform read() and write() operations. Though it is a fairly simple object, there exist many different types of it, here are different possibilities:

1. What values can be stored in the register? (Boolean or Multi-valued)

2. How many processes are allowed to read and write in the register? The possibilities are: Single Reader Single Writer (SRSW) (only one process can read and only one process can write to the register), Multiple Reader Single Writer (MRSW) (any process can read but only one process can write to the register) and Multiple Reader Multiple Writer (any process can read and write).

3. What is the behavior of the register under concurrent access? Here we have 3 possible types:

    - Safe register: guarantees only that a read() will return the last value written if no concurrent read/write occurred. If a concurrent read/write occurred, all bets are off and the register is free to return any value.

    - Regular register: stronger "contract" than a safe register. Under concurrent read/write the read() operation must return the value of the register before the write() operation or the value currently being written to the register.

    - Atomic register: The strongest "contract" is the atomic register. Every operation can be viewed as happening at a single point in time. We will illustrate these concepts with plenty of examples later.

Goals

In this chapter our goal is to build wait free MRMW multi-valued atomic registers from SWSR binary safe registers, it comes down to implementing the read and write operations for the atomic MRMW register in terms of the read and write operations of safe SWSR registers. Remember that wait free condition implies that any call of read() or write() should return no matter the state of other processes. This excludes any kind of classical concurrent methods like mutual exclusion. Clearly mutual exclusion violates wait free condition: simply imagine two processes A and B, A acquires a lock and crashes, B is stuck if it tries to acquire the same lock as A, thus the calls of B may never return.

Now that we know the goals and tools we can use, let us start transforming binary SRSW registers to more complex registers. Through this section we will refer to the operations of lower level register as read() and write() and higher level register operations as Read() and Write().

Binary SRSW safe → binary MRSW safe

```
Snippet 1

Reg[1..N]

Read()
  return(Reg[i].read())
Write(v)
  for j = 1 to N
    Reg[j].write(v)
```

The basic idea is to use an array of SRSW register to simulate multiple reader behavior. Here is a proposed algorithm in pseudo code given in Snippet 1.

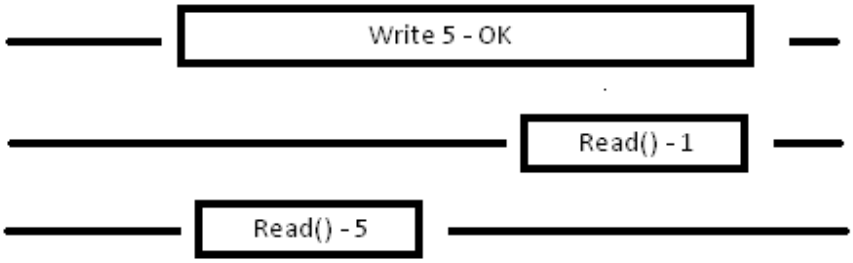
The produced register is MRSW because any number of processes can read the value written by the writer (the value is simply written to N distinct registers). It is also safe.

We never imposed a constraint on the values inside the registers, so this implementation would also work for multi-valued registers.

We now argue that this implementation produces regular MRSW registers from SRSW regular registers. We need to look at the behavior of our implementation under concurrent accesses. It is clear that our SRSW registers contain only two possible values: the newly written value or the previous value (during the write operation some may still contain the old value). So any Read() to a register not under the write operation will return either the new or the previous value, which is fine for regular registers. A read() operation on the register under write() operation is also fine because the SRSW registers are regular and this will return one of the two accepted values.

Let us show now that this transformation cannot produce atomic registers even if we suppose that we have SRSW atomic registers. Here is an execution which could occur: we perform a Read() on the first and the Nth register while the write is still updating the N registers. The first Read() will return the new value, while the second one will return the old value.

Figure 1. Inconsistent execution.



There is clearly no way to place the linearization point on figure 1 for the write operation for this execution to be consistent.

Binary MRSW safe → Binary MRSW regular

The idea is to exploit to the maximum the regular property and the fact that only binary values are allowed. The pseudo code for this transformation is presented in Snippet 2. We are writing only when necessary, this implies that when a write is performed, the acceptable values for the read() are 0 and 1 (the previous value of the register is different from the new one, and because the registers are binary any value for the read() is accepted). So even if we have a concurrent read/write on a safe register, the returned value still satisfies the regular property.

