DHA SUFFA UNIVERSITY
Department of Computer Science
CS-2004L
Operating Systems
Fall 2020

Lab 10 – Using Mutex to Fix Race Condition

Objective(s):

- Learn about race condition
- Learn about Mutex

What is a Race Condition?

Race condition is a kind of a bug that occurs in multithreaded applications.

When two or more threads perform a set of operations in parallel, that access the same memory location. Also, one or more thread out of them modifies the data in that memory location, then this can lead to unexpected results some times.In multithreading environment data sharing between threads is very easy. But this easy sharing of data can cause problems in application.This problem is called race condition.

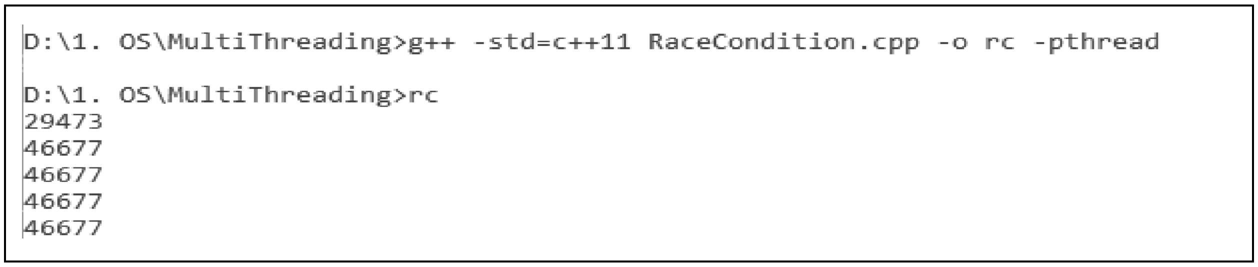
Race conditions are usually hard to find and reproduce because they don't occur every time. They will occur only when relative order of execution of operations by two or more threads leads to an unexpected result.

Example 01 - Race Condition Example

```
#include<iostream>
#include<thread>
#include<vector>


class Wallet{

    private:
        int money=0;
    public:
        void addMoney(int amount)
        {
            for(int i=0;i<amount;++i)
            {
                money++;
            }
        }
        int getMoney()
        {
            return money;
        }
};

int main()
{
    Wallet walletObject;
    std::vector<std::thread> threads;
    for(int i=0; i<5; ++i)
        threads.push_back(std::thread(Wallet::addMoney,&walletObject,10000));
    for(int i=4; i>=0; i--){
        threads[i].join();
        std::cout<<walletObject.getMoney()<<std::endl;
    }
}
```

As addMoney() member function of the Wallet class object is executed 5 times hence it's internal money is expected to be 50000. But as addMoney() member function is executed in parallel hence in some scenarios Money will be much lesser than 50000 i.e.

<u>Output:</u>



```
D:\1. OS\MultiThreading>g++ -std=c++11 RaceCondition.cpp -o rc -pthread

D:\1. OS\MultiThreading>rc
29473
46677
46677
46677
46677
```

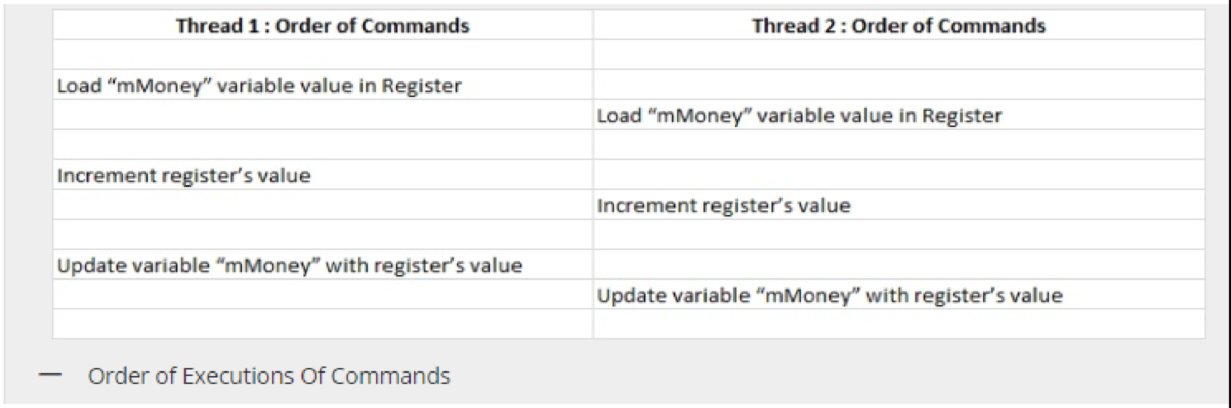
This is a race condition, as here two or more threads were trying to modify the same memory location at same time and lead to unexpected result.

