TECHNOLOGICAL UNIVERSITY DUBLIN

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

by

Dahir Mussa, Patricia Mc Keon, Brian Maher, Robert Preston

A report submitted in partial fulfillment for the
Operating System Module

in the
School of Computing
Department of Informatics and Engineering

26th April 2021

Declaration of Authorship

"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 University 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 us 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 our 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 our work."

Signed: Dahir Mussa, Patricia Mc Keon, Brian Maher, Robert Preston

Date: 26th April 2021

Contents

List of Figures

Chapter 1

Introduction

This report is based on our research of the four-chair solution, we will explain the running of the program as in how to run the program and the use of the makefile. We will also explain the visualization of the solution with the assistance of the ThreadMentor interface. Using screenshots of diagrams to show the main interface of ThreadMentor, how it functions and how it is used to visualise the program. At the end we will provide a personal reflection from each member of the group to explain how the group members felt during the working of this project.

1.1 Background

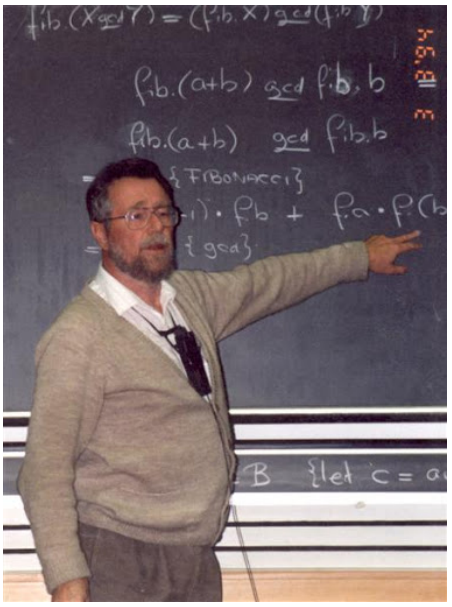

FIGURE 1.1: E. W. Dijkstra

Edsger Wybe Dijkstra invented the dining philosopher's problem in 1965[2]. It is used to illustrate the deadlock concept when it comes to sharing resources in Operating systems. This is represented by a round table with a bowl of spaghetti in the centre of the table. On the table are five chopsticks and five chairs which are placed around the table. The problem occurs when five philosophers sit down to eat as they need two chopsticks to eat the spaghetti. Each philosopher has only one chopstick placed on the right-hand side of their sitting position. This means that none of the philosophers can eat, therefore they might starve[3].

In this example the chopsticks are the shared resource, and the philosophers are the processes. The processes are dependent on each of the other processes and for one process to run it must wait until the other is finished. Otherwise, we have deadlock if everyone wants to eat at the same time. The philosophers spend their whole time doing one of three things eating, waiting, or thinking[3].

1. Dijkstra image - Http://cs-exhibitions.uni-klu.ac.at/index.php?id=52

2. https://www.studytonight.com/operating-system/introduction-to-semaphores

3. https://medium.com/swlh/the-dining-philosophers-problem-bbdb92e6b788 All accessed on 27th March 2021

1.2 Types of Solutions to this Problem

There are three solutions to this problem:

- Mutual Exclusion Locks
- Semaphores
- Monitors [3]

Mutual Exclusion Locks:

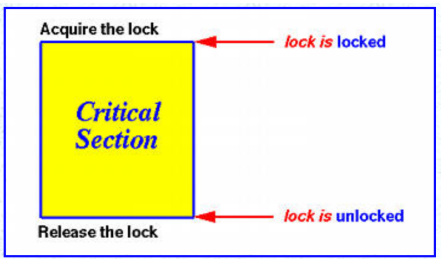

FIGURE 1.2: Critical Section [4]

When a data item is shared with a few threads to prevent certain conditions it is best to make sure the item is protected securely. The easiest way to do this is with a lock. Before a thread can access the data, it must acquire the lock. Only one thread can have access to the lock at any given time. Once the thread has acquired the lock it becomes the owner of the lock. The lock is then locked, and the owner can access the protected data. When the tread is finished it must release the lock and then the lock is unlocked. We can see this in figure 1.2 above. The thread still has access to the protected data until another thread acquires the lock, and the lock becomes locked. When the lock is locked, if a second thread tries to acquire the lock it will be unsuccessful and it will have to queue at the lock until the lock is released. By using a lock, we establish a critical section which can also be seen in the figure 1.2 [4].

[4] https://pages.mtu.edu/ shene/NSF-3/e-Book/MUTEX/locks.html accessed 14th April 2021

Semaphores:

There are two types of semaphores one called Binary semaphore and the other called Counting semaphore. The Binary semaphore has two values zero and one. It is known as the Mutex lock. The Counting semaphore can take in values greater than one. Semaphores are always initialised to one [5]. Semaphores are an extension of the Mutex Lock with two methods Wait () and Signal (). They work very similar to the Lock () and Unlock () with a critical section as you can see in figure 1.3 below [6].

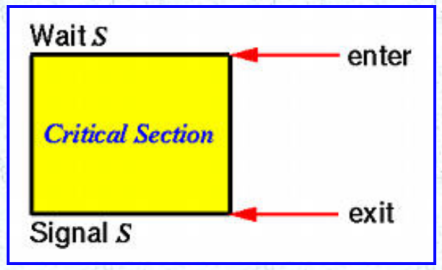

FIGURE 1.3: Wait-Signal Sequence [6]

[5] https://www.geeksforgeeks.org/semaphores-in-process-synchronization/

[6] https://pages.mtu.edu/ shene/NSF-3/e-Book/SEMA/basics.html accessed 14th April 2021

Monitors:

We can use Monitors as another way of achieving process synchronisation. To achieve mutual exclusion between the different processes we can use the java programming language. The java language provides building blocks for the wait () and signal () constructs.

