**ECSE 420 Assignment 2 Report Group 19**

Stuart Mashaal Oliver Tse Sakkwun

260639962 260604362

Due: November 7th, 2018

Assignment 2

ECSE 420: Parallel Computing

**Question 1**

Question​ ​1.2

The​ ​Filter​ ​lock​ ​allows​ ​threads​ ​to​ ​overtake​ ​other​ ​threads​ ​an​ ​arbitrary​ ​number​ ​of​ ​times. There​ ​could​ ​be​ ​​**n**​ ​**-**​ ​**l**​ ​​threads​ ​spinning​ ​at​ ​level​ ​​**l**​.​ ​The​ ​order​ ​in​ ​which​ ​the​ ​for-loop​ ​is​ ​executed​ ​is​ ​up to​ ​the​ ​JVM​ ​and​ ​is​ ​nondeterministic.​ ​Therefore,​ ​the​ ​for-loop​ ​could​ ​go​ ​from​ ​iteration​ **k**​​ ​​​to​ ​iteration **k**​ ​**+**​ ​**1**​​ ​at​ ​any​ ​time​ ​for​ ​any​ ​given​ ​thread.​ ​Due​ ​to​ ​unfortunate​ ​timing,​ ​a​ ​thread​ ​could​ ​be​ ​overtaken many​ ​times.

Question​ ​1.4

The​ ​Bakery​ ​lock​ ​does​ ​not​ ​allow​ ​thread​ ​to​ ​overtake​ ​other​ ​threads.​ ​If​ ​thread​ ​A​ ​enters​ ​the​ ​doorway​ ​before​ ​thread​ ​B,​ ​then​ ​its​ ​label​ ​will​ ​be​ ​inevitably smaller.​ ​Therefore,​ ​it​ ​will​ ​be​ ​served​ ​first,​ ​and​ ​no​ ​other​ ​thread​ ​will​ ​be​ ​allowed​ ​to​ ​overtake​ ​it.​ ​The only​ ​scenario​ ​in​ ​which​ ​thread​ ​B​ ​overtakes​ ​thread​ ​A​ ​is​ ​if​ ​a​ ​label​ ​was​ ​assigned​ ​to​ ​thread​ ​B​ ​before it​ ​was​ ​assigned​ ​to​ ​thread​ ​A.​ ​This​ ​can​ ​only​ ​happen​ ​once.​ ​After​ ​thread​ ​B​ ​releases​ ​the​ ​lock,​ ​and attempts​ ​to​ ​lock​ ​again,​ ​its​ ​new​ ​label​ ​will​ ​be​ ​greater​ ​than​ ​thread​ ​A’s​ ​label.

Question​ ​1.5

Mutual​ ​exclusion​ ​can​ ​be​ ​tested​ ​by​ ​executing​ ​steps​ ​that​ ​would​ ​cause​ ​a​ ​race​ ​condition.​ ​To simulate​ ​a​ ​sequence​ ​of​ ​events​ ​that​ ​would​ ​cause​ ​a​ ​race​ ​condition,​ ​we​ ​could​ ​first​ ​read​ ​a​ ​shared variable,​ ​sleep​ ​for​ ​a​ ​number​ ​of​ ​seconds,​ ​and​ ​then​ ​write​ ​to​ ​the​ ​variable.​ ​During​ ​the​ ​time​ ​that​ ​the first​ ​thread​ ​is​ ​sleeping,​ ​another​ ​thread​ ​could​ ​do​ ​the​ ​same.

In​ ​this​ ​sequence​ ​of​ ​events,​ ​the​ ​first​ ​thread​ ​will​ ​be​ ​sleeping​ ​between​ ​read​ ​and​ ​write​ ​at​ ​the moment​ ​when​ ​the​ ​second​ ​thread​ ​will​ ​read​ ​the​ ​variable.​ ​This​ ​is​ ​a​ ​race​ ​condition.​ ​If​ ​mutual exclusion​ ​is​ ​properly​ ​implemented,​ ​the​ ​second​ ​thread​ ​will​ ​wait​ ​before​ ​reading​ ​and​ ​the​ ​end​ ​result will​ ​be​ ​as​ ​expected.​ ​In​ ​the​ ​case​ ​that​ ​there​ ​is​ ​no​ ​mutual​ ​exclusion,​ ​the​ ​end​ ​result​ ​will​ ​be​ ​off.

