Lecturer: Nir Shavit

Handout 7

The following homework assignment is due in two weeks.

1 Second Homework Assignment

1. What is the consensus number of the MIT *Alewife* machine's *(full/empty)-bit* register operation. Such a register has in addition to the value an additional *full/empty* field. It allows, in addition to simple reads and writes, a *write* on the condition that the bit is *empty*, setting it to *full*, and a *read* on the condition that it is *full*, setting it to *empty*. Prove your claim.

2. What is the consensus number of a *compare&swap* operation on a register r if r is *boolean*. What if r is three valued (has values in $\{0, 1, 2\}$)?

3. The following is the *gs* object, shared among two processes P and Q. Each process can perform a *label* operation once and a *read* operation many times. The sequential automaton specification of the object is as follows. It has 4 states, an initial one s_0, from which it moves to state s_1 if P performs a label, and from which it moves to state s_2 if Q performs a label. From s_1 it moves to s_2 if Q performs a label operation. From s_2 it moves to s_3 if P performs a label operation. All read operations return the current state without causing a transition. What is the consensus number of the *gs* object? Prove your claim.

4. What is the consensus number of *k-set consensus* object defined as follows: each process P_i has an input X_i from a set $V = \{0, 1, \ldots, v\}$, and is required to return a *valid* output $Y_i \in V$ such that the collective set of decided values $Y = \{Y_i \mid i \leq n\}$ is of size at most k. Note that for $k = 1$ this is just consensus. Prove your claim.

5. Two process *approximate agreement* for a given ϵ is the following object. Given two processes P and Q, each can perform a $decide(x_p)$ or $decide(x_q)$ operation, where x_p and x_q are real numbers. The operations return a values y_p and y_q respectively, such that y_p, y_q are in the interval $[min(x_p, x_q), max(x_p, x_q)]$, and $|y_p - y_q| \leq \epsilon$ for $\epsilon > 0$. What is the consensus number of the *approximate agreement* object? Prove your claim.

2 The Relative Power of Synchronization Operations

In Lecture 6 we saw the *snapshot* primitive, which had 2 operations: Update and Scan. We saw a wait-free implementation of it using atomic registers.

We now introduce a new primitive - the *Multiple Assignment* primitive, which has 2 operations: Scan and Update, just like a snapshot, yet each update operation by process i can update atomically a *subset* of chosen locations, not just the ith one.

We are trying to answer the following question:

> Given two concurrent objects X and Y, is there a wait free implementation of X by Y ?

Up to now we have seen the following implementations of X from Y, where by implementation we mean a wait-free implementation:

1. X is regular and Y is safe.
2. X is atomic and Y is regular.
3. X is multi-writer atomic and Y is single-writer atomic.
4. X is a snapshot and Y is single-writer atomic.

Is it possible to implement X from Y when:

1. X is a multiple assignment snapshot and Y a single write atomic register?
2. X is a FIFO queue and Y is a R/W register?

3 Impossibility Results

We will prove, for some cases, that there is no wait free implementation of an object X by Y.

3.1 Consensus Numbers

consensus numbers

Informally, a *consensus protocol* is a system of n processes that communicate through a set of shared objects $X_1, ..., X_n$. Each process starts with an input value from some domain D, they communicate by applying operations on the shared objects, and they eventually agree on a common output value and halt.

A consensus protocol is required to be :

1. *consistent*: distinct processes always decide on the same value.
2. *wait-free*: each process decides after a finite number of steps.
3. *valid*: the common decision value is an input of at least one of the processes.

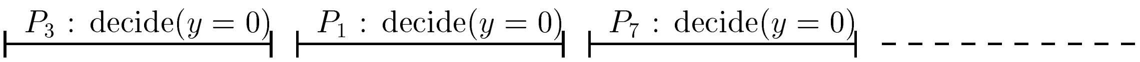

Figure 1: Execution where first to decide returns $y = 0$.

We will express the consensus problem as an abstract object with a sequential implementation where each process can perform one type of operation. See Figure **??**.

Definition A *consensus object* is an n process object, where each process has a single operation *decide*(*input*) *returns*(*value*). A protocol's sequential specification is such that every *decide* operation is an execution returns the value of the first decide.

A linearizable implementation of a consensus object is a *consensus protocol.*

Definition An object X *solves* n-process consensus if there exist a consensus protocol which uses a collection of objects of type X and a set W of atomic registers.

Definition The *consensus number* of an object X is the largest n for which X solves n-process consensus. If no largest n exists, the consensus number is said to be infinite.

