**Report lab 7.7**

Exercise 1:

Program 1 (inputReceiptProc) processing flow:

1. Program 1 creates a pipe using the `pipe` function to establish communication with the child process.

2. It forks a child process using the `fork` function to create a separate process for communication.

3. The child process is responsible for receiving input from the parent process and performing further actions.

4. The parent process is responsible for accepting user input from the console and sending it to the child process through the pipe.

5. The child process reads the input value from the pipe using the `read` function.

6. It generates a key for the message queue using `ftok` and acquires the message queue ID using `msgget`.

7. The child process constructs a message containing the received input value and sends it to Program 2 through the message queue using `msgsnd`.

8. It waits to receive a response from Program 2 by calling `msgrcv`.

9. Upon receiving the response, it prints the received result on the console.

10. The child process closes the read end of the pipe and exits.

11. The parent process continues to prompt the user for input and repeats the process.

Program 2 (opeProc) processing flow:

1. Program 2 generates a key for the message queue using `ftok` and acquires the message queue ID using `msgget`.

2. It creates shared memory using `shmget` to access the shared memory region.

3. The program attaches the shared memory segment to its address space using `shmat`.

4. It initializes a mutex using `pthread\_mutex\_init` for synchronization.

5. Program 2 enters an infinite loop to receive messages from Program 1.

6. Upon receiving a message from Program 1, it locks the mutex using `pthread\_mutex\_lock`.

7. It reads the current value stored in shared memory and the received value from Program 1.

8. Program 2 prints the received value and the stored value on the console.

9. It sleeps for 5 seconds using `sleep` to simulate some processing time.

10. Program 2 calculates the result by adding the received value and the stored value.

11. It updates the shared memory with the calculated result.

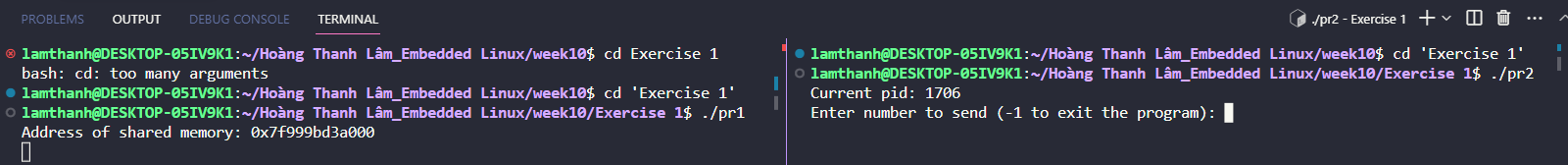
12. Program 2 unlocks the mutex using `pthread\_mutex\_unlock`.

13. The program constructs a message containing the calculation result and sends it back to Program 1 through the message queue using `msgsnd`.

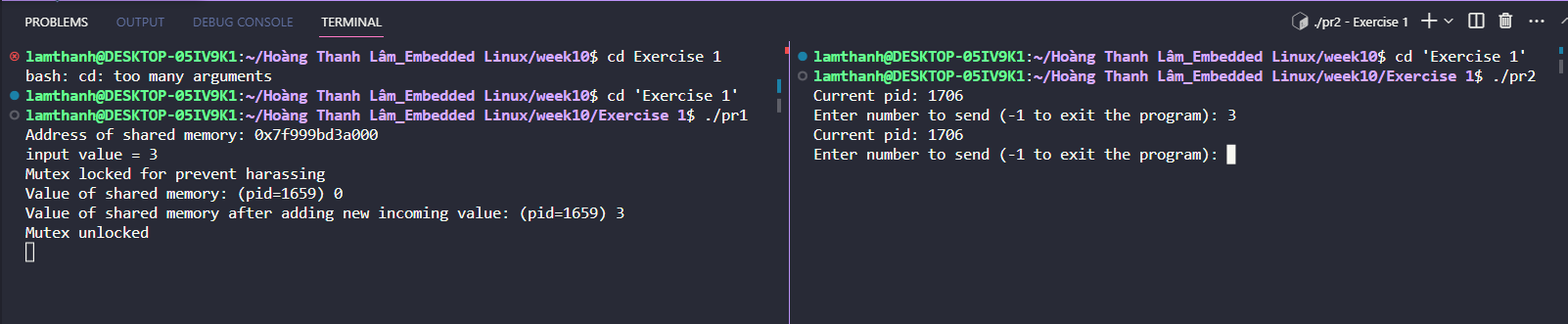
14. The loop continues, and the program waits to receive the next message from Program 1.

