In the last lecture, we saw that in order to create atomic MRSW registers, the readers and writer need to communicate with each other. This communication is necessary for correctness. Since there is only writer in this case, the MRSW register has a global timestamp $t_s$, hence we can use these timestamps for ordering. However, in MRMW atomic, there are several writers and there cannot be a global time timestamp, as each writer has its own timestamp. Hence the question arises as to how do we globally order the events when there are multiple timestamps. For example, there might be two processors $A$ and $B$, and the first event on both processors will have a timestamp $t$ as shown below.

```
A ---------|-------------------
   t
B -------------|---------------
   t
```

Thus based only on the timestamps, it is not possible to order the events correctly. The idea here is to use vector clocks. A vector clock of a system of $n$ processes is an $n$-element vector of $n$ logical clocks, one per process. A local copy of the vector clock is kept by each process. If the timestamps of processors 1, 2, ..., $n$ are $t_1, t_2, ..., t_n$, then the vector clock will be $[t_1, t_2, ..., t_n]$. The new timestamp of a processor is the vector of the local timestamp read from all processors, with its own timestamp incremented by 1. This is a partial order. Total ordering is useful for ordering events in distributed systems. So, we convert it to a total order by applying lexicographic order to the vector of timestamps.

### 1 Atomic Snapshot Memory

In distributed and concurrent systems, obtaining an instantaneous global picture of a system, from partial observations made over a period of time as the system state evolves is a fundamental problem. Proving correctness of concurrent algorithms is difficult because of the “inconsistent” views of shared memory obtained by processors concurrently with other processors’ modifications of shared memory. Verification of concurrent algorithms is complicated by the need for non-interference. By simplifying non-interference, atomic snapshot memories can greatly simplify the design and verification of many concurrent algorithms.
We will study a general formulation of atomic snapshot memory, a shared memory partitioned into different segments. This memory is instantaneously read (scanned) in its entirety or written (updated) by individual processes.

1.1 Atomic Snapshot Register

Atomic snapshot is a powerful register that allows several data items to be read atomically. For a system of $n$ processors, the atomic snapshot object is a single object with $n$ segments, one corresponding to each of the $n$ processors.

```
|---|---|---|---|---|---|---|---|---|
P1  P2  P3  P4  ................Pn
```

The property of this atomic snapshot object is that in a single update operation, a processor $P_i$ updates its segment in the snapshot object and in a single scan operation, the processor $P_i$ reads the contents of all the segments of the object.

Because of the above property of a atomic snapshot object, it is a preferred object from the programmers point of view, since it is intuitively suitable for the design of shared memory algorithms and their verification. Also, reasoning about various operations is also easier using snapshots. We will discuss a wait-free simulation of atomic snapshot object from bounded size SRSW registers.

The operations of update and scan involved in an atomic snapshot register can be described as follows for $P_i$, $(0 \leq i \leq n - 1)$:

- **scan$_i$**: Invocation whose response is return$_i$(V).
- **update$_i$($d$)**: Invocation whose response is ack$_i$.

Where, (1) $V$ is a $n$-element vector, called a view, and (2) $d$ is the data being written to $P_i$’s segment.

Intuitively, the scan operation will read the value of all the segments of the snapshots serially one after the other. However, the snapshot has the property that this read of all the segments can be linearized as a single point. Note that $\forall i$, a scan$_i$ and update$_i$ operation sequence is in the allowable set iff for each $V$ returned by a scan operation $V[i]$ equals the parameter of the latest preceding update$_i$ operation.

We first consider an unbounded register algorithm. The scan reads all the segments twice(double-collect). If no segment is updated during the double-collect, then the result of each collect is clearly a snapshot, and scan can return the value read. The problem arises when there is an update between the double-collect.

We have two issues to answer in that case.

1. How do we tell whether an update has happened ?
2. What do we do if the snapshot object has changed ?
To answer the first issue, we can use unbounded sequence numbers for each write operation, so that if the value of any sequence number increases between the double-collect, then we can know that an update has occurred. If we want to make a resource bounded system, we can use a handshaking mechanism instead of the unbounded sequence numbers. The handshaking mechanism differentiates between 0, 1 and many updates.

To answer the second issue we will show that if a scanner observes several changes in the segment of a specific updater, then the updater has performed a complete update during the scan. We will modify the update to embed a scan operation at the beginning of the update. The view obtained in this scan is written with the data in the update operation. The scanner returns the value obtained in the last collect. We prove below that this is correct.

Also, note that seeing 2 identical collects does not prove that updates did not happen. To illustrate this, consider the following 3 processor execution of the snapshot.

![Figure 1: Each line is a snapshot. But the value of double-collect [124] is not a snapshot value.](image)

We can see that the scan returns the value [124]. This value is incorrect because the snapshot object does not contain this value at any moment. Thus seeing two identical collects does not prove that update did not happen. In order to prevent such a situation, we use a handshaking mechanism which differentiates between multiple updates.

