

# Project 2 - Synchronization

## CECS 326 - Operating Systems

Due Date:	March 16, 2023
Contributors:	@029136612 Twan TRAN @025295541 Bharath VARMA KAKARLAPUDI
Instructor:	Hailu XU

## 1 Lab Summary

This lab involves implementing a solution to the dining-philosophers problem using monitors, either with POSIX mutex locks and condition variables or Java condition variables. The problem involves five philosophers, each identified by a number  $0 \dots 4$ , and each philosopher runs as a separate thread. The philosophers alternate between thinking and eating, and to simulate both activities, each thread sleeps for a random period between one and three seconds.

### 1.1 Objective

The goal of this lab is to provide a solution to the dining-philosophers problem and to demonstrate how to implement it using monitors.

### 1.2 Design

In this lab, we implemented a solution to the Dining Philosophers problem using Java. We used the `ReentrantLock`, `Lock` and `Condition` classes to implement a shared monitor between the philosophers. The `DiningServerImpl` class is responsible for handling the synchronization between the philosophers, while the `Philosopher` class represents each philosopher thread. And lastly, the `Philosopher` class represents each philosopher thread, which alternating between eat, sleep, think.

## 2 Implementation

### 2.1 DiningServer.java (Bharath)

The `DiningServer` interface provides an interface that will be implemented in `DiningServerImpl` class. It contains the methods for the philosophers to communicate with the `DiningServer` object. In the dining-philosophers problem, the `DiningServer` is responsible for coordinating access to the shared resources, which in this case are the forks on the table. The `takeForks()` method is called by a philosopher when they want to eat, and `returnForks()` is called when they are finished eating and ready to return the forks back on the table.

### 2.2 DiningServerImpl.java (Twan)

The `DiningServerImpl` class implements the `DiningServer` interface, which contains the methods called by the philosophers. It uses the `ReentrantLock`, `Condition`, and `Lock` classes from the `java.util.concurrent.locks` package. The constructor initializes the lock, forks, and available boolean arrays, and creates a `Condition` object for each fork using `lock.newCondition()`.

The `takeForks()` method is called by a philosopher when they wish to eat. It first locks the monitor using the `lock.lock()` method, then calculates the index of the left and right forks based on the philosopher number. It enters a while loop that waits for both forks to be available using the `await()` method on each fork's `Condition` object. When both forks are available, it sets the forks to unavailable, prints a message indicating which forks are with which philosopher, and unlocks the monitor using the `lock.unlock()` method.

The `returnForks()` method is called by a philosopher when they are finished eating. It first locks the monitor using `lock.lock()`, then sets both forks to available, signals the Condition objects for each fork using the `signal()` method, and unlocks the monitor using `lock.unlock()`.

This implementation ensures that only one philosopher can hold a fork at a time, and that no two adjacent philosophers hold their respective left forks at the same time, thus preventing deadlock.

## 2.3 DiningPhilosophers.java (Bharath)

The `DiningPhilosophers` class has the main class of the program that starts the dining philosophers problem. It creates 5 philosopher threads and a `DiningServerImpl` object which implements the `DiningServer` interface.

