# Concurrency Simulator (Dining Philosophers Problem)

by

Borys Railean

29th April 2025

# Declaration of Authorship

We Students, are aware of the University policy on plagiarism in assignments and examinations (3AS08)

- We understand that plagiarism, collusion, and copying are grave and serious offences in the Institute and We will accept the penalties that could be imposed if we engage in any such activity.
- This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.
- We declare that this material, which We now submit for assessment, is entirely of my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Borys Railean

Date: 29/04/2025

# Table of Contents

# List of Figures

Section 2

# 1 Introduction

“Concurrency in operating systems refers to the ability to execute multiple processes or threads simultaneously, improving resource utilization and system efficiency” (GeeksforGeeks, 2024). It’s the fundamental concept within operating systems. It presents complicated challenges associated with synchronisation and resource sharing.

Discussing concurrency, the Dining Philosophers Problem, made by Edsger Dijkstra in 1965, was a well-known classical problem in the synchronisation of concurrent processes. The Dining Philosophers Problem highlights the adversity of resource sharing (chopsticks) with multiple processes (philosophers) without the cause of starvation or deadlock (Lehmann, Rabin, 1981).

This report explores in-depth detail on the Philosopher's Problem, highlighting various solutions, including the lefty-righty and the 4-chair. We will go into details about the solution, exploring aspects such as mutexes and semaphores and their conclusiveness in the causes of deadlock or starvation.

We'll introduce these aspects through a detailed introduction and background, followed by a detailed layout, continued with our solution, results and analysis, and conclusion, finalised by our personal reflections and planning and management.

## 1.1 Background

### Concurrency and Synchronisation Issues in OSes

A way to enhance the performance and responsiveness of a system when developing operating systems is concurrency; however, it can also create issues (e.g., deadlocks) when threads access shared resources (or a shared resource) concurrently and unsafely. Access to shared resources needs to be synchronised for consistency and correctness. Mutex locks and semaphores are common implementations of synchronisation used to protect a critical section and prevent issues uniquely through serialisation of access to shared resources. Such synchronisation mechanisms can assure reliable and predictable system performance and data integrity in a multi-threaded environment (Silberschatz, Galvin, and Gagne, 2018).

### Mutex Locks

Mutex locks, standing for mutual locks extension, are synchronisation functions used to avert simultaneous possessions of resources that are shared by multiple threads in multitasking programming (Patro, 2023). Mutexes are split into three different processes. Mutex lock, mutex unlock and mutex wait. In the dining philosopher's problem, each chopstick is represented by a mutex. When the philosopher is holding a single chopstick awaiting their turn to eat (wait). They must acquire both left and right chopsticks (lock). When they finish eating, they let go (unlock) the chopsticks, allowing for the next philosopher to use them.

### Semaphores

A semaphore is a synchronisation tool used in concurrent programming to manage access to shared resources (Comment et al., 2025). Semaphores are better than mutexes because they allow more threads to access a resource. They can attempt to access an instance, and if it's not available, they try other instances, unlike mutex locks. In the thread mentor-based solution, we have 3 semaphore usages in the graph. SS, SW, and SE are used, and we'll explain more about them. In the philosopher's problem, we use an initialized semaphore to change the order of eating to allow for better results.

### ThreadMentor

ThreadMentor is a pedagogical tool built to help teach multithreaded and concurrent programming. It enables students to look at the management of threads and how to use synchronization primitives like mutex locks and semaphores. The tool's visual representation allows students to see thread states, state changes, and the effects of synchronization on those states, which can help make abstract concepts like race conditions and deadlocks more approachable (Shene, n.d.).

In the field of adaptive learning environments to teach concurrency, ThreadMentor is an example of a system that can aid student understanding by providing immediate interactive feedback and structured educational functionality (Lönnberg, 2006).

## 1.2 The Dining Philosophers Problem

The Dining Philosophers Problem, introduced by E. W. Dijkstra and later explored using ThreadMentor (2001), involves five philosophers sitting at a table with a plate of food in the middle, there are also five chopsticks between them. Each philosopher switches between thinking and eating. For a philosopher to eat, they have to pick up the two closest chopsticks to them. However, if all of the philosophers pick up only one chopstick at a time, none of them can eat, this causes deadlock where all of them are held up for one to finish with a chopstick.

## 1.3 Outline/Layout of your Report

Whilst reading through this report, the reader will gain insight into the fundamentals and detailed side of the philosopher's problem, our solutions, and methods in section 2 and 3. We will also dive into extensive background research in section 1, allowing full understandability and figures alongside. This report will be carried out with a considerable amount of research, including references proving our arguments. We will then conclude with our personal reflections and time spent concluding this report.

# 2 The Solution

Shene, Dr. C.K. (2001) For the solutions we will be addressing the two topics of lefty and righty. We do a comparison of the two solutions showing the different types of problems that can occur between the two along with the differences between the two and which is more logical with the Pros and Cons between them. Both have different problems with each other, which allows for an easier comparison with the different outcomes they come out with and what logics they serve within the solution.

