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 $\Delta x_i$ 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:

```plaintext
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 $\Delta x_i$ 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

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

and is correct for the same reasons as in b). Reads are again simple.

d) It can be done. We use a special encoding on $X$. Either it stores a regular value (i.e., $x$) or an identifier $id(i)$ of a process $p_i$ in a distinguishable manner (e.g., marked by the first bit). 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

```plaintext
1: while true do  
2: \( x := X \)  
3: if \( x \) is not an identifier then  
4: \hspace{1em} id := compare-and-swap(\( X, x, id(i) \)) //write own ID to \( X \)  
5: end if  
6: if id = id(i) then  
7: \hspace{1em} X := x + \Delta x_i  
8: \hspace{1em} break  
9: end if  
10: end while
```

The first if condition ensures that only one process at a time can “lock” \( X \) with its identifier, i.e., between the compare-and-swap and the assignment of \( x + \Delta x \) to \( X \) no other process will change the value of \( X \). Moreover, the second if condition is true for a process \( p_i \) if and only if the compare-and-swap succeeded, i.e., \( p_i \) wrote its identifier to \( X \). This means, that \( X = x \) before the assignment \( X := x + \Delta x_i \), implying that \( X \) is changed by \( \Delta x_i \). Because of the while loop and its abort condition this happens exactly once, therefore \( X \) is updated correctly.

Since \( X \) now may temporarily contain an identifier instead of \( x \), the read may also have to wait:

```plaintext
1: x := X  
2: while x is an identifier do  
3: \hspace{1em} x := X  
4: end while
```

Again, the solution is free of deadlocks when considered as a whole: At least one process can write, because if \( X \) is not changed the compare-and-swap must succeed, and if no process writes then all reads succeed. However, if nodes keep on writing all the time, reads may consistently fail. As in b) and c), the solution is prone to lockouts.

## 2 Space Efficient Binary Tree Algorithm

a) In any splitter, in expectation at least half of the active processes decide differently. Thus, with probability at least 1/3, at least one quarter of the processes decide differently (cf. the proof of Theorem 4.11). Therefore, by linearity of expectation (Theorem 4.9), after 3 log_{4/3} \( k \) nodes the number of expected remaining processes is at most 1, implying that a particular process will stop after at most this number of expected steps.

b) Analogously to the proof of Corollary 4.14.

c) Since w.h.p. means with probability 1 - 1/k\( c \) for arbitrary \( c \geq 1 \), in particular we have that each individual process will stop after \( O(\log k) \) many steps with probability 1 - 1/k\( c+1 \) for any choice of \( c \geq 1 \). Thus, the probability that any of the \( k \) processes will take more steps to stop is at most \( \sum_{i=1}^{k} 1/k^{c+1} = 1/k^c \). In other words, the depth of the subtree induced by the marked nodes is w.h.p. at most \( O(\log k) \) as claimed.