```
patrick@nightmare:~/OS_examples$ ls
'Data Interleave'  'Data Race Illustrated'   pthread_eg.c   test_fork.c   test_fork_exec.c
patrick@nightmare:~/OS_examples$ gcc test_fork.c -o tf -lpthread
patrick@nightmare:~/OS_examples$ ./tf
Bye from the child process! PID= 0
Child 29894 Completed with status = 0
patrick@nightmare:~/OS_examples$ gcc test_fork_exec.c -o tfe -lpthread
patrick@nightmare:~/OS_examples$ ./tfe

------ BEFORE COMMAND----------
total 60
drwx------ 3 patrick patrick  4096 May 28 21:03 'Data Interleave'
drwx------ 3 patrick patrick  4096 May 28 21:08 'Data Race Illustrated'
-rw------- 1 patrick patrick  1051 May 28 19:52  pthread_eg.c
-rw------- 1 patrick patrick   619 May 27 18:50  test_fork.c
-rw------- 1 patrick patrick   576 May 27 18:48  test_fork_exec.c
-rwx------ 1 patrick patrick 16928 May 29 19:34  tf
-rwx------ 1 patrick patrick 16920 May 29  2021  tfe

-------Child Complete-----
patrick@nightmare:~/OS_examples$ vim test_fork
```

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
        pid_t pid;
        /* fork a child process */
        pid = fork();

        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) { /* child process */
                printf("\n------ BEFORE COMMAND----------\n");
                execlp("/bin/ls","ls","-l", NULL);
                printf("\n-------AFTER COMMAND-----------\n");
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait(NULL);
                printf("\n-------Child Complete-----\n");
        }
```

- With child process, have two print statements (BEFORE and AFTER)
    - AFTER never printed – because execlp is executed
    - Once execlp executed – overlays parent program, destroys whatever was in the parent address space
        - Now have new address space – load ls program into that and execute it
- Once exec operates – parent no longer exists
    - Have new program running until termination
    - Makes no sense to put code after exec statement – will never be executed

```
package interleaving;

public class STest {
        public static void main(String[] args) {
                int numberThreads =100;
                AThread [] threads = new AThread[numberThreads];

           for (int i=0;i<numberThreads;i++)
                        threads[i]= new AThread(i);

                System.out.println("Starting simulation with " + numberThreads + " threads");

                for (int i=0;i<numberThreads;i++)
                        threads[i].start(); // threads don't run until you start them
                System.out.println("Parent thread completed");
        }
}
~
~
~
~
~
~
~
~
"STest.java" [dos] 17L, 501C                                          11,2-16
```

crucial for being able to solve many of the synchronization problems.

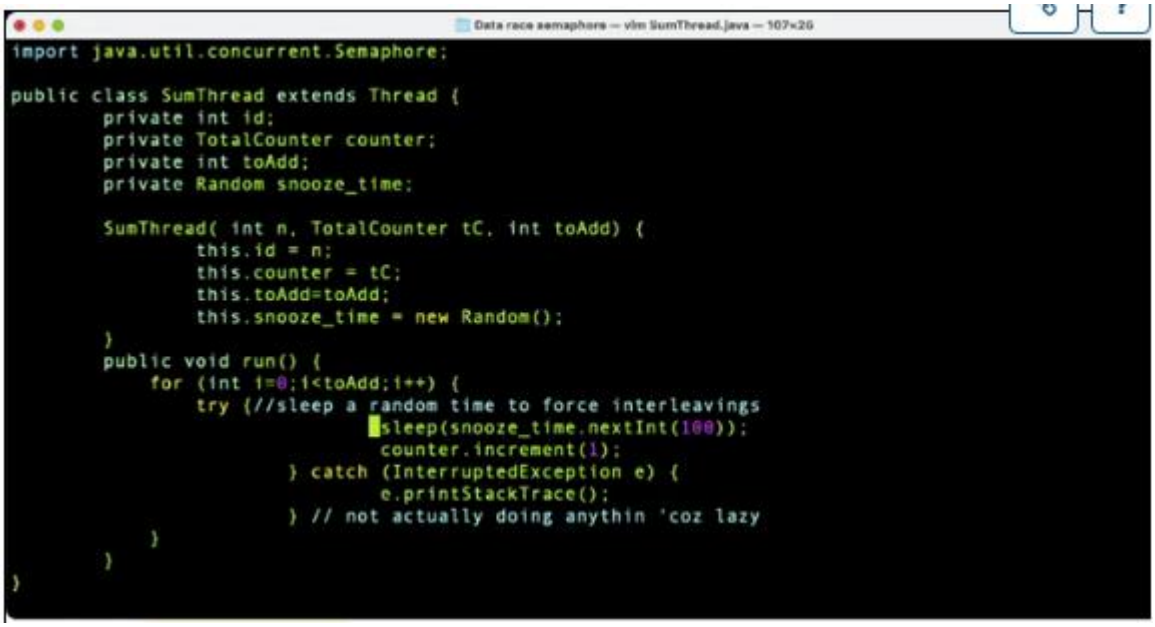
- With this code, parent stops before child => issues!

