

Written Assignment 1

Vimarsh Sathia

February 25, 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.

```

1 public class datarace implements Runnable {
2
3     // Access to variables is not data race-free
4     // In addition, the program will have race-conditions
5     // during runtime
6     int reward;
7     int num_workers;
8
9     int get_reward(){return this.reward;}
10    int get_workers(){return this.num_workers;}
11
12    void set_reward(int v){this.reward = v;}
13    void set_workers(int v){this.num_workers = v;}
14
15    void addPushups(int reps){
16        int t1 = get_reward();
17        set_reward(t1 + reps);
18        int t2 = get_workers();
19        set_workers(t2 + 1);
20    }
21
22    void cheatDay(){
23        int t1 = get_reward();
24        set_reward(t1/2);
25        int t2 = get_workers();
26        set_workers(t2-1);
27    }
28
29    @Override
30    public void run(){
31        try{Thread.sleep(0,5);} catch (InterruptedException e){}
32        addPushups(50);
33    }
34
35    void startExecution(){
36        Thread t = new Thread(this);
37        try {
38            t.start();
39            Thread.sleep(0,10);
40            this.cheatDay();
41            System.out.println("Middle: [" + this.get_workers() + "," + this.get_reward()+"]");
42            t.join();
43        } catch (InterruptedException e) { e.printStackTrace();}
44    }
45
46    public static void main(String args[]){
47        datarace o = new datarace();
48        System.out.println("Initial: [" + o.get_workers() + "," + o.get_reward()+"]");
49        o.startExecution();

```

```

50     System.out.println("Final: [" + o.get-workers() + "," + o.get-reward()+"]");
51 }
52 }

```

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

```

PPAnalysis [main] ~ PowerShell
(base) + ~\Desktop\PPAnalysis\al\1_datarace [main == +8 ~1 -0 !]> javac datarace.java
(base) + ~\Desktop\PPAnalysis\al\1_datarace [main == +9 ~1 -0 !]> java datarace
Initial: [0,0]
Middle: [-1,0]
Final: [0,50]
(base) + ~\Desktop\PPAnalysis\al\1_datarace [main == +9 ~1 -0 !]> java datarace
Initial: [0,0]
Middle: [-1,0]
Final: [0,50]
(base) + ~\Desktop\PPAnalysis\al\1_datarace [main == +8 ~1 -0 !]> java datarace
Initial: [0,0]
Middle: [-1,0]
Final: [0,50]
(base) + ~\Desktop\PPAnalysis\al\1_datarace [main == +9 ~1 -0 !]> java datarace
Initial: [0,0]
Middle: [-1,0]
Final: [0,50]
(base) + ~\Desktop\PPAnalysis\al\1_datarace [main == +8 ~1 -0 !]> java datarace
Initial: [0,0]
Middle: [-1,0]
Final: [-1,0]
(base) + ~\Desktop\PPAnalysis\al\1_datarace [main == +8 ~1 -0 !]> java datarace
Initial: [0,0]
Middle: [-1,0]
Final: [0,50]
(base) + ~\Desktop\PPAnalysis\al\1_datarace [main == +8 ~1 -0 !]> java datarace
Initial: [0,0]
Middle: [-1,0]
Final: [0,50]
(base) + ~\Desktop\PPAnalysis\al\1_datarace [main == +8 ~1 -0 !]> |

```

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 guaranteed, since reads and writes may be interleaved between both the running threads.

This behaviour can be observed in Figure 2.

```

1 public class dataracefree implements Runnable {
2
3     // Access to variables is now data race-free
4     // However absolute atomicity is not guaranteed in methods
5     // Race conditions can still occur at runtime.
6     int reward;
7     int num_workers;
8
9     synchronized int get_reward(){return this.reward;}
10    synchronized int get_workers(){return this.num_workers;}
11
12    synchronized void set_reward(int v){this.reward = v;}
13    synchronized void set_workers(int v){this.num_workers = v;}
14
15    void addPushups(int reps){
16        int t1 = get_reward();
17        set_reward(t1 + reps);
18        int t2 = get_workers();
19        set_workers(t2 + 1);
20    }

