A blue and white logo

Description automatically generated

A Study of the Synchronisation and Concurrency Issues in the Dining Philosophers’ Problem completed using the ThreadMentor Visualisation Tool

**Submitted by: Habiba Nour, B00151078**

**Steven Kelly, B00150588**

**Rochelle Mullen, B00156311**

**Piotr Momat, B00156112**

**Submission date:** *25/April/2024*

Declaration Page:

*We 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.*

Author: Habiba Nour Dated: 25/April/2024

Author: Steven Kelly Dated: 25/April/2024

Author: Rochelle Mullen Dated: 25/April/2024

Author: Piotr Momat Dated: 25/April/2024

**Abstract Page**

In this report, we are going to talk about one of the most interesting topics in operation systems, which is “A Study of the synchronisation and Concurrency Issues in the Dining Philosophers’ Problem Completed Using the ThreadMentor Visualization Tool." We are going to discuss what concurrency and synchronisation mean and what issue they could cause, the dining philosopher problem and its solutions, mutex locks and semaphore, which solution we think is the best, and how it works in the result and analysis section. Finally, we will discuss the advantages and disadvantages of our solution and draw a conclusion from what we discussed in the report.

Contents

[1 Introduction 5](#_Toc164957559)

[1.1 Background 5](#_Toc164957560)

[1.2 The Dining Philosophers Problem 9](#_Toc164957561)

[1.3 Outline/Layout of your Report 10](#_Toc164957562)

[2 The Dining Philosophers Problem with Four Chairs 10](#_Toc164957563)

[2.1 Theory/How it works 10](#_Toc164957564)

[3 Results and Analysis 13](#_Toc164957565)

[4 Conclusions 17](#_Toc164957566)

[5 References 18](#_Toc164957567)

[6 Appendix: Personal Reflections 19](#_Toc164957568)

[7 Appendix: Project Planning and Management 21](#_Toc164957569)

# Introduction

In the following report we will talk about concurrency and synchronisation issues in operating systems and the Dining Philosophers Problem, while using the Threadmentor tool to visualise these problems. Problems with concurrency and synchronisation are prevalent in operating systems and these problems need to be solved for the smooth running of the users OS. We will go into these solutions and problems with code running in Threadmentor to see these issues in real time. The Dining Philosophers Problem is a classic issue in computer science and is a great thought experiment to see how threads and processes interact with one another in an operating system.

## Background

**1.1.1 C****oncurrency and Synchronisation Issues in OSes**

**Concurrency:** Concurrency refers to the ability of an Operating System to execute tasks simultaneously. It happens when multiple process threads are running at the same time. The running threads communicate through shared memory or message passing. This can result in sharing resources and will lead to problems like deadlocks and resource starvation. **Processes** are instances of a program that can execute independently whereas a **thread** is a process that shares the same memory as its **parent process.** Concurrency in operating systems is a key feature that can enable efficient use of system resources, responsiveness, scalability, and parallelism. However, it can introduce some challenges that require careful design and implementation of OS’ and applications. *(Linkedin, 2024).*

Processing in operating systems can be the following types:

1. Independent processes: These cannot affect other processes or be affected by them. Execution result depends on the input state, the execution result will always be the same for the input and termination will not terminate another process.
2. Cooperating processes: These can affect or be affected by other processes in the system. Execution depends on relative execution sequence and can’t be predicted in advance (non- deterministic), the execution result will not always be the same for the input and the termination may affect other processes. *(Geeksforgeeks, 2020)*

**Benefits:**

* **Performance:** One of the best uses of concurrency in operating systems is performance improvement. As multiple tasks are being executed simultaneously, this improves the overall performance of the system as using multiple processes or threads will reduce overall processing time in the system.
* **Resource utilization:** This allows resources that are not currently being used by an application and can utilize them for a running application.
* **Response time:** With using concurrency it allows multiple application to run at once. Without this, an application will have to wait for completion before another to run. *(Geeksforgeeks, 2020)*
* **Scalability:** Concurrency will improve scalability of a system by allowing it to handle multiple tasks without impacting performance. *(Tutorialspoint, 2024)*

**Disadvantages:**

* **Deadlock:** Deadlock is a situation where multiple processes/ threads are unable to proceed due to them both waiting for each other the release a resource. This is less than desirable in operating systems as it will halt performance of the system.