Corollary .1 *If there is an implementation of X by Y, and X solves n-consensus then Y solves n-consensus.*

3.2 The consensus number of read/write registers

In this section we show that there exists no two-process consensus protocol using R/W atomic registers.

Definition A protocol state is *bivalent* in a given execution if either decision value is still possible, that is, the execution can be extended to yield different decision values. Otherwise it is *univalent.* An *x-valent* state is a univalent state with eventual decision value x.

Definition A *decision step* is an operation that carries a protocol from a bivalent state to a univalent state.

Theorem .2 *Read/Write registers have consensus number 1.*

Proof: Assume there exists a two-process consensus protocol implemented from atomic read/write registers. We derive a contradiction by constructing a sequential execution that keeps any such protocol in a bivalent state. If the processes have different input values, the validity condition implies that the initial state is bivalent. Consider the following sequential execution, starting from the initial state: In the first stage, P executes a sequence of operations until it reaches a state where the next operation will leave the protocol in a univalent state. P must eventually reach such a state, since it cannot run forever, and it cannot block. In the second stage, Q executes a sequence of operations until it reaches a similar state, and in successive stages, P and Q alternate sequences of operations until each is about to make a decision step. Such a step exists since the implementation is wait-free (see Figure **??**).

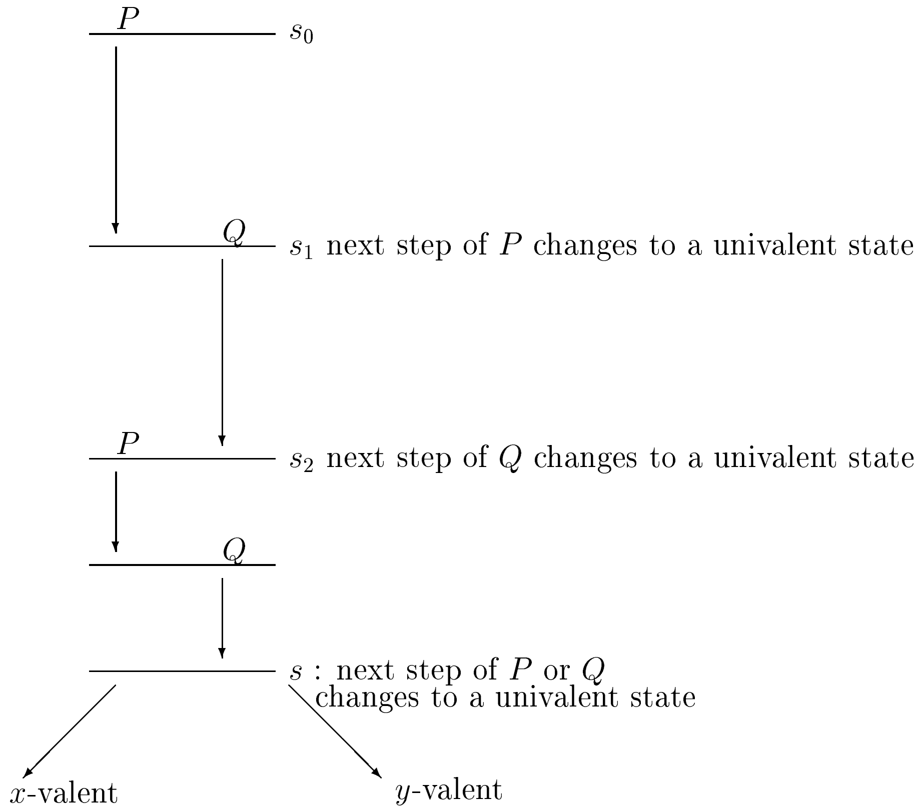

Figure 2: Reaching a critical state s

Suppose P's operation carries the protocol to an x-valent state, and Q's operation carries the protocol to a y-valent state, where x and y are distinct.

Let s be the bivalent state from which a step of either process is a decision step.

There are 3 kinds of operations which can carry the protocol to a univalent state.

1. A process (say P) reads a register (r).
2. The processes P and Q write to different registers (r and r').
3. The processes P and Q write to the same register (r).

