

Lecture 6: Timestamps

Lecturer: Faith Ellen

Scribe: Lily Li

6.1 Lock-Freedom, Obstruction-Freedom

Definition 6.1 An implementation is **wait-free** if in every admissible execution, every non-faulty process completes its operations on the implemented object within a finite number of its own steps.

A **non-blocking/ lock-free** implementation is one where every configuration has some process that finishes its operation within a finite number of its own steps. Consider the following example.

Algorithm 1 Implementation of Fetch&Increment using Weak CAS

```

1: Repeat
2:  $v \leftarrow \text{read}(C)$ 
3: until  $\text{CAS}(C, v, v + 1) = \text{true}$ 
4: return  $v$ 

```

Obstruction-free/ solo-terminating: from every reachable configuration in which some process has a pending operation the process can complete its operation if it is given sufficiently many consecutive steps.

6.2 Timestamps (AW 220-222)

TimeStamp objects are used to record info about various operations or even occurring in relation to one another. Suppose T is a TimeStamp object. The function $\text{GetTS}(T)$ returns the values from a partially ordered set U such that if $\text{GetTS}(T)$ operation g_2 that returns t_2 is invoked after $\text{GetTS}(T)$ operation g_1 that returns t_1 . Then $t_1 < t_2$. If g_1 and g_2 are concurrent then any order of t_1 and t_2 is possible, even t_1 and t_2 are incomparable. Below is the implementation of a TimeStamp object: Algorithm 3.

6.3 Atomic Snapshots (AW 10.3)

This object has m -components S_0, \dots, S_{m-1} . It supports two operations $\text{update}(S, i, w)$ which sets S_i to have value w and $\text{scan}(S)$ which returns (S_0, \dots, S_{m-1}) . Observe that when $m = 1$ then we simply have a register. Let us consider how to implement these two operations so that they are linearizable.

Notice that the sequence numbers are really problematic since they are unbounded. Here is what we can do to improve the situation: the scanner is going to have to write (though in a really limited way). The problem is called the ABA problem (the register changed from A to B then back to A and we want a way to distinguish between the first and last A). What we need to do is implement a register that avoids the ABA problem without using sequence numbers and timestamps. For single writer/reader register R with reader p_r and writer p_w . We make use of two additional single bit registers H_w (single writer by p_w) and H_r (single reader by p_r). The new write will be:

Algorithm 2 Implementation of TimeStamp Object

```

1: # Using a Fetch and Increment
2: GetTS( $T$ )
3: returns Fetch&Increment( $T$ )
4:
5: # Using a Counter:  $U = \mathbb{N} - \{0\}$ 
6: GetTS( $C$ )
7: Increment  $C$ 
8: return read  $C$ 
9:
10: # Using a Fetch and Increment again (for processor  $i$  when there are a total of  $n$  processors)
11: GetTS( $T$ )
12: Increment  $C$ 
13: return  $n \cdot \text{read}(C) + i$ 
14:
15: # Use registers for process  $p_i$ 
16: GetTS( $R$ )
17:  $y = 0$ 
18: for  $j$  from 0 to  $n - 1$  do
19:    $y \leftarrow \max y, \text{read}(x_j)$ 
20: end for
21:  $y \leftarrow y + 1$ 
22:  $x_i \leftarrow y$ 
23: return  $y$ 
24:
25: # Vector timestamps  $U = \mathbb{N}^n$ 
26: GetTS
27: for  $j$  from 0 to  $n - 1$  do
28:    $v_j \leftarrow \text{read}(x_j)$  # this is called a collect
29: end for
30:  $v_i \leftarrow v_i + 1$ 
31:  $x_i \leftarrow v_i$ 
32: return  $v$ 
33: # Here we would use lexicographical or component-wise for  $v$ 
34:
35: # Bounded timestamps: process only has timestamp from its last GetTS operation
36:  $n = 2$ 
37:  $U = \{0, 1, 2\}$  and  $0 < 1, 1 < 2, 2 < 0$ 
38: GetTS by process  $p_i$ 
39:  $x \leftarrow \text{read}(T_{1-i})$ 
40:  $T_i \leftarrow \text{write}(x + 1 \bmod 3)$ 

```

Algorithm 3 Implementation of TimeStamp Object

```

1: Scan

```

Algorithm 4 Implementation of TimeStamp Object

```

1: WRITE( $R, v$ ) by  $p_w$ 
2: write( $R, v, processIDofwriter$ )
3:  $h \leftarrow \text{read}(H_r)$ 
4: write( $H_w, 1 - h$ )
5:
6: READTWICE( $R$ ) by  $p_r$ 
7:  $h \leftarrow \text{READ}(H_w)$ 
8: write( $H_r, h$ )
9:  $r \leftarrow \text{read}(R)$ 
10:  $r' \leftarrow \text{read}(R)$ 
11:  $h' \leftarrow \text{read}(H_w)$ 
12: if  $r = r'$  and  $h = h'$  then
13:   return True
14: else
15:   return False
16: end if

```

Claim 6.2 If *READ – TWICE* return *F* then either the value of R changed between the two reads of R or the value of H_w changed between the reads of H_w .

Claim 6.3 If *READ – TWICE*(R) returns *True* then the value of T did not change between the two reads of R .

Proof: Suppose that *READ – TWICE*(R) returns *True* and $r = r'$ and $h = h'$. If you think about it a little bit you see that it is impossible to have some writes between the two reads. Every time a write occurs the writer will force H_w to differ from H_r . Since only w can write to H_w there is no way that $H_w = H_r$ until r successfully finished a write. Thus the reader can detect any writes that occurred. ■

The way to extend this construction to n processes who behave as readers and writers is to have such a pair of bits for every pair of process (actually you want two pair for process pair since both might want to be readers and writers).

6.3.1 Handshaking (AW 10.3.1)

Actually we already saw an implementation of handshaking from the above involving registers. This idea can be further implemented to Atomic-Snapshot objects to remove the need of the sequence numbers.