A Demonstration of The Race Condition in a Java Application

The race condition occurs in multi-threaded or distributed systems when the timing or order of specific events can determine the outcome or behaviour of the system. In software, race conditions usually occur when multiple threads or processes have access to a certain shared state.

What we demonstrate in the application is known as a data race. It occurs when multiple threads have access to a shared memory location with one thread attempting to read from that location while another thread is writing to it. This can result in some data being discarded or even produce data that is an arbitrary combination of bits from different thread operations.

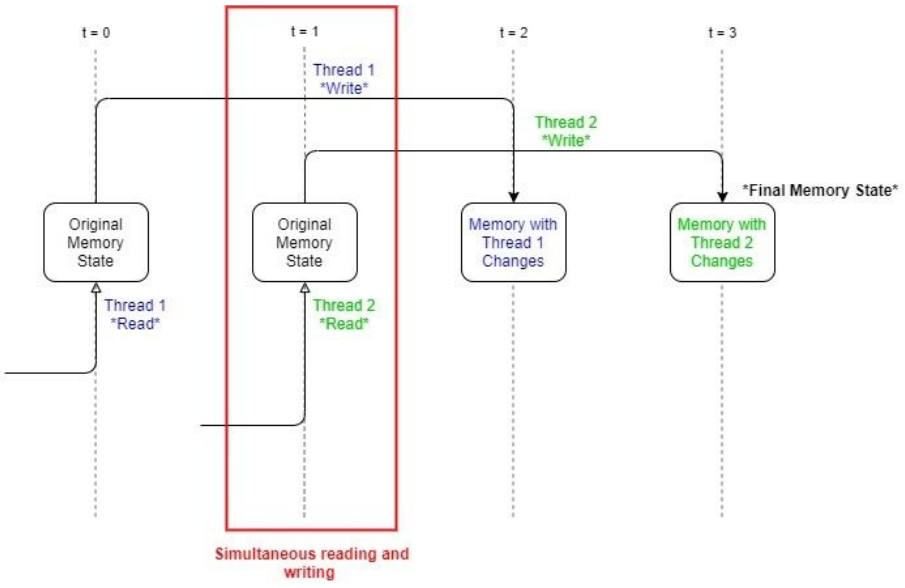

Fig.6: The changes made by thread 1 are lost in the final state due to race condition

In order to protect the shared state from corruption, the critical sections, which are the sections of code accessible by multiple threads, must be identified and made mutually exclusive. This means that only one thread or process is allowed access to the critical section at a time to prevent concurrent reading and writing on the same memory location.

The Application

In our simulation, we consider the case of a retail business that requires management of its sales across multiple outlets. An application is developed to receive regular sales reports from each outlet and update the central record. At the end of a certain period (month/quarter/year), a report must be produced and it must be up-to-date, complete and consistent.

The race condition becomes an issue when the different outlets are attempting to read from and write to the central database concurrently. If the application is not properly synchronized, some outlet sales will not be recorded and the final report will be inaccurate.

In the following section we demonstrate two versions of the application; one with, and one without, synchronization of threads.

I. In-depth Review and Discussion of the Source Code (Thread Synchronization)

Overview

The application is implemented using java with the java thread library. The full code can be found at the end of this report.

The business has three products of different prices. The “sell” and “returns” functions are defined to allow the outlets to update the sales status. Each time an item is sold or returned, the outlet updates the totalSales, productSales and totalRevenue in the central business record.

```
public Business() {
    this.totalRevenue = 0.00;
    this.totalSales = 0;
    this.productSales = new HashMap<String,Integer>();
```

Fig.7

There are six threads representing six outlets for the business “Sprint”, (“utama”, ”sunway”, ”KLCC”, ”mid valley”, ”pavilion” and ”paradigm”). By the end of the month, a report is produced from these records:

```
----- Sprint's Monthly Reports -----
Total Sales: 1800
T-shirts: 600
Hoodies: 600
Caps: 600
Total Revenue: 99000.0
```