*(Theknowledgeacademy, 2024).*

* **Starvation:** This is a situation where a thread is not able to gain access to shared resources and cannot make progress. This can happen when shared resources are made unavailable for long periods by “greedy” threads. *(Oracle, 2024)*
* **Blocking:** This is a system call where the process execution is blocked until the requested operation is complete (can be used in synchronous I/O or asynchronous I/O)

**Synchronisation:** According to Ravulakollu (2024), synchronisation refers to the coordination and control of multiple processes sharing resources and data with an operating system. Synchronisation is essential for preventing conflicts, ensuring consistent multitasking, and maintaining multitasking order. Synchronisation helps maintain consistency by ensuring only a single process can change shared memory at a time. Synchronisation has many solutions tied to it like semaphores and mutex locks.

Programs will have 4 elements**: An entry section, critical section, exit section and remainder section.**

* **Entry section:** Entry of a process
* **Critical section:** Ensures only one process modifies shared data
* **Exit section:** When other processes enter shared data after another processes executes. Handled by the exit section.
* **Remainder section:** Remainder of code that is not categorized.

*(Image sourced from: https://www.geeksforgeeks.org/introduction-of-process-synchronization/)*

A **race condition** is when multiple processes race to say their result is correct when trying to modify a shared resource. As many processes use the same data, the result of the process depends on the order of their execution *(Yadav, 2022)*. This mostly occurs within the **critical section** of a process. This means the result of multiple thread executions may differ based on the order in which they are executed. This means results can vary when running the program at different times.

**Solutions to the critical section problem:**

* **Mutual exclusion:** When a process is executing in the critical section it means no other processes can execute in their critical sections. *(Geeksforgeeks, 2024)*
* **Progress:** If a process is executing in critical section and a process wants to enter its critical section, only processes that are not executing in remainder section can decide which will enter critical section next. This cannot be postponed indefinitely.
* **Bounded waiting:** Needs to be a bound on the number of times a process can execute in its critical section, after another process requests to enter its critical section and before the request is accepted. *(Educative, 2024)*

**1.1.2 Mutex Locks**

According to Shene(2001), mutex locks which is the short for mutually exclusive locks is one solution to the dining philosopher problem. These locks work as a pair in that if one is locked it cannot unlock until the other has locked too and are seeking to unlock. If only one lock is acquired the philosopher must wait to acquire the second lock and cannot unlock unless it has both. With the locks the philosopher locks the chopstick that they pick up and will then unlock the chopsticks. Each philosopher will pick up their right chopstick and if they can, they will pick up the left chopstick. When they finish and put down their chopsticks other sitting nearby that are waiting to acquire chopstick will be able to lock those chopsticks. One way to ensure there is no deadlock with this method is to make one of the philosophers to firstly pick up the chopstick that is on their left while the rest will pick up the chopstick on the right. You can also make it so that all the philosophers pick up their left chopstick bar one. The issue with mutex locks is that it can lead to starvation as those waiting on the chopstick may be waiting for some time especially if the philosopher that picks up the chopstick that is opposite to the rest eats and thinks quickly. As they are quicker it means they will have more access to resources than the others as they will have to wait to acquire their second chopstick. Mutex locks work using a lock method and an unlock method. The handedness of the philosopher can be assigned too within the code.

**1.1.3 Semaphores:**

According to (Shene, 2001), semaphores can be viewed as an extension to mutex locks. “Semaphores used to solve the critical section problems and to active process synchronisation in multiprocessing environment”. It has two methods “Wait” and “Signal” (atomic), a private integer counter, and a private queue (of threads).

