Problem 1 (5 points): Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: deposit (amount) and withdraw (amount). These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the withdraw() function and the wife calls deposit(). Write a short essay listing possible outcomes we could get and pointing out in which situations those outcomes are produced. Also, propose methods that the bank could apply to avoid unexpected results.

Assume that two function: deposit(amount) and withdraw(amount) have 3 stages:

- read account balance from system
- increase/decrease account balance
- send new account balance to the system

If the balance in the account is 1000, the husband calls withdraw(500) and the wife call deposit(1000), since these two function run concurrently, we can have some possible outcome:

1.

S0: Husband calls withdraw(500) execute register1 = 1000

S1: Husband calls withdraw(500) execute register1 = register1 – 500 = 500

S2: Wife calls deposit(1000) execute register2 = 1000

S3: Wife calls deposit(1000) execute register2 = register2 + 1000 = 2000

S4: Husband calls withdraw(500) execute balance = register1 (balance = 500)

S5: Wife calls deposit(1000) execute balance = register2 **(balance = 2000)**

2.

S0: Husband calls withdraw(500) execute register1 = 1000

S1: Husband calls withdraw(500) execute register1 = register1 – 500 = 500

S2: Wife calls deposit(1000) execute register2 = 1000

S3: Wife calls deposit(1000) execute register2 = register2 + 1000 = 2000

S4: Wife calls deposit(1000) execute balance = register2 (balance = 2000)

S5: Husband calls withdraw(500) execute balance = register1 (**balance = 500**)

3.

S0: Husband calls withdraw(500) execute register1 = 1000

S1: Husband calls withdraw(500) execute register1 = register1 – 500 = 500

S2: Husband calls withdraw(500) execute balance = register1 (balance = 500)

S3: Wife calls deposit(1000) execute register2 = 500

S4: Wife calls deposit(1000) execute register2 = register2 + 1000 = 1500

S5: Wife calls deposit(1000) execute balance = register2 **(balance = 1500)**

Method: Bank can use Mutex lock: at a time, only one function can change and sent account balance. If husband call withdraw() and the wife call deposit() concurrently, then if withdraw() function is running, deposit() may have to wait until withdraw() function complete its execution in critical section and vice versa.

```
void withdraw() {
    pthread_mutex_lock(&lock);
    //withdraw
    pthread_mutex_unlock(&lock);
    //remainer section
}
```

```
void deposit() {
    pthread_mutex_lock(&lock);
    //deposit
    pthread_mutex_unlock(&lock);
    //remainer section
}
```

Problem 2 (5 points): In the Exercise 1 of Lab 5, we wrote a simple multi-thread program for calculating the value of pi using Monte-Carlo method. In this exercise, we also calculate pi using the same method but with a different implementation. We create a shared (global) count variable and let worker threads update on this variable in each of their iteration instead of on their own local count variable. To make sure the result is correct, remember to avoid race conditions on updates to the shared global variable by using mutex locks. Compare the performance of this approach with the previous one in Lab 5.

When running this program and the pi_multi-thread that we created in lab5, we have the following result:

```
kuzu@NaruKiri:/mnt/d/tailieu/He_Dieu_Hanh/Lab/final/HDH/lab5$ time ./pi_multi-thread 999999999
total of inside point: 785395487
3.141582

real    0m7.007s
user    0m50.465s
sys     0m0.200s
kuzu@NaruKiri:/mnt/d/tailieu/He_Dieu_Hanh/Lab/final/HDH/lab5$ cd ../lab6
kuzu@NaruKiri:/mnt/d/tailieu/He_Dieu_Hanh/Lab/final/HDH/lab6$ time ./pi_multi-thread 999999999
total of inside point: 785392527
3.141570

real    0m10.818s
user    0m10.779s
sys     0m0.000s
```

With the same input (999.999.999), we see that program using the same global variable and mutex lock in Lab6 run slower that the program with local variable we have created in Lab5. I think this is correct, since we're using mutex lock, when 1 thread is using the global variable, other threads have to wait until that thread complete its execution in critical section and return the lock.