EE 360P, TEST 1, Vijay K. Garg, Spring'18

NAME:

UT EID:

Honor Code: I have neither cheated nor helped anybody cheat in this test.
Signature:

Time Allowed: 75 minutes
Maximum Score: 75 points
Instructions: This is a CLOSED book exam. Attempt all questions.

Q.1 **(10 points)** Please give concise answers to the following questions.

(a, 2 pts) What is the difference between a process and a thread?

A process (or a heavy-weight process) has its own memory image. It does not share its data, stack or heap with any other process. Each thread has its own stack but may share code, data and the heap with other threads. Multiple threads may exist for the same process.

(b, 2 pts) Suppose that we want that a thread executes `bar()` inside a monitor method `foo()` only if `count` is greater than 0. Will the following code snippet work in Java? If not, show how you will fix it.

```
public synchronized void foo() throws InterruptedException {
  if (count <= 0)
      wait();
   // count must be greater than 0 before bar() is executed
   bar();
}
```

No, because before the thread starts to execute bar(), some other thread may have made the count less than or equal to zero. It can be fixed as follows.

```
public synchronized void foo() throws InterruptedException {
  while (count <= 0)
      wait();
   // count must be greater than 0 before bar() is executed
   bar();
}
```

(c, 2 pts) Define deadlock-freedom and starvation-freedom for a program that has resources such as critical sections. Can a program be starvation-free but not be free from deadlocks? Justify your answer.

Deadlock-freedom ensures that there is no deadlock. Deadlock happens when a subset of processes cannot access resources because they are waiting for each other in a cyclic manner. Starvation-freedom ensures that there is no starvation. Starvation happens when a process that wants a resource does not get the resource forever. If a program is starvarion-free then all processes that want resources will eventually get it and therefore is free from deadlock (otherwise, there exists a subset of processes that cannot get the resource).

(d, 2 pts) A computer program has a method that takes 25% of the time and cannot be parallelized. What is the maximum speedup possible with 4 cores? Your manager has asked you to buy a machine with enough cores to get a speedup of 8. Specifically, he has asked you to calculate the number of cores that will enable the speedup of 8. What will you tell him?

T_p = parallel execution time
T_s = sequential execution time

$T_p \geq (1/4)T_s + 3/4 * T_s/4$.

Therefore, speedup = $T_s/T_p \leq 1/(1/4 + 3/16) = 16/7$.
The maximum speedup with 4 cores is 16/7.

It is impossible to get a speedup of 8 because the maximum speedup of this program is 4.

(e, 2 pts) What is advantage of using `ThreadPool` instead of creating explicit `Thread` for each task?

ThreadPool can *reuse* a thread instead of creating a new thread. Creating a new thread for each task may add to overhead.

Q.2 **(10 points)** Assume that there is a monitor object `queue` that implements a queue of Integers with the following methods.

```
poll(): returns and deletes the first element in the queue. If the queue is empty it returns a null value
isEmpty() returns true iff the queue is empty
enq(Integer x): enqueues x and signals all the threads waiting on the queue
```

.

A programmer wants to block for an empty queue and return only a non-null value when the queue becomes nonempty. Will the following code work? (i.e. is it possible for the program to reach line 15). Justify your answer. If the program is faulty, show how you will fix it.

```
01 public Integer deq() {
02     Integer retVal = null;
03     synchronized (queue) {
04          try {
05               while (queue.isEmpty()) {
06                    queue.wait();
07               }
08           } catch (InterruptedException e) {
09               e.printStackTrace();
10           }
11    }
12    synchronized (queue) {
13        retVal = queue.poll();// returns first item from the linked list unconditinally
14        if (retVal == null) {
15            System.err.println("retVal is null");
16
17        }
18    }
19    return retVal;
20 }
```

Yes, it's possible to reach line 15 because, if there is one element in the queue, multiple threads can get past the *isEmpty*() check. Then when the threads reach line 13, only one thread will get a non-null value from *poll*().

Another possible scenario is as follows. A thread T1 finds queue to be empty and goes to wait. Some other thread (enqueuer) wakes it up. However, before T1 acquires the lock at line 12, some other thread comes in, acquires the lock and removes the item. It can be fixed by combining the two separate synchronized blocks into one.

```
01 public Integer deq() {
02     Integer retVal = null;
03     synchronized (queue) {
04          try {
05               while (queue.isEmpty()) {
06                    queue.wait();
07               }
08           } catch (InterruptedException e) {
09               e.printStackTrace();
10           }
11           retVal = queue.poll();// returns first item from the linked list unconditinally
12           if (retVal == null) {
13               System.err.println("retVal is null");
14
15           }
16    }
17    return retVal;
18 }
```

Q.3 **(5 points)**

Consider the following mutual exclusion protocol based on busy-waiting. Show that it does not satisfy mutual exclusion. The variables `turn`, `wantCS1`, and `wantCS` are shared variables.

```
    shared int turn = 1;
    shared boolean wantCS1 = false;
    shared boolean wantCS2 = false;

//      code for process 1                  //   code for process 2
a1:     turn = 2;                           b1:  turn = 1;
a2:     wantCS1 = true;                     b2:  wantCS2 = true;
a3:     while(wantCS2 && (turn==2));        b3:  while(wantCS1 && (turn==1));
a4:     criticalSection();                  b4:  criticalSection();
a5:     wantCS1 = false;                    b5:  wantCS2 = false;
```

The following execution would cause two processes to be in critical section: a1, b1, b2, b3, b4 (CS), a2, a3, a4 (CS)