Two possibilities when “Wait” is executed by a thread:

- The counter of S (Semaphore) is positive: in this case, the counter is decreased by one and the thread resumes its execution.

- The counter of S is zero: in this case, the thread is suspended and put into the private queue of S.

Two possibilities when “Signal” is executed by a thread:

- The queue has no waiting thread: the counter of S increased by one and the thread resumes its execution.

- The queue of S has a waiting thread: in this case, the counter of S must be zero. one of the waiting threads will be allowed to leave the queue and resume its execution.

When the element is one that means that the element could be used but if it is zero that means that the element must wait. All the elements are supposed to be initialized to 1. (Neso Academy, 2021)

As shown above that “Wait” and “Signal” are atomic which means once the activities of “Wait” start, they will continue with no interruption. There are many steps for “Wait” and “Signal”, these steps are considered as a single non-interruption instruction. The same thing applies to “Signal”. If more than one threads try to execute “Wait” or “Signal”, only one of them will succeed. (Shene, 2001)

**How “wait” and “Signal” are working**

“Wait”: out of many threads, only one thread can successfully execute "Wait." This will cause the counter to decrease by 1 and enter the critical section. Once the thread enters the critical section, the counter becomes 0, and as a result, all subsequent attempts at executing the wait will be blocked. (Shene, 2001)

“Signal”: when the thread exits the critical section, it executes "Signal." If there are threads waiting, only one of them will release and enter the critical section, but the counter will not increase by 1, the “Signal” will not increase, and the “Wait” will not decrease. If there are no waiting threads, the execution of “Signal” causes the value of the counter to increase by 1. Then the next thread that executes “Wait” can enter the critical section. (Shene, 2001)

This type of semaphore is called a “binary semaphore,” and the counter is either 1 or 0. Binary semaphore can be used to control access to a single resource which can be used by one process. (Neso Academy, 2021)

**1.1.4 ThreadMentor:**

One of the tools that we are using for learning multithreaded programming in operating systems is a ThreadMentor. It provides a visual representation and a user-friendly approach while studying. This is what Steve Carr, Jean Mayo and Ching-Kuang Shene said in their publication called “ThreadMentor: A pedagogical tool for multithreaded programming”:

“ThreadMentor is a multiplatform pedagogical tool designed to ease the difficulty in teaching and learning multithreaded programming. It consists of a C++ class library and a visualisation system. The class library supports many thread management functions and synchronisation primitives in an object-oriented way, and the visualization system is activated automatically by a user program and shows the inner workings of every thread and every synchronisation primitive on the fly. Events can also be saved for playback. In this way, students will be able to visualize the dynamic behaviour of a threaded program and the interaction among threads and synchronisation primitives.” (Carr, S. et all, 2003) ThreadMentor proves to be an invaluable asset in navigating the complexities of multithreaded programming, offering a comprehensive and user-friendly approach to understand thread management and synchronisation primitives in operating systems.

## The Dining Philosophers Problem

A diagram of a circular object with text and symbols

Description automatically generatedDining Philosophers problem happens when we have food and a chopstick in front of each philosopher. When one of the philosophers gets hungry, he will pick up his chopsticks and start eating. All the philosophers must have two chopsticks to eat, and in our case, only two philosophers will eat. e.g., if philosophers P5 and P2 want to eat, P5 will pick up chopsticks one and five and P2 will pick up chopsticks two and three, leaving P1, P3, and P4 with either one or no chopstick, and they must wait until one of them finishes their food and puts down his chopstick.

https://miro.medium.com/max/662/1\*bBGR80ixw0l6ZELyNh7fxw.png

This scenario causes problems because we want all of them to eat in order. To solve the philosopher's problem, we are using either mutex locks or semaphore. (Neso Academy, 2021)

## Outline/Layout of your Report

In Section 2, we are going to describe the solution for semaphores, which is going to be a four-chair solution. We are going to discuss the theory behind it and show screenshots of the code for the solution with an explanation.

