6.891 Multiprocessor Synchronization October 11, 1995

Lecturer: Nir Shavit

Handout 4

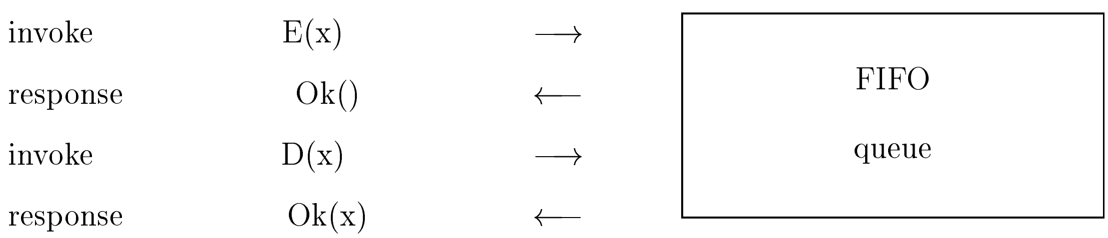
1 Reminder of Definitions From Our Last Class

Lets begin by summarizing the definitions of two weeks ago. Informally, a concurrent system consists of a collection of *processes* that communicate through shared data structures called *objects*. Each object has a unique *name* and a *type*. The type defines a set of possible *values*, and a set of primitive *operations* that provide the only means to manipulate that object (For example, Enq and Deq in a FIFO queue). Processes are sequential: each process applies a sequence of operations to objects, alternately issuing an invocation and then receiving the associated response.

Formally, an execution of a concurrent system is modeled by a *history*, which is a finite sequence of operation *invocation* and *response events*.

Definition A *subhistory* of a history H is a subsequence of the events of H.

An operation invocation is written as $< x\ op(args^*)\ A >$, where x is an object name, op is an operation name, $args^*$ denotes a sequence of arguments values, and A is a process name. The response to an operation invocation is written as $< x\ term(res^*)\ A >$ where $term$ is a termination condition and res^* is a sequence of results. We use "Ok" for normal termination. A Schematic representation is given in the following Figure.



Definition A response *matches* an invocation if their objects names agree and their process names agree.

Definition An invocation is *pending* in a history if no matching response follows the invocation.

Definition If H is a history, *complete*(H) is the maximal subsequence of H consisting only of invocations and matching responses.

Definition A history H is *sequential* if:
(1) The first event of H is an invocation.
(2) Each invocation, except possibly the last, is immediately followed by a matching response.

A history that is not sequential is *concurrent*.

Definition A *process subhistory*, H|P (H at P), of a history H is the subsequence of all events in H whose process names are P.
(An *object subhistory* H|x is similarly defined for an object x.)

Definition Two histories H and H' are *equivalent* if for every process P, H|P = H'|P.

Definition A history H is *well-formed* if each process subhistory H|P of H is sequential.

All histories we deal with are assumed to be well-formed.

Notice that whereas process subhistories of well-formed history are necessarily sequential, object subhistories are not.

Definition An *operation* in a concurrent system is a pair consisting of an invocation *inv(x)*, and the next matching response *res(x)*.

2 The Linearizability Concept Continued

We will now continue with few more definitions, before we get to the definition of *linearizability*.

Definition A set S of histories is *prefix-closed* if, whenever H is in S, every prefix of H is also in S.

Definition A *sequential specification* for an object is a prefix-closed set of sequential histories for the object.

Definition A sequential history H is *legal* if each object subhistory H|x belongs to the sequential specification for x.

For example, for a FIFO queue, a sequential specification will be the set of all sequential histories that satisfy:

$$(\forall xy)(E(x) \longrightarrow E(y) \quad \Rightarrow D(x) \quad \longrightarrow D(y)) \wedge$$
$$(\forall x)(E(x) \longrightarrow D(x)).$$

Definition An operation is *total* if it is defined for every object state, otherwise it is *partial*.

For example, *Enq* is a total operation, and *Deq* is a partial operation (because it is left undefined for the empty queue). A history H induces an irreflexive partial order $\longrightarrow_H$ on operations: e1 $\longrightarrow_H$ e2 if res(e_1) precedes inv(e_2) in H. If H is sequential, then $\longrightarrow_H$ is a total order.