We now go through the case analysis to derive a contradiction:

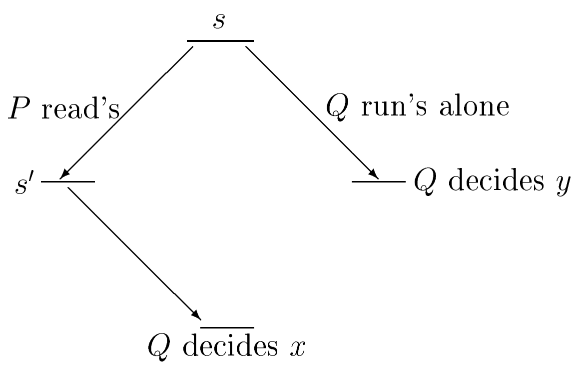

Figure 3: Case of P reading register r

1. Suppose the decision step for a process, say P, is to read a shared register (Figure ??).

   Let's call the state after the read s'. s' is an X-valent state (since P has already decided on X). If we let Q run from s' then it will decide X. On the other hand, if we let Q run from s, it will reach a Y-valent state. Since the states s and s' differ only in the internal state of P, Q should have reached the same decision in both cases, leading to a contradiction.

2. Suppose the processes write to different registers(Figure ??).

   The state that results if P's write is immediately followed by Q's, is identical to the state that results if the writes occur in the reverse order, which leads to a contradiction since s' is x-valent and s'' is y-valent.

3. Suppose the processes write to the same register (Figure ??).

   Let s' be the x-valent state immediately after P's write. There exists a history fragment starting from s' consisting entirely of operations of P that yields the decision value x. Let s'' be the y-valent state reached if Q's write is immediately followed by P's. Because P overwrites the value written by Q, s' and s'' differ only in the internal state of Q and therefore the protocol has the same history fragment starting from s'', an impossibility since s'' is y-valent.

■

Corollary .3 *It is impossible to construct a wait free implementation of any object with consensus number greater than 1 using atomic r/w registers.*

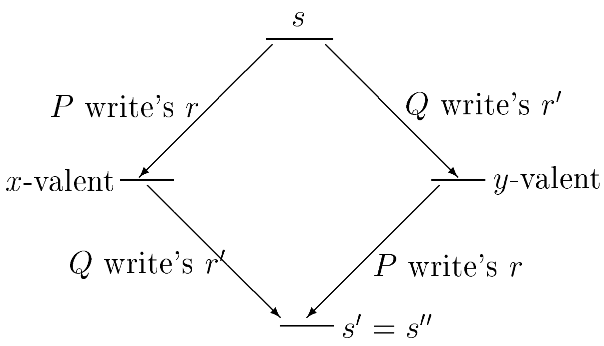

Figure 4: Case of two writes to registers r and r'

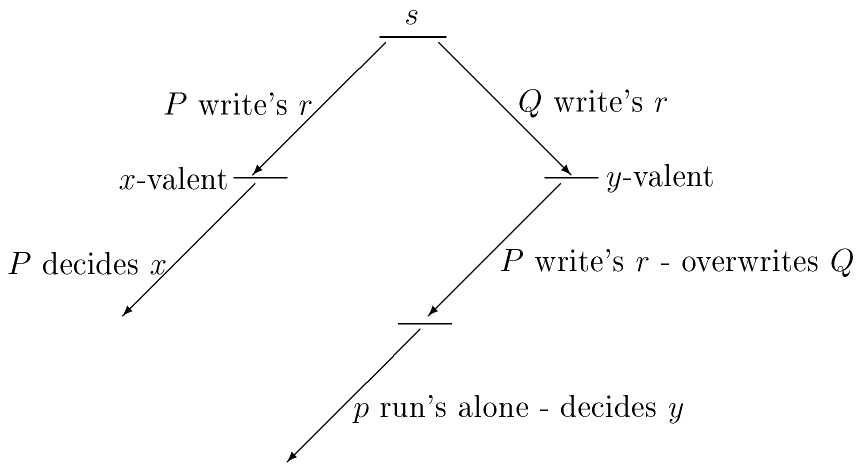

Figure 5: Case of two writes to a single shared register r

3.3 Multiple assignment

Theorem .4 *There is no wait free implementation of multiple-assignment by atomic registers.*

Proof: We will show a 2-consensus protocol implementation using multiple assignment, so according to the previous corollary, we conclude that there is no implementation of multiple assignment by atomic registers. The idea is that the processes return the value of the first process that performed an *update* operation on a given location r of the multiple assignment register. P and Q will be able to detect the first process who wrote by analyzing the returned value of their *scan* operation. The code of the protocol appears in Figure **??** (for Q's protocol we switch 1 and 3).

```
DECIDE(input) returns(value);
   update(v[1,2]) := (input, input);
   u := scan(v);
   if (u[3] = null)or(u[2] = u[3])
      then return (u[1])
      else  return (u[3])
      end if
   end DECIDE
```

Figure 6: Code of DECIDE with multiple assignment for P_i

Let us look at P's operations. In order to determine who performed the first *update* operation on v, P performs a scan operation on v. If $u[3] = null$, then from linearizability, Q's *update* was performed *after* P's scan, so P can decide its own value - $u[1]$. If $u[2] = u[3]$, then Q must have overwrite the value of $v[2]$, so again P decides it's own value - $u[1]$. Otherwise P was the last to update v, so P decides the value of Q - $u[3]$. The same arguments hold for Q.