Question​ ​1.6

The​ ​test​ ​has​ ​been​ ​implemented​ ​in​ ​class​ ​​**FilterTest**​.​ ​Class​ ​should​ ​be​ ​compiled​ ​and​ ​the main​ ​method​ ​should​ ​be​ ​run.​ ​Test​ ​output​ ​will​ ​be​ ​printed​ ​to​ ​standard​ ​output.​ ​See​ ​the​ ​following code​ ​for​ ​sample​ ​output.

**Question 2**

The​ ​Peterson​ ​two-thread​ ​mutual​ ​exclusion​ ​algorithm​ ​works​ ​if​ ​we​ ​replace​ ​shared​ ​atomic registers​ ​with​ ​regular​ ​registers​ ​because​ ​regular​ ​registers​ ​behave​ ​differently​ ​when​ ​two​ ​reads​ ​are done​ ​concurrently​ ​with​ ​a​ ​write.​ ​For​ ​example,​ ​​​when​ ​a​ ​write​ ​to​ ​flag​ ​by​ ​a​ ​thread​ ​overlaps​ ​the​ ​other thread​ ​that​ ​is​ ​reading​ ​in​ ​the​ ​while​ ​loop.

If​ ​thread​ ​A​ ​is​ ​writing​ ​flag ​in​ ​the​ ​lock()​ ​method,​ ​thread​ ​B​ ​reading​ ​this​ ​field​ ​might​ ​read​ ​the value​ ​of​ ​true,​ ​therefore​ ​it​ ​stays​ ​in​ ​the​ ​loops​ ​and​ ​it​ ​might​ ​later​ ​read​ ​a​ ​temporary​ ​value​ ​of​ ​false. This​ ​means​ ​B​ ​would​ ​acquire​ ​the​ ​lock,​ ​which​ ​is​ ​acceptable​ ​since​ ​A​ ​is​ ​writing​ ​and​ ​its​ ​next​ ​step would​ ​be​ ​to​ ​the​ ​victim,​ ​and​ ​then​ ​B​ ​would​ ​acquire​ ​the​ ​lock.​ ​A​ ​would​ ​then​ ​get​ ​caught​ ​in​ ​the​ ​while loop​ ​and​ ​only​ ​one​ ​thread​ ​would​ ​be​ ​able​ ​to​ ​enter.

If​ ​the​ ​write​ ​to​ ​flag​ ​in​ ​the​ ​unlock()​ ​method,​ ​we​ ​can​ ​conclude​ ​that​ ​a​ ​read​ ​can​ ​lead​ ​to​ ​a​ ​true value​ ​after​ ​a​ ​previous​ ​read​ ​had​ ​the​ ​value​ ​of​ ​false.

If​ ​we​ ​look​ ​at​ ​the​ ​victim​ ​value,​ ​and​ ​this​ ​value​ ​flickers​ ​on​ ​the​ ​first​ ​read​ ​of​ ​the​ ​new​ ​value, the​ ​condition​ ​in​ ​the​ ​while​ ​loop​ ​will​ ​be​ ​false​ ​and​ ​the​ ​second​ ​read​ ​could​ ​not​ ​occur.​

**Question 3**

Question​ ​3.1

The​ ​protocol​ ​does​ ​satisfy​ ​mutual​ ​exclusion.​ ​We​ ​will​ ​prove​ ​it​ ​by​ ​contradiction.​ ​Suppose that​ ​it​ ​was​ ​not,​ ​and​ ​that​ ​two​ ​threads​ ​are​ ​in​ ​their​ ​critical​ ​sections​ ​at​ ​the​ ​same​ ​time.​ ​This​ ​must mean​ ​that​ ​thread​ ​1​ ​read​ ​the​ ​​**turn**​​ ​variable​ ​(and​ ​checked​ ​that​ ​it​ ​was​ ​its​ ​turn)​ ​and​ ​entered​ ​its critical​ ​section​ ​before​ ​thread​ ​2​ ​wrote​ ​to​ ​the​ ​​**turn**​​ ​variable.