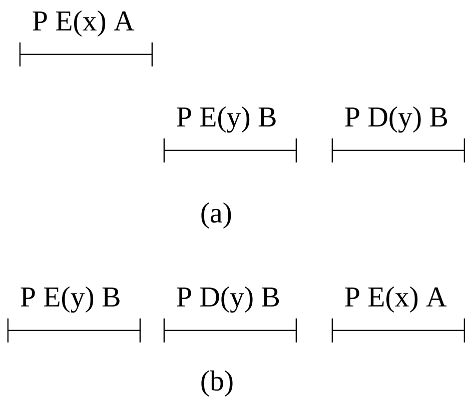


Figure 1: History H7 and it's equivalent sequential history.

2.1 Definition of Linearizability

Definition A history H is *linearizable* if it can be extended (by appending zero or more response events) to some history H' such that:

L1 : complete(H') is equivalent to some legal sequential history S, and

L2 : $\longrightarrow_H \subseteq \longrightarrow_S$.

Informally, extending H to H' captures the idea that some pending invocations may have taken effect even though their responses have not yet been returned to the caller. Extending H to H' restricting attention to complete(H') makes it possible to finish unfinished operations or just ignoring them. *L2* states that this apparent sequential interleaving respects the precedence ordering of operations.

3 Properties of Linearizability

Theorem .1 *H is linearizable iff for each object* x, $H \mid x$ *is linearizable.*

This property of linearizability is called *locality*. *Locality* is important because it allows concurrent systems to be designed and constructed in a modular fashion; linearizable objects can be implemented, verified, and executed independently. *Locality* should not be taken for granted; as the following discussion shows, the literature includes proposals for alternative correctness properties that are not local.

Definition [Lamport] A history is *sequentially consistent* iff it is equivalent to a legal sequential history.

This condition is weaker than *linearizability* since it only requires condition *L1* of the definition of the *linearizability*.

The history *H8*, shown in Figure 4.6a, shows that sequential consistency is not a local property. $H8|p$ is sequentially consistent (by switching $op1$ and $op2$). So is $h8|q$ (by switching $op3$ and $op4$), but *H8* itself is not - because by switching the order for the two objects (p, q) it looses its sequential consistency.

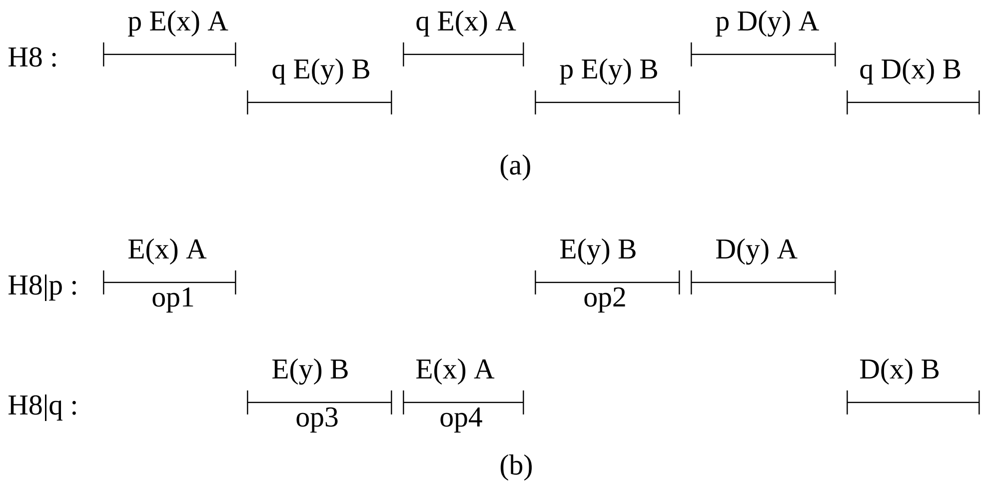


Figure 2: History H8.

3.1 Comparison to Other Correctness Conditions

Much work on databases and distributed systems uses *serializability* as the basic correctness condition for concurrent computations. In this model, a *transaction* is a thread of control that applies a finite sequence of primitive operations to a set of objects shared with other transactions.

Definition A history is *serializable* iff it is equivalent to one in which transactions appear to execute sequentially, i.e., without interleaving. This condition is similar to condition L1 in the definition of the *linearizability.* The main different is that here we deal with a set of objects, while in the linearizability we deal with transaction that restricted to consist of a single operation applied to a single object.

