CMSC330 Spring 2011
Practice Problems 6 (SOLUTIONS)

1. Multi-threading. For each of the following programs in the language we considered in class, give the possible final values that could be stored in x.

   a.
   ```
            spawn p1
            spawn p2
            halt
   p1:      x = 1
            halt
   p2:      x = 0
            halt
   ```

   x = 1 or x = 0

   b.
   ```
            b = 1
            spawn p1
            spawn p2
            halt
   p1:      if b != 0 goto p12
            halt
   p12:     x = 1
            halt
   p2:      if b != 0 goto p22
            x = 2
            halt
   p22:     halt
   ```

   x = 1

   c.
   ```
            x = 0
            spawn p1
            spawn p2
            halt
   p1:      y = x
            x = 2*y
            halt
   p2:      z = x
            x = z + 1
            halt
   ```

   x = 0 or x =1 or x = 2

2. Locks

   a. What values may x hold when the following program terminates?

```
        x = 0
        spawn p1
        spawn p2
        halt
p1:     lock l
        y = x
        x = 2*y
        release l
        halt
p2:     lock l
        z = x
        x = z + 1
        release l
        halt
```

x = 1 or x = 2

b. Explain how locks may be used to implement a new statement for our language, `await x with l`. The meaning of this statement is as follows. The program blocks until x is non-zero and l is available, then begins execution of the statements immediately following with lock l locked and x initially guaranteed to be non-zero. Your explanation should contain a code snippet. You may use any new variables / labels you need.

```
await:  acquire l
        if x != 0 goto exit
        release l
        goto await
exit:   …
```

The idea is to loop: acquire l, test if x is non-zero, and if it is, exit the loop, else release l and go back to the top of the loop. If all processes lock l before accessing x then this protocol ensure that when the loop is exited, l is held and x is non-zero.

c. Explain why the following program may deadlock.

```
        spawn p1
        spawn p2
        halt
p1:     acquire l1
        acquire l2
        release l2
        release l1
        halt
p2:     acquire l2
        acquire l1
```

```
        release l1
        release l2
        halt
```

Depending on the scheduler, p1 may lock l1 and p2 lock l2. Then the threads are deadlocked: neither can acquire the lock held by the other, and so both are blocked.

d. A commonly used policy to avoid deadlocks when using locks is the following. Processes must not release any locks until they have acquired all locks they need, and they must attempt to lock individual locks in the same order. Explain how this would fix the situation in the immediately previous problem.

In the above situation, both threads already obey the first part of the policy. However, they acquire the locks l1 and l2 in different order. If they were to try the same order, one would acquire the first lock, and the other would have to wait. Then the first would be guaranteed access to the second lock.

3. Signaling

a. Using wait and notifyAll, implement a three-thread system consisting of one withdrawer and two depositors. Each depositor deposits $10 into an account x (originally x is 0); the withdrawer may withdraw $20 as soon as the account contains $20. You must ensure that account x is never overdrawn (i.e. that variable x is never negative).

```
        x = 0
        spawn d
        spawn d
        spawn w
        halt
d:      acquire xlock
        x = x+10
        z = 20-x
        if z != 0 goto ddone
        notifyAll xlock
ddone:  release xlock
        halt
w:      acquire xlock
        y = 20 - x
        if y != 0 goto wwait
        x = x-20
        release xlock
        halt
wwait:  wait xlock
        x = x-20
        release xlock
```

```
        halt
```

b. In class we implemented `wait` using a waiter set; if a thread i executes `wait l` then an insertion is made into the wait set. Explain how busy-waiting (use of while loops to emulate waiting) may instead be used to implement `wait`. How would `notifyAll` be implemented in this case?

The idea behind this implementation is to use two variables, lw and ls, and an auxiliary lock laux for each lock l. lw counts the number of waiters waiting on l, while ls is true if a notifyAll command has been issued on l. Here is the code for wait l.

```
entry:  acquire laux /* wait for notifies to finish */
        if ls != 0 goto loop
        lw = lw+1
        release laux
        goto await
loop:   release laux
        goto entry
await:  acquire laux
        if ls != 0 goto woken
        release laux
        goto await
woken: lw = lw-1
        if lw != 0 goto rel
        ls = 0 /* reset when all waiters notified */
rel:    release laux
        acquire l
```

For notify, the signal variable ls needs to be set, and then the notifier needs to block until the number of waiters becomes 0.

```
        /* assume l already held */
        acquire laux
        if lw != 0 goto wtrs
        goto done /* no waiters, nothing to do */
wtrs:   ls = 1
        release laux
waking: acquire laux
        if lw != 0 goto loop
        goto done
loop:   release laux
        goto waking
done:  release laux
```

c. The term *busy waiting* refers to a style of programming in which threads remain in loops, testing for the availability of a condition or shared resource. Explain why the use of locks / wait / notifyAll is preferable to busy-waiting.

Busy waiting consumes cycles, as waiting processes must execute repeated tests in order to see if they may resume.