```

```

21
22 void cheatDay(){
23     int t1 = get_reward();
24     set_reward(t1/2);
25     int t2 = get_workers();
26     set_workers(t2-1);
27 }
28
29 @Override
30 public void run(){
31     try{Thread.sleep(0,5);} catch (InterruptedException e){}
32     addPushups(50);
33 }
34
35 void startExecution(){
36     Thread t = new Thread(this);
37     try {
38         t.start();
39         Thread.sleep(0,10);
40         this.cheatDay();
41         System.out.println("Middle: [" + this.get_workers() + "," + this.get_reward()+"]");
42         t.join();
43     } catch (InterruptedException e) { e.printStackTrace();}
44 }
45
46 public static void main(String args[]){
47     datarace o = new datarace();
48     System.out.println("Initial: [" + o.get_workers() + "," + o.get_reward()+"]");
49     o.startExecution();
50     System.out.println("Final: [" + o.get_workers() + "," + o.get_reward()+"]");
51 }
52 }

```

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

```

(base) + ~\Desktop\PPAnalysis\al\l_datarace [main ≡ +8 ~1 -0 !]> javac .\dataracefree.java
(base) + ~\Desktop\PPAnalysis\al\l_datarace [main ≡ +9 ~2 -1 !]> ls

Directory: C:\Users\vimar\Desktop\PPAnalysis\al\l_datarace

Mode                LastWriteTime         Length Name
----                -
-a---             2/21/2021  3:38 PM          96702 data_race_free_output.png
-a---             2/22/2021 12:52 AM         172429 datarace_output.png
-a---             2/22/2021 12:50 AM           2007 datarace.class
-a---             2/22/2021 12:50 AM           1539 datarace.java
-a---             2/22/2021 12:54 AM           2039 dataracefree.class
-a---             2/22/2021 12:54 AM           1629 dataracefree.java

(base) + ~\Desktop\PPAnalysis\al\l_datarace [main ≡ +9 ~2 -1 !]> java dataracefree
Initial: [0,0]
Middle: [0,50]
Final: [0,50]
(base) + ~\Desktop\PPAnalysis\al\l_datarace [main ≡ +9 ~2 -1 !]> java dataracefree
Initial: [0,0]
Middle: [0,50]
Final: [0,50]
(base) + ~\Desktop\PPAnalysis\al\l_datarace [main ≡ +9 ~2 -1 !]> java dataracefree
Initial: [0,0]
Middle: [-1,50]
Final: [-1,50]
(base) + ~\Desktop\PPAnalysis\al\l_datarace [main ≡ +9 ~2 -1 !]>
(base) + ~\Desktop\PPAnalysis\al\l_datarace [main ≡ +9 ~2 -1 !]>
(base) + ~\Desktop\PPAnalysis\al\l_datarace [main ≡ +9 ~2 -1 !]>
(base) + ~\Desktop\PPAnalysis\al\l_datarace [main ≡ +9 ~2 -1 !]>

```

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 program. 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.

```

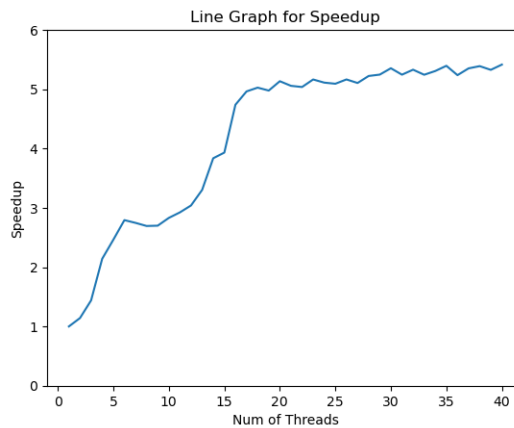
1 import java.util.Random;
2
3 public class Saxpy {
4     int num_workers;           // # of threads launched
5     int N;                     // 1M elements
6     int G;                     // generations
7     float A;                   // momentum
8     long seed = 2021;
9     Random rng;
10    float X[], Y[], S[];       // arrays in global memory
11
12    // Serial code for initialization
13    public Saxpy(int num_workers, int N, int G, float A){
14        this.num_workers = num_workers;
15        this.N = N;
16        this.G = G;
17        this.A = A;
18        rng = new Random(this.seed);
19        X = new float[N];
20        Y = new float[N];
21        S = new float[N];
22
23        // initialize X, Y
24        for(int i=0; i<N; ++i){
25            X[i] = rng.nextFloat();
26            Y[i] = rng.nextFloat();
27        }
28    }
29
30    // Parallel kernel called by thread(s)
31    class Kernel implements Runnable{
32
33        int start, stop;
34
35        Kernel(int start, int stop){
36            this.start = start;
37            this.stop = stop;
38        }
39
40        @Override
41        public void run(){
42            // perform saxpy on the current slice
43            for(int g=0; g<G; ++g){
44                for(int i = start; i < stop; ++i){
45                    S[i] = A * X[i] - Y[i];
46                    Y[i] = X[i];
47                    X[i] = S[i];
48                }
49            }
50        }
51    }
52
53    // create threads and execute
54    void startExecution(){
55
56        Kernel [] kList = new Kernel[this.num_workers];
57        Thread [] tList = new Thread[this.num_workers];
58
59        int offset = this.N / this.num_workers;
60
61        for(int i = 0; i < this.num_workers; ++i){
62            int start = i * offset;
63            int stop = Math.min(start+offset, this.N);

