# CS6235: Analysis of Parallel Programs

Parallelism, Parallel constructs and Java Concurrecy

Soham Tripathy (CS20B073)

08-02-2023

# Assignment 1 : CS6235 Analysis of Parallel Programs

## Question 1

Write Java code snippets to show (i) data-races, (ii) how atomicity violations can exist despite race freedom (using **synchronized** blocks or methods).

### Answer

The code of the class of the shared data is given below. The class has a function which can change the value of **stock_price** and a function to get the value of **stock_price**.

```java
class CompanyStocks {
    Double stock_price = 1000.0;
    Boolean move = false;

    public Double get_price() {
        return stock_price;
    }

    public void price_change() {
        Double fluc = Math.random();
        Double change = 100*fluc;
        if(stock_price >= 1250) move = false;
        else if(stock_price <= 750) move = true;
        if(!move) change = -change;
        stock_price = stock_price+change;
    }
}
```

The code for the Market Thread is given below, it calls the **price_change** function in the stock, in an indefinite loop. Observe that the **stock_price** is being changed by this thread.

```java
class Market implements Runnable {
    CompanyStocks stock;

    Market(CompanyStocks s) {
        this.stock = s;
    }

    public void run() {
        while(true) {
            stock.price_change();
        }
    }
}
```

The code for the trader is given below, it has it's alogrithm which signals to have long or short position. This algorithm depends on the stock_price, which is being changed by the market thread continuouesly. Hence, leading to a Data-Race Condition.

```java
class Trader implements Runnable {

    CompanyStocks stock;
    char type;
    Double ulimit;
    Double dlimit;

    Trader(CompanyStocks s, Double l, Double u) {
        this.stock = s;
        this.dlimit = l;
        this.ulimit = u;
    }

    public void run() {
        while(true) {
            Double p = stock.get_price();
            if(p < dlimit) {
                System.out.println("Take Long Position; Indicated at " + p + "; Asset bought at "+stock.get_price());
                System.out.println("Clear Short Position; Indicated at " + p + "; Asset bought at " + stock.get_price());
            }
            else if(p > ulimit) {
                System.out.println("Clear Long Position; Indicated at " + p + "; Asset sold at "+ stock.get_price());
                System.out.println("Take Short Position; Indicated at " + p + "; Asset sold at "+stock.get_price());
            }
        }
    }
}
```

The main code is given below.

```java
public class DataRace {
    public static void main(String[] args) throws InterruptedException {
        CompanyStocks stock = new CompanyStocks();
        Thread market = new Thread(new Market(stock));
        Thread trader = new Thread(new Trader(stock, 900.0, 1100.0));
        market.start();
        trader.start();
        market.join();
        trader.join();
    }
}
```

For the second part, we change all the functions that access the shared variable to synchronized methods.

> Note: It is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object. Thus when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object.

```java
class CompanyStocks {
    Double stock_price = 1000.0;
    Boolean move = false;

    public synchronized Double get_price() {
        return stock_price;
    }

    public synchronized void price_change() {
        Double fluc = Math.random();
        Double change = 100*fluc;

        if(stock_price >= 1250) move = false;
        else if(stock_price <= 750) move = true;
        if(!move) change = -change;
        stock_price = stock_price+change;
    }
}
```

This eliminates all the Data-Races, thereby achieving Race freedom, but atomicity is violated. As a trader, we want to take a position on assest at the price we compute the alogoritm. But our indicated price and price at which we take the position may change. The *get_price* on **line 16** and **line 18** may be different. Hence, this violates atomicity even though we have reached race freedom. Below is the output for the second part indicating atomic violations.

```
Take Long Position; Indicated at 810.6214231372069; Asset bought at 1262.8217980131476
Clear Short Position; Indicated at 810.6214231372069; Asset bought at 1266.1203094028274
Take Long Position; Indicated at 886.2649073190403; Asset bought at 1048.2705512332382
Clear Short Position; Indicated at 886.2649073190403; Asset bought at 1195.0703784001441
Clear Long Position; Indicated at 1210.9796240546195; Asset sold at 919.7476947323403
Take Short Position; Indicated at 1210.9796240546195; Asset sold at 1159.7135823226354
```

## Question 2

Write a Java program and use it to illustrate Amdahl's law. Show the execution time numbers to empirically establish the law. You may use any multi-core system (with at least 8 cores) for the experiment.