The condition variables and procedure combined within a package help synchronise the methods. Any process going on outside of the monitor can only call the procedures of the monitor and not the internal variables. Only one process can access the code in the monitor at a time. The two condition statements we have with monitors are:

- Wait
- Signal [7]

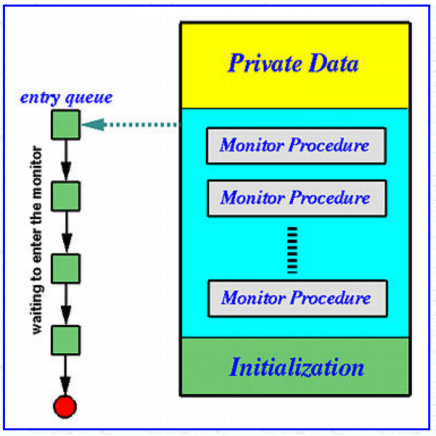

FIGURE 1.4: Monitor Components [7b]

The four main component of a monitor are Initialisation, private data, monitor procedure, and the entry queue. This can be seen in figure 1.4 above. The code is contained in the initialisation component and when the monitor is created it is only used once. The private data contains all the private variables or private procedures and can only be used inside the monitor. As mentioned above these private variables are not visible outside of the monitor. All the threads that call the monitor procedure that have not been granted permission are contained in the entry queue.[7b]

[7] https://www.geeksforgeeks.org/monitors-in-process-synchronization/ accessed 27th March 2021

[7b] https://pages.mtu.edu/ shene/NSF-3/e-Book/MONITOR/basics.html

1.3 Semaphores - Four Chair Solution

We have chosen the Four chairs solution which is one of the dining philosophers problems. When four philosophers have already sat down, the fifth philosopher is blocked, allowing four philosopher to sit down at the table at the same time. Those four philosophers will pick up four chopsticks that is nearest to them [4].

There will be one chopstick left on the table, which is the right, one of the philosophers can start to eat. When the philosopher is finished eating, he puts down both chopsticks to allow the next philosopher to have a chance to pick up his right chopstick then he starts to eat.

Each philosopher takes it in turn to eat, think or wait. We require to initialized semaphore to four, before each philosopher pick up their chopsticks they must wait for the semaphore, after the philosopher finish eating, they signal the semaphore to release their chair [4].

[4]. https://pages.mtu.edu/ shene/NSF-3/e-Book/SEMA/TM-example-philos-4chairs.html

1.4 Deadlock

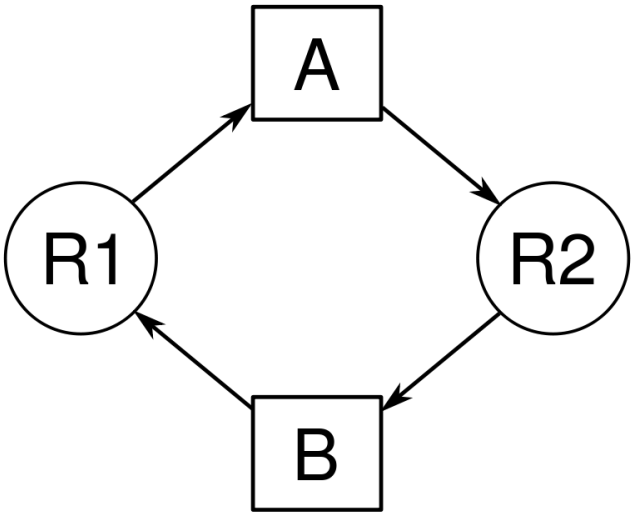

FIGURE 1.5: Deadlock

Definition of Deadlock

A process is a program in execution and requires a variety of resources while executing. Initially, the process requests a resource. After acquiring the resource, it uses it and releases the resource. Deadlock can happen when a process holds a resource and waits to get a resource that is held by another process.

Deadlocks occur when the following conditions occur:

Mutual Exclusion – One process uses a resource at a time.

Hold and Wait – A process that holds one or more resources is waiting to acquire additional resources held by other processes.

Below are some ways to eliminate a deadlock after it has occurred:

Pre-emption – Obtain a resource from a process and allocate it to another.

Rollback – Roll everything to the last safe state and divide up the resources differently to avoid a deadlock situation.

1.5 Starvation

Definition of Starvation

"Starvation occurs when a process waits for a resource for a longer period and when these processes wait for an infinite amount of time, we refer to it as starvation of resource starvation".

One solution to overcome starvation is aging. It is a technique that steadily increases the priority of processes that wait for a long time. The process can obtain the required resources when the priority increases.

Chapter 2

Running the Program

2.1 How to Run the Program

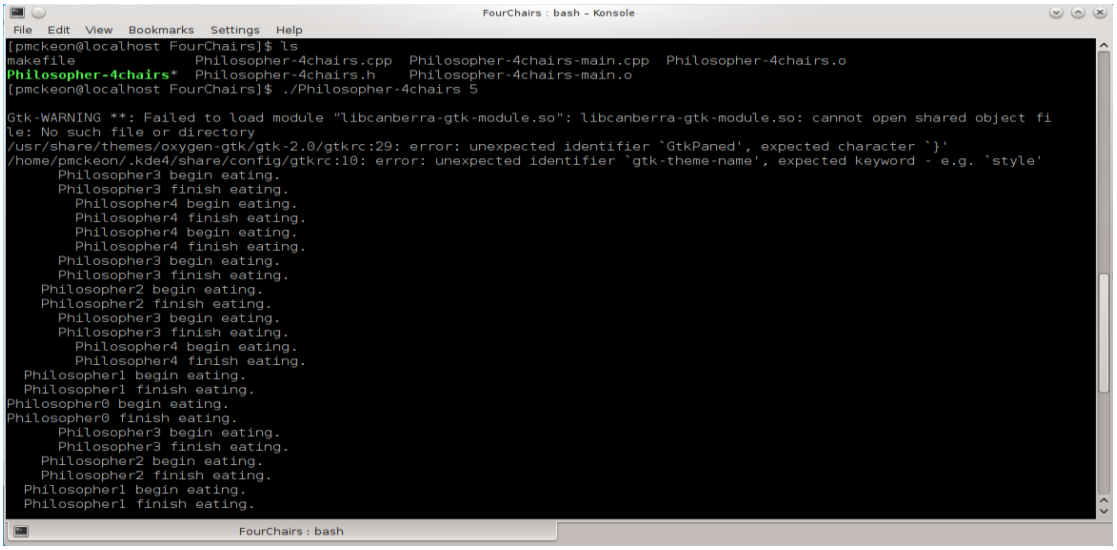

FIGURE 2.1: Running Program

Firstly, we created an OS directory and downloaded the thread mentor package from thread mentor source code files on Moodle than coped the ThreadMentor-fedora4 tarball from downloads into the OS directory. We unzipped the threadmentor-fedora tarball in one go by typing tar -zxvf ThreadMnetor-fedora4 on the command line, this created a new subdirectory called Thread Mentor[6].

Subdirectory was created in the os directory which is called myprograms to contain the programs. Another subdirectory called FourChairs was created in the OS directory that contains the coded solution which are the two C++ files, one header file from thread Mentor website and also it contains makefile[6].

