THE POWER AND THE LIMITATIONS OF REGISTERS

INTRODUCTION

In the previous lectures we have learnt that, if the hardware doesn't give us powerful atomic wait-free tools, we can build them by ourselves. Concretely we have seen in class how to implement different types of registers with different features and properties. We have been able to build a MRMW multi-valued atomic register from SRSW binary safe registers.

Now, following the same scheme, we wonder what other kind of structures we can build with these powerful tools. We can reformulate the question in an even more general way: Can we build any kind of structure using registers?

WHAT CAN WE IMPLEMENT?

COUNTER

Let´s start with a simple structure which is the counter. We can define a counter as an object that has two main operations:

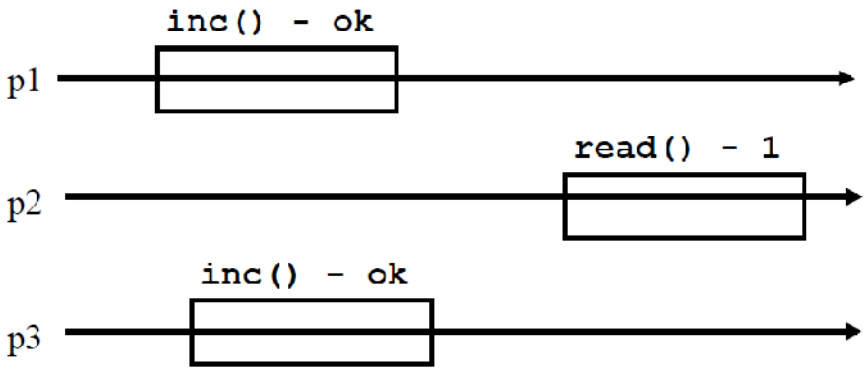
- Read() = Returns the current value of the counter
- Inc() = Increments the value stored by the counter and returns "ok"

A first approach to implement a counter using only registers could be sharing a register and implementing both operations by accessing that register as follows.

```
read():
1  return(Reg.read())
inc():
1  temp:= Reg.read()+1;
2  Reg.write(temp);
3  return(ok)
```

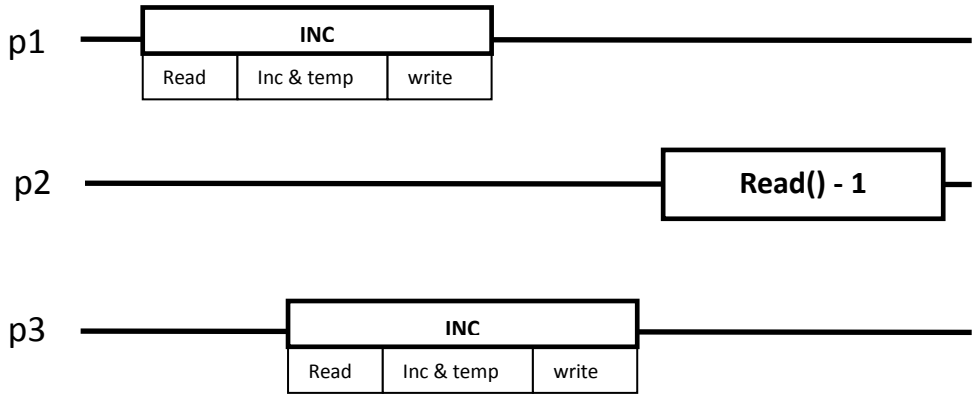
This implementation could be a possibility but we can easily check that it doesn´t respect the atomicity. Why? Let´s try to prove it by giving an execution in which we cannot establish a linearization point.

Imagine a situation in which we have two concurrent processes that try to increment the value by invoking the operation inc() and a third process that reads the value of the counter after the concurrent increments have finished.