## 2.1 Lefty Righty

The dining philosopher problem, five philosophers sit around a table, each needing two chopsticks. One of their lefty and righty hand to eat. If every philosopher picks up the left chopstick first and then waits for the right, a deadlock can occur where no one can proceed.

To avoid this, the Lefty and Righty solution allows for a simple change.
All the philosophers are ‘lefties’ so they pick the left chopstick, except one philosopher who is a ‘righty’ who picks up the right chopstick first. This will allow the philosophers to eat and finish at the same intervals. It may not allow philosophers to have equal amounts of eating intervals, and some philosophers may not get the chance, but we see this, and they do eventually eat, maybe in larger amounts some may not be able to eat as much due to the larger quantity which disallows eating equivalency.

How the Mutex is used:
Each chopstick is protected by a mutex. There are mutex locks, mutex unlocks and mutex wait. The philosopher must acquire the mutex for both the adjacent chopsticks to proceed eating. This allows for a consistent but slightly altered order of lock which allows deadlock to be avoided. (lefties pic left then righty, righty picks righty then left).

Why it Works:
The key idea is that not all philosophers are competing for the resources that being the chopsticks in the same order, which prevents circular wait.

An Example:
If all the philosophers grabbed the left chopsticks first, they would each hold one and wait forever for the right, that is a deadlock.
With one righty however, that philosopher breaks the cycle and may acquire both chopsticks, eat, then release them for another philosopher to use.

Problems & Faults that may occur:
Deadlock is prevented, however,

- Starvation is still possible. A philosopher might never get both chopsticks if the Neighbours are faster or constantly eating.
- Unfairness can occur as some philosophers may eat way more compared to others sitting around.
- The Lefty and Righty does not scale well with many philosophers involved and only one is a ‘righty’. (Shene, Dr. C.K. (2001).

## 2.2 4 Chair Solution

Shene, Dr. C.K. (2001) The Four Chairs solution is a deadlock prevention mechanism that, more-or-less, limits the number of philosophers that can sit at the table at the same time, switching one and adding the waiting on in.

This approach limits the number of philosophers who can try to eat at the same time to four out of five. At least one philosopher is always guaranteed to be able to access both chopsticks without any problems.
This simple change breaks the circulation wait condition, which is one of the necessary conditions for deadlock.

Semaphores:
Four Chairs is initialized to 4, representing four available seats at the table.

```
wait(FourChairs);            // Request permission to sit
wait(left_chopstick);        // Pick up left chopstick
wait(right_chopstick);       // Pick up right chopstick

eat();

signal(right_chopstick);     // Put down right chopstick
signal(left_chopstick);      // Put down left chopstick
signal(FourChairs);          // Leave the table
```

Explanation:
wait(FourChairs) ensures only four philosophers may sit at a time.
Each chopstick is still protected with a binary semaphore or mutex.
Once finished, the philosopher releases both chopsticks and the seat too.

Benefits -

- Deadlock Prevention:
  Circular wait is eliminated, since it is not possible for all of the philosophers to be holding only one chopstick and waiting for the second.

- Simple Implementation:
  Adding one counting semaphore does not require you to alter the mutual exclusion operations for the chopsticks themselves.

Problems and Faults that can Occur -

- Starvation:

Some philosophers may be starved because others may continuously enter and exit the table before them, we discovered this when running it on Thread Mentor a few times, it could lead to the philosopher being terminated as shown on the hierarchy.

- Reduced concurrency:
  One philosopher can always be waiting, even if chopsticks might be available to him.

- Fairness Problems:
  There are no guarantees of fairness or equality for eating to each of the philosophers like with Lefty and Righty. The semaphore will not guarantee access, only limited access. (Shene, Dr. C.K. (2001)

## 2.3 Thread Mentor

To run all this, we have to first access Thread Mentor which is where we run our codes. It is a simple and effective way we learned by downloading the application in our lab which helped us to run the appropriate solutions and problems we have selected.

Within thread mentor there are main aspects used with it to help view the code. Thread Status allows you see the philosophers and what they're doing in each scenario when the code is running and to see any problems that may arise with the solution. It enables all the different options that the threads are going through in the headers above them when the code is being run. The history graph is the main option that enables to the user to see what all the philosophers are doing in real time and shows the different options to be enabled to viewed for how they are passed around the philosophers (the chopsticks) it is displayed as lines. There is further information on the graphs being used in part 3.

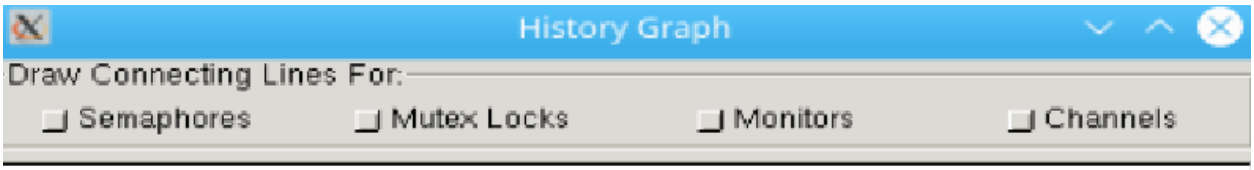



Figure 2.1

The thread hierarchy is like the thread status but a little easier to understand as it shows all the way the philosophers are being dealt with in the solutions. There is not much of a difference however and they both do the same job by displaying the information they need to.


