C# ATM - Group 6

Matthew Foulis

Jüri Ilmjärv

Word Count: 520

Approach:

To start the project, we first studied the command line ATM code and the threading lab practical code. This helped us to understand the functionality required from the ATM program and gave us a starting point for creating and using threads. We then planned and implemented the design and layout of our ATM as a windows form. Once we had the basic functionality working we added a thread to the ATM when a withdrawal was requested. We then added a semaphore around the critical code (the decrement from account balance) and tested this to ensure it was working as required.

Problems and solutions:

The main problem that we faced during the project was demonstrating a data race condition. We were able to implement a thread for each individual ATM and a semaphore to block the critical code however we were unable to demonstrate the difference this made by making the threads access the data at the same time. When the semaphore was in place we could see from the console output we had setup that it was working as required and that one thread was waiting for the other to exit the critical code before continuing. However, when the semaphore was not being used we could not get more than one thread to access the same data. We tried a number of different methods including a countdown timer, an integer check which would continue after two threads were present, and continually putting the thread to sleep for 1 millisecond until the other was present.

To solve this problem, we kept the 1 millisecond check, added a second balance variable, and added additional thread sleeps so that one thread would access balance before it had been updated by another. The 1 millisecond check ensures that the threads are arriving at almost the same time, the first thread then accesses the account balance and assigns it to another integer variable called balanceRC. Whilst this happens the second thread is asleep. The decrement amount is then performed on balanceRC and the second thread awakes and takes the, now incorrect, amount from balance. Balance is then updated by the first thread and the data race occurs. This problem does not occur when the semaphore is in place which allows us to demonstrate our solution.

Another problem we faced was due to our initial design which involved each ATM creating a new version of the account being accessed. This meant that each ATM was decrementing from different instances of the same account and there was no connection between them. To solve this, we moved the account class to a separate file and had each ATM access the same account object.

Features:

The main feature of the ATM is the implementation and demonstration of our solution to the race condition problem. With the use of threads and semaphores we are able to prevent this problem occurring and ensuring only one transaction takes places at a time. An additional feature we have is an account lock which prevents a user from accessing their account if they have entered the wrong pin three times in a row.