We assume that the initial value of the counter is 0 so the reader should receive the value 2 but this is not the case. The reader gets the value 1 as a result of two increments from 0. Let´s try to explain the result with a possible sequence of setps and the corresponding diagram

1) p1 starts the inc() operation. It reads the value of the register in the statement 1 and start to increment and store it in the temp variable.
2) p3 starts the inc() while p1 is storing the value of the incremented value in the temp variable. p3 gets the value of the register (***still 0***) in the statement 1.
3) p1 writes the value of the temp variable (1) inside the register in the statement 2. Now the value stored in the shared register is 1.
4) p3 writes the value of its temp variable (1) inside the register in the statement 2. The value of the register is overwritten with 1!!
5) Both, p1 and p3, returns ok and later p2 starts reading the value of the counter. p2 receives the value 1 as a result.



Now that we have proved that this implementation is not atomic, let´s think about other one. Let´s try to do it with an array of registers (one register per process) in which each process increments its own register when they want to increment. We assume that the array is called Reg[1,..,n] where n is the number of processes.

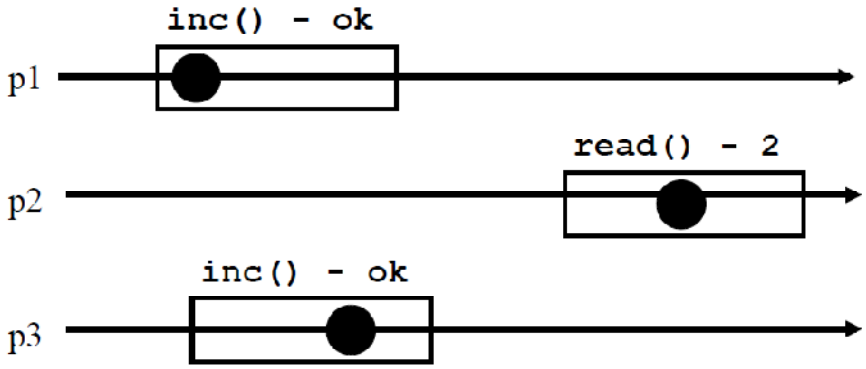
```
inc():
1 Reg[i].write(Reg[i].read() +1);
2 return(ok)

read():
1 sum := 0;
2 for j = 1 to n do
2.1     sum := sum + Reg[j].read();
3  return(sum)
```

Note that in this implementation the reader must sum the value of all the registers from the array. It is done in the "for" loop in the statement 2.

Each register stores the number of increments done by the corresponding process.

With this implementation we can always find a linearization point because each register can only be modified by one

process so the situation that we had in the previous implementation will never exist. The following diagram shows the linearization point established for the situation described before.



SNAPSHOT

Now that we have implemented a counter with atomic registers, let´s try to implement a more complex structure such as the snapshot. The snapshot is an object that represents an array of values and has two main operations:

- Update(I,v): changes the state of the object. In this case, it writes the value v into the register i.
- Scan(): returns the current state of the whole structure, in this case the values of all the registers.

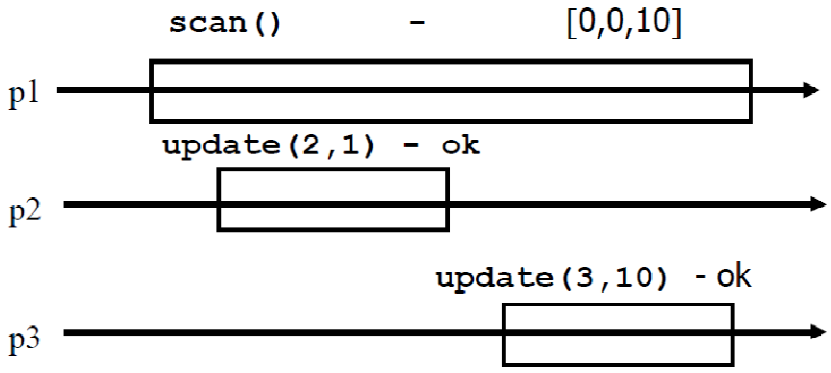
There is a quite easy implementation which we can think of. We can try to implement it with a local array of registers per process but we can easily check that it is not valid since each process updates only its own array. When a process scans the snapshot, it doesn´t receive the modifications done by other processes. The right way to do it may be using shared registers.