Definition A history is *strictly serializable* if the transactions' order in the sequential history is compatible with their precedence order. This condition is similar to condition L2 in the definition of the *linearizability* , except, again, here we deal with set of objects.

While linearizability is a *local* property, *serializability* and *strict serializablity* are not.

Another important formal difference between linearizability and serializability is that both serializability and strict serializability are blocking properties, while linearizability is non-blocking. Under certain circumstances, a transaction may be unable to complete a pending operation without violating serializability. Such a transaction must be rolled back and restarted, implying that additional mechanism must be provided for that purpose. For example, consider the history H9, involving two register objects x and y, and two transactions A and B. A and B read x and y and then attempt to write new values to y and x. It is easy to see that both pending invocations cannot be completed without violating serializability.

Perhaps the major distinction between serializability and linearizability is that the two notions are appropriate for different problem domains. Serializability is appropriate for database systems, in which it must be easy for application programmers to preserve complex application-specific invariants spanning multiple objects. Linearizability is appropriate for generalized individual shared object implementations.

H9:

x READ() A
y READ() B
x OK(0) A
y OK(0) B
x WRITE(1) B
y WRITE(1) A

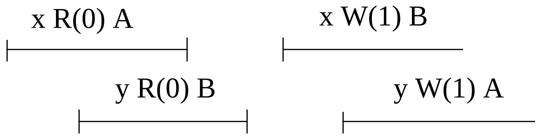


Figure 3: History H9

4 Granularity of Operations

The *Granularity* of an implementation describes the size of command sets that must be executed *atomicly*, that is, be totally ordered by "$\longrightarrow$". Figure **??** describes a hierarchy of *granularity*. Our previous implementation of the FIFO queue had coarse granularity. We will try now to improve the implementation in order to reach lower level of granularity, thus improving the efficiency of the application. The code for the lower granularity algorithm appears in Figure **??**.

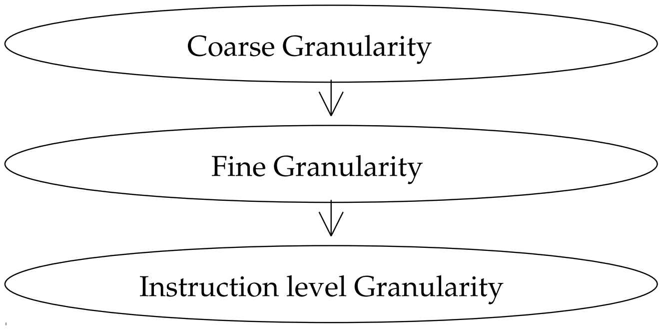


Figure 4: Granularity Hierarchy.

As it can be seen in Figure 4.9, the improved algorithm moves the Critical Section from Deq and Enq functions to smaller fraction of the functions: the functions INC and SWAP.

Though we will not prove that the improved algorithm is *linearizable* we will illustrate the above on a specific example.

In history H1, shown in Figure **??**, we see that the history is linearizable, because it has an equivalent sequential history, decided based on who actually put the information in the queue.

In history H1*, shown in Figure **??** the history is linearizable with an equivalent sequential history decided according to the *Inc* operation - the history is decided by the one who occupies

```
Enq( q : queue , x : item );
    i := INC(q.back);
    WRITE(q.item[i] ,x);
    end Enq

Deq(q:queue) returns(item);
    while true do
            range := READ(q.back) - 1 ;
            for i in 1..range do
                x := SWAP(q.items[i], null);
                if x ≠ null then return (x)
                  end if
                end for
            end while
    end Deq

INC(x:item);
    mutex_entry;
    saved_x := x;
    x := x + 1;
    mutex_exit;
    return (saved_x);
    end INC

SWAP(x, y : item);
    mutex_entry;
    saved_x := x;
    x := y;
    mutex_exit;
    return (saved_x);
    end SWAP
```

Figure 5: Revisited routines

the queue cell first.

