Section 5
Foundations of Shared Memory:
Fault-Tolerant Simulations of read/write objects

Simple Read/Write Register Simulations
✓ We show that registers that may seem more complicated, i.e., multi-writer (MW) multi-reader (MR) multi-valued registers have a wait-free implementation using simpler registers, i.e., single-writer (SW) single-reader (SR) binary registers.

- single-writer single-reader binary
- single-writer single-reader multi-valued
- single-writer multi-reader multi-valued
- multi-writer multi-reader multi-valued
Multi-valued SW SR Registers from Binary SW SR Registers

Basic Objects
- Binary registers, each of which can be read by just one process and written by just one process.

Implemented (or high-level) object
- A k-valued register which can be read by just one process and written by just one process.
- We represent values in unary.
- We use an array of k binary SW SR registers B[0..k-1].
- The value j is represented by a 1 in the jth entry and 0 in all other entries.

A Simple Algorithm

```plaintext
read() {
    for j = 0 to k-1
        if (B[j] == 1) return j;
}

write(v) {
    B[v] = 1;
    for j = 0 to k-1, j ≠ v,
        B[j] = 0;
    return <ack>;
}
```

This algorithm is not linearizable ☹️


Write(1)  Write(2)  Read() → 1
A Correct Algorithm

Main Ideas

- A write operation clears only the entries whose indices are smaller than the value it is writing.
- A read operation does not stop when it finds the first 1, but makes sure there are still zeroes in all lower indices.

```java
read(R) {
    i = 0;
    while B[i] == 0 do i = i+1;
    up = i;
    v = i;
    for i = up -1 down to 0 do
        if B[i] == 1 then v = i;
    return v;
}

write(R,v) {
    B[v] = 1;
    for i = v-1 down to 0 do B[i] = 0;
    return <ack>;
}
```

Multi-Valued from Binary Registers

Linearizability

- Let a be any admissible execution of the algorithm.
- We say that a (low-level) read r of any B[v] in a reads from a (low-level) write w to B[v], if w is the latest write to B[v] that precedes r in a.
- We say that a (high-level) Read R in a reads from a (high-level) Write W, if R returns v and W contains the write to B[v] that R's last read of B[v] reads from.

- We construct a sequential execution σ containing all the high-level operations in a, such that
  (1) σ respects the order of non-overlapping operations in a, and
  (2) every Read operation in σ returns the value of the latest preceding Write.
Construction of the sequential execution $\sigma$

- In two steps:
  1. We put in $\sigma$ all the Write operations according to the order in which they occur in $a$:
     - Since we have a unique writer, this order is well-defined.
  2. Consider the Reads in the order they occur in $a$; since we have a unique reader, this order is well-defined.
     - For each Read $R$, let $W$ be the Write that $R$ reads from.
     - Place $R$ immediately before the Write in $\sigma$ just following $W$ (i.e., place $R$ after $W$ and after all previous Reads that also read from $W$)

- By the defined placement of each Read, every Read returns the value of the latest preceding Write and therefore $\sigma$ is legal.

We have to prove that $\sigma$ preserves the real-time ordering of non-overlapping operations.

---

**Lemma 1**

Let $op_1$ and $op_2$ be two high-level operations in $a$ such that $op_1$ ends before $op_2$ begins. Then, $op_1$ precedes $op_2$ in $\sigma$.

**Proof**

- By construction, the real-time ordering of Write operations is preserved.
- Consider some Read operation, $R_i$ by $p_i$.
- If $R$ finishes in $a$ before a Write $W$ begins, then $R$ precedes $W$ in $\sigma$, because $R$ cannot read from a Write that starts after $R$.
- We proceed by case analysis.
  - Case 1: Write before Read.
  - Case 2: Read before Read.
**Multi-Valued from Binary Registers**

**Lemma 2:** Consider two values $u$ and $v$ with $u < v$. If Read $R$ returns $v$ and $R$'s read of $B[u]$ during its upward scan reads from a write contained in Write $W_i$, then $R$ does not read from any Write that precedes $W_i$.

