CS4410 Homework 2

Due 9am Tuesday February 17, via the CMS system

100 points as shown below

Q1. Bryant Park Automated Dog Walkers Synchronization.

1-3) [50 points] Posted separately

Q2. Bakery Algorithm (4) 10 points.

To overflow, we would need some set of threads that play a kind of relaying game, such that there is always some thread with a picked ticket. For example, suppose that threads T1 and T2 take turns entering the critical section, with T1 remaining in the critical section until T2 picks a ticket and vice versa. Then T1 will pick 1, T2 2, T1 3, etc. But if even a single moment occurs during which no thread has a ticket, the values reset to zero. As long as at least 2 threads play this relaying game, it doesn't matter if N is larger than 2. With N=1, overflow cannot occur.

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?

First, as a general comment, when we talk about a rate R over a period such as 1 second, we typically mean that if you computed the rate over some larger period, like 2 seconds, you would still get the same average rate R. In some sense, by calculating R during a 10 second period (in our example) is long enough to know that the rate is pretty stable.

*Just the same, the units of R are in objects per second, not per 10 seconds. Thus when we say B=R we implicitly convert from a rate (objects/sec) to a number (objects/sec * 1-second).*

So our question becomes: suppose that a producer producers (say) 8 items every second. And the consumer consumes 8 items every second, too. Will a buffer of size 8 be large enough to ensure that the producer won't fill it? In fact the answer is: probably not.

To see this, suppose that at time 0 we start the producer and consumer simultaneously. The buffer is empty so the consumer blocks. The producer begins thinking. Now, sooner or later it will produce a BURST of items – maybe 4 of them. Now the consumer finally starts executing.

This situation is just as if we hadn't started the consumer until some time after we started the producer. And if you view the situation that way, you can see that we'll gravitate towards a mode in which the consumer is sufficiently lagged behind the producer to never quite catch up.

Now, over a 1 second period we produce R items in total (yes, in BURSTS, but a total of R items). Thus we could actually have an extreme case in which the consumer basically doesn't even start running until there are all R items in the buffer. Moreover, since the producer is basically as much as 1 second ahead of the consumer, the producer could be on its way to creating the next R items as well.

*All of this adds up to: probably if the buffer size B is smaller than 2*R, the producer will probably block because of a full buffer at least now and then, and perhaps very frequently.*

*Some people have asked whether BURST could be a very big number, like 10*R, reasoning that if so, during a 10 second period we would still calculate the same rate, but it would all happen at once. Obviously if you think this is reasonable, you'll answer that if B is less than 10*R the producer is likely to block. But in fact we don't think such a large BURST fits the problem statement: if BURST could be as large as 10*R, the average rate would vary from 0/sec to 10/sec and back down to 5/sec as we look at intervals ranging from 0 to 20 seconds. Nobody would call such an unstable situation a "rate". In practice, a BURST would have to be at most R items at a time. And this leads back to B=2*R being large enough.*

6) 10 points. How does this system behave if B is set to a size much larger than max(R, BURST)?

*As the discussion from (5) makes clear, if BURST<=B, then once B becomes large enough – probably around 2*R, the producer is unlikely to block at all. So the producer may block now and then. Now, if BURST>R, the same logic holds, but for B around 2*BURST. So you actually need about 2*max(R,BURST) space in the buffer to be reasonably sure the producer won't run low on space. And in fact we can see from (5) that as a practical matter, max(R,BURST) will always be R.*

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?

Answered as part of the answer to (6).

Q4. Recursion and critical sections. (8) 10 points.

Some people will probably have figured out that Java locking and synchronization support recursion, so that answer is acceptable as long as you said so. PThreads (used in cs4410) would need something along the following lines:

```
int currently_inside = 0;
int recursion_cntr = 0;
semaphore mutex = new semaphore(1);

Void CSEnter(int i)
{
    if(currently_inside == i)  // either zero or the thread-id of the current CS holder
    {
        ++recursion_ctr;
        return;
    }
```

```
        mutex.acquire();
        currently_inside = i;
        recursion_ctr = 1;
}
Void CSExit(int i)
{
        if(currently_inside == i)
        {
                if(--recursion_ctr > 0)
                    return;
        }
        currently_inside = 0;
        mutex.release();
}
```