Figure 2.2

## 2.4 Makefile

Along with the threadmentor the makefile is very important as for each solution used it must contain the correct files to run. There are files already generated on a website Multithreaded

Programming with ThreadMentor: A Tutorial. Simply downloading them and on our lab sheet we download the makefile which allows us to fully convert the downloaded files into the threadmentor.

While in the konsole and in the appropriate folders for the downloaded files when use the command $make, which will generate all the files onto the threadmentor. Afterwards we access the solution using the appropriate commands to run the solution on threadmentor.

## 2.5 Tags

Shene, Dr. C.K. (2001) In ThreadMentor, tags are unique integers assigned to synchronisation primitives such as semaphores and monitor entries. Tagging allows for a differentiation of objects, so that in the graphical simulation interface of ThreadMentor we can tell one semaphore from another. When multiple threads are working with shared resources, the tags provide additional information, making it clear which thread is working with which resource and when. This is especially effective when visually tracking complex interactions of threads to identify bottlenecks or to discover synchronisation problems, for example deadlocks. Without tagging, the ThreadMentor visualization would have lost its meaning, as it would be impossible to accurately label or associate each semaphore or monitor respectively. (Shene, Dr. C.K. (2001).

### Lefty and Righty

Shene, Dr. C.K. (2001) In the Lefty and Righty variation of the Dining Philosophers, each chopstick is modeled as a binary semaphore. For ThreadMentor to accurately show the interactions between philosophers and chopsticks, you must invoke setTag() on each semaphore object for the chopsticks to give them unique tags. For example, if we have five chopsticks (with the chopsticks being indexed from 0 - 4), the chopstick semaphore would have a tag value of 0 - 4 to match the index of the chopstick we are picking up.
This enables ThreadMentor to correctly visualize any of the events for a scholar picking up or putting down a chopstick. Tags are necessary in this solution to demonstrate the impact of the asymmetric fork-picking behavior, which allows users to see how deadlock is avoided. (Shene, Dr. C.K. (2001).

### Four Chairs

Shene, Dr. C.K. (2001) In the Four Chairs solution, there is an important semaphore called Four Chairs which will limit how many philosophers can attempt to eat at the same time.

We use something called setTag() in the ThreadMentor implementation of the solution, so it gives this semaphore a unique tag to help visualize what is going on in the animation. Each chopstick is also a semaphore and needs to have a tag too.
The tags help ThreadMentor visually indicate when a philosopher sits down, picks up the chopsticks, eats and ultimately puts everything back. It would be difficult to figure out which resources are being used, and the program is operating correctly to avoid deadlock without the tags. (Shene, Dr. C.K. (2001).

# 3 Results and Analysis

## 3.1 Lefty and Righty

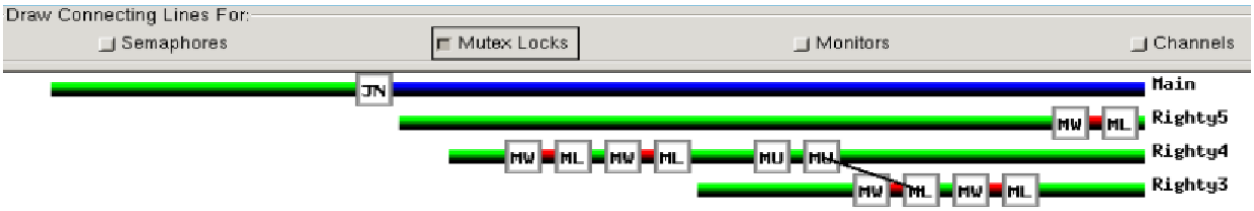



Figure 3.1

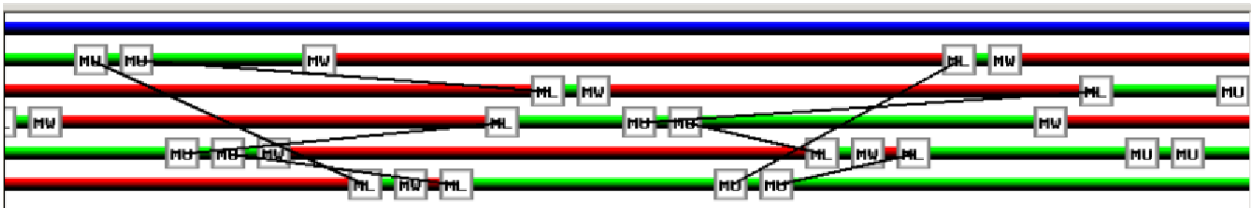


Figure 3.2

This screenshot provides a visual representation of the timeline of thread interactions in ThreadMentor so you will see the horizontal lanes associated with a thread (the philosophers) or a semaphore (the chopsticks). Each lane has coloured lines and labelled boxes that suggest the actions they have taken to synchronise. The letters like "P/U" (=Pick Up/Put Down) are approximations of the events that involve the chopsticks being accessed. The thick red, green and black lines represent the threads waiting for chopsticks or using chopsticks that allow tracing when each philosopher is trying to eat. The Lefty/Righty model gets around deadlock because one philosopher (Lefty1) picks up chopsticks in the reverse order, so it introduces asymmetry which removes the mutual waiting as a cycle. In this visual we can see that while some philosophers are waiting, the system proceeds without becoming deadlocked.

