

**Operating Systems**

**A Study of the Synchronisation and**

**Concurrency Issues in the Dining Philosophers’**

**Problem completed using the ThreadMentor**

**Visualisation Tool**

**Robert Cowper – B00096855  
James Finglas - B00094138**

**Cathal Murphy – B00095643**

**Nadia Merrigan – B00100237**

***Department of Computing***

***Institute of Technology, Blanchardstown***

***Dublin 15.***

**Word Limit for Assignment: Max 5000 Actual Word Count: 2531**

**Bachelor of Science**

**Operating Systems**

**14/04/2018**

Table of Contents

[Introduction 3](#_Toc506661300)

[Dining Philosophers 3](#_Toc506661301)

[Threads & Their Importance 4](#_Toc506661306)

[Synchronization 4](#_Toc506661311)

[Thread Mentor 5](#_Toc506661312)

[The Solution Explained 6](#_Toc506661313)

[Conclusions 11](#_Toc506661300)

[References 12](#_Toc506661300)

# Introduction The goal of this project was to evaluate the Dining Philosophers problem and to understand how this problem relates to computing. This is a crucial step in understanding how one must program computers effectively in order to achieve maximum efficiency without causing programs to crash, conflict and compete for access to system resources. In order to facilitate this, a visual aid is needed to provide the ability to visually inspect running programs and their orders of execution. For this project Thread Mentor was selected and run on our virtual operating systems. Linux, and more specifically Mageia 5 was the operating selected for carrying out these tasks. Once our group had achieved an understanding of the problem at hand, we then had to implement a solution to the problem, which is provided. We then had to code, compile and run this solution and evaluate it, in order to understand how it resolves the problem of conflicting tasks competing for access to system resources which might then create a deadlock causing a system to hang on a process. Several presentations where also required in order to access our understanding of the problem and the given solution, with the first being a progress report presentation and the second being a final presentation to report our findings and conclusions. while emphasis is placed on each group member to understand all of the details pertaining to both problem and solution, we also delegated specific aspects of the project to individual members who could then brief the group in order to achieve a good level of efficiency and to ensure that each member understood each aspect of the project as we progressed. This proved to be a very effect method for our team.

# Dining Philosophers What exactly is the dining philosopher’s problem? Well it is in essence a philosophical approach to the problem faced when competing programs try to execute simultaneously try to access system resources. Image a family seated at a dining table eating using chopsticks. However, each family member has only one chopstick. If each member tries to pick up two chopsticks at the same time, a conflict is created in which nobody can eat, and this continues unabated until each family member starves. It is only when two chopsticks are available that any family member competes. As you can see while it is technically possible for each family member to eat, when this idea is applied to computers where each individual program is only concerned with its own needs, a terrible situation results during which no program is prepared to take a back seat and let the other go ahead of it. It is therefore the job of a system designer, or individual programmer designing a singular program to organize his code in such a way the each task knows when to wait, stop, execute or die in order such that each task is performed with maximum efficiency without conflict.

# Threads & Their Importance

Threads are basically tasks within a program which execute a function at the running phase of a program. If you imagine the philosophical approach for a moment, as each family member reaches for a chopstick, this would represent a Thread. So, if all family members attempt to eat simultaneously we have multiple Threads trying to execute together. This is known as a race condition, and race conditions, while a normal function for computing, are very bad if left uncontrolled.  
  
This problem is exasperated in computing because each Thread may create a sub Thread, or Child Thread. The original Thread then becomes known as a parent Thread. Each Child Thread may then create a Thread of its own. So as a program proceeds you may see a cascading sequence of Thread creation starting from one Thread and rapidly multiplying into hundreds of Threads.  
  
While Threads are a fantastic tool for a programmer drastically reducing the need for code repetition where a task must be carried out many times in rapid succession in many variations they also create a large problem. The problem is twofold. Race conditions created by Threads rushing to access system resources may obviously create deadlocks or bottlenecks which will cause the program or even the entire system to crash. However, let us imagine for a moment that somehow ideal conditions allowed for each thread that accesses system resources to que up one by one and each gained access without causing deadlock, what might then happen?  
  