```

```

64         kList[i] = new Kernel(start, stop);
65         tList[i] = new Thread(kList[i]);
66         tList[i].start();
67     }
68
69     try {
70         for(int i=0; i<this.num_workers; ++i){
71             tList[i].join();
72         }
73     } catch (InterruptedException e) {
74         e.printStackTrace();
75     }
76 }
77
78 public static void main(String[] args) {
79
80     int num_workers = 1;    // default # of threads
81     int G = 2000;          // default # of generations
82
83     // Parse args
84     for(int i=0; i<args.length; ++i){
85
86         if(args[i].equals("--workers") || args[i].equals("-j")){
87             num_workers = Integer.parseInt(args[++i]);
88         }
89
90         if(args[i].equals("--generations") || args[i].equals("-g")){
91             G = Integer.parseInt(args[++i]);
92         }
93     }
94
95     // Start execution
96     Saxpy o = new Saxpy(num_workers, 1<<20, G, 0.9f);
97     o.startExecution();
98 }
99

```

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

```

1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class heapshare {
5
6     class InnerClass implements Runnable{
7
8         List<String> s;
9
10        InnerClass(){
11            s = new ArrayList<String>();
12        }
13
14        // called by worker threads
15        synchronized void append(String tid){
16            s.add(tid);
17            System.out.println("Added ["+tid+"]");
18            System.out.println("Contents are :: "+s.toString());
19        }
20
21        // called by Main thread
22        void show(){
23            System.out.println(s);
24        }
25
26        @Override
27        public void run(){
28            this.append(Thread.currentThread().getName());
29        }
30    }
31    public static void main(String[] args) {
32
33        heapshare obj = new heapshare();
34        heapshare.InnerClass base = obj.new InnerClass();
35
36        System.out.print("Heap contents before calls :: ");
37        base.show();
38
39        // Declare threads and call inner class
40        Thread t1 = new Thread(base, "Thread 1");
41        Thread t2 = new Thread(base, "Thread 2");
42        Thread t3 = new Thread(base, "Thread 3");
43
44        t1.start();
45        t2.start();
46        t3.start();
47
48        try {
49            t1.join();
50            t2.join();
51            t3.join();
52        } catch (InterruptedException e) {e.printStackTrace();}
53
54        System.out.print("Heap contents after calls ::");
55        base.show();
56    }
57 }

```

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

```

1 Heap contents before calls :: []
2 Added [Thread 1]
3 Contents are :: [Thread 1]
4 Added [Thread 3]
5 Contents are :: [Thread 1, Thread 3]
6 Added [Thread 2]

