Lab 1 - Basic Concurrency in Java

- Group 23
- Fatohi, Kristian

Task 1: Creating and joining threads

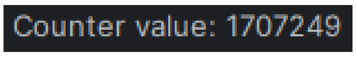
Source files:
- Task1/MainA.java (main file)
- Task1/MainB.java (main file)
- Task1/MainC.java (main file)

Task 1A: Unsynchronized Testing

Source files:
- Task1/MainA.java (main file)

To compile and execute: javac MainA.java java MainA

When I ran my program, I got the following counter value:



```
Counter value: 1707249
```

Figure 1: Image of my output from running the first task

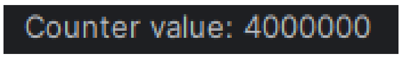
Now if I'm running this with 4 threads, I should expect 4,000,000 however, since there is no synchronization, any possible value could appear.

Task 1B: Synchronized Testing

Source files:
- Task1/MainB.java (main file)

To compile and execute: javac MainB.java java MainB

Here I use synchronization and we should expect 4,000,000, which we do as seen in Figure 2



```
Counter value: 4000000
```

Figure 2: Image of my output from running the second task

To achieve this, I have used a static final object, that will work as a lock to ensure thread safety when incrementing the counter and ofcourse the synchronized keyword in Java

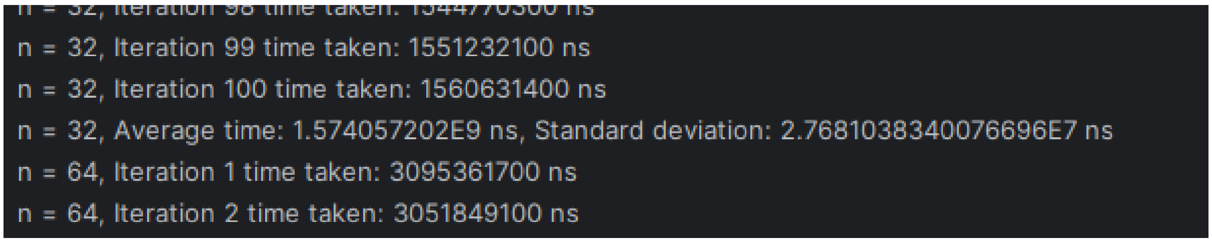
Task 1C: PDC Vs Local

Source files:

- Task1/MainC.java (main file)

To compile and execute: javac MainC.java java MainC

The figure below shows an snippet of what the terminal displayed.



```
n = 32, Iteration 98 time taken: 1544770300 ns
n = 32, Iteration 99 time taken: 1551232100 ns
n = 32, Iteration 100 time taken: 1560631400 ns
n = 32, Average time: 1.574057202E9 ns, Standard deviation: 2.7681038340076696E7 ns
n = 64, Iteration 1 time taken: 3095361700 ns
n = 64, Iteration 2 time taken: 3051849100 ns
```

Figure 3: What terminal displayed when executing

I did a 10 iteration warmup followed by a 100 measurement phase. I found that just setting the measurement phase high enough would be sufficient enough, maybe not the most efficient, so instead of trying to find the most efficient value, I “efficiently” just tried 100 and the standard deviation looked alright to me. Figure 4 shows the plotted values, as we can see, PDC manages to perform more computations over time compared to the Local system.

