Lecturer: Nir Shavit

Handout 5

0.1 Summary from last week on 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 .1 *An implementation containing a mutually exclusive section is not wait-free.*

Corollary .2 *A processor eventually returns an answer in a wait-free implementation, even if all other processors fail.*

1 Register types

1.1 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.
- $R2$ 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:

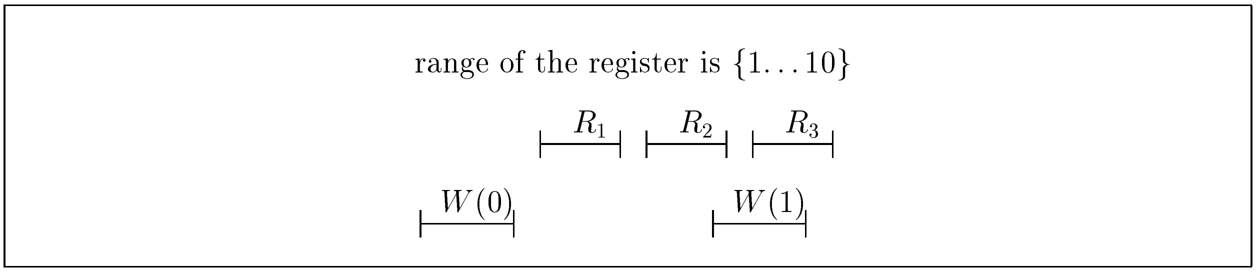


Figure 1: a run of *SRSW Register*

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..

1.2 Register constructions

Let us begin this weeks class with a sequence of constructions starting with our weakest assumed register, the safe register.

CONSTRUCTION 1

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

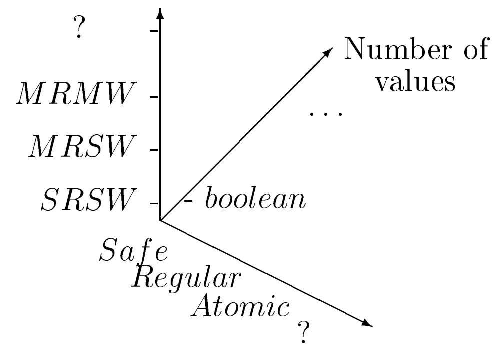


Figure 2: Register Hierarchy

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

procedure WRITE(x) ;
    for all i **in** $\{1 \ldots \text{n}\}$ **do** $V_i := x$;
    end WRITE ;

    procedure READ$_i$(x) ;
        $x := V_i$;
        return x ;
    end READ$_i$;

Figure 3: MRSW Safe Register

Lemma .3 *The construction in Figure* **??** *is a 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 .4 *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 .5 *The construction in Figure* **??** *is a* 2^m *MRSW Safe Register.*

```
V1, V2, ..., Vm : Boolean MRSW Safe Registers

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

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

Figure 4: 2^m valued MRSW Safe Register

CONSTRUCTION 3

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

Lemma .6 *The construction in Figure* **??** *is a 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.

```
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 5: Boolean MRSW Regular Register

■

CONSTRUCTION 4

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

```
V_x : Boolean MRSW Regular Register

Init : V_m := 1 ;

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

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

Figure 6: *m-valued* MRSW Regular Register

Proof:

Lemma .7 *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. ■

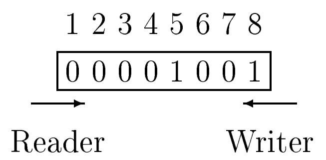


Figure 7: example for *8-valued* Register

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

Lemma .8 *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 .9 *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.

```
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 8: MRMW Atomic Register

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 $\rightarrow_H$.

   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.

■

2 Atomic Snapshots of Shared Memory

This section introduces a new primitive: an atomic snapshot of shared memory. The section will cover the definition, motivation for the definition and a wait-free implementation using SWMR atomic registers.

3 The Snapshot Memory Automaton

A Single-Writer Atomic Snapshot Memory is an abstract data type with an input procedure called an *update*, and an output procedure called a *scan*. These procedures are defined for n processes $P_1, \ldots, P_n$. In every sequential execution, a $scan_j$ returns a vector of n values, where the i-th entry is the value of the latest $update_i$ operation execution immediately preceding the $scan_j$ operation execution.

