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:** *18/MARCH/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: 18/MARCH/2024

Author: Steven Kelly Dated: 18/MARCH/2024

Author: Rochelle Mullen Dated: 18/MARCH/2024

Author: Piotr Momat Dated: 18/MARCH/2024

Abstract Page

<NEW PAGE>

Table of Contents

<NEW PAGE>

List of Figures

<NEW PAGE>

List of Tables (if you have them)

<NEW PAGE>

<https://libguides.ucd.ie/harvardstyle/introduction>

# Introduction (Steven)

## Background

**1.1.1 Concurrency and Synchronisation Issues in OSes** (Steven)

**1.1.2 Mutex Locks** (Rochelle)

**1.1.3 Semaphores:** (Habiba)

Semaphores can be viewed as an extension to mutex locks. “Semaphores used to solve the critical section problems and to active process synchronization 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 has to wait. All the elements are supposed to be initialized to 1.

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.

**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.

“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.

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.

**1.1.4 ThreadMentor:** (Piotr)

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 synchronization 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.

## A diagram of a circular object with text and symbols Description automatically generatedThe Dining Philosophers Problem (Habiba)

Dining 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 have to 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 have to wait until one of them finishes their food and puts down his chopstick.

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.

## Outline/Layout of your Report

* In section 2, we describe “this”. In section 3, we describe “that”…
* Tie the sections together: *briefly* describe how they are related

# The Dining Philosophers Problem with Four Chairs

# 2.1 Theory/How it works (Rochelle)

**2.2 ThreadMentor:** (Piotr)

In this section, we compile the Semaphore four-chair problem using three C code snippets. First, in Philosopher-4chairs.h, the Philosopher class is 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:** (Piotr)

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\* (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 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 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 5\* (Philosopher-4chairs-main.cpp)*

The threads follow the logic outlined in Philosopher-4chairs.cpp, simulating the philosophers' dining behaviour with synchronization. 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 6\* (Philosopher-4chairs-main.cpp)*

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

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.

# Results and Analysis (Piotr & Steven)

Screenshots of a **single** program run, to include:

* Several History Graph screenshots + proper captions
* Screenshots of highlighted code corresponding to various ThreadMentor tags for Philosopher threads in the History graph above + proper captions
* Screenshots of main ThreadMentor window showing relevant and corresponding information relating to the above.
* Screenshots of Thread Status window showing relevant and corresponding information relating to the above.
* Detailed descriptions in the text of what is happening to each of the Philosopher threads making reference to the screenshots.

Deadlock happens when all the philosophers pick up their left chopstick at the same time, making them all wait for the right chopstick forever. To solve this problem, we can let the philosophers be no more than four instead of five, and we will leave the chopsticks as they are. 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.

# Conclusions (Habiba & Rochelle)

Technical conclusions about your solution; for example:

* “our solution avoids a specified problem”, or
* “our does had a specified problem”
* Advantages/Disadvantages of our solution

Advantages

Disadvantages:

* Semaphore can be complex, as any mistake or error will cause synchronisation problems.
* Might need more resources, such as additional memory or CPU.
* All the threads might be racing over which read we should access the critical section first, and that can lead to unpredictable behaviour.
* Can cause deadlock as all the threads could be initialised to zero, causing them to be waiting forever.

The conclusions should **not**include things like: “I loved this project!”, “I hated this project!”, “I learned x, y and z on this project”. These things should go in the Personal Reflections section.

# References

**Habiba’s references:**

**Piotr’s references:**

*1. 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*

2. 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).*

**Steven’s references:**

**Rochelle references:**

The references listed here **must** be cited within the text of your report.

It is not good enough simply to list the references here and have no citations in the text – marks **will** be lost if you leave out the citations or if you use Wikipedia as a reference source.

# Appendix: Personal Reflections

* A brief summary of what you learned
* What you liked about the project
* What you didn’t like about the project
* What would you have done differently if you could do it again
* Any other recommendations/feedback for the Lecturer and/or future Students

# Appendix: Project Planning and Management

(Ask if we can put Diary here)

* Gantt chart
* Description of your Gantt chart
* How you managed the project on a week-to-week basis