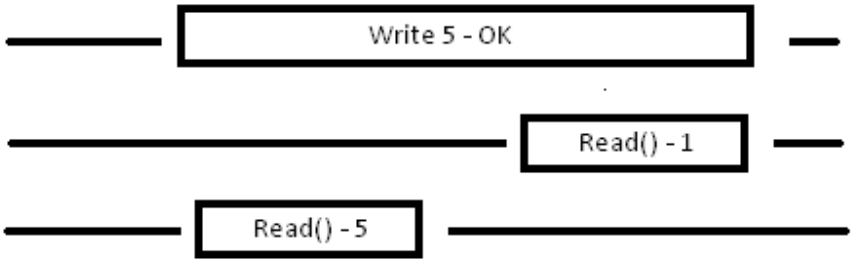
Snippet 2

```
Read()
  return(Reg.read());
Write(v)
  if old ≠ v then
   Reg.write(v);
   old := v;
```

We restricted the possible values for the registers to binary, this transformation thus doesn't work for multi-valued registers for obvious reasons (the safe register can return anything while only 2 values are accepted).

This transformation cannot produce atomic MRSW registers from MRSW safe registers. A concurrent read/write can return 0 or 1 arbitrarily, which is fine for regular registers, but is not acceptable for atomic ones. Here is a possible execution with one write and two reads.

Figure 2. Inconsistent execution.



Again there is no way to place the linearization point on the figure 2 for the write operation for this execution to be consistent.

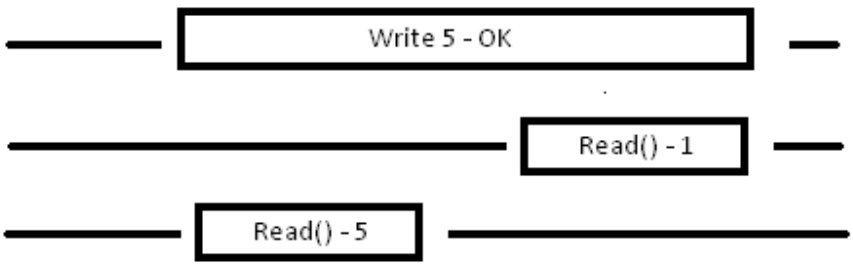
Binary MRSW regular →M-valued MRSW regular

The idea is to represent a value using M binary MRSW registers. The Read() operation search the first 1 in the array, when we find it, we return the index. The Write() operation is more tricky: we will write a 1 at the desired position and start cleaning the array **IN THE INVERSE** order starting at the newly written 1. This ensures that the register contains either the old value (if it is smaller than the new value and the Write() operation hasn't cleaned it yet) or the new value (if the new value is smaller than the old one or the Write() has already cleaned the old value). Thus this register satisfies the regular property.

```
Snippet 3

Reg[0,1,..,M] init to [1,0,..,0]

Read()

  for j = 0 to M

    if Reg[j].read() = 1 then
return(j)

Write(v)

 Reg[v].write(1);

 for j=v-1 downto 0

  Reg[j].write(0);
```

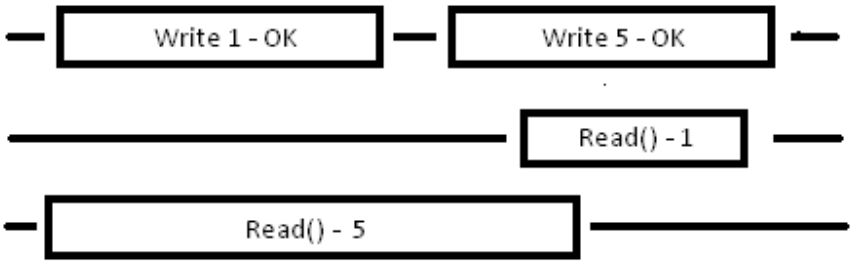
Again, this transformation would not produce atomic registers, even if the low level registers are atomic. Let us try to imagine a scenario where the read/write inversion is present first for regular, then for atomic low level registers. For regular low level registers, the scenario could be the following: we start off with our registers initiated to value 1. We write value 5, and the write start cleaning. When the writer is changing the register 1, the reader starts reading it. Because the register is regular it can return 1 or 0. Suppose it returns 0, thus the first read will return the value 5. Suppose the writer is slow, and the second reader also hits the register 1, again it can receive a 1 or a 0, suppose 1 is returned. The results are presented in Figure 3. We observe the read/write inversion, thus the register produced is not atomic.