The final step was compiling the Makefile using the command make than the executable came up. To run the executable type the following in the konsole:

./Philosopher-4chairs 5

./ which is the current working directory, Philosopher-4chairs is the executable program and 5 is the parameter[6].

Figure 2.1 above shows the running program, log in as an ordinary user after that changed the directory's using the cd command[6]. For example, changing into the OS directory by typing cd os in Konsole, than changed the directory into the subdirectory called myprograms by typing cd myprograms in the Konsole after that changed into another subdirectory called FourChairs by typing cd FourChairs in the Konsole , the Fourchairs contains the program , the program was run by typing ./Philosopher-4chairs 5

[6].file:///C:/Users/35383/Downloads/OS

2.2 Makefile

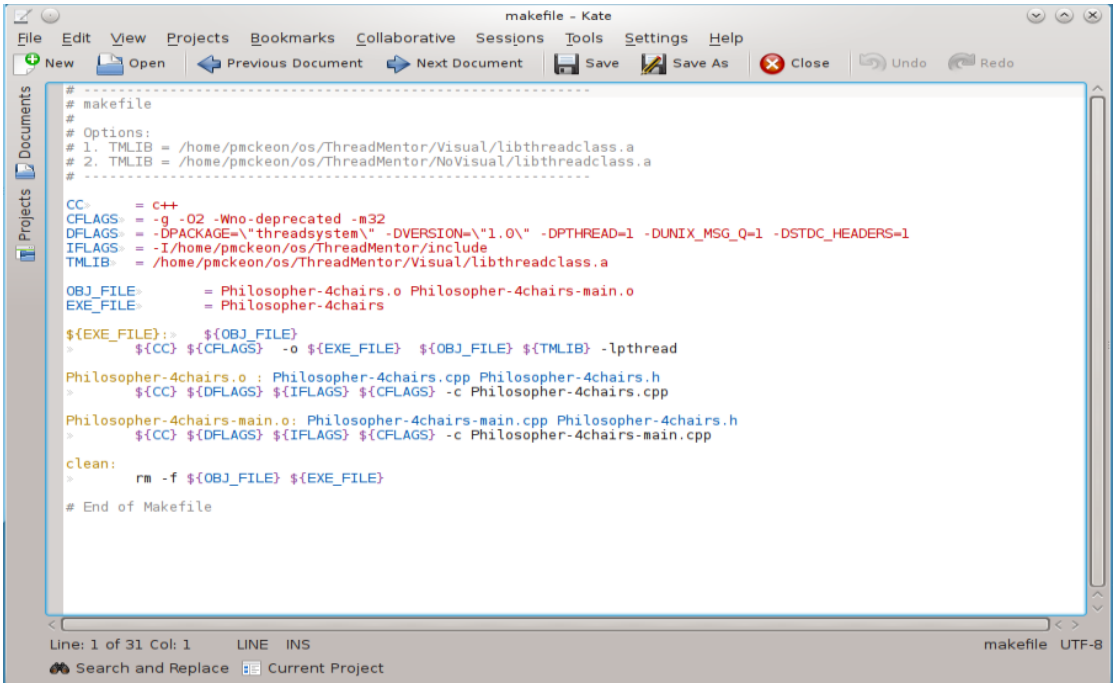

FIGURE 2.2: Makefile

The Makefile was coped from producer consumer into the FourChairs directory after that changes have been made on the object files and the targets and also dependency to match it with the names of the files from the thredmentor website[2].

Figure 2.2 shows what is in the Makefile above image, CC is a C compiler which is use to compile a program, in this case is a C++ compiler[1]. CFLAGS is used in the makefile, when the compiler is invoked[4].

-g is a debugging symbols.

-02 is level two of optimisation.

-W is warning, no-deprecated is old code fashion.

-m32 is bit which is 32[].

We changed the IFLAGS and TMLIB to match it with the os directory and username set-up. The EX-File is the target and OBJ FILES is Dependency[2].

Compiling some flags CC, GFLAGS, putting the output into EXE FILE which is the executable called Philosopher-4chairs.

Compiling .o files which are the object files, Philosopher-4chairs.o Philosopher-4chairs-main.o.

TMLIB is library that was created by the people who wrote the threadmentor application. The libthreadclass.a has codes for running the windows on the thread mentor and also allow you to press bottoms.

-l is library build in and POSIX thread allow us to create threads.

Philosopher-4Chairs.o is the target, to make the target it depends on Philosopher-4chairs.cpp and Philosopher-4chairs.h, which they are dependencies.

CC is compiler, IFLAGS is include because of the header files in the include subdirectory of thread mentor directory[2]. -c is used only small bit of the program instead it creates the object file.

The clean target removes the object files and executable by force -f option.[9]

[1] https://www.google.com/search?q=what+is+cc+in+makefilehl=ensxsrf=ALeKk03C9qt0RCoW39q

[2]file:///C:/Users/35383/Downloads/OS

[4]https://www.google.com/search?sxsrf=ALeKk03hrvkZDbUKd$_F jEqxfZdhMib5tJQ$

[5].

[9]. https://www.google.com/search?q=what+is+clean+in+makefilesxsrf=ALeKk01JP1AId0IAKN5M

2.3 The PATH environment variable

The PATH environment variable was set up, including the path to the ThreadMentor visualisation executable which contains in /bin subdirectory of the thread mentor directory[7].

We include the following PATH=PATH:/home/¡username¿/os/ThreadMentor/bin export PATH in the .bashrc file so that every time we open a konsole it will executes and we don't have to set the PATH each time we start a konsole window. The dollar symbol must be typed, it has nothing to do with command prompt, instead is refer to as the Path environment variable. We are adding a directory to the existing path instead of creating a path[7].