Q.4 **(5 points)** Suppose that your system provides a class `Synch` with a single static method `public void swap(AtomicInteger x, AtomicInteger y)`. This method atomically swaps the contents of Integer objects `x` and `y`. Use the swap method in the `Synch` class to implement the SwapLock class. Your lock should ensure mutual exclusion and deadlock-freedom. You do not have to worry about starvation-freedom. Note that you cannot use any synchronization construct except `Synch.swap`. Your implementation should be based on busy-waiting.

```
public class SwapLock {
// declare your variables
 AtomicInteger lockedBit = new AtomicInteger(0);

   public void lock() {
     AtomicInteger myBit = new AtomicInteger(1);
     while (myBit.get() == 1)
       Synch.swap(myBit, lockedBit);
   }


   public void unlock() {
     AtomicInteger myBit = new AtomicInteger(0);
     Synch.swap(myBit, lockedBit);
   }
```

Q. 5 **(20 points)** Suppose that the state regulation for any child care center is that there must be at least one employee in the center for every four children. So, if there are five children in the center, then there must be at least two employees. Write a monitor class `ChildCare` that supports four methods. Parents call `enterChild()` whenever they want to drop a child at the center. They call `exitChild()` whenever they come to pick up the child. Parents can pick up their child whenever they want; however, they can drop the child only if the ratio of the children to the employees does not exceed four. If the ratio on dropping the child will exceed four, the method `enterChild()` should block (wait). Employees call `enterEmp()` whenever they enter the child care center. This method does not block. Employees call `exitEmp()` whenever they want to leave the center. This method should block if the state regulation gets violated if the employee leaves the center.

Do not worry about starvation but your program should be free from deadlocks. You can use built-in Java monitors or Reentrant locks and condition variables.

```
public class DayCare {
    // declare your variables
    int numChld = 0;
    int numEmp = 0;

    public synchronized void enterChild() throws InterruptedException {
       while (4*numEmp < (numChld+1))
         wait();
       numChld++;

    }
    public synchronized void exitChild() {
       numChld--;
       notifyAll();

    }
    public synchronized void enterEmp() {
       numEmp++;
       notifyAll();
    }
    public synchronized void exitEmp() throws InterruptedException {
       while (4*(numEmp-1) < numChld)
         wait();
       numEmp--;

    }
}
```

Q.6 **(25 points)** FactoryX uses one or more *machines* to manufacture *items*. In order to save on storage costs, an item is manufactured only when there is space in the warehouse. The warehouse has a capacity of w units where $w \geq 1$. Space is freed up in the warehouse whenever a *Consumer* buys the product. The raw materials used by machines are provided by two independent sources, MetalsInc and ChemicalsInc and are stored in the depot with capacity d. In order to manufacture 1 unit, FactoryX requires 2 units from MetalsInc and 3 units from ChemicalsInc. You can assume that $d \geq 5$.

There are some constraints which need to be met:

1. FactoryX should not manufacture products if the warehouse is full.
2. There should be at least 2 units from MetalsInc and 3 units from ChemicalsInc in the depot to manufacture a product.
3. One type of raw material should not fill up the depot such that an item cannot be manufactured. For example, if the depot has a storage capacity of 10 units, then at any point in time, it should have at most 7 units from MetalsInc or 8 units from Chemicals Inc.

You do not have to worry about exceptions for this question. The program must guarantee all the constraints and deadlock-freedom. Do not worry about starvation-freedom. You must use Java Reentrant locks and condition variables. Your program will be graded for correctness (18 pts) and efficiency (7 pts).

```
import java.util.concurrent.locks.Condition; import java.util.concurrent.locks.ReentrantLock;

// There are four types of threads that access this monitor object.
// MetalsInc threads call receiveMetal() which adds one unit of metal
// ChemicalsInc threads call receiveChemical() which adds one unit of chemical
// Machine threads call manufacture() which produces one unit of the finished item
// Consumer threads call getProduct() which consumes one finished item

public class FactoryX {

    private int w;
    private int d;

    private int metal, chem, items;

    private ReentrantLock lock;
    private Condition depotFull, lowRaw, lowProd, warehouseFull;


    public FactoryX(int warehouse_capacity, int depot_capacity){
        w = warehouse_capacity;
        d = depot_capacity;
        metal = chem = items = 0;
        lock = new ReentrantLock();
        depotFull = lock.newCondition();
        warehouseFull = lock.newCondition();
        lowRaw = lock.newCondition();
        lowProd = lock.newCondition();
    }

    public void receiveMetal() throws InterruptedException{
        lock.lock();

        while ((d-metal <= 3) || (metal+chem == d))
            depotFull.await();

        metal++;
```

```
    if (metal >=2 && chem >= 3)
        lowRaw.signal();

    lock.unlock();
}

public void receiveChemical() throws InterruptedException{
    lock.lock();
    while ((d-chem <= 2) || (metal+chem == d))
        depotFull.await();

    chem++;

    if (metal >=2 && chem >= 3)
        lowRaw.signal();

    lock.unlock();
}

public void getProduct() throws InterruptedException{
    lock.lock();
    while(items < 1)
        lowProd.await();
    items -= 1;
    warehouseFull.signal();
    lock.unlock();
}

public void manufacture() throws InterruptedException{
    lock.lock();

    while ((items == w) || (metal < 2) || (chem < 3 )) {
       if (items == w) warehouseFull.await();
       if (metal < 2 || chem < 3 ) lowRaw.await();
    }

    // createItem();

    metal -= 2;
    chem -= 3;
    items++;

    lowProd.signal();
    depotFull.signalAll();

    lock.unlock();
}
```