CMU 15-418/618: Parallel Computer Architecture and Programming
Practice Exercise 4

Problem 1: Miscellaneous

A. (3 pts) You and your business partner buy a **single-core computer** to host your startup's web site. Your site receives exactly one type of request: it involves a disk read with latency 900 ms (during which the processor is idle, and waits) followed by 600 ms of processing. Thus, its response latency is 1.5 sec.

   As your business grows you start receiving requests at steady rate of one request every 100 ms. Your partner claims, "Let's buy 14 more machines so we can keep up!" You claim, "Wait a minute, we can save money and handle this load without all those machines!" How many machines do you actually need to service 10 requests per second, and how would you thread your web server to achieve this throughput? **Assume that disk I/O bandwidth is infinite, multiple I/O operations can be outstanding at once, and that all requests are unique and so optimizations like caching are not possible.**

B. (2 pts) At 8:30am, you go to grab a coffee at the Gates 3rd floor cafe. There is no line, so you immediately walk up and order. Later that day, you go to get your daily double-creme, super-duper, organic soy latte between classes at 12:50pm to take to 15-418/618, and the line is 15 students long. You tell the manager, "you really need to hire more workers,". She responds, "I can't, because it would be cost too much to double my staff for the entire workday". Using the words "throughput" and "elasticity" in your answer, make a suggestion on how the manager might run her business.

C. (2 pts) Consider a parallel version of the 2D grid solver problem from class. The implementation uses a 2D tiled assignment of grid cells to processors. (Recall the grid solver updates all the red cells of the grid based on the value of each cell's neighbors, then all the black cells). Since the grid solver requires communication between processors, you choose to buy a computer with a cross-bar interconnect. Your friend observes your purchase and suggests there there is another network topology that would have provided the same performance at a lower cost. What is it? (Why is the performance the same?)

D. (3 pts) Recall a basic test and test-and-set lock, written below using compare and swap (`atomicCAS`) Keep in mind that `atomicCAS` is atomic although it is written in C-code below.

```
int atomicCAS(int* addr, int compare, int val) {
  int old = *addr;
  *addr = (old == compare) ? val : old;
  return old;
}

void LOCK(int* lock) {
   while (1) {
      if (*lock == 0)
         if (atomicCAS(lock, 0, 1) == 0)
            return;
   }
}
```

Imagine a program that uses this lock to synchronize access to a variable shared by 32 threads. This program is run on a processor with **four cores**, each of which are **eight-way multi-threaded** (32 total execution contexts across the machine.) You can assume that **the lock is highly contended, the cost of lock acquisition on lock release is insignificant and that the size of the critical section is large (say, 100,000's of cycles)**. You can also assume there are no other programs running on the machine.

Your friend (correctly) points there is a performance issue with your implementation and it might be advisable to not use a spin lock in this case, and instead use a lock that de-schedules a thread off an execution context instead of busy waiting. You look at him, puzzled, mentioning that the test-and-test-and-set lock means all the waiting threads are just spinning on a local copy of the lock in their cache, so they generate no memory traffic. What is the problem he is referring to? **(A full credit answer will mention a fix that specifically mentions what waiting threads to deschedule.)**

Problem 2: Hash Table Parallelization (10 points)

Below you will find an implementation of a hash table (a linked list per bin). The hash table has a function called `tableInsert` that takes two strings, and inserts both strings into the table **only if neither string already exists in the table**. Please implement `tableInsert` below in a manner that enables maximum concurrency. You may add locks wherever you wish. (Update the structs as needed.) To keep things simple, your implementation SHOULD NOT attempt to achieve concurrency within an individual list (notice we didn't give you implementations for `findInList` and `insertInList`). **Careful, things are a little more complex than they seem. You should assume nothing about `hashFunction` other than it distributes strings uniformly across the 0 to `NUM_BINS` domain. (HINT: deadlock!)**

```
struct Node {
  string value;
  Node* next;

};

struct HashTable {
   Node* bins[NUM_BINS];  // each bin is a singly-linked list


};

int   hashFunction(string str);           // maps strings uniformly to [0-NUM_BINS]
bool  findInList(Node* n, string str);    // return true is str is in the list
void  insertInList(Node* n, string str);  // insert str into the list

bool tableInsert(HashTable* table, string s1, string s2) {
   int idx1 = hashFunction(s1);
   int idx2 = hashFunction(s2);




   if (!findInList(table->bins[idx1], s1) &&
       !findInList(table->bins[idx2], s2)) {


     insertToList(table->bins[idx1], s1);


     insertToList(table->bins[idx2], s2);


     return true;
   }



   return false;
}
```