```
package dataRace;

public class TotalCounter {
        private int sum;

        TotalCounter() {sum=0;}

        public void increment(int incrVal) {
                sum+=incrVal;
        }

        public int getValue() {
                return sum;
        }
}
~
~
~
~
~
~
~
~
~
~
"TotalCounter.java" 15L, 199C                                         8,23-30
```

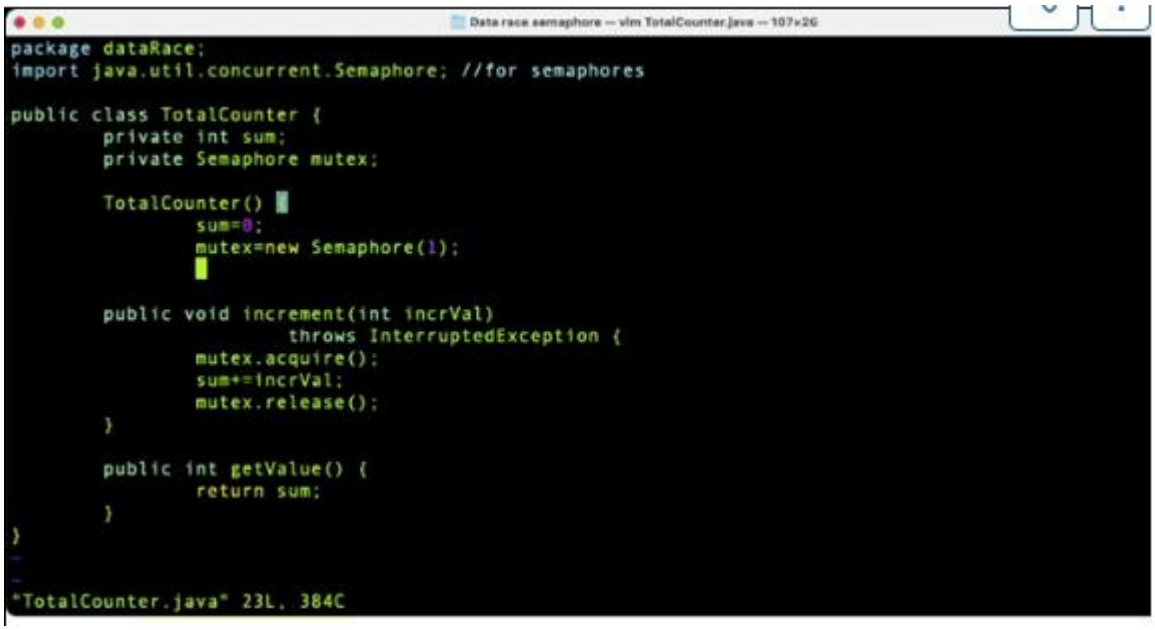
```
package dataRace;

public class SumThread extends Thread {
        private TotalCounter counter;
        private int toAdd;

        SumThread(TotalCounter tC, int toAdd) {
                this.counter = tC;
                this.toAdd=toAdd;
        }
        public void run() {
            for (int i=0;i<toAdd;i++) {
                counter.increment(1);
            }
        }
}
~
~
~
~
~
~
~
~
"SumThread.java" [noeol] 16L, 291C                                    6,1-8
```

```
package dataRace;

public class RaceTest {
        public static void main(String[] args) {
                int numberThreads =100;
                int incr = 100; //divide adding task among threads
                TotalCounter counter = new TotalCounter();
                SumThread [] threads = new SumThread[numberThreads];
                for (int i=0;i<numberThreads;i++)
                        threads[i]= new SumThread(counter,incr); //create the threads

                System.out.println("Starting simulation with " + numberThreads + " threads");
                for (int i=0;i<numberThreads;i++)
                        threads[i].start();// threads don't run until you start them
/*
               for (int i = 0;i  < numberThreads; i++)
               try {
                        threads[i].join();
               }
               catch (InterruptedException e) {}
*/
                System.out.println("Counter  is :"+ counter.getValue());
        }
}
~
"RaceTest.java" 24L, 774C                                             14,23-44
```

- These three programs show how data races occur



```
import java.util.concurrent.Semaphore;

public class SumThread extends Thread {
        private int id;
        private TotalCounter counter;
        private int toAdd;
        private Random snooze_time;

