INDIVIDUAL ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT074-3-2-CCP

CONCURRENT PROGRAMMING

UC2F1908CS(DA) /UC2F1908CS

HAND OUT DATE: 24 FEBRUARY 2020

HAND IN DATE: 24 APRIL 2020

WEIGHTAGE: 20%

INSTRUCTIONS TO CANDIDATES:

1 Submit your assignment at the administrative counter

2 Students are advised to underpin their answers with the use of references (cited using the Harvard Name System of Referencing)

3 Late submission will be awarded zero (0) unless Extenuating Circumstances (EC) are upheld

4 Cases of plagiarism will be penalized

5 The assignment should be bound in an appropriate style (comb bound or stapled).

6 Where the assignment should be submitted in both hardcopy and softcopy, the softcopy of the written assignment and source code (where appropriate) should be on a CD in an envelope / CD cover and attached to the hardcopy.

7 You must obtain 50% overall to pass this module.

Version 1.3 UCTI ExCv 2009-12-02

INDIVIDUAL ASSIGNMENT

(PART 1)

CT074-3-2-CCP

CONCURRENT PROGRAMMING

UC2F1908 CS(DA)

Name: Chan Jia Le (TP049952)

Proposal Title: Concurrent Programming

Hand Out Date: 23 FEBRUARY 2020

Hand in Date: 24 APRIL2020

Table of Contents

1.0 | Introduction

1.1 | Assumption

The owner, waiter, customer, serving area and clock will be constructed as the main threads. The café only has one table with 10 seats. Once all the seats are occupied, the customer will wait outside of the restaurant, the customers that queue will stay on the queue based on random time and will leave once the waiting time exceeds their tolerable waiting time. The simulation will state the customer unfortunately left because of waiting too long. Once entered, customers will randomly order a drink, the owner and waiter will then make the order in a first come first come basis, but if one is using the juice tap or the cupboard in the serving area, the other will wait until the other had achieve what they need to complete the drink. Once the drink is served, the customers will take some random time to consume the drinks and will have a small chance of ordering another one. After "last call" is called, the waiter will finish his current order and stop, at this time no customers can enter or queue anymore. Thus, the owner will be responsible for serving all the remaining customers even if after closing time. It is assumed that the user will provide the details of the closing time, how many customers to serve, the speed of completing drinks and it is assumed that the ingredient is enough to serve the customers for the day.

2.0 | Design Decision

2.1 | Thread

```
public class Waiter implements Runnable {
    ServingArea sv;
    String name;
    public Waiter(String name, ServingArea s)
    {
        this.name = name;
        sv=s;
    }

    public void run()
    {


    }
}
```

Figure 2.1.1 Thread initialization

```
public class MainClass {
    public static void main(String ar[])
    {
        Waiter w1 = new Waiter("John");
        Thread w2 = new Thread(w1);
        w2.start();

    }
}
```

Figure 2.1.2 Thread generation and beginning task

Thread allows multiple operations simultaneously. Thus, stated above, the owner, waiter, customer and clock will be planned to be implemented as a thread. Figure 2.1.1 and 2.1.2 shows the example of creation of threads, inside run() will be the what action will the thread perform. Once it is generated, it will use start() to begin running the thread. The reason of implementing runnable is because it allows for flexibility of inheritance in the future. Furthermore, the owner, waiter, customers and clock as a thread is because they all perform different actions and to simulate those different actions, they should be a thread.

2.2 | Time Simulation

```
System.out.println(name + " : is performing operation " + i );
Thread.sleep(1000);
```

Figure 2.2.1 Thread sleep()

When owner and waiter is taking advantage of the fruit tap juice or the cupboard or customers are drinking his juice or coffee. Under any circumstances where time is consumed to finish a task, the sleep() function will be implemented to simulate the time taken to finish a task as it pauses the thread for a certain amount of time as specified by the programmer or user.

As sleep function uses milliseconds to specify the sleep time, all the requested input for the user will be in seconds, the system will have a function to convert it into the milliseconds.

2.3 | Semaphore

```
public class Semaphore {
    private int availableSeat;
    public Semaphore (int seatNum){
        availableSeat = seatNum;
    }

    synchronized public void up(){
        ++availableSeat;
        notify();
    }

    synchronized public void down() throws InterruptedException{
        while (availableSeat == 0)wait();
        -- availableSeat;
    }
}
```

Figure 2.3.1 Semaphore details

Available seat in the café will be 10 seats, once the customers (threads) had fully occupied the space, which means available seat = 0, thus customer (thread) wishes to use down() which is sitting down. The semaphore will then use wait() function to call the thread to wait until the other customer (thread) is done, once the other customer (thread) is done it will then call the up() function as shown in the figure above to tell the system there is available seat and then notify() the waiting thread. The thread will then proceed to decrement the available seat once it is notified. Semaphore is implemented in order to simulate that when seat is full, customers should wait, and then be served in a first come first served basis.