**Proof:** Suppose in contradiction that $R$ reads for a Write $W(v)$ that precedes $W_i(v_i)$ (see figure).

- It should hold that (1) $v_i > u$ (since $W_i$ writes 1 in $B[v_i]$ and then does a downward scan), and (2) $v_i < v$ (since otherwise $W_i$ would overwrite $W$'s value to $v$).
- $R$’s upward SCAN reads $B[u]$, then $B[v_i]$, then $B[v]$.
- This SCAN should read 0 in $B[v_i]$ (otherwise $R$ would return $v_i$ and not $v$).
- Thus, there must be another Write $W_2(v_2)$ after $W_i$ that writes 0 in $B[v_i]$ before $R$ reads $B[v_i]$.
- It should be that $v_2 > v_i$ and $v_2 < v$ (for similar reasons as above).
- We apply this argument repeatedly to get an infinite increasing sequence of integers $v_i, v_2, ...$, all of which are less than $v$. A contradiction!

---

**Multi-Valued from Binary Registers**

**Proof of Lemma 1 (case analysis continued)**

**Case 1: Write before Read**

- Suppose in contradiction $R$ is placed before $W$ in $\sigma \Rightarrow R$ reads from some Write $W'(v')$ that precedes $W$.
- $v' < v$: Then $W$ overwrites the write to $B[v']$ by $W'$ before $R$ begins. A contradiction (since then $R$ does not read from $W'$, as assumed).
- $v' > v$: By Lemma 2, $R$ cannot read from $W'$. 

---
Multi-Valued from Binary Registers

Case 2: Read before Read

Suppose in contradiction that $R_1$ follows $R_2$ in $\sigma$ ⇒ $R_1$ reads from a Write $W_i(v_i)$ that follows the Write $W_j(v_j)$ from where $R_2$ reads.

- $v_1 = v_2$: When $W_1$ writes 1 to $B[v_1]$ it overwrites the 1 that $W_2$ wrote to $B[v_2]$ earlier. Thus, $R_2$ cannot read from $W_2$. A contradiction!

- $v_1 > v_2$: Since $R_1$ reads 1 from $B[v_1]$, the write of $W_1$ to $B[v_1]$ precedes the read of $R_1$ from $B[v_1]$. The write of 1 to $B[v_2]$ by $W_2$ precedes the write of $W_1$ to $B[v_1]$. Thus, from the write of $W_1$ to $B[v_1]$ until the read of this value from $R_2$, no write to $B[v_2]$ occurs.

Thus, during the downward scan, $R_1$ must read 1 in $B[v_2]$, and therefore, $R_1$ does not return $v_1$. A contradiction!

Multi-Valued from Binary Registers

Case 2: Read before Read (continued)

$v_1 < v_2$:

- Since $R_1$ reads from $W_j$, $W_j$’s write of 1 to $B[v_j]$ precedes $R_1$’s last read of $B[v_j]$.

- Since $R_2$ returns $v_2 > v_1$, $R_2$’s first read of $B[v_j]$ must return 0.

- So, there must be another Write after $W_j$ containing a write of 0 to $B[v_j]$ that $R_2$’s read of $B[v_j]$ reads from.

- Lemma 2 implies that $R_2$ cannot read from $W_2$. A contradiction!
Multi-Valued from Binary Registers

Theorem

- There exists a wait-free simulation of a K-valued register using K binary registers in which each high-level operation performs $O(K)$ low-level operations.

Multi-Reader from Single-Reader Registers

A Simple Algorithm

Shared Variables: value Val[n]: // an array of n elements, one for each reader

write(v) {
    for (j=1; j ≤ n; j++) Val[j] = v;
}

read { // code for $p_i$, 1 ≤ i ≤ n
    return(Val[i]);
}

➢ This algorithm is not linearizable 😞
Multi-Reader from Single-Reader Registers

**Theorem 3**

- In any wait-free implementation of a single-writer multi-reader register from any number of single-writer single-reader registers, at least one reader must write.