        SumThread( int n, TotalCounter tC, int toAdd) {
                this.id = n;
                this.counter = tC;
                this.toAdd=toAdd;
                this.snooze_time = new Random();
        }
        public void run() {
            for (int i=0;i<toAdd;i++) {
                try {//sleep a random time to force interleavings
                                sleep(snooze_time.nextInt(100));
                                counter.increment(1);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        } // not actually doing anythin 'coz lazy
            }
        }
}
```



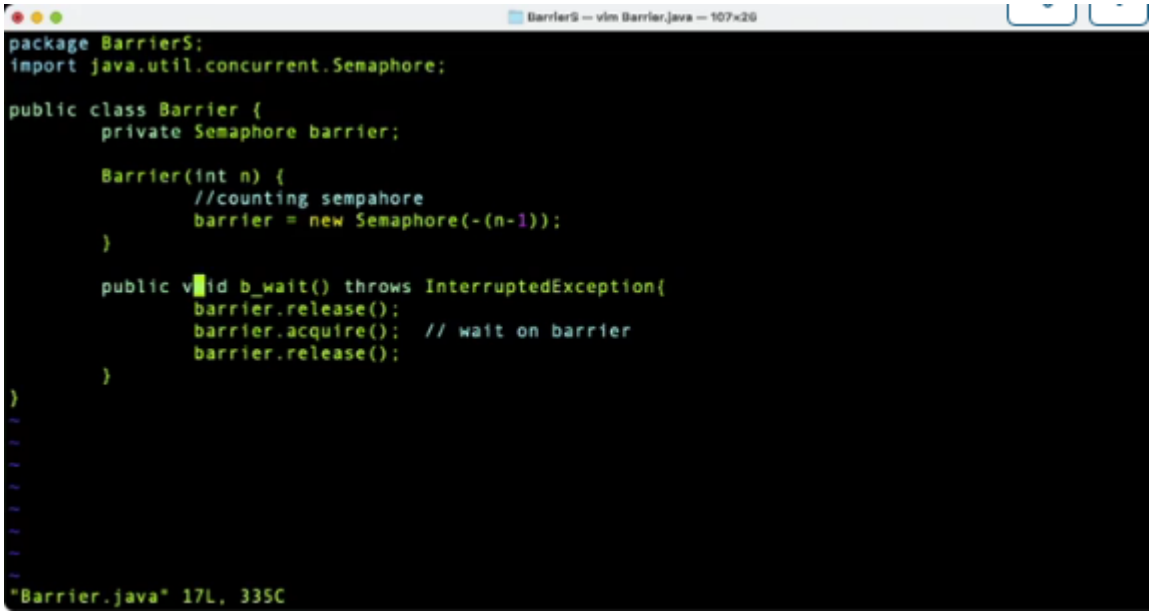
```
package dataRace;
import java.util.concurrent.Semaphore; //for semaphores

public class TotalCounter {
        private int sum;
        private Semaphore mutex;

        TotalCounter() {
                sum=0;
                mutex=new Semaphore(1);


        public void increment(int incrVal)
                        throws InterruptedException {
                mutex.acquire();
                sum+=incrVal;
                mutex.release();
        }

        public int getValue() {
                return sum;
        }
}
~
~
"TotalCounter.java" 23L, 384C
```

- Semaphores protect shared data – guarantees that only one thread can increment at any specific time

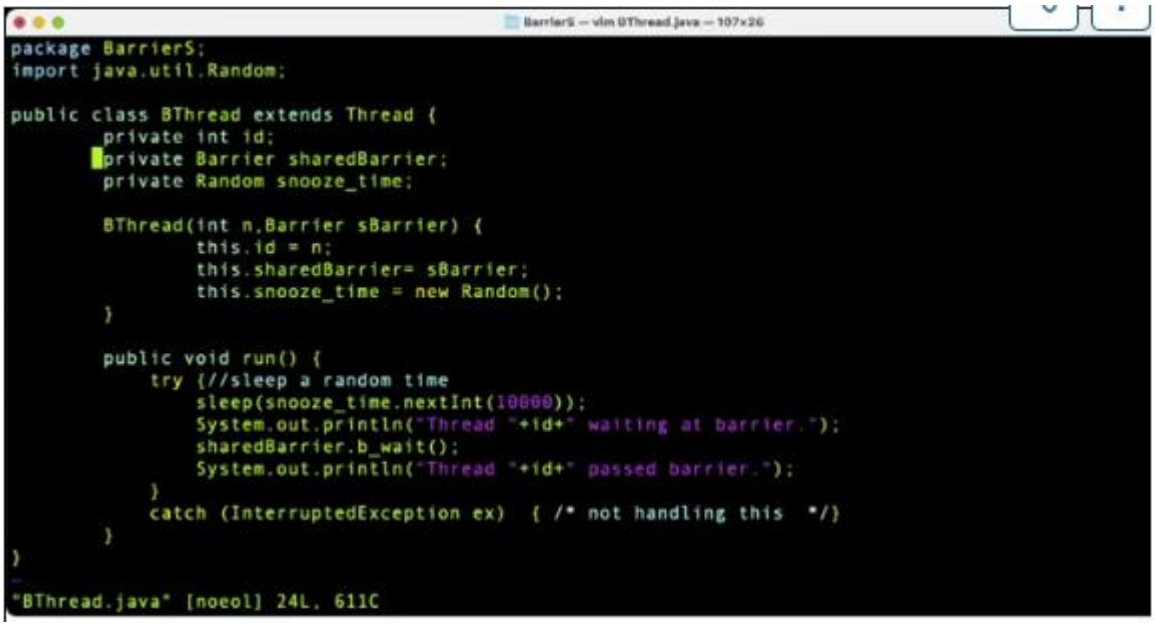
```
package rendezvous;
import java.util.Random;
import java.util.concurrent.Semaphore;