2.4 | Atomic Variable

```
private final AtomicInteger juiceServed = new AtomicInteger(0);
```

Figure 2.4.1 Atomic Integer initialization

The statistics that will be printed in the end will be how much customers, coffee, juice is served. The variables that is used to contain the results are planned to be set as atomic values, because if many threads are accessing the variable it could cause erroneous results.

3.0 | Safety Aspect Implemented

As mentioned in the assumption, the owner and the waiter will share only one serving area, while two threads are sharing the same resources, the system will be planned to include synchronization function in order to lock the resources that the two threads are sharing while one is using, and releasing it once the thread is done. Thus, atomic value will be implemented as well, atomic value will be implemented to avoid interference of from other functions. Both implementations are used in order to avoid undesired values or results.

3.1 | Atomic Operation

```
public class Counter {
    private final AtomicInteger juiceServed = new AtomicInteger(0);

    public int getValue() {
        return juiceServed.get();
    }
    public void increment() {
        while(true) {
            int existingValue = getValue();
            int newValue = existingValue + 1;
            if(juiceServed.compareAndSet(existingValue, newValue)) {
                return;
            }
        }
    }
}
```

Figure 3.1.1 Juice Served Counter

Atomic operations are specified as an action that will be performed as a single work without any other interruption made by other operations (Vogella, 2016). The operation is implemented to avoid any erroneous actions caused by the interruption of multiple threads. The atomic variable will first retrieve the value to a variable as shown in figure 3.1.1 as existingValue, having a new value being the increment of the existing value, once it is incremented return the value, if multiple threads are trying to update the counter at the same time, the one that performs first will update the value, and then the others after. This allows for multiple threads to have access the same counter and perform the counter update without fault. This will be implemented to count the customer served in the simulation to avoid the interference of multiple customer (threads) incrementing the value at the same time causing it to lose count.

3.2 | Synchronization

```
public class Cafe {


   synchronized void visit(Customer use)
    {

        System.out.println("Customer "+use.id+ " Visit Cafe ");
        try{
                Thread.sleep(new Random().nextInt(5)*1000);
            }catch(Exception e){}
        System.out.println(use.id+ " left ");

    }


}
```

Figure 3.2.1 Synchronized function visit

```
public class MainClass {
    public static void main(String ar[])
    {
        Cafe s=new Cafe();
        Customer u1 =new Customer(1,s);
        Customer u2 =new Customer(2,s);
        Customer u3 =new Customer(3,s);
        Customer u4 =new Customer(4,s);

        u1.start();u2.start();u3.start();u4.start();


    }
}
```

Figure 3.2.2 Main function

```
Customer 1 Visit Cafe
1 left
Customer 2 Visit Cafe
2 left
Customer 3 Visit Cafe
3 left
Customer 4 Visit Cafe
4 left
```

Figure 3.2.3 Sample Output

Synchronization is a concurrent programming concept that allows one thread to access the resource and lock it, thus releasing the resources when it is done (Java Revisited, 2017). This implementation will avoid any unwanted duplicated execution of operations made by the multiple threads. Figure 3.2.1 shows a café that only allows one customer to visit at one time. Furthermore, when visiting the customer will take a random time to have a drink. In order to only allow one customer to enter and have a drink, thus allowing the next customer to enter when the last customer is done, synchronized function is called. This is planned to be implemented for owner and waiter when using the only serving area in the simulation, where waiter will lock the resources (juice tap or cupboard) and if owner wishes to use the shared resources, he will need to wait.

4.0 | References

[1] Geeks for Geeks, 2020. *Multithreading in Java.* [Online]
Available at: https://www.geeksforgeeks.org/multithreading-in-java/
[Accessed 21 April 2020].

[2] Java Revisited, 2017. *Java Synchronization Tutorial : What, How and Why?.* [Online]
Available at: https://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html
[Accessed 20 April 2020].

[3] Vogella, 2016. *Java concurrency (multi-threading) - Tutorial.* [Online]
Available at: https://www.vogella.com/tutorials/JavaConcurrency/article.html
[Accessed 18 April 2020].