We can think about using a shared array of registers and implement the operations like follows.

```
scan():
1 for j = 1 to N do
1.1    x[j] := Reg[j].read();
2 return(x)
update(i,v):
1 Reg[i].write(v); return(ok)
```

Note that the update operation consists only of writing the value in the corresponding register and scan one consists only in recollecting the values from all the registers and returning them.

It seems to be a good implementation of a wait-free snapshot but it is not an atomic one. Let´s see the atomicity violation with the following diagram.

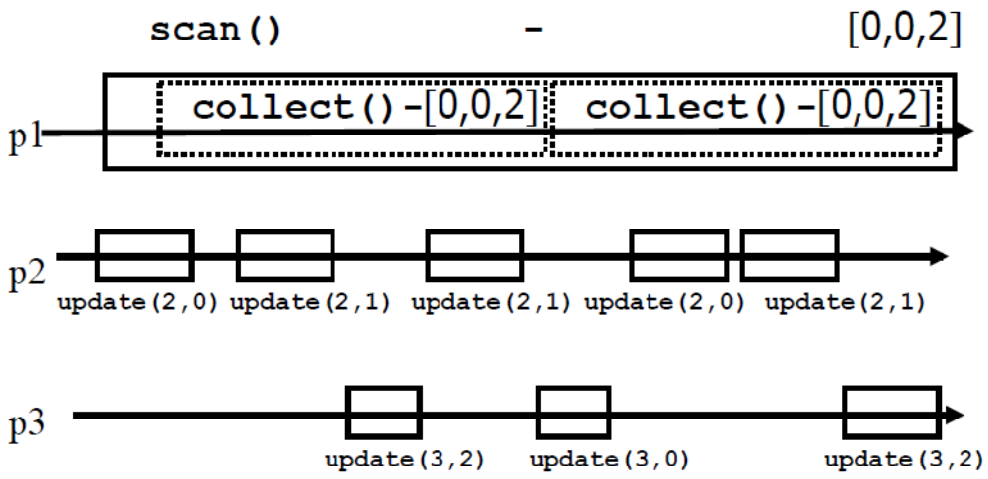


In the situation that we can see over these lines the scan returns the wrong values while it occurs concurrently with two update operations. If we analyze it carefully we can understand what is happening. The scan starts and when it collects the value of the second register the first update has not modified it yet so it gets the old value (assuming that the registers were initialized with value 0). Then, when the scan collects the value from the third register, both updates, the one of p2 and the one of p3, have finished so p1 receives the new

value of the third register. As a result the snapshot returns a mixture of old and new values which can be regular but not atomic.

Let´s try to solve the problem of the atomicity. From now, the scan algorithm that we have used is going to be renamed as “collect”. We will implement the scan operation by doing several collects. The idea is finding two equal consecutive collects. Then, we could say that we have reached a stable situation and we can return the collect.

Despite the fact that it may not be a wait-free operation, we have to consider another situation represented in the following graph in which we can find two consecutive collects that are equal but there have been changes between them.



There is a way of solving this problem that is using timestamps. One possible implementation could be the following.

```
scan():
1 temp1 := self.collect();
2 while(true) do
2.1  temp2 := self.collect();
2.2  temp1 := temp2;
2.3  if (temp1 = temp2) then
       return (temp1.val)
update(i,v):
1 ts := ts + 1;
2 Reg[i].write(v,ts);
3 return(ok)
```

Note that each process has a local timestamp that is incremented in each update.

Now, we are sure that this is the implementation of an atomic snapshot but we have the problem that we mentioned before. This is not a wait-free snapshot.