Our formal specification is made by describing the single-writer snapshot memory in terms of a canonical single-writer snapshot automaton. The canonical automaton is a concurrent object that can easily be seen as linearizable to the above sequential specification for an atomic snapshot memory. A correct implementation of the single-writer snapshot memory is one which the processes cannot distinguish from the canonical single-writer snapshot automaton.

$UpdateRequest_i(d)$	$\rightarrow$	S.W.
$UpdateReturn_i$	$\leftarrow$	Snapshot
$ScanRequest_i$	$\rightarrow$	Memory
$ScanRequest_i(d_1, \ldots, d_n)$	$\leftarrow$	Automaton

d, d_i is an object of type DATA.

Figure 9: The Single-writer Snapshot Memory Automaton

The canonical automaton has the interface snapshot actions of the snapshot memory automaton and in addition two internal actions: $Update_i(d)$ and $Scan_i(d_1, \ldots, d_n)$ for any $i \in \{1, 2, \ldots, n\}$ and $d_1, d_2, \ldots, d_n \in Data$. The states of the canonical automaton contain the n-entry array *Mem* of type *Data* and n interface variables H_i. The interface variables may hold as value any of the interface snapshot actions or a special value $\perp$.

3.1 The Canonical Single-Writer Snapshot Automaton

$$\begin{array}{llcl}
\multicolumn{4}{l}{\mathit{UpdateRequest}_i(d)} \\
\mathit{Effect:} & H_i & := & \mathit{UpdateRequest}_i(d) \\
\\
\multicolumn{4}{l}{\mathit{Update}_i(d)} \\
\mathit{Precondition:} & H_i & = & \mathit{UpdateRequest}_i(d) \\
\mathit{Effect:} & Mem[i] & := & d \\
 & H_i & := & \mathit{UpdateReturn}_i \\
\\
\multicolumn{4}{l}{\mathit{UpdateReturn}_i} \\
\mathit{Precondition:} & H_i & = & \mathit{UpdateReturn}_i \\
\mathit{Effect:} & H_i & := & \perp
\end{array}$$

Figure 10: *Update*$_i$ process

Figures ?? and ?? show a specification of the automaton's *scan* and *update* processes.

$$\begin{array}{llcl}
\multicolumn{4}{l}{\mathit{ScanRequest}_i} \\
\mathit{Effect:} & H_i & := & \mathit{ScanRequest}_i \\
\\
\multicolumn{4}{l}{\mathit{Scan}_i(d_1, \ldots, d_n)} \\
\mathit{Precondition:} & H_i & = & \mathit{ScanRequest}_i \\
 & Mem & = & (d_1, \ldots, d_n) \\
\mathit{Effect:} & H_i & := & \mathit{ScanReturn}_i(d_1, \ldots, d_n) \\
\\
\multicolumn{4}{l}{\mathit{ScanReturn}_i(d_1, \ldots, d_n)} \\
\mathit{Precondition:} & H_i & = & \mathit{ScanReturn}_i(d_1, \ldots, d_n) \\
\mathit{Effect:} & H_i & := & \perp
\end{array}$$

Figure 11: *Scan*$_i$ process

The automaton is equivalent to one in which all $scan_i$ and $update_i$ operations are serial (sequential), and therefore it's a *linearizable* object. An example of a valid history of the automaton is presented in figure ??.