### Answer

> **Amdahl's Law:** The overall performance improvement gained by parallel processing is limited by the fraction of the computation that must be done sequentially.

This means that for a given program having fractions of both serial and parallel code, if we keep increasing the number of cores on which the program runs, we will reach an upper limit on the speedup.

The following code contains various instances of a time consuming iterative addition procedure. These are arranged in such a way that one instance always runs before any other. The rest of them can run simultaneously, given that we provide enough cores for computation.

The code prints the time taken for executing these various instances.

```java
class Amdahl {
    public static void main(String[] args) throws InterruptedException{
        long startTime;
        long endTime;
        long duration;
        startTime = System.nanoTime();
        //time consuming iterative addtion
        long k = 0;
        while(k < 1e10) k++;

        Thread[] t = new Thread[8];
        //Parallelizable Part
        for(int i = 0; i<8; i++) t[i] = new Thread(new HeavyWork());
        for(int i = 0; i<8; i++) t[i].start();
        for(int i = 0; i<8; i++) t[i].join();

        endTime = System.nanoTime();
        duration = endTime-startTime;
        System.out.println((double) duration/1000000000);
    }
}

class HeavyWork implements Runnable {
    public void run(){
        //time consuming iterative addtion
        long k = 0;
        while(k < 1e10) k++;
    }
}
```

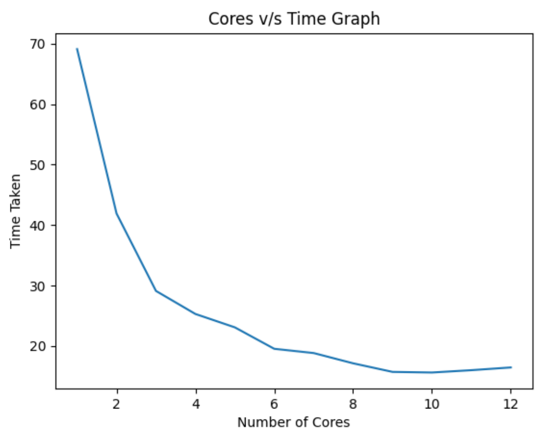
After compiling the .java file using javac, I run the python script given below.

```python
import subprocess
time = 0.0
a = input("Enter cores to use: ")
for i in range(0, 5):
    getTime = subprocess.Popen('taskset --cpu-list '+a+' java Amdahl',
       shell=True, stdout=subprocess.PIPE).stdout
    t = getTime.read()
    time = time + float(t.decode())

time/=5;
print(time)
```

This script uses the `taskset` command to set the cpu affinity (number of cores) on which the program runs. The `--cpu-list` flag is used to specify the cores to use.

```
python3 run.py
Enter cores to use: 0
69.0991381052

python3 run.py
Enter cores to use: 0-1
41.9201990946
```

| Number of Cores | Time taken(s) |
|---|---|
| 1 | 69.099138105 |
| 2 | 41.920199094 |
| 3 | 29.119227384 |
| 4 | 25.310652128 |
| 5 | 23.096165720 |
| 6 | 19.552013724 |
| 7 | 18.846306764 |
| 8 | 17.153923644 |
| 9 | 15.734864226 |
| 10 | 15.633210785 |
| 11 | 16.017235148 |
| 12 | 16.468612959 |



From the graph we can clearly see the decrease in time taken to run the program, as we increase the number of cores. The time taken almost caps at 15.7 approximately, and infact starts to increase a little as we force the program to run on cores more than a certain amount.

## Question 3

Write a Java program to prove that Java threads share the heap.

### Answer

When we create an object of class in the main function it is stored in the heap and a reference to the heap is stored in the main stack.

