CS465 Distributed Systems Problem Set 3

Sherwin Yu

December 2, 2011

1 Consensus number of a restricted queue

The consensus number for the restricted queue is 1. We show this with a proof similar to shared memory FLP in which we show that a bivalent configuration (an initial one must exist by the same logic as the original proof) can never lead to exclusively univalent (0-valent and 1-valent) successors. Assume the opposite. Let C be a bivalent configuration. Then an operation x performed by process x, and an operation y performed by process y, must exist such that that Cx is 1-valent Cy is 0-valent without loss of generality. x and y must be operations of the same object, assuming objects do not interact behind the scenes, otherwise Cxy = Cyx and we get a contradiction, as one is 1-valent and the other is 0-valent. In all the cases below, the contradictions we arrive at center around the fact that Cx is 1-valent and Cy is 0-valent, which must be distinguishable to all processes (otherwise they would still be bivalent).

- x and y are both enq. Then we have either C enq(x) enq(y) or C enq(y) enq(x). Run process x until it is about to do a deq(), which it must, as otherwise it can't tell what to decide). In one configuration, allow the deq to go thorugh and in the other, kill process x. This is indistinguishable to process y. Contradiction.
- X and Y are enq/deq. The queue is empty. Let operation x be enq(x) and operation y be deq(y), WLOG. Then we have either C enq(x) deq(y) or C deq(y) enq(x). However, deq(y) will return false and do nothing else, and then these two configurations are indistinguishable to x; contradiction.
- X and Y are enq/deq. The queue's first element is X, WLOG. Let operation x be enq(x) and operation y be deq(y). Then we have either C enq(x) deq(y) or C deq(y) enq(x). However, deq(y) will return false and do nothing else, and then these two configurations are indistinguishable to x; contradiction.
- x and y are enq/deq. The queue's first element is X, WLOG. Let operation x be deq(x) and operation y be enq(y). Then we have either C deq(x) enq(y) or C enq(y) deq(x). However, as x is already the head and enq(y) doesn't change the head, these two operations commute, so C deq(x) enq(y) = C enq(y) deq(x) and are indistinguishable. Contradiction.

- x and y are both deq. The queue's first two elements of the queue are X, X, WLOG. Then we have Cdeq(x)deq(y) or C deq(y) deq(x). Deq(y) always returns false without doing anything else, so these configurations are indistinguishable to process x. Contradiction.

- x and y are both deq. The queue's first two elements of the queue are X, Y, WLOG. Then we have Cdeq(x)deq(y) or C deq(y) deq(x). In Cdeq(x)deq(y), if we kill y before it is able to dequeue, then to process x the configuration is indistinguishable from C deq(y) deq(x), as the deq(y) returns false and doesn't do anything else. Contradiction

- x and y are both deq. The queue's has one element, X, WLOG. Then we have C deq(x) deq(y) or C deq(y) deq(x). Then the call to deq(y) will always fail and not do anything eles, so the two configurations are indistinguishable to process x. Contradiction.

- x and y are both deq. The queue is empty. Then we have Cdeq(x)deq(y) or C deq(y) deq(x). Then all deq calls fail and the configurations are indistinguishable to both x and y. Contradiction.o

Thus, there is always a bivalent successor, and consensus with two restricted queues is impossible by FLP. It follows then that the consensus number of the restricted queue is 1.

2 Writable fetch-and-increment

We implement the writable fetch-and-increment by using normal fetch-and-increment registers to account for atomic fetch-and-increment calls, and an additional atomic register to hold an offset for the write. Our object's augmented fetch-and-increment will then call the underlying register's fetch-and-increment, sum that with the offset, and return it. However, doing a write requires resetting both the current fetch-and-increment and setting the offset register, atomically. To get around this, we take advantage of our unbounded number of registers. We keep two arrays: $FAIs$, an endless array of fetch-and-increment registers and $offsets$, an endless array of atomic registers, and also maintain an index (as a fetch-and-increment) into both arrays. Everytime we do a write, we move to a new index and overwrite the new offset (assume $FAIs[i]$ is initialized to 0 for all i).