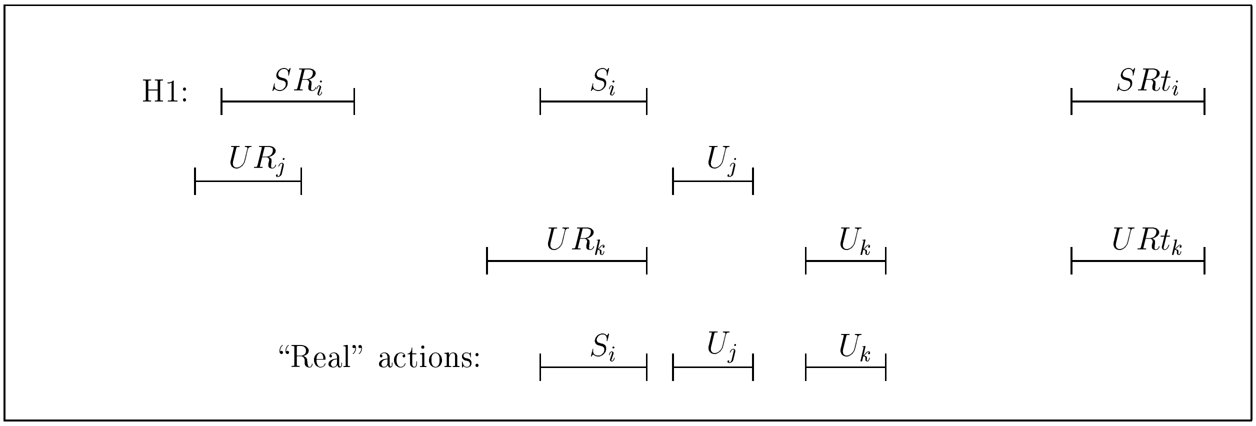


Figure 12: An example of the automaton history

4 The Unbounded Snapshot Algorithm

4.1 A Simple, Non-wait-free Algorithm

Let us use two fields in each register, one for the data and another for a *sequence number*, which counts the number of write operations into that register. Each processor writes the data and the sequence number simultaneously (an atomic write), and executes a *collect* operation which reads each register r_i, $i \in 1, \ldots, n$ in an arbitrary order. We note the following: **Observation** If every update leaves a unique mark when it writes to the memory, then if two sequential reads of the memory return identical values, where second read starts after the first is completed, then the values returned constitute a snapshot. We can thus ensure that a read operation will return a true snapshot of memory, by using the sequential number field; however, this algorithm is non wait-free, because assuming we have several writers, who repeatedly write into memory, a large amount of snapshots would be missed, and the condition of **??** will not be met.

4.1.1 A Wait-free Algorithm

In constructing a wait-free version of the algorithm, we use the following observation: **Observation** If a scan sees another process "move" (complete an update) twice, that process executed a complete update operation within the interval of the scan. Figure **??** lists the *scan* and *update* operations of the Atomic Snapshot Memory Single-writer Unbounded Algorithm. Each process P_i has a single writer, n-reader atomic register r_i that P_i writes and all the processes read. The register has three fields – *data* (of type *Data*), *seq* (of type integer) and *view* (an array of n *Data* values). The *data* field and the *view* array are initialized to d_{init} and the *seq* fields are initialized to 0.

Each *scan* operation maintains a local array *moved*, in which it records, for each other process, whether that process has been observed to change the memory during the course of the scan.

The *collect* operation of each P_i reads each register $r_j, j \in 1, \ldots, n$, in arbitrary order.

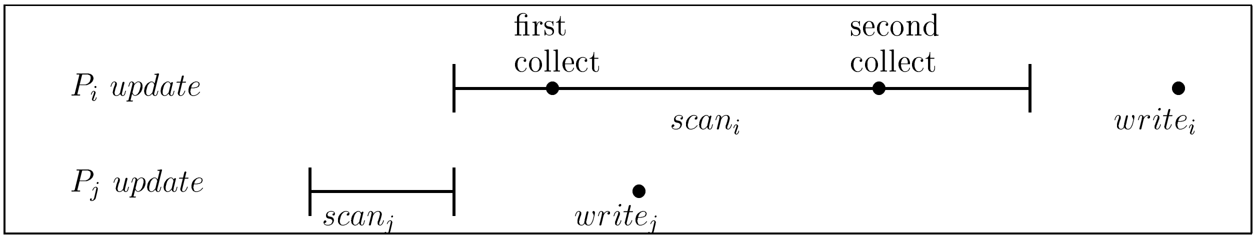
```
procedure scan_i () ;
   begin
   0: for j := 1 to n do moved[j] := 0 od ;
   1: while true do
   2:    a[1..n] := collect; /* (data, seq, view) triples */
   3:    b[1..n] := collect; /* (data, seq, view) triples */
   4:    if (∀j ∈ 1,...,n)(a[j].seq = b[j].seq) then
   5:       return (b[1].data,...,b[n].data) ; /* Nobody moved */
   6:    else for j = 1 to n do
   7:    if a[j].seq ≠ b[j].seq then /* P_j moved */
   8:        if moved[j] = 1 then /* P_j moved once before */
   9:            return (b[j].view) ;
   10:       else moved[j] := moved[j] + 1 ;
         od ;
      od ;
   end scan_i ;



procedure update_i (data) ;
   begin
   1:   s[1..n] := scan_i;
   2:   r_i := (data, r_i.seq + 1, s[1..n]);
   end scan_i ;
```

Figure 13: Unbounded single-writer atomic snapshot register

We cannot take the snapshot of a processor which changed its sequence number *once* during our scan. This is exemplified by the following scenario:



P_i sees P_j "move", but P_j's snapshot was taken before P_i started the update operation, which means this snapshot did not occur within the interval of P_i's scan.

