ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

FS 2019 Prof. R. Wattenhofer

Computer Engineering II

Solution to Exercise Sheet Chapter 10

Quiz

1 Quiz

a) The AtomicBoolean uses the atomic version of getAndSet() implemented in hardware. If a processor runs getAndSet() it invalidates the cache on the other processors. Other processes that want to execute getAndSet() must wait as the bus can only be used by one process at at time. Running commands atomically in hardware is expensive since caches need to be invalidated. Hence it is crucial to the performance of a locking algorithm to reduce the usage of these atomic operations in hardware.

b) Mutual exclusion is possible using only read-write registers, assuming a sequentially consistent 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 getAndSet() operations 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. Thus, all the processes execute getAndSet() at about the same time to acquire the lock and thrash the bus.

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) One possible solution is to use an atomic integer to hand out ticket numbers to processes requesting the lock and serve (unlock) the processes in the order of the ticket numbers so that at most two of them are unlocked at any time.

The algorithm for this solution is shown below. There is an array `ticket` and `flag` of size `capacity` having a slot for each process. In `ticket`, each process stores its ticket number in the corresponding slot. The process spins on its slot in the `flag` array to acquire

the lock. A process can rely on another process to set its flag if there is at least one that already holds a lock (`remainingLocks < 2`). Otherwise, it gets the lock for itself by setting its flag and decreasing `remainingLocks`. The unlock method unsets its flag, find the next process (`nextThreadID`) waiting for the lock and sets that process's flag. If there is no such process (`nextThreadID` = -1), it increments `remainingLocks` by one. We use `aLock` to protect these methods. FIFO property follows as `aLock` is a FIFO lock and processes are served in the order of their ticket numbers.

```
class ALock2 implements Lock {
    protected Integer initialValue() {
      return 0;
    }
  };
  AtomicInteger tail;
  int remainingLocks;
  int[] ticket;
  boolean[] flag
  ALock aLock;
  int size;

  ALock2(int capacity) {
    size = capacity;
    // number of locks that can be still acquired
    remainingLocks = 2;
    aLock = new ALock(size);
    tail = new AtomicInteger(0);
    ticket = new int[size];
    flag = new boolean[size];
    Arrays.fill(ticket, 0, size, -1);
    Arrays.fill(flag, 0, size, false);
  }

  public void lock() {
    // spin until slot becomes lockable
    int slot = tail.getAndIncrement() % size;
    i = ThreadID.get();
    aLock.lock();
    ticket[i] = slot;
    // if < 2 processes are holding the lock
    if(flag[i] == false && remainingLocks > 0) {
      remainingLocks--;
      flag[i] = true;
    }
    aLock.unlock();
    while (! flag[i]) {};
  }

  public void unlock() {
    i = ThreadID.get();
    aLock.lock();
    ticket[i] = -1;
    flag[i] = false;
    int min = tail.get(), nextThreadID = -1;
    for(int id = 0; id < size; id++) {
      if( ticket[id] != -1 && ticket[id] < min && flag[id] == false) {
        min = ticket[id];
        nextThreadID = id;
      }
    }
    if (nextThreadID != -1) {
      flag[id] = true;
    } else {
      // nonone waiting for the lock
      remainingLocks++;
    }
    aLock.unlock();
  }
}
```

4 Read-Write Queue Lock

a) and b) This solution builds on CLH lock; similar can be written to make it MCS-style.

We modify the QNode class to hold an AtomicInteger indicating the number of read requests. -1 will mean a write request. Some read-requesting processes will not add a node to the queue, so to check if they can have the lock, we add the predecessor field.

Recycling is not as easy as before, so we add a ThreadLocal pointer 'recycle' to remember the node that the process can take.

```
public class ReadWriteQueueLock implements Lock{
  ...
  ThreadLocal<QNode> recycle;
  ...
  public ReadWriteQueueLock(){
    recycle = new ThreadLocal<QNode>(){
      protected QNode initialValue(){
        return null;
      }
    };
    ...
  }
  ...
  class QNode{
    boolean locked = false;
    AtomicInteger reads = new AtomicInteger(0);
    QNode pred = null;
  }
}
```

The write-lock request can be appended to the queue almost exactly as before.

```
public void write_lock{
  QNode qn = myNode.get();
  qn.locked = true;
  qn.reads.set(-1);
  QNode pred = tail.getAndSet(qn);
  qn.pred = pred;
  while (pred.locked) {}
}

public void write_unlock(){
  QNode qn = myNode.get();
  qn.locked = false;
  myNode.set(qn.pred);
}
```

Read locking and unlocking is more complicated.

```
public void read_lock{
  QNode qn = myNode.get();
  qn.locked = true;
  qn.reads.set(1);

  while (true){
    QNode p = tail.get();
    int readers = p.reads.get();
    if (readers < 1){
      // Adding a read request to the queue
      qn.pred = p;
      if(tail.compareAndSet(p, qn)){
        // I added my node, so later I recycle the predecessor
        recycle.set(p);

        while (p.locked) {}
        return;
      }
    }
    else{
      // Just add 1 to the number of reading processes in the existing node
```

```
      if(p.reads.compareAndSet(readers, readers+1)){
        //I didn't use my node, so I keep it later
        recycle.set(qn);
        myNode.set(p);

        p = p.pred;
        while (p.locked) {}
        return;
      }
    }
  }
}

public void read_unlock(){
  QNode qn = myNode.get();
  int readers = qn.reads.decrementAndGet();
  if (readers == 0) qn.locked = false;
  myNode.set(recycle.get());
}
```

c) Once a write lock request is appended to the queue, following read and write requests will be granted only after that write is unlocked. Hence, the lock is FIFO with respect to write-locks.

Read-locks however might be "outrun" by other read and writes, because the compareAndSet operation in the read-lock might fail and be reattempted multiple times.

5 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 boolean lock(){
  ... insert qnode in queue ...
  qnode.aborted = false;
  qnode.next = null;
  if(pred ! = null) {
    qnode.locked = true;
    pred.next = qnode;
    while (qnode.locked && ! isAbort()) {};
    if (is_abort()) {
      qnode.aborted = true;
      // if predecessor unlocked meanwhile
      if(! qnode.locked) {
        unlock();
      }
      return false;
    }
  }
  return true;
}

public void unlock(){
  ... qnode = ...
  if( ... qnode misses successor ... ){
    if( ... really no successor ... )
    return;
  }
  else{
    ... 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;
  }
}
```

b) The solution of a) does not yet work for multiple aborted nodes in the queue. 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.