In Section 3, we are going to talk about our results and analysis. We will have screenshots of our solution, showing how semaphore works, explaining each screenshot, and mentioning what problem we might face, such as starvation.

# The Dining Philosophers Problem with Four Chairs

## Theory/How it works

According to Shene(2001),the four chair solutions is a deadlock free solution. It works by only allowing 4 philosophers to sit at the table at one time. It does this using a private queue and counter as well as having a signal method. When a philosopher sits down to eat they must wait on the semaphore before picking up chopsticks and must signal to it when they are finished eating to release the chair for another to use.

Within the code the constructor takes in number and iter which are number assigned to the tread and the number of cycles respectively. Each chopstick is declared individually and stored in an array. There is also a semaphore called Fourchair which has a value of 4. This semaphore is what is used for the signal and waiting which are needed for the locking to occur. The semaphore and the locks are also static so they can only be used within the files it is declared.

As the locks are declared in philosopher file the main method does not have to initials the locks and contains the creation of the philosophers and number of cycles.

**2.2 ThreadMentor:**

In this section, we compile the Semaphore four-chair problem using three C code snippets. First, Philosopher-4chairs.h is the Philosopher class which defined to control individual thread. Then, Philosopher-4chairs.cpp sets dining rules using Semaphores and Mutex locks for dining. And main.cpp, ensures ThreadMentor operates effectively.

**2.2.1 Makefile:**

The Semaphore four-chair problem, and compilation/make process:

*The first code snippet*, named Philosopher-4chairs.h, defines a class called Philosopher, which extends the Thread class. It has a public constructor that takes two arguments: **NUMBER**, representing the philosopher's number, and **ITER**, specifying the number of eating cycles. This code snippet sets the foundation for the Philosopher class, establishing the structure and behaviour of individual philosopher threads:

A close-up of a computer code

Description automatically generated

*Picture 1.1\* (Philosopher-4chairs.h)*

*In the second snippet of the code* Philosopher-4chairs.cpp, the program establishes rules to help philosophers dine together without having issues. It introduces a concept called a Semaphore or also called as “**The Dining Philosophers Problem with Four Chairs**”, which acts like a bouncer, allowing only four philosophers to sit and eat at the same time. Additionally, it uses Mutexes locks, to make sure each philosopher picks up and puts down their "chopsticks" in order, to maintain order during the dining process. The code essentially creates a proper dining environment where philosophers follow routine, avoiding any confusion or deadlock situations.

A white paper with black text

Description automatically generated

*Picture 1.2\* (Philosopher-4chairs.cpp)*

In the final section main.cpp, the program coordinates a simulated dining experience for philosophers.

A white paper with black text

Description automatically generated

*Picture 1.3\* (Philosopher-4chairs-main.cpp)*

The program depends on a command-line argument to specify the number of eating cycles that each philosopher can take:

A white background with black text

Description automatically generated

*Picture 4\* (Philosopher-4chairs-main.cpp)*

In the next step, the program generates threads for individual philosophers and assigns unique responsibilities to individual threads. So, it simultaneously initiates their execution:

A close up of a text

Description automatically generated

*Picture 1.5\* (Philosopher-4chairs-main.cpp)*

The threads follow the logic outlined in Philosopher-4chairs.cpp, simulating the philosophers' dining behaviour with synchronisation. And the last part ensures that the main program waits for all philosopher threads to complete their designated cycles before performing any final clean tasks and concluding the program:

A black text on a white surface

Description automatically generated

*Picture 1.6\* (Philosopher-4chairs-main.cpp)*

**2.2.2 Explanation of “Tags” and other ThreadMentor issues related to your solution:**

Using ThreadMentor as a visualisation application we can observe seven tags in the solution that we have picked “The Dining Philosophers Problem with Four Chairs “: JN(Join), SW(Semaphore Wait), SE(Semaphore Enter), SS(Semaphore Signal), MW(Mutex Wait), ML(Mutex Lock), MU(Mutex Unlock).

