Task 4 Report

Yes, I experienced deadlock when testing this task. If at some point a thread attempted to access the semaphore that dictated the free space in another threads mailbox and found it full, the thread would go into that semaphores queue and be put to sleep. If that same recipient attempted to send a message to that same thread and its mailbox was full, it would go to sleep. At that point, there is no way for either thread to wake up being that there is no way for those threads to read their mail. There are other deadlocking examples I found but they are all based on the principle of that previous example. It could involve more than 2 threads in a circular chain, but it boils down to there not being a possible way for those threads to free their resource.

**ENTER GERALD’s PORTION HERE**

Task 6 Report

Task 4: Question 1

Task 4 was implemented using 2 functions as thread controllers and 4 helper functions. The post office data was stored in the form of a struct which contains all but 2 of the necessary global variables, the other 2 being the mailbox semaphore and free space semaphore which are included outside of the struct for ease of readability purposes. The other structure of note is a mail struct which contains the message being sent as well as an integer representation of the sender. When the task is run, the root thread forks to the main controller thread for task 4. This thread prompts the user for the 3 specified inputs required. The thread calls the helper function for input validation and the program will loop infinitely until appropriate input is received from the user (integer values between 1 and 10000). When all 3 valid inputs are received, the thread checks if the user input 1 as the number of people participating in the simulation. If that is the case, a message indicating that this has occurred is printed and the program exits. Otherwise all the globals are initialized using the input and the number of people are forked as individual threads. Each person thread then goes onto the cpu and executes the algorithm provided in the specs. Access to each mailbox is restricted by a semaphore only allowing one person access at a time whether it be for reading or writing. The free space semaphore limits the number of letters allowed for each person to receive. If resources are not available, the thread is placed into the corresponding semaphores waiting queue and put to sleep. In addition, there are 2 other semaphores that are used to limit the threads from accessing other global variables that are constantly being changed by the threads. The writer semaphore only allows for one thread at a time to send a message. This is to prevent the total mail count from being changed by multiple threads concurrently which leads to more than the allotted number of mail being sent. The other semaphore is a deadlock semaphore which is used to prevent the manipulation of a deadlock counter variable from being changed by multiple threads at a time. This corresponds with an array of Boolean values that is equal to the number of threads in the simulation. When a thread attempts to access the free space semaphore, prior to the wait() call, the thread is marked as having deadlock potential, changes its value to true, and increments the counter. When the free space semaphore is successfully attained, the value is reverted to false and the count is decremented. This is important because unlike in task 3 where deadlock prevention is implemented after failure to send 3 times, deadlock has to be avoided entirely in this task because if threads deadlock, being that semaphores are being used the threads are no longer accessible because they are asleep and even if a wake is called on the thread it is stuck in the semphores while loop and continuously put to sleep. To handle this, there is a deadlock prevention statement that checks if the recipients mail is full prior to calling the wait on free space. If that condition is true and the deadlock count is greater than or equal to half of the total number of people in the task, the thread double checks it's mailbox, if it has mail again already from another thread, it reads it then yields once. Otherwise, if no mail it will just yield once and try again when it regains the cpu. The program will continue through the algorithm until all messages have been sent and read.

I encountered a number of bugs while implementing this task, one due to my own oversight, and the rest due to unforeseen circumstances. The first bug was my code seg faulting because I didn’t make a temp message char array large enough to handle the message lengths I had at the time. The next issue I had was everything working absolutely fine until the number mailbox capacity was dropped down to 1 or most of the time with 2. This was when the program would deadlock and not come anywhere close to completing. This issue stemmed from the statements for task 3 where we were limited to calling deadlock prevention methods until 3 attempts to send a message were failed. Seeing as the threads would go to sleep in the semaphores queue and not wake up until free space was available, free space could never be available again if the threads deadlocked and I wasn’t sure how to approach that with those constraints. After talking to Zack to confirm what was allowed for the specs of task 4, working around the deadlocking was simple. The other major bug I ran into was before I implemented a semaphore that limited the program to one writer at a time. Before that semaphore was present, the message count at the end of a successfully ran simulation were grossly higher than the limited number of messages imposed by the user. Adding the semaphore instantly fixed the issue. Other than those problems, everything worked fine.

Task 4: Question 2

Data structures used in this implementation were structs, a 2d matrix, arrays, and instances of semaphores. One struct was used to incapsulate the global variables necessary for the post office simulation and the other was a representation of the mail being sent. The 2d matrix was used to represent the mailboxes and there were various 1d arrays used as well. The semaphores were used as semaphores. The algorithm used is the one provided in the specs of project 2.