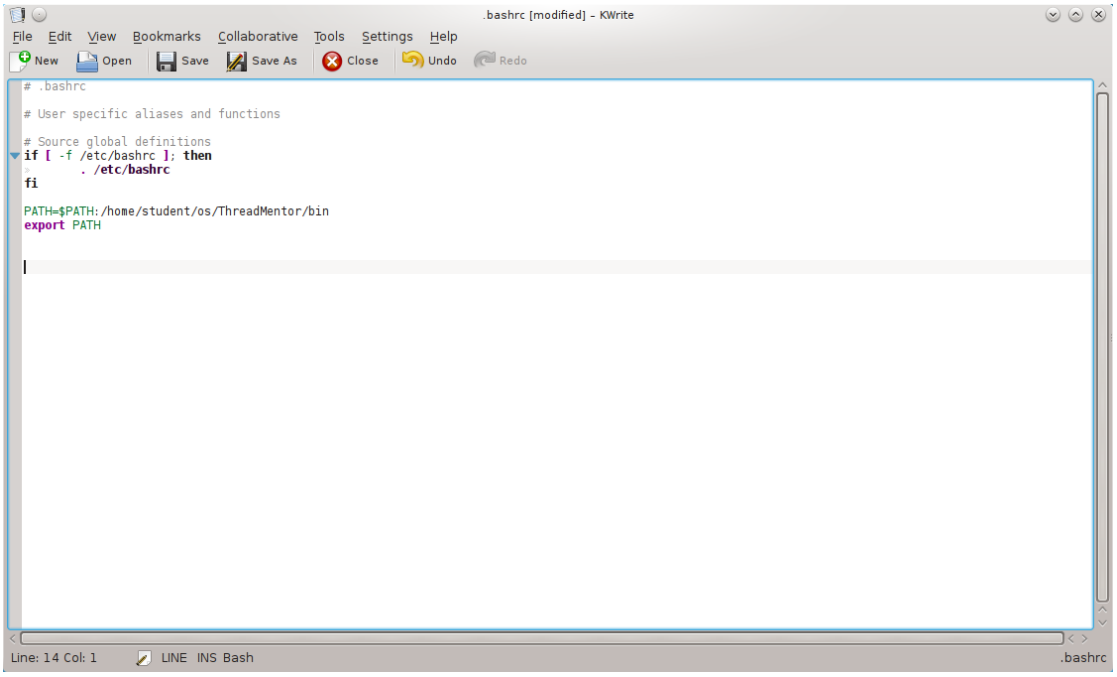

FIGURE 2.3: .bashrc

[7].file:///C:/Users/35383/Downloads/OS

Chapter 3

Visualization

3.1 Thread Mentor Interface

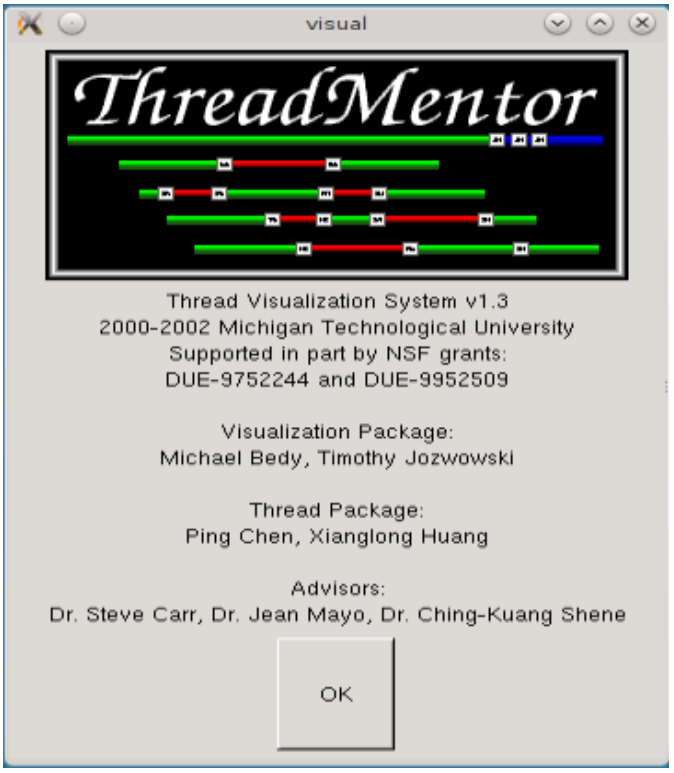

FIGURE 3.1: Thread Mentor Interface

When a program that is compiled with ThreadMentor starts, it automatically brings up ThreadMentor's visualization subsystem.

The visualization system then starts, it then brings up the information window:

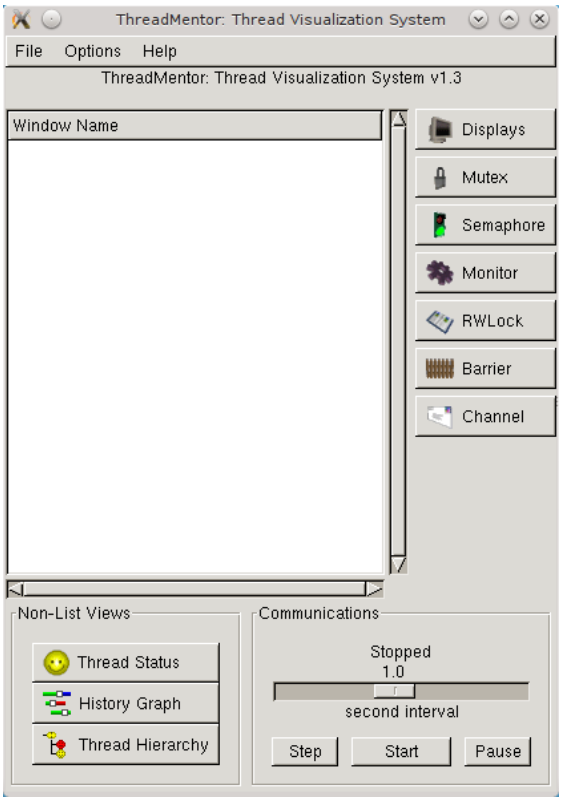

FIGURE 3.2: Thread Mentor Main Window

Once OK is clicked the program will display the main window. The main elements of this main windows are:

Display Area There are seven buttons on the right side of the window. Six of these buttons permit selecting of one of the six synchronization primitives to be displayed in the display area.

Control of the execution of the threaded program.

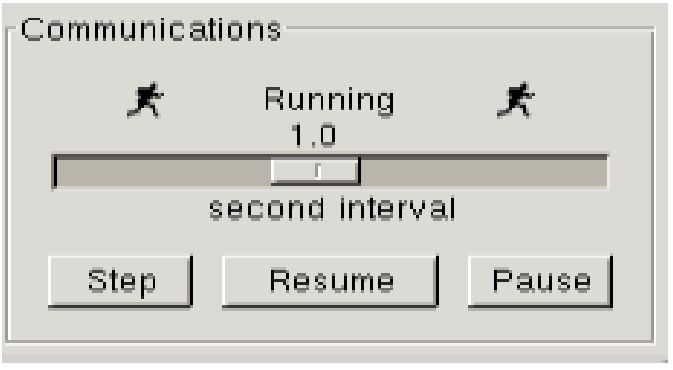

FIGURE 3.3: Communications

Moving the slider to the right to increase the execution speed and moving the slider to the left to slow down the execution.