**Proof:** By the way of contradiction!

Since the implementation is linearizable, \( \forall i \in \{1,2\}: \exists j_i, 1 \leq j_i \leq k, \) such that, \( v_j = 0 \) for all \( j < j_i \) and \( v_j = 1 \) for all \( j \geq j_i \).

**Why is this TRUE?**

- It holds that \( j_1 \neq j_2 \). Wlog, assume that \( j_1 < j_2 \).
- \( R_{j1} \) returns 1, whereas \( R_{j2} \) returns 0.
- This contradicts linearizability!!

---

**A Correct Algorithm**

**Shared Variables:**

- \(<\text{value,seq}>\ Val[i]\); // 1 \( \leq i \leq n \), value written by \( p_i \) for each of reader \( p_i \)
- \(<\text{value,seq}>\ Report[i,j]\); // 1 \( \leq i, j \leq n \), value returned by the most recent Read // operation performed by \( p_i \); // written by \( p_i \) and read by \( p_j \), initially, \(<v_0,0>\)

**// code for each reader \( p_r \), 1 \( \leq r \leq n \)**

```plaintext
read() {
    <v[0],s[0]> = Val[r];
    for i=1 to n do
        <v[i],s[i]> = Report[i,r];
    let j be s.t. s[j] = max{s[0], s[1], ..., s[n]};
    for i=1 to n do Report[r,i] = <v[j],s[j]>;
    return v[j];
}
```

**// code for the single writer \( p_w \)**

```plaintext
write(v) {
    seq = seq +1;
    for i=1 to n do Val[i] = <v,seq>;
    return <ack>;
}
```

---
Construction of $\sigma$

- In two steps:
  1. We put in $\sigma$ all the Write operations according to the order in which they occur in $a$:
     - Since we have a unique writer, this sequence is well-defined.
     - This order is consistent with timestamps associated with the values written.
  2. Reads are considered, one by one, in the order of their responses in $a$
  3. A Read operation that returns a value with timestamp $T$ is placed immediately before the Write that follows the Write operation that generated timestamp $T$.

✓ By the defined placement of each Read, every Read returns the value of the latest preceding Write and therefore $\sigma$ is legal. 😊

➢ We have to prove that $\sigma$ preserves the real-time ordering of non-overlapping operations.

---

**Lemma 4:** Let $op_1$ and $op_2$ be two high-level operations in $a$ such that $op_1$ ends before $op_2$ begins. Then, $op_1$ precedes $op_2$ in $\sigma$.

**Proof:** By construction, the real-time order of Write operations is preserved.

- Consider some Read operation, $R$, by $p_i$ that returns a value associated with timestamp $T$.

```
Read R by p_i
\sigma: \ldots \* W \* R \ldots
Write W
\sigma: \ldots \* W \* R \ldots
Read R
```

If $R$ follows $W$ in $\sigma$, then the write $W'$ that generates timestamp $T$ is either $W$ or a later Write, implying that $W'$ occurs after $R$ in $a$. A contradiction!!

Since $R$ occurs after $W$, $R$ reads from $\text{Val}[i]$ the value written by $W$ or a later Write $\Rightarrow$ $R$ returns a value whose associated timestamp is generated by $W$ or a later Write. Thus $R$ is not placed before $W$ in $\sigma$.

---

**Theorem 5:** There exists a wait-free implementation of an $n$-reader register using $O(n^2)$ single-reader registers in which each high-level operation performs $O(n)$ low-level operations.
Multi-Writer from Single-Writer Registers

Main Ideas
- Have each writer announce each value it wants to write to all the readers by writing it in its own SW MR register; each reader reads all the values written by the writers and picks the most recent one among them.
- \( p_1, ..., p_m \): writers, \( p_1, ..., p_n \): readers
- Each timestamp is now a vector of \( m \) components, one for each writer.
- The new timestamp of a processor is the vector consisting of the local timestamps read from all other processors, and its local timestamp increased by one.
- We order timestamps according to the lexicographic order on the timestamps (i.e., according to the relative order of the values in the first coordinate in which the vector differs).
- The algorithm uses the following shared arrays of SW MR r/w registers:
  - vector \( \text{TS}[i] \): \( 1 \leq i \leq n \), the vector timestamp of writer \( p_i \)
  - \( \langle \text{vector, value} \rangle \text{Val}[i] \): \( 1 \leq i \leq n \), the latest value written by writer \( p_i \), \( 1 \leq i \leq n \), together with the vector timestamp associated with that value. It is written by writer \( p_i \) and read by all readers.