JN – Means that the current thread has joined with another thread.

SW – Means that the current thread is waiting on semaphore to signal.

SE – Means that the current thread has been let through a semaphore.

SS – Means that the current thread has signalled a semaphore.

MW – Means that the current thread is waiting to obtain(lock) a mutex.

ML – Means that the current thread has obtained(locked) a mutex.

MU – Means that the current thread has obtained(unlocked) a mutex.

*(ThreadMentor 2000-2001)*

# Results and Analysis

When we are launching the ThreadMentor visualisation tool, it's essential to include the './' command before the executable file. To successfully execute the tool, we ensure that our current directory matches the location where we extracted the philosopher tarball. Additionally, we have to provide a parameter indicating the number of eating turns each philosopher should take. In our scenario, this value is set to 6.

A screenshot of a computer

Description automatically generated

*Passing Value to the ThreadMentor*

*Picture 2.1\* (ThreadMentor 2000-2001)*

Once the program runs, we are presented with a visual starter and with a simple menu, which gives us a couple of option to choose. For analysing our case we used options such as ‘Thread Status’ to see if the Thread is Running, Joined, Blocked, Suspended or Terminated, ‘History Graph’ shows us what action happened with each thread/Philosopher whether they were waiting, locked, entering the table etc. and the ‘Thread Hierarchy’ that gives us pretty much similar output as the ‘Thread Status”. Below we can see the Thread Status and Thread Hierarchy in process, the screenshot was taken in the middle of the process.

A screenshot of a computer

Description automatically generated

*‘Thread Status’ and ‘Thread Hierarchy’*

*Picture 2.2\* (ThreadMentor 2000-2001)*

Our Four Chairs solution is avoiding deadlock which happens when all 5 philosophers pick up their left chopstick at the same time, making them all wait for the right chopstick forever. Our solution solves this problem by letting only 4 philosophers to sit at the same time. As a result, when all the philosophers pick up their left chopstick, one will be able to pick up the other chopstick, and in that way, we will prevent problems like deadlock.

*A screenshot of a computer

Description automatically generated*

*History Graph*

*Picture 2.3\* (ThreadMentor 2000-2001)*

The picture above “shows that threads Philosopher0, Philosopher1, Philosopher2 and Philosopher3 have sit down and are trying to pick up chopsticks. Since the right-most philosopher (i.e., Philosopher3) has no right neighbour, his right chopstick is free and can be used. Therefore, Philosopher3 is eating. Philosopher4 is waiting because he could not get a chair!” (*Dr. C.-K. Shene, et all 2001 – 2014)*

A screenshot of a computer

Description automatically generated

*History Graph*

*Picture 2.4\* (ThreadMentor 2000-2001)*

The above is a screenshot of the History Graph window. From the left edge of the History Graph window, we learn that Philosopher3 finished eating and released the chair, which caused Philosopher4 to take a seat. This relation is shown with a SS tag on the history bar of Philosopher3, a SE tag on the history bar of Philosopher4, and a link between them. After this, Philosopher3 started waiting (the SW tag) and after Philosopher 4 released the chair, Philosopher 3 got the chair. Meanwhile, Philosopher0 finished eating, Philosopher1 was waiting for a chair (the SW tag), Philosopher2 just got his chair, Philosopher3 is about to pick his right chopstick, and Philosopher4 is waiting for his right chopstick.