```

```
7 Contents are :: [Thread 1, Thread 3, Thread 2]
8 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](#)

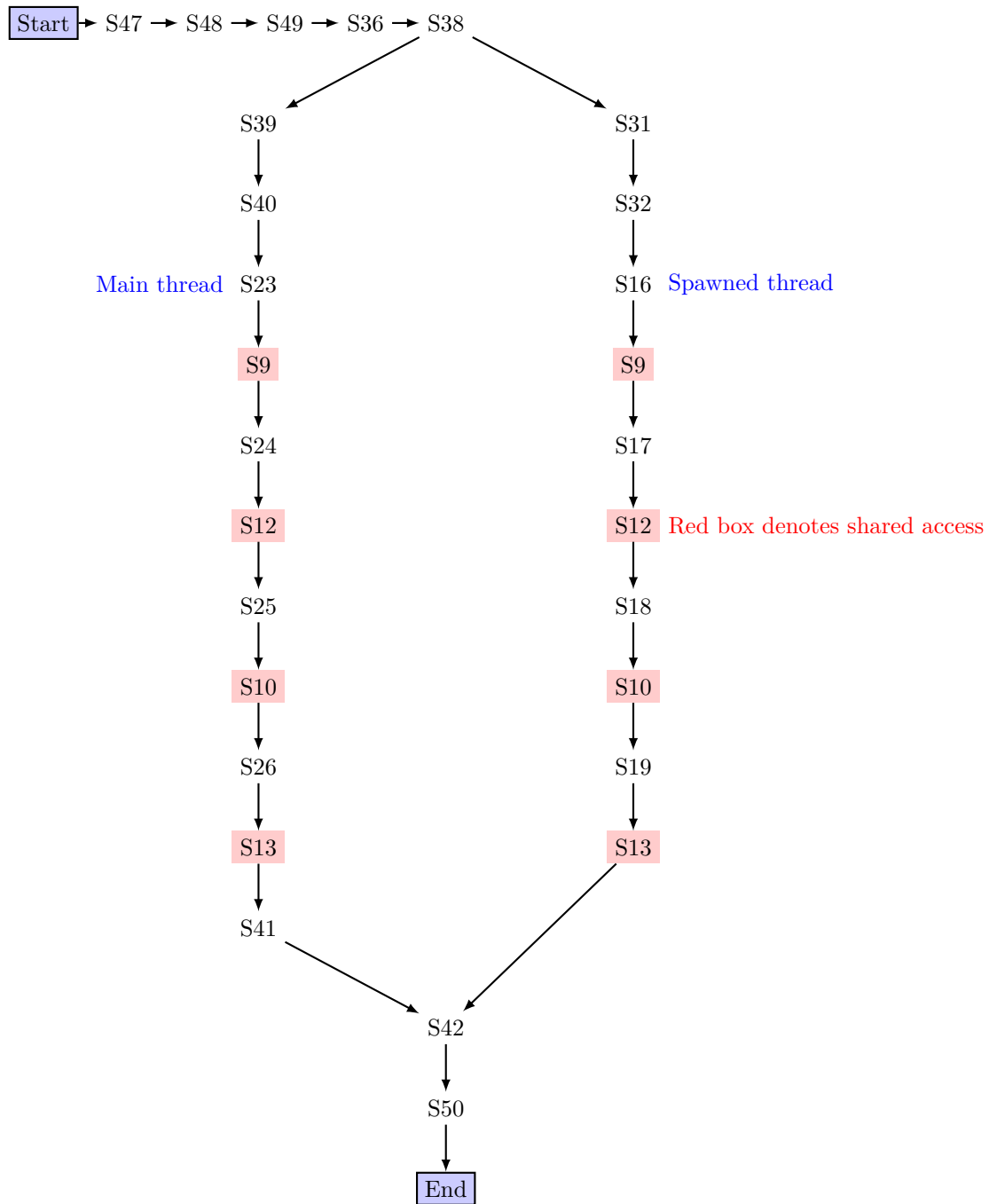


Figure 4: Showing all **Happens Before** Relations in `datarace.java` as a directed graph. The statements S9, S10, S12 and S13 can be simultaneously accessed, with no happens before relation between the accesses, leading to a data race.

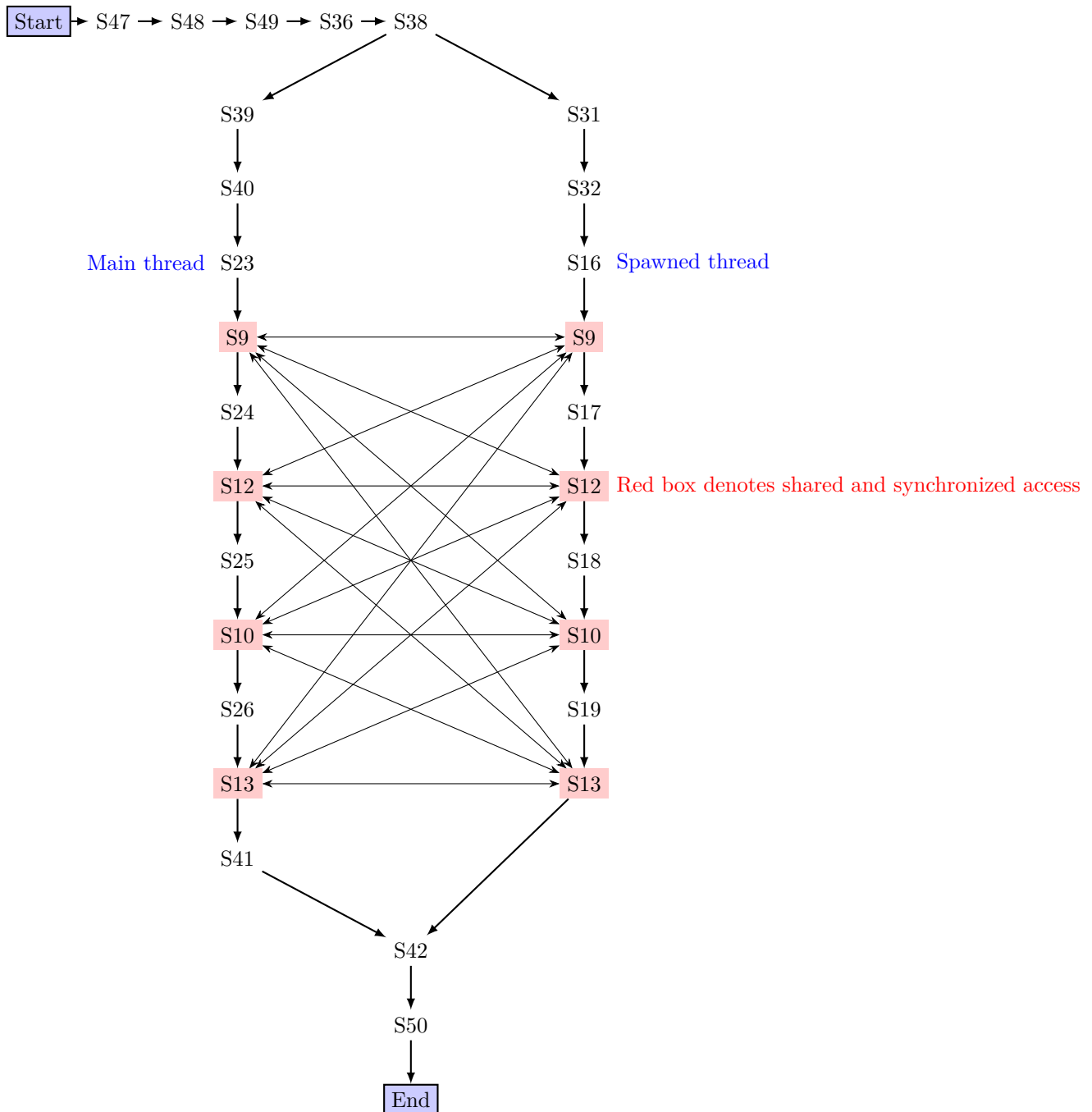


Figure 5: Visualizing all **Happens Before** Relations in `dataracefree.java` as a directed graph. The red block denoted synchronized