Concurrent Algorithms October 10, 2018

Solutions to Exercise 2

Problem 1. Omitted.

Problem 2. Hint: notice that if the writer first clears the array by writing 0's, it is possible for the value of the array to be all 0's, which is not a valid state. An example execution omitted.

Problem 3. The transformation does not work for multiple readers (the result is not an atomic register). The non-atomic execution in Figure 1 is possible in this case. Since the register is regular, the read by R1 may read the value 2 being concurrently written by W. Since this is R1's first read operation, the timestamp it obtains for value 2 is higher than its local timestamp. Later, the read by R2 (also concurrent with *Write(2)*) may read the previous value of the register (1). Since this is R2's first read operation, the timestamp it obtains for value 1 is also higher than its local timestamp.

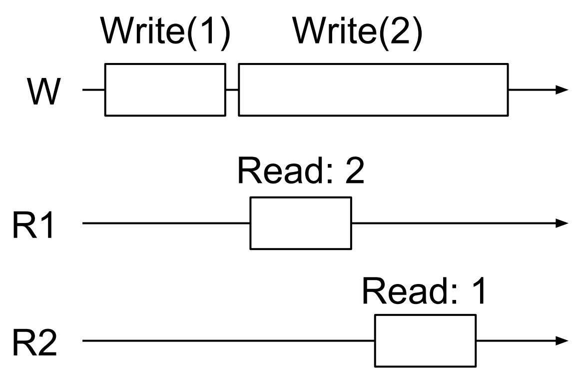

Figure 1: An execution that is possible with a regular register but not with an atomic register. There are three processes: a writer (W) and two readers (R1 and R2). The two reads are concurrent with the second write. The read by R1 completely precedes the read by R2. The execution is not atomic because it is impossible to assign linearization points to all operations: if the linearization point of *Write(2)* is before that of the read by R1, then the read by R2 cannot have a linearization point; if the linearization point of *Write(2)* is after that of the read by R1, then that read cannot have a linearization point.

Problem 4. Given that the *splitter* will be called concurrently by a number of N threads, we can think about this as selecting 1 thread to return *stop*. All the threads arriving during this election but not chosen to return *stop* can return *left*, and the ones arriving after the election can return *right*. It is acceptable to not have any threads selected to get *stop* (e.g., in case more than 1 thread executes *splitter*), but it must never be possible to have more than 1 thread return *stop* during a concurrent execution.

We use two registers:

- P (multi-valued), and
- S (binary, initialized to *false*)

P holds the *id* of the thread that should get *stop*. S marks whether a *stop* thread has been selected. When a thread calls *splitter*, it needs to check whether S is *false*, and if so, set it to *true* and return *stop*. The difficulty is that we cannot use atomic *getAndSet*-type primitives, so multiple threads first reading the value of S and then updating it could mistakenly think they each got *stop*. In order to decide which thread should get *stop*, each thread volunteers itself by setting the value of P to their own id. The last thread to update P wins.

After volunteering, a thread checks the S flag, and if it is *true*, then the thread knows it arrived after the election, and so it gets *right*. If S still *false*, then the thread (one of potentially many) arrived during the election, so it sets S to *true*, and checks if it won (i.e., if the value of P is equal to its own id). If the thread won, it simply gets *stop*. Otherwise, it means some other thread managed to change P after it, hence the current thread lost and gets *left*.

It is possible that a thread updates P and becomes the winner just as another thread sets S to *true*, but before checking to see if it won. In this case, 0 threads get *stop*, as the winner then reads S, finds it *true*, concludes it arrived after the election, and gets *right*.

However, it is impossible for more than 1 thread to get *stop*. Assume by way of contradiction that 2 threads with identifiers i and j both return *stop*. Furthermore, assume without loss of generality that thread i first performed the read of P and then thread j read P. Therefore, the order of events will be $read_i(P = i) \to read_j(P = j)$ (i.e., since both threads return *stop* they read their own identifier when reading from P). We furthermore know that both threads write register P at the beginning of their execution and since both threads return *stop* they read S to be *false*. So we have the following ordering of events:

- $write_i(P \leftarrow i) \to read_i(S = false) \to write_i(S \leftarrow true) \to read_i(P = i)$.
- $write_j(P \leftarrow j) \to read_j(S = false) \to write_j(S \leftarrow true) \to read_j(P = j)$.

Since thread i read $P = i$ (and thread j read $P = j$) it means that $write_j(P \leftarrow j)$ takes place after $read_i(P = i)$. So we have:

- $write_i(P \leftarrow i) \to read_i(S = false) \to write_i(S \leftarrow true) \to read_i(P = i) \to write_j(P = j) \to read_j(S = false)$.

This is a contradiction, since thread i wrote *true* to S and then j read *false* from S.

upon $splitter_i$

```
P ← i;
if S then return "right";
S ← true;
if P = i then return "stop";
return "left";
```

Algorithm 1: Sample implementation of the *splitter* object.

