ECSE 420 – Assignment 2

**Group 11**

Yan Ren 260580535 Fei Feng 260575129

Q1.

1.1 Source code Filter.java is the implementation for filter lock, also see in appendix

1.2 Filter lock allows some threads to overtake others an arbitrary number of time. Suppose thread A has just successfully entered level 1 (it has completed the ﬁrst for iteration and is about to enter the next iteration with i = 2 ). If A is now paused for some reason (for example scheduling), every other thread which has successfully completed level 1 may overtake A.

1.3 Source code Bakery.java is the implementation for filter lock, also see in appendix

1.4 Bakery lock doesn’t allow some threads overtake others an arbitrary number of time. It satisfies the first-come-first-served property by assigning each thread a “number” in the doorway interval. In the waiting interval, the thread waits until no thread with an earlier number is trying to enter the critical section.

1.5 test design: create multiple threads to increment a shared variable. In our test case, we create 80 threads, each thread will call a function that increment variable ‘balance’ by one. We use locks for locking critical session. When program finishes, we expected final ‘balance’ is equal to 80.

1.6 source code Test.java is the implementation of 1.5, also see in appendix.

Q2.

A **regular** register either read the old or the new value when a read overlaps with a write.

An **atomic** register guarantees that the reads and writes appears to happen at a single point in time, which cannot switch between old and new value back and forth. The figure1 is from textbook to show the Peterson Lock implementation.

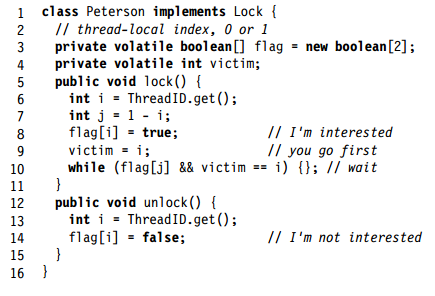


figure1

‘flag’ registers only read and write at line 8 and line 10. If read and write don’t overlap, the algorithm is correct. Overlap occurs when one thread is writing line 8 and another is reading line 10. Assume B is writing line 8, A is reading line 10. A may read the old value of B flag and enters the critical session or A may read the new value of B flag and waits. Then B will wait at line 10 since A already writes A flag. Therefore we still have mutual exclusion, the algorithm is correct.

Q3.

3.1 This protocol satisfies mutual exclusion. To proof this, assume it’s not mutual exclusion, which means threads A and B occur in the critical section at the same time. Suppose A went in first, it means turn must equal to A. Same for B, it must have seen turn equals to B to go into critical section. The only way can change turn=A to turn=B is that B reached the line 8. However, A set the busy=true when it entered the critical section, so that turn cannot equal to B at this time, therefore the assumption is false. Hence, the protocol satisfies mutual exclusion.

3.2 This protocol is NOT deadlock-free. If A runs until line 11, then busy=true and turn=A. Next if B runs until line 9, then busy=true and turn=B. Now A will wait outside since turn is not equal to A, meanwhile B will also wait since busy=true. Neither of them can enter the critical section. It caused a deadlock.

3.3 Since the protocol is NOT deadlock-free, it’s also NOT starvation-free.

Q4.

History(a): It is linearizable. The events can be executed in order of r.write(1), r.read(1), r.write(2), r.read(2). Since it is linearizable, it is also sequentially consistent.

History(b): It is linearizable. The events can be executed in order of r.write(2), r.write(1), r.read(1), r.read(1). Since it is linearizable, it is also sequentially consistent.

Q5.

The reader method may divide by zero. X is a non-volatile variable, the update of x may happen after the update of v, it means that v already equals to true but x is still equal to 0. Hence, reader method may divide by zero.

Q6.

6.1 True. If read doesn’t overlap any write call, the read will return the most recently written value. If overlapping occurs, read may return any legal value since the component registers are safe.

6.2 True. If read doesn’t overlap any write call, the read will return the most recently written value. If overlapping occurs, read may return the new value or the old value since the component registers are regular.

Q7.

Show that if binary consensus using atomic registers is impossible for two threads, then

it is also impossible for n threads, where n > 2.

Assume binary consensus is possible for n threads. Any n -thread consensus protocol must still work if only two threads take steps, so the two thread can simply run the n -thread protocol. Therefore we have binary consensus for two threads which is contradicts with binary consensus is impossible for two threads. So we proof if binary consensus is impossible for two threads, then it is impossible for n threads.

Q8.

Show that if binary consensus using atomic registers is impossible for n threads, then

so is consensus over k values, where k > 2.