Figure 3. Inconsistent execution.



This argument won't hold if the low level registers are atomic (we heavily relied on the regular property of the low level registers in the previous argument). We imagine a different scenario for this case: we start with a reader that starts reading from left to right; the first writer writes to a register that reader 1 has already read. The second writer writes to a register that reader 1 hasn't read yet, we suppose that writer 2 is slow for cleaning and will not clean the register that writer 1 modified until reader 2 is done. The second reader will now read the value written by writer 1 (because it wasn't cleaned yet by writer 2), and reader 1 will return the value written by writer 2, because it "missed" the value written by writer 1. The results are presented in Figure 4.

Figure 4. Read/Write inversion.



SRSW regular → SRSW atomic

```
Snippet 4
Local variables : x,t
Regular register : Reg
Read()
  (t',x') = Reg.read()
  If t' > t then t = t'; x = x'
  Return x
Write(v)
  t = t + 1
  Reg.write(v,t)
```

As seen in previous sections, the main difference between regular and atomic registers is read/write inversion. In order to counter this problem we will use a timestamp for the writer. The code for the Read and Write operations is presented in the Snippet 4. This time, the reader will check the time stamp and will return a value if it is older than the one it has read previously.

This algorithm will not work for multiple readers if we just use the naïve approach of using one register per reader, reader 1 could read the new value and reader N could read the old value. We already encountered this behavior in SRSW safe to MRSW transformation, but because the register we were building was safe, it was not a problem. Now we need to address it in some way that will preserve atomicity.

SRSW atomic → MRSW atomic

The basic idea is to make the readers help the writer, and delay the reading until EVERY reader will read the new value. The code is presented in the Snippet 5.

```
Snippet 5

NxN SRSW registers: RReg

N SRSW registers: WReg

Read()
  for j = 1 to N do
    (t[j],x[j]) = RReg[i,j].read()
  (t[0],x[0]) = WReg[i].read()
  (t,x) := highest(t[..],x[..])
    for j = 1 to N do
      RReg[j,i].write(t,x)
   return(x)
Write(v)
  t1 = t1 + 1
  for j = 1 to N
    WReg.write(v,t1)
```

Several explanations are necessary. RReg(i,j) represents a register for communication between readers i and j, the writer of this register is j, and the reader is i. A register i in WReg is read by the process i, and is naturally written by the writer. The highest function simply searches for the largest timestamp. The Write() operation writes to all WReg registers the value and a timestamp. The Read() operation is more complex. We first read what other readers have written for us (RReg read), then we read what the original writer has written for us (WReg.read). We choose the highest timestamp from the reads we performed earlier and propagate the change to other readers (RReg write). This way, when a Read completes, EVERY reader is aware of the latest value (Because of the write to RReg). This approach ensures that any subsequent read will never read a value with an older timestamp. Thus we have ensured that no read/write inversions are possible with this implementation.

This implementation will however not work with multiple writers, because the timestamp is local to a writer process. We need some way of having a global timestamp between writers.

MRSW atomic → MRMW atomic

```
Snippet 5
N MRSW registers: Reg
Read()
  for j = 1 to N do
    (t[j],x[j]) = Reg[j].read()
  (t,x) := highest(t[..],x[..])
  return(x)
Write(v)
  for j = 1 to N do
    (t[j],x[j]) = Reg[j].read()
  (t,x) := highest(t[..],x[..])
  t = t + 1
  Reg[i].write(v)
```

The major change from before is that now, the timestamp is distributed among the writers. The technique used is similar to what we used with the readers in MRSW atomic registers: the writers communicate through N registers and increment the highest timestamp at every write. The read operation reads all the writer's registers (remember that we used MRSW registers as low level registers). There is no possibility of read/write inversion because the reader will always choose the highest timestamp, and the latest write will always have the highest timestamp. Note however that we need some way to reset the timestamp, because it grows indefinitely otherwise.

Conclusion

During this chapter, we learned how to construct an atomic MRMW register from SRSW safe registers. Many transformations and resources were necessary. What would be interesting is to calculate how many SRSW safe registers are necessary in order to implement a single MRMW atomic register.