*read*1(*turn* == 1) → *CriticalSection*1 → *write*2(*turn* = 2)

This​ ​implies​ ​the​ ​following​ ​sequence:​ ​thread​ ​1​ ​wrote​ ​the​ **b**​​ **usy**​ ​​variable​ ​(to​ ​true),​ ​then thread​ ​1​ ​read​ ​the​ ​​**turn**​ ​​variable​ ​(its​ ​own​ ​thread​ ​ID),​ ​then,​ ​thread​ ​2​ ​wrote​ ​the​ ​​**turn**​ ​​variable​ ​(to​ ​its thread​ ​ID)​ ​and​ ​then​ ​thread​ ​2​ ​read​ ​the​ ​​**busy**​​ ​variable​ ​with​ ​has​ ​to​ ​be​ ​​**false**​​ ​for​ ​thread​ ​2​ ​to​ ​continue on.

*write*1(*busy* = *true*) → *read*1(*turn* == 1) → *write*2(*turn* = 2) → *read*2(*busy* == *false*)  
But,​ ​​**busy**​ ​​was​ ​never​ ​reset​ ​to​ ​​**false**​.​ ​Therefore,​ ​we​ ​have​ ​a​ ​contradiction.​ ​Therefore,​ ​the

Flaky​ ​algorithm​ ​satisfies​ ​mutual​ ​exclusion. Question​ ​3.2

The​ ​following​ ​sequence​ ​of​ ​execution​ ​will​ ​lead​ ​to​ ​deadlock.

*write*1(*turn* = 1) → *read*1(*busy* == *false*) → *write*2(*turn* = 2) → *write*1(*busy* = *true*)

After​ ​this​ ​sequence​ ​of​ ​execution,​ ​thread​ ​1​ ​will​ ​be​ ​stuck​ ​in​ ​the​ ​outer​ ​loop​ ​because​ ​the condition​ ​at​ ​line​ ​11​ ​will​ ​never​ ​be​ ​false.​ ​Thread​ ​2​ ​will​ ​be​ ​stuck​ ​in​ ​the​ ​inner​ ​loop​ ​because​ ​the​ **b**​​**usy** variable​ ​will​ ​never​ ​be​ ​reset.​ ​Therefore,​ ​the​ ​algorithm​ ​is​ ​not​ ​deadlock-free.

Question​ ​3.3  
As​ ​shown​ ​in​ ​Question​ ​3.2,​ ​the​ ​algorithm​ ​is​ ​not​ ​deadlock-free.​ ​Therefore,​ ​it​ ​isn’t starvation-free.

**Question 4**

**Question 5**

Yes,​ ​there​ ​is​ ​a​ ​possibility​ ​that​ ​the​ ​​**reader**​​ ​method​ ​divides​ ​by​ ​0.​ ​The​ ​boolean​ ​variable​ ​​**v** has​ ​been​ ​declared​ ​volatile.​ ​This​ ​means​ ​that​ ​after​ ​one​ ​thread​ ​writes​ ​to​ ​it,​ ​all​ ​other​ ​threads reading​ ​it​ ​are​ ​guaranteed​ ​to​ ​read​ ​the​ ​new​ ​value.​ ​The​ ​problem​ ​here​ ​is​ ​that​ ​variable​ **x**​​ ​​​has​ ​not been​ ​declared​ ​volatile.​ ​This​ ​means​ ​the​ ​even​ ​after​ ​one​ ​thread​ ​has​ ​written​ ​to​ ​it,​ ​other​ ​threads reading​ ​it​ ​immediately​ ​after​ ​are​ ​not​ ​guaranteed​ ​to​ ​read​ ​the​ ​new​ ​value.​ ​Therefore,​ ​division​ ​by zero​ ​would​ ​occur​ ​if​ ​the​ ​volatile​ ​variable​ ​​**v**​​ ​is​ ​written​ ​to​ ​before​ ​non-volatile​ ​variable​ ​​**x**​.

