CMPE 261 - Large Scale Programming
Lecture Notes: Week-08

Murat ORHUN

November 21, 2016

The topics of this week are about,concurrent programming, mutual exclusion,synchronization and semaphore. We will explain these topics with specific examples. So all of the students recommended to participate and try to solve the problems.

Concurrent Programming

Concurrent computing is a form of computing in which programs are designed as collections of interacting computational processes that may be executed in parallel. Concurrent programs can be executed sequentially on a single processor by interleaving the execution steps of each computational process, or executed in parallel by assigning each computational process to one of a set of processors that may be close or distributed across a network. The main challenges in designing concurrent programs are ensuring the correct sequencing of the interactions or communications between different computational processes, and coordinating access to resources that are shared among processes. A number of different methods can be used to implement concurrent programs, such as implementing each computational process as an operating system process, or implementing the computational processes as a set of threads within a single operating system process.
Java is a multi-threaded programming language that makes programming with threads easier, by providing built-in language support for threads. Actually we have implemented multi-threaded programming already. Please remember the example that we have done in the recitation which is about animation. In that example we have three threads. The main thread responsible to start the program and draw the GUI. While the other two draws moves two shapes.
Also, the last assignment also is a kind of simple concurrent application. That main thread starts the program and draw the GUI. The other two treads are generates works independently. In concurrent programming, if your threads work on independent task such as animation etc, it is very simple. Just create threads and start them. Of course you should know start and stop them. But in your application,the threads share some data, then the things not so simple. Execute the following example and try to understand it.

Example 8-1:

```
// A simple class simulate an ATM
 class ATM{

 private int balance;

 public ATM(int balance){
 this.balance=balance;
 }

 public int getBalance(){
 return balance;
 }

 public  String getMoney(int amount){

 if(balance>=amount){ // you should have enough balance

 try{
 Thread.sleep(1000);
 balance=balance-amount; // update the balance
 }
 catch(InterruptedException e){};

 return "  You able to take "+ amount + "  Please count your money!...";
 }

 else
return " The maximum money you can take is:"+balance+" !.";

 }

}

// A class that withdraw money
class Mehmet extends Thread {
ATM atm;
int amount;

public Mehmet(ATM atm,int amount) {
this.atm=atm;
this.amount=amount;


}
```

```
public void run(){


System.out.println(getName()+", "+atm.getMoney(amount));

}
}
// A class that withdraws money
// Actually you don't have to define this class. You can create two instances from
// Because both of the class do the same thing.
class Ahmet extends Thread{
ATM atm;
int amount;

public Ahmet(ATM atm,int amount) {
this.atm=atm;
this.amount=amount;


}

public void run(){

System.out.println(getName()+", "+atm.getMoney(amount));

}
}

// Main Class
public class ParaCekme {

public static void main(String[] args) {

ATM atm=new ATM(500);
int paraCek=300;


Ahmet ahmet=new Ahmet(atm,paraCek);
ahmet.setName("Ahmet");

Mehmet mehmet=new Mehmet(atm,paraCek);
mehmet.setName("Mehmet");

ahmet.start();
mehmet.start();
```

```
}

}
```

If you compile and execute the program you will get the following output:

```
Mehmet,   You able to take 300  Please count your money!...
Ahmet,   You able to take 300  Please count your money!...
```

It means,that Ahmet and Mehmet able to withdraw 300 at the same time even the balance is 500. In multi-threaded programming,we can not sure which thread finish the task first, or how long will it take to finish its task.
For this example, the **update balance** task must be finished before than other threads withdraw money.

Mutual Exclusion

Mutual exclusion (often abbreviated to mutex) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections. A critical section is a piece of code in which a process or thread accesses a common resource. The critical section by itself is not a mechanism or algorithm for mutual exclusion. A program, process, or thread can have the critical section in it without any mechanism or algorithm which implements mutual exclusion.
The critical section the last example is, accessing the **balance**. Because if you have enough **balance**, you can get money, and also the system have update once you get the money. It means, the getting and updating balance must be finished without interrupting by the other threads. While one of the thread doing these tasks, other threads have to wait for it.
In order to do this, we use the **ReentrantLock** class.

ReentrantLock

A **ReentrantLock** is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking **lock** will return, successfully acquiring the lock, when the lock is not owned by another thread.
Some of the methods that used commonly are:

- **ReentrantLock**() : Creates an instance of **ReentrantLock**.
- **lock()** : Acquires the lock.
- **tryLock()** : Acquires the lock only if it is not held by another thread at the time of invocation.

- **tryLock**(long timeout, TimeUnit unit) : Acquires the lock if it is not held by another thread within the given waiting time and the current thread has not been.
- **unlock()** : Attempts to release this lock.