Fig.8

In this program, the critical sections in the code are the sections allowing access from the outlet threads to the main business object. These are the “sell” and “returns” functions in the “business” class called by the threads.

```java
public void sell(String product, int quantity) {
    for (int i = 0; i < quantity; i++) {
        Double price = productPrices.get(product);
        totalRevenue += price;
        totalSales++;
        productSales.put(product,productSales.get(product)+1);
    }
}

public void returns(String product, int quantity) {
    for ( int i = 0; i < quantity; i++) {
        Double price = productPrices.get(product);
        totalRevenue -= price;
        totalSales--;
        productSales.put(product,productSales.get(product)-1);
    }
}
```

Fig.9: Critical sections

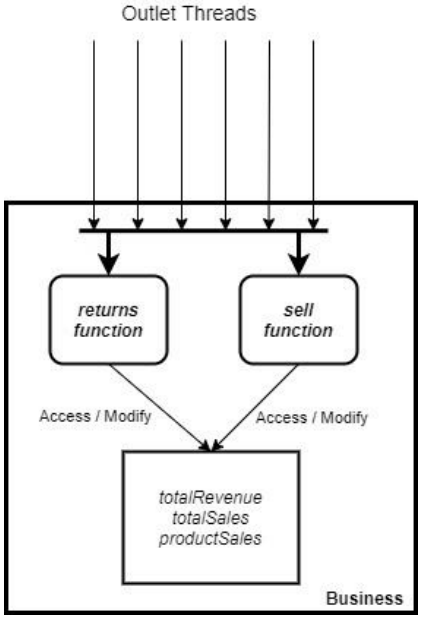

Fig.10

Case 1: No synchronization

If we assume that for some month, each of the outlets sold 2 and returned 1 of each product. The run function of the Outlet thread is called for each of the six outlets concurrently. The outlets each attempt, 10 times, to:

- Get the totalRevenue
- Increment it by the product price
- Get the totalSales
- Increment it by 1

- Get the number of sales of that particular product
- Increment it by one

The process for returns in the same but with decrements.
For small numbers of sales, the resulting report is consistent:

Fig.11

However, if inputs are increased the effect of race condition can be observed:

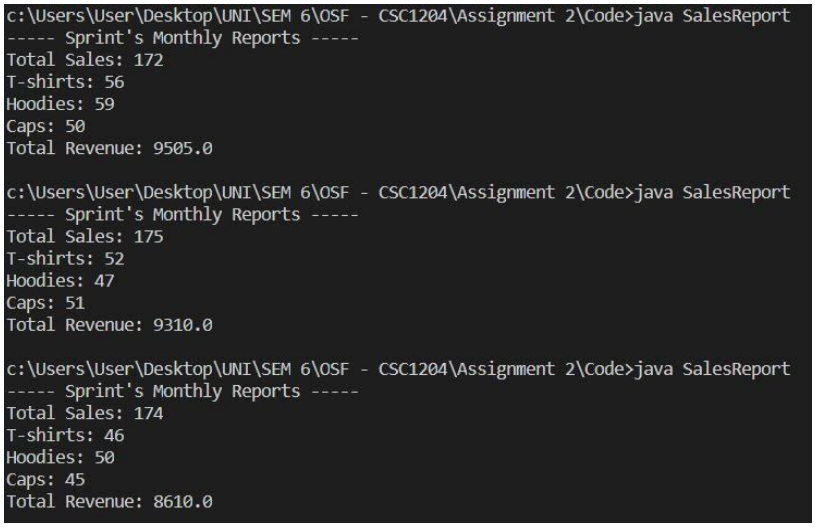

Fig.12: 10x sales & returns. Ran 3 times with same input

The numbers do not add up and are inconsistent.
<u>Expected result:</u>
Total sales : 180
T - shirts : 60
Hoodies : 60
Caps : 60
Total Revenue : 9900.0

Case 2: Synchronization

```java
public synchronized void sell(String product, int quantity) {
    for (int i = 0; i < quantity; i++) {
        Double price = productPrices.get(product);
        totalRevenue += price;
        totalSales++;
        productSales.put(product,productSales.get(product)+1);
    }
}

public synchronized void returns(String product, int quantity) {
    for ( int i = 0; i < quantity; i++) {
        Double price = productPrices.get(product);
        totalRevenue -= price;
        totalSales--;
        productSales.put(product,productSales.get(product)-1);
    }
}
```

