THEORY ASSIGNMENT 2

ABHAY SHANKAR K: CS21BTECH11001

(1) 6.8

```
1   void bid(double amount) {
2       if (amount > highestBid)
3           highestBid = amount;
4   }
```

A race condition can be caused when two `bid()` calls and let occur simultaneously, and both `amount` in both cases is greater than `highest bid`. Let these calls be c_1 and c_2, $c_1.amount > c_2.amount$. So if c_1 enters the `if` block in 2 first, and then c_2 enters. Then c_1 executes 3, setting `highestBid` to the correct value, $c_1.amount$. Lastly, c_2 executes 3 (which it ought to, since it has already entered the `if` block), setting the value of `highestBid` to $c_2.amount$.

So, race conditions cause problems.

(2) 6.11

Yes, the given 'compare and CAS' idiom is viable. Since any number of threads can read a shared variable, the surrounding `if` block does not impact the efficacy of the algorithm. Since the `compare_and_swap()` function is still called within the `if` block, no two threads can acquire the lock simultaneously. The extra branch instruction will preserve mutual exclusion.

(3) 6.23

An implementation of a limited open-socket-count with semaphores is fairly straightforward. We can create a semaphore initialised to N, and the code for accepting a new connection can be placed between a `sem_wait()` call and a `sem_post()` call. If more requests arrive, they will be blocked by the wait call.

(4) Reader - writer.

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
int history = 0;

    writer() {
        while (true) {
            wait(rw_mutex);
            history = 0;
            ...
            /* writing is performed */
```

```
12                ...
13                signal(rw_mutex);
14            }
15        }
16
17        reader() {
18            while (true) {
19                while(history > 20); // keeps incoming threads occupied, ensures that semaphore released in 34 is not taken up by a reader > 20 times in a row.
20                wait(mutex);
21                read_count++;
22                history++;
23                if (read_count == 1 || history == 1)
24                    wait(rw_mutex);
25                signal(mutex);
26
27                ...
28                /* reading is performed */
29                ...
30
31                wait(mutex);
32                read_count--;
33                if (read_count == 0 || history > 20) // too many readers executed in a row, gives writer a chance.
34                    signal(rw_mutex);
35                signal(mutex);
36            }
37        }
```

If twenty readers acquire the lock in a row, then further requests by readers are blocked (busy wait), and once all active readers terminate, a writer (since no readers are waiting on the lock) will enter. It then resets the `history` variable, allowing readers to make requests for the lock again. Once the writer finishes, any one of the reader/writer threads may acquire the lock.

(5) Atomic increment - using critical section. Starvation-freedom ensured through bounded waiting.

```
1    int incr(int n) {
2        while (true) {
3            waiting[i] = true;
4            key = 1;
5            while (waiting[i] && key == 1)
6                key = compare and swap(&lock,0,1);
```

```
        waiting[i] = false;
        n++;
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
            j = (j + 1) % n;
            if (j == i)
                lock = 0;
            else
                waiting[j] = false;
    }
}
```