Here we try to run two threads, both of which read the default initialized value of the data object of class MyData. Then, one thread makes changes to the data object, which is visible to the other thread when it reads the value from the same object. This proves that the data object is visible to both the threads, thereby proving that the heap space is shared between threads

```java
import java.util.concurrent.CyclicBarrier;

public class ThreadSharingHeap {
    public static void main(String[] args) throws InterruptedException
        {
        MyData data = new MyData(0);
        CyclicBarrier barrier = new CyclicBarrier(2);

        Thread t1 = new Thread(new Runnable() {
            public void run() {
                System.out.println("Data Value is " + data.get_val() +
                    " from Thread 1.");
                try {
                    barrier.await();
                } catch (Exception e) {
                    System.out.println(e);
                }
                synchronized (data) {
                    data.set_val(10);
                    System.out.println("Data Value changed to " + data.get_val() + " from Thread 1.");
                    data.notify();
                }
            }
        });

        Thread t2 = new Thread(new Runnable() {
            public void run() {
                System.out.println("Data Value is " + data.get_val() + " from Thread 2.");
                try {
                    barrier.await();
                } catch (Exception e) {
                    System.out.println(e);
                }
                synchronized (data) {
                    try {
                        while (data.get_val() == 0)
                            data.wait();
                    } catch (Exception e) {
                        System.out.println(e);
                    }
                    System.out.println("Data Value is " + data.get_val() + " from Thread 2.");
                }
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();
    }
}

class MyData {
    private int val_data;

    MyData(int a) {
        val_data = a;
    }

    void set_val(int a) {
        val_data = a;
    }

    int get_val() {
        return val_data;
    }
}
```

Given, below shows the output of the program when run.

```
Data Value is 0 from Thread 1.
Data Value is 0 from Thread 2.
Data Value changed to 10 from Thread 1.
Data Value is 10 from Thread 2.
```

## Question 4

For the code written in Q1(i) and Q1(ii): show the static Happens Before (*HB*) relation between the different Java statements. You would need to add a line number to each line in the Java code to illustrate the *HB* relation

### Answer

I have provided the code for part (i) and part(ii) above in **Question 1**. Hereby, I will refer to the line number 1 in main code as M1, in trader code as T1, and in market code as Mk1.

Let me first establish intra-thread *HB* relations.

- Main Thread
  - Since there are no loops in the main code, hence each line *HB* the next line.
  - M7 *HB* M8, M8 *HB* M9, and such more.
  - M3 *HB* M4, M4 *HB* M5.
  - From the above relation, we can say all statements in the constructor of *CompanyStocks* happens before all statements in the constructor of Market. Similarly, statements in Market *HB* statements in Trader.
  - By transitivity, we can establish many more such relations. Such as M7 *HB* M9, Mk5 *HB* T9, and more.
- Market Thread
  - We only have one statement in market thread.
- Trader Thread

- In the if condition, T17 *HB* T18, T18 *HB* T19, T17 *HB* T19.
- In the else-if condition, T21 *HB* T22, T22 *HB* T23, T21 *HB* T23.
- T16 *HB* T17, T17 *HB* T21.
- Since, they are in a loop. T21 *HB* T16, T17 *HB* T16, T23 *HB* T16, T19 *HB* T16.
- From transitive relations (if a *HB* b, and b *HB* c then a *HB* c), we can establish more such *HB* relations.

> Note: Intra-Thread *HB* relations remain same in both part (i) and (ii) of the first question.

For Inter-Thread *HB* relations.

Part (i)

- We cannot establish any *HB* relation between Trader and Market Thread, because they execute concurrently.
- All statements in main thread before line 6, happen before Trader and Market threads.

Part (ii)

- All statements in main thread before line 6, happen before Trader and Market threads.
- Using the properties of synchronized methods, we can say, Mk10 *HB* T16 and T16 *HB* Mk10, T19 *HB* Mk10, T23 *HB* Mk10.
- By the above *HB* relations we can using transitivity to establish various other relations as well.

### Question 6

Write a Java program (using CyclicBarriers) that implements a single consumer, single producer scenario. Assume that the producer produces an item, waits for the consumer to consume. Similarly, the consumer thread waits for an item to be available, and then consumes it. The consumer and producer threads repeat the same tasks in an infinite loop.