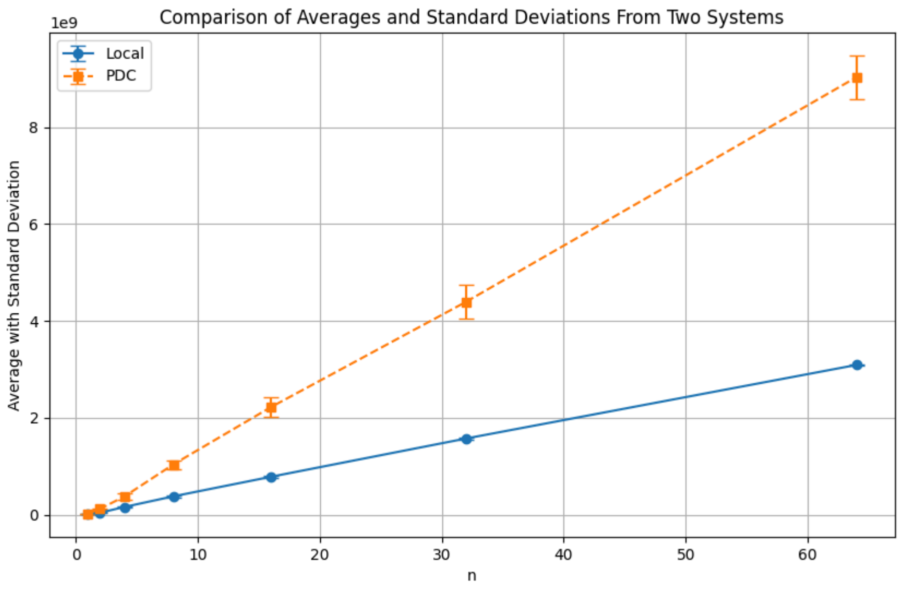


Figure 4: Comparison of Averages and StdDev.

Task 2: Simple Synchronization

Source files:

- Task2/MainA.java (main file)
- Task2/MainB.java (main file)
- Task2/MainC.java (main file)

Task 2A: Asynchronous Sender-Receiver

Source files:

- Task2/MainA.java (main file)

To compile and execute: javac MainA.java java MainA

I get a value between $0 < x < 1000000$ after running the program multiple times.

Task 2B: Busy-Waiting Receiver

Source files:

- Task2/MainB.java (main file)

To compile and execute: javac MainB.java java MainB

It does print 1,000,000, everything worked as intended, and all threads incremented correctly. However it is fair to say that it did not work when the body of the while-loop was empty, Then it just looped forever, but whenever I put in a print statement, then it worked out fine.

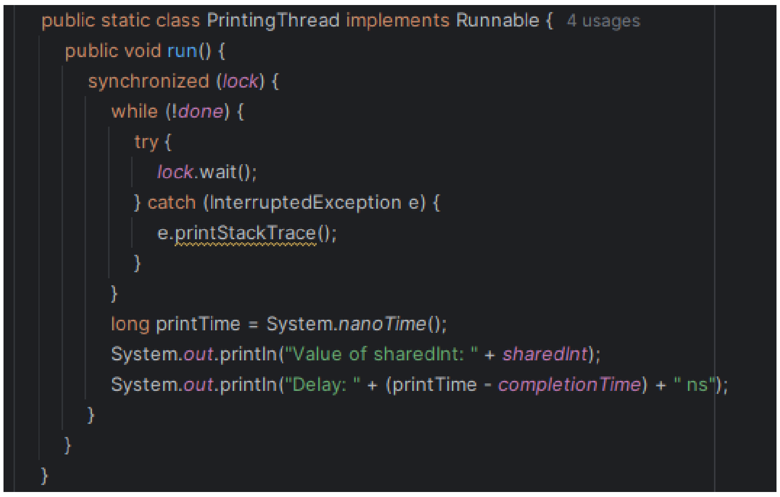
Task 2C: Waiting With Guarded Block

Source files:

- Task2/MainC.java (main file)

To compile and execute: javac MainC.java java MainC

It printed out the value., everything worked as intended. Figure 5 shows the code snippet that I inputted to solve the task about “wake up spuriosly”. In summary, The while loop ensures that even if a thread is woken up spuriously, it will recheck the condition (!done) and wait again if necessary.



```
public static class PrintingThread implements Runnable {  4 usages
    public void run() {
        synchronized (lock) {
            while (!done) {
                try {
                    lock.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            long printTime = System.nanoTime();
            System.out.println("Value of sharedInt: " + sharedInt);
            System.out.println("Delay: " + (printTime - completionTime) + " ns");
        }
    }
}
```

Figure 5: Handling Spurious Wakeups

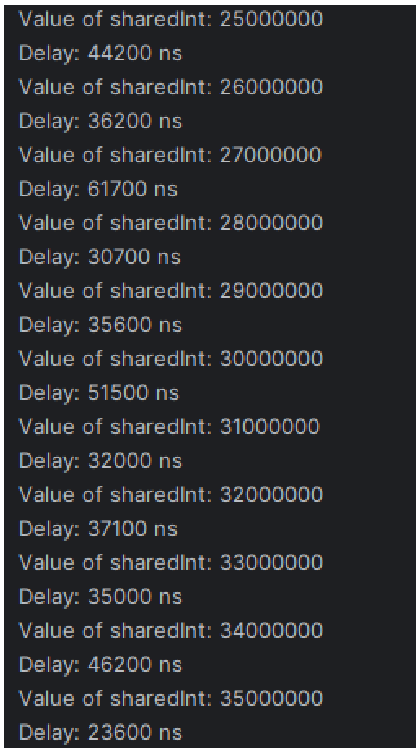
Task 2D: Guarded Block Vs Busy-Waiting

Source files:

- Task2/MainB.java (main file)
- Task2/MainC.java (main file)

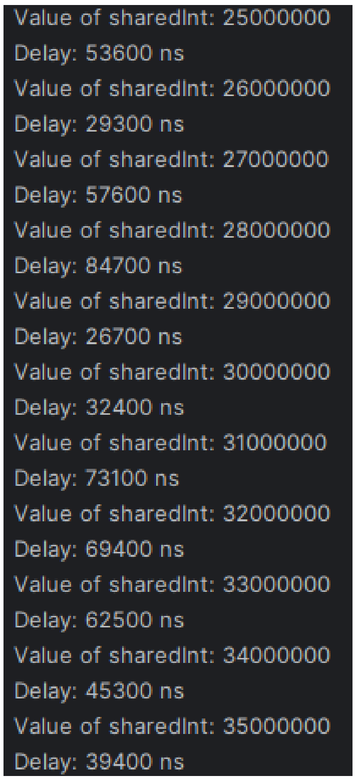
To compile and execute: javac MainC.java MainB.java java MainC MainB

Figure 6 and 7 show the respective approaches and the delay when running 25 times in the measurement phase and 10 times in the warmup phase.



```
Value of sharedInt: 25000000
Delay: 44200 ns
Value of sharedInt: 26000000
Delay: 36200 ns
Value of sharedInt: 27000000
Delay: 61700 ns
Value of sharedInt: 28000000
Delay: 30700 ns
Value of sharedInt: 29000000
Delay: 35600 ns
Value of sharedInt: 30000000
Delay: 51500 ns
Value of sharedInt: 31000000
Delay: 32000 ns
Value of sharedInt: 32000000
Delay: 37100 ns
Value of sharedInt: 33000000
Delay: 35000 ns
Value of sharedInt: 34000000
Delay: 46200 ns
Value of sharedInt: 35000000
Delay: 23600 ns
```

Figure 6: Busy-Waiting



```
Value of sharedInt: 25000000
Delay: 53600 ns
Value of sharedInt: 26000000
Delay: 29300 ns
Value of sharedInt: 27000000
Delay: 57600 ns
Value of sharedInt: 28000000
Delay: 84700 ns
Value of sharedInt: 29000000
Delay: 26700 ns
Value of sharedInt: 30000000
Delay: 32400 ns
Value of sharedInt: 31000000
Delay: 73100 ns
Value of sharedInt: 32000000
Delay: 69400 ns
Value of sharedInt: 33000000
Delay: 62500 ns
Value of sharedInt: 34000000
Delay: 45300 ns
Value of sharedInt: 35000000
Delay: 39400 ns
```

Figure 7: Guarded Blocks

Busy-waiting involves constantly checking a condition in a loop, which wastes CPU cycles. This is evident in the first image where the delays are spread over a wide range, with values fluctuating between **23600 ns** and **61700 ns.** Guarded blocks use synchronization mechanisms like locks, semaphores, or condition variables. The second image shows a slightly different pattern in delay times, with delays ranging between **26700 ns** and **84700 ns**.

Busy-waiting appears to have slightly more consistent (though inefficient) performance, as the CPU is always engaged in checking the condition. In contrast, guarded blocks introduce more variance in delay times, but they are more efficient in terms of CPU usage. Guarded blocks, while introducing some overhead and slightly longer delays in certain cases, are generally more efficient than busy-waiting because they allow the CPU to be used for other tasks. Busy-waiting wastes CPU cycles, making it inefficient for longer tasks, even though it may produce slightly lower delay times.

Task 3: Producer-Consumer Buffer using Condition Variables

Source files:

- Task3/Main.java (main file)
- Task3/Buffer.java (my producer-consumer buffer)