Imagine that there is someone (a process) that updates the snapshot very fast repeatedly and another process that wants to perform a scan operation. The process that wants to do a scan has to do at least two collects to check if there have not been changes between them but it never ends because the updater modify the structure faster, so the one who performs the scan never finds two equal collects.

Let´s try to solve this problem with the following idea. To avoid waiting infinitely for a stable snapshot, we will make the updaters save a snapshot too. It will be like asking the fastest to save the snapshot before finishing so the processes that want to scan only have to look for two equal consecutive collects or for the last snapshot that has happened between them.

As before, we will use an array of registers but now each register will store the corresponding value, the timestamp and a complete snapshot done by the corresponding process in its last update operation. The implementation is the following.

```
update(i,v):
1 ts := ts + 1;
2 Reg[i].write(v,ts,self.scan());
3 return(ok)

scan():
1 t1 := self.collect();
2  t2:= t1;
3 while(true) do
3.1  t3:= self.collect();
3.2  if (t3 = t2) then
        return (t3[1..N,1]);
3.3 for j = 1 to N do
3.3.1  if(t3[j,2] ≥ t1[j,2]+2) then
          return (t3[j,3])
3.4  t2 := t3
```

Note that in each component of the array there are three fields: value (1), timestamp (2) and snapshot (3).

In the update operation each process has to store the value, the timestamp (still local for each process) and the full snapshot values given by the scan operation.

In the statement 3.2 of the scan operation, the process checks if there has been any changes between the two collects (t2 and t3). If there has not any change the values are returned (all the components 1), otherwise the recent snapshot is searched in the “for” statement 3.3.

Note that in the statement 3.3.1 we check for a timestamp incremented by 2. This is because we must be sure that if there is a concurrent update with the scan operation and it finishes before the scan operation, we take the new changes in account. With this implementation we do not have the problem mentioned before of the wait-free violation.

WHAT WE CANNOT IMPLEMENT ONLY WITH REGISTERS

We´ve seen in the previous pages that we can implement different structures using registers combined in some cases with timestamps. But, what can we implement using ‘only’ registers?

We can think about different structures like queues, Fetch&Inc, Test&Set, Comp&Swa, etc. These are very powerful objects that we can try to implement with registers. The main problem is what we have seen before while we were designing a counter or a snapshot. After

implementing them, we have to check conscientiously the validity of the structure and it is not always easy. In this section we will use a shortcut to prove that we can (or we cannot) implement a structure using only registers.

The main idea will be that if we know that we cannot implement the consensus between two or more processes using only register and we can do it with a given structure, there will be a contradiction, so, that structure cannot be built using only registers.

Let´s remember firstly the specification of the consensus.

- A consensus has one operation **propose()** that returns a value. When a process receives a value v from this operation we say that the process has decided the value v.
- No two processes decide differently.
- A decided value is a proposed one.

In 1985, Fisher, Lynch and Patterson published an article in which they proved that consensus among any number of processes cannot be implemented using only registers. We will use this proposition to prove by contradiction that some structures cannot be implemented with registers because if we can do it, we would be able to implement the consensus. Let´s see how we can implement the consensus using different structures.

CONSENSUS BETWEEEN 2 PROCESSES USING A QUEUE

Shared objects

R_0, R_1 : Registers

Q : Queue

propose(v_i)

```
1 Ri.write(vi)
2 item := Q.dequeue()
3 if item = winner
      return(vi)
4 return(R1-i.read())
```

In this example we build the operation propose using a queue. A queue is an object that has two operations: **queue(v)** that puts the value v in the last position of the queue and **dequeue()** that returns the value stored in the first position of the queue.

If we assume that we initialize the queue with values {winner,looser} and we are using wait-free registers and queues, then we can easily check that this is a wait-free valid implementation since it satisfies all the conditions of the consensus and there is no wait statement in the implementation.