The Step button will step through the execution of synchronization primitives.

The two buttons in the lower-left corner display the thread status and history.

3.2 Thread Visualization System

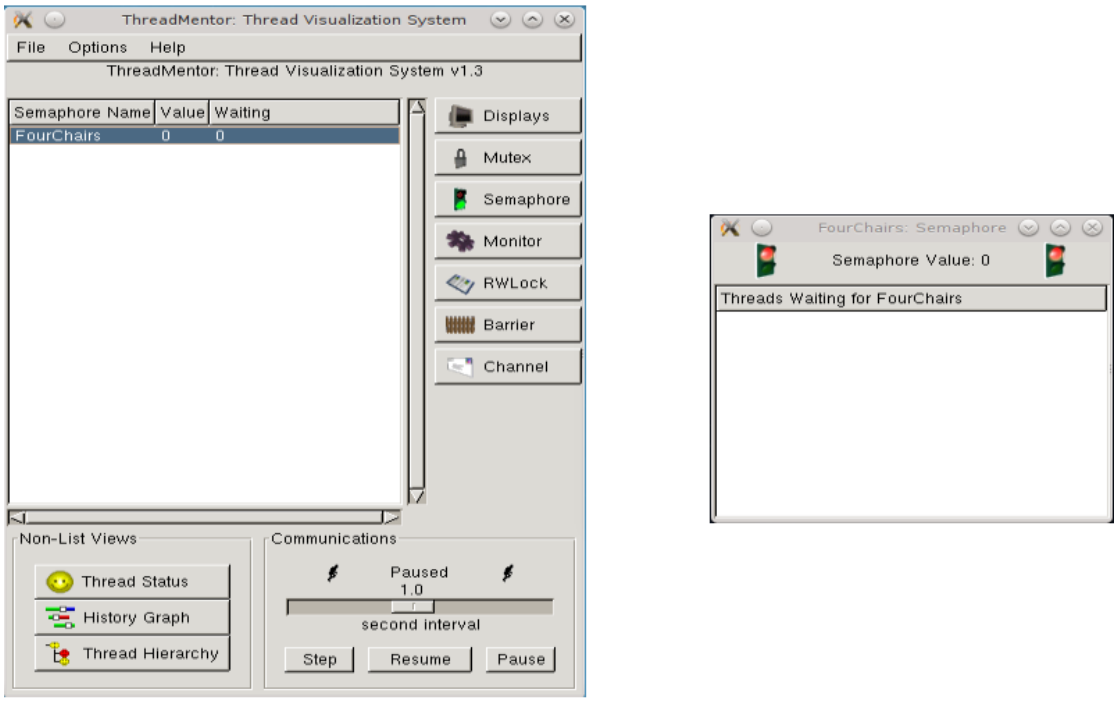

FIGURE 3.4: Thread Visualization System

There are three main windows that are very important when it comes to visualising the execution behaviour of semaphores. In the image above we can see two of these windows and what happens when you click on the semaphore button. It shows the semaphore we are running in the display window along with the value and how many threads are waiting. There are no threads waiting in this screen and it has a value of zero. However, when you click on the semaphore FourChairs name it brings up the FourChairs window which give you more information. Between the two red traffic lights we have the semaphore counter value which is zero. As the counter value is positive and the lights are red this means that the next thread will not have to wait. The other important window is the history graph which we will discuss in more detail further down in the report [8].

[8] https://pages.mtu.edu/ shene/NSF-3/e-Book/SEMA/VISUAL/VISUAL-sema.html Accessed 11th April 2021

3.3 Thread Status

On the Thread Mentor visualization main window on the bottom left-hand side in the non-list view we can see the button for the thread status. When we press this button, another screen opens to show all the threads that are available. Some of these include the main thread, running threads, joined thread, blocked threads, suspended threads, and the terminated threads. At each process we can see the name of each thread, and what process they are at. Each stage of the thread is represented by a different icon. In this image we can see that the threads philosopher 1, and philosopher 4 are running, while the threads philosopher 0, philosopher 2, and philosopher 3 are blocked. The main thread is joined which means it is waiting for its child thread to finish. None of our threads are in the suspended or terminated state. As the status of the thread keeps changing it will be illustrated in the thread status window as the change occurs and while the program keeps running [9].

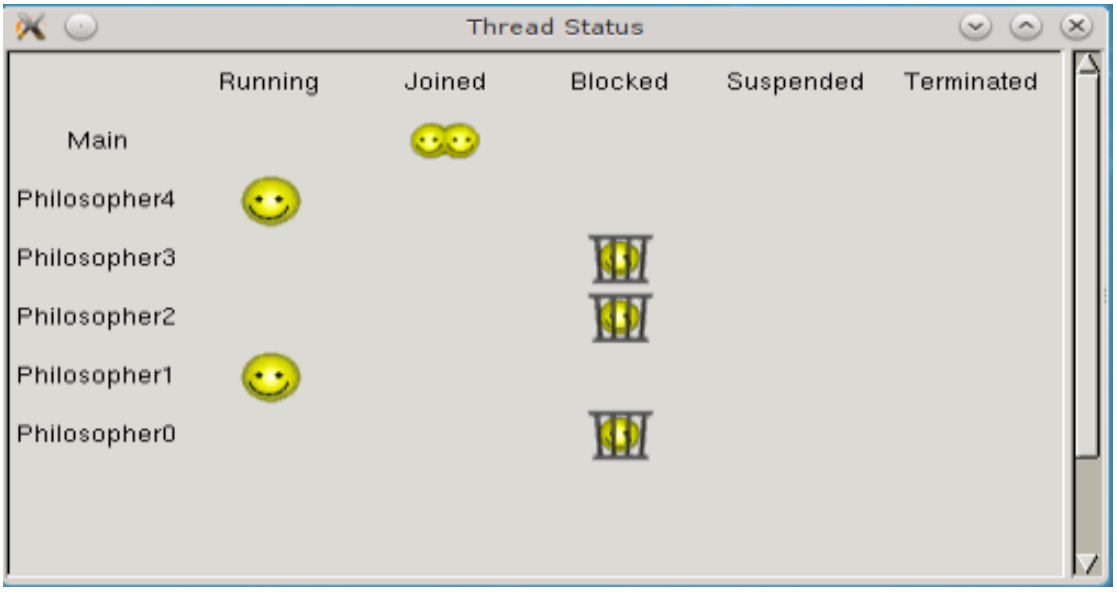

FIGURE 3.5: Thread Status

[9] https://pages.mtu.edu/ shene/NSF-3/e-Book/FUNDAMENTALS/VISUAL/basic.html

3.4 Thread Hierarchy

In the image below we can see the level of thread hierarchy. As in the Thread Status this is just another way of representing the threads in order of their priority. The main thread been the most important, and then philosopher 4 which is running. Philosopher 3, philosopher 2 is blocked while philosophers 1 is running, and philosopher 0 is blocked. They are in the same order as they appear in the history graph.