A screenshot of a computer

Description automatically generated

*History Graph and Code*

*Picture 2.4\* (ThreadMentor 2000-2001)*

This code shows us what is happening behind the curtains of SE tag. When Philosopher/Thread get SE signal means the Philosopher takes a sit and whenever Philosopher is finished the SS tag is issued which means “Release the tag”:

A computer screen shot of a computer screen

Description automatically generated

*History Graph and Code*

*Picture 2.4\* (ThreadMentor 2000-2001)*

SO once every Philosopher finish eating n-times we assigned them to eat (in our case it is 6) we are prompt with a quit message and also “Thread Status” and “Thread Hierarchy” that are showing us that ThreadMentor terminated Philosophers from eating.

*A screenshot of a computer

Description automatically generated*

*‘Thread Status’ and ‘Thread Hierarchy’*

*Picture 2.5\* (ThreadMentor 2000-2001)*

# Conclusions

In conclusion our solution avoids the issue of deadlock. As only four philosophers can be at the table at once and there are still 5 chopsticks at least one philosopher, possibly 2, will be able to eat at any time thus deadlock cannot occur. In technical terms, this means there are more locks available to be acquired by the threads thus making it easier for them to gain both locks needed to enter the critical section.

There may however be an issue in that some of the philosophers may have to wait longer to eat as they period between thinking and eating will be different and one or more philosophers may think and eat faster thus making the wait time longer. This may lead to starvation. In technical terms, the above means that each thread may enter the critical section however the time in which they will have to wait will depend on which threads obtain the locks first. It also means that some of the threads can enter the critical section more times than others who may still be waiting to enter this section for the first time.

**Advantages**

* **Synchronisation**: semaphore is a type of synchronisation as it allows us to share resources, prevent race conditions as mentioned above, and solve other problems such as deadlocks.
* **Flexibility:** We can access different types of resources by using semaphores.
* **Efficiency:** It is a fast and efficient synchronisation tool as it can be implemented using low instructions. (Eorslf, 2024)

**Disadvantages:**

* **Complexity:** semaphore can be complex, as any mistake or error will cause

synchronisation problems.

* **Overhead**: might need more resources, such as additional memory or CPU.
* **Race condition**: all the threads might be racing over which read we should access the critical section first, and that can lead to unpredictable behaviour.
* **Starvation**: Semaphores can cause starvation (Eorslf, 2024)