access to that statement. There are edges between every unlock and subsequent lock of the statements in S9, S10, S12 and S13

5 Deadlocks

5.1 Using synchronized methods

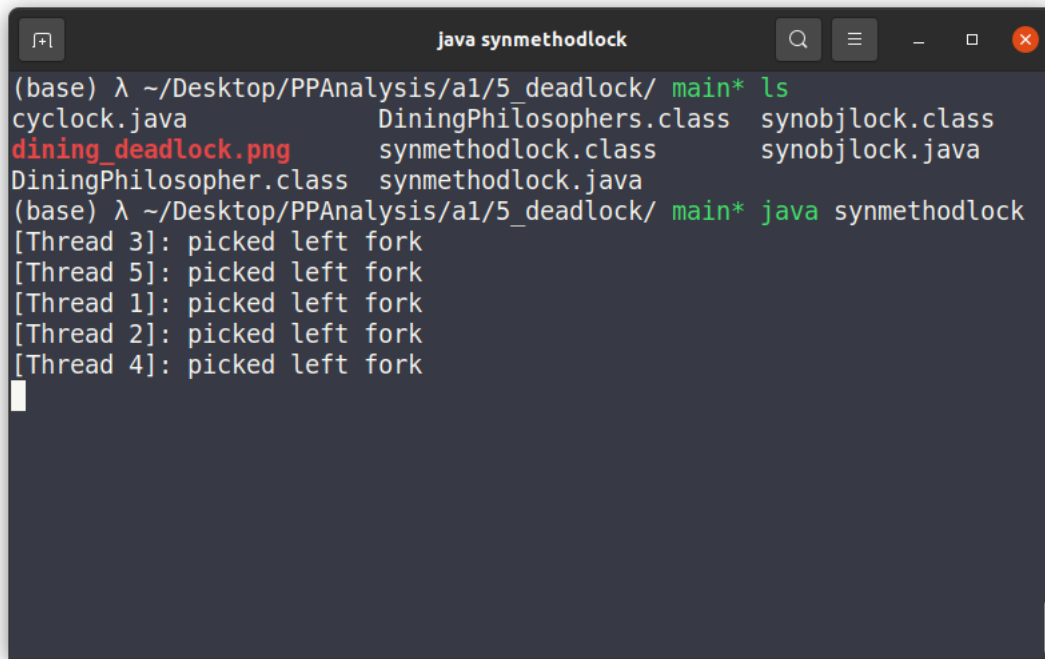
I wrote a Java snippet which re-creates the classic Dining Philosopher's deadlock problem, where 5 philosophers are seated at a table with only 5 available forks. The code deadlocks when each philosopher picks up exactly 1 fork for the meal. The snippet is present in Listing 6. The output of the code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

