1 Algorithm for Reads and Writes

In the previous lecture, the read write algorithm was introduced. It is reproduced below:

Algorithm (For Processor $P_i$)

- **copy**[$X$] contains initial value of shared object $X$.

  **when** $read_i(X)$ **occurs**
  
  $tbc - send_i(read, X)$

  **when** $write_i(X,v)$ **occurs**
  
  $tbc - send_i(write, X, v)$

  **when** $tbc - recv_i(read, X)$ **occurs from** $P_j$
  
  if $j = i$ then $return_i(X, copy[x])$

  **when** $tbc - recv_i(write, X, v)$ **occurs from** $P_j$
  
  $copy[X] := v$
  
  if $j = i$ then $ack_i(X)$

Note that the time taken for both a read and a write is at most $d$, where $d$ is the maximum message delay. This is because both actions complete when $P_i$ receives its own message. We now proceed by proving some properties of this algorithm.

**Theorem 1** The algorithm satisfies linearizability given a correct TBC.

**Proof:** To prove linearizability, we must show both legality, and the order preservation of events.

Let $\alpha$ be an execution of the algorithm and let $\sigma$ be the sequence of invocations and responses of the Memory Consistency System (MCS) processes. Clearly, $\sigma$ satisfies correct interaction and liveness, since the algorithm terminates as long as the Total-Broadcast System delivers all messages.

Define $\Pi$ by ordering operations in $\sigma$ according to the total order of when their corresponding broadcasts are received (since each invocation initiates a single broadcast). We show $\Pi$ satisfies
legality of each object (The value read is in fact the value of the last write). Let $X$ be a Read-Write object. Since broadcasts are a total order, each MCS receives messages for operations on object $X$ in the same order, the order of $\Pi|X$. Hence, legality is satisfied at each MCS.

We also check that $\Pi$ respects the real-time order of operations in $\alpha$. Let $op_i$ by $P_i$ complete before $op_j$ by $P_j$. Then,

$op_i \Pi op_j$

We know this because $op_i$ completes if and only if it receives its own broadcast, that is, $P_i$ receives $op_i$’s broadcast message before $P_j$ even sends $op_j$’s broadcast message. This implies $op_i$’s broadcast is ordered before $op_j$’s broadcast at all processors. It follows that $op_i \Pi op_j$

It seems intuitive that this algorithm could allow for local reads without broadcast. Unfortunately, there are executions that would violate linearizability. Consider the example of Figure 1. Initially, $X = 0$. Processor $P_i$ initiates a write to change $X$ to 1, and broadcasts it. It receives its own broadcast and acknowledges it prior to processor $P_j$ initiating its read. However, processor $P_j$ completes its local read before receiving the broadcast about $P_i$’s write. Hence, $P_j$ returns zero even though its read took place after the acknowledgement of the write, and hence it should have returned one.

![Figure 1: An example execution showing that this algorithm does not support local reads.](image)

We can relax the need for broadcasts during reads by implementing a sequential consistency algorithm for local reads. The new algorithm is as follows:

**Sequential Consistency Algorithm for Local Reads (For Processor $P_i$)**

- $copy[X]$ contains initial value of shared object $X$.

  - **when** $read_i(X)$ **occurs**
    - $return_i(X, copy[X])$

  - **when** $write_i(X, v)$ **occurs**
    - $tbc - send_i(write, X, v)$
when \( tbc - recv_i(write, X, v) \) occurs from \( P_j \)

\[
\text{copy}[X] := v
\]

if \( j = i \) then \( ack_i(X) \)

**Lemma 1** For every \( P_i \):

1. \( P_i \)'s local copies take on all values of the writes.
2. All updates occur in the same order at all processors.
3. This order preserves the order of writes by individual processors.

**Proof:** (1) holds because a message is sent on every write. (2) holds because of total ordering of the receiving of broadcasts. (3) holds because the ack must happen prior to beginning the next write, which means the prior write had in fact completed.

**Lemma 2** For all \( P_i \), if \( op_1 \) precedes \( op_2 \) in \( \sigma|i \), where \( \sigma \) is the sequence of invocations and responses in an execution \( \alpha \), then the local read/write for \( op_1 \) at \( P_i \)'s copy precedes the local read/write for \( op_2 \) at \( P_i \)'s copy.