public class RThread extends Thread {
        private Semaphore signal1;
        private Semaphore signal2;
        private Random snooze_time;
        private String name;

        RThread(Semaphore s1, Semaphore s2,String nm) {
                this.signal1= s1;
                this.signal2= s2;
                this.snooze_time = new Random();
                this.name= nm;
        }
        public void run() {
            try {//sleep a random time
                sleep(snooze_time.nextInt(100));
                System.out.println("Hello "+ name);
                signal1.release();
                signal2.acquire();
                System.out.println("Goodbye "+ name);
            }
            catch (InterruptedException ex) {
"RThread.java" [noeol] 28L, 689C
```

```
package rendezvous;
import java.util.concurrent.Semaphore; //for semaphores

public class STest {
        public static void main(String[] args) {
                Semaphore signal1 = new Semaphore(0); //start closed
                Semaphore signal2 = new Semaphore(0); //start closed

                //note signal order changed
                RThread threadA = new RThread(signal1, signal2,"Anne");
                RThread threadB = new RThread(signal2, signal1,"Bob");

                System.out.println("Starting simulation");
                threadA.start(); // start thread
                threadB.start(); // start thread
                System.out.println("Parent thread completed");
        }
}
~
~
~
~
~
~
"STest.java" 18L, 571C
```

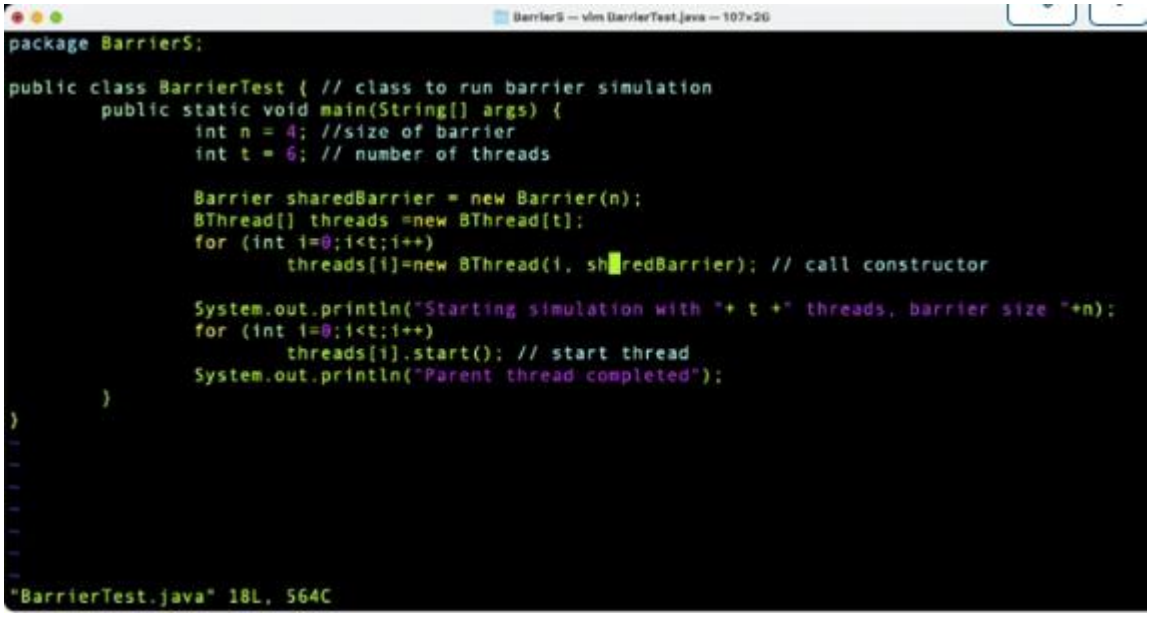
- Rendezvous with semaphores
- Need to create two semaphores that're closed – that first instruction needs control when executed
- Need to swop S1 and S2 when passed in as parameter for one of the threads