Why this happened?

Each thread increments the same "Money" member variable in parallel. Although it seems a single line but this "money++" is actually converted into three machine commands,

- Load "money" variable value in Register
- Increment register's value
- Update variable "Money" with register's value

Now suppose in a special scenario, order of execution of above these commands is as follows,



Thread 1 : Order of Commands	Thread 2 : Order of Commands
Load "mMoney" variable value in Register	
	Load "mMoney" variable value in Register
Increment register's value	
	Increment register's value
Update variable "mMoney" with register's value	
	Update variable "mMoney" with register's value

— Order of Executions Of Commands

In this scenario one increment will get neglected because instead of incrementing the "Money" variable twice, different registers got incremented and the "money" variable's value was overwritten.

Mutex:

To fix race conditions in a multi-threaded environment we need mutex i.e. each thread needs to lock a mutex before modifying or reading the shared data and after modifying the data each thread should unlock the mutex.

std::mutex

In the C++11 threading library, the mutexes are in the <mutex> header file. The class representing a mutex is the std::mutex class.

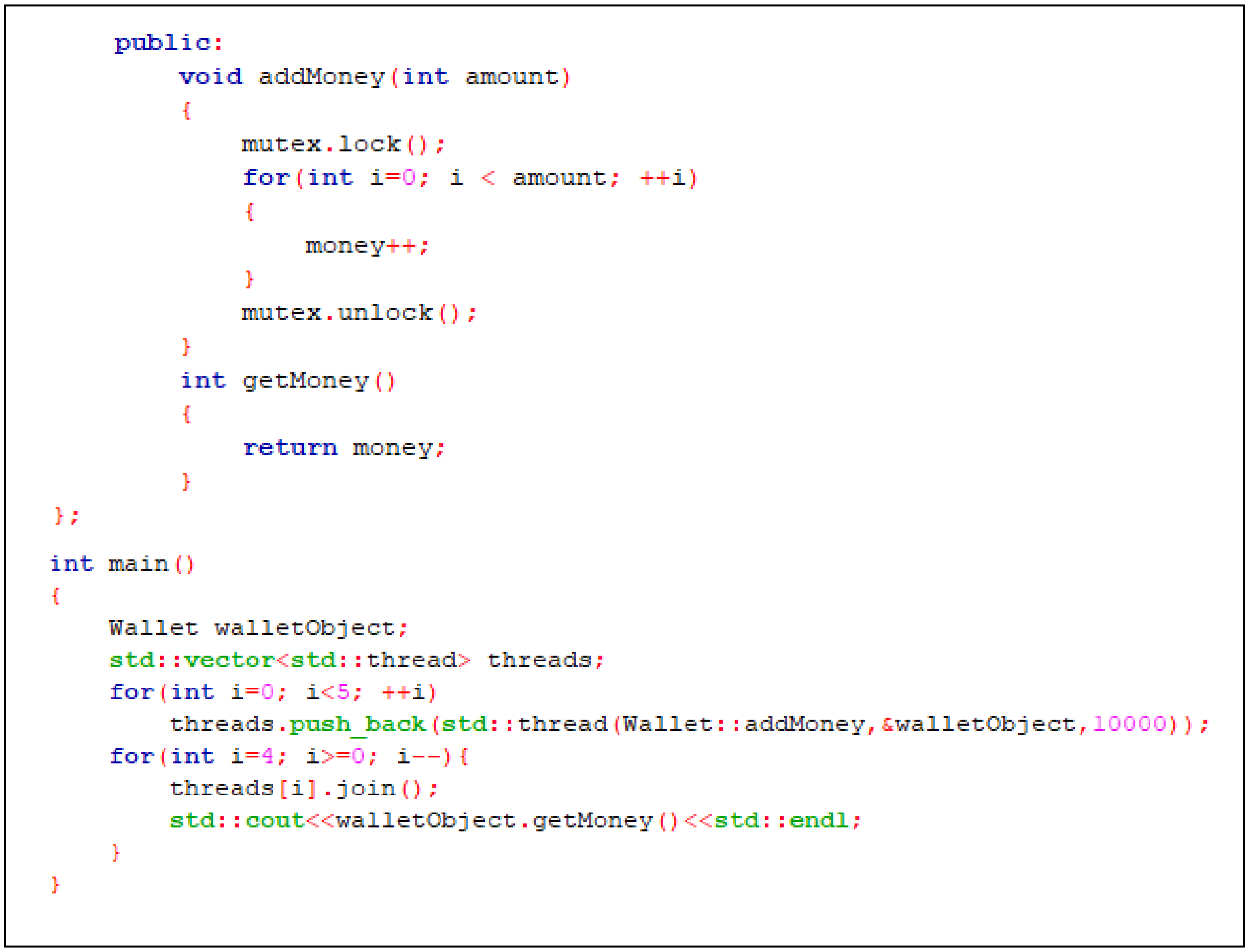
There are two important methods of mutex:

