Written Assignment 1

Vimarsh Sathia

February 22, 2021

1 Non-atomicity with and without data races

1.1 Data Race Snippet

Listing 1 has a data race in the access of variables reward and num_workers in their respective get and set methods. This is also observed in the output screenshot in Figure 1.

```java
public class datarace implements Runnable {

    // Access to variables is not data race-free
    // In addition, the program will have race-conditions
    // during runtime
    int reward;
    int num_workers;

    int get_reward(){return this.reward;}
    int get_workers(){return this.num_workers;}

    void set_reward(int v){this.reward = v;}
    void set_workers(int v){this.num_workers = v;}

    void addPushups(int reps){
        int t1 = get_reward();
        set_reward(t1 + reps);
        int t2 = get_workers();
        set_workers(t2 + 1);
    }

    void cheatDay(){
        int t1 = get_reward();
        set_reward(t1/2);
        int t2 = get_workers();
        set_workers(t2-1);
    }

    @Override
    public void run(){
        try{Thread.sleep(0,5);}catch(InterruptedException e){}
        addPushups(50);
    }

    void startExecution(){
        Thread t = new Thread(this);
        try {
            t.start();
            Thread.sleep(0,10);
            this.cheatDay();
            System.out.println("Middle: [" + this.get_workers() + "," + this.get_reward()+"]");
            t.join();
        } catch (InterruptedException e) { e.printStackTrace();}
    }

    public static void main(String args[]){
        datarace o = new datarace();
        System.out.println("Initial: [" + o.get_workers() + "," + o.get_reward()+"]");
        o.startExecution();
```

```
        System.out.println("Final: [" + o.get_workers() + "," + o.get_reward()+"]");
    }
}
```

Listing 1: Code of datarace.java (cheatDay() and addPushups() are called in different threads)