```
package BarrierS;
import java.util.concurrent.Semaphore;

public class Barrier {
        private Semaphore barrier;

        Barrier(int n) {
                //counting sempahore
                barrier = new Semaphore(-(n-1));
        }

        public void b_wait() throws InterruptedException{
                barrier.release();
                barrier.acquire();  // wait on barrier
                barrier.release();
        }
}
~
~
~
~
~
~
~
~
"Barrier.java" 17L, 335C
```

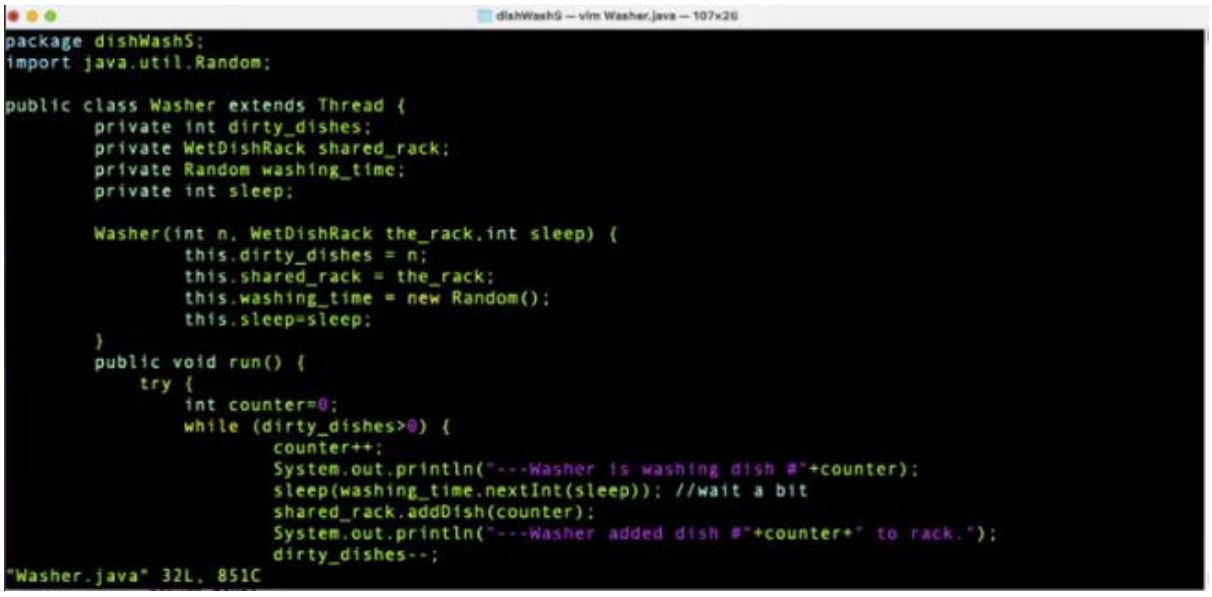


```
package BarrierS;
import java.util.Random;

public class BThread extends Thread {
        private int id;
        private Barrier sharedBarrier;
        private Random snooze_time;

        BThread(int n,Barrier sBarrier) {
                this.id = n;
                this.sharedBarrier= sBarrier;
                this.snooze_time = new Random();
        }

        public void run() {
            try {//sleep a random time
                sleep(snooze_time.nextInt(10000));
                System.out.println("Thread "+id+" waiting at barrier.");
                sharedBarrier.b_wait();
                System.out.println("Thread "+id+" passed barrier.");
            }
            catch (InterruptedException ex)  { /* not handling this  */}
        }
}
~
"BThread.java" [noeol] 24L, 611C
```

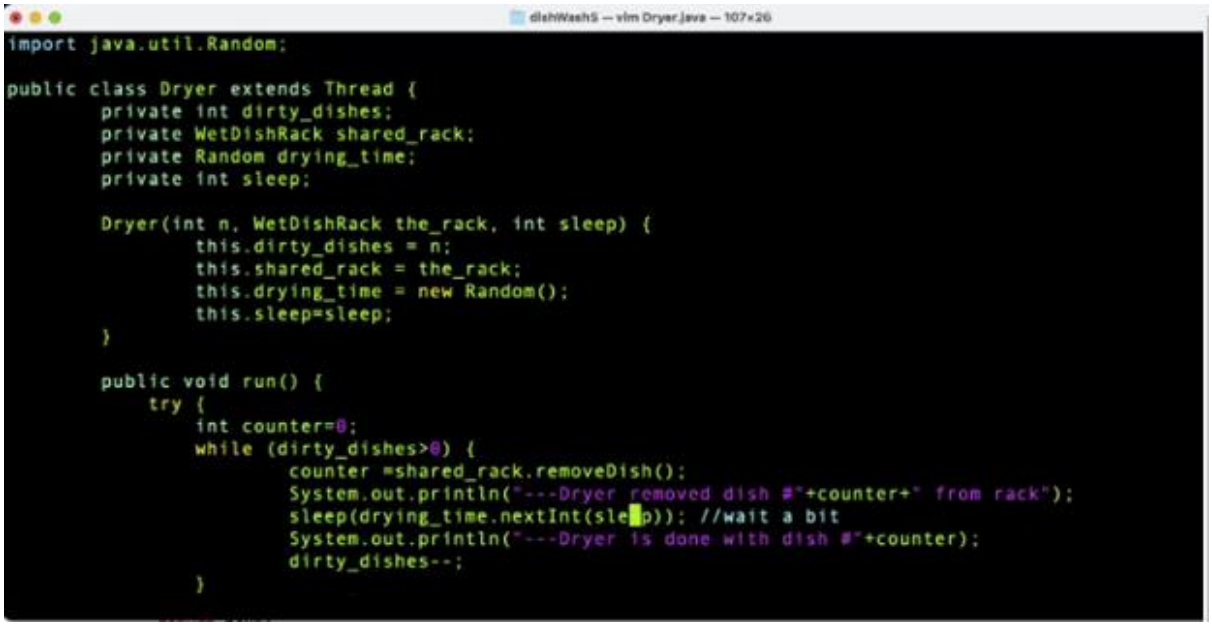