## 3.2 Hierarchy

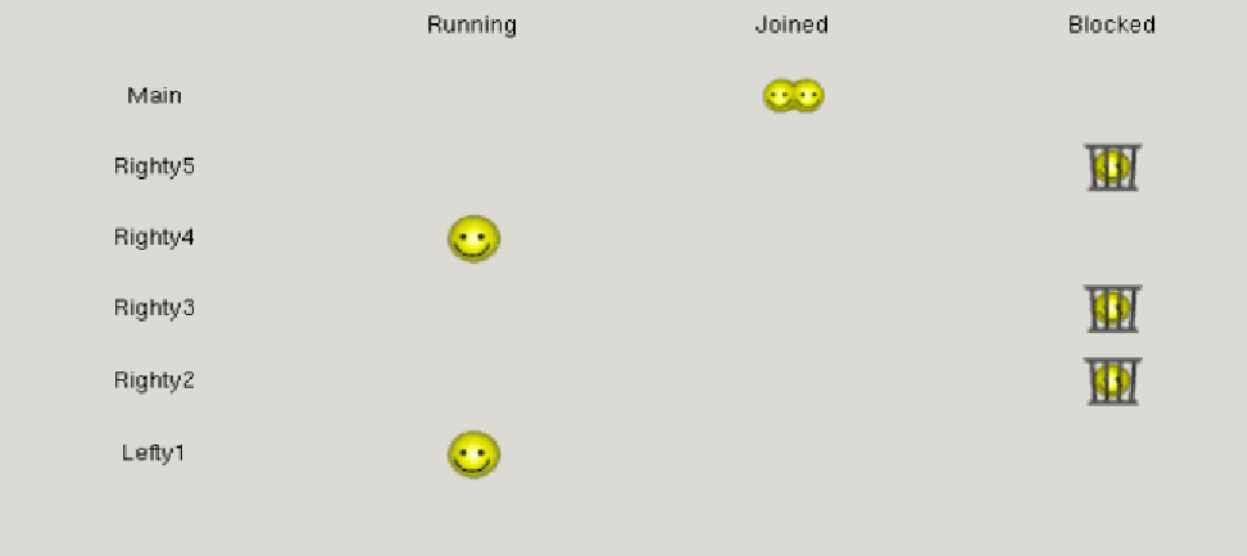



Figure 3.3

The hierarchy screenshot shows a simple representation of the thread states while they are executing. Each row represents a philosopher thread, and the columns represent that philosopher's state: Running, Joined, or Blocked. Righty4 and Lefty1 are currently in the running state, which means the philosophers are eating food. Righty5, Righty3, and Righty2 are in the Blocked state, which means they are waiting for chopsticks to be free. This screenshot shows that philosophers will still block, but they can still make progress with the asymmetric Lefty and Righty philosophers. This is a good indication that the system has avoided deadlock and starvation.