```

1 class DiningPhilosopher implements Runnable{
2
3     public int N;                // the number of people
4     public DiningPhilosopher right; // the philosopher to the right
5     final String[] choices = {"left", "right"};
6
7     DiningPhilosopher(int N){
8         this.N = N;
9         right=null;
10    }
11
12    void set_right(DiningPhilosopher rt){
13        this.right=rt;
14    }
15
16    synchronized void pick(String tid, int ch){
17        System.out.println("[ " + tid + " ]: picked "+choices[ch]+" fork");
18        right.pick(tid, 1-ch);
19    }
20
21    @Override
22    public void run(){
23        String tid = Thread.currentThread().getName();
24        // Force deadlock
25        while(true){
26            pick(tid, 0);
27        }
28    }
29 }
30
31 public class synmethodlock {
32     public static void main(String[] args) {
33         DiningPhilosopher p1 = new DiningPhilosopher(5);
34         DiningPhilosopher p2 = new DiningPhilosopher(5);
35         DiningPhilosopher p3 = new DiningPhilosopher(5);
36         DiningPhilosopher p4 = new DiningPhilosopher(5);
37         DiningPhilosopher p5 = new DiningPhilosopher(5);
38
39         p1.set_right(p2);
40         p2.set_right(p3);
41         p3.set_right(p4);
42         p4.set_right(p5);
43         p5.set_right(p1);
44
45         // Setup threads and start
46
47         Thread t1 = new Thread(p1, "Thread 1");
48         Thread t2 = new Thread(p2, "Thread 2");
49         Thread t3 = new Thread(p3, "Thread 3");
50         Thread t4 = new Thread(p4, "Thread 4");
51         Thread t5 = new Thread(p5, "Thread 5");
52
53         t1.start();
54         t2.start();
55         t3.start();
56         t4.start();
57         t5.start();
58     }
59 }

```

Listing 6: Code implementing the dining philosopher's problem



```

(base) λ ~/Desktop/PPAnalysis/a1/5_deadlock/ main* ls
cyclock.java      DiningPhilosophers.class  synobjlock.class
dining_deadlock.png  synmethodlock.class      synobjlock.java
DiningPhilosopher.class  synmethodlock.java
(base) λ ~/Desktop/PPAnalysis/a1/5_deadlock/ main* java synmethodlock
[Thread 3]: picked left fork
[Thread 5]: picked left fork
[Thread 1]: picked left fork
[Thread 2]: picked left fork
[Thread 4]: picked left fork

```

Figure 6: Output of code in Listing 6

5.2 Using a CyclicBarrier

The code snippet in Listing 7 spawns 5 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 code present at a common execution point. At that point, the last thread spawns 5 new Worker() threads, which call barrier.await(). This works since the barrier has been reset. The above scenario leads to a deadlock. This deadlock exists between for entry and exit to the common execution point in BarrierHandler().

This cyclic dependency can be noted through the following 2 points.

- The last thread needs it's children threads to finish execution and join(), to exit the shared barrier execution point.
- The children threads of the last thread need the last thread to release hold of the shared barrier execution point to enter and finish execution.

A screenshot of the execution is shown in Figure 7