```
package BarrierS;

public class BarrierTest { // class to run barrier simulation
        public static void main(String[] args) {
                int n = 4; //size of barrier
                int t = 6; // number of threads

                Barrier sharedBarrier = new Barrier(n);
                BThread[] threads =new BThread[t];
                for (int i=0;i<t;i++)
                        threads[i]=new BThread(i, sharedBarrier); // call constructor

                System.out.println("Starting simulation with "+ t +" threads, barrier size "+n);
                for (int i=0;i<t;i++)
                        threads[i].start(); // start thread
                System.out.println("Parent thread completed");
        }
}
~
~
~
~
~
~
~
"BarrierTest.java" 18L, 564C
```

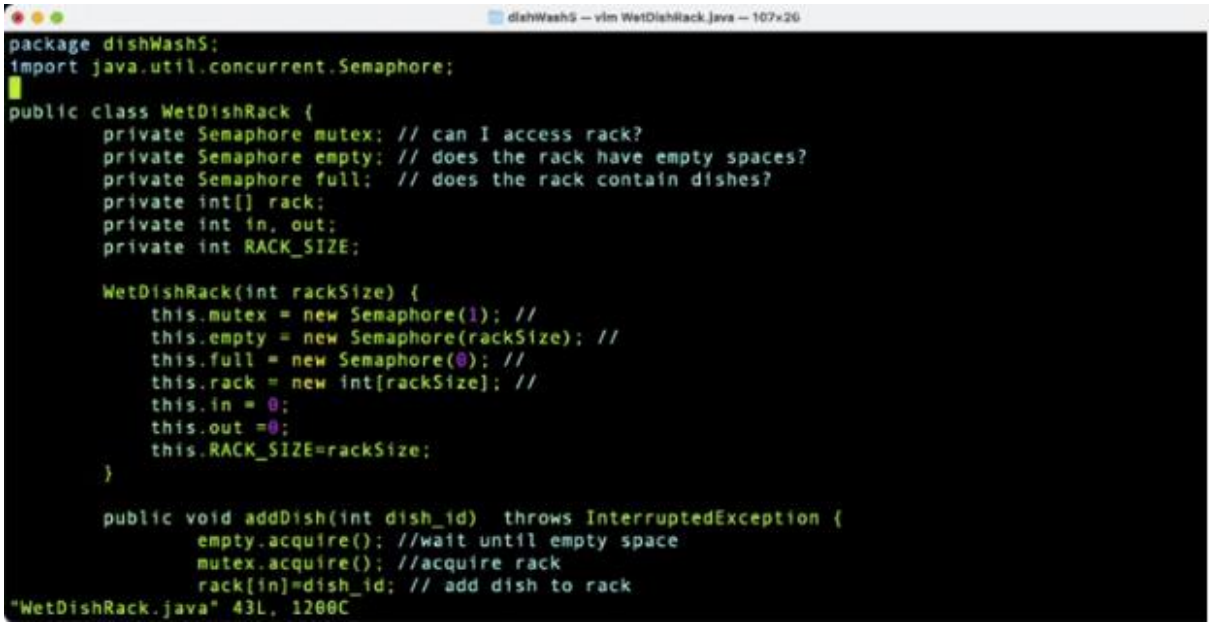
- Turnstile – wait, signal
- Works because initialise barrier to –(n-1) – when -ve, nothing can progress, inhibit anything from passing the barrier before the semaphore becomes positive
    - Spins – but we can release
- Trick – start with -ve semaphore value, so every new thread coming is pushes up that value
    - Until we can pass through busy waiting loop
- BarrierTest – have to wait until a number of threads arrive at the barrier before others pass through
    - Once barrier is opened, all threads can pass through (even those that weren't explicitly waiting at the barrier)



