



Principles of Distributed Computing

Exercise 5: Sample Solution

1 Shared Sum

In the following, let X (initialized to 0) always denote the shared register used to hold the sum $x = \sum_{i=1}^n x_i$, and assume that all x_i (and thus also x) are initially 0. Denote by Δx_i the amount by which x_i is changed by process p_i at some time, i.e., if $x_i := x'_i$ is assigned by p_i , then $\Delta x_i = x'_i - x_i$.

- a) To update x , p_i calls `fetch-and-add($X, \Delta x_i$)`. Therefore, X changes exactly the same as x_i and holds the correct value. Since no process has to wait or retry, we have neither lockouts nor deadlocks. A simple read on X (or `fetch-and-add($X, 0$)`) gets the current value of x .
- b) An update is done by the following code:

```
1:  $x := X$ 
2: while not compare-and-swap( $X, x, x + \Delta x_i$ ) do
3:    $x := X$ 
4: end while
```

The loop is left after X changed by Δx_i exactly once, thus the code is correct. Again, x can be obtained by a simple read. Since the compare-and-swap may only fail if another process p_j changed the value of X between p_i reading it and calling compare-and-swap, there is no deadlock. However, other updates may delay a change by some p_i indefinitely, hence lockouts are possible.

- c) A write is implemented by

```
1:  $x := \text{load-link}(X)$ 
2: while not store-conditional( $X, x + \Delta x_i$ ) do
3:    $x := \text{load-link}(X)$ 
4: end while
```

and is correct for the same reasons as in **b**). Reads are again simple. However, the solution differs from **b**) in that we may have deadlocks, since e.g. two processes can fight endlessly for getting the register linked to them for sufficiently long to write a value (At least for the case where a load-link can destroy the link to the register of another processor; i.e. weak LL/SC).

- d) It can be done. We use a special encoding on X . Either it stores a regular value and \perp (i.e., (x, \perp)) or the value and an additional identifier $id(i)$ of a process p_i . A node will effectively acquire a lock on X by writing its ID to X and only afterwards write its update to X .

When x_i is changed, p_i executes

```
1: while true do
2:    $(x, id) := X$  // simple read
3:    $(x, id) := \text{compare-and-swap}(X, (x, \perp), (x, id(i)))$  // try to lock  $X$  with own ID
4:   if  $id = id(i)$  then
5:      $X := (x + \Delta x_i, \perp)$  // regular write, but compare-and-swap would also do
6:     break while loop
7:   end if
8: end while
```

Because writing by compare-and-swap works only if the second argument equals the value of the register, once a process “locks” X with its identifier, no other process may do so until the same process performs the write enclosed in the if-condition. Thus, this write happens exactly if the compare-and-swap was successful. The only reason to check the identifier by an if-statement rather than using compare-and-swap again is that we need to ensure that the process leaves the loop after changing X by Δx_i . On the other hand, the while loop can only be left after a succesful write, thus X is updated correctly. Reads are again plain reads.

As before, the solution is free of deadlocks: At least one process can write, because after each write the ID part of X contains \perp , i.e., one process will succeed in “locking” X . As in **b)** and **c)**, the solution is prone to lockouts.