1 Legality of sequential schedule constructed for local read algorithm

We have proved in the previous lecture that there exists a \( \Pi \) for any execution for the local read algorithm such that the ordering of events is preserved (Theorem 2). Now, we shall prove that legality also holds in such a \( \Pi \). As we shall be referring to Lemma 1 and Lemma 2 in the following proof, it is convenient to have them here.

**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 in all processors.
3. This order preserves the order of writes by individual processors.

**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.

**Theorem 3** The \( \Pi \) constructed from \( \sigma \) produced by the local read algorithm satisfies legality.

**Proof**: Consider read \( r_i \) by \( P_i \) that reads \( v \) from \( X \). Let \( w_j \) be the write that caused the latest update to \( P_i \)'s copy of \( X \) and comes before \( r_i \). This write \( w_j \) can belong to any processor \( P_j \). Legality of \( \Pi \) requires that \( w_j < r_i \) in \( \Pi \) and there does not exist a \( w' \) such that it writes to \( X \) and \( w_j < w' < r_i \) in \( \Pi \).

Suppose for contradiction there exists such \( w' \) by \( P_k \). By definition of \( \Pi \), \( r_i \) should be placed immediately after: \( w_j \) (by condition 2 of \( \Pi \) construction) or the latest operation by \( P_i \) that comes before \( r_i \). Since \( w_j < w' < r_i \) in \( \Pi \), by condition 1 of \( \Pi \) construction, there can be one or more than one operations by \( P_i \) that are eligible to be placed between \( w' \) and \( r_i \) in \( \Pi \). Let \( op_i \) be the earliest of such operations. So, we have \( w_j < w' \leq op_i < r_i \).

Case 1: \( op_i \) is a write.

By Lemma 1, since \( w_j < w' \) in \( \Pi \), update for \( w_j \) precedes update for \( w' \). By Lemma 2, since \( op_i \) precedes \( r_i \) in \( \sigma | i \), the update for \( op_i \) precedes the local read for \( r_i \). Since \( w' \leq op_i \) in \( \Pi \), by lemma 1, update for \( w' \) precedes or equals the update for \( op_i \). In either case, update for \( w_j \) precedes update for \( w' \) which in turn precedes the local read for \( r_i \). This contradicts the
assumption that $w_j$ is the latest update of $X$ preceding local read for $r_i$.

Case 2: $op_i$ is a read of some object $Y$.

As per definition of $\Pi$, $op_i$ should be placed in $\Pi$ immediately after: the previous operation by $P_i$ or the latest write $w''$ of $Y$, whose update precedes the local read for $op_i$. Since $op_i$ is the first operation to occur in $P_i$ after $w'$, there is no "previous" operation by $P_i$ that comes between $w'$ and $op_i$. So, the placement of $op_i$ in $\Pi$ has to be done according to the second condition of $\Pi$ construction. That is, there must exist a write $w''$ on $Y$ whose update precedes local read for $op_i$ and so: $w_j < w' \leq w'' < op_i < r_i$ in $\Pi$.

By Lemma 1, update for $w_j$ precedes update for $w'$, which precedes (or equals) update for $w''$. By our assumption, update for $w''$ precedes update for $op_i$. Since $op_i < r_i$ in $\sigma \mid i$, by Lemma 2, local read of $op_i$ precedes local read of $r_i$. So, update for $w'$ precedes local read for $r_i$. Finally we have update for $w_j$ preceding update for $w'$ which in turn precedes local read for $r_i$. This contradicts the fact that $w_j$ is the latest update of $X$ preceding local read for $r_i$. So, legality is satisfied by $\Pi$.

**Theorem 4** The local read algorithm preserves sequential consistency

**Proof:** From Theorem 2 (of lecture 17) and Theorem 3 we can conclude that $\Pi$ maintains ordering of events and satisfies legality. So, the local read algorithm preserves sequential consistency.

## 2 Local write algorithm

Initially, we proposed a local read algorithm that satisfies sequential consistency. The following algorithm which makes local writes instead of local reads can also satisfy sequential consistency.