Assume consensus over k > 2 values is possible for n threads. We can have a mapping where [0, k/2] maps to 0, [k/2, k] maps to 1, therefore we have a binary consensus for n threads. This is contradicts with the binary consensus is impossible for n threads. Therefore if binary consensus is impossible, k consensus, where k>2, is impossible.

Appendix

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

class Bakery implements Lock {

private int n;

private volatile boolean[] flag;

private volatile int[] label;

public Bakery(int n) {

this.n = n;

flag = new boolean[n];

label = new int[n];

for (int j = 0; j < n; j++) {

flag[j] = false;

label[j] = 0;

}

}

public void lock() {

int i = ThreadID.get();

flag[i] = true;

for (int j = 0; j < n; j++) {

if (label[j] > label[i]) {

label[i] = label[j];

}

}

label[i]++;

for (int j = 0; j < n; j++) {

while (flag[j] && (j != i) && (label[j] < label[i]) || ((label[j] == label[i]) && j < i)) {};

}

}

public void unlock() { // exit protocol

flag[ThreadID.get()] = false;

}

@Override

public void lockInterruptibly() throws InterruptedException {

// TODO Auto-generated method stub

}

@Override

public Condition newCondition() {

// TODO Auto-generated method stub

return null;

}

@Override

public boolean tryLock() {

// TODO Auto-generated method stub

return false;

}

@Override

public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {

// TODO Auto-generated method stub

return false;

}

}

import java.util.concurrent.TimeUnit;

import java.util.concurrent.atomic.AtomicInteger;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

class Filter implements Lock {

private AtomicInteger[] level;

private AtomicInteger[] victim;

int n;

public Filter(int n) {

this.n = n;

level = new AtomicInteger[n];

victim = new AtomicInteger[n]; // use 1..n-1

for (int i = 0; i < n; i++) {

level[i] = new AtomicInteger();

victim[i] = new AtomicInteger();

}

}

public void lock() {

int me = ThreadID.get();

for (int i = 1; i < n; i++) { // attempt level 1

level[me].set(i);

victim[i].set(me);

// spin while conflicts exist

boolean conflicts\_exist = true;

while (conflicts\_exist) {

conflicts\_exist = false;

for (int k = 0; k < n; k++) {

if (k != me && level[k].get() >= i && victim[i].get() == me) {

conflicts\_exist = true;

break;

}

}

}

}

}

public void unlock() {

int me = ThreadID.get();

level[me].set(0);

}

@Override

public void lockInterruptibly() throws InterruptedException {

// TODO Auto-generated method stub

}

@Override

public Condition newCondition() {

// TODO Auto-generated method stub

return null;

}

@Override

public boolean tryLock() {

// TODO Auto-generated method stub

return false;

}

@Override

public boolean tryLock(long arg0, TimeUnit arg1) throws InterruptedException {

// TODO Auto-generated method stub

return false;

}

}

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.locks.Lock;

public class Test {

private static int threads = 80;

private static Account account = new Account(threads);

public static void main(String[] args) {

ExecutorService executor = Executors.newCachedThreadPool();

// Create and launch 100 threads

for (int i = 0; i < threads; i++) {

executor.execute(new AddAPennyTask());

}

executor.shutdown();

// Wait until all tasks are finished

while (!executor.isTerminated()) {

}

System.out.println("What is balance ? " + account.getBalance());

}

// A thread for adding a penny to the account

public static class AddAPennyTask implements Runnable {

public void run() {

account.deposit(1);

}

}

// An inner class for account

public static class Account {

private static Lock lock; // Create a lock

private int balance = 0;

public Account(int threads) {

// uncomment one for testing different locks

lock = new Bakery(threads);

// lock = new Filter(threads);

}

public int getBalance() {

return balance;

}

public void deposit(int amount) {

lock.lock(); // Acquire the lock

try {

int newBalance = balance + amount;

// This delay is deliberately added to magnify the

// data-corruption problem and make it easy to see.

Thread.sleep(5);

balance = newBalance;

} catch (InterruptedException ex) {

} finally {

lock.unlock(); // Release the lock

}

}

}

}

public class ThreadID {

private static volatile int currentID = 0;

private static ThreadLocalID threadLocalID = new ThreadLocalID();

public static int get() {

return threadLocalID.get();

}

public static void reset() {

currentID = 0;

}

private static class ThreadLocalID extends ThreadLocal<Integer>{

protected synchronized Integer initialValue() {

return currentID ++;

}

}

}