CSC 256/CSC 456
Homework 1
October 2, 2018
Due in class on Tuesday, October 9th, 2018

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

1. (a) In a system with n processes and m resources of the same type, assume that the sum of all maximum needs is less than $m + n$ and that the maximum need of each process is between 1 and m resources. Resources are requested and released by processes only one at a time. Prove that this system cannot deadlock.

   (b) Consider a system with 10 units of a single resource. There are four processes in the system, named [A,B,C,D], each of which have a maximum resource requirement that can be specified by the vector [6,5,4,7] units, where each element indicates the maximum needs of the corresponding process. If the processes currently hold [1,2,2,4] resource units, is the system in a safe state (i.e., is deadlock possible given the maximum needs of each process)?

2. Consider a single-CPU system with only CPU-bound tasks and assume CPU scheduling is always non-preemptive. Also assume each task's runtime is known in advance, no two tasks have the same runtime, and that all tasks are ready at system start. Show that shortest-job-first (SJF) scheduling is the *sole* optimal scheduling order in terms of mean task turnaround time (where turnaround time is defined as the elapsed time from when the task enters the system to the moment the task finishes).

3. 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. How would you implement a fair version of this lock (i.e., lock requests are granted in the order received)? Please present your pseudo-code and explain your rationale.

4. 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; }
```

```
18  Thread(integer i) {
19      while (true) {
20          lock(i);
21          // The critical section goes here...
22          unlock(i);
23          // non-critical section...
24      }
25  }
```

Today's processors provide some form of 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.