1.) lock()

2.) unlock()

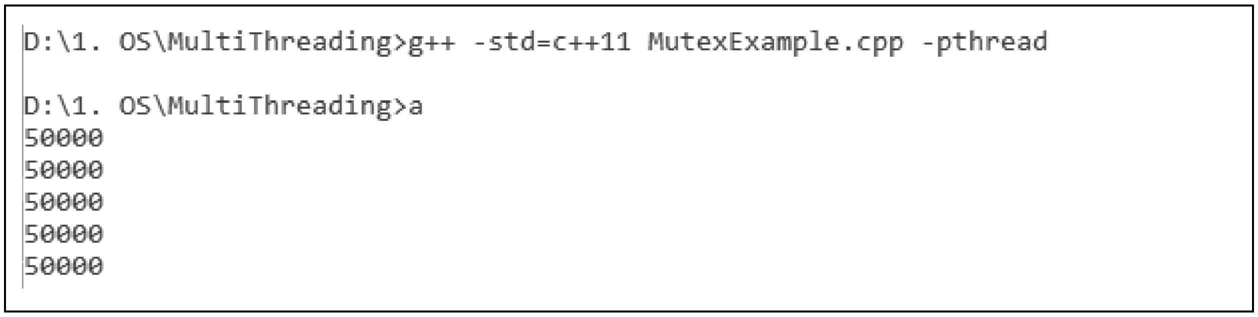
Example 02 - Mutex Example

```
#include<iostream>
#include<thread>
#include<vector>
#include<mutex>

class Wallet{

    private:
        int money=0;
        std::mutex mutex;
```



```
    public:
        void addMoney(int amount)
        {
            mutex.lock();
            for(int i=0; i < amount; ++i)
            {
                money++;
            }
            mutex.unlock();
        }
        int getMoney()
        {
            return money;
        }
};

int main()
{
    Wallet walletObject;
    std::vector<std::thread> threads;
    for(int i=0; i<5; ++i)
        threads.push_back(std::thread(Wallet::addMoney,&walletObject,10000));
    for(int i=4; i>=0; i--){
        threads[i].join();
        std::cout<<walletObject.getMoney()<<std::endl;
    }
}
```

Output:



```
D:\1. OS\MultiThreading>g++ -std=c++11 MutexExample.cpp -pthread

D:\1. OS\MultiThreading>a
50000
50000
50000
50000
50000
```

As, Wallet provides a service to add money in Wallet and same Wallet object is used between different threads, so we need to add Lock in addMoney() method of the Wallet i.e.

