PROBLEM 1:

What is "race condition"?

- "race condition" is the situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the order in which the access takes place.

- It is typically behaved in two situations:

    - "Read-Modify-Write critical sections

    - "Check-Then- Act critical sections

The function deposit(amount) can be understood as depositing money into account with "amount".

As a result, we can translate it into CPU instructions like this:

    - Read current amount of money in the account (memory location)

    - Add "amount" to current amount

    - Write new amount of money to the memory

Decomposing the withdraw(amount), we still get almost the same principle:

    - Read current amount of money in the account (memory location)0

    - Subtract "amount" from current amount

    - Write new amount of money to the memory

-> the problem lies into Read-Modify-Write situation.

Assume that the current amount of money in the bank is 1000 USD. Then assume thread A is the husband, thread B is the wife.

Then if thread A withdraw amount of 500USD, the final amount would be 500USD.

Same with thread B deposit amount of 500USD, the final amount would be 1500USD.

-> because 2 threads are running parallel, then both situations are totally wrong.

--> thread which execute last would be the result in the bank memory. So, the bank account will either have more than 500USD or missing half of their money.

=> Solution is preventing the race condition here, synchronization is needed.

- Use mutex lock or semaphore.
- use atomic instructions.

PROBLEM 2:

Thread 2 or 3 might be executed once or none before the continuous execution of watch_count() thread. This problem happened due to the lack of synchronization. To explain more clearly, there was no signal passing mutually from the inc_count() thread to the watch_count() thread at all, and race condition happened; comparing to the program cond_usg, in which the global variable "count" should reach the threshold of 20 for the signal to be passed

PROBLEM 3:

About the performance of 2 type of implementation (using count variable to store count value at each thread and using mutex lock to synchronize data shared between threads).

Lab 3 : It doesn’t exists race condition, more stable and faster. But required more memory segments.

Lab 5: Using less memory segments (1 count variable compared to each thread-counts) However, the mutex lock technique has to ensure the synchronization of the algorithm. Mutex lock is slower because each thread has to wait until one thread has done.