4.2 Correctness proof

Lemma .10 *Let $\alpha = \pi_1, \pi_2, \ldots$ be a run of the unbounded algorithm in which a particular $scan_i$ operation has a successful double collect: $a[1..n]$:= collect in line 2 and $b[1..n]$:= collect in line 3. Let π_u and π_w be the last READ of the first collect and the first READ of the second collect, respectively. Then for every prefix α_v of α, $u < v < w$, snapshot(α_v) = ($b[1].data, \ldots, b[n].data$).*

Proof: Suppose a write by P_j is serialized between two successive reads by P_i or r_j in lines 2 and 3. Since the sequence number in r_j is incremented with each write the sequence number returned by the second read will be strictly greater than that returned by the first. It follows that if the sequence numbers are not observed to change, no write by P_j is serialized between the successive reads. This implies the result. ■

Lemma .11 *Let $\alpha = \pi_1, \pi_2, \ldots$ be a run of the unbounded algorithm in which a particular $scan_i$ operation observes changes in process P_j's sequence number field during two different double collects. Then the value of r_j read during the last collect was written by an $update_j$ operation that began after the first of these four collects started.*

Proof: If two successive reads by P_i of r_j in lines 2 and 3 return different sequence numbers, then at least one write by P_j to r_j is serialized between this pair of reads. Process P_j writes to r_j only as the final step of each $update_j$ operation. Hence, one $update_j$ operation ended sometime after the first read by P_i, and the write step of another occurs between the last pair of reads by P_i. Since $update_j$ operations run serially (only one $UpdateRequest_j$ is outstanding at a time), the lemma follows. ■

These two lemmas imply that all scans can be correctly linearized somewhere in their intervals.

Lemma .12 *Let $\alpha = \pi_1, \pi_2, \ldots$ be a run of the unbounded algorithm in which a particular $scan_i$ operation beginning in event π_u returns $(d_1, \ldots, d_n)$ in event π_w. Then snapshot(α_v) = $(d_1, \ldots, d_n)$ for some $v, u < v < w$.*

Proof: If the $scan_i$ operation has a successful double collect, the result follows from **??**. Assume instead the $scan_i$ operation borrows a snapshot value read in r_j. By **??** the snapshot value read in r_j was obtained by a $scan_j$ operation, embedded in an it update$_j$ operation, which in turn started after the first read by P_i of r_j and wrote before the last read by P_i of

r_j. Hence the interval of the embedded $scan_i$ is contained between the first and last read by P_i of r_j. Either the $scan_j$ operation had a successful double collect, and the result again follows from **??** or there is another embedded $scan_k$, occurring entirely within the interval of the $scan_j$ operation, from which P_j is borrowed. This argument can be applied inductively, noting, that there can be at most n concurrent operations in the system. Hence, eventually the embedded scan must have succeeded via a successful double collect, and the result follows by **??**, and transitivity of containment of the embedded scan intervals. ■

Lemma .13 *Every scan or update operation by process P_i returns after $O(n^2)$ atomic steps of P_i, $\forall i \in 1, \ldots, n$.*

Proof: In $n+1$ double collects one double collect must be successful or some updater must be observed moving twice. Therefore the scans are wait-free, which implies that the updates are wait-free. The complexity follows from $(n+1)$ possible loops of $O(n)$ complexity each. Lemma **??** states, that during the interval of each scan operation, there is at least one moment at which the data is kept in all the registers simultaneously. Each scan is linearized at this point. The update operation is linearized at the embedded writes.