## 3.3 Main Thread Mentor on Konsole:

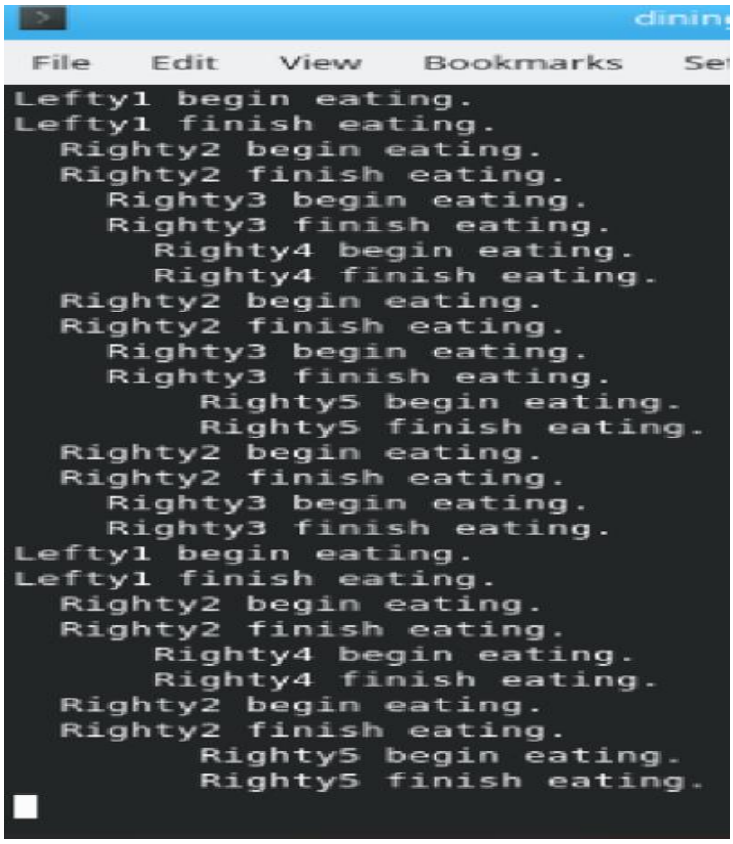



Figure 3.4

The console output shows when the philosophers begin and finish their eating, and the number of times that beginning eating and finishing eating appear for each philosopher indicates that things are fair across the threads. For instance, Lefty1 can repeatedly begin and finish eating; Righty2, Righty3, Righty4, and Righty5 are also able to repeatedly begin and finish eating. The repeated instances of begin eating and finish eating by a thread illustrate that the system is working as we expect; no philosophers are starving, and, eventually, all the threads will access the resources. This corroborates the visual representations we see in 3.1 and 3.2 and demonstrates that the algorithm allows for efficiency and fairness.

## 3.4 Four Chairs

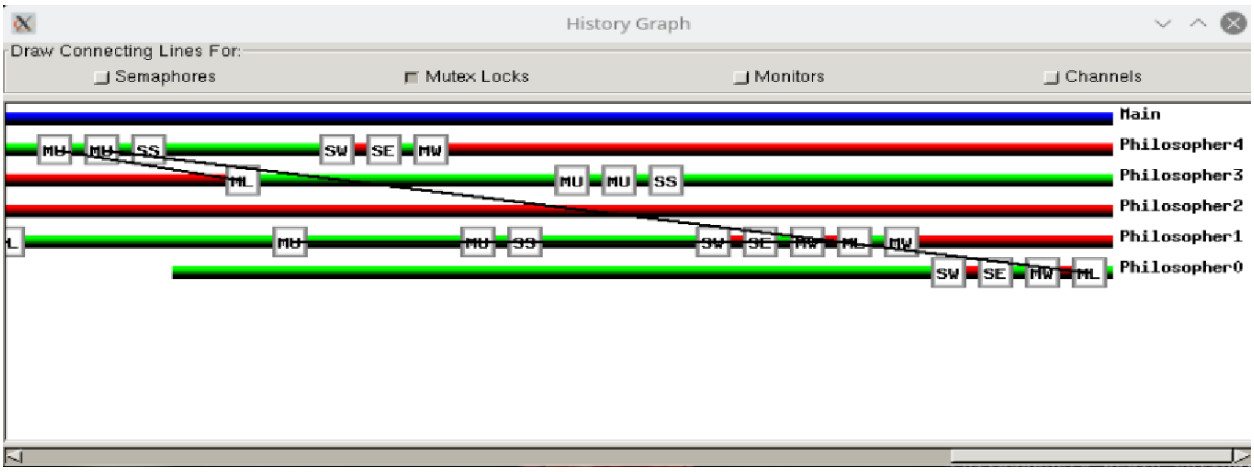



Figure 3.5

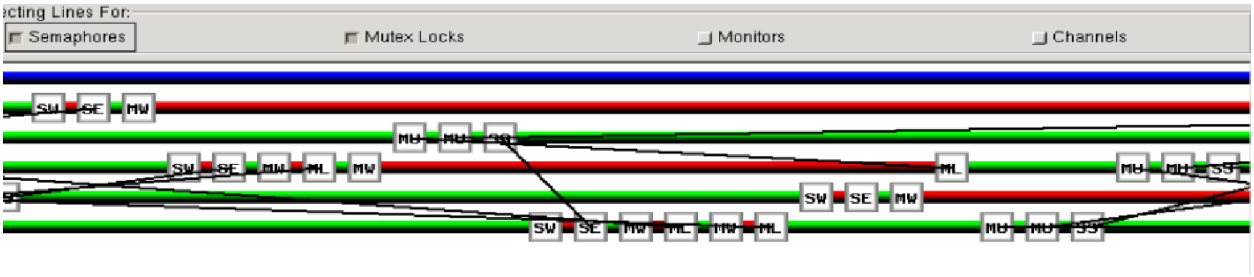



Figure 3.6

This screenshot provides a view of the thread activity created using the Four Chairs solution in ThreadMentor. Under this solution, a maximum of four philosophers would be able to sit and attempt to eat at the same time. It stops the full five from holding onto one fork and waiting for another (the intended deadlock scenario). While this may do the trick, Four Chairs, while a tolerable compromise, may be more conservative than Lefty/Righty. Philosophers, as indicated by repeated segments on red, are often blocked, and even when forks are available, can be made to wait for no reason, except for the limit to the number of philosophers. When compared to the asymmetric solution of Lefty/Righty, Four Chairs substitutes efficiency for safety, limits concurrency, and responds to limit the risk of deadlock.