Acquire lock before increment the money of Wallet and release lock before leaving that function.Wallet class that internally maintains money and provides a function i.e. addMoney().

This member function, first acquires a lock then increments the internal money of the wallet object by specified count and then releases the lock.

Now Let's create 5 threads and all these threads will share the same object of class Wallet and add 10000 to internal money using its addMoney() member function in parallel.

So, if initially money in the wallet is 0. Then after completion of all thread's execution money in Wallet should be 50000.

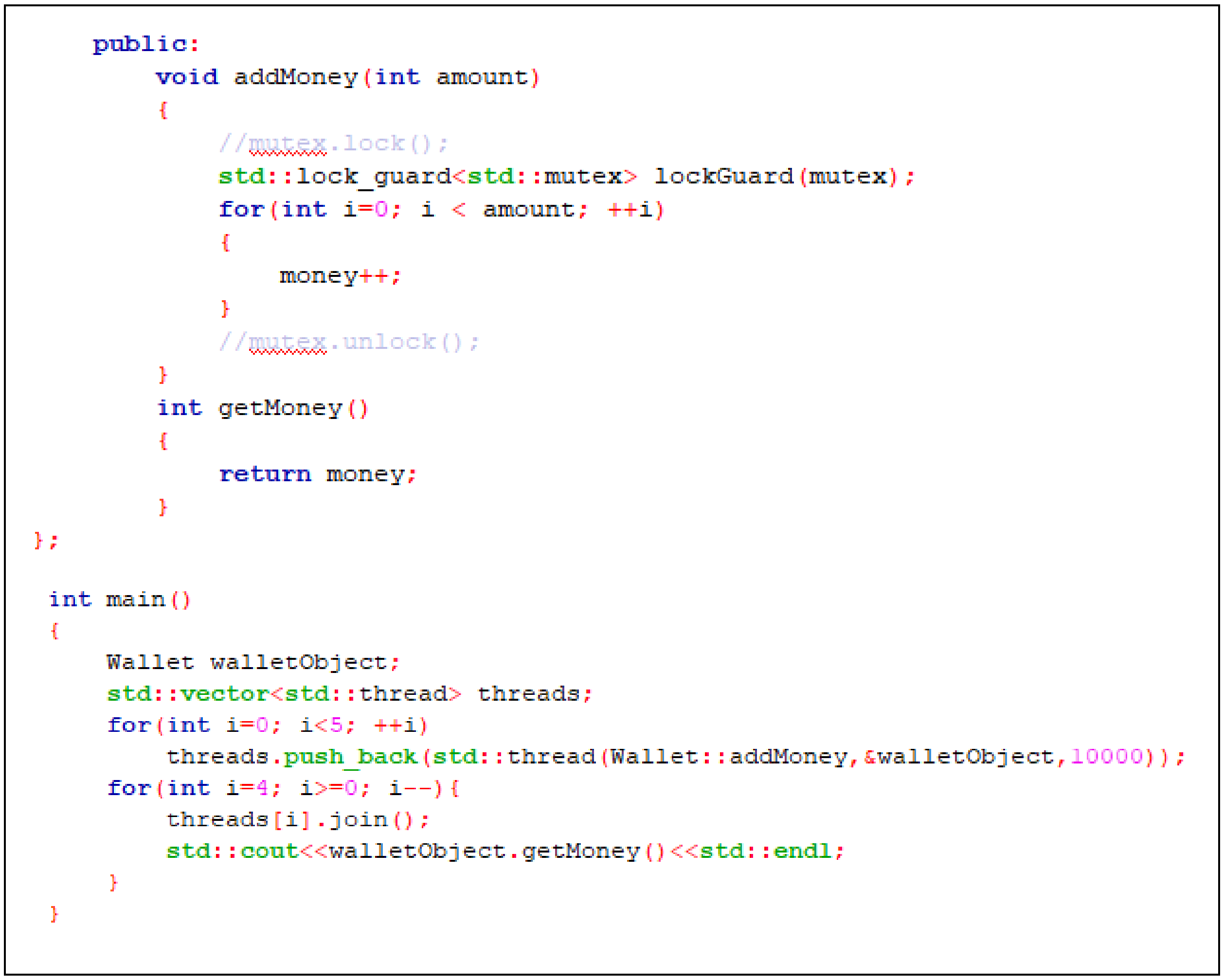
And this mutex lock guarantees that Money in the Wallet will be 50000 at the end.