Task 3A: Producer-Consumer Buffer

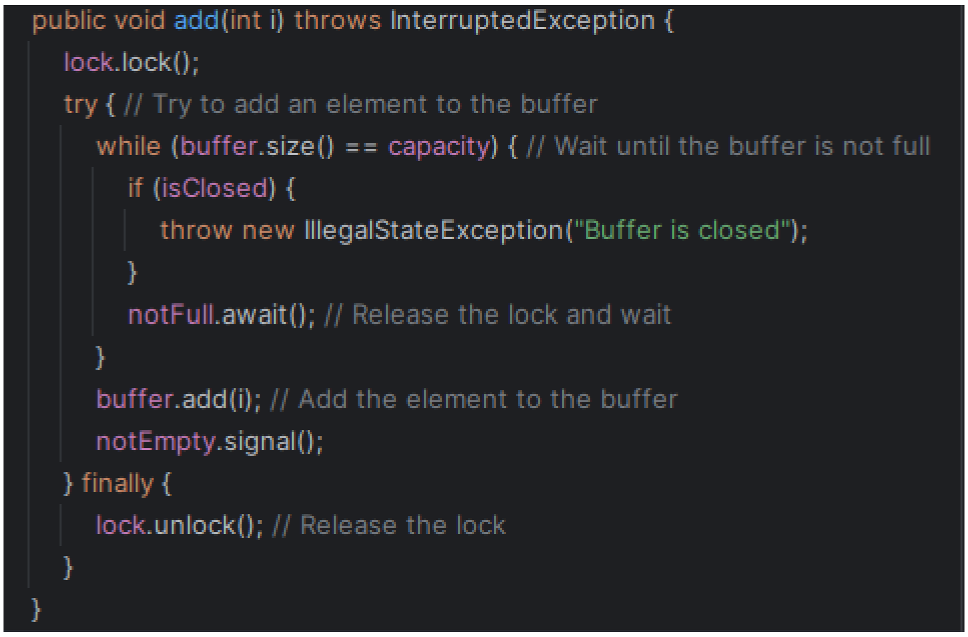
Source files:

- Task3/Buffer.java (my producer-consumer buffer file)

The Buffer class is a thread-safe bounded buffer implemented using a LinkedList and ReentrantLock with conditions for synchronization.

- **buffer:** A LinkedList that holds the buffer elements.
- **capacity:** The maximum number of elements the buffer can hold.
- **lock:** A ReentrantLock used to ensure mutual exclusion.
- **notFull:** A Condition that signals when the buffer is not full.
- **notEmpty:** A Condition that signals when the buffer is not empty.
- **isClosed:** A boolean flag indicating whether the buffer is closed.

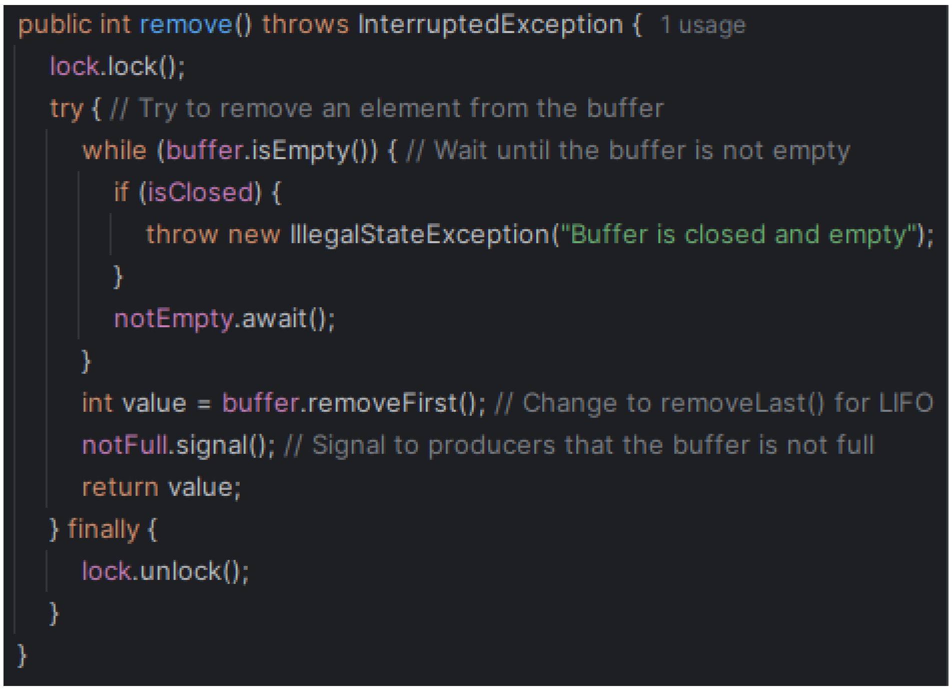
Some methods are listed and explained below:



```java
public void add(int i) throws InterruptedException {
    lock.lock();
    try { // Try to add an element to the buffer
        while (buffer.size() == capacity) { // Wait until the buffer is not full
            if (isClosed) {
                throw new IllegalStateException("Buffer is closed");
            }
            notFull.await(); // Release the lock and wait
        }
        buffer.add(i); // Add the element to the buffer
        notEmpty.signal();
    } finally {
        lock.unlock(); // Release the lock
    }
}
```