# References

* Neso Academy. (2021, July 13). The Dining Philosophers problem [Video]. YouTube. <https://www.youtube.com/watch?v=FYUi-u7UWgw&t=436s>
* [https://miro.medium.com/max/662/1\*bBGR80ixw0l6ZELyNh7fxw.png](https://miro.medium.com/max/662/1*bBGR80ixw0l6ZELyNh7fxw.png)
* Neso Academy. (2021, July 8). Semaphores [Video]. YouTube. Retrieved February 6, 2024, from <https://www.youtube.com/watch?v=XDIOC2EY5JE&t=660s>
* Eorslf. (2024) Semaphores in OS: Types, Advantages, and Disadvantages [online], Available from: <https://eorslf.com/computer-education/semaphores/#:~:text=Semaphores%20in%20OS%3A%20Types%2C%20Advantages%2C%20and%20Disadvantages%201,synchronization%20problems%2C%20such%20as%20deadlocks%20and%20livelocks.%20> .
* *Carr, S., Mayo, J., & Shene, C. (January 2003) "ThreadMentor: A pedagogical tool for multithreaded programming". Available at URL:* [*https://www.researchgate.net/publication/220094570\_ThreadMentor\_A\_pedagogical\_tool\_for\_multithreaded\_programming*](https://www.researchgate.net/publication/220094570_ThreadMentor_A_pedagogical_tool_for_multithreaded_programming)*. Accessed: 01/MARCH/2024*
* Picture 1, Picture 2, Picture 3, Picture 4, Picture 5, Picture 6: *Dr. C.-K. Shene (MTU), 2001-2014, "ThreadMentor: The Dining Philosophers Problem with Four Chairs" Screenshot, Available at: https://pages.mtu.edu/~shene/NSF-3/e-Book/SEMA/TM-example-philos-4chairs.html (Accessed: 28/FEB/2024).*
* *ThreadMentor 2000-2001, Explanation of "Tags" in ThreadMentor four chair solution, First version, viewed 10/MARCH/2024.*
* Picture 2.1, Picture 2.2, Picture 2.3, Picture 2.4, Picture 2.5, Picture 2.6: *Dr. C.-K. Shene (MTU), 2001-2014, "ThreadMentor: Visualizing Dining Philosophers: with Four Chairs" Screenshot, Available at: https://pages.mtu.edu/~shene/NSF-3/e-Book/SEMA/VISUAL/VISUAL-sema-philos.html (Accessed: 17/APRIL/2024).*
* Dr. C.-K. Shene. Multithreaded Programming with ThreadMentor. Aug. 2001, COMP H2014 KF: ThreadMentor E-Book (tudublin.ie). Accessed 31 Jan. 2024.
* Anonymous (2020a) *Concurrency in Operating System* [online], Available from: <[https://www.geeksforgeeks.org/concurrency-in-operating-system/>](https://www.geeksforgeeks.org/concurrency-in-operating-system%3E%3C/a%3E) [Accessed Apr 24, 2024].
* Anonymous (2015) *Introduction of Process Synchronization* [online], Available from: <[https://www.geeksforgeeks.org/introduction-of-process-synchronization/>](https://www.geeksforgeeks.org/introduction-of-process-synchronization%3E%3C/a%3E) [Accessed Apr 24, 2024].
* Anonymous (2010) *Critical Section in Synchronization* [online], Available from: <[https://www.geeksforgeeks.org/g-fact-70/>](https://www.geeksforgeeks.org/g-fact-70%3E%3C/a%3E) [Accessed Apr 24, 2024].
* Anonymous *Concurrency in Operating System* [online], Available from: <[https://www.tutorialspoint.com/concurrency-in-operating-system>](https://www.tutorialspoint.com/concurrency-in-operating-system%3E) [Accessed Apr 24, 2024].
* Anonymous *Deadlock in OS: Necessary Conditions and Examples* [online], Available from: <[https://www.theknowledgeacademy.com/blog/deadlock-in-os/>](https://www.theknowledgeacademy.com/blog/deadlock-in-os%3E%3C/a%3E) [Accessed Apr 24, 2024].
* 'Starvation and Livelock (the Java™ Tutorials > Essential Java Classes > Concurrency)'.
* Anonymous (a) *What is the Critical Section Problem in Operating Systems?* [online], Available from: <[https://www.educative.io/answers/what-is-the-critical-section-problem-in-operating-systems>](https://www.educative.io/answers/what-is-the-critical-section-problem-in-operating-systems%3E) [Accessed Apr 24, 2024].
* Ravulakollu, N. (2024) *Process Synchronization in OS: Types, Problems & Solutions* [online], Available from: <[https://www.almabetter.com/bytes/articles/process-synchronization-in-os>](https://www.almabetter.com/bytes/articles/process-synchronization-in-os%3E) [Accessed Apr 23, 2024].
* Yadav, B. (2022) *Process Synchronisation in OS* [online], Available from: <[https://www.scaler.com/topics/operating-system/process-synchronization-in-os/>](https://www.scaler.com/topics/operating-system/process-synchronization-in-os%3E%3C/a%3E) [Accessed Apr 24, 2024].
* Anonymous (2024) *What are the main benefits and challenges of concurrency in operating systems?* [online], Available from: <<https://www.linkedin.com/advice/0/what-main-benefits-challenges-concurrency-operating>> [Accessed Apr 23, 2024].

# Appendix: Personal Reflections

**Steven’s reflection:**

During this project I learned a lot about issues we can run into in operating systems. Problems such as deadlocks, starvation, and using the Threadmentor tool to visualize these problems and their solutions.

I really enjoyed learning about the Dining Philosophers Problem, it’s an interesting thought experiment and helped with understanding the issues faced in operating systems.