There is no one way to decide how the operation occurred. The definition of *linearizability*, as discussed before, is thus not a deterministic one: there may be more than one way to linearize a history.

```
H1:
    Enq(x) A
            Enq(y) B
                    INC(q.back) A
                    OK(1) A
                    INC(q.back) B
                    OK(2) B
                    WRITE(q.items[2], y) B
                    OK() B
            OK() B
            Deq() C
                   READ(q.back) C
                   OK(3)
                   SWAP(q.item[1],null) C
                   OK(null) C
                   SWAP(q.item[2],null) C
                   OK(y) C
            OK(y) C
            .
            .
            .
```

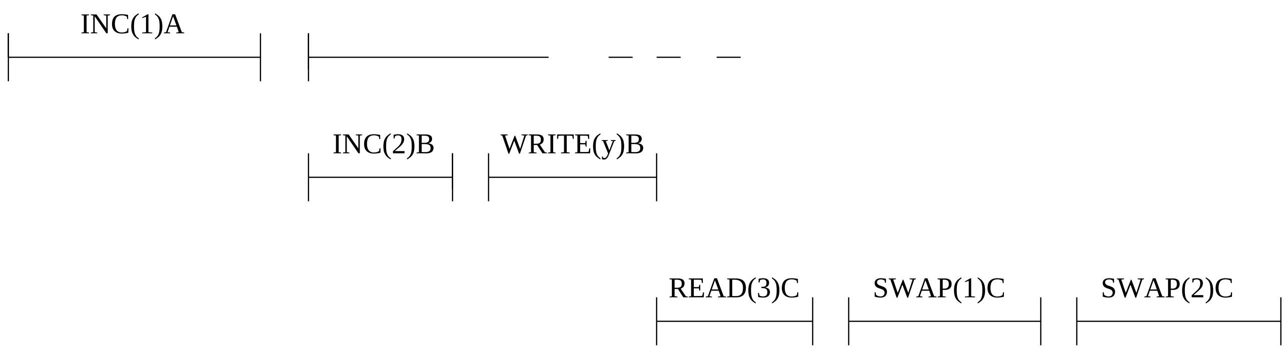


Figure 6: History H1

5 Granularity and Wait Free Synchronization

Clearly there is a price to be paid by using critical sections and having coarse granularity. We will talk about this price later in the coarse. The key idea is that the extent to which we can enhance performance by parallelizing our program is limited by the amount of inherent sequntial sections in it, which in turn is determined in part by the level of granularity, that is, what lengths of operation sequences must be performed one completely after another.

Define $Fraction_{Enhanced}$ to be the fraction of the program that can be enhanced, and $Speedup_{Enhanced}$ to be the speedup in that portion. Enhanced in our case means parallelized.

```
H1*:
    Enq(x) A
           Enq(y) B
                  INC(q.back) A
                  OK(1) A
                  INC(q.back) B
                  OK(2) B
                  WRITE(q.item[2], y) B
                  OK()B
           OK() B
           WRITE(q.item[1], x) A
           OK() A
    OK() A
    Deq() C
          READ(q.back) C
          OK(3)
          SWAP(q.item[1], null) C
          OK(x) C
    OK(x) C
```

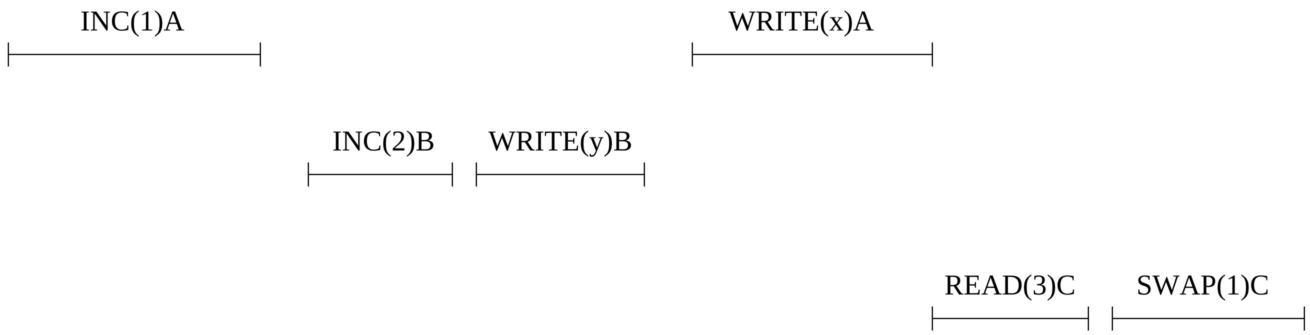


Figure 7: History H1*

Corollary .2 Amdahl's law*:*

$$Speedup\ Overall = \frac{Execution\_time_{Old}}{Execution\_time_{New}} = \frac{1}{1 - Fraction_{Enhanced} + \frac{Fraction_{Enhanced}}{Speedup_{Enhanced}}}$$