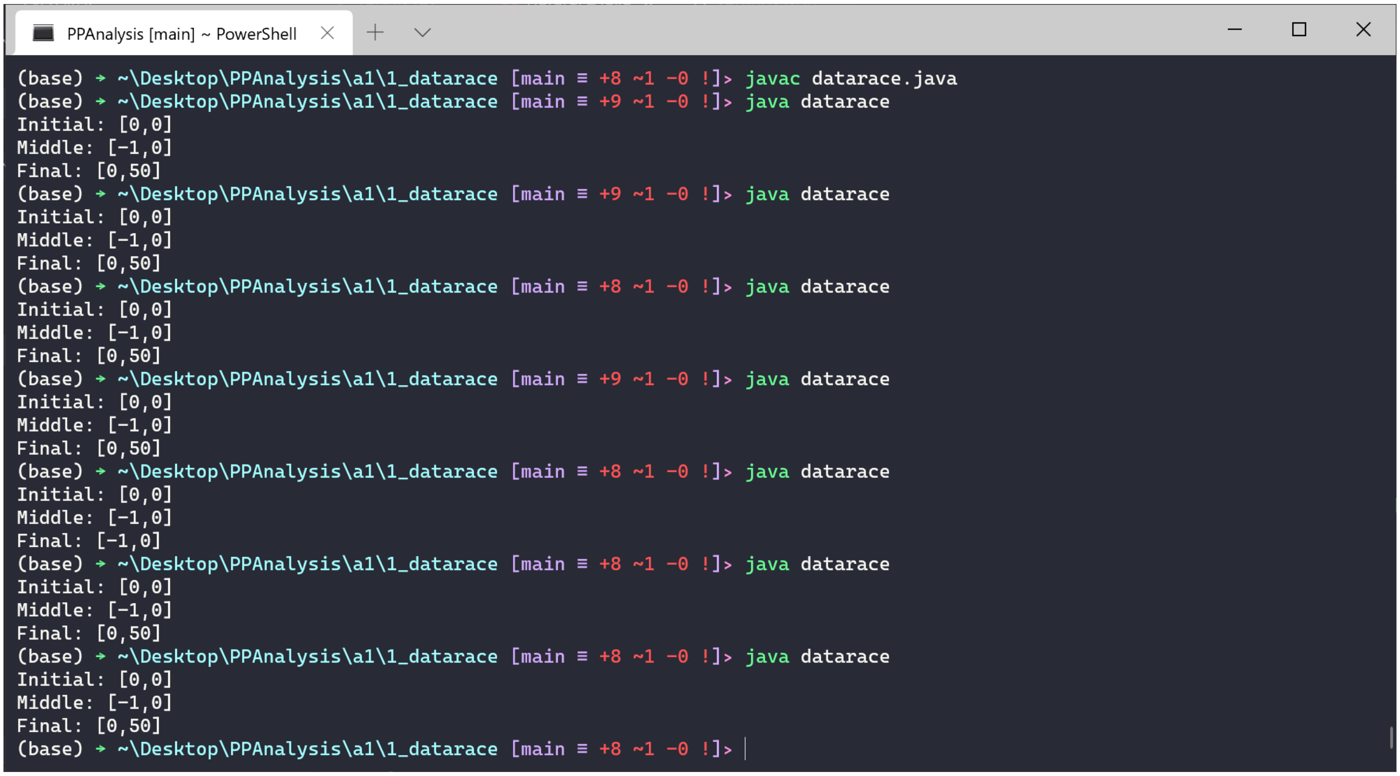

Figure 1: Screenshot of output showing inconsistent results between runs.

1.2 Data Race-free snippet

In Listing 2, the variables reward and num_workers are accessed exclusively in a synchronized method. Hence there are no data-races between threads t and main.
However even in this case, atomicity of addPushups() and cheatDay() is not gauranteed, since reads and writes may be interleaved between both the running threads.
This behaviour can be observed in Figure 2.

```
public class dataracefree implements Runnable {

    // Access to variables is now data race-free
    // However absolute atomicity is not gauranteed in methods
    // Race conditions can still occur at runtime.
    int reward;
    int num_workers;

    synchronized int get_reward(){return this.reward;}
    synchronized int get_workers(){return this.num_workers;}

    synchronized void set_reward(int v){this.reward = v;}
    synchronized void set_workers(int v){this.num_workers = v;}

    void addPushups(int reps){
        int t1 = get_reward();
        set_reward(t1 + reps);
        int t2 = get_workers();
        set_workers(t2 + 1);
    }
```

```java

    void cheatDay(){
        int t1 = get_reward();
        set_reward(t1/2);
        int t2 = get_workers();
        set_workers(t2-1);
    }

    @Override
    public void run(){
        try{Thread.sleep(0,5);}catch(InterruptedException e){}
        addPushups(50);
    }

    void startExecution(){
        Thread t = new Thread(this);
        try {
            t.start();
            Thread.sleep(0,10);
            this.cheatDay();
            System.out.println("Middle: [" + this.get_workers() + "," + this.get_reward()+"]");
            t.join();
        } catch (InterruptedException e) { e.printStackTrace();}
    }

    public static void main(String args[]){
        datarace o = new datarace();
        System.out.println("Initial: [" + o.get_workers() + "," + o.get_reward()+"]");
        o.startExecution();
        System.out.println("Final: [" + o.get_workers() + "," + o.get_reward()+"]");
    }
}
```

Listing 2: Code of dataracefree.java (cheatDay() and addPushups() are again called in different threads)