## 3.5 Hierarchy

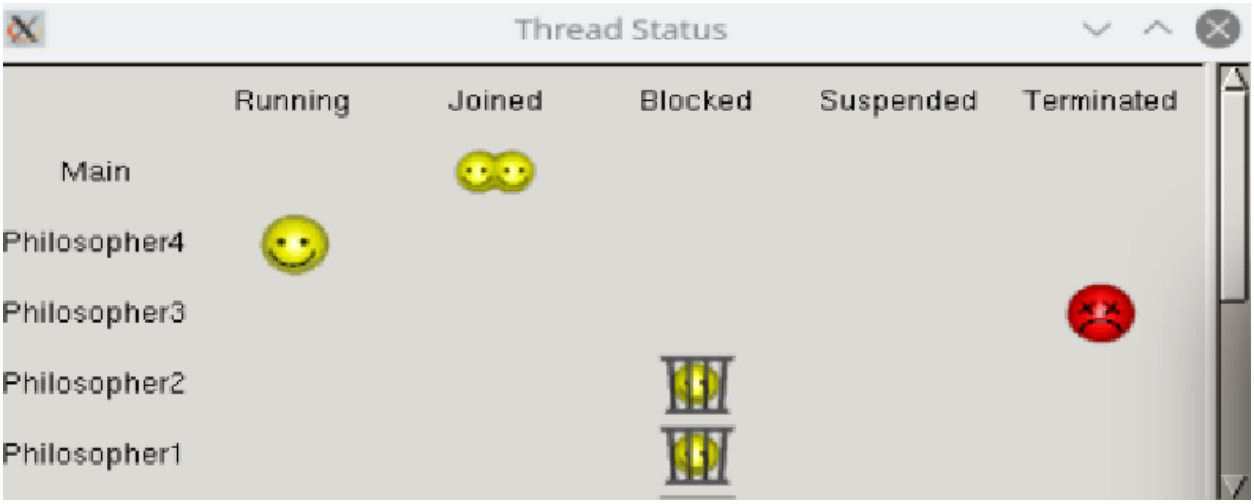


Figure 3.7

This screenshot illustrates the thread hierarchy and status under the Four Chairs solution. We can see that each row represents a philosopher thread, and each column displays the state: Running, Joined, Blocked, Suspended, and Terminated. Philosopher4 is running (eating), while Philosopher1 and Philosopher2 are blocked, waiting for chopsticks. The most troubling state is for Philosopher3, which is in a terminated state (meaning this thread has died because of starvation), e.g., the other philosophers are eating multiple times before Philosopher3. This exposes a significant flaw of the Four Chairs solution. It precluded deadlock by limiting the number of philosophers who could attempt to eat at one time, but it did not provide fairness across philosopher threads. Philosopher3 would be consistently allowable starvation with respect to the run time of the threads. The ability for a philosopher to “die” from starvation condition is simply unacceptable in a proper synchronisation solution and is in stark contrast to the Lefty/Righty model, which both prevents deadlock and starvation.

## 3.6 Main Thread Mentor on Konsole:

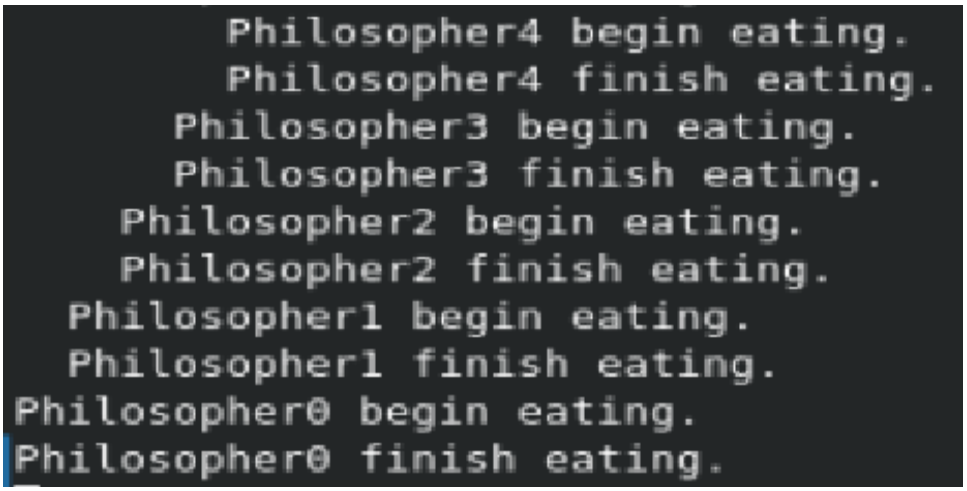


Figure 3.8

In the screenshot we see the run-time log of the philosophers using the Four Chairs solution. The log shows Philosophers 4 through 0 were all able to begin and finish eating at least once, which suggests progress and an avoidance of deadlock. However, in combination with the visual timelines in 3.4, we observe that thread execution is highly serialised; only one philosopher can eat at a time, and threads wait a considerable amount of time to gain access. This indicated starvation given that Philosopher3 is shown terminated in 3.5, which is a frightening indication of starvation. In 3.6, the console generates no comparable starvation output but offers no evidence of fair distribution or fairness in scheduling. Moreover, comparisons with the Lefty/Righty model with its notion of parallelism and avoidance of starvation make Four Chairs lacking in efficiency and fairness.