Add(int i)

- **Parameters**: i - the element to be added.
- Its purpose is to add an element to the buffer. It acquires the lock. Waits if the buffer is full unless the buffer is closed. Adds the element to the buffer. Signals that the buffer is not empty. Releases the lock.



```java
public int remove() throws InterruptedException {  1 usage
    lock.lock();
    try { // Try to remove an element from the buffer
        while (buffer.isEmpty()) { // Wait until the buffer is not empty
            if (isClosed) {
                throw new IllegalStateException("Buffer is closed and empty");
            }
            notEmpty.await();
        }
        int value = buffer.removeFirst(); // Change to removeLast() for LIFO
        notFull.signal(); // Signal to producers that the buffer is not full
        return value;
    } finally {
        lock.unlock();
    }
}
```

int Remove()

- **Returns:** The removed element.
- Its purpose is to remove and return an element from the buffer. It acquires the lock. Waits if the buffer is empty unless the buffer is closed. Removes the first element from the buffer. Signals that the buffer is not full. Releases the lock.

Task 3B: Main & Buffer

Source files:

- Task3/Main.java (main file)

The Producer adds items to the buffer until it reaches 1,000,000 items, then closes the buffer. The Consumer continuously removes items from the buffer and prints them until the buffer is closed. All of this is happening in main which initializes the buffer and starts the producer and consumer threads, then waits for them to complete.

Task 4: Counting Semaphore

Source files:

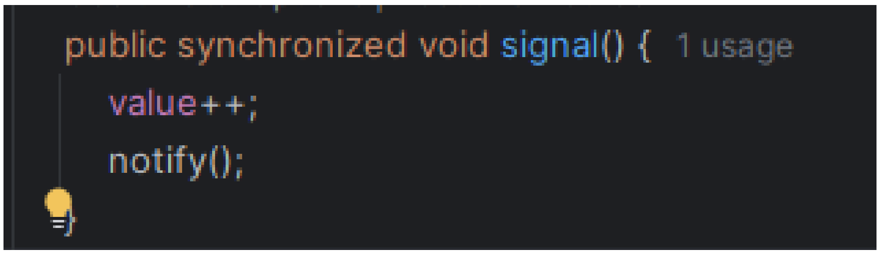
- Task4/Main.java (main file)
- Task4/CoutningSemaphore.java

Task 4A: Counting Semaphore

Source files:

- Task4/CoutningSemaphore.java

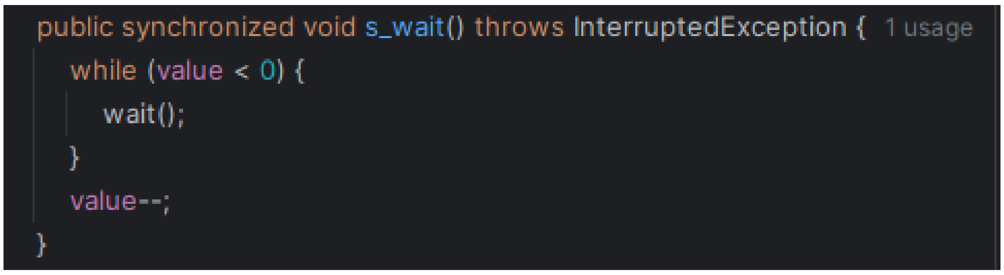
The CountingSemaphore class is a simple implementation of a counting semaphore, which is used to control access to a shared resource by multiple threads. It maintains a counter to keep track of the number of available permits.



```java
public synchronized void signal() {  1 usage
    value++;
    notify();
}
```

signal()

- Its purpose is to release a resource, increasing the number of available resources. It starts by incrementing the value field. Calls notify() to wake up a waiting thread, if any.



```java
public synchronized void s_wait() throws InterruptedException {  1 usage
    while (value < 0) {
        wait();
    }
    value--;
}
```

s_wait()

- Its purpose is to acquire a resource, decreasing the number of available resources. If the value is less than 0, the thread waits until a resource is available. The while loop handles spurious wakeups by rechecking the condition. It decrements the value field.

Task 4B: Main & Counting Semaphore

Source files:

- Task4/Main.java (main file)

In summary, the main function initializes the semaphore with an initial counter value and spawns multiple threads to test the semaphore. Each thread attempts to acquire the semaphore, does some work, and then releases it.