#### Answer

```java
import java.util.concurrent.CyclicBarrier;

public class PandC {
    public static void main(String[] args) throws InterruptedException
        {

        //item to produce and consume
        Items item = new Items();

        //Producer Barrier
        CyclicBarrier Pbarrier = new CyclicBarrier(2, new Runnable() {
             public void run() {
                System.out.println("Producer has produced the item.");
             }
        });

        //Consumer Barrier
        CyclicBarrier Cbarrier = new CyclicBarrier(2, new Runnable() {
            public void run() {
                System.out.println("Consumer has consumed the item.");
            }
        });

        //producer thread
        Thread producer = new Thread(new Runnable() {
            public void run() {
                //initial production
                item.quantity = item.quantity+1;
                System.out.println("Item Quanitity: " + item.quantity);
                //signals production
                try {
                    Pbarrier.await();
                } catch (Exception e) {
                    System.out.println(e);
                }

                while(true /*indefinite execution*/) {
                    //waits for consumption of item
                    try {
                        Cbarrier.await();
                    } catch (Exception e) {
                        System.out.println(e);
                    }
                    item.quantity = item.quantity+1;
                    System.out.println("Item Quanitity: " + item.
                       quantity);
                    //signals production
                    try {
                        Pbarrier.await();
                    } catch (Exception e) {
                        System.out.println(e);
                    }
                }
            }
        });

        Thread consumer = new Thread(new Runnable() {
            public void run() {
                while(true/*indefinite execution*/) {
                    //waits for producer to produce item
```

```
                        try {
                            Pbarrier.await();
                        } catch (Exception e) {
                            System.out.println(e);
                        }
                        item.quantity = item.quantity-1;
                        System.out.println("Item Quanitity: " + item.
                            quantity);
                        //signals production
                        try {
                            Cbarrier.await();
                        } catch (Exception e) {
                            System.out.println(e);
                        }
                    }
                }
            });

            producer.start();
            consumer.start();

            producer.join();
            consumer.join();
    }
}

class Items {
    int quantity = 0;
}
```

In the code we use to cyclic barriers waiting for two threads (one producer and one consumer). One barrier is used for signaling and waiting for production and the other is used for the signalling and waiting of consumption.

In the **producer** thread we produce the item and reach the *Pbarrier* (this is consider as signal of production). Meanwhile in the **consumer** thread, we are already waiting at the *Pbarrier* (this is considered waiting for production). Similarly, in the **producer** thread, we have reached the *Cbarrier* (this is considered as waiting for consumption). This is reached by the **consumer** thread, after consumption (considered as signal of consumption).

The whole process is in an indefinite while loop, and continues till stopped. The ouput is given bellow.

```
Item Quanitity: 1
Producer has produced the item.
Item Quanitity: 0
Consumer has consumed the item.
Item Quanitity: 1
Producer has produced the item.
Item Quanitity: 0
```

```
Consumer has consumed the item.
Item Quanitity: 1
Producer has produced the item.
```

## Question 5

Write a Java program that leads to a deadlock due to parallelism related constructs: using threads and cyclic-barriers.

### Answer

The code given for the producer and consumer problem is an classic example for deadlock due to parallelism related constructs (using cyclic-barriers), if we remove the *initial production* part in the **producer** thread, ie. by removing lines from 27 to 35.

The run method for the producer thread looks like the following.

```java
public void run() {
    while(true /*indefinite execution*/) {
        //waits for consumption of item
        try {
            Cbarrier.await();
        } catch (Exception e) {
            System.out.println(e);
        }
        item.quantity = item.quantity+1;
        System.out.println("Item Quanitity: " + item.quantity);
        //signals production
        try {
            Pbarrier.await();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

In this version of the code, consumer thread waits at the *Pbarrier* for the producer to produce the item, while the producer thread waits at the *Cbarrier* for the consumer to consume the item. This is a deadlock condition, because the producer waiting for consumer while consumer is waiting for the producer.