1. 1) Conditional variables signal( ) will not do anything unless there is a thread in wait( ) whereas semaphores post ( ) and wait ( ) will increment and decrement regardless. 2) when we have a semaphore initialized to some number of resources as a thread makes use of that resource semaphores are decremented. When semaphores reach zero the thread that attempts to access the resource is put on wait. On the other hand, conditional variables put threads on wait until a condition is met waking all waiting threads.

2. A correct mutex must be 1) mutually exclusive 2) deadlock free 3) starvation free. Semaphores when initialized to some other higher value than 1 does not guarantee mutual exclusion which is a possible initialization for semaphores to work as supposed. However for semaphores to work properly they have to be deadlock free and starvation free. If a semaphore is currently zero and a process attempts to enter the semaphore, the process is put on a wait state until some other process increments that semaphore. However if there was a deadlock the process attempting to enter the semaphore will be put on an infinite wait state. Also we want semaphores to be starvation free because in the case where multiple processes are waiting to enter the semaphore we want all of them to be processed instead of one or more processes reentering multiple times.

   Condition variables are also not mutually exclusive. It can have multiple threads all waiting at the same time for a signal. We want condition variables to be deadlock free because we want to free the threads that are waiting for them to synchronize when the condition is met. Also we want it to be starvation free because giving the lock back to the thread that just went into wait and unlocked will result in an infinite process of locking and unlocking.

3. Std:: mutex mtx;

```
int aggregateStats( double stat_a, double stat_b, double stat_c) {

        mtx.lock( );

        sum_stat_a += stat_a;

        sum_stat_b -= stat_b;

        sum_stat_c -= stat_c;

        int temp = sum_stat_a + sum_stat_b + sum_stat_c;

        mtx.unlock( )

        return temp;

}
```

4.
```
Std:: mutex mtx1;

Std:: mutex mtx2;

Std:: mutex mtx3;

int aggregateStats( double stat_a, double stat_b, double stat_c) {

        mtx1.lock( );

        sum_stat_a += stat_a;

        int temp1 = sum_stat_a

        mtx1.unlock( );


        mtx2.lock( );

        sum_stat_b -= stat_b;

        int temp2 = sum_stat_b;

        mtx2.unlock( );


        mtx3.lock( );

        sum_stat_c -= stat_c;

        int temp3 = sum_stat_c;

        mtx3.unlock( );


        return temp1 + temp2 + temp3;

}
```

By keeping a snap shot of the sum_stat_* within the locks the return value is guaranteed to return the correct sum of the original sum_stat_* and the parameter values.

5. Taking a queue for example, two threads attempt to dequeue from an empty queue and thus is put on wait. Let's assume that enqueue is implemented in a way that the signal is sent to the thread only when we enqueue from an empty state to a non-empty state. When enqueue ( ) is called a signal is sent to the thread. However, the signal is sent to only one thread as they share the same condition variable. The other thread will be indefinitely waiting. In order to fix this we must implement enqueue ( ) so that it sends signals every time enqueue ( ) is called or instead of sending a signal, broadcast to all threads so that no thread is left in the lost wake up state.

6. A)

```
        mutext M
        Int size = 0;
        Int MAX_SIZE = max size of queue
        Condition variable CV


bool push ( item ) {
        M.lock( );
        While ( size == MAX_SIZE ) {
                Wait( & CV, M );
        }
        Size ++;
        Enqueue( item) ;
        Signal( &CV );
        M.unlock( );
        Retun true;
}
```

```
item pop ( void ) {
        M.lock( );
        If ( size == 0 ) return null;
        Item item= dequeue ( );
        Size --;
        Signal ( &CV );
        M.unlock ( )
        Return item;
}


Item listen( ) {
        M.lock( );
        While ( size == 0 ) {
                Wait( &CV , M );
        }
        Size -- ;
        Item item = dequeuer ( );
        Signal ( &CV );
        M.unlock ( )
        Return item;
}
```

B) Sem_t full initialized to 0

```
            Sem_t empty initialized to number of empty slots in queue

            Sem_t mtx initialized to 1

Bool push ( item ) {

            Sem_wait ( &empty ) ;

            Sem_wait ( &mtx);

            Enqueue ( item);

            Sem_post ( &mtx);

            Sem_post ( &full);

            Return true;

}

Item pop ( ) {

            Int temp = sem_getvalue ( & full );

            If ( temp == 0 ) return null;

            Sem_wait ( &mtx);

            Item item = dequeue ( );

            Sem_post ( &mtx);₩

            Return item;

}

Item listen( ) {

            Sem_wait ( &full );

            Sem_wait ( &mtx );

            Item item= dequeue ( );

            Sem_post ( &mtx );

            Sem_post ( &full );

            return item;
```