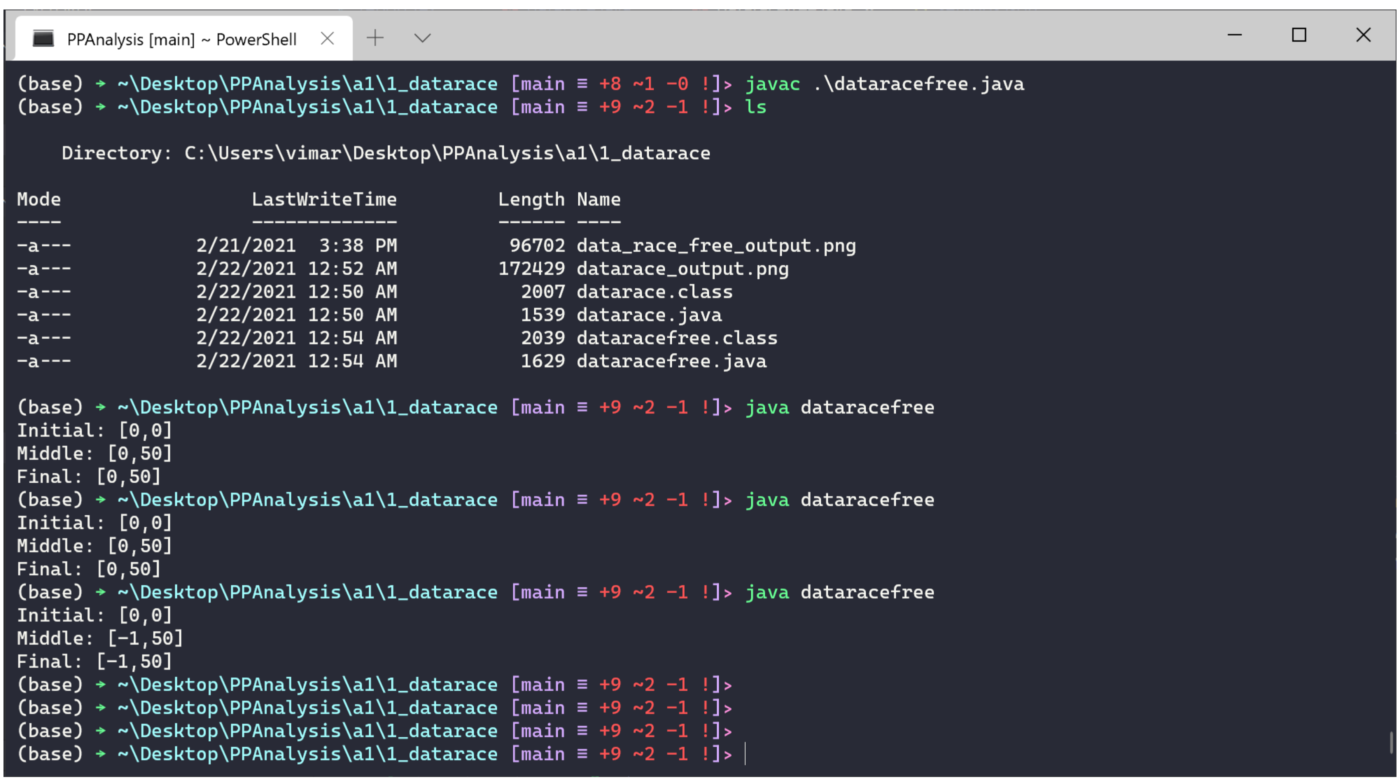

Figure 2: Screenshot of inconsistent results, despite the variables being accessed by a single thread at a time (race-free)

2 Verification of Amdahl's law

Amdahl's law states that the total speedup of a program is ultimately decided by the non-parallelizable portion of the porgram.
To verify this, the snippet in Listing 3 performs a modified parallel version of the **saxpy** routine and calculates execution time and speedups by varying the number of threads.
The execution time for every thread configuration was taken as the mean of 10 runs, with all statistics compiled into **benchmarks.csv**.
A plot of the execution speedup w.r.t the mean single threaded computation time is also provided in Figure 3.

```java
import java.util.Random;

public class Saxpy {
    int num_workers;        // # of threads launched
    int N;                  // 1M elements
    int G;                  // generations
    float A;                // momentum
    long seed = 2021;
    Random rng;
    float X[], Y[], S[];    // arrays in global memory

    // Serial code for initialization
    public Saxpy(int num_workers, int N, int G, float A){
        this.num_workers = num_workers;
        this.N = N;
        this.G = G;
        this.A = A;
        rng = new Random(this.seed);
        X = new float[N];
        Y = new float[N];
        S = new float[N];

        // initialize X, Y
        for(int i=0;i<N;++i){
            X[i] = rng.nextFloat();
            Y[i] = rng.nextFloat();
        }
    }

    // Parallel kernel called by thread(s)
    class Kernel implements Runnable{

        int start, stop;

        Kernel(int start, int stop){
            this.start = start;
            this.stop = stop;
        }

        @Override
        public void run(){
            // perform saxpy on the current slice
            for(int g=0;g<G;++g){
                for(int i = start;i < stop; ++i){
                    S[i] = A * X[i] - Y[i];
                    Y[i] = X[i];
                    X[i] = S[i];
                }
            }
        }
    }

    // create threads and execute
    void startExecution(){

        Kernel [] kList = new Kernel[this.num_workers];
        Thread [] tList = new Thread[this.num_workers];

        int offset = this.N / this.num_workers;

        for(int i = 0;i < this.num_workers;++i){
            int start = i * offset;
            int stop = Math.min(start+offset, this.N);
```

```java
            kList[i] = new Kernel(start,stop);
            tList[i] = new Thread(kList[i]);
            tList[i].start();
        }

        try {
            for(int i=0;i<this.num_workers;++i){
                tList[i].join();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {

        int num_workers = 1;     // default # of threads
        int G = 2000;            // default # of generations

        // Parse args
        for(int i=0;i<args.length;++i){

            if(args[i].equals("--workers") || args[i].equals("-j")){
                num_workers = Integer.parseInt(args[++i]);
            }

            if(args[i].equals("--generations") || args[i].equals("-g")){
                G = Integer.parseInt(args[++i]);
            }
        }

        // Start execution
        Saxpy o = new Saxpy(num_workers, 1<<20, G, 0.9f);
        o.startExecution();
    }
}
```