A single scan or update operation can be linearized as to be equivalent to the Scan and Update operations of the canonical automaton. If a number $k-1$ between $1, \ldots, n$ of scans and updates can be thus linearized, then k'th scan operation can be linearized as the last scan between the matching update and the next one, k'th update operation can be linearized before the first matching scan operation (if there isn't one – between other updates and after it's scan – in its interval). ■

This implies that the scan and update operations of the single-writer snapshot memory are equivalent to the canonical automaton. We thus establish the following theorem:

Theorem .14 *The unbounded algorithm implements a wait-free single-writer snapshot memory.*

5 The Single-Scanner Algorithm

Suppose we only want to use the scan for a backup by a single scanner. Then one can greatly reduce the overhead of the multi-scanner algorithm like the one above. The following algorithm is a greatly simplified variant of the Kirousis, Spirakis, and Tsigas single scanner snapshot algorithm. Their algorithm is based on the innovative idea of letting the scanner change, in each scan, the memory location where updaters write their values. The key to the algorithm presented below is a simple pragmatic structure that achieves optimal time complexity with very small constants: $2n$ read/write operations for a scan and 4 read/write operations for an update. The algorithm uses a sequential time-stamp system (a register of 64 bits will suffice, though one can readily replace it by a more sophisticated bounded sequential time-stamp system to achieve a simple *bounded* read/write register *single-scanner* protocol).

The algorithm is designed to allow carrying the *update* operation structure without change over to the case of multiple scanners. The code for *scan* and *update* operations appears in Figure **??**.

```
Shared data structures:
  curr_seq: integer;
  type register: record [ value:integer, seq:integer ];
  memory: array [1..n] of record [ high:register,low:register ];

Scan()
begin
       curr_seq := curr_seq+1;
       for j=1 to n
       begin
           high_r := memory[j].high;
           if (high_r.seq < curr_seq) then
                view[j] := high_r;
           else
                view[j] := memory[j].low;
       end;
       return(view);
end;

Update(val)
begin
       seq := curr_seq;
       high_r := memory[i].high;
       if (seq = high_r.seq) then
           memory[i].high := [val,seq];
       else begin
           memory[i].low := high_r;
           memory[i].high := [val,seq];
       end;
end
```

Figure 14: The Single Scanner algorithm - code for P_i

The algorithm uses a shared array `memory[1..n]` of records, each having a `high` and a `low` field, and a shared variable `curr_seq` which holds a current time-stamp (sequence number). The basic idea is to let a single scanner set a new timestamp as its first step and then collect updaters values, while the first operation in each update is to read this timestamp. The scanner classifies the collected updaters values according to its newly created timestamp, and returns only those values that were written by update operations which did not read its time-stamp.

These updates must have started and performed the read before the scan, and so all their associated values *could have existed* in memory at the point in the execution when the increment of the time-stamp was performed. To guarantee that such a value is found for each updater, even if scanning is concurrent with updating we keep, in an additional memory location (the low field), the latest value updated with a timestamp preceding that of the current/latest scan. Therefore, in case an updater "sees" a scan which has started after its last update operation it modifies the low and high fields of its update location. First it updates the low field to hold its previous update value which should be available for concurrent scans, and later then the high field to hold the new update value.

To prove the correctness of this construction we must show that each possible history of this algorithm is equivalent to a legal history. In a nutshell, given a concurrent history of the *single-scanner* algorithm, we order every *Scan* operation S and *Update* operation U in the following way: If U read `curr_seq` after S incremented it, then U is ordered after S. If U read the content of `curr_seq` before S changed it, then if S read `memory[i].high` before U wrote into it, U is ordered before S otherwise U is ordered after S. Clearly, this total order extends the partial order induced by the concurrent history: take for example, two update operations $U_1 \to U_2$ that are totally ordered in the concurrent history. If U_2 is ordered in the sequential history before some scan operation S, then U_2 must have read the content of `curr_seq` before S incremented it. This implies that U_1 updated its registers in the `memory` array before S incremented `curr_seq` and therefore before S read the `memory` array. Therefore U_1 should also be ordered before S.

6 Bibliography

- Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, N. Shavit
  Atomic Snapshots of Shared Memory
  Journal of the ACM 40:4 (1993), 873-890.

- Y. Riany, N. Shavit, D. Touito.
  Towards a Practical Snapshot Algorithm.
  Proceedings of the Third Israel Symposium on Theory and Computing Systems (ISTCS), Tel Aviv, (1995), 121-129.

- 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.

- L.Lamport
  "On interprocess communication",
  Part 2, Dist. Computing 1, 1986, pp. 86-101.
- B. Awerbuch and P.Vitanyi
  "Construction of a MRMW atomic Register",
  MIT Technical Report.