For detailed information about these methods,please read the API of the **ReentrantLock** class.

Synchronization

With respect to multi-threading , synchronization is the capability to control the access of multiple threads to shared resources. Without synchronization, it is possible for one thread to modify a shared object while another thread is in the process of using or updating that object's value. This often leads to significant errors. See the example6-1.

We can implement synchronized application with ReentrantLock very easily. Lets modify the example 6-1 and try to solve the problems. **Example8-2:**

```
import java.util.concurrent.locks.ReentrantLock;


class ATM{

private int balance;

public ATM(int balance){
this.balance=balance;
}

public int getBalance(){
return balance;
}

public  String getMoney(int amount){

if(balance>=amount){ // you should have enough balance

try{
Thread.sleep(1000);
balance=balance-amount; // update the balance
}
catch(InterruptedException e){};

return "  You able to take "+ amount + "  Please count your money!...";
```

```
 }

 else
return " The maximum money you can take is:"+balance+" !.";

 }

}
class Mehmet extends Thread {
ATM atm;
int amount;
ReentrantLock kilit;
public Mehmet(ATM atm,int amount,ReentrantLock kilit) {
this.atm=atm;
this.amount=amount;
this.kilit=kilit;


}
public void run(){


kilit.lock(); //Acquires the lock.
System.out.println(getName()+", "+atm.getMoney(amount));
kilit.unlock(); //Releases the lock.

}
}
// Class for Ahmet
class Ahmet extends Thread{
ATM atm;
int amount;
ReentrantLock kilit;

public Ahmet(ATM atm,int amount,ReentrantLock kilit) {
this.atm=atm;
this.amount=amount;
this.kilit=kilit;


}

public void run(){
    kilit.lock(); //Acquires the lock.
System.out.println(getName()+", "+atm.getMoney(amount));
```

```
kilit.unlock(); //Releases the lock.
}
}

// Main Class
public class ParaCekme {

public static void main(String[] args) {

ATM atm=new ATM(500);
int paraCek=300;
ReentrantLock kilit=new ReentrantLock();


Ahmet ahmet=new Ahmet(atm,paraCek,kilit);
ahmet.setName("Ahmet");

Mehmet mehmet=new Mehmet(atm,paraCek,kilit);
mehmet.setName("Mehmet");

ahmet.start();
mehmet.start();
}

}
```

Now compile and execute this program again. You will get the following output:

```
Ahmet,   You able to take 300  Please count your money!...
Mehmet,  The maximum money you can take is:200 !.
```

It means, that Ahmet able to get 300, but Mehmet not able to get. Because the maximum money that Mehmet can get is 200.
By this way, we able to solve this problem.

Optimization

If you look these codes, we have repeated some codes. The **Ahmet** and **Mehmet** classes are actually do the same thing. So we don't need to implement two different classes. So we can optimize the last example.
Example8-3:

```
import java.util.concurrent.locks.ReentrantLock;

class ATM{
```

```
 private int balance;

 public ATM(int balance){
 this.balance=balance;
 }

 public int getBalance(){
 return balance;
 }

 public  String getMoney(int amount){

 if(balance>=amount){ // you should have enough balance

 try{
 Thread.sleep(1000);
 balance=balance-amount; // update the balance
 }
 catch(InterruptedException e){};

 return "  You able to take "+ amount + "  Please count your money!...";
 }

 else
return " The maximum money you can take is:"+balance+" !.";

 }

}
class Mehmet extends Thread {
ATM atm;
int amount;
ReentrantLock kilit;
public Mehmet(ATM atm,int amount,ReentrantLock kilit) {
this.atm=atm;
this.amount=amount;
this.kilit=kilit;
}
public void run(){

kilit.lock(); //Acquires the lock.
System.out.println(getName()+", "+atm.getMoney(amount));
kilit.unlock(); //Releases the lock.

}
```

```
}


// Main Class
public class ParaCekme {

public static void main(String[] args) {

ATM atm=new ATM(500);
int paraCek=300;
ReentrantLock kilit=new ReentrantLock();


Mehmet ahmet=new Mehmet(atm,paraCek,kilit); // First instance
ahmet.setName("Ahmet");

Mehmet mehmet=new Mehmet(atm,paraCek,kilit); // Second instance
mehmet.setName("Mehmet");

ahmet.start();
mehmet.start();
}

}
```

If you execute, you will get the following out put:

```
Ahmet,   You able to take 300  Please count your money!...
Mehmet,  The maximum money you can take is:200 !.
```

Now compare this example with example 8-2 and try to understand the difference between them.