Well the second problem of race conditions is that each Child Thread is invariably tied to its Parent Thread in most basic programs and/or operating systems. So, if for example a parent Thread completes its task and terminates before its Children Threads have completed theirs, all Children will automatically terminate, thus destroying the functionality of the program being designed.

**Synchronization**

To overcome issues surrounding Threads with regard to race conditions, we can employ a method called synchronization. So, what is Synchronization? Well a simple analogy is traffic lights. If we had traffic junctions with no traffic lights, we would inevitably see untold numbers of crashes as each driver tries to determine when it is safe to proceed. As traffic increases crashes increase until the point where the junction is impassable. This would create a bottleneck at least or a deadlock at worst and all traffic would cease progressing.  
  
This is the same in computing. So, we must create a set of traffic lights which control the flow of data traffic. This is Synchronization.

There are two types of synchronization in C++ code that pertain to this project. Mutual Exclusion Locks, or mutex for short & Semaphores. A mutex lock is in theory a simple key to a door. If a piece of code, or access to a resource system resource must be protected such that only one thread can access it at any one time a door is placed around it and a key is used to access the door. As each Thread attempts to enter the door, a check is undertaken to see if the door is locked. If the door is unlocked, The Thread enters and locks the door behind it. When another Thread approaches the door and finds it locked, this Thread enters the wait stage of the mutex lock. When the original Thread finishes its task, it unlocks the door for the next Thread to enter.

A Semaphore is considered to be an extension of a mutex lock. It is however more complex in its execution. Rather than the door being locked, each Thread is assigned a non-negative counter value. As it proceeds through its tasks the value of the counter is decreased until all tasks have been executed.   
  
In semaphores a que system is used for each Thread to store its current counter value and evaluate those of other Threads if needed. For example, two Threads A & B wish to access a resource. Thread A proceeds and executes a task after which it enters the wait mode, it then sets it counter value to 1. As Thread B attempts to enter access the resource, and it performs a test to see if a counter exists. If a counter for A is found and it is not 0, Thread B will enter the wait mode. When Thread A finishes its task or tasks it will change its counter value to 0 and Thread b Will also proceed. The que section of semaphores is known as signal. So, it is a WAIT/SIGNAL method of synchronization.  
  
What is interesting about semaphores is that they can used to allow a series of Thread’s to access a resource, and execute a task then enter the wait phase while another task is being carried out, where the same resource may be needed several times creating a simple loop allowing for many Threads to carry out complex calculations efficiently. As each Thread attempts to access a resource it checks for a semaphore counter value and a signal value. If the currently accessing thread is not in wait mode, the Thread trying to access enters wait mode until it can proceed. When the First Thread finished the current task or all tasks, it moves on and the next Thread can proceed. This is a more complex locking system both in terms of coding and execution and the programmer must think very carefully about his or her logic when applying semaphores in their code.

# Thread Mentor Thread Mentor is a relatively simple program contained is own library classes, these are not import for use to understand. It also contains a visual GUI system that allows us to visually inspect running Threads which represent tasks attempting to be carried out by running processes.

This tool is crucial for evaluation of any proposed solutions to the problem the project faces, however it also is extremely powerful tool going forward into our bright futures in the dynamic world of programming as it gives us a very simple method of evaluating our programs during the coding phase of the design process to discover if we have implemented our Threads correctly to avoid conflicts.

# The solution Explained For our solution we chose to use the Mutex lock solution. However, this solution creates a potential deadlock. See the mutex locks will only protect the resources which threads are trying to access. It will however control when these resources try to access the resources initially. Imagine if you will that two of the family members discussed above each try to access the chopstick at the same time. Neither will obtain two chopsticks and thus neither can eat. So, we must create a method of control such so that when each family member attempts to eat if only one chopstick is found the family member will not attempt to eat. We achieve this by creating an array with 5 indexes’, each representing a chopstick. As each thread attempting to access the array is synchronized, only one may access at any one time. As we have 5 available chopsticks, this means that two family members may eat at the same time, and as one family member finishes eating, a new family member may enter the array and eat. here we see such an array being created to control the Threads access to chopsticks which determines if they can eat or not. On the following page we will see the Threads being created through E-macs a virtual install OF Mageia.

