1 Introduction

Last time we saw that two-processor (wait-free) consensus cannot be solved by using just read/write registers. This gives read/write registers a consensus number of 1 and puts them in level 1 of the consensus hierarchy.

In this lecture we look at some objects that have a consensus number $\geq 2$.

2 Test and set

Theorem 1 Theorem: Test and Set can solve two-processor consensus.

Proof: An algorithm to solve two-processor consensus using a Test and set register (and read/write registers) is given below.

Algorithm 1 Code for Processor $P_i$, $i = 0,1$.

1: $val[i] = x_i$
2: if test&set($R$) then
3: \hspace{1cm} $y_i := x_i$
4: else
5: \hspace{1cm} $y_i := val[1 - i]$
6: end if

In line 1 above, $val$ is a 2-element array, implemented using two read/write atomic registers.

Theorem 2 Test and Set cannot implement three-processor consensus.

Proof Hint: Use case analysis as in the infinite bivalency sequence for Read/Write registers. It follows from the above two theorems that test and set objects have a consensus number of 2.
3 FIFO Queue

Theorem 3 2-dequeuer FIFO queue can solve two-processor consensus.

Proof: An algorithm to solve 2-processor consensus using a FIFO queue (and read/write registers) is given below.

Algorithm 2 Code for Processor $P_i$, $i = 0, 1$.

Initially: Queue $Q = \langle \text{Win, Lose} \rangle$

1: $val[i] = x_i$
2: $outcome := \text{deq}(Q)$
3: if $outcome = \text{Win}$ then
4: $y_i := x_i$
5: else
6: $y_i := val[1 - i]$
7: end if

In line 1 above, $val$ is a 2-element array, implemented using two read/write atomic registers.

Corollary 4 There is no wait-free simulation of a FIFO Queue with read/write objects for any number of processors.

This follows from the above algorithm, because if there was a wait-free simulation of FIFO queue with read/write registers, then there would be a 2-processor consensus algorithm using only read/write objects—which is a contradiction.

Theorem 5 There is no three-processor, wait-free consensus algorithm using only FIFO queues and read/write objects.

Proof: We will prove that in any bivalent configuration, there exists at least one non-critical processor. The proof then follows using an inductive construction similar to the one used in the read/write impossibility argument of the previous lecture.

Let $C$ be some bivalent configuration, assume for contradiction that all three processors $P_0$, $P_1$ and $P_2$ are critical in $C$, i.e. $P_0(C)$, $P_1(C)$ and $P_2(C)$ are all univalent. At least two of these need to have different decision values, because $C$ is bivalent. Without loss of generality, let $P_0(C)$ be 0-valent and $P_1(C)$ be 1-valent. Let us analyze the various cases based on the next step taken by $P_0$ and $P_1$ in $C$:

Case 1: They either access different registers or both read from the same register. This is same as case 1 in the proof for read/write objects in the previous lecture.
**Case 2:** One processor reads and the other writes to the same register. This is same as case 2 in the proof for read/write objects in the previous lecture.

**Case 3:** Both processors dequeue. Referring to Figure 1, it is clear that the queues in configuration $D$ and $E$ will be identical. Since the state of processor $P_2$ is also identical in $D$ and $E$, therefore $D \overset{P_2}{\Rightarrow} E$. But, as $D$ is 0-valent and $E$ is 1-valent, $D \overset{P_2}{\Rightarrow} E$ is a contradiction to Lemma 2 of the previous lecture.

**Case 4a:** One processor enqueues while the other dequeues with a non-empty initial queue. Without loss of generality let $P_0$ enqueue and $P_1$ dequeue. From Figure 2, it is clear that the actions of $P_0$ and $P_1$ will ‘commute’ to end up in the same configuration $F$. But this implies that $F$ is both 1-valent as well as 0-valent, which is a contradiction.

**Case 4b:** One processor enqueues while the other dequeues with an empty initial queue. Without loss of generality let $P_0$ enqueue and $P_1$ dequeue. Referring to Figure 3, it is clear that the queue is empty in either $F$ or $E$, and since the state of processor $P_2$ is identical in $F$ and $E$, therefore $F \overset{P_2}{\Rightarrow} E$. But, as $F$ is 0-valent (reachable from $D$) and $E$ is 1-valent, $F \overset{P_2}{\Rightarrow} E$ is a contradiction to Lemma 2 of the previous lecture.
Figure 3: Case 4b—$P_0$ enqueues and $P_1$ dequeues with empty queue.

**Case 5:** $P_0$ and $P_1$ both enqueue. Let $P_0$ enqueue $a$, and $P_1$ enqueue $b$, with $k - 1$ elements initially in the queue of configuration $C$. Referring to Figure 4, $E = P_1(P_0(C))$ and $E' = P_0(P_1(C))$.