**Proof:** The order of operations in \( \sigma \) are:

\[
\text{inv}(op_1) < \text{resp}(op_1) < \text{inv}(op_2) < \text{resp}(op_2)
\]

The local op for \( op_1 \) must occur before \( \text{resp}(op_1) \), and the local op for \( op_2 \) must occur before \( \text{resp}(op_2) \), but after \( \text{resp}(op_1) \). It follows then that the local op for \( op_1 \) is ordered before the local op for \( op_2 \).

**Theorem 2** The algorithm preserves the order of events.

**Proof:** Define the permutation \( \Pi \) of ops in \( \alpha \) as follows. Order the writes in total broadcast order in \( \alpha \). Consider each read in the invocation order in \( \alpha \). Read \( r \) by processor \( P_i \) on object \( X \) is placed immediately after the latest (in \( \Pi \)) of:

1. The previous op of \( P_i \) in \( \alpha \).
2. The write that caused the latest update of \( P_i \)'s copy of \( X \) preceding the local read for \( r \).

Note that \( \sigma|i = \Pi|i \). Let \( op_1, op_2 \) be two ops in \( \Pi|i \). We show that \( op_1 \) precedes \( op_2 \) in \( \Pi|i \) if and only if \( op_1 \) precedes \( op_2 \) in \( \sigma|i \).

**Case 1:** \( op_1 = w_1 \) and \( op_2 = w_2 \), \( w_1 \prec w_2 \). If \( w_1 \) precedes \( w_2 \) in \( \sigma \), then by Lemma 1, \( w_1 \) precedes \( w_2 \) in total broadcast order. So by the definition of \( \Pi \), \( w_1 \prec w_2 \).
Case 2: $op_1 = r_1$ and $op_2 = r_2$, $r_1 \sigma < r_2$. If $r_1$ precedes $r_2$ in $\sigma$, then $r_1$ is inserted first into $\Pi$. $r_2$ has to be inserted somewhere after $r_1$, by condition 1. Hence, $r_1 < r_2$.

Case 3: $op_1 = w_i$ and $op_2 = r_i$, $w_i \sigma < r_i$. Since all writes are placed first, when $r_i$ is inserted, it is placed after $w_i$ by condition 1. So, $w_i < r_i$.

Case 4: $op_1 = r_i$ and $op_2 = w_i$, $r_i \sigma < w_i$. Suppose by contradiction that there exists $r_i$ and $w_i$ such that $r_i \sigma < w_i$, but $w_i < r_i$. Choose the $w_i, r_i$ pair with the earliest placed $r_i$ in $\Pi|\{i\}$.

Case (a): $r_i$ is placed in $\Pi$ immediately after $op_i$, the previous op of $P_i$ in $\alpha$. Now, $w_i \neq op_i$ since $r_i \sigma < w_i$. So $w_i < op_i < r_i$. But, $op_i \sigma < r_i$ and $r_i \sigma < w_i$, so it must be that $op_i \sigma < w_i$. Therefore, $op_i \sigma < w_i$ and $w_i \Pi < op_i$. If $op_i$ is a write, this contradicts case 1. If $op_i$ is a read, then this contradicts the assumption that $w_i, r_i$ are the pair with the earliest reversal in $\Pi$. Either way, this is a contradiction.

Case (b): $r_i$ is placed in $\Pi$ immediately after the $w_j$ that caused the latest update of $P_i$’s copy preceding the local read for $r_i$. Now, $w_i \neq w_j$ since $r_i \sigma < w_i$, and by Lemma 2, the local read for $r_i$ precedes the update for $w_i$ at $P_i$. So it must be that $w_i \Pi < w_j \Pi < r_i$. Since $w_i \Pi < w_j$, by Lemma 1, the update for $w_i$ precedes the update for $w_j$ at $P_i$. But then, by the definition of $w_j$, the update for $w_j$ precedes the local read for $r_i$. But since $r_i \sigma < w_i$, by Lemma 2, the local read for $r_i$ precedes the update of $w_i$, a contradiction.

In all possible cases, it holds that the $\Pi$ constructed from this algorithm preserves the order of events.

To show this algorithm implements a sequentially consistent shared memory with local reads, it remains to demonstrate that this algorithm satisfies legality.