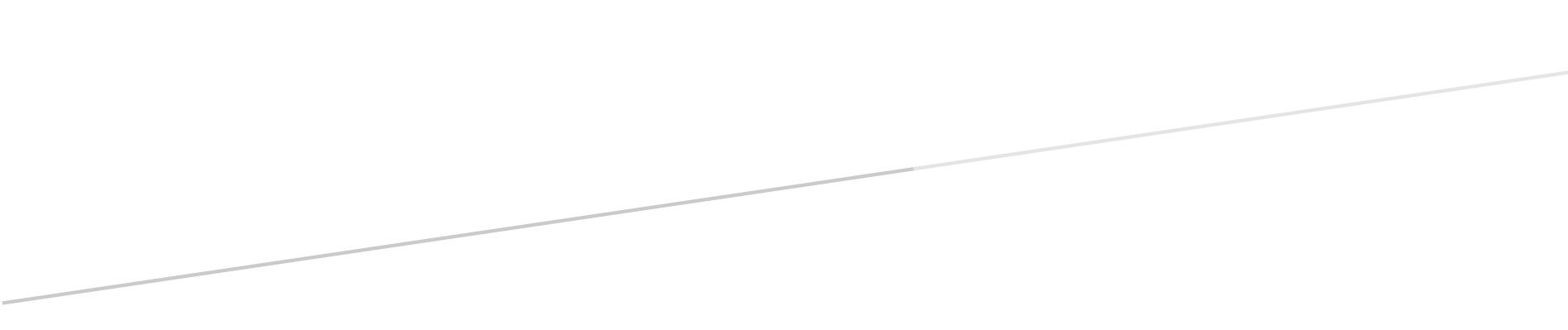
­



**Alex Carbajal**

**Washington State University Tri-Cities**

**May 3, 2024**

Homework 4

**Table of Contents:**

|  |
| --- |
|  |
| 1. Introduction |
| 1.0 Purpose |
|  |
| 1. Problems |
| 2.0 Data Abstraction |
| 2.1 Synchronization |
| 2.2 Experiments |
| 2.3 Deadlock |
|  |
| 1. Design/Solution |
| 3.0 Design Overview |
| 3.1 Design Process |
| 3.1.1 Data Abstraction |
| 3.1.2 Synchronization |
| 3.1.3 Experiments |
| 3.1.4 Deadlock |
|  |
| 1. Experiment Result |
|  |
| 1. Experiment Result Analysis |
|  |
| 1. Analysis of Learning |
|  |
| Appendix |
| Source Code |
| Commands for Running |

**1 Introduction:**

This homework’s focus is on problems related to concurrency and the exploration of solutions to those problems. These problems include concepts such as synchronization, resource sharing and deadlock. The complication used to explore these problems in this document is the dining philosophers problem.

1. Purpose:

The goal of this document is to outline the problems and solutions to the ideas that the dining philosophers problem tackles. Experiments are performed in order to analyze the solutions and their results.

**2 Problems:**

2.0 Data Abstraction: To abstract the data for the dining philosophers problem, I implemented several classes in C++ that work in conjunction to execute the processes of the dining philosophers. The main class that does the work of concurrent programming is the Philosopher class. This class includes several member variables that indicate important data about a Philosopher object and several methods that run the instructions required for the dining problem. Each Philosopher runs in its own thread and works concurrently with multiple other Philosophers. The Philosophers share mutexes between each other to ensure the program meets the expectations of the dining problem and no erroneous experiences occur. The Philosopher class utilizes the Syncro (more information in 2.1) and Chopstick classes to share data between the Philosophers. Each Philosopher has a left and right Chopstick and each Chopstick object includes a mutex to ensure they can only be accessed by one Philosopher at a time. The method that executes all the instructions for the Philosopher is the *run* method, which runs a loop for a specified amount of time that performs the action of the Philosopher (think, take Chopsticks, eat, and release Chopsticks).

2.1 Synchronization: The system that keeps all the Philosophers synchronized and behaving as expected is the Syncro class. The Syncro class is relatively simple and contains a limited amount of variables and methods to keep it quick and easy to manage. The only two variables it has are the *chopsticks* and *states* variables. They both are lists of a size corresponding to the total number of Philosophers. One object of this class is shared between all of the Philosophers in order to keep them synchronized by sharing each Philosopher’s state and the state of the Chopsticks with each other. If a Philosopher want to pick up or put down a Chopstick, they have to perform that action through this class. The class checks the status of each Chopstick being picked up or put down to ensure there is no violation of the rules. A Philosopher can also check the state of other Philosophers through this class (e.g. is another Philosopher eating?). This is how the Philosophers are kept in sync with each other.

2.2 Experiments:

2.3 Deadlock:

**3 Design/Solution:**

3.0 Design Overview: This section goes over the design process and solution to each of the problems listed in section 2. Each solution was designed diligently by carefully following the instructions stated in the homework document in order to ensure accuracy.

3.1 Design Process:

3.1.1 Data Abstraction: In order to design the Philosopher class, I first considered how each of the Philosophers should be able to communicate with one another, since that is a vital part of running a concurrent program smoothly. I decided to have all Philosophers share a single Syncro object which will synchronize all their actions together. For the *run* method, I had each Philosopher perform four actions for a specified amount of time. Those actions are the methods *think*, *take\_chopsticks*, *eat*, and *release\_chopsticks*. Each one of these methods have checks that analyze the state of the Philosopher to determine which action to take and what to print out. The *test* method is the method for checking if a Philosopher can and should pickup both Chopsticks. For my solution, I decided to check if both the neighbors of the Philosopher are eating and to only pickup the Chopsticks if both are currently not eating. This ensures that the correct Chopsticks are picked up only when it is appropriate.

3.1.2 Synchronization: To design the Syncro class, I had to make sure the class had all of the data required to be shared between the Philosophers, because otherwise the concurrency in the program would not work properly. The only data required to be shared is the state of each of the Philosophers and the Chopsticks that the Philosophers are sharing. For the methods of the class, the only ones that I found necessary to implement were getters, setters, and methods to put down and pick up the chopsticks. The methods *putDownChopstick* and *pickUpChopstick* both perform checks to make sure the Chopstick is in the opposite state before being picked up or put down.

3.1.3 Experiments:

3.1.4 Deadlock:

**4 Experiment Result:**

**5 Experiment Result Analysis:**

* 1. **6 Analysis of Learning:**

In this programming assignment we got the opportunity to learn …..

* 1. **Appendix**

**Source Code:**

* 1. **Commands for Running**: A Makefile is provided, so simply type *make* in the directory to build the executable. Run the executable with *./hw4*.