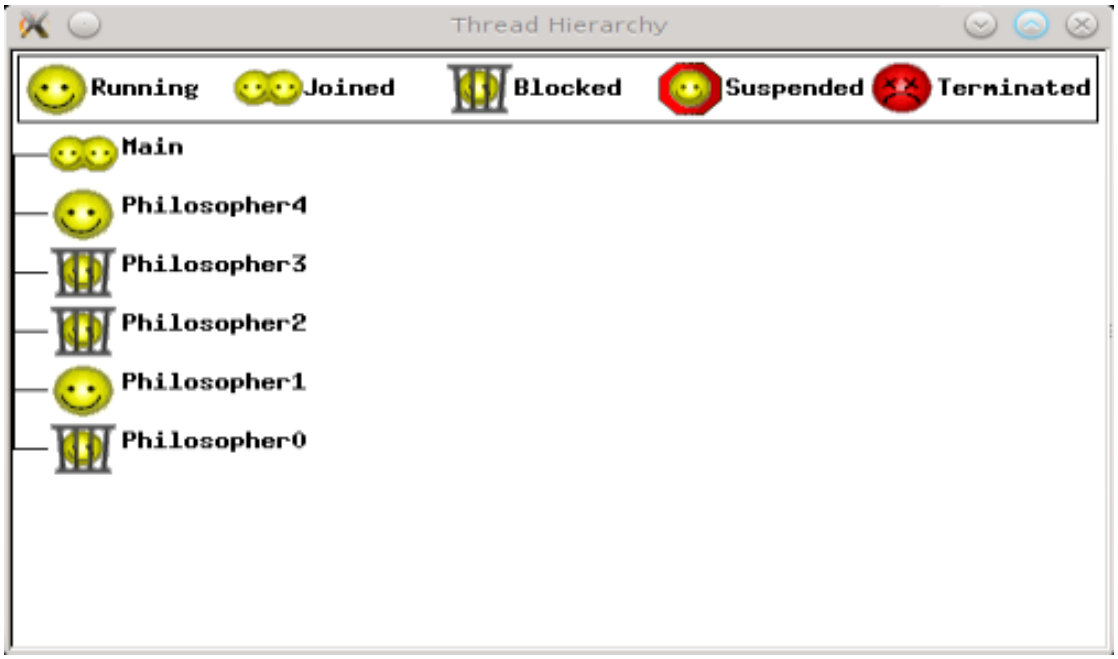

FIGURE 3.6: Thread Hierarchy

3.5 History Graph

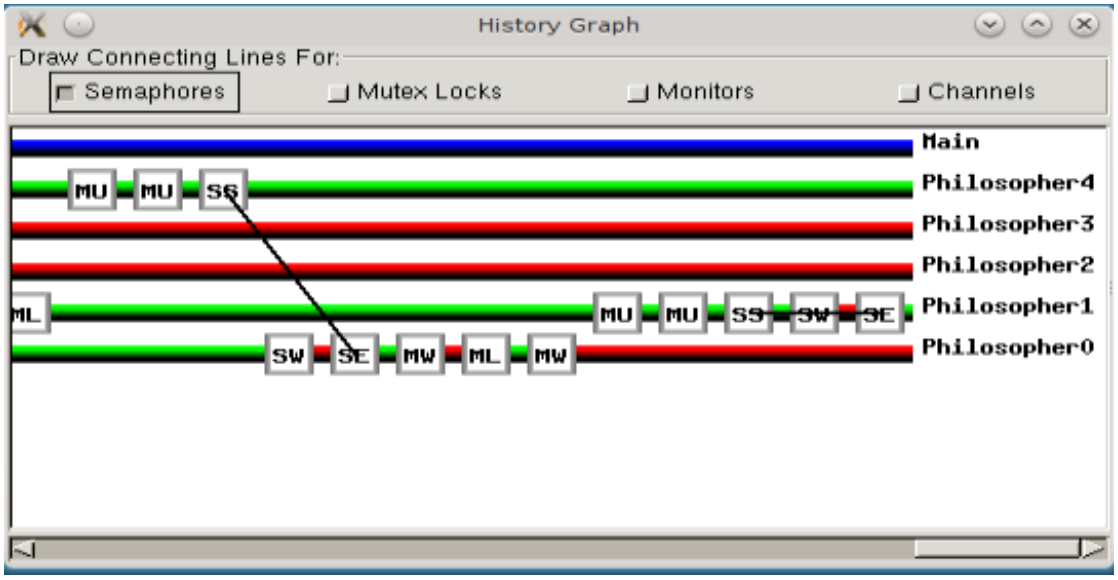

FIGURE 3.7: History Graph

The history graph displays all the threads in the order that they appear starting from the left to the right of the graph. The main thread is coloured blue and is the first thread to appear on the graph. Philosopher four is the next to appear, then philosopher three, then philosopher two and so on. The red colour shows the threads that are blocked, and the green represents the threads that are running. Also, on the graph we have loads of tags and on the top just below where the "draw connecting lines for" is we have four

check boxes. The first one is semaphore and if you click on it a black line appears linking the Semaphore Signal (SS) tag to the Semaphore Execute (SE) tag. This link is referred to as a signal exit link [10].