# 4 Conclusions

The Lefty-Righty solution does a good job preventing deadlock because it changes the pickup pattern of the philosophers. Also, instead of each philosopher picking up only their left chopstick first, we made one philosopher pick up their right chopstick first. Therefore, it's much harder to get stuck in a cycle deadlock situation, and it allows things to keep moving forward. Starvation can still exist, if some philosophers are faster, some philosophers may rarely get to eat. As time goes on, this may result in our solution being inefficient.

We decided that the 4-Chair solution worked better because only 4 philosophers were allowed at the table at once. There is always someone who can get their chopsticks together and eat. That solution completely removes deadlock and minimizes the chances of starvation.

One of the things we liked about the Lefty-Righty method was how simple it is to implement. Essentially, we only changed the behavior of one philosopher, so it was quite simple to code and test. However, it does not scale very well. For example, in a larger group of philosophers, just having one righty philosopher would not be enough to balance the rest of the problem.

The 4-Chair method is a little more challenging because you have to control how many people are present at the table and when. It is more equitable and scales better.

# 5 References

- Shene, Dr. C.K. (2001) *Threadmentor: The dining philosophers problem: The lefty-righty version*. Available at: https://pages.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-left-right.html
- Shene, Dr. C.K. (2001) *Threadmentor: The dining philosophers problem: The lefty-righty version*. Available at: https://pages.mtu.edu/~shene/NSF-3/e-Book/SEMA/TM-example-philos-4chairs.html
- GeeksforGeeks (2024) *Concurrency in operating system*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/concurrency-in-operating-system/
- Lehmann, D. and Rabin, M.O., 1981, January. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (pp. 133-138).
- SILBERSCHATZ, A., GALVIN, P.B. and GAGNE, G., 2018. Operating System Concepts, <https://os.ecci.ucr.ac.cr/slides/Abraham-Silberschatz-Operating-System-Concepts-10th-2018.pdf>
- Lönnberg, J., 2006, *Adaptive learning environments for concurrent programming*, *ConcurrencyALE-2.pdf*. <https://www.cs.hut.fi/~jlonnber/ConcurrencyALE-2.pdf>
- Shene, C.K., 2001, *ThreadMentor: An Overview*.<https://pages.mtu.edu/~shene/NSF-3/e-Book/FUNDAMENTALS/TM-overview.html>
- Comment *et al.* (2025) *Semaphores in process synchronization*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/semaphores-in-process-synchronization/
- Patro, D. (2023) *Mutex locks*, *Tutorialspoint*. Available at: https://www.tutorialspoint.com/mutex-locks

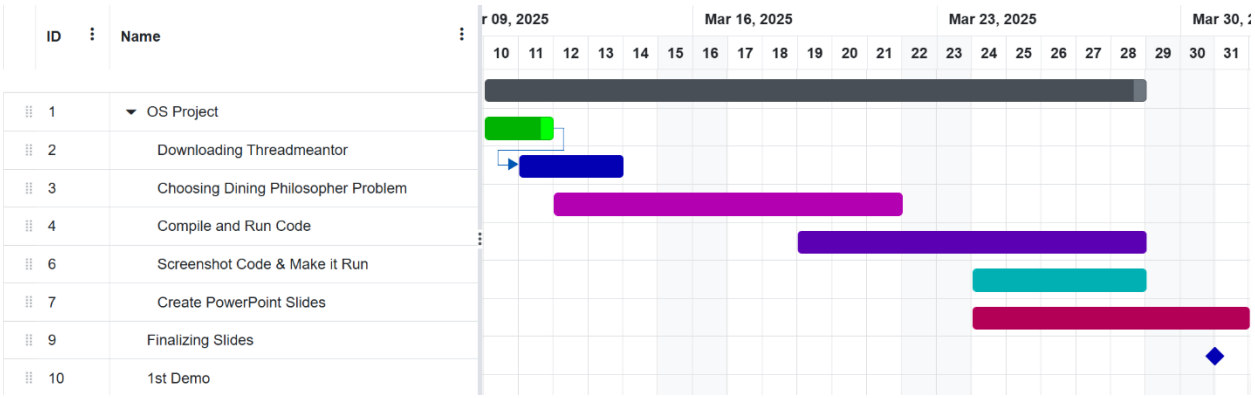

# 6 Appendix: Personal Reflections

## 6.1 Borys Railean:

I personally believe all went well throughout the report. I felt very involved, and I believe my work was fair and up to standard. I took the role of giving out the topics and keeping up with each member, ensuring all work is completed. I completed most of section 1; I compiled all the research together into a document, completing all requirements such as declarations, table of contents, list of figures, etc., and my personal self-reflection for this project. I believe my research was valid, using multiple references on the needed sections. The only thing I'd change is when keeping up with my team members, to give out specifications such as word count, as we ran into this issue towards the end of the report. Overall, everything else went well, and I learnt a lot about concurrency and synchronisation in the philosopher's problem through operating systems.