■

4 Read Modify Write (RMW) Operations

Kruskal, Rudolph and Snir (see bibliography) have observed that many, if not all, of the classical synchronization primitives can be expressed as *read-modify-write* operations, defined as follows. Let r be a register, and f a function. The operation $rmw(r, f)$ is informally defined by the procedure shown in Figure **??**, which is executed atomically. If f is the identity, $rmw(f, r)$ is simply a *read* operation. Otherwise we say that a read-modify-write operation is *nontrivial*.

Examples of well-known nontrivial read-modify-write operations include *test&set*, *swap*, and *fetch&inc* (see Figures **??**, **??** , **??**).

Theorem .5 *A register with any nontrivial read-modify-write operation has a consensus number at least 2.*

```
RMW(register, f : function) returns(value);
    previous := r;
    r := f(r);
    return (previous);
    end RMW
```

Figure 7: The *rmw* operation

```
T&S(r : register) returns(value);
    previous := r;
    r := 1;
    return (previous);
    end T&S
```

Figure 8: The *test&set* operation

```
SWAP(r : register, v : value) returns(value);
    previous := r;
    r := value;
    return (previous);
    end SWAP
```

Figure 9: The *swap* operation

```
INC(r : register) returns(value);
    previous := r;
    r := r + 1;
    return (previous);
    end INC
```

Figure 10: The *inc* operation

Proof: Since f is not the identity, there exist a value v such that $f(v) \neq v$. Let P and Q be two processes that share two-register array *prefer*, where each entry is initialized to *null*, and a read-modify-write register initialized to v. Let us look at the protocol for P in Figure **??** (Q's protocol is symmetric).

Because r and *prefer* are linearizable, it suffices to check correctness for sequential executions. The protocol chooses P's input if P's operation occurs first, and Q's operation otherwise. If $P's$ *rmw* operation has returned value v, P will return P's preference. Because *rmw* is atomic and P returned value v, P must have changed v before Q did so Q will return a value different from v and will return P's preference in the agreement protocol. The same applies for the symmetric case.

■

Corollary .6 *It is impossible to construct a wait free implementation of any nontrivial read-modify-write operation from a set of atomic read/write registers in a system with* $n \geq 2$ *processes.*

4.1 RMW Operations with Interfering Functions

Definition Let F be a set of functions indexed by an arbitrary set S. F is *interfering* if for all values v and all i and j in S, either:

1. f_i and f_j commute: $f_i(f_j(v)) = f_j(f_i(v))$
2. One function "overwrites" the other: $f_i(f_j(v)) = f_i(v)$ or $f_j(f_i(v)) = f_j(v)$.

Examples of such primitives are *swap* , *inc* and *test&set*.

The *test&set* operation (Figure **??**) uses $f(v) = 1$ and satisfies properties (1) and (2). The *swap* operation (Figure **??**) does not satisfy property (1) but does satisfy property (2). The *inc* operation (Figure **??**) satisfies property (1) but not property (2).

Theorem .7 *Any non-trivial interfering rmw operation has consensus number 2.*

Proof: We have seen that nontrivial *rmw* operation has a consensus number at least 2 (Theorem **??**). Therefore, it is enough to show that there is no three process consensus protocol implemented from *rmw* interfering registers.

```
DECIDE(input) returns(value);
  prefer[p] := input;
  if RMW(r, f)=v
    then return (prefer[P])
    else return (prefer[Q]);
    end if
  end DECIDE
```

Figure 11: Code for DECIDE by P with nontrivial RMW operation

By contradiction we will assume there is a three process consensus protocol implemented from *rmw* interfering registers. Let there P, Q and R be three processes. As in the proof of theorem **??**, we construct a sequential execution that keeps the protocol in a bivalent state by letting P and Q run and keeping R not running. P and Q run until they reach a state s where the next operation that each one will execute is a decision step from the point of view of P and Q (See Figure **??**).