Let as assume that a specific calculation that accounts for 40% of the execution time and can be sped-up by a factor of 10. Then

$$SpeedupOverall = \frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64} = 1.56$$

Linear Speedup If we have a machine with 100 processors and we want to speed a computation up 100 times, we get, by Amdahl's law that

$$Fraction_{Enhanced} = 1.$$

I.e. the program must be totally parralelized and there must not be any serial parts in it.
If we settle for a speedup of only 99 times, then

$$1 - Fraction_{Enhacned} = 0.0001.$$

5.1 Wait-freeness

Can we devise efficient algorithms that work on the level of granularity of the machine itself, avioding the use of critical sections altogether?

Definition An implementation of an object X is called *wait-free* if in any execution history each operation applies a *finite* number of lower-level operations (a processor is thus *independent* of other processors' execution rate).

Definition An implementation of an object X is *non-blocking* if every infinite execution history consists of an *infinite* number of operations, each in turn consisting of a *finite* number of lower-level operations.

Corollary .3 *An implementation containing a mutually exclusive section is not wait-free.*

Corollary .4 *A processor eventually returns an answer in a wait-free implementation, even if all other processors fail.*

6 Register types

6.1 Register hierarchy

Is the lowest level of *granularity* made of *atomic* operations? To answer this question, we wish to construct a kind of "*Turing-Machine*" for concurrent operations using the lowest level shared objects that are still usable. Given our claim from the first class that communication must at least be persistent, the weakest object we can define is probably a *Single-Reader Single-Writer (SRSW) Safe Register.*

Definition A register is called *Safe* if every *READ* which is not concurrent with a *WRITE*, obtains the last written value, and a concurrent *READ* obtains one of the *possible* set of values.

The example in Figure **??** is implemented on *SRSW Safe Register*. The values, returned by $R1$,$R2$ and $R3$ are :

- $R1$ returns the value written before: 0.

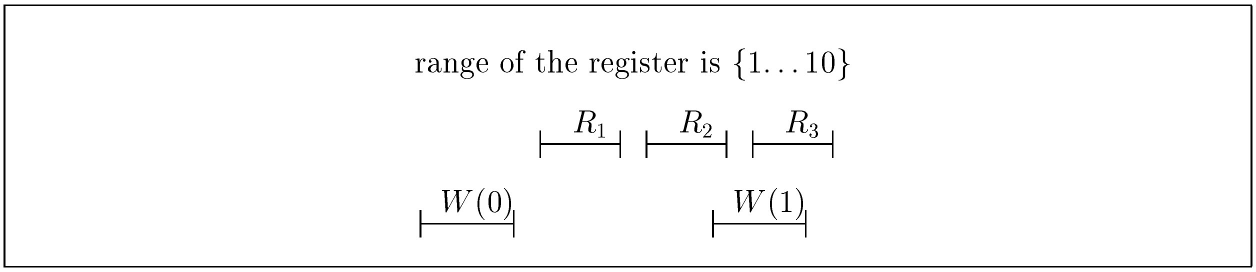


Figure 8: a run of *SRSW Register*

- $R2$ and $R3$ are concurrent with $W(1)$, so they can return each value from $\{1 \ldots 10\}$.

A stronger type of register is the *Regular Register*.

Definition A register is called *Regular* if it has the following properties:

1. It is a *Safe Register*.
2. Every *READ* which is concurrent with a *WRITE* obtains either the old or the new value.

For an implementation of Figure ??, running on *SRSW Regular Register*, the read operations return:

- $R1$ returns the value written before: 0.
- $R2$ and $R3$ return either 0 or 1.

Definition An *Atomic Register* is:

1. *Regular*.
2. *Linearizable* to a sequential register.