### 1.2 The Handshaking Mechanism

Consider any two processors \( (P_i, P_j) \), where \( i \neq j \). We have four handshaking procedures of processors \( P_i \) and \( P_j \) of try and check. There are two shared SRSW registers \( h_i \) and \( h_j \), where \( h_i \) is written by \( P_i \) and read by \( P_j \) and \( h_j \) is written by \( P_j \) and read by \( P_i \). Briefly, the handshaking mechanism does the following.
Handshake:
P_i modifies h_i to make them equal. P_j modifies h_j to make them unequal.

Check:
P_i and P_j check to see if a handshake has occurred since it modified the bit.

Algorithm 1 Handshake Algorithm

1. procedure try HS_i()
   \( h_i := h_j \) \{P_i tries to make bits equal\}
   return

2. procedure try HS_j()
   \( h_j := \neg h_i \) \{P_j tries to make bits unequal\}
   return

3. procedure check HS_i()
   return \((h_i \neq h_j)\) \{P_i checks if handshake occurs\}

4. procedure check HS_j()
   return \((h_i = h_j)\) \{P_j checks if handshake occurs\}

Observation:
try HS_i and try HS_j does one read and one write, while check HS_i and check HS_j does one read.

The SRSW registers used to construct the Atomic Snapshot object are linearizable as we have seen earlier, hence there exists a linearization of the basic operations. We now have to linearize the whole algorithm.

To prove this, consider any legal execution multiple calls to the handshaking procedures and a fixed linearization of read and write operations embedded in the procedures.

Handshaking Property 1 If a check HS_i returns TRUE, then there exists a try HS_j whose write occurs between the read of previous try HS_i and the read of check HS_i.

Proof: Consider the below figure illustrating the above property.
Suppose a call to check HS_i returns TRUE. Then P_i observes the bits to be unequal i.e. its read of \( h_j \) returns the opposite of what it wrote to \( h_i \) in preceding try HS_i.
So, in preceding try HS_i, P_i wrote \( h_i := b \) because it read \( h_j = b \). Now, P_i reads \( h_j = 1 - b \) in the check HS_i. Therefore, there must be a call to try HS_j, whose write is linearized between the two reads of P_i.

Is is clear that the linearization of try HS_j must happen between the two dotted lines because this is the region where the value of \( h_j \) has changed due to a write \( h_j \) operation. This is illustrated in Figure 2.
Handshaking Property 2 If a call to check $HS_i$ returns FALSE, then there is no $try HS_j$ whose read and write both occur between the write of the previous $try HS_i$ and read of check $HS_i$.

Proof: Consider Figure 3 illustrating the above property. Suppose a call to check $HS_i$ returns FALSE. Then $P_i$ observes the bits to be equal, i.e. its read of $h_j$ returns the same value it wrote to $h_i$ in preceding $try HS_i$. So, in preceding $try HS_i$, $P_i$ wrote $h_i := b$ because it read $h_j = b$. Now, $P_i$ still reads $h_j = b$.

Suppose, for contradiction, there is a call to $try HS_j$ whose read and write occur between $P_i$’s write of $h_i = b$ and $P_i$’s subsequent read of $h_j$. The read by $P_j$ would return $h_i = b$ and the write by $P_j$ would be $h_j := 1 - b$, contradicting the fact that $P_i$ reads $h_j = b$.

Note that only read or only write of $try HS_i$ can happen. Here, there are two possibilities.

- $try HS_j$ read the value but did not write to make them different.
- $try HS_j$ read $h_i$ sometime before, and wrote the same value that $try HS_i$ read.

1.3 Bounded Memory Simulation

For each $P_i$ and $P_j$, where $P_i$ is a scanner and $P_j$ is an updater, we have a handshaking mechanism in the atomic snapshot simulation denoted $(P_i, P_j)$.
The scanner does the following:

1. Handshake with all other processors.
2. Does a double − collect.
3. Checks handshake with all processors.

Because handshaking and collecting by scanner and writing and handshaking by updater’s are done in separate operations, an update to some segment may not be observed. This is shown in Figure 4.
To differentiate two consecutive updates to the same segment, we include a bit with the data of each segment, called as toggle bit, which the updater toggles in each update. So both toggle and handshake bits are used to for differentiating updates. Processor $P_i$ first does a read of $h_j$, followed by the write making $h_i := h_j$. Once this is done, it does the first collect. But before $P_i$ does the second collect, $P_j$ writes to its segment and changes the toggle bit. When $P_i$ does the check $HS_i$, the writer has not yet started the handshaking($try \ HS_j$). So even though the check $HS_i$ returns false, it still means that $P_j$ did an update.

Because the handshake bits $h_i$ and $h_j$ are not written atomically, a scan may observe changes by the same update operation twice: once while changing the handshake bits, and once while changing the toggle bit. Hence, a scan operation must observe three updates from the processor $P_j$ before the value from $P_j$’s view (say $view_j$), can be borrowed.

The scanner $P_i$ repeatedly (1) Handshakes (2) Double-collects and (3) Checks for handshakes, until it sees three changes in $P_j$’s segment. Seeing three changes implies that $P_j$ has performed a complete update during $P_j$’s scan, so it is ok to return the value of $P_j$’s embedded scan.

We alter the update code to first perform a scan and then include the view returned by the scan with the data and toggle bit. The scanner $P_i$ returns the view obtained from $P_j$ in the last collect.