CSC 256/CSC 456
Homework 1
September 26, 2013
Due in class on Thursday, October 3rd, 2013

Please be sure to show all your work and write down any assumptions you make.

1. Consider a reader-writer lock as we discussed in class — a writer requires exclusive access to the lock, while readers may concurrently hold the lock with other readers. The version we implemented/sketched out on the board used $n + 1$ semaphores.

   (a) Can you discuss the properties of the version discussed in class in terms of number of operation per read and write, and fairness guarantees? For the purposes of this discussion, the important operations are semaphore signals and waits. Please start by writing out the algorithm.

   (b) Outline an implementation of the lock that does not use as many semaphores as there are participating processes. Discuss its properties in terms of number of operations per read and write, and fairness guarantees. *Hint: Introducing additional variables in addition to semaphores is okay, as long as the space requirements are kept constant.*

2. Lamport's bakery algorithm provides mutual exclusion for multiple processes using ordinary reads and writes. Lamport envisioned a bakery where each customer is given a unique number and a global counter displays the number of the customer currently being served. However, slight modifications are required in order to work with just reads and writes. The algorithm (for N processes) is below:

```
// declaration and initial values of global variables
Entering: array [1..N] of bool = {false};
Number: array [1..N] of integer = {0};

lock(integer i)
{
    Entering[i] = true; /* indicate intent to choose number */
    Number[i] = 1 + max(Number[1], ..., Number[N]);
    Entering[i] = false;
    for (j = 1; j <= N; j++) {
        // Wait until thread j receives its number:
        while (Entering[j]) { /* nothing */ }
        // Wait until all threads with smaller numbers or with the same
        // number, but with higher priority (lower id), finish their work:
        while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) {
            /* nothing */
        }
    }
}
unlock(integer i) { Number[i] = 0; }

Thread(integer i) {
    while (true) {
        lock(i);
        // The critical section goes here...
        unlock(i);
        // non-critical section...
    }
}
```

Some of today's processors provide an atomic *fetch and increment (fai)* instruction that takes a single operand *addr*. *fai* loads the original contents of the memory location pointed to by *addr* into a register and increments the value in the memory location in a single indivisible operation. Logically:

```
int fai ( int *addr) {
    int temp = *addr;
    *addr = temp+1;
    return (temp);
}
```

Assuming that the processor you are executing on provides an atomic *fai* instruction, would you be able to simplify the bakery algorithm so it more closely resembles that at a bakery? If so, what is the new algorithm? You can ignore any consistency model issues. Discuss your new algorithm's properties with respect to mutual exclusion, progress, and bounded waiting.