ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

HS 2017 Prof. R. Wattenhofer

Distributed Systems Part II

Solution to Exercise Sheet 9

Quiz

1 Quiz

a) For consensus we considered distributed participants without shared memory, which could crash or even be Byzantine. The focus was on designing algorithms that could always progress, even if some messages or nodes were lost.

For locking we only consider uncrashable, faithful processes with shared memory and are interested in the practical performance on real-life system architectures.

b) Mutual exclusion is possible using only shared memory! The classic example is *Peterson's algorithm* (for 2 processes only):

```
boolean[] interested = {false, false};
int lockowner = 0;
const int myid; // 0 or 1

void lock() {
  interested[myid] = true;
  lockowner = 1 - myid;
  while (interested[1 - myid] && lockowner == 1 - myid);
}

void unlock() {
  interested[myid] = false;
}
```

This idea can also be generalized to an arbitrary amount of processors.

c) The extra test avoids writes (such as the TAS operation) and thus traffic on the bus while the lock is held. However, as all waiting processes are spinning on the same memory location an *invalidation storm* erasing each of their cache lines will occur upon lock release.

Basic

2 Spin Locks

a) We use the shared integer `state` to indicate the state of the lock. The lock is free if `state` is 0. The lock is in write mode if `state` is -1. And it is in read-mode if `state` is n, with $n > 0$.

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

// acquire the lock for a read operation
void read_lock() {
  while (true) {
    int value = state.read();
    if (value >= 0) {
      if (state.compareAndSet(value, value + 1)) {
        // lock acquired
        return;
      }
    }
  }
}

// release the lock
void read_unlock() {
  while (true) {
    int value = state.read();
    if (state.compareAndSet(value, value - 1))) {
      return;
    }
  }
}


// acquire the lock for a write operation
void write_lock() {
  while (true) {
    int value = state.read();
    if (value == 0) {
      if (state.compareAndSet(0, -1)) {
        // lock acquired
        return;
      }
    }
  }
}

// release the lock
void write_unlock() {
  // no need to test, no other process can call this at
  // the same time.
  state.compareAndSet(-1, 0);
}
```

b) Starvation is a problem. Example: if many processes constantly acquire and release the read-lock, then the `state` variable always remains bigger than 0. If one process wants to acquire the write-lock, it will never get the chance.

c) The basic idea behind this lock is a ticketing service as can be found in Swiss post offices.

   i) The tail is the ticket which can be drawn by the next process. The head denotes the ticket which can acquire the lock. If we assume an integer consists of 32 bits, then we can use the first 16 bits for the head, and the last 16 bits for the tail.

   ii) The process reads the value of the tail, and then increments the tail. This should of course happen in a secure way, i.e. no two processes have the same ticket.

   iii) When its ticket equals the head.

   iv) The process increments the head by one.

d)
```
// the shared integer containing head|tail
AtomicInteger queue = new AtomicInteger(0);

// the ticket of this process
ThreadLocal<Integer> local = new ThreadLocal<Integer>();
```

```
// acquire the lock
void lock() {
  // 1. add this process to the queue
  local = add();
  // 2. wait until the lock is acquired
  while (head() != local);
}

// add this process to the queue
int add() {
  while (true) {
    int value = queue.read();
    if (queue.compareAndSet(value, value + 1)) {
      return value & 0xFFFF;
    }
  }
}

// returns the current head of the queue
int head() {
  int value = queue.read();
  return (value >> 16) & 0xFFFF;
}

// releases the lock
void unlock() {
  while (true) {
    int value = queue.read();
    int head = (value >> 16) & 0xFFFF;
    int tail = value & 0xFFFF;
    int next = (head + 1) << 16 | tail;
    if (queue.compareAndSet(value, next)) {
      return;
    }
  }
}
```

Advanced

3 ALock2

a) The author wants that two processes can acquire the lock simultaneously.

b) The lock is seriously flawed. An example shows how the lock will fail: Assume there are n processes, all processes try to acquire the lock. The first two processes (p_1, p_2) get the lock, the others have to wait. Process p_1 keeps the lock a very long time, while p_2 releases the lock almost immediately. Afterwards every second process (p_4, p_6, ...) acquires and releases the lock. One half of all process are waiting on the lock (p_3, p_5, ...), the others continue to work (p_4, p_6, ...). If the working process now starts to acquire the lock again, then they wait in slots that are already in use.

It is also not FIFO (first in, first out) anymore. If p_1 keeps the lock after p_2 has released its lock, p_4 can acquire the lock before p_3.

c) A solution would be to increase the size of the array to at least $2 * n$ and further block the lock() method if a process holds the other lock for a (too) long time. By using the `FINISHED` state we guarantee that all lock holding processes are within an interval of slot numbers of size n. This is how we can make sure, that our slot numbers do not run into the still running processes from behind.

Unfortunately FIFO (first in, first out) is still not guaranteed. In a second step one could make the `unlock` method more intelligent: instead of jumping two slots, the method

searches for the oldest slot waiting for a lock. To simplify this search, the boolean array is replaced by an enum (or integer) array holding four states: unused, lockable, working, and finished. We can use a CAS operation to protect the `unlock` method against race conditions (two process may invoke the method concurrently).