15. Finally, Program 2 cleans up the mutex using `pthread\_mutex\_destroy` and detaches the shared memory segment using `shmdt`. It also removes the shared memory segment using `shmctl`.

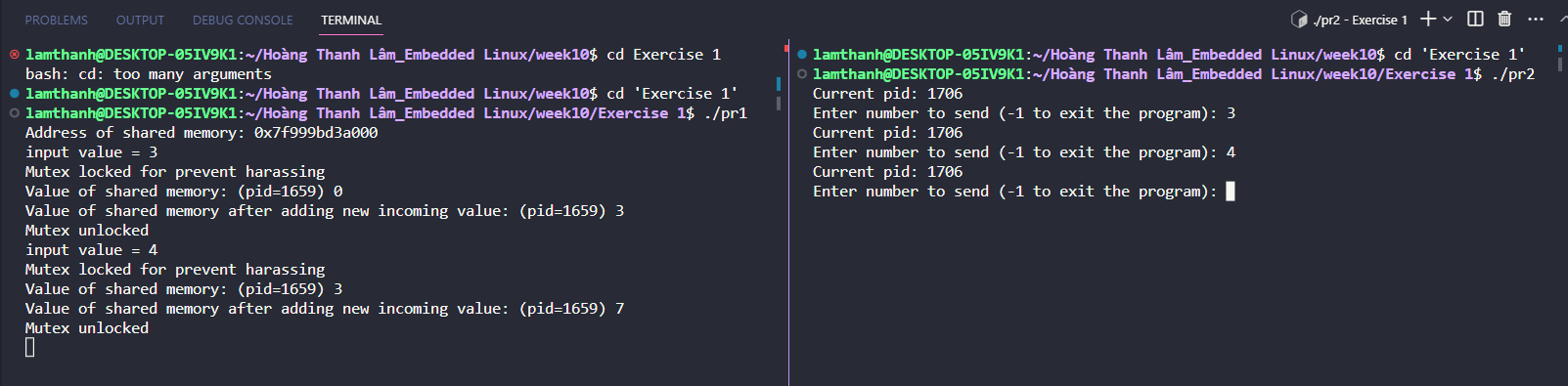
Note that both programs run in an infinite loop to continually receive and process messages, except for Program 1, which waits for user input.



After sent the number:

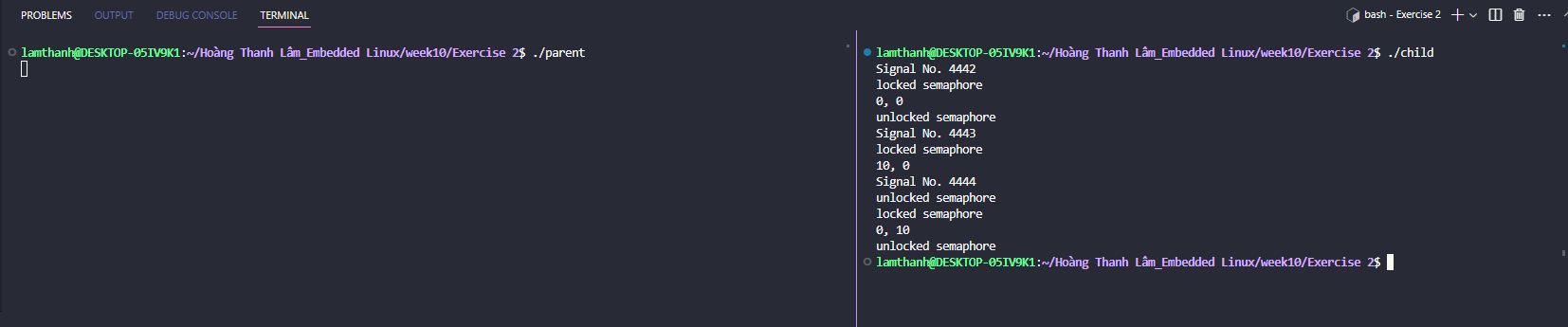


Sent number again:



Exercise 2:

In this program, use use an additonal library of doubly linked list to save information of incoming child processes using the same shared memory. child.c will spawn 3 processes which will make use of the shared memory to change values of the variables. parent.c will take notice of the comming order of these processes by putting them into a linked list (which can be consider as a queue here). After each done process, it will pick that process out of the queue and allow the next process to execute.



* We can see on the terminal of child, there are 3 process forked. Since they are monitored by parent, the value of x and y will be changed respectively from (0, 0), to (10, 0) and (0, 10) without any conflict.