**AC22005 - Computer Systems 2b: Architecture and Operating Systems (17/18)**

**C# ATM simulator report (… words)**

**Team 11:**

*Blair Dignan, Saidnassimov Darkhan*

# Introduction

This report summarizes the work done in a team to create a simulator of a multi-ATM bank system based on the console ATM code and account numbers given by the teacher. The solution represents a centralized banking system with two buttons on the main form. After the “Create ATM” button is clicked, it will create an ATM simulator in a new window. The amount of ATMs that can be opened at a time is only limited by the memory space. The second button will switch the data race fix to ACTIVE or INACTIVE. The ATM’s design, on the other hand, remained rather traditional due to the fact that people are used to that look of an ATM which makes its utilization extremely easy. It is intuitively easy to explore the features of this simulator as it gives all the important data including the balance and the withdrawal options for different account sets.

# Multithreading

In a banking system, multithreading allows different cardholders to use the account. C# has a library dedicated to threads and it can be called by using System. Threading; command. Our solution implemented this library to run every single ATM on a different thread. However, this often causes a data race, which we have encountered and successfully carried out. When speaking about the banking system, data race happens when two or more users access the account and try to withdraw cash and this might lead to a potential loss of money for the bank if not carried out properly.

# Data race

A data race in programming is a condition when two or more instructions from different threads are sent to the same memory location and at least one of those instructions is a “write”. In the banking system data race can be noticed when two users withdraw cash within the same time limit. Hence it is considered an error and some sort of a mechanism should be implemented to fix it. In our solution we have implemented a multithreading system using locks. Lock is essentially designed to enforce a mutual exclusion concurrency control policy. Thus, this solution does not go against concurrency. For example, when two ATMs send an inquiry simultaneously, the system accepts both requests and puts them in the queue and these requests are handled one after another, allowing only one to process at the critical point. Additionally, we used System.Threading.Thread.Sleep function to show the effect of data race. This allowed us to see the process in slow motion.

# Problems encountered

Additionally, we have tried to implement the blocking of the account, if a wrong PIN is entered more than three times in a row, however, due to the lack of time, we were not able to finish it.

Apart from that, another future idea for this assignment could be a possibility to deposit money as well as adding the control of the accounts by the central bank, for example, a possibility to see the amount of money stored on each account.

# Conclusion

We found this assignment extremely useful as it helped us to understand the principles of multithreading and it emphasized the importance of multithreading in the real world.