CS4410 Homework 2

Due 9am Tuesday February 17, via the CMS system

100 points as shown below

Q1. Bryant Park Automated Dog Walkers Synchronization. Biscuit, Coco, Flower, Blue and Simcha, love to run into one another during walks. But Rocky and Butch, who also live in the neighborhood, are very unfriendly and will attack any dogs they encounter (including one-another). The Bryant Park community has hired you to prevent dog-fights with an automated system.

Each dog has an RFID tag on its collar. Bryant Park has installed RFID readers equipped with a little traffic light (it has red and green leds), one on the porch of each dog owner, and connected these to a central supercomputer, the HAL-1. You've proposed to implement a state of the art technology for dog-walk coordination: it makes a dog owner wait until it is safe to walk their dog by illuminating the red led during periods when a "conflicting dog" (or dogs) is already walking. The solution should be safe, live and deadlock-free. We would like to see all code in Java. Your solution should be sensitive to environmental considerations: Please don't leave the porch lamp illuminated if there is nobody on the porch: set it back to black (off).

The solution has three parts and you need to code each of them, in syntactically correct Java, complete with brief comments. We expect you to write actual Java code, compile it, write a test program that has a bunch of threads (one per household) and test it. You can work in the CSUG lab or just do this on your home computer. *Your work must be your own: you can discuss the general question with friends but not a line of code should be written jointly or shared with them.*

1) [10 points] Write a main "procedure body" that creates one thread per household and simulates the life of a dog, which is basically: nap, then take a walk. (We can skip other activities). To take a walk, the dog (and owner) follow the procedure described above. Java has a thread level sleep call and you can use it for the delays associated with the dog napping or actually doing the walking. For your tests, we're fine with a pretty rapid pace of naps and walking. Your code can just print a single line for each interesting thing that happens ("Biscuit wants to take a walk. The led turns red. Biscuit is waiting.) Your test threads will issue calls to the methods discussed in (2).

2) [20 points] Write code for the RFID reader and light controller associated with the porch object. When the RFID sensor detects that the owner has come out on the porch, it will call the method RFID_Detect(Dog name). A second method, RFID_Offline(Dog name) is called when the dog leaves the porch, either because it stepped down onto the street or because it reentered the house. The argument is the dog's name (treat this as an enumerated list). To set the color of the light call use the method LIGHT_Set(Colors color), where the color should be one of RED, GREEN, or BLACK (off).

Thus if I walk Biscuit, I step onto the porch and RFID_Detect(Biscuit) is invoked in the porch object associated with my house. Perhaps Butch is being walked, so the red led lights up and stays red for a while, and Biscuit and I cool our heels (well, paws). Eventually the street is safe and then the light turns green. Now Biscuit and I head down the street, and RFID_Offline(Biscuit) will be invoked. Later when we return, RFID_Detect will trigger again as we come back onto the porch, and then when we reenter the house, RFID_Offline(Biscuit) will be triggered. Had Butch not been walking, the

led might have blinked red for a moment, but the would have turned green. (Obviously, ideally, it would be best not to have the led flash red at all in this case – if you can figure out how to do that).

3) [20 points] Write code for a class BryantPark that provides the coordination needed by the RFID reader. All porches will share a single instance of this class, with the global name ThePark, accessing it through methods it exposes that you will need to define. These methods should tell the code you wrote for part 1 which colors to use for the porch lights.

Hint: Don't worry about practical questions, such as whether Biscuit and I will be willing to wait on the porch for twenty minutes, or will jump in the car and drive to Hammond Hill – in real life those things happen, but for your homework, they don't. And don't freak out if your solution always sets the light to red as soon as I step on the porch, even if it will turn green instantly. Even if it would be nice not to do that, it won't upset us very much if your code has this feature.

Q2. Bakery Algorithm (4) 10 points. In class we discussed a modified version of the Bakery Algorithm in which we added some simple code to ensure that ticket values could never exceed a prespecified limit, MAXIMUM. Ken explained that this would eliminate any worry about tickets overflowing the counter size. But suppose that we implement the Bakery Algorithm in the standard way, without that extra loop (the standard code was given in class on the prior slide, and is also the version you'll find in the textbook). Briefly describe *all* the requirements for a collection of N threads using the normal Bakery Algorithm to have a ticket value overflow. You don't need to give the actual code, but we need a very precise (short) description that touches on all the required behavior.

Q3. Producer Consumer Synchronization. Consider a producer-consumer application in which a producer and a consumer share a bounded-buffer of size B objects. They run at the same average rate of R objects consumed, or produced, per second, but the consumer is very steady whereas the producer is bursty: it generates BURST objects at a time. For example, it might produce an average of 5 objects per second, but in bursts of 10 (i.e. the producer could generate 10 objects during a period of 0.05 seconds, but then none at all for 1.95 seconds, then 10 more, etc). When we say that R is the average rate, assume that we computed R over a time period of 10 seconds.

5) 10 points. How does this system behave if B=R? E.g. in our example above, B would be set to 5.

6) 10 points. How does this system behave if B is set to a size much larger than max(R, BURST)? E.g. in our example above, we have R=5/sec and BURST=10, but perhaps we could set B to 100.

7) 10 points. Under the assumptions we've stated, do we know enough to select a value of B that would ensure that the buffer never becomes empty and never becomes full? If so, explain how to compute this value for B. If not, explain why we don't have enough information.

Q4. Recursion and critical sections. (8) 10 points. In class we saw a number of ways to implement CSEnter and CSExit methods, for example by using mutual-exclusion locks (0/1 semaphores). We assumed that the code inside the critical section (between the CSEnter and the corresponding CSExit) would "drop straight through" without doing anything else. But suppose that we are interested in using CSEnter and Exit within a recursive procedure:

```
Void Recursive(int i)
{
```

```
        CSEnter(my_threadid);
        if(i > 0)
                Recursive(i-1);
        CSExit(my_threadid);
}
```

Using Java notation and Mutex variables, give code for a version of CSEnter(ThreadID myid) and CSExit(ThreadID myid) such that can be called as shown above, and won't deadlock.