1 Atomic Snapshot Memory (Contd....)

1.3 Bounded Memory Simulation (Contd....)

Algorithm 1: Bounded Register Algorithm for \( P_i \)

Require: \( \text{segment}[i].data := v_i \) and \( \text{segment}[i].view := <v_0, v_1, \ldots, v_n> \) {Initialization 1}

Require: \( \forall j, \text{shook}[j] := 0 \) {Initialize local variable shook}

1: if \( \text{scan}_i() \) occurs then
   2: \( \text{view} := \text{SCAN}() \)
   3: return \((\text{view}_i)\) \{This was scan() operation\}
   4: end if

5: if \( \text{update}_i(d) \) occurs then
   6: \( \text{view} := \text{SCAN}() \)
   7: \( \text{segment}[i] := (d, \text{view}, \neg \text{segment}[i].\text{toggle}) \)
   8: \( \forall j \neq i, \text{do try} \: \text{HS}^{(j,i)}() \) \{H/S with scanners. \( j \) is scanner index and \( i \) is updater index\}
   9: \( \text{ack}_i \)

10: end if

11: procedure \( \text{SCAN}() \)
12: \( \forall j \neq i, \text{do} \: \text{shook}[j] := 0 \)
13: while TRUE do
   14: \( \forall j \neq i, \text{do try} \: \text{HS}^{(i,j)}() \) \{H/S with all updates\}
   15: \( \forall j \neq i, \text{do a}[j] := \text{segment}[j] \) \{First Collect\}
   16: \( \forall j \neq i, \text{do b}[j] := \text{segment}[j] \) \{Second Collect\}
   17: if for some \( j \neq i, \text{check} \: \text{HS}^{(i,j)}() \) or \( (a[j].\text{toggle} \neq b[j].\text{toggle}) \) then
      18: if \( \text{shook}[j] = 2 \) then
         19: return \((b[j].\text{view})\) \{Indirect Scan\}
      else
         21: \( \text{shook}[j] := \text{shook}[j] + 1 \)
      end if
   23: else
      24: return \(<b[0].\text{data}, \ldots, b[n-1].\text{data}>\) \{Direct Scan\}
   25: end if
26: end while
Lemma 1  If in $P_i$’s execution of line 17, the condition is true for some $j$ (some writer), then $P_j$ either executes line 7 or the write of handshake with $P_i$ in line 8 between the previous execution in line 14 of the read in try $HS^{(i,j)}_i$ and execution of line 17.

Proof: Condition of line 17 is true either because of change in toggle value or because of check $HS^{(i,j)}_i$ returning TRUE. If the first case, $P_j$ writes to $segment[j]$ (line 7) during $P_i$’s double collect (line 15,16). In the second case, HP 1 implies that there has been a write of try $HS^{(i,j)}_j$ (line 7) since the read of previous try $HS^{(i,j)}_i$ (line 14).

In the first case, $P_j$ writes $segment[j]$ (line 7) during $P_i$’s double-collect (lines 15,16)

In the second case, $P_j$, Handshaking Property 1 implies that there has been a write of try $HS^{(i,j)}_j$ (line 7) since the read of previous try $HS^{(i,j)}_i$ (line 14).

Note:

1. Whichever $scan()$ returns the value of line 24, that $scan()$ has to be linearized.

2. Each processor has embedded $scan$’s of maximum $n$ processors.

Lemma 2  An indirect scan returns the view of a direct scan whose execution is enclosed within the execution of the indirect scan.

Proof: Let $P_i$ be the process to execute an indirect $scan$ (line 19), by borrowing the view of process $P_j$. So $P_i$ see’s the condition TRUE in line 17 for some process $P_j$ three times.

By Lemma 1, $P_j$ performs line 7 or line 8 three times. So, the third execution of line 7 or line 8 is part of an $update$ operation that starts after $P_i$’s $scan$ starts. The first two executions of line 7 or line 8 could still be part of the same $update$ operation (line 7 and then line 8), which may have started before $P_i$’s $scan$ starts. In other words, since $P_i$ sees the condition TRUE in line 17 three times, it follows that between two consecutive executions of line 13 by $P_i$, the process $P_j$ starts a new instance of $update$. Hence, it follows that the view returned by $P_i$ in line 19 is embedded in an $update$ operation whose execution is enclosed within the the indirect $scan$ by $P_i$ since it is in line 6.

If the embedded $scan$ is direct, then we are done. Otherwise, the argument is applied inductively, with at most $n - 1$ nested $scan$’s. Eventually, an embedded $scan$ is direct and the result follows from transitivity of containment of the embedded $scan$’s.

Lemma 3  A direct scan in $\alpha$ returns the actual value of the snapshot object in the configuration immediately following the last read in the first collect of the successful double-collect.

Proof: Suppose $P_i$ does a direct $scan$. Now consider $P_i$’s execution during lines 13-24. We define the various operations as follows:
Lemma 4  
Every scan returns a view that is the actual state of atomic snapshot object at the linearization point of the scan.

Proof: A direct scan is linearizable immediately after the last read of first collect in the successful double-collect. An indirect scan is linearized at the point of the embedded direct scan, whose value is borrowed by the indirect scan. This direct scan exists as per Lemma 2. Hence, both kinds of scan’s are linearized within their intervals. By Lemma 3, the actual value of the snapshot object at that linearization point is the value returned.

1.4 Linearizability and Wait-Freedom

Lemma 5  Both the update and scan operations are linearizable.

Proof: An update operation by \( P_i \) is linearized at the point of the embedded write to \( \text{segment}[i] \). By Lemma 4, the values returned by a scan is the configuration of the segments after the last read of the first collect, which is its linearization point for the scan’s. Therefore, the scan returns the value of the last update to each segment in the linearization order. This proves linearizability.

Lemma 6  Both the update and scan operations are wait-free.

Proof: The scan by \( P_i \) will complete no later than when three unsuccessful double-collects are seen for some \( P_j \). By Pigeon-Hole Principle, it is implied that in \( 2n + 1 \) double-collects, at least one update was successful i.e at least one \( P_j \) caused three double-collects (2 incrementing \( \text{shook}[j] \) and 1 borrowing \( P_j \’s \) view), which would terminate the scan. So, scan is wait-free. All update’s have an embedded scan and since scan’s are wait-free, so are the update’s.

Lemma 7  Every scan or update operation by process \( P_i \), returns after \( O(n^2) \) atomic steps of \( P_i \), \( \forall i \in \{1, 2, ..., n\} \)

Proof: Each scan by processor \( P_i \) can have at most \( n - 1 \) nested scan’s in the worst case if all the remaining \( n - 1 \) processors are doing an update inside the scan. The last scan will definitely have a clean double-collect. Thus, each high-level operation performs \( O(n^2) \) low level operations.