Consider a finite $P_0$-only schedule, $\sigma$, starting from $E$ which results in $P_0$ deciding 0. Such a schedule exists because of wait-free-ness. Let $\sigma'$ be the longest prefix of $\sigma$ which does not contain the dequeue of the $k^{th}$ element by $P_0$. We will now show that $\sigma'$ is not equal to $\sigma$, i.e. $\sigma$ must contain a dequeue of the $k^{th}$ element.

Applying $\sigma'$ to both $E$ and $E'$ gives us $\sigma'(E)$ and $\sigma'(E')$. Note that $\sigma'(E')$ is admissible, because of the following:

1. the state of $P_0$ was the same in $E$ and $E'$, and
2. the first $k - 1$ elements in the queue are identical in both $E$ and $E'$, and
3. $\sigma'$ contains fewer than $k$ dequeues.

Clearly, the state of $P_0$ is the same in $\sigma'(E)$ and $\sigma'(E')$. This means that $P_0$ cannot have decided at the end of $\sigma'$. Thus $\sigma$ must contain at least one more step than $\sigma'$. By the definition of $\sigma'$, the next step in $\sigma$ following $\sigma'$ has to be a dequeue of the $k^{th}$ element. We let $P_0$ take this next step to reach configurations $F$ and $F'$ by dequeuing, respectively, $a$ from the queue in $\sigma'(E)$, and $b$ from the queue in $\sigma'(E')$. Again, note that this next step is legal in $\sigma'(E')$ because the state of $P_0$ is the same in $\sigma'(E)$ and $\sigma'(E')$. Note that elements $b$ and $a$ are now at the head of the queue in $F$ and the queue in $F'$ respectively—while all other elements are identical in both queues.

Now, consider a finite $P_1$-only schedule starting from $F$ which results in $P_1$ deciding 0. Again, such a schedule exists because of wait-free-ness. Let $\tau'$ be the longest prefix of $\tau$ which does not contain a dequeue by $P_1$. We will now show that $\tau'$ is not equal to $\tau$, i.e. $\tau$ must contain a dequeue.

Applying $\tau'$ to both $F$ and $F'$ gives us $\tau'(F)$ and $\tau'(F')$. Note that $\tau'(F')$ is admissible, because of the following:
1. the state of $P_1$ was the same in $F$ and $F'$, and

2. $\tau'$ contains no dequeues, so the differences in the queues of $F$ and $F'$ (only in the head element, actually) do not matter.

Clearly, the state of $P_1$ is the same in $\tau'(F)$ and $\tau'(F')$. This means that $P_1$ cannot have decided at the end of $\tau'$. Thus $\tau$ must contain at least one more step than $\tau'$. By the definition of $\tau'$, the next step in $\tau$ following $\tau'$ has to be a dequeue. We let $P_1$ take this next step to reach configurations $G$ and $G'$ by dequeuing, respectively, $b$ from the queue in $\tau'(F)$, and $a$ from the queue in $\tau'(F')$. Again, note that this next step is legal in $\tau'(F')$ because the state of $P_1$ is the same in $\tau'(F)$ and $\tau'(F')$.

With the removal of the differing head elements, all elements are identical in the queue of $G$ and the queue of $G'$. Since the state of $P_2$ is also identical in $G$ and $G'$—as it has taken no steps since $C$—therefore $G \not\sim G'$. But, as $G$ is 0-valent and $G'$ is 1-valent (reachable from $D$ and $D'$ respectively), $G \not\sim G'$ is a contradiction to Lemma 2 of the previous lecture.

Figure 4: Case 5—both enqueue.
4 Compare and Swap

This object has a consensus number of $\infty$, since it can solve n-processor consensus for any n.

4.1 Sequential specification

\texttt{compare\&\texttt{swap} (X: memory address; old, new: value) returns value}

1: previous := contents of X
2: \textbf{if} previous = old \textbf{then}
3: contents of X := new
4: \textbf{end if}
5: return previous

4.2 Implementation of consensus for any number of processors

The following algorithm shows how a compare and swap object can be used to solve n-processor consensus, for any n.

\begin{verbatim}
Algorithm 3 Code for Processor \texttt{P}, 0 \leq i \leq n - 1
Initially: contents of \texttt{First} = ⊥
1: v := \texttt{compare\&\texttt{swap}}(First, ⊥, \texttt{x}_i)
2: \textbf{if} v = ⊥ \textbf{then}
3: \quad y_i := \texttt{x}_i
4: \textbf{else}
5: \quad y_i := v
6: \textbf{end if}
\end{verbatim}

References