Fig.13: 10x sales & returns. Ran 3 times with same input

After synchronization of critical sections the report becomes consistent:

```
c:\Users\User\Desktop\UNI\SEM 6\OSF - CSC1204\Assignment 2\Code>java SalesReport
----- Sprint's Monthly Reports -----
Total Sales: 180
T-shirts: 60
Hoodies: 60
Caps: 60
Total Revenue: 9900.0

c:\Users\User\Desktop\UNI\SEM 6\OSF - CSC1204\Assignment 2\Code>java SalesReport
----- Sprint's Monthly Reports -----
Total Sales: 180
T-shirts: 60
Hoodies: 60
Caps: 60
Total Revenue: 9900.0
```

Fig.14: 10x sales & returns. Ran 3 times with same input

Code for reference:

```
import java.util.*;
public class SalesReport {
    public static void main(String args[]) {
        Business Sprint = new Business();
        Outlet KlangValley = new Outlet(Sprint);

        Thread utama = new Thread(KlangValley);
        Thread sunway = new Thread(KlangValley);
        Thread KLCC = new Thread(KlangValley);
        Thread midvalley = new Thread(KlangValley);
        Thread pavilion = new Thread(KlangValley);
        Thread paradigm = new Thread(KlangValley);

        utama.start();
        sunway.start();
        KLCC.start();
        midvalley.start();
        pavilion.start();
        paradigm.start();

        while
((utama.isAlive()||sunway.isAlive()||KLCC.isAlive()||midvalley.isAli
ve()||pavilion.isAlive()||paradigm.isAlive()))    { }
        System.out.println("----- Sprint's Monthly Reports -----");
        System.out.println("Total Sales: "+Sprint.getTotalSales());
        System.out.println(Business.products[0]+":
"+Sprint.getProductSales()[0]);
        System.out.println(Business.products[1]+":
"+Sprint.getProductSales()[1]);
        System.out.println(Business.products[2]+":
"+Sprint.getProductSales()[2]);
        System.out.println("Total Revenue: "+Sprint.getRevenue());
    }
}
class Outlet implements Runnable {
```

```
    private Business business;

    public Outlet(Business business) {
        this.business = business;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            business.sell(Business.products[0],2);
            business.sell(Business.products[1],2);
            business.sell(Business.products[2],2);
            business.returns(Business.products[0],1);
            business.returns(Business.products[1],1);
            business.returns(Business.products[2],1);
        }
    }
}
class Business {
    public Random rnd = new Random();
    public static final String[] products = new
String[]{"T-shirts","Hoodies","Caps"};
    public static Map<String,Double> productPrices = new
HashMap<String,Double>();

    private double totalRevenue;
    private int totalSales;
    private Map<String,Integer> productSales;

    public Business() {
        this.totalRevenue = 0.00;
        this.totalSales = 0;
        this.productSales = new HashMap<String,Integer>();
        productSales.put(products[0],0);
        productSales.put(products[1],0);
        productSales.put(products[2],0);

        productPrices.put(products[0],50.00);
        productPrices.put(products[1],100.00);
```

```
        productPrices.put(products[2],15.00);
    }


    public synchronized void sell(String product, int quantity) {
        for (int i = 0; i < quantity; i++) {
            Double price = productPrices.get(product);
            totalRevenue += price;
            totalSales++;
            productSales.put(product,productSales.get(product)+1);
        }
    }

    public synchronized void returns(String product, int quantity) {
        for ( int i = 0; i < quantity; i++) {
            Double price = productPrices.get(product);
            totalRevenue -= price;
            totalSales--;
            productSales.put(product,productSales.get(product)-1);
        }
    }

    public double getRevenue() {
        return this.totalRevenue;
    }
    public int[] getProductSales() {
        return new
int[]{productSales.get(products[0]),productSales.get(products[1]),pr
oductSales.get(products[2])};
    }
    public int getTotalSales() {
        return this.totalSales;
    }
}
```