Listing 3: Code for parallelization of the **saxpy** operation with num_workers threads

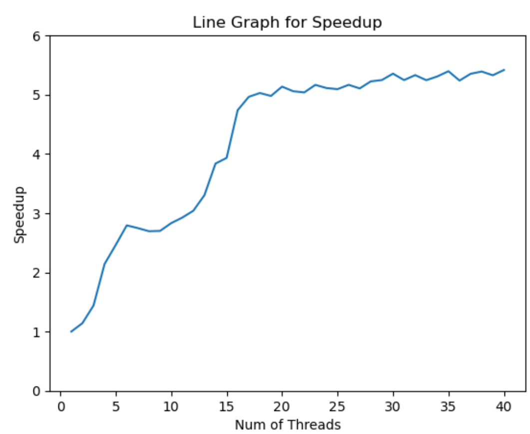

Figure 3: Graph of mean speedup as a function of the number of threads. We see that for more number of threads, the speedup plateaus out. The exact timing values are recorded in **benchmarks.csv**

3 Java threads share the heap

We show that java threads share the heap using the snippet in listing 4 and it's corresponding output in listing 5.
Since the list is in heap memory, and all thread names are reflected in it, we can conclude that java threads share the heap.

```
import java.util.ArrayList;
import java.util.List;

public class heapshare {

    class InnerClass implements Runnable{

        List<String> s;

        InnerClass(){
            s = new ArrayList<String>();
        }

        // called by worker threads
        synchronized void append(String tid){
            s.add(tid);
            System.out.println("Added ["+tid+"]");
            System.out.println("Contents are :: "+s.toString());
        }

        // called by Main thread
        void show(){
            System.out.println(s);
        }

        @Override
        public void run(){
            this.append(Thread.currentThread().getName());
        }
    }
    public static void main(String[] args) {

        heapshare obj = new heapshare();
        heapshare.InnerClass base = obj.new InnerClass();

        System.out.print("Heap contents before calls :: ");
        base.show();

        // Declare threads and call inner class
        Thread t1 = new Thread(base, "Thread 1");
        Thread t2 = new Thread(base, "Thread 2");
        Thread t3 = new Thread(base, "Thread 3");

        t1.start();
        t2.start();
        t3.start();

        try {
            t1.join();
            t2.join();
            t3.join();
        } catch (InterruptedException e) {e.printStackTrace();}

        System.out.print("Heap contents after calls ::");
        base.show();
    }
}
```

Listing 4: Code for showing java threads share the heap in heapshare.java

```
Heap contents before calls :: []
Added [Thread 1]
Contents are :: [Thread 1]
Added [Thread 3]
Contents are :: [Thread 1, Thread 3]
Added [Thread 2]
```

```
Contents are :: [Thread 1, Thread 3, Thread 2]
Heap contents after calls ::[Thread 1, Thread 3, Thread 2]
```

Listing 5: Output of the code in listing 4

4 Happen Before Relation between statements

In this section, the statement at line i is represented as S_i. If there is a happens before relation from the action on line i to line j, it is represented as $S_i \rightarrow S_j$. For statements with both $S_1 \rightarrow S_2$ and $S_2 \rightarrow S_1$, the happens before relations are denoted by $S_1 \longleftrightarrow S_2$

4.1 For *datarace.java*

The Happens Before relations for all statements in datarace.java are shown in Figure 4.
From the graph, we see that there are conflicting accesses on statements S9, S12, S10 and S13, which are not ordered by a happens-before relationship. Hence we can also verify that a datarace exists in datarace.java.