Pseudocode for processor $P_i$:

1. When $read_i(X)$ occurs
   if num = 0 then
     $return_i(X, copy[X])$
   2. when $write_i(X, v)$ occurs
     $num = num + 1$
     $tbc - send_i(X, v)$
     $ack_i(X)$
   3. when $tbc - recv_i(X, v)$ occurs from $P_j$
     $copy[X] := v$
     if $j = i$ then
       $num = num - 1$
     if $num = 0$ and read on $X$ is pending then
       $return_i(X, copy[X])$
In the above algorithm, writes return instantly and reads involve message passing. When a read occurs, it return only after there are no pending writes by processor \( P_i \). In this case also the order of writes is same across all processors as the actual write is triggered by a \( tbc - recv \).

To construct \( \Pi \) for an execution resulting from the above algorithm: we can order all writes in the order of actual updates as decided by ordering of totally ordered broadcast. In a given \( \sigma \) the reads can have two kinds of relationships with the writes: The read completed prior to the write, the read overlaps the write (this happens when the read is waiting from num to be 0 but another write has arrived and finished). In the first case, we can obviously place it before the write as the write has no affect on it. In the second case, the read has to be placed in \( \Pi \) after all the overlapping writes because it returns the value of the latest overlapping write. So, the following conditions can be used to place reads in \( \Pi \). \( read_i(X) \) has to be placed immediately after which ever is the latest in \( \sigma \):

1. The latest operation by \( P_i \) preceding the \( read_i(X) \)
2. The write that caused the latest update of \( P_i \)'s copy of \( X \) preceding \( ack_i \) associated with the \( read_i(X) \).

### 3 Lower Bounds on Time Complexity

To discuss about time complexity we need to first decide on the model in which the algorithm is running. Based on relative synchronization between the clocks and magnitude of message delay, there are 4 models:

**Model 1**: Perfect Clocks (every processor has same clock value and same speed) and message delay is exactly \( d \).

**Model 2**: Perfect Clocks and message delay \( \in [d - u, d] \).

**Model 3**: Message delay is \( d \) and no perfect clocks.

**Model 4**: No perfect clocks and message delay \( \in [d - u, d] \).

Model 2 is as strong as Model 1. That is, if an algorithm can work in Model 1, it can be made to work in Model 2 also. The only difference between Model 1 and Model 2 is that the messages can reach in less than \( d \) time but never more then \( d \) time. So, the algorithm can postpone processing of the message for \( (d - t) \) time if the message reaches in \( t \) time. Since there are perfect clocks, the algorithm can easily calculate how much time the message spent in transit.

Model 1 is also as strong as Model 2. If an algorithm can work in Model 2, it can work in Model 1 also. Any algorithm that can work with perfect clocks and message delay \( [d - u, d] \) can work in the environment of perfect clocks and message delay \( d \) because the latter environment is like a sub-case of the former environment (because \( d \subset [d - u, d] \)).

Model 3 is as strong as Model 1. If an algorithm can work in Model 1 then it can be made to work in Model 3. The essential difference between Model 1 and Model 3 is every clock on
model 1 knows about the time of every other clock. To do this in Model 3, when a processor receives a message, it can look at the time stamp and know how much ahead or behind the sender clock is as compared to its own clock. It can do this because it knows that message delay is exactly $d$. Suppose there are two processors $P_1$ and $P_2$ and $P_1$ sends a message to $P_2$. Let the time in $P_1$ when the message is sent be $t_1$ and let the time in $P_2$ at that instant be $t_2$.

The message reaches $P_2$ at $t_2 + d$ ($P_2$ time) and $t_1 + d$ ($P_1$ time). The offset from $P_1$ to $P_2$ is simply $t_1 - t_2$. But $P_2$ does not know $t_2$. It knows that the time at which the message arrives which is $t_2 + d$. It also knows $t_1$ from time stamp of message and $d$ is a system property. So, to obtain $t_1 - t_2$, it calculates $(t_1 + d) - (t_2 + d) = t_1 - t_2$.

$P_1$ gets the reply from $P_2$ at $t_1 + 2d$ ($P_1$ time). But the reply is time stamped with the time at which $P_2$ received the initial message which is $t_2 + d$ ($P_2$ time). $P_1$ can also follow the above procedure to calculate offset: add $d$ to the message time stamp and subtract from it the time at which the message is received. $(t_2 + d) + d - (t_1 + 2d) = t_2 - t_1$.

This kind of awareness enables the processors to perform any time critical operations concerning the sending and receiver as required by the algorithm. Model 1 environment is again a sub-case of Model 3 environment. So, Model 1 is as strong as Model 3.

Model 4 is strictly weaker than all of the other models.