```

1 import java.util.concurrent.BrokenBarrierException;
2 import java.util.concurrent.CyclicBarrier;
3
4 public class cyclicbarrierlock {
5
6     CyclicBarrier barrier;
7     int num_workers;
8
9     class Worker implements Runnable{
10         @Override
11         public void run(){
12             String tid = Thread.currentThread().getName();
13             System.out.println("[ "+tid+" ] waiting at barrier");
14             try {
15                 barrier.await();
16             } catch (BrokenBarrierException e) {
17                 e.printStackTrace();
18             } catch (InterruptedException e) {
19                 e.printStackTrace();
20             }
21         }
22     }

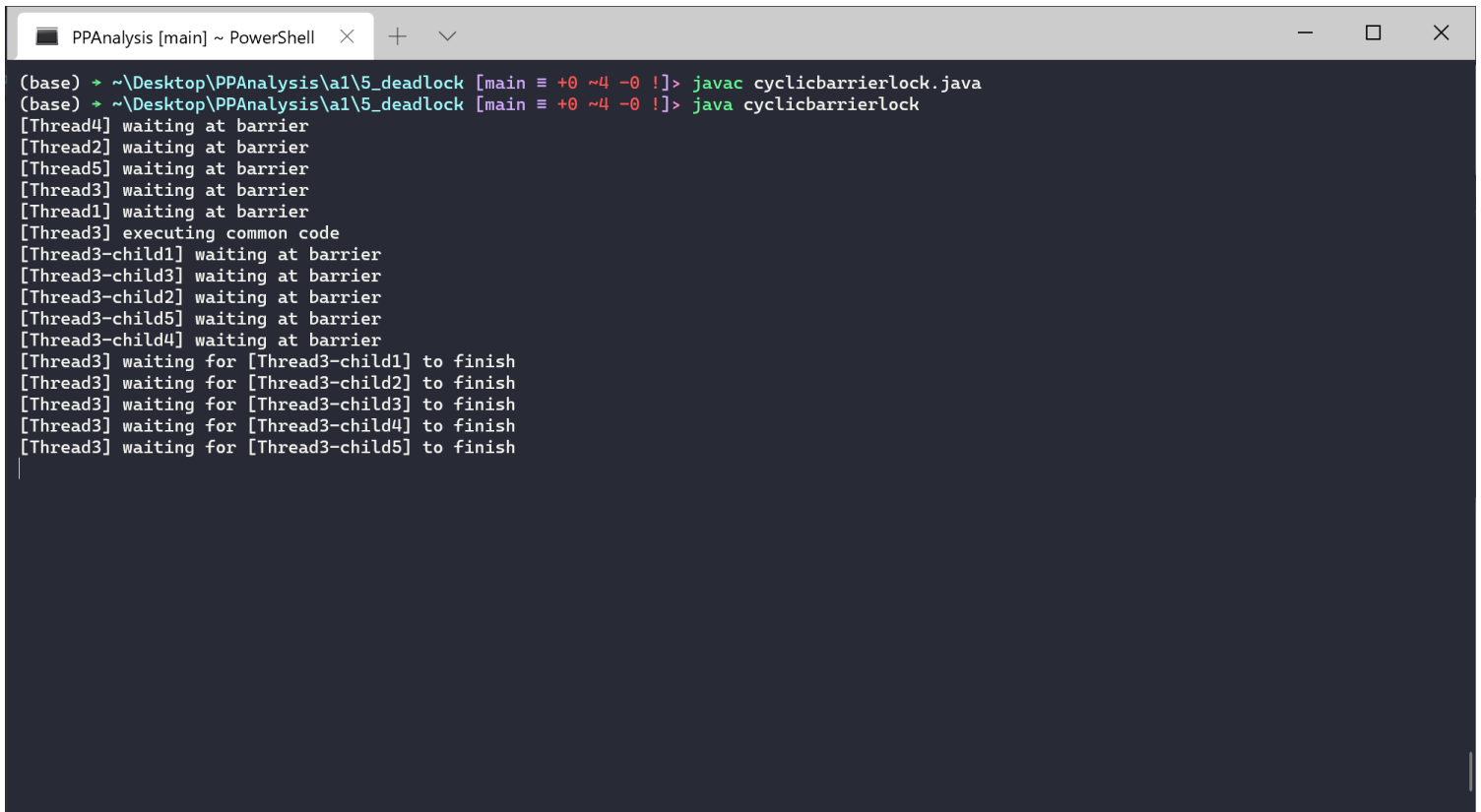
```

```

23
24 class BarrierHandler implements Runnable{
25     @Override
26     public void run(){
27         String tid = Thread.currentThread().getName();
28         System.out.println("["+tid+"] executing common code");
29
30         Worker w = new Worker();
31
32         Thread t1 = new Thread(w, tid+"-child1");
33         Thread t2 = new Thread(w, tid+"-child2");
34         Thread t3 = new Thread(w, tid+"-child3");
35         Thread t4 = new Thread(w, tid+"-child4");
36         Thread t5 = new Thread(w, tid+"-child5");
37
38         t1.start();
39         t2.start();
40         t3.start();
41         t4.start();
42         t5.start();
43
44         System.out.println("["+tid+"] waiting for ["+t1.getName()+"] to finish");
45         System.out.println("["+tid+"] waiting for ["+t2.getName()+"] to finish");
46         System.out.println("["+tid+"] waiting for ["+t3.getName()+"] to finish");
47         System.out.println("["+tid+"] waiting for ["+t4.getName()+"] to finish");
48         System.out.println("["+tid+"] waiting for ["+t5.getName()+"] to finish");
49
50         // Unreachable code at runtime – results in deadlock
51         // There's a cyclic dependency between (tid and it's child threads)
52         // Child threads need tid to exit to execute common code,