4.2 For *dataracefree.java*

For *dataracefree.java* whose snippet is provided in Listing 2, the happens before relations are visualized in Figure 5. Note that since S9, S10, S12 and S13 are accessed through a synchronized methods, there is a happens-before relationship between the accesses.
Since all conflicting memory accesses are ordered by a happens before relationship, there is no data race in the program.
However, the total order of execution might still vary depending on the Java scheduler. Thus the given code is data-race free, but still not sequentially consistent.
Reasoning based on explanation given at Java Memory Model(JMM) docs

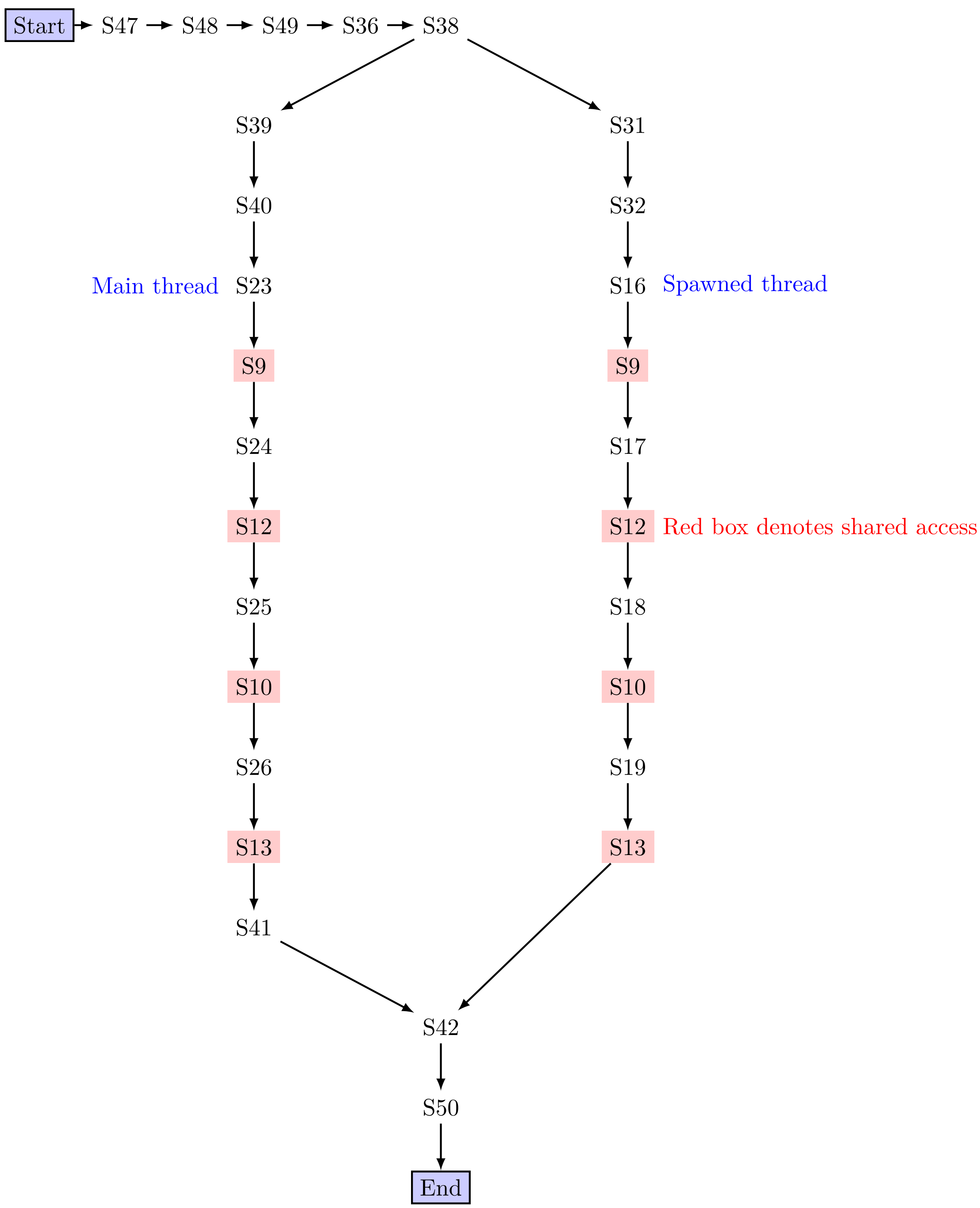

Figure 4: Showing all **Happens Before** Relations in datarace.java as a directed graph. The statements S9, S10, S12 and S13 can be simaltaneously accessed, with no happens before relation between the accesses, leading to a data race.