std::lock_guard

std::lock_guard is a class template.It wraps the mutex inside it's object and locks the attached mutex in its constructor. When it's destructor is called it releases the mutex.

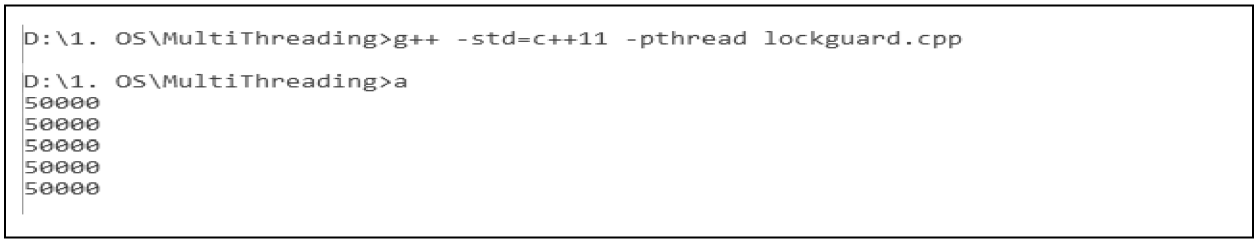
Example 03 - lock_guard Example

```
#include<iostream>
#include<thread>
#include<vector>
#include<mutex>

class Wallet{

    private:
        int money=0;
        std::mutex mutex;
```



```
    public:
        void addMoney(int amount)
        {
            //mutex.lock();
            std::lock_guard<std::mutex> lockGuard(mutex);
            for(int i=0; i < amount; ++i)
            {
                money++;
            }
            //mutex.unlock();
        }
        int getMoney()
        {
            return money;
        }
};

 int main()
 {
     Wallet walletObject;
     std::vector<std::thread> threads;
     for(int i=0; i<5; ++i)
         threads.push_back(std::thread(Wallet::addMoney,&walletObject,10000));
     for(int i=4; i>=0; i--){
         threads[i].join();
         std::cout<<walletObject.getMoney()<<std::endl;
     }
 }
```

Output:



```
D:\1. OS\MultiThreading>g++ -std=c++11 -pthread lockguard.cpp

D:\1. OS\MultiThreading>a
50000
50000
50000
50000
50000
```

DHA SUFFA UNIVERSITY
Department of Computer Science
CS-2004L
Operating Systems
Fall 2020

<u>Lab Task:</u> Submit the example given in the lab manual.

ASSIGNMENT # 10

Create global integer variable counter

- Create 4 threads and each thread: – 10000000-times increment the counter(use function pointer)
- Print the resulting value of the counter after all the threads are done.
- Resolve the race condition using Mutex.

SUBMISSION GUIDELINES

- Take a screenshot of each task(code and output).
- Place all the screenshots in a single word file labeled with Roll No and Lab No. e.g. **'cs181xxx_Lab01'.**
- Convert the file into PDF.
- Submit the file at LMS