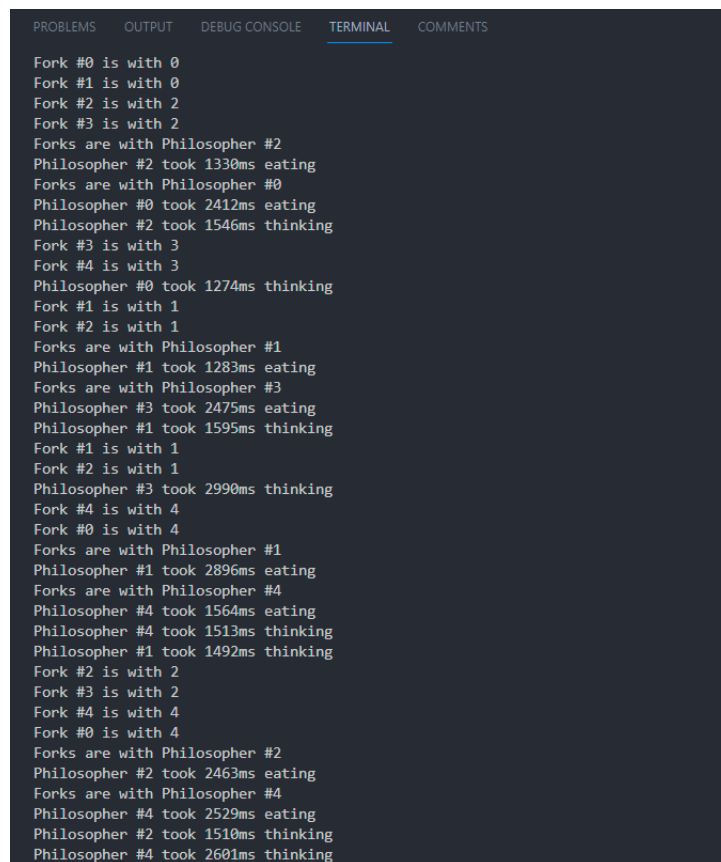
In the `main()` method, it first creates a `DiningServerImpl` object with the number of philosophers as the argument. Then, it creates an array of `Philosopher` objects and assigns each object a unique philosopher number and the `DiningServerImpl` object. Finally, it starts each philosopher thread by creating a new thread object and passing the corresponding `Philosopher` object as the target.

## 2.4 Philosophers.java (Twan)

The `Philosopher` class represents each philosopher thread. The `run()` method is called when the thread starts, and it loops indefinitely, alternating between taking forks, eating, returning forks, and thinking. The `eat()` and `think()` methods sleep for a random amount of time from 1-3s to simulate the philosopher eating and thinking.

# 3 Result

## 3.1 Output



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

Fork #0 is with 0
Fork #1 is with 0
Fork #2 is with 2
Fork #3 is with 2
Forks are with Philosopher #2
Philosopher #2 took 1330ms eating
Forks are with Philosopher #0
Philosopher #0 took 2412ms eating
Philosopher #2 took 1546ms thinking
Fork #3 is with 3
Fork #4 is with 3
Philosopher #0 took 1274ms thinking
Fork #1 is with 1
Fork #2 is with 1
Forks are with Philosopher #1
Philosopher #1 took 1283ms eating
Forks are with Philosopher #3
Philosopher #3 took 2475ms eating
Philosopher #1 took 1595ms thinking
Fork #1 is with 1
Fork #2 is with 1
Philosopher #3 took 2990ms thinking
Fork #4 is with 4
Fork #0 is with 4
Forks are with Philosopher #1
Philosopher #1 took 2896ms eating
Forks are with Philosopher #4
Philosopher #4 took 1564ms eating
Philosopher #4 took 1513ms thinking
Philosopher #1 took 1492ms thinking
Fork #2 is with 2
Fork #3 is with 2
Fork #4 is with 4
Fork #0 is with 4
Forks are with Philosopher #2
Philosopher #2 took 2463ms eating
Forks are with Philosopher #4
Philosopher #4 took 2529ms eating
Philosopher #2 took 1510ms thinking
Philosopher #4 took 2601ms thinking
```

Figure 1: This is the output of the program after it ran.

The output is showing the actions of each philosopher in the Dining Philosophers problem. Each philosopher alternates between eating and thinking, and must acquire two forks to eat, which

represent the resources being shared.

The output also shows the different times each philosopher spends eating and thinking, as well as which forks are being used at any given time. The random sleep times are meant to simulate a more realistic scenario where each philosopher doesn't always behave exactly the same way.

### **3.2 Demonstration (Twan)**

Link: <https://youtu.be/exC1WsAhjoc>