CONSENSUS BETWEEN 2 PROCESSES USING FETCH&INC

Shared objects

R_0, R_1 : *Registers*

F : fetch&inc

propose(v_i)

```
1 Ri.write(vi)
2 val := F.fetch&inc()
3 if(val = 1) then
3.1   return(vI)
  else
3.2   return(R1-i.read())
```

Here, the consensus operation is implemented using fetch&inc. This structure has only one operation that returns the stored value incremented by one and at the same time it stores that incremented value.

Assuming that the structure is initialized with the value 0 and following the reasoning carried in the previous implementation we can check that this implementation is wait-free and valid.

CONSENSUS BETWEEN 2 PROCESSES USING TEST&SET

Shared objects

R_0, R_1 : *Registers*

T : Test&Set

propose(v_i)

```
1 Ri.write(vi)
2 val := T.test&set()
3 if(val = 0) then
3.1     return(vI)
  else
3.2     return(R1-i.read())
```

In this case, the **propose()** operation is implemented with test&set that is a binary structure that holds one bit. It has only one operation (**test&set()**, hence, the name of the object) that returns the value of that bit (0 or 1) and set the value of that bit to 1.

Assuming that the test&set is initialized to 0 and following again the same reasoning than before we can assure that this is also a wait-free implementation that satisfies all the conditions established by the consensus specifications.

CONSENSUS BETWEEN N PROCESSES USING COMP&SWAP

Shared objects

C : Compare&Swap

propose(v_i)

1 val:= C.c&s($\perp$,v_i)
2 if(val = $\perp$) then
2.1 return(v_i)
else
2.2 return(val)

Finally we can easily implement also the consensus between more than 2 processes using com&swap. This structure has only one operation (**c&s(v1,v2)**) that compares the stored value with v1 and if they are the same v2 is stored instead. This operation always returns the value stored in the structure before the operation.

If we assume that the stored initial value is $\perp$, we can check again that the consensus is well implemented with this structure and it doesn´t violate the wait-free condition.

We have seen how to implement consensus among two processes with different structures and among more than two with C&S. As aforementioned we can use the proposition proved by Lynch, Fisher and Patterson to prove by contradiction that these structures cannot be implemented using only registers but we need to prove the proposition.

Proposition: *There is no algorithm that implements consensus among two processes using only registers.*

If we prove this for two processes we can extend it to any number of processes. We will try to prove by contradiction but firstly we have to give several definitions that are going to be used through the proof. We will assume that there is an algorithm A that implements consensus between two processes p1 and p2.

We say that a ***configuration*** is a state of the system in one instant. This state is composed by the state of each process and the registers that are used in the algorithm.

We obtain a new configuration by executing a ***step***. A step is the minimal execution unit that changes something in the configuration.

The processes collaborate to carry out the consensus against the ***adversary***. The adversary tries to confuse the processes to make them not achieve a consensus situation. We will play the role of the adversary to proof the proposition. The adversary will decide which process is going to execute one step, in other words, the order of step executions.

We say that a ***schedule*** is a sequence of process identifiers (let´s say for example p1, p2, p3…,pN) that establishes the order of steps executions. For example a schedule p1|p1|p1|p1|p2 means that p1 executes 4 steps consecutively and then p2 executes one. The goal of the adversary is setting up a schedule in order to make them fail with their aim.

Let “u” be 0 or 1, we say that a configuration C is “u”-valent if, no matter what happens between the processes, starting from C, “u” is the only decision possible. Otherwise we say that the configuration is bivalent, in other words, 0 or 1 can be decided.

Now, that we have the necessary elements and definitions we will divide the proof in two parts by formulating two lemmas and proving them.