Formally:

Init: (shared)
FAIs[i] = 0 for all i // an array of fetch and increment registers
offsets[i] = 0 for all i // an array of atomic registers
indexFetchAndIncrement = 0 // a fetch and increment register
index = 0 // an atomic register

write(new-value): (process i)

```
    index = indexFetchAndIncrement.fetchAndIncrement();
    offset[index] = new-value

augmentedFetchAndIncrement(): (process i)
    localindex = index // local variable in the local memory of proc i
    a = offset[localindex]
    b = fai[localindex].fetchAndIncrement()
    return a + b
```

AugmentedFetchAndIncrement simply calls the fetchAndIncrement for the underlying register (whichever one we're currently at) and combines the return value with the offset (which was determined by the most recent write, or 0 at initialization). Write increments the indexFetchAndIncrement register, storing the new index in a separate index variable (which is used in augmentedFetchAndIncrement). This causes us to move to a new offset, containing the value of our write, and new fetch-and-increment register, which is at zero, thereby fulfilling the write operation.

These operations are linearizable and interleaving them does not cause a problem. Interleaving two augmentedFetchAndIncrement() calls is safe because the only operation that is writes to shared memory (and thus does not commute) is the call to the fai[localindex].fetchAndIncrement(), but this is fine because this operation is atomic. Interleaving an augmentedFetchAndIncrement() with a write(x) is also safe because even though we have multiple atomic actions in each call, only one from each matters: pulling the shared *index* into *localindex*, and changing the shared *index* to $indexFetchAndIncrement.fetchAndIncrement()$. If the former happens first, then the write occurs first, and the entire fetch and increment will be applied to the new register indexes. If the latter statement happens first, then this is equivalent to having calling the fetch and increment immediately followed by a write (overwriting the fetchAndIncrement) because the write moves us to a new (fresh) pair of registers (the fetchAndIncrement will still execute, possibly even after the write, but it is already writing on outdated registers; future reads will pull the new *index* into *localindex*. Interleaving two writes is also fine: whichever write operation executes offset[index] = new-value second occurs second, and is also the one that overwrites the previous write, because *index* is a shared register.

The protocol is clearly wait-free: we are never blocking or spinning, and we've already shown that having other operations ocur by other processes does not affect the correctness of our protocol.

3 A box object

We implement this object with double-collecting snapshots on $2n$ atomic registers. Similar to implementing a counter out of snapshot (19.5.2), except we are snapshotting two variables simultaneously (both width and height). Details:

Init (shared):

```
    registers[1..n] = 0 // represent width
    registers[n+1..2n]= 0 // represent height
Init (process i):
   myIncWidths = 0; // keep track of how many increments this process has
done
    myIncHeights = 0;

incWidth(): (process i):
    myIncWidths = myIncWidths + 1
    registers[i] = myIncWidths

incHeight(): (process i):
    myIncHeights = myIncHeights + 1
    registers[i] = myIncheights

getArea(): (process i)
    arr = snapshot(registers)
    width = sum(arr, from 1 to n) // operations local to i
    height = sum(arr, from n+1 to 2n)
    return height*width
```

This implemetation is obstruction free. We differ from the snapshot object created from atomic registers described in The Gang of Six algorithm (19.2) in that our updates do not require writers to perform a snapshot; instead, we implement obstruction-free snapshots with double collects (25.2.2). The only line that might be obstructed is the call to snapshot(). But to keep a snapshot going forever, we need infinitely many updates to interrupt the double collects; if other processes halted; then the double collects would succeed, snapshot() would go thorugh, and getArea would terminate. So getArea is obstruction free. The other functions are obstruction free by inspection. The worst-case step-complexity of getArea is $O(N)$. If the snapshot operation is left to proceed unobstructed, then we it will read $2N$ registers once, then $2N$ registers again for the double collect. This is a total of $4N$ reads, which is $O(N)$ steps. I'm assuming we are ignoring the local operations (summing and multiplying). But even if we aren't, summing for width and height are also both $O(N)$. We use the JTT lower bound for perturbable objects (Chapter 21) to show that this is within a constant factor of optimal. First, note that we must have at least N registers to accomodate N processors: to see why this is, consider the case when we have fewer than N registers. Then at least two processors will have to share a register to write values (for incrementing height/width). Because we are working with atomic registers, we only have atomic reads and atomic writes. So it is possible for two increments to interleave (both read old value, both write old+1, but new value should be old+2). By this reasoning, we always need at least N registers for N processes. Then, because atomic registers are perturbable objects, applying the JTT bound gives us a lower bound of $N-1$ space and $N-1$ steps for a read operation in the worst case. $N-1$ is within a

constant factor of the worst-case step-complexity for getArea ($O(N)$).