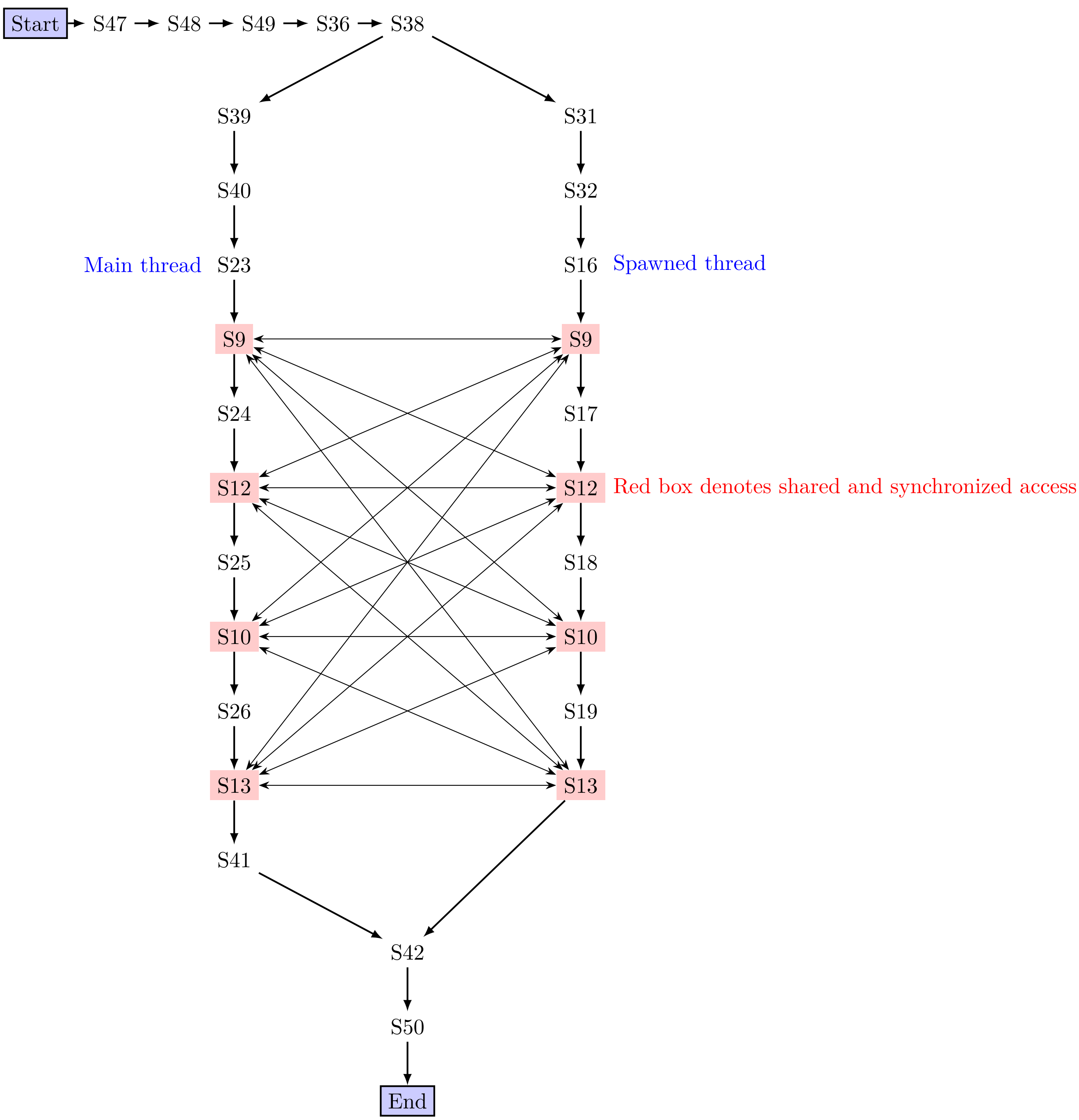

Figure 5: Visualizing all **Happens Before** Relations in `dataracefree.java` as a directed graph. The red block denoted synchronized access to that statement. There are edges between every unlock and subsequent lock of the statements in S9, S10, S12 and S13