# 7 Appendix: Project Planning and Management

## 7.1 Gantt Chart:




## 7.2 Description of First Gantt Chart:

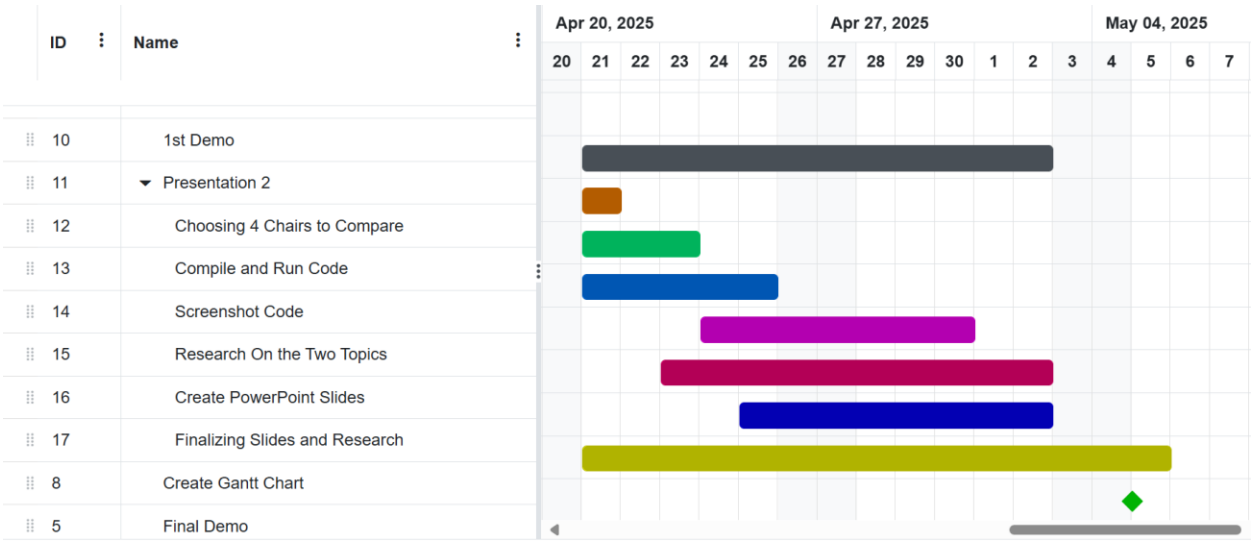
This showcases the start to finish of our project along with the different important steps we took into making our project for our first presentation and then lastly demonstrating. We started of with Thread Mentor to use the dining philosopher solutions. Without it we are unable to run and showcase the code for the presentation.

We then selected the Lefty and Righty problem as it looked the most simple and easy to understand.

It then took a little bit to investigate the code and run it because we were facing some difficulties getting the code up and running for us to view it. After managing to run the code we to screenshots of the necessary parts that were run on Thread Mentor to include it in our presentation.

While looking into the code and screenshotting what we need, we were creating slides for PowerPoint to present and learning them off a few days prior to the day of the demonstration.

## Gantt Chart:




### Description of Second Gantt Chart:

We started to look into our second demo during the break, and we had a specific idea on what we wanted to accomplish. Deciding on the comparison of the Four Chairs and the Lefty and Righty solution along with the different problems that occur with each solution. We got the code and ran it, but this was shorter compared to the first demo due to the knowledge of knowing how to run it and having more understanding of how to operate it.

Taking the important screenshots, it was easier to get them because of how differently it ran compared to the Lefty and Righty solution. There were more things added to the Four Chairs which allowed for more research for the project and allowed us to make more comparisons.

The research was a little basic, but we did also go in depth with a few things about the Four Chairs. We also had to do further research on Lefty and Righty and looked more into the good and bad sides along with the problems that can occur with the solution.

Adding our specified research into the PowerPoint slides for presentation with the code and screenshots for the solutions. We were finalizing all this to the last day to make sure everyone in the group understood the research done by everyone, even if they didn't specifically do anything on it.

## 7.3 How you managed the project on a Week-to-Week Basis:

For every presentation, we managed our time well because we spread the work out each day and week. This helped us stay on track and get our work done without feeling pressed. We took care to manage our time so that we spent enough time to create the slides but also prepare for the demonstration that went along with the slides. So, each member of the team knew what was completed in the project, the topics we researched, and what we were demonstrating. Because of the way we planned our time, we were able to finish researching, synthesizing, and gathering all our knowledge with no issue. By the time we were getting into the demo, we were familiar with what we represented and very confident in what we were presenting.