Question 1

A. No, MapReduce would not be good as the backend for such a service because of two reasons. First of all, MapReduce is not designed for processing streams of data. It is designed to operate in batch over an existing large dataset. Secondly, MapReduce is not designed for low-latency computations.

B. No, MapReduce is not good at providing low-latency responses needed for interactivity, and in addition, MR is not capable of aborting early and providing partial results. Either an entire MR job completes, or it doesn't and has to be re-run.

C. No, again MapReduce is not good at low-latency computations, and it has no support for streaming data like stock ticker updates.

D. Yes, this is precisely what MapReduce was designed for. There is an input set of millions of images, and they need a transformation applied to all of them. It is a batch workload with no latency requirements.

E. Yes, again this is precisely the type of workload that MapReduce was designed for. In this case we have a large corpus of text input (log entries) that we want to parse. There are not tight latency equirements as long as the job can complete in time for daily reports.

Question 2

A. We are not using any locks, therefore we might have race conditions. We might have corrupted state by storing the x value of some average and the y value of some other average.

B. Yes, there is always the very slight chance that the scheduling of threads happens in such a way that at most one client is in the critical section at a time. The chances of being lucky drop dramatically as the number of clients goes up.

C. Yes, we use a very coarse grain lock to guarantee mutual exclusion.

D. We lock the entire data structure to perform the operation. If we have multiple clients, they will run sequentially.

E. It is using reentrant locks because we might grab the same lock twice when we want to write the value of the average. If we don't use reentrant locks the thread can block itself!

F. Yes it can deadlock because we don't release the read locks before acquiring the write lock. Suppose client 1 calls storeAverageInIndex(0, 3, 5) and client 2 calls StoreAverageInIndex(4,6,0).

Question 3

A. Yes, there is a bug. The semaphore-based counting of filled and empty hydrogen buffers is wrong on lines 33 and 40. Each iteration consumes 3 buffers. So lines 33-40 should look like this:

```
33.       P(HydroFull); P(HydroFull); P(HydroFull);
34.       PointerMutex.lock();
35.       myhydro[0] = HydroBuf[FirstFullHydro];
36.       myhydro[1] = HydroBuf[FirstFullHydro+1];
37.       myhydro[2] = HydroBuf[FirstFullHydro+2];
38.       FirstFullHydro = (FirstFullHydro + 3) % (3*MAX);
39.       PointerMutex.unlock();
40.       V(HydroEmpty); V(HydroEmpty); V(HydroEmpty);
```

B. Assume that the rates of production of each chemical are the same. Also assume that tankers are always available for bringing in nitrogen and hydrogen, and taking away ammonia. We need 3 times as much hydrogen as nitrogen to produce one unit of ammonia. So the number of MakeHydrogen, MakeNitrogen, MakeAmmonia, and FillTanker threads should be in the ratio 3:1:1:1. The absolute number of threads can be increased for higher volume production, provided the ratios are always 3:1:1:1.

Question 4

With only a single thread, it would be hard to support crisp user interaction while simultaneously doing something in the background (such as downloading a large file). One could write convoluted code to achieve that, but the simple solution of a foreground thread for interaction and a background thread for download would not be feasible. Another challenge would be to quickly render a complex page on which there multiple independent components (e.g. advertising versus real content). This was acceptable largely because (i) users had lower expectations, based on the state of the art that was available to them, (ii) typical desktop computers only had a single processor, which in turn had only a single core; so multiple threads would have competed for this limited processing resource, and (iii) networks were much slower (typically dialup lines to homes), so using their very limited bandwidth for both foreground and background purposes would have been counterproductive.

Question 5

There could be a deadlock if two people attempt to talk to each other at the same time. The solution is to impose some ordering on People (e.g. give each person an ID number), then use locks instead of **synchronized** so that you can control the order in which locks are grabbed for each interaction. A lower numbered person can ask for a lock to a higher numbered person, but not vice versa. This constrains the patterns of communication, but ensures no deadlock.