5 Deadlocks

5.1 Using synchronized methods

I wrote a Java snippet which re-creates the classic Dining Philosopher's deadlock problem, where 5 philosophers are seated at a table with only 5 available forks. The code deadlocks when each philosopher picks up exactly 1 fork for the meal. The snippet is present in Listing 6. The output of the code is given in the Figure 6.

```java
class DiningPhilosopher implements Runnable{

    public int N;                          // the number of people
    public DiningPhilosopher right;        // the philosopher to the right
    final String[] choices = {"left","right"};

    DiningPhilosopher(int N){
        this.N = N;
        right=null;
    }

    void set_right(DiningPhilosopher rt){
        this.right=rt;
    }

    synchronized void pick(String tid, int ch){
        System.out.println("[" + tid + "]: picked "+choices[ch]+" fork");
        right.pick(tid,1-ch);
    }

    @Override
    public void run(){
        String tid = Thread.currentThread().getName();
        // Force deadlock
        while(true){
            pick(tid,0);
        }
    }
}

public class synmethodlock {
    public static void main(String[] args) {
        DiningPhilosopher p1 = new DiningPhilosopher(5);
        DiningPhilosopher p2 = new DiningPhilosopher(5);
        DiningPhilosopher p3 = new DiningPhilosopher(5);
        DiningPhilosopher p4 = new DiningPhilosopher(5);
        DiningPhilosopher p5 = new DiningPhilosopher(5);

        p1.set_right(p2);
        p2.set_right(p3);
        p3.set_right(p4);
        p4.set_right(p5);
        p5.set_right(p1);

        // Setup threads and start

        Thread t1 = new Thread(p1,"Thread 1");
        Thread t2 = new Thread(p2,"Thread 2");
        Thread t3 = new Thread(p3,"Thread 3");
        Thread t4 = new Thread(p4,"Thread 4");
        Thread t5 = new Thread(p5,"Thread 5");

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();
    }
}
```

Listing 6: Code implementing the dining philosopher's problem

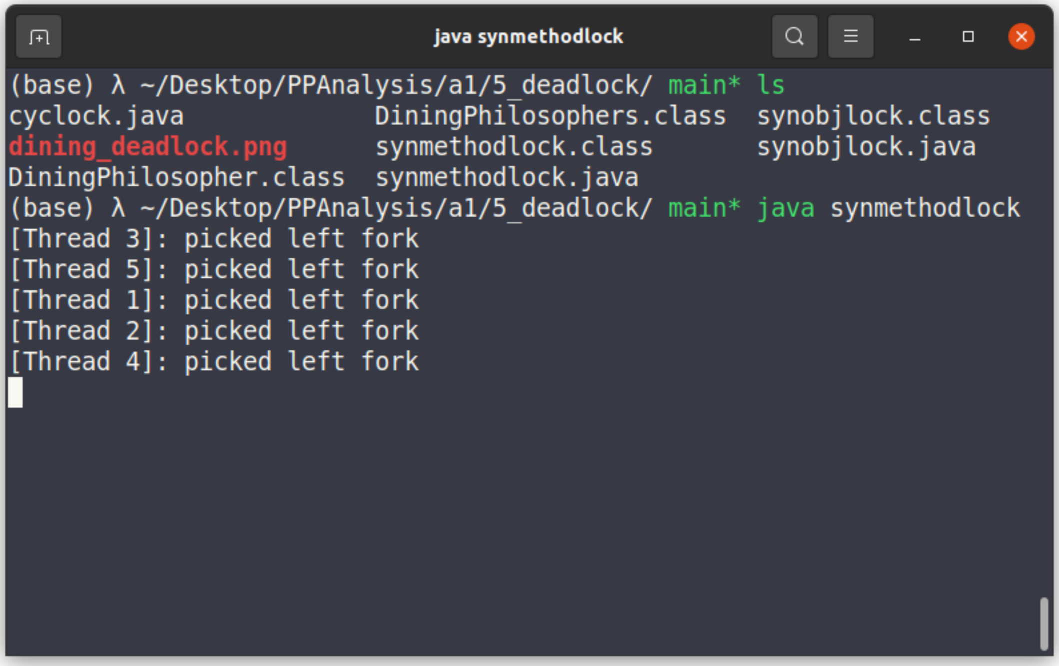

Figure 6: Output of code in Listing 6

5.2 Using a CyclicBarrier

The code snippet in Listing 7 spawns N threads which print a message and then synchronize at a barrier. Once the barrier is tripped by the last thread, it is again reset (hence **cyclic**), and the last thread runs the barrier action.
This barrier action involves spawning a new thread which again awaits at the same barrier(which is reset). Thus we have a cyclic dependency between the post-barrier thread, the cyclic barrier and the last thread, which are all waiting for each other. This leads to a deadlock between the last thread and the post-barrier thread.
A screenshot of the execution is shown in Figure 7

