Tutor: Julien Dallot

Exercise Sheet 5

Shared Memory

Exercise 1. Consider Lamport's "bakery algorithm" in the asynchronous setting. We use the notation $(a_1, a_2) < (b_1, b_2)$ if and only if $a_1 < b_1$ or $a_1 = b_1$ and $a_2 < b_2$ (similar to comparing two 2-digit numbers by looking at their digits). Let us assume that we have N nodes. Furthermore, assume that we have a shared memory initialized as follows: choosing$[i] \leftarrow$ false, number$[i] \leftarrow 0$, for each node $1 \leq i \leq N$. Each node i executes the following code:

```
 1: while true do
 2:     choosing[i] ← true
 3:     number[i] ← 1 + max_{1≤k≤N}(number[k])
 4:     choosing[i] ← false
 5:     for j = 1 to N do
 6:         while choosing[j] do
 7:             nothing
 8:         end while
 9:         while number[j] ≠ 0 and (number[j], j) < (number[i], i) do
10:             nothing
11:         end while
12:     end for
13:     perform critical section
14:     number[i] ← 0
15:     perform noncritical section
16: end while
```

1. Show that this algorithm solves the *concurrent programming problem*:
   (a) At any time, at most one computer may be in its critical section.
   (b) Each computer must eventually be able to enter its critical section (unless it halts).
   (c) Any computer may halt in its noncritical section.
2. How big can number$[i]$ get for each node i?
3. What happens if we remove line 3 and line 14, and replace the while-condition in line 9 with $j < i$?
4. What happens if in line 3, we instead just set number$[i] \leftarrow 1$?
5. Consider the following adjustment to the asynchronous model: If a read and write operation on the same part of the shared memory occur simultaneously, the write operation will return an arbitrary value. What happens to the algorithm?

Exercise 2 (Shared Sum)**.** Consider the following scenario: Each process p_i computes a local variable x_i and we want to make the sum $x := \sum_{i=1}^{n} x_i$ available to all processes.

We want to guarantee the following: If a process updates x_i, it should first ensure that x is updated accordingly before proceeding. However, we do not want to use a large number of registers or a huge register. In the following, you are given a single register that can store

$O(\log n)$ bits (the choice of the constant is up to you). Moreover, we assume that "x cannot become too large", i.e., the x_i (and thus x) are of size polynomial in n and hence can be encoded using $O(\log n)$ bits.

1. Give a solution using a shared register supporting the fetch-and-add operation with a constant update and access complexity. If possible, prevent both lockouts and deadlocks.

2. Give a solution using a compare-and-swap register, also with constant access complexity. If successful, an update should need a constant number of steps (otherwise the process may retry). Are lockouts excluded?

3. Give a solution using a load-link/store-conditional register. Compare it to the preceding solutions.

4. Assume now that the return value of compare-and-swap is not whether the operation succeeded, but the value stored in the register after the operation. Can the problem still be solved? Prove your claim!