I found challenges in finding specific information, especially in regard to Threadmentor. I also found it difficult at the start to understand the tags used in the program.

If I were to do this project again, I would like to explore the Dining Philosophers problem but not using Threadmentor I would like to understand visualising it in different ways/ scenarios.

For a future version of this project, I would like to have a choice between using Threadmentor or a different tool for the Dining Philosophers Problem.

**Piotr’s reflection:**

Throughout this project, I gained valuable knowledge about concurrency and synchronisation issues in operating systems, particularly through the exploration of the Dining Philosophers Problem.

One aspect I particularly enjoyed in this project was delving into the intricacies of the Dining Philosophers Problem. It served as an excellent thought experiment of resource allocation and thread coordination in operating systems.

Initially, I encountered some challenges in getting to understand tags and History Graph in ThreadMentor, which caused a bit of frustration. However, with the assistance of my group members, we were able to overcome these obstacles.

If I had the opportunity to redo this project, I would explore alternative solutions to the Dining Philosophers Problem to gain a deeper understanding of how different approaches can impact resource allocation and synchronisation.

For future iterations of this project, I would recommend providing clearer guidelines on using ThreadMentor and troubleshooting common issues that students may encounter.

**Rochelle’s reflection:**

From the project I learned about semaphore, mutex locks and how to use ThreadMentor software. I also learned about the Dining philosophers problem and I liked learning about the dining philosopher problem as it can help to visualize resource allocation in operating systems and how you can stop deadlock from occurring in the system. I also learned about processes and threads and how starvation can occur, although in our solution we did not have any instances of it occurring when we ran ThreadMentor.

I had a bit of difficulty in the beginning getting the ThreadMentor program to work but with the help of my group I was then able to get it working. I also found it hard to figure out what to include in some of the report so that it was written well and informative. If I was to do this again, I would try one of the other solutions just to see how differently it can solve the solution and so that I can learn about the solution in more detail. With this project I would recommend to students to start it as soon as you get the brief as the project can be time consuming and cannot be left until the last minute. I would have liked if the marking scheme for the presentations and the report went into a bit more detail so that we can ensure that our report and presentations contain everything that is needed and also goes into the detail required

**Habiba’s reflection**

In this project, I learned about different synchronisation tools such as mutex locks and semaphore. I also learned about dining philosopher problems like deadlock and their solutions, like the four-chair solution.

I liked learning about the dining philosopher problems and their solutions, such as semaphore and the four-chair solution. I enjoyed the part when I was downloading ThreadMentor in the lab to see how the solution actually works, which helped me understand more about the solution that we picked.

I didn’t like the way that it wasn’t clear at the start of the semester what we should specifically talk about, which made me stressed from the first week, and it got worse when I joined a group without clearly knowing what I was going to talk about. I also didn’t like how I still don’t know why I am doing a project on this topic. I don’t know why it is important for a cyber security student, as it wasn’t mentioned in the lecture.

I don’t think that I will do something differently. I started to read about the dining philosopher problems from the first week because I was worried about them, as they were mentioned before, which made me take the topic seriously. I would probably read more about the solutions and other solutions, but other than that, I don’t think that there is something different I could do.

# Appendix: Project Planning and Management

**7.1 Gantt chart:**

A screenshot of a computer screen

Description automatically generated

*(Gantt chart, 2024)*

Above is a screenshot of the Gantt chart used for the module. It highlights a timeline of our project deadlines etc. and was an essential part in our timekeeping and ensuring our report was up to the best standard possible.

**7.2 How we managed the project on a week-to-week basis:**

We kept recording of Diary’s week-to-week basis where we recorded weekly progress and uploaded them on Trello. All project Diary’s are reflecting our progress over the weeks.

A project diary with text and numbers

Description automatically generated

*First weeks Diary(09/FEB/2024)*

*A project diary with numbers and text

Description automatically generated*

*Last weeks Diary(23/APRIL/2024)*