The​ ​reader​ ​thread​ ​would​ ​pass​ ​the​ ​condition​ ​at​ ​line​ ​nine,​ ​which​ ​checks​ ​that​ **v**​​ ​​​is​ ​set​ ​to true.​ ​But​ ​even​ ​though​ ​​**v**​​ ​is​ ​true,​ ​there​ ​is​ ​no​ ​guarantee​ ​that​ ​variable​ ​​**x**​​ ​has​ ​already​ ​been​ ​updated. Therefore,​ ​this​ ​is​ ​when​ ​a​ ​division​ ​by​ ​zero​ ​could​ ​occur.

**Question 6**

Question​ ​6.1

The​ ​answer​ ​is​ ​true.​ ​In​ ​the​ ​first​ ​case,​ ​if​ ​reads​ ​don’t​ ​overlap​ ​with​ ​writes,​ ​then​ ​the​ ​most recent​ ​value​ ​will​ ​be​ ​returned​ ​because​ ​all​ ​registers​ ​have​ ​been​ ​written​ ​to​ ​already.

In​ ​the​ ​case​ ​where​ ​read​ ​calls​ ​overlap​ ​with​ ​write​ ​calls,​ ​by​ ​definition​ ​of​ ​a​ ​safe​ ​MRSW register,​ ​the​ ​value​ ​read​ ​must​ ​be​ ​any​ ​value​ ​within​ ​the​ ​domain​ ​of​ ​the​ ​values.​ ​Because​ ​the​ ​register array​ ​is​ ​composed​ ​of​ ​safe​ ​SRSW​ ​registers,​ ​then​ ​this​ ​condition​ ​is​ ​satisfied.

Question​ ​6.2

The​ ​answer​ ​is​ ​true.​ ​As​ ​for​ ​the​ ​previous​ ​question,​ ​for​ ​non-overlapping​ ​calls,​ ​read​ ​calls return​ ​the​ ​most​ ​recently​ ​written​ ​value.

If​ ​read​ ​calls​ ​overlap​ ​with​ ​write​ ​calls,​ ​then,​ ​by​ ​definition​ ​of​ ​a​ ​regular​ ​MRSW​ ​register,​ ​the value​ ​returned​ ​must​ ​either​ ​be​ ​the​ ​old​ ​value​ ​or​ ​the​ ​new​ ​value.​ ​Because​ ​the​ ​array​ ​contains regular​ ​registers,​ ​that​ ​condition​ ​is​ ​satisfied.

**Question 7**

Suppose​ ​we​ ​had​ ​a​ ​protocol​ ​for​ ​binary​ ​consensus​ ​for​ ​n​ ​>​ ​2​ ​threads.​ ​We​ ​could​ ​reduce​ ​it​ ​to a​ ​protocol​ ​for​ ​2​ ​threads​ ​by​ ​simply​ ​have​ ​2​ ​threads​ ​take​ ​steps​ ​and​ ​the​ ​remaining​ ​n​ ​-​ ​2​ ​threads hold.​ ​We​ ​then​ ​have​ ​a​ ​protocol​ ​for​ ​2​ ​threads.​ ​Because​ ​that​ ​is​ ​impossible,​ ​we​ ​have​ ​a contradiction.​ ​Therefore,​ ​binary​ ​consensus​ ​for​ ​n​ ​>​ ​2​ ​threads​ ​using​ ​atomic​ ​registers​ ​is impossible.

**Question 8**

Suppose​ ​consensus​ ​over​ ​k​ ​values​ ​is​ ​possible​ ​following​ ​some​ ​protocol.​ ​We​ ​could​ ​then “reduce”​ ​the​ ​consensus​ ​protocol​ ​to​ ​a​ ​binary​ ​consensus​ ​protocol​ ​by​ ​mapping​ ​one​ ​value​ ​to​ ​0​ ​and all​ ​other​ ​values​ ​to​ ​1.​ ​This​ ​implies​ ​that​ ​we​ ​have​ ​binary​ ​consensus.​ ​But,​ ​since​ ​binary​ ​consensus​ ​is impossible,​ ​we​ ​have​ ​a​ ​contradiction,​ ​therefore​ ​consensus​ ​over​ ​k​ ​values​ ​is​ ​impossible.