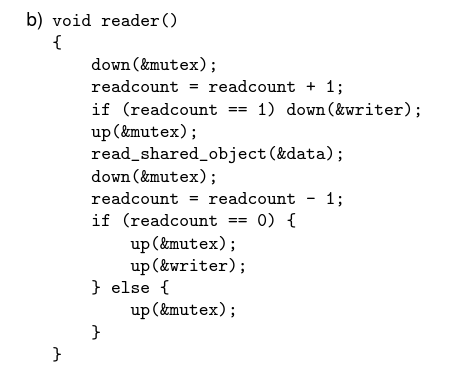
3.1)  
A white screen with black text

Description automatically generated  
a) In the first scenario, we have two last lines exchanged. So we are giving access to critical section, which mutates readcount variable, before we check it for zero to allow writer’s critical section to be executed, and therefore we may end up with not giving access to writers critical section, because readcount may be modified by another process before last line is executed and therefore, it may result in unexpected behavior.  
  
b) In the second scenario we perform the readcount check for 0 before upping the mutex, so now it is fine, but another problem raises. Because we are ‘upping’ mutex before writer, so another process will continue execution from the first line and it may also ‘down’ writer, so It may result in deadlock in this case.  
  
A computer code with black text

Description automatically generated

c) In the third scenario, if reader and writer are executed simultaneously, It will result in deadlock.  
Firstly, reader downs ‘mutex’ (it’s the first one, so it does not have to wait), and writer() downs ‘writer’ semaphore, and then writer() downs ‘mutex’ and waits for the reader to up it. But at the same time, reader waits for the writer() to up ‘writer’ semaphore, so they are waiting for each other =>  
deadlock.

3.2) task2.c and on the next page of this word document

3.2) c)

I chose the interval [1,10000000] ( from one to 10 millions ). I am executing my program without verbose option, because it lowers the performance by printing to the standard error.  
  
Thread count: 1   
- Average execution time: 1.195 seconds  
Thread count: 2  
- Average execution time: 0.77 seconds  
Thread count: 3  
- Average execution time: 0.56 seconds  
Thread count: 4  
- Average execution time: 0.43 seconds  
Thread count: 5  
- Average execution time: 0.40 seconds  
Thread count: 10  
- Average execution time: 0.40 seconds  
Thread count: 50  
- Average execution time: 0.36 seconds  
Thread count: 100  
- Average execution time: 0.36 seconds  
Thread count: 500  
- Average execution time: 0.38 seconds  
Thread count: 1000  
- Average execution time: 0.40 seconds  
  
A graph with a line drawn on it

Description automatically generated  
Before 100 threads, the derivative is negative, so there is a point of increasing the number of threads, but after 100 threads, it is positive and there is no point of increasing the number of threads. The thread allocation itself is highly time-consuming and resource-consuming, so after 100 threads, we are not gaining performance, but the allocation itself takes more time, than computing the result in one thread.