For an implementation of Figure ??, running on *SRSW Atomic Register*, the read operations return:

- $R1$ returns the value written before: 0.
- If $R2$ returns 1 then $R3$ will also return 1.
- If $R2$ returns 0 then $R3$ could return 0 or 1.

Figure ?? shows a *3-D* hierarchy, where the *SRSW Safe Register* with *boolean* set of values is the lowest. Higher level register objects are constructed from such registers in a *wait-free* manner..

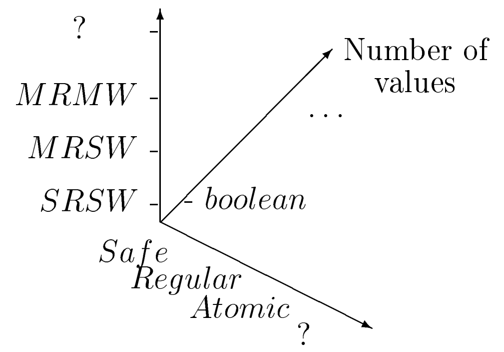


Figure 9: Register Hierarchy

6.2 Register constructions

CONSTRUCTION 1

Given : *SRSW Safe Register*
Wanted : *MRSW Safe Register*

Lemma .5 *The construction in Figure* **??** *is a MRSW Safe Register.*

$V_1, V_2, \ldots, Vn$: SRSW Safe Registers
(Processor P_i reads from V_i.)

```
procedure WRITE(x) ;
    for all i in {1 ... n} do V_i := x ;
    end WRITE ;

    procedure READ_i(x) ;
        x := V_i ;
        return x ;
    end READ_i ;
```

Figure 10: MRSW Safe Register

Proof: For sequential $READ_i$ and *WRITE*, each V_i holds the current x, and processor P_i returns x. If $READ_i$ and *WRITE* are concurrent then the reader might return any legal value because V_i is a *Safe Register.* ■

Lemma .6 *If* $V_1, V_2, \ldots, V_n$ *are SRSW Regular Registers than Figure* **??** *implements MRSW Regular Register.*

Proof: It is similar to lemma **??**, but if the reader is executing $x := V_i$ and the writer's $V_i := x$ is concurrent, then the reader obtains the old or the new value because V_i is a *Regular Register.* ■

CONSTRUCTION 2

Given : *Boolean MRSW Safe Register*
Wanted : 2^m *valued MRSW Safe Register*

Lemma .7 *The construction in Figure* **??** *is a* 2^m *MRSW Safe Register.*

```
V_1, V_2, ..., V_m : Boolean MRSW Safe Registers

procedure WRITE(X[1...m]) ;
   for all i in {1 ...m} do V_i := X[i] ;
   end WRITE ;

   procedure READ_i(X) ;
      for all j in {1 ...m} do X[j] := V_j ;
         return X ;
      end READ ;
```

Figure 11: 2^m valued MRSW Safe Register

CONSTRUCTION 3

Given : *Boolean MRSW Safe Register*
Wanted : *Boolean MRSW Regular Register*

Lemma .8 *The construction in Figure* **??** *is a Boolean MRSW Regular Register.*

```
V : Boolean MRSW Safe Register

procedure WRITE(x) ;
   if old ≠ x
     then
          V := x ;
          old := x ;
     fi ;
end WRITE ;

procedure READ(x) ;
   x := V ;
   return x ;
end READ ;
```

Figure 12: Boolean MRSW Regular Register

Proof: If the reader and writer are not *concurrent* then the READ returns a right value. If they are *concurrent* then:

1. If the writer's new value $\neq$ old value then it writes the new boolean value in the Safe Register. The reader can obtain each value in {'0','1'} because of the concurrency, but '0' and '1' are the old and new values in this case.

2. If the writer's new value = old value then the writer doesn't write the value again, so the reader obtains the right value.