```
package dishWashS;
import java.util.Random;

public class Washer extends Thread {
        private int dirty_dishes;
        private WetDishRack shared_rack;
        private Random washing_time;
        private int sleep;

        Washer(int n, WetDishRack the_rack,int sleep) {
                this.dirty_dishes = n;
                this.shared_rack = the_rack;
                this.washing_time = new Random();
                this.sleep=sleep;
        }
        public void run() {
            try {
                int counter=0;
                while (dirty_dishes>0) {
                        counter++;
                        System.out.println("---Washer is washing dish #"+counter);
                        sleep(washing_time.nextInt(sleep)); //wait a bit
                        shared_rack.addDish(counter);
                        System.out.println("---Washer added dish #"+counter+" to rack.");
                        dirty_dishes--;
"Washer.java" 32L, 851C
```

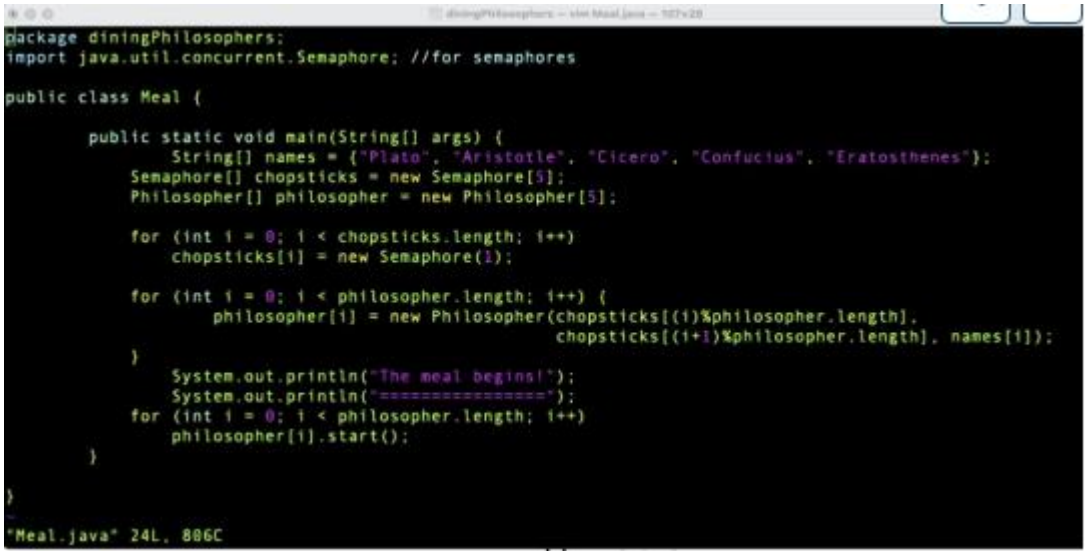


```
import java.util.Random;

public class Dryer extends Thread {
        private int dirty_dishes;
        private WetDishRack shared_rack;
        private Random drying_time;
        private int sleep;

        Dryer(int n, WetDishRack the_rack, int sleep) {
                this.dirty_dishes = n;
                this.shared_rack = the_rack;
                this.drying_time = new Random();
                this.sleep=sleep;
        }

        public void run() {
            try {
                int counter=0;
                while (dirty_dishes>0) {
                        counter =shared_rack.removeDish();
                        System.out.println("---Dryer removed dish #"+counter+" from rack");
                        sleep(drying_time.nextInt(sleep)); //wait a bit
                        System.out.println("---Dryer is done with dish #"+counter);
                        dirty_dishes--;
                }
```

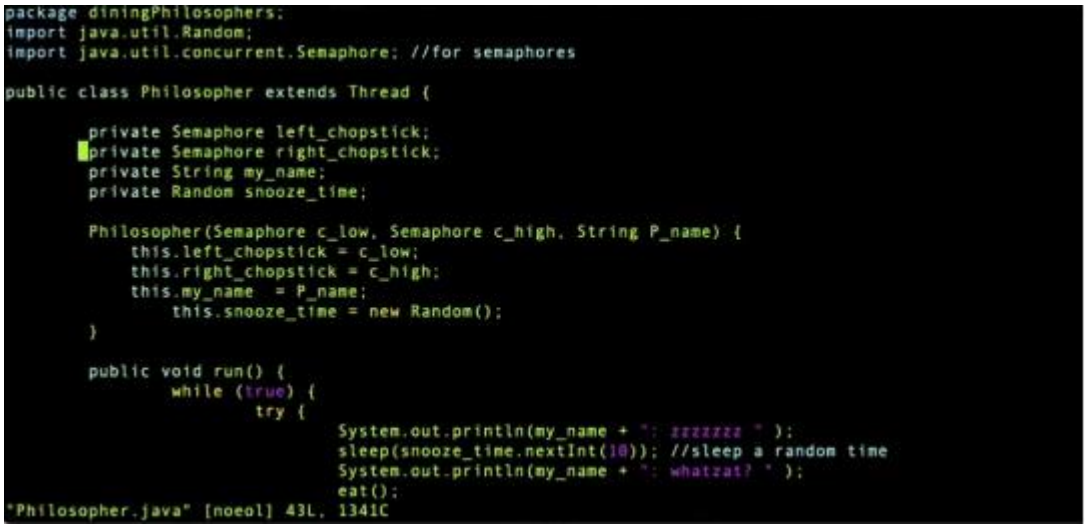