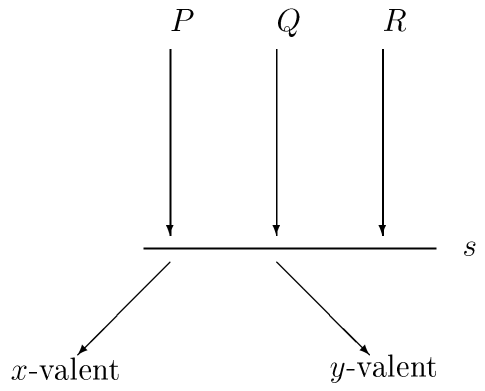

Figure 12: three process consensus protocol

Like in theorem **??**, there can be 3 kinds of operations which will lead the protocol to a univalent state:

1. A process reads a register.
2. The processes write to different registers.
3. The processes write to the same register.

Cases **??** and **??** will lead us to a contradiction for the same reasons as in the proof of theorem **??**. In case **??**, if the processes write to atomic registers, again we will reach a contradiction as before. The difference can be if the processes write to *rmw* interfering registers.

Suppose that the register is r and that P executes *rmw*(r, f_P) and Q executes *rmw*(r, f_Q). There are 2 possible cases:

1. The "commute" property holds: $f_i(f_j(v)) = f_j(f_i(v))$. Let s' be the state that results if P executes $rmw(r, f_P)$ and then Q executes $rmw(r, f_Q)$. s' is x-valent, thus there exists a history fragment consisting entirely of operations of R that will yield a decision value x. Let s'' be the state that results if P and Q perform their operations in the reverse order. s'' is y-valent. Since the register value is the same in s' and s'' from the

point of view of R (the commute property), the protocol has the same history fragment starting from s'', which yields a decision value y. This is a contradiction since R decides different values in each execution. See Figure ??.

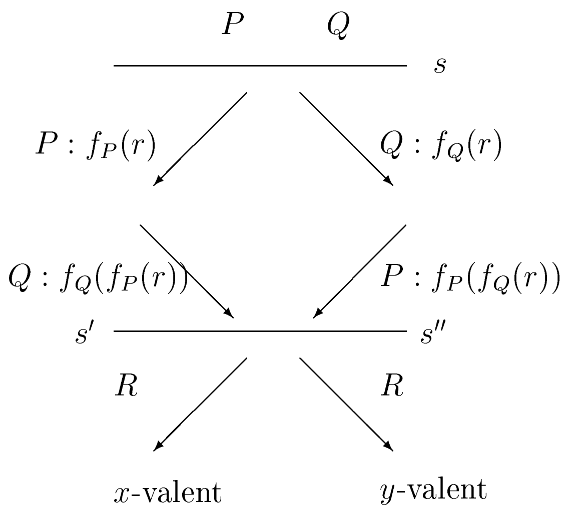

Figure 13: the case where commute property holds.

2. The "overwrite" property holds: $f_i(f_j(v)) = f_i(v)$. Let s' be the state results if P executed $rmw(r, f_P)$ and then Q executes $rmw(r, f_Q)$. s' is x-valent , thus there exist a history fragment consisting entirely of operations of R that will yields a decision value x. Let s'' be the state that results if Q alone executes its operation. Since the register value is the same in s' and s'' from the point of view of R (the "overwrite" property) the protocol has the same history fragment starting from s'' which yield a decision value y. This is a contradiction since R decides different values in each execution. See Figure ??

■

5 Bibliography

1. M. Herlihy - Wait-free Synchronization. ACM Trans. on Programming Languages and Systems, Vol. 11, No.1, Jan 1991, pp. 124-149.

2. C.Kruskal, L. Rudolph and M. Snir, Efficient Synchronization of multiprocessors with shared memory. In Proceeding of 5'th PODC, Aug 1986.

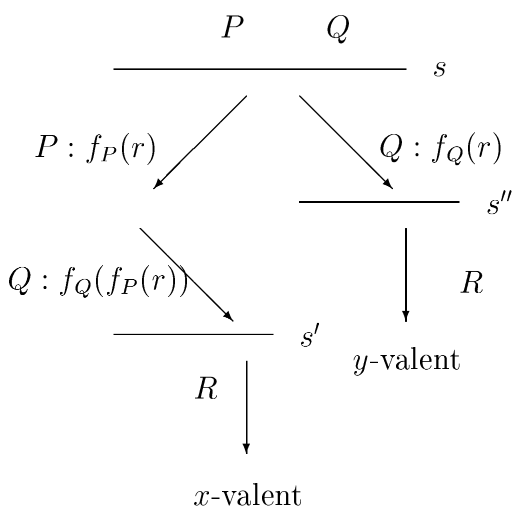

Figure 14: the case where the overwrite property holds