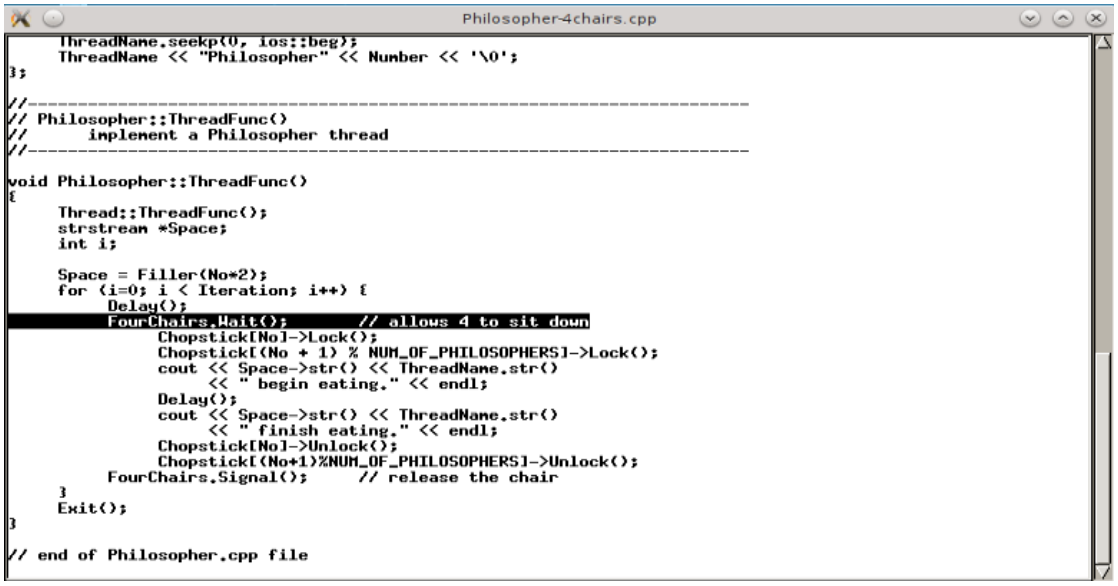

FIGURE 3.8: SW Tag

The philosopher zero thread in the history graph, if you click on the SW tag the screen above appears. This shows the source code, and the highlighted line is the code that is running. We can see that the wait function is been executed which in this case will allow philosopher four to sit down. The semaphore solution is an extension of the Mutual Exclusion Locks so we can see mutual lock and mutual unlock in our history graph. Each tag will bring up a window with the relevant source code highlighting the code that is been executed [10].

[10] https://pages.mtu.edu/ shene/NSF-3/e-Book/SEMA/VISUAL/VISUAL-sema.html Accessed 10th April 2021

3.6 Program End

When the program has finished and all processes and threads are terminated, the user will be presented with the screens below. Clicking Cancel will continue the program while clicking OK will close all the windows and quit the program.

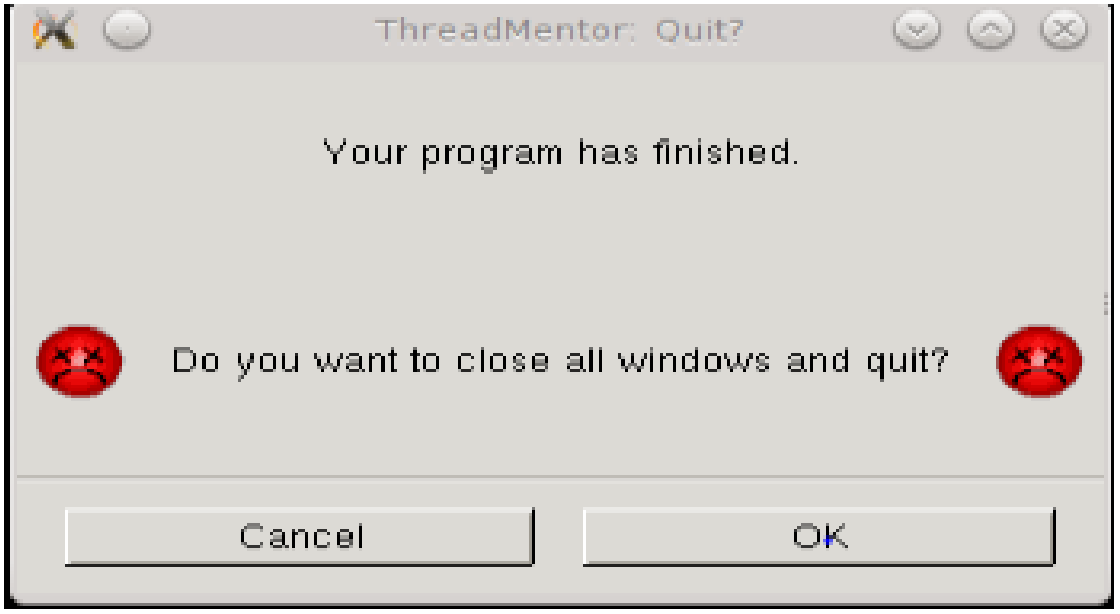

FIGURE 3.9: Program End

FIGURE 3.10: Thread Status End

Chapter 4

Conclusion and Future Work

4.1 Conclusions

In conclusion based on the research we have done and the diagrams we have presented, we can tell that the Dining Philosopher's problem can be a very complex problem and that there are many ways to solve this problem like the semaphore solution especially the four-chair version. With the four-chair solution and the use of semaphores you can easily solve the problem and prevent further problems like deadlock and starvation that can damage your findings. Also, with the help of the ThreadMentor software you can visualize and obtain a better understanding of not only the problem itself but also any solution you may wish to use.

As a result of busy waiting in semaphores there is no resource wastage. It only allows one process into the critical section at a time. However, wait and signal must be implemented in the correct order to prevent deadlock. Large scale use can lead to loss of modularity. It is more prone to programmer error [7c].

[7c] https://www.emblogic.com/blog/12/advantages-and-disadvantages-of-semaphores/

4.2 Future Work

As mentioned above we have only looked at one solution to this problem. Leading on from this we could review the other solutions and their code to see how they differ to the semaphore solution. By doing this we will have a better understanding of the Dining Philosophers Problem as well as all the solutions and how they work. This will give us an in-depth knowledge of the synchronisation and concurrency issues in the Dining Philosopher Problem. By running all the solutions, we will learn to use the Thread

Mentor visualisation tools and understand what is happening as the program runs in the terminal to the different ThreadMentor display windows. We have seen a wealth of information on the History graph alone and the tags that show the line of code as it has been executed. This information is shown when you click on the tag in the history graph. This will prepare us for other problems like Producer Consumer, the Smokers Problem and help us to understand what is happening as we are already familiar with using the Thread Mentor Visualisation Tools.

Chapter 5

Bibliography

Appendix A

Personal Reflection

A.1 Brian Maher

"I enjoyed this project and found it challenging and fun. I familiarised myself with a Linux distro that is new to me which is called Mageia. I also found it quite challenging to solve how to create a make-file for use with the chosen solution. Seeing the solution running on ThreadMentor was also a highlight as well as exploring the interface and seeing each stage of the threads in action."