# 

# *[1]* C/O Multithreaded Programming with ThreadMentor: A Tutorial

Next, we will create the mutex locks for the Threads which will which Threads will be allowed to access resources and when each Thread will enter wait mode as required. We will also create a loop which will recreate the problem of the Dining Philosophers which we will then need to create using the create application of Mutex locks.  
  
Our aim in facilitating this solution will be to create a series of cascading locks, allowing only one Thread to flow through the arrays at a time. Then the Thread will be allowed to proceed, at which time it will be in possession of two chopstick and eating can commence. Once Eating has finished the Thread will put down its chopsticks and terminate once it exits the loop , finally it will send its termination signal to its Parent Thread.

# 

# *[2]* C/O Multithreaded Programming with ThreadMentor: A Tutorial

# Lastly, we have our solution. Each chopstick is protected by a mutex lock, so we must be careful in our accessing of chopsticks lest we create a deadlock. As seen in the code below, we avoid deadlock by having only one Thread enter the loop. The Thread will delay to simulating a Philosopher thinking. It will then pick up the left chopstick, followed by the right chopstick. Eating may now begin for a period of time. Once eating has concluded the chopsticks are replaced one by one and the loop is exited placing the mutex locks in the unlock setting. This allows the next Thread, if any, which will be in the wait setting to proceed. Thus, a smooth cascading sequence of lock, wait and unlock is created as seen in the following image. *[3]* C/O Multithreaded Programming with ThreadMentor: A Tutorial

# A screenshot of a cell phone Description generated with very high confidence

***Thread mentor screenshot taken from a virtual operating system executing the mutex lock dining philosopher’s solution***

# Conclusions

The problem posed by the Dining Philosophers And it becomes immediately clear how we can and will face this issue in all aspects of program design. From operating systems to simple Threaded calculations. It is therefore really important to understand Threads, their importance, issues surrounding implementation and solutions to those issues.   
  
Both mutex and semaphores are interesting. In comparing the two, we can see the contrasts between them. While dead lock can occur with mutex locks, starvation can occur with semaphores. We learned that while semaphores are adequate in situations where the sequence of execution is not incredibly important, so long as any parent Threads are joined to their Children such that the parent will not terminated first, thus breaking the program.   
  
Whereas in contrast, a standard mutex lock allows a more linear form of control over how resources are access by Threads.  
  
Standard mutex locks appear to be much more efficient in terms of memory usage in that they will attempt to access system resources until the unlock signal is issued by the currently controlling Thread. Semaphores on the other hand will make repeated attempts to evaluate the signal loop until any relevant signal counters have been reduced to zero. It seems clear that each has a valuable place in the programmer’s toolkit and the next logical step will be to gain the experience needed to understand where and when each can be employed to maximum benefit.

Thread mentor is an amazingly lightweight yet powerful tool for analysing not only the Dining Philosophers solutions, but also for junior designers such as us, and I am confident I will get a lot of use from it in the future in terms of controlling our Threads and our program efficiency. This will be especially important when designing for micro systems such as the basic raspberry pi right up to fully modular blade servers.

We feel as a group, that we have gained some very valuable insights into the more advanced fundamentals of program design and implementation. And our understanding of C programming has increased greatly. And while the focus of the project was on understanding Threads specifically if has helped us gain a greater solid footing in terms of our overall understanding on coding C.

# References [1,2,3] Multithreaded Programming with ThreadMentor: A Tutorial

### Dr. C.-K. Shene

### Professor Department of Computer Science Michigan Technological University

© 2001-2014 C.-K. Shene

## 