Lemma 1: *There exists at least one initial bivalent configuration for the given algorithm A.*

Lemma 2: *There exists an arbitrary long schedule that, from a given bivalent configuration, leads to another bivalent configuration.*

Assuming that these two lemmas are correct, we can easily see that there is an infinite sequence of steps (schedule) from a bivalent configuration (that exists by lemma 1) and leads to another bivalent configuration, in other words, from a given state of the systems at least one process (or more) takes an infinite number of steps and doesn´t decide. It is a strong contradiction with the consensus validity because every process eventually decides a value and it proves that A cannot solve the consensus problem satisfying the wait-free condition.

Now if we prove these lemmas separately the job would be finished.

Lemma 1 proof: Let´s assume that we have two processes p1 and p2 and an initial configuration represented as C(A,B) where A is going to be the value that the processes will decide if p1 proposes first and B the value that the processes will decide if p2 proposes first. We are going to prove that C(0,1) is a bivalent configuration.

We start from C(0,0) and keep p2 stopped, so it doesn´t take any step. If the only one taking steps is p1, the value proposed and decided will be 0. p1 cannot distinguish from C(0,0) and C(0,1) because under the same conditions (p2 stopped) p1 will decide always 0. So from C(0,1) p1 will also decide 0.

With a similar reasoning, starting from C(1,1) and keeping p1 stopped, p2 is the only one that takes steps and at the end is going to decide 1. As before, p2 cannot distinguish from C(0,1) and C(1,1) because if p1 is stopped, p2 will always decide 1. So, from C(0,1) p2 will always decide 1.

Thus, from C(0,1), and *only depending on the schedule*, the processes can decide 0 or 1. Then the configuration C(0,1) is bivalent. Lemma 1 is proved because there is at least one bivalent configuration. Actually we can use the same procedure to prove that C(1,0) is also bivalent.

Lemma 2 proof: We are going to prove this lemma by contradiction. Let´s assume that S is the longest possible schedule such as, from a given initial bivalent configuration C, S(C) is bivalent but if one of the processes takes a step it becomes univalent.

We are going to use the following representation pX(K) to say that the process pX takes a step from the configuration K. We are going to denote D as the bivalent configuration that we get after taking all the steps from the aforementioned schedule S, in other words, D is the last possible bivalent configuration.

Now, let p0(D) 0-valent and p1(D) 1-valent, so they are univalent but different. Note that this can be a possible case and, if we reach a contradiction with this case, we will prove the lemma.

We must remember that we are using only registers to implement the consensus with the algorithm A so taking a step, in other words going from D to pX(D), means that something is changing in the configuration, so, something is changing in one register. There are two possible operations with registers, read() and write(), but only write() modifies something. Thus, to change the state of the system, that is going from D to pX(D), the step taken must be a write access. Otherwise, the configuration obtained from pX(D) would still be bivalent.

Apart from that we must take in account that both processes are accessing (modifying) the same register because if this is not the case, p0(p1(D)) is the same that p1(p0(D)) and one of them is 0-valent while the other is 1-valent. This is a contradiction.

Finally, if we know that both processes are writing in the same register R we can see that for example p1(p0(D)) should be 0-valent but no matter what p0 does with R, p1 is going to overwrite the register and p1(p0(D)) will be the same than p1(D) so it will be 1-valent. This is the contradiction because p1 eventually decides 1 in both cases so p0(D) cannot be 0-valent. We can use the same argument to prove that p0(p1(D)) is the same that p0(D).

CONCLUSION

We have learnt that we can implement different structures with registers but usually we have to use other things apart from them like local timestamps. In the first part of the current lesson we have experienced the difficulties of implementing a specific object using registers and the problems that we have to satisfy all the conditions given including atomicity and wait-freedom.

In the last section of the lesson we have discovered a very useful way of proving that a given structure cannot be implemented using only registers. It´s true that it doesn´t give us the implementation of the structure but it is an easy way to discard trying to implement it using only registers. In other words, if we can implement consensus with the structure and at the same time we can implement the structure only with registers there is something wrong with one of both implementations.