53         // and tid needs child threads to finish to exit common code.
54         try{
55             t1.join();
56             t2.join();
57             t3.join();
58             t4.join();
59             t5.join();
60         } catch (InterruptedException e){
61             e.printStackTrace();
62         }
63     }
64 }
65
66 cyclicbarrierlock(int num_workers){
67     this.num_workers = num_workers;
68     this.barrier = null;
69 }
70
71 // Create cyclic barrier, threads and call them
72 void startExecution(){
73
74     this.barrier = new CyclicBarrier(this.num_workers, new BarrierHandler());
75
76     for(int i=1;i<=this.num_workers;++i){
77         Thread t = new Thread(new Worker());
78         t.setName("Thread" + i);
79         t.start();
80     }
81 }
82 public static void main(String[] args) {
83     cyclicbarrierlock o = new cyclicbarrierlock(5);
84     o.startExecution();
85 }
86 }

```

Listing 7: Code implementing a cyclic barrier deadlock



```
(base) ~ \Desktop\PPAnalysis\al\5_deadlock [main ≡ +0 ~4 -0 !]> javac cyclicbarrierlock.java
(base) ~ \Desktop\PPAnalysis\al\5_deadlock [main ≡ +0 ~4 -0 !]> java cyclicbarrierlock
[Thread4] waiting at barrier
[Thread2] waiting at barrier
[Thread5] waiting at barrier
[Thread3] waiting at barrier
[Thread1] waiting at barrier
[Thread3] executing common code
[Thread3-child1] waiting at barrier
[Thread3-child3] waiting at barrier
[Thread3-child2] waiting at barrier
[Thread3-child5] waiting at barrier
[Thread3-child4] waiting at barrier
[Thread3] waiting for [Thread3-child1] to finish
[Thread3] waiting for [Thread3-child2] to finish
[Thread3] waiting for [Thread3-child3] to finish
[Thread3] waiting for [Thread3-child4] to finish
[Thread3] waiting for [Thread3-child5] to finish
```

Figure 7: The cyclic barrier deadlock in action. We see that the children of Thread-3 are unable to enter the shared barrier execution point, due to the presence of Thread 3, leading to **cyclic wait**.