```
class ALock2 implements Lock {
  ThreadLocal<Integer> mySlotIndex = new ThreadLocal<Integer> () {
    protected Integer initialValue() {
      return 0;
    }
  };
  AtomicInteger tail;
  enum State {UNUSED, LOCKABLE, WORKING, FINISHED};
  State[] states;
  AtomicBoolean setLockable = new AtomicBoolean(false);
  int size;

  ALock2(int capacity) {
    size = capacity;
    tail = new AtomicInteger(0);
    // >= 2n elements: 2 lockable, >= (n-2) unused, n finished
    states = new State[2 * size];
    states[0] = LOCKABLE;
    states[1] = LOCKABLE;
    Arrays.fill(states, 2, size, UNUSED);
    Arrays.fill(states, size, 2 * size, FINISHED);
  }

  public void lock() {
    // spin until slot becomes lockable
    int slot = tail.getAndIncrement() % (2 * size);
    mySlotIndex.set(slot);
    while (states[slot] != LOCKABLE) {}

    // mark as working
    states[slot] = WORKING;

    // wait for other lock if its process is too slow (larger array helps here)
    // & mark the slot as unused to support wrap-arounds
    while (states[(slot + size) % (2 * size)] != FINISHED) {}
    states[(slot + size) % (2 * size)] = UNUSED;
  }

  public void unlock() {
    int slot = mySlotIndex.get();
    states[slot] = FINISHED; // mark my slot as finished...

    // set next unused slot to lockable
    int index = slot + 1;
    while (setLockable.getAndSet(true)) {}
    while (states[index % (2 * size)] != UNUSED) {
      index++;
    }
    states[index % (2 * size)] = LOCKABLE;
    setLockable.set(false);
  }
}
```

4 MCS Queue Lock

a) There is more than one solution, but we can solve this problem without using RMW registers or other locks. It is important to set and read the flags in the right order: The `unlock` method first sets `locked`, then reads `aborted`. The `abort` method on the other hand first sets `aborted`, then reads `locked`. This way if `unlock` and `abort` run in parallel, one of them must already have written its flag before the other can read it. In the worst case `unlock` is called twice for some process, but that is not a problem. Unlocking an already unlocked lock results in no action.

```
public void unlock(){
  if( ... missing successor ... )
    ... wait for missing successor

  qnode.next.locked = false;
  if( qnode.next.aborted ){
    if( ... qnode.next misses successor ... ){
      if( ... really no successor ... )
        return;
    }
    else{
      ... wait for missing successor ...
    }
    qnode.next.next.locked = false;
  }
}

public void abort(){
  qnode.aborted = true;
  if( !qnode.locked ){
    unlock();
  }
}
```

b) The solution of a) does not yet work for concurrent aborts. Making the `unlock` method recursive will help.

```
public void unlock(){
  unlock( qnode );
}
private void unlock( QNode qnode ){
  // as before...
  if( ... missing successor ... )
    ... wait for missing successor

  qnode.next.locked = false;
  if( qnode.next.aborted ){

    // wait for successor of qnode.next
    if( ... ){ ... } else{ ... }

    unlock( qnode.next );
  }
}
```

c) There are four combinations of values the `locked` and `aborted` flag can have. We can easily encode these combinations in an integer. We would not need too worry about the order in which we read and write to the flags, as we could do this atomically. So the algorithm would get easier. We could also ensure that `unlock` is called only once. Depending on the benchmark this could increase the performance. On the other hand, a `CAS` operation is quite expensive and could decrease performance.

d) - There could be problems with caches: spinning on a value that "belongs" to another process can introduce additional load on the bus, and thus slow down the entire application.

+ The implementation is much easier: when releasing the lock one has only to set its own `locked` flag to `false`.

+ Also aborting is easier: a blocked process could read the state of its predecessor. If the predecessor is aborted, then the successor can just remove the node from the queue, and continue reading values from its predecessor's predecessor.