CO21BTECH11002
Aayush Kumar
OS2 Theory Assignment 2

1)

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

Race conditions can happen if multiple bidders call the function simultaneously. Let's say there are two bidders with bid amounts 1000 and 1100 and the previous bid being 900.

- Bidder1 calls the bid function with the amount 1000.
- Bidder2 calls the bid function with the amount 1100.
- Bidder2 checks that the previous amount 900 is less than 1100.
- Bidder1 checks that the previous amount 900 is less than 1000.
- Bidder2 sets highestBid = 1100.
- Bidder1 sets highestBid = 1000.

In this case highestBid ends up being 1000 which is wrong. This way data race can happen in this situation.

This data race condition can be solved by placing the 'if' statement in the critical section and making it mutually exclusive.

```
void bid(double amount) {
        // wait while some other person is changing highestBid
        while(atomic_flag_test_and_set(&lock));

        if(amount > highestBid) {
                highestBid = amount;
        }

        lock.clear() // release the lock
}
```

2)

Compare-and-Swap:

```
void lock_spinlock(int *lock) {
   while (compare_and_swap(lock, 0, 1) != 0)
      ; /* spin */
}
```

Compare and Compare-and-Swap:

```
void lock_spinlock(int *lock) {
{
   while (true) {
      if (*lock == 0) {
         /* lock appears to be available */

         if (!compare_and_swap(lock, 0, 1))
            break;
      }
   }
}
```

The “compare and compare-and-swap” idiom works appropriately for implementing spinlocks. CAS takes the current value of lock and sets its new value based on the new and expected value parameters passed. Regardless of any changes, CAS returns the actual value of the lock. Hence, when the lock becomes free, i.e., its value becomes zero, the first method breaks out of the loop since CAS returns actual value of lock, i.e., 0. The loop continues as long as lock is equal to 1. Similarly, in the second method, the loop continues while lock is 1. When lock becomes 0, the if statement is executed and the loop breaks since CAS returns 0. Hence,

both the methods work in a similar way and can be used to implement spin locks.

3)
Semaphores are used for interprocess communication. To limit the number of connections to the server, a semaphore can be created of size equal to the number of max connections. Everytime a new connection is requested, the server would wait on semaphore and everytime the connection is released, the server would signal on semaphore. A sample code for the same can be:

```
S = N // no. of max connections
mutex = 1

while(true) {
        // wait if connection limit exceeds
        wait(S);
        wait(mutex);

        // Create Connection

        // Signal when connection is closed
        signal(mutex);
        signal(S);
}
```

4)
The traditional reader-writer solution uses two mutexes and one variable initialized as:

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

The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated. The semaphore rw_mutex functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers that enter or exit while other readers are in their critical sections.
This solution can result in starvation of writers when trying to enter the critical section. This can happen if readers keep coming one after another, thus the writers would never acquire the rw_mutex, thus being starved.

This problem can be solved by introducing another mutex, say entry_mutex initialized as 1. Every time a process, be it reader or writer, tries to enter the critical section, it will first have to acquire the entry_mutex and it is signaled when the process enters the critical section. This way, if a reader comes after a writer then it won't be able to enter the critical section since the writer would have already acquired the entry_mutex.

A pseudo-code for above approach can be:

Reader:

```
while(true) {
        // wait if entry_mutex is already occupied
        wait(entry_mutex);

        // wait if entry mutex is free but mutex is already occupied
        wait(mutex);

        read_count++;

        // first reader has entered
        if(read_count==1) {
                wait(rw_mutex)
        }

        signal(mutex);
```

```
        // reader has entered critical section, hence, entry_mutex is signaled
        signal(entry_mutex)

        // reading is performed

        wait(mutex);
        read count--;

        // last reader frees rw_mutex
        if (read count == 0) {
                signal(rw_mutex);
        }

        signal(mutex);

}

Writer:

while (true) {

        // wait if entry_mutex is already occupied
        wait(entry_mutex);

        // wait if rw_mutex is already occupied
        wait(rw_mutex);

        // writer has entered critical section, hence, entry_mutex is signaled
        signal(entry_mutex)

        /* writing is performed */

        signal(rw_mutex);
}
```

5)
A thread can invoke a non terminating atomic increment call if there are a huge number of processes that are continuously incrementing the value of the same variable.
This situation can be handled using semaphores. We can introduce a mutex, say entry_mutex, which needs to be acquired anytime a thread tries to use atomic increment. Each request will be stored in a waiting queue. Priority will be given based on how long the threads have waited. Hence, whichever thread is in the waiting queue for the longest time will acquire the mutex.
This ensures that every process will get a chance to execute.

```
atomic int v;
while(true) {

        wait(entry_mutex);

        int temp = v;

        compare_and_swap(*v,temp,temp+1);

        signal(entry_mutex);
}
```