Problem 5.

Algorithm 2 presents the pseudocode of an atomic wait-free snapshot as described in class. For a program running N threads, in order to run a *scan* or a *collect* operation, all the registers of the N threads need to be read. Writes are done only on a thread's register $R[i]$. Since we know beforehand that many of the N threads will not use the snapshot, a better solution is to assign registers to threads on demand.

We assume that there exists an *obtain*() operation that each thread can call to get a register that is assigned only to itself. Algorithm 3 presents the implementation of *update*() and *scan*() using the aforementioned operation. Importantly, the number of registers that need to be parsed now in *scan*() is dependent on the number of threads that have written to the object (and thus have been assigned a register).

```
upon scan_i
    t1 ← collect(), t2 ← t1;
    while true do
        t3 ← collect();
        if t3 = t2 then return ⟨ t3[1].val, ..., t3[N].val ⟩ ;
        for k ← 1 to N do
            if t3[k].ts ≥ t1[k].ts + 2 then return t3[k].snapshot ;
        t2 ← t3;

procedure collect()
    for k ← 1 to N do
        x[k] ← R[k];
    return x;

procedure update_i(v)
    ts ← ts + 1;
    snapshot ← scan();
    R[i] ← ⟨ ts, v, snapshot ⟩;
```

Algorithm 2: Sample implementation of a non-adaptive snapshot. Each thread has its own register.

```
procedure update(v)
    if myreg = ⊥ then
        myreg ← obtain();
    ts ← ts + 1;
    snapshot ← scan();
    R[myreg] ← ⟨ ts, v, snapshot ⟩;

upon scan_i
    t1 ← collect(), t2 ← t1;
    while true do
        t3 ← collect();
        if t3 = t2 then return ⟨ t3[1].val, ..., t3[t3.length].val ⟩ ;
        for k ← 1 to t3.length do
            if t3[k].ts ≥ t1[k].ts + 2 then return t3[k].snapshot ;
        t2 ← t3;
```

Algorithm 3: Sample implementation of *update*() and *scan*() in an adaptive snapshot. Each thread that affects the snapshot calls *obtain*() to get assigned a register.

Implementing *obtain*()

Recall the *splitter* object implemented in the previous exercise: it allows selecting at most 1 thread out of multiple accessing the object concurrently, while partitioning the remaining threads into 2 separate pools (*left*, *right*). Keeping this in mind, we create a matrix of *registers* and *splitters*, as presented in figure 2. A thread calling *obtain*() starts from the top-left corner and calls the *splitter* in that cell. If it gets *stop*, then it obtains that register. Otherwise, it moves 1 column to the *right*, or 1 row downwards for *left*, and repeats the process.

procedure *obtain*()
  $x \leftarrow 1, y \leftarrow 1;$
  while *true* **do**
    $s \leftarrow S[x,y].splitter();$
    if $s =$ *"stop"* **then return** $R[x,y];$
    else if $s =$ *"left"* **then** $y \leftarrow y+1;$
    else $x \leftarrow x+1;$

Algorithm 4: Implementation of *obtain*() using a matrix of registers and *splitter* objects.

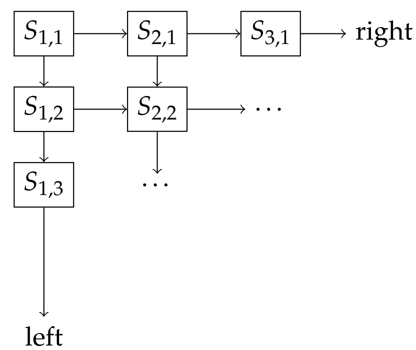

Figure 2: Matrix of registers and *splitter* objects.

Implementing *collect*()

Finally, we need to adapt the *collect*() call to the matrix of registers now being used. The insight here is that all the registers that have been assigned from each matrix diagonal that has had <u>at least</u> 1 splitter used need to be taken into account.

procedure *collect*()
  $C \leftarrow \langle\,\rangle;$
  $d \leftarrow 1;$
  while *diagonal d has a splitter that has been traversed* **do**
    $C \leftarrow C \cdot \langle$ values of all non-$\perp$ registers on diagonal $d\,\rangle;$
    $d \leftarrow d+1;$
  return $C;$

Algorithm 5: Implementation of *collect*() using a matrix of registers and *splitter* objects.

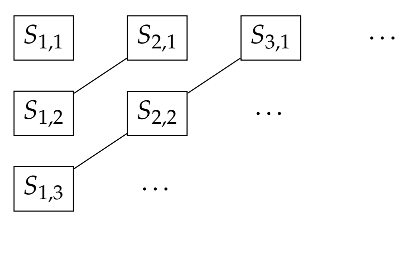

Figure 3: Matrix of registers and *splitter* objects.