Exam 26.11.2015

**Question 1**

**1.a**

The implementation utilize one global lock to lock entire hash table

Interface hashtable{

Read(key)

Add(key)

Remove(key)

Replace(key)

}

hashtable {

HashTable table = new hashtable()

MutexLock globalMutex = new mutexLock()

function add(key) {

globalMutex.lock()

//*do add value to table…*

globalMutex.release()

}

function remove(key) {

globalMutex.lock()

//*do remove value to table…*

globalMutex.release()

}

}

**1.b**

The hashtable is only opened for reading if no write thread is accessing the table. We add the following code. The mutate funciton still acquire a lock to prevent other mutations, then it sets readPossible to false, prevented further reading. Before mutating it checks whether a read is in progress, and wait until reading is done. Solution can still result in starvation, if just as reaedPossible is set to false, a thread (call it 1) enters read mode and another already in read exits and set readinprogress to false. Then the mutation will happen while thread 1 continue reading. This can be prevented using a counting semaphore however this is not allowed.

hastable {

boolean readPossible = true

sophomore

function add(key) {

globalMutex.lock()

readPossible = false;

if (readInProgrss) spin until false

//*queue until all readers have left table*

//*do add value to table…*

readPossible = true

globalMutex.release()

}

function read() {

while (readPossible) {

readInProgess = true

//*execute reading procedure*

readInProgess = false

}

}

}

**c)**

writer.addToQue()

while(writer.position.next is equal to reader type)

writer.position.promoteOne

Solution always promotes a writer to the front of queue. Only next position is necessary to check as all other writer fill the top of the queue. This can result in starvation for read threads.

**d)**

allow a queue sequence of maximum e.g. 5 threads of the same type (read, write)