Student’s name: Nguyễn Đức Dũng

Class: ICT.02-K61

**Class Exercises**

**Module: Distributed Systems**

**Chapter 6: Synchronization**

**Practical Exercises:**

**Question 1**: Launch this program several times. What do you notice? Explain it!

The results of the program are random numbers smaller than 3000. The reason iAf that when a thread called *exploit()*, it accessed the shared resource *rsc*, which is also being processed by other threads. Therefore, the value it get from *getRsc()* is not the updated value.

**Question 2**: Change the code of the general executable program by replacing *ThreadedWorkerWithoutSync* with *ThreadedWorkerWithSync* to initiate three instances *worker1-3*. What is the difference between the output of this program and that of question 1? Explain it!

The results are always 3000, because *synchronized* is applied for the shared resource, which means only 1 thread can access and process the resource at a time, other threads have to wait.

**Question 3**: Change the code of the general executable program by replacing *ThreadedWorkerWithSync* with *ThreadedWorkerWithLock* to initiate three instances *worker1-3*. What is the difference between the output of this program and the output in question 1? Explain it!

The results are always 3000, because only one thread can hold the lock at a time and only the thread with the lock can use the shared resource, other threads have to wait until the lock is released.

**Question 4**: Complete this file above (in the part **YOUR-CODE-HERE**) with a loop to increase the variable shared by 1 for 5 seconds.  
(hint: time(NULL) will return the present system time in second).

A close up of a screen

Description automatically generated

**Question 5**: Try to increase the value of threads and the value of the constant NUM\_TRANS after each execution time until you obtain the different results between *Balance* and *INIT\_BALANCE+credits-debits.* Explain why do you get this difference.

Because *balance, credits, debits* are shared resources, and without locking mechanism, all threads can freely access them. Therefore, when a thread perform *transactions(),* the resources it use for calculation have already been changed by other threads → incorrect result.

**Question 6**: Try to build and run this program. Launch it repeatedly until you see the difference between *Shared* and *Expect* values. Analyze the source code to understand the problem that leads to this difference.

The problem is that *lock* is also a shared resource, which means when the lock is released, multiple threads can hold the lock at the same time.

**Question 7**: Now, you have to modify the code of the file *without-lock.c* in implementing the *mutex lock* above (you can name it differently like *mutex-lock- banking.c*). Try to launch it repeatedly and evaluate the obtained output. What is the improvement after using *mutex lock*?

The *shared* value is equal the expected value. The improvement is that the lock is guaranteed to be atomic in operation and the act of acquiring a lock cannot be interrupted, which means only 1 thread can be in critical session at a time.

**Question 8**: compare the run times of the two strategies to prove that Fine Locking is faster and much faster on larger load sets.

* Testing with *NUM\_TRANS* = 100000, *n\_threads* = 5, it can be seen that fine lock is faster than coarse lock
  + Coarse lock: A screenshot of a cell phone

    Description automatically generated
  + Fine lock:

A screenshot of a cell phone

Description automatically generated

* Testing with *NUM\_TRANS* = 10000000, *n\_threads* = 5, it can be seen that fine lock is faster than coarse lock:
  + Coarse lock: A screenshot of a cell phone

    Description automatically generated
  + Fine lock:
* In real banking system, each transaction takes a few seconds to process. To simulate this, a small delay is added before releasing each lock using a function:

A screen shot of a computer

Description automatically generated

* Testing results with *NUM\_TRANS* = 5, *n\_threads* = 5, balance calculation delay = 1s, credit and deposit calculation delay = 2s. Fine lock is much faster than coarse lock in this case:
  + Coarse lock: A screenshot of a cell phone

    Description automatically generated
  + Fine lock:

A screen shot of a smart phone

Description automatically generated

**Question 9**: Run this program and what do you get as output? Explain what the *deadlock* is.

There is no output from this program, it stuck somewhere around iteration 6000 → 7000 due to deadlock. Deadlock is a state in which each member of a group is waiting for another member, including itself, to take action. In this case, suppose that *fun\_1* is locking a and wait for *fun\_2* to release lock b, but at the same time, *fun\_2* is locking b and wait for *fun\_1* to release lock a. Both lock a and b are locked and can only be unlocked if one of them is unlocked.