ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

FS 2018 Prof. R. Wattenhofer

Computer Engineering II

Exercise Sheet Chapter 10

We categorize questions into four different categories:

Quiz Short questions which we will solve rather interactively at the start of the exercise sessions.

Basic Improve the basic understanding of the lecture material.

Advanced Test your ability to work with the lecture content. This is the typical style of questions which appear in the exam.

Mastery More interesting, but also more challenging. These questions are harder and we do not expect you to solve such exercises during the exam. Simpler questions about the same topics are however possible in the exam.

Questions marked with $^{(g)}$ may need some research on Google.

Quiz

1 Quiz

a) In Java we can use an AtomicBoolean object to implement locking. How can the method getAndSet() in AtomicBoolean be implemented efficiently?

b) Is it possible to achieve mutual exclusion without any RMW primitives?

c) What purpose does the extra test in a TTAS lock serve? Why is it still not scalable?

Basic

2 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 global shared integer and the compare-and-set (CAS) operation. Do not use other global variables (it is ok to have a variable within a method, but not outside). In Java compare-and-set can be used as shown in the following example.

```
if (state.compareAndSet(expectedValue, newValue)) {
  // successful
}
```

You can use the following template to implement your locking algorithm.

```
// the shared integer
AtomicInteger state = new AtomicInteger(0);

// acquire the lock for a read operation
void read_lock() {
  ...
}

// release the lock
void read_unlock() {
  ...
}

// acquire the lock for a write operation
void write_lock() {
  ...
}

// release the lock
void write_unlock() {
  ...
}
```

b) What is the problem with this lock?

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

We now build a queue lock using only spinning, one shared integer, one local integer per process and the compare-and-set (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 are the "head" and the "tail", and how can they both be stored in one integer?

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

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

Hint: You need the global 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.

Advanced

3 ALock2

Have a look at the source code below. It is a modified version of the ALock for `capacity` processes (lecture notes page 150).

```
public class ALock2 implements Lock {
  ThreadLocal<Integer> mySlotIndex = new ThreadLocal<Integer> () {
    protected Integer initialValue() {
      return 0;
    }
  };
  AtomicInteger tail;
  boolean[] flag;
  int size;
  public ALock2(int capacity) {
    size = capacity;
```

```
    tail = new AtomicInteger(0);
    flag = new boolean[capacity];
    flag[0] = true;
    flag[1] = true;
  }
  public void lock() {
    int slot = tail.getAndIncrement() % size;
    mySlotIndex.set(slot);
    while (! flag[slot]) {};
  }
  public void unlock() {
    int slot = mySlotIndex.get();
    flag[slot] = false;
    flag[(slot + 2) % size] = true;
  }
}
```

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

b) Will ALock2 work properly? Is it still a FIFO queue? Why not?

c) Suggest a way to repair ALock2 so that it satisfies the FIFO property. You can use an object of the ALock class and assume that it is implemented.

Mastery

4 MCS Queue Lock

See lecture notes page 154 ff.

a) A developer suggests to add an `aborted` flag to each node: if the process does not want to wait, it sets this flag to `true`. To acquire the lock, the process now checks both the `locked` flag and also an already implemented `is_abort()` condition. If `is_abort()` returns true, the lock function should set the aborted flag to `true` and return false without acquiring the lock. If a process unlocks the lock, it may see the `aborted` flag of the next node, jump over the aborted node, and check the successor's successor node. Modify the basic algorithm to support aborts. Assume for this part that atmost one node in the queue is aborted.

Optional: sketch a proof for your answer.

Hint: Be aware of race-conditions!

b) Modify a) when more than one node in the queue can be aborted simultaneously.

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 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?