VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEMS

Report #06

Lab 06 : Synchronization

Advisor: Tran Viet Toan
Students: Tran Long Vi - 1814804

HO CHI MINH CITY, SEPTEMBER 2020

1 PROBLEM 1

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 These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance.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.

Proof:
When the husband calls the *withdraw()* function and the wife calls *deposits()*, the results may happen 3 possible cases as follow:

- The *withdraw()* function has been executed to decrease static variable *money*(total amount of money's both husband and wife) before the *deposits()* function is executed to increase *money*. The result of this case is:
$$money = money - amount\_withdrawn + amount\_deposit$$
- The *withdraw()* function is executed after the *deposits()* function has been executed. The result of this case is:
$$money = money + amount\_deposit - amount\_withdrawn$$
- The *withdraw()* function is executed concurrently with the *deposits()* function. There are 2 results of this case:
  - withdraw(): $money = money - amount\_withdrawn$
    deposit(): $money = money + amount\_deposit$
    → As a result, the amount of money increases by *deposit()* function without being reduced by *withdraw()* function.
  - deposit(): $money = money + amount\_deposit$
    withdraw(): $money = money - amount\_withdrawn$
    → As a result, the amount of money decreases by *withdraw()* function without being increased by *deposit()* function.

  ⇒ This case is happened because static variable *money* has not been saved and updated before other function accesses that variable's address.

To overcome this problem, the bank can lock account balance when an account is trying to change it. The *mutex* key will lock all other processes trying to change the value whenever current process is executing and only unlock when the current process finished.

2 PROBLEM 2

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.

Proof:
It takes more time to execute a program with *mutex* dispute resolution than an equivalent program without using *mutex*. When using mutex, to protect global resources, a thread must go through all three steps:

- Current thread calls *mutex* to lock critical section after current thread has accessed to it.
- Executing commands in critical section.
- Unlocking *mutex* after current thread has finished.

Although it takes more time to execute the program, using *mutex* can solve critical problems about disputing common resources, global variables.