ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

HS 2010

Prof. R. Wattenhofer / R. Meier

Distributed Systems Part II

Exercise Sheet 9

1 Spin Locks

A read-write lock is a lock that allows either multiple processes to read some resource, or one process to write some resource.

a) Write a simple read-write lock using only spinning, one shared integer and the CAS operation. Do not use local variables (it is ok to have variable within a method, but not outside).

b) What is the problem with your lock?

Hint: what happens if a lot of processes access the lock repeatedly?

We now build a queue lock using only spinning, one local integer per process and the CAS operation.

c) To prepare for this task, answer the following questions:

i) Head and tail of the queue have to be stored in the shared integer. What is the "head" and the "tail", and how can they be stored in one integer?

Hint: could the head be a process id? Or is there a much easier solution?

ii) How could a process add itself to the queue?

Hint: you need the local integer of the process for this operation.

iii) When has a process acquired the lock?

iv) How does a process release the lock?

d) Write down the lock using pseudo-code. Do not forget to initialize all variables.

2 ALock2

Have a look at the source code below. It is a modified version of the ALock (slides 8/42 ff).

```
public class ALock2 implements Lock {
  int[] flags = {true,true,false,...,false};
  AtomicInteger next = new AtomicInteger(0);
  ThreadLocal<Integer> mySlot;

  public void lock() {
    mySlot = next.getAndIncrement();
    while (!flags[mySlot % n]) {}
    flags[mySlot % n] = false;
  }
```

```
  public void unlock() {
    flags[(mySlot+2) % n] = true;
  }
}
```

a) What was the intention of the author of "ALock2"?

b) Will ALock2 work properly? Why (not)?

c) Give an idea how to repair ALock2.

Hint: don't bother about performance.

3 MCS Queue Lock

See slides 8/61 ff.

a) A developer suggests to add an `abort` flag to each node: if a process no longer wants to wait it sets this `abort` flag to `true`. If a process unlocks the lock, it may see the `abort` flag of the next node, jump over the aborted node, and check the successor's successor node. Modify the basic algorithm to support aborts.

Optional: sketch a proof for your answer.

Hint: Be aware of race-conditions!

b) Assuming many processes may abort concurrently, does your answer from a) still work? Explain why. If it does not work: modify your algorithm to allow concurrent aborts.

Optional: sketch a proof for your answer.

c) Instead of a `locked` and an `aborted` flag one could use an integer, and modify the integer with the CAS operation. What do you think about this idea? How is the algorithm affected? How is performance affected?

d) The CLH lock (slide 8/49) is basically the same as an MCS lock. Conceptually the only difference is, that a process spins on the `locked` field of the predecessor node, not on its own node. What could be an advantage of CLH over MCS and what could be a disadvantage?