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

public class cyclicbarrierlock {

    CyclicBarrier barrier;
    int num_workers;

    class Worker implements Runnable{
        @Override
        public void run(){
            String tid = Thread.currentThread().getName();
            System.out.println("["+tid+"] waiting at barrier");
            try {
                barrier.await();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    class BarrierHandler implements Runnable{
        @Override
        public void run(){
            String tid = Thread.currentThread().getName();
            System.out.println("["+tid+"] cleaning up barrier");

```

```java
            Worker w = new Worker();
            Thread t = new Thread(w,"Cleanup Thread");
            t.start();
            System.out.println("["+tid+"] waiting for ["+t.getName()+"] to finish");

            // Unreachable code at runtime - results in deadlock
            // There's a cyclic dependency between (t, this.barrier, tid)
            try{
                t.join();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
        }
    }

    cyclicbarrierlock(int num_workers){
        this.num_workers = num_workers;
        this.barrier = null;
    }

    // Create cyclic barrier, threads and call them
    void startExecution(){

        this.barrier = new CyclicBarrier(this.num_workers, new BarrierHandler());

        for(int i=1;i<=this.num_workers;++i){
            Thread t = new Thread(new Worker());
            t.setName("Thread" + i);
            t.start();
        }
    }
    public static void main(String[] args) {
        cyclicbarrierlock o = new cyclicbarrierlock(5);
        o.startExecution();
    }
}
```

Listing 7: Code implementing a cyclic barrier deadlock

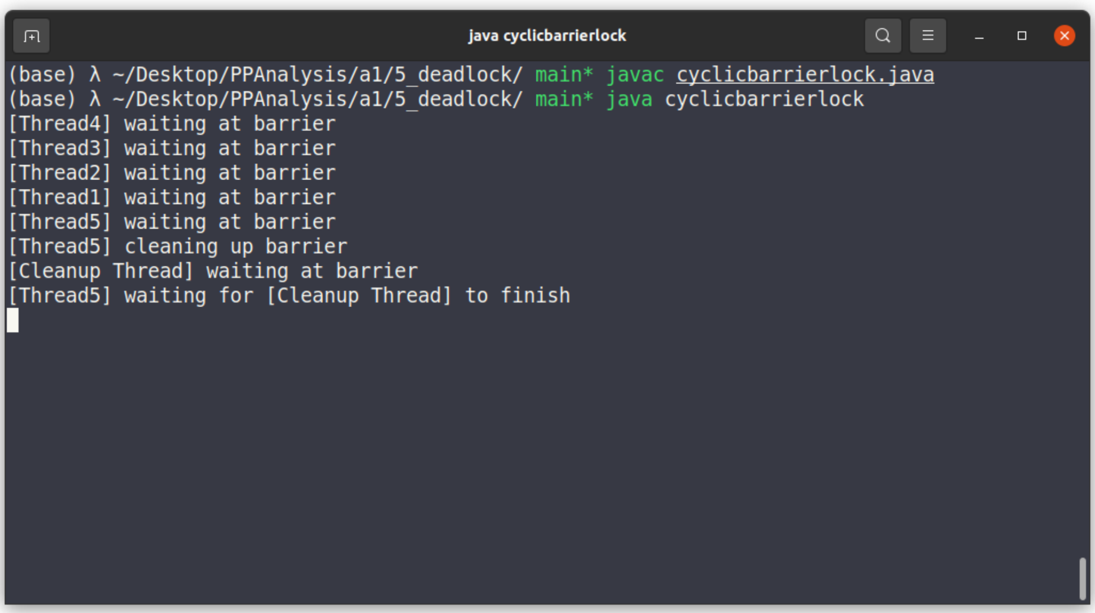

Figure 7: The cyclic barrier deadlock in action.