■

CONSTRUCTION 4

Given : *Boolean MRSW Regular Register*
Wanted : *m-valued MRSW Regular Register*

```
Vx : Boolean MRSW Regular Register

Init : Vm := 1 ;

procedure WRITE(x) ;
    Vx := 1 ;
    for i := x − 1 step -1 until 1 do Vi := 0 od ;
    end WRITE ;

    procedure READ(x) ;
        x := 1 ;
        while Vx = 0 do x := x + 1 od ;
                return x ;
        end READ ;
```

Figure 13: *m-valued* MRSW Regular Register

Proof:
Lemma .9 *The construction in Figure* **??** *always returns a value.*
V_m always contains '1' since it initially contains a "1" and no writer ever attempts to write a '0' in V_m, as this would mean it wrote a value of $m+1$ which is outside the registers range. No reader ever reads beyond V_m. ■

1 2 3 4 5 6 7 8

0	0	0	0	1	0	0	1

Reader $\longrightarrow$ $\longleftarrow$ Writer

Figure 14: example for *8-valued* Register

The example in Figure **??** demonstrates an *8-valued* register with a value of 5.

Lemma .10 *The construction in Figure* **??** *is a m-valued MRSW Regular Register.*

Proof: Since regular boolean registers are used, there is no way to get a futuristic value, that is, one that was not written yet. The reader always returns the "1" corresponding to a single V_i. Since there is a single writer, let x be the value of the last totally preceding write (there is a unique one). V_x was written a 1 and all V_i's $(i < x)$ written a 0, all prior to the read. Any read that obtains a value different from x obtains the single 1 value of a write that must have been later than x and by definition was concurrent with the read. ■ We will not construct the following two constructions:

1. SRSW Regular Register to SRSW Atomic Register.
2. SRSW Atomic Register to MRSW Atomic Register.

CONSTRUCTION 5

Given : *MRSW Atomic Register*
Wanted : *MRMW Atomic Register*

The symbol $\succ$ defines a lexicographic order on pairs of the form $(label, i)$ where $label \in R^+$ and $i \in \{1 \ldots n\}$. $Order(l)$ is a function that returns a permutation that describes the order $\succ$. The construction essentially uses an unbounded impelmentation of a concurrent time stamp system ($CTSS$), as in the bakery algorithm. For example, if $l =$ (5, 4, 6, 2, 2, 4) then $order(l) =$ (4, 5, 2, 6, 1, 3).

The label for each *WRITE* command in Figure **??** is monotonically increasing between two sequential writes. The label is also monotonically non-decreasing between sequential *READ* commands.

Lemma .11 *The construction in Figure* **??** *is a MRMW Atomic Register.*

Proof:

1. The construction is a Safe Register. This is trivial because we always write legal values so we will always return legal values when we read.
2. The construction is Linearizable.
   We need to prove that:
   L1: For every history H, $\rightarrow_H$ is a partial order that induces a sequential order $\rightarrow_S$.
   L2: $\rightarrow_S$ is an order that completes $\rightarrow_H$.

```
V_1,...,V_n : MRSW Atomic Registers (Unbounded)
        ( Processor P_i writes to Vi. )

procedure COLLECT(U_1,...,U_n) returns U;
    for all j in {1 ...n} do U_i := V_j od ;
        return (U) ;
    end COLLECT ;

    procedure WRITE_i(x) ;
        U := COLLECT(V_1,...,V_n) ;
        label := MAX(U[1].label, ..., U[n].label) +1 ;
        V_i :=< label, x > ;
    end WRITE ;

    procedure READ(x) ;
        U := COLLECT(V_1, ..., V_n);
        return ( U[i].value : such that
              (U[i].label, i)  ≻  (U[j].label, j)
                         ∀j ∈ {1...n} ) ;
    end READ ;
```

Figure 15: MRMW Atomic Register

Any two *WRITE* operations are ordered according to their *label* values, which are consistent with "$\rightarrow$." If the WRITEs are *concurrent*, and the same *label* was written by both, then the order is lexicographically determined according to the *id* of the processors. This v induces a total order that completes the writes of $\rightarrow_H$ to $\rightarrow_S$. The reads are ordered so that the READ is immediately after the WRITE command with a matching label value. Since the registers used are atomic, this value was obtained by the read from a write immediately preceding it or concurrent, are reads can be mapped so that $\rightarrow_H$ on READs and WRITEs is completed to $\rightarrow_S$ and the proof follows.

■

7 Bibliography

- J. Hennessey & M. Patterson, Computer Architecture - Quantitative Approach, Morgan Kaufmann Publishers, 1990.
- M.P Herlihy and J.M. Wing, Linearizability: A correctness condition for concurrent objects, ACM Trans. on Programming Languages and Systems, Vol.1,No. 3, July 1990, pp.463-492.