CSC2221: Introduction to Distributed Computing **Fall 2017**

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 ← read(C)
3: until CAS(C, v, v + 1) = 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 GetTS(T) returns the values from a partially ordered set U such that if GetTS(T) operation g_2 that returns t_2 is invoked after 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. Algorithm 2 shows several implementation of a TimeStamp object.

6.3 Atomic Snapshots (AW 10.3)

This object has m-components $S_0, ..., S_{m-1}$. It supports two operations update(S, i, w) which sets S_i to have value w and scan(S) which returns $(S_0, ..., S_{m-1})$. 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. See Algorithm 3.

A similar way to define a atomic snapshot object according to the book is the following: we have n users indexed with i where $0 \leq i \leq n-1$. There are two kinds of operations as before:

scan$_i$: invocation by user i returns V which is an n-element vector called a view (with a value for each of the n segments corresponding to each user).

update$_i(d)$: invocation by user i returns ack_i and data d is written to p_i's segment.

Algorithm 2 Implementation of TimeStamp Object

```
# Using a Fetch and Increment
GetTS(T)
returns Fetch&Increment(T)

# Using a Counter: U = ℕ − {0}
GetTS(C)
Increment C
return read C

# Using a Fetch and Increment again (for processor i when there are a total of n processors)
GetTS(T)
Increment C
return n · read(C) + i

# Use registers for process p_i
GetTS(R)
y = 0
for j from 0 to n − 1 do
    y ← max{y, read(x_j)}
end for
y ← y + 1
x_i ← y
return y

# Vector timestamps U = ℕ^n
GetTS(U)
for j from 0 to n − 1 do
    v_j ← read(x_j) # this is called a collect
end for
v_i ← v_i + 1
x_i ← v_i
return v
# Here we would use lexicographical or component-wise order for v

# Bounded timestamps: process only has timestamp from its last GetTS operation
n = 2
U = {0, 1, 2} and 0 < 1, 1 < 2, 2 < 0
GetTS by process p_i
x ← read(T_{1−i})
T_i ← write(x + 1  mod 3)
```

The goal is to simulate an atomic snapshot object form single-reader, single-writer read/write registers of bounded size. A sequence of scan and update operations are allowable if and only if for each view V returned by a scan operation, $V[i]$ equals the parameter of the most recent *preceding* update_i for each i (or the initial value of p_i if no update_is occurred). The really clever idea for the scan operation is to collect (i.e. read) all the segments twice in a process called *double collect*. The double collect serves the purpose of carving out a chunk of time ripe for linearization. In particular, if both collects are the same then we know no changes were made during this interval of time and the scan can proceed as normal. However double collect requires two considerations: (1) detecting when a change occurred between the two collects and (2) figuring out what to do when change is detected.

There are two ways to handle (1). A simple solution would be to just tack on a sequence number to the end of every chunk of data. Every update would also update the sequence number so a change is easy to detect. The down side of the strategy is the unbounded sized registers needed to store the sequence numbers. An alternative (just read: better) approach would be to implement a *handshaking mechanism*. Here, every change can be thought of as a pair of hands being shaken. The first collect extends a hand. A change can be thought of as another processor reaching out and shaking the outstretched hand. One this handshake occurred any further attempts to shake the hand will fail. In addition the second collect can just look to see if the outstretched hand is there to see if any changes have occurred. This mechanism does not tell us how many changes have occurred, simply "$\geq$ one change must have occurred", but that's just what we need!

Algorithm 3 Implementation of Atomic Snapshots Object using unbounded registers: code for process p_i.

```
1:  update_i(S, d) :
2:  s ← scan()
3:  S[i] ← ⟨S[i].count + 1, d, s⟩
4:
5:  scan_i(S) :
6:  inital ← collect(S)
7:  previous ← initial
8:  while true do
9:      s ← scan(S)
10:     if s = previous then
11:         return s
12:     else if there exist a j such that s[j].count ≥ initial[j].count + 2 then
13:         return s[j].snapshot
14:     else
15:         previous ← s
16:     end if
17: end while
```

Observe that the step complexity of a process for $\mathsf{scan}(S)$ is $O(n^2)$ — it is actually exactly n^2+1 can you see why? — and n^2 for each update operation. Let us show that this algorithm is linearizable. Namely, we will find a linearization point for each operation. Looking at the algorithm we see that there are really two types of operations. Ones which terminate the if loop through the if-clause and ones that exist the if loop through the else if-clause (since all operations requires a scan, there is no real distinction between scan and update). For operations of the first type, choose a point in time between the two collects (we already explained why this works. For operation of the second type, place the linearization point just after the linearization point of the operation whose scan we just returned.

Notice that the sequence numbers are really problematic since they are unbounded. Here is what we can do to improve the situation: the scan is going to have to write (though in a really limited way). The problem is call 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. We do so using handshaking procedures.

6.3.1 Handshaking (AW 10.3.1)

Let R be a single writer/reader register 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 writer by p_r). The new write will be:

Algorithm 4 Implementation of Handshake Object

```
1: WRITE(R, v) by p_w
2: write(R, v, process ID of writer)
3: h ← read(H_r)
4: write(H_w, 1 − h)
5:
6: READTWICE(R) by p_r
7: h ← READ(H_w)
8: write(H_r, h)
9: r ← read(R)
10: r' ← read(R)
11: h' ← 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 $READ-TWICE(R)$ return $False$ 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). This idea can be further implemented in Atomic-Snapshot objects to remove the need for unbounded size sequence numbers.