# Theory Assignment 2

**Aaryan, CO21BTECH11001**

**Q.1** Exercise 6.8 on page EX-17 of the book (page 346 of the pdf).
Solution:

The given function is as follows:

```
void bid(double amount) {
    if (amount > highestBid)
        highestBid = amount;
}
```

Let's suppose that two threads $T_1$ and $T_2$ are executing this function concurrently.

Then it may result that both the threads are accessing and manipulating the value of the variable $highestBid$ concurrently and the outcome of execution will be different from what is expected.
Consider an example with value of $highestBid$ is $50$. The thread $T_1$ tries to place amount $100$ as bid and $T_2$ tries to place $150$ as a bid.
The order of execution can be following:

$T_1$: executes $if\ (amount > highestBid)$ {Condition holds true}

$T_2$: executes $if\ (amount > highestBid)$ {Condition holds true}

$T_2$: executes $highestBid = amount$ $\{ highestBid = 150 \}$

$T_1$: executes $highestBid = amount$ $\{ highestBid = 100 \}$

Notice that we arrived at an incorrect value of $highestBid = 100$. The expected value was $150$.

To prevent the race condition to occur, the function $bid$ should be marked synchronized and should be executed atomically.

**Q.2** Exercise 6.11 on page EX-19 of the book (page 366 of the pdf).
Solution:

The given approach is:

```
void lock_spinlock(int *lock)
{
    while (true) {
    if (*lock == 0) {
        /* lock appears to be available */
        if (!compare and swap(lock, 0, 1))
            break;
        }
    }
}
```

This will work appropriately for implementing the spinlocks. Although the approach is different.
It may happen that even if the lock appears to be available for a thread, it may not be able to enter the critical section.
To show this, let's suppose there are two threads executing concurrently.

The order of execution can be following:

Thread 1: executes $if\ (*\ lock\ ==\ 0)$ { Condition holds true}
Thread 2: executes $if\ (*\ lock\ ==\ 0)$ { Condition holds true}
Thread 1: executes $if\ (!\,compare\_and\_swap(lock, 0, 1))$ { Condition holds true, lock = 1}
**Thread 1 enters critical section**
Thread 2: executes $if\ (!\,compare\_and\_swap(lock, 0, 1))$ { lock is 1 so, Condition is false}
**Thread 2 is still spinning in spin lock.**

In this way, lock appears to be available for thread 2, but it didn't enter the critical section.
However, mutual exclusion is held because if one thread is inside the critical section, another thread will certainly be spinning in spin lock.

**Q.3** Exercise 6.23 on page EX-23 of the book (page 370 of the pdf).
Solution:

Let the maximum number of connections that server can make is $N$.
Therefore, we can make a semaphore $available$, which is initialized to $N$.
Whenever a new connection wants to connect, it has to wait using the $available$ semaphore. When an existing connection is released, it will signal via $available$ semaphore.

In this way, it will be ensured that the maximum number of connections at any point of time in $N$.

Here is the implementation for the same:

```
semaphore available = N;

void connect()
{
   wait(available);

   /***CONNECTED***/

   signal(available);
}
```

**Q.4** In the class we discussed a solution for solving the readers-writers problem. But it can cause the writer threads to starve. Can you develop a solution in which no thread starves?

Solution:

In readers-writers problem, starvation of writers can be prevented by introducing an `entry_mutex`. In this case everyone (reader or writer) has to first acquire the `entry_mutex` and release it before it enters the critical section. Let's suppose that $k$ readers are reading (i.e., inside the critical section). Then, a writer came and acquired the `entry_mutex`. So, even if readers keep coming in the queue, they have to wait for `entry_mutex` to be released. Thus, when the $k$ readers exit the critical section, the writer will enter the critical section and everyone else (readers or writers) will wait.

For implementation of the above method, three semaphores `entry_mutex, rw_mutex, mutex` will be initialized to 1.
Also, `read_count` will be initialized to 0.

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

Here is the implementation of reader thread according to above mentioned method:

```
void reader()
{
    wait(entry_mutex);
    wait(mutex);
    read_count++;

    if (read_count == 1)
        wait(rw_mutex);

    signal(mutex);
    signal(entry_mutex);

    /***CRITICAL SECTION***/

    wait(mutex);
    read_count--;

    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
}
```

Here is the implementation of writer thread according to above mentioned method:

```
void writer()
{
    wait(entry_mutex);

    wait(rw_mutex);

    signal(entry_mutex);

    /***CRITICAL SECTION***/

    signal(rw_mutex);
}
```

**Q.5** In the class we discussed about atomic increment operation implemented using CAS instructions. But that implementation does not ensure guaranteed termination of each method, i.e. starvation-freedom. Can you modify this method so that the modification ensures that every thread invoking increment method will eventually terminate.

Solution:

If we implement atomic increment operation using CAS using the following:

```
void increment(atomic_int *v)
{
      int temp;
      do {
            temp = *v;
      }
      while (temp != (compare_and_swap(v,temp,temp+1));
}
```

Then, it may happen that threads will keep coming and increment the value of v and for some thread, the value of v is never equal to the expected value ( value of `temp` ). Therefore, this implementation doesn't guarantee the termination of each thread.

To guarantee the termination of each thread, we can introduce two semaphores `increment_mutex` and `entry_mutex` which will ensure that no thread starves.

```
semaphore increment_mutex = 1;
semaphore entry_mutex = 1;
void increment(atomic_int *v)
{
      wait(entry_mutex);
      wait(increment_mutex);

      signal(entry_mutex);

      // Now, only one thread is incrementing at a time.
      int temp = *v;
      // Using compare and swap to increment v
      compare_and_swap(v,temp,temp+1);
      signal(increment_mutex);
}
```

In the above implementation, only one thread is incrementing at a particular time and no thread will starve.
We can also use `atomic_fetch_add()` function or ++ operator to increment the atomic variable.