Lab 1 - TOCTOU

Group 21 | Olof Lindberg oloflind@student.chalmers.se &
Rikard Roos rikardro@student.chalmers.se

Part 0

To compile and run ShoppingCart.java follow these steps:

1. Open the directory lab1/part0/
2. Run “make all”
3. Run “java ShoppingCart.java”

Part 1

What is the shared resource? Who is sharing it?

> The shared resources are the two files wallet.txt and pocket.txt which can be accessed and edited through the methods getBalance, setBalance, addProduct and getProducts. They are shared between several threads running the programs.

What is the root of the problem?

> The root of the problem is that multiple threads can access these resources simultaneously which can disturb the intended control flow of the program. Specifically, one thread can make changes to the files without considering the other thread’s changes.

Explain in detail how you can attack this system.

> The system can be attacked by starting two threads that make purchases simultaneously. A purchase is made through these steps:
>
> 1. Choose product
> 2. Call getBalance
> 3. Check if product price <= balance
> 4. Update wallet balance to old balance - product price
> 5. Add product to pocket
>
> We attack the system by creating two threads t1 and t2 where t1 makes a call to purchase a car and t2 makes a call to purchase a pen. The concurrent processing can lead to a situation where t1 purchases the car and pays for it but

t2 successfully purchases a pen and overwrites the balance with 29960. We ensure this situation by adding delays after calls to getbalance.

Provide the program output and result, explaining the interleaving to achieve them.

```
$ java ShoppingCart.java
Your current balance is: 30000 credits.
car     30000
book    100
pen     40
candies 1

Your current pocket is:

What do you want to buy? (type quit to stop) simulateRace
Your current balance is: 29960 credits.
car     30000
book    100
pen     40
candies 1

Your current pocket is:
car
pen

What do you want to buy? (type quit to stop) quit
```

We added a case for the entry “simulateRace” which creates two threads t1, t2 where t1 purchases a car and t2 purchases a pen. We get the following interleaving:

1. t1 chooses car
2. t1 calls getBalance -> t1 balance = 30000
3. t2 chooses pen
4. t2 calls getBalance -> t2 balance = 30000
5. t1 checks if car price <= t1 balance -> true
6. t1 calls setBalance -> t1 balance - 30000 = 0
7. t1 adds car to pocket
8. t2 checks if pen price <= t2 balance -> true
9. t2 calls setBalance -> t2 balance - 40 = 29960
10. t2 adds pen to pocket

After the execution, we have a pen and a car in the pocket and the wallet will have 29960 credits, i.e. we successfully purchased a car but paid for a pen.

Part 2

Were there other APIs or resources suffering from possible races? If so, please explain them and update the APIs to eliminate any race problems.

> Other concerns we had with the code were the getBalance and setBalance methods in wallet.java. Both were public with no security whatsoever. We made the decision to make setBalance to private since we saw no scenario where something outside the wallet class should edit the balance. For example, we could add a safeDeposit function similarly to safeWithdraw. Since setBalance can only be accessed through locked functions inside Wallet, we eliminated race problems through setBalance.
>
> The getBalance method is used in ShoppingCart.java when the balance is displayed to the user, therefore making this method private is not an option. Therefore, we added a lock to this method so that only one thread at a time can execute the code in the getBalance method.

When eliminating all race problems, it is important to not implement the protections too aggressively, as it could lead to performance hits in the real world. This is something you should consider in the lab. Why are these protections enough and at the same time not too excessive?

> We updated the backend so that the frontend could not directly update the balance by making setBalance private, and put locks on the public methods getBalance and safeWithdraw. By doing so, requests sent to either method must wait for the other to be executed. While this could be a performance hit, it is absolutely necessary that both resources are secure in order to ensure a correct control flow.

On the next page we can see the output log of running simulateRace which does not work anymore.

```
$ java ShoppingCart.java
Your current balance is: 30000 credits.
car     30000
book    100
pen     40
candies 1

Your current pocket is:

What do you want to buy? (type quit to stop) simulateRace
You can not afford that product.
Your current balance is: 0 credits.
car     30000
book    100
pen     40
candies 1

Your current pocket is:
car

What do you want to buy? (type quit to stop) quit
```