read() {
    // code for reader \( p_r \), \( 1 \leq r \leq n \)
    for \( i = 1 \) to \( m \) do
        \( \langle v[i], t[i] \rangle = \text{Val}[i]; \)
    let \( j \) be s.t. \( t[j] = \max\{t[1], t[2], ..., t[m]\}; \)
    return \( v[j] \);
}

write(v) {
    // writer \( p_w \) writes \( v \) in \( R \)
    \( \text{ts} = \text{NewTS}(w); \)
    \( \text{val}[w] = \langle v, \text{ts} \rangle; \)
    return \langle \text{ack} \rangle;
}

procedure NewTS(int \( w \)) {
    for \( i = 1 \) to \( m \) do
        \( \text{lts}[i] = \text{TS}[i][1]; \)
    \( \text{lts}[w] = \text{lts}[w] + 1; \)
    \( \text{TS}[w] = \text{lts}; \)
    return \( \text{lts} \);
}

Shared Variables:
- \( \langle \text{value, vector} \rangle \text{Val}[i] \); \( 1 \leq i \leq m \), initially \( \langle v_0, (0, ..., 0) \rangle \)
- vector \( \text{TS}[i] \); \( 1 \leq i \leq m \), initially \( (0, ..., 0) \)
Multi-Writer from Single-Writer Registers

Linearizability

- In a way similar to that we proved linearizability in the previous algorithm.

Construction of $\sigma$

- In two steps:
  - We put into $\sigma$ all the Write operations according to the lexicographic ordering on the timestamps associated with the values they write.
  - A Read operation that returns a value with timestamp VT is placed immediately before the Write operation that follows (in $\sigma$) the Write operation that generated timestamp VT.

- **Lemma 6**: The lexicographic order of the timestamps is a total order consistent with the partial order in which they are generated.

- **Lemma 7**: For each $i$, if $VT_1$ is written to $Val[i]$ and later $VT_2$ is written to $Val[i]$, then $VT_1 < VT_2$.

- By the defined placement of each Read, every Read returns the value of the latest preceding Write and therefore $\sigma$ is legal.

---

Multi-Writer from Single-Writer Registers

**Lemma 8**: Let $op_1$ and $op_2$ be two high-level operations in a such that $op_1$ ends before $op_2$ begins. Then, $op_1$ precedes $op_2$ in $\sigma$.

**Proof**: By Lemma 6, the real time order of Write operations is preserved.

Consider a Read operation, $R$, by $p_i$ that returns a value associated with timestamp VT.

**Case 1**: Arguments similar to corresponding case of Lemma 4.

**Case 2**: $R$ reads from $Val[j]$ the value written by $W$ or some later Write. By semantics of max and Lemma 6, $R$ returns a value whose associated timestamp is generated by $W$ or a later write. Thus, $R$ is not placed before $W$ in $\sigma$.

**Case 3**: During $R$, $p_i$ reads all $Val$ variables and returns the lexicographic maximum. During $R'$, $p_j$ does the same thing.

By Lemma 7, the timestamps appearing in each $Val$ variable are in non-decreasing order. By Lemma 6, they are in non-decreasing order of when they were generated. Thus, $R'$ obtains timestamps from $Val$ that are at least as large as those obtained by $R$. Thus, the timestamp associated with the value returned by $R'$ is at least as large as that associated with the value returned by $R$. 
Multi-Writer from Single-Writer Registers

**Theorem 9**: There exists a wait-free implementation of an m-writer register using $O(m)$ single-writer registers in which each high-level operation performs $O(m)$ low-level operations.