```
package dishWashS;
import java.util.concurrent.Semaphore;

public class WetDishRack {
        private Semaphore mutex; // can I access rack?
        private Semaphore empty; // does the rack have empty spaces?
        private Semaphore full;  // does the rack contain dishes?
        private int[] rack;
        private int in, out;
        private int RACK_SIZE;

        WetDishRack(int rackSize) {
            this.mutex = new Semaphore(1); //
            this.empty = new Semaphore(rackSize); //
            this.full = new Semaphore(0); //
            this.rack = new int[rackSize]; //
            this.in = 0;
            this.out =0;
            this.RACK_SIZE=rackSize;
        }

        public void addDish(int dish_id)  throws InterruptedException {
                empty.acquire(); //wait until empty space
                mutex.acquire(); //acquire rack
                rack[in]=dish_id; // add dish to rack
"WetDishRack.java" 43L, 1200C
```

- Dishwashing Sim
    - Producer is washer – produce dishes that go onto common dish rack (rack is the buffer)
    - Consumer is dryer – picks a dish from the rack to dry it
    - Here, have a limit to amount of dishes want to use
- Add randomness by having threads wait
- In run() – as washer, try and push dishes onto rack
    - addDish – ensures that don't try to insert wet dish onto rack with no space
- Dryer has similar behaviour
    - But removeDish instead of addDish
- Dishrack
    - Control access to rack
- No dryer removes the dish before its been added to the rack



```
package diningPhilosophers;
import java.util.concurrent.Semaphore; //for semaphores

public class Meal {

        public static void main(String[] args) {
                String[] names = {"Plato", "Aristotle", "Cicero", "Confucius", "Eratosthenes"};
            Semaphore[] chopsticks = new Semaphore[5];
            Philosopher[] philosopher = new Philosopher[5];

            for (int i = 0; i < chopsticks.length; i++)
                chopsticks[i] = new Semaphore(1);

            for (int i = 0; i < philosopher.length; i++) {
                    philosopher[i] = new Philosopher(chopsticks[(i)%philosopher.length],
                                                     chopsticks[(i+1)%philosopher.length], names[i]);
            }
                System.out.println("The meal begins!");
                System.out.println("================");
            for (int i = 0; i < philosopher.length; i++)
                philosopher[i].start();
        }

}
"Meal.java" 24L, 806C
```



```
package diningPhilosophers;
import java.util.Random;
import java.util.concurrent.Semaphore; //for semaphores

public class Philosopher extends Thread {

        private Semaphore left_chopstick;
        private Semaphore right_chopstick;
        private String my_name;
        private Random snooze_time;

        Philosopher(Semaphore c_low, Semaphore c_high, String P_name) {
            this.left_chopstick = c_low;
            this.right_chopstick = c_high;
            this.my_name  = P_name;
                this.snooze_time = new Random();
        }

        public void run() {
                while (true) {
                        try {
                                System.out.println(my_name + ": zzzzzz " );
                                sleep(snooze_time.nextInt(10)); //sleep a random time
                                System.out.println(my_name + ": whatzat? " );
                                eat();
"Philosopher.java" [noeol] 43L, 1341C
```

- Meal is main class
- Have semaphores, init to value 1 – regular mutex locks
    - Create philosophers with correct semaphore representing their l and r chopstick
- Philosopher class – have semaphores representing l and r chopsticks
    - Use sleep to have thread be suspended and another to be scheduled in
        - Larger chance of triggering sync problem if it exists
    - Eat() – routine that acquires and releases semaphores
        - Init wait for left chopstick – acquire blocks until avail
        - Then try to acquire right
        - Then eat
        - Then release left and right chopstick
- Classically deadlock occurs when they all try to acquire one chopstick at a time e.g. all pick up their left chopstick before their right
    -