Problems:

- I initially forgot to handle the case where the semaphore value becomes negative correctly, leading to threads not waiting as expected.
- I also did not handle spurious wakeups, causing threads to proceed without acquiring the semaphore properly.
- At last, ensuring that signal() and s_wait() methods are synchronized is essential to prevent race conditions.

Task 5: Dining Philosophers

Source files:

- Task4/Main.java (main file)
- Task4/Philosopher.java

Task 5A: Model

After a long time, my simulation deadlocked. This occurred when all five philosophers simultaneously grabbed the chopstick on the left, this is a deadlock since no one can access the second chopstick because all are taken. I tried with more philosophers and it did deadlock faster. This I think is because there is more competition for chopsticks.

Task 5B: Deadlock Debugging

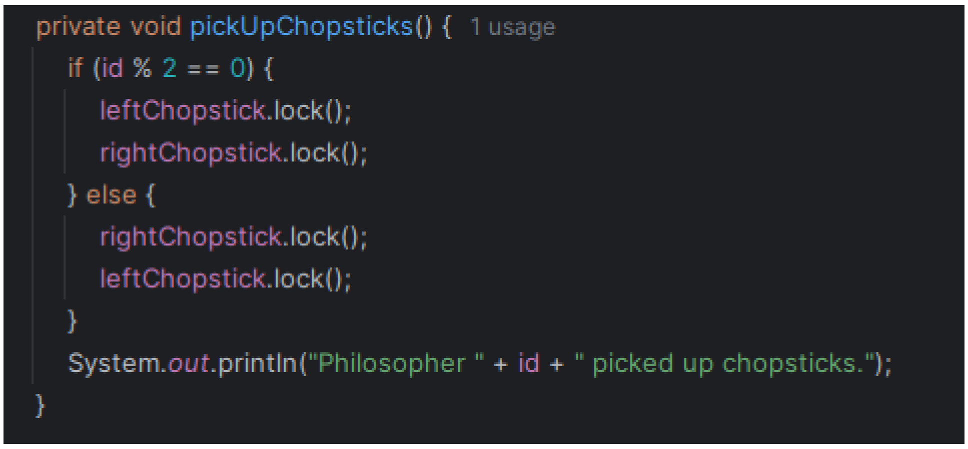
jstack: It generates a thread dump of a running Java application. This dump provides a snapshot of the current state of all threads, including information about any locks held or waited for by the threads. It is fairly simple to use, you can start by writing jstack <pid> where <pid> is the process ID of the Java program. The output will show the current stack trace of each thread, including whether a thread is blocked or waiting for a lock.

Task 5C: Implementation

Source files:

- Task4/Main.java (main file)
- Task4/Philosopher.java

I did resort to an external source for input that source is written as a footnote[1],

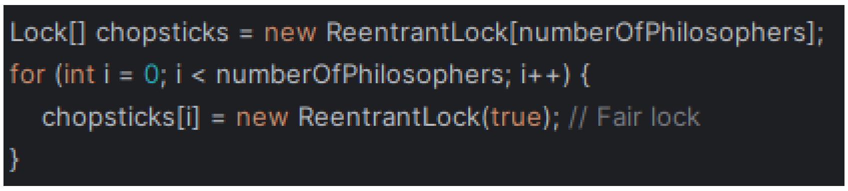


```java
private void pickUpChopsticks() {  1 usage
    if (id % 2 == 0) {
        leftChopstick.lock();
        rightChopstick.lock();
    } else {
        rightChopstick.lock();
        leftChopstick.lock();
    }
    System.out.println("Philosopher " + id + " picked up chopsticks.");
}
```

The code snippet above is how I manage to avoid deadlocks, it uses an alternating strategy. Philosophers with even IDs pick up the left chopstick first, while those with odd IDs pick up the right chopstick first. This alternating strategy prevents a circular wait condition, which is

[1] https://youtu.be/FYUi-u7UWgw?si=wLtsytrcsCe1lG3v

a common cause of deadlock. By ensuring that not all philosophers try to pick up the same chopstick first, it reduces the chances of a deadlock occurring.



```
Lock[] chopsticks = new ReentrantLock[numberOfPhilosophers];
for (int i = 0; i < numberOfPhilosophers; i++) {
    chopsticks[i] = new ReentrantLock(true); // Fair lock
}
```

The code snippet above is how I ensure the solution is starvation-free. The use of ReentrantLock for chopsticks ensures that the locking mechanism is fair. The ReentrantLock class in Java can be configured to use a fair ordering policy, which grants access to the longest-waiting thread.