A.2 Patricia Mc Keon

"When I started the module, I was a bit apprehensive as I was not familiar with Thread Mentor or Megia. I had used Ubuntu Linux before but just the basic for setting up a small network of three laptops, although some of the commands are the same. The operating system module is different in the sense that I was looking at a solution to a problem I never heard of before starting the module.

It was very interesting just to see what is happening in the background when a process is running and the wealth of information that is available. The make file, bash file and path file took a bit of getting use too. The thread mentor visualization system was simple enough to manoeuvre around, however, to interpret that information was a little tricky. Getting use to Emacs was the biggest hurdle, and I need to do more work using emacs to get familiar with it. The history graph had loads of very useful information, the tags showed the source code that is been executed when you click on the tag.

To see the Semaphore solution to the Dining Philosopher Problem up and running at the end of our lab work was a great achievement. I really enjoyed the work; it would

have been great if we had a bit more time to spend working on the problem and the other solutions. I hope to try and run the other solutions when I have more free time to do so."

A.3 Dahir Mussa

"I enjoyed this project, I learned a lot from this project, I improved my knowledge of the dining philosopher problem and also C++ programming. I found the challenges interesting and also finding a way to solve them, I was already familiar with the thread mentor Visualisation Tools and also some of the solutions such as Mutual Exclusion Locks, this time I have increased my knowledge of the other solution such as Monitors and Semaphores in which we have done in our project. It was interesting to learn different solution to the dining philosopher problem and different ways to solve them."

A.4 Robert Preston

"When this subject began at the start of the semester, I felt worried because I had no knowledge or understanding of the ThreadMentor software or even the Mageia operating system. This was the first time I used a Linux based system and I felt a bit confused at first as I was struggling with the different kind of features that can be used and when I first saw the Dining Philosopher's Problem, I got even more confused.

However, as I progressed through the semester, I became less confused with each day passing and completely interested in this project. It was very interesting just even learn about its history and background to learn how this problem was created. When I first created the necessary files I needed, it took some time to get use it but with the help of my teammates' I was able to fully understand the function of each file created. The ThreadMentor Visual System was, however, my favorite part of the project the interface was easy to operate, and it was easy to understand the information it presented. When I first saw, our chosen solution running on ThreadMentor I saw this as our ultimate achievement. I absolutely enjoyed the work that was done, I enjoyed working with my team mates even more and I hope to do this kind of work again soon."

Appendix B

Group Project Meeting Minutes

Saturday 06/03/2021

Group Members:
Patricia Mc Keon (B00123338), Robert Preston (B00128295), Brian Maher (B00123312), Dahir Mussa (B00107811).

Chairperson: Brian Maher
Attendees: All members were in attendance.
Meeting Minutes: Dahir Mussa

Time: 7:00 pm to 7:50 pm **Agenda:** Discuss the Mid-Semester Progress PowerPoint Presentation.

Minutes of Meeting

Brian	Welcomed everyone to the meeting.
Group	Everyone had looked over their slides and we timed how long it is going to take us to finish the presentation.
Brian	The Chosen Solution, and explain what Deadlocks and Starvation is.
Patricia	Visualizing, History graph, Conclusion, and Reference.
Robert	Will example about the thread interface tool.
Dahir	How to run the program, running program.
Patricia	Meeting is ended.

Next Meeting

Meeting	Tuesday 09/03/2021 at 7pm

FIGURE B.1: Meeting Minutes - Saturday 6th March 2021

Tuesday 09/03/2021

Group Members:
Patricia Mc Keon (B00123338), Robert Preston (B00128295), Brian Maher (B00123312), Dahir Mussa (B00107811).

Chairperson: Patricia Mc Keon
Attendees: All members were in attendance.
Meeting Minutes: Patricia Mc Keon

Time: 7:00 pm to 7:45 pm **Agenda:** Run through the Mid-Semester PowerPoint Presentation and upload it to Moodle.

Minutes of Meeting

Patricia	Welcomed everyone to the meeting.
Group	Everyone looked over their slides with the intention of keeping their section to just slightly over a minute each. We need to keep under the 5-minute limit for the overall presentation.
Brian	Is covering the slides on the Chosen Solution, he will explain what Deadlocks and Starvation is and a general overview of the presentation.
Robert	Will explain the tools we will use along with the Thread Mentor interface buttons and what they do.
Dahir	Will explain how to run the program (four-chair solution) and what is happening when it is running.
Patricia	Will explain what is happening with the ThreadMentor visualization tool and what is happening in the History graph. She will end with what our next step is.
Group	Meeting is ended.

FIGURE B.2: Meeting Minutes - Tuesday 9th March 2021

Wednesday 31/03/2021

Group Members:
Patricia Mc Keon (B00123338), Robert Preston (B00128295), Brian Maher (B00123312), Dahir Mussa (B00107811).

Chairperson: Brian Maher
Attendees: All members were in attendance.
Meeting Minutes: Brian Maher

Time: 7:00 pm to 7:55 pm **Agenda:** Discuss the Final-Semester Report.

Minutes of Meeting

Brian	Welcomed everyone to the meeting.
Group	Make progress on the completion of the report on overleaf.
Brian	Made progress on the ThreadMentor Interface section and references.
Patricia	Patricia will work on the tread status and hierarchy for the next meeting.
Robert	Completed his work on the introduction and will contribute to other sections of the report.
Dahir	How to run the program, running program completed and made some changes to the heading structure of the report.
Patricia	Meeting is ended.

Next Meeting

Meeting	Saturday 03/04/2021 at 7pm

FIGURE B.3: Meeting Minutes - Wednesday 31st March 2021

Saturday 03/04/2021

Group Members:
Patricia Mc Keon (B00123338), Robert Preston (B00128295), Brian Maher (B00123312), Dahir Mussa (B00107811).

Chairperson: Dahir Mussa
Attendees: All members were in attendance.
Meeting Minutes: Robert Preston

Time: 7:00 pm to 8.00 pm **Agenda:** Discuss the next set of chapters for the report.

Minutes of Meeting

Dahir	Welcomed everyone to the meeting.
Group	Everyone had looked over their current work on the final report.
Brian	Explained the interface of the thread mentor.
Patricia	Visualizing of the thread mentor, history graph and the future work.
Robert	Will example about the thread interface tool.
Dahir	Running of the program.
Group	Assigned the next set of chapters
Dahir	Meeting is ended.

Next Meeting

Meeting	Tuesday 06/04/2021 at 7pm

FIGURE B.4: Meeting Minutes - Saturday 3rd April 2021

Appendix C

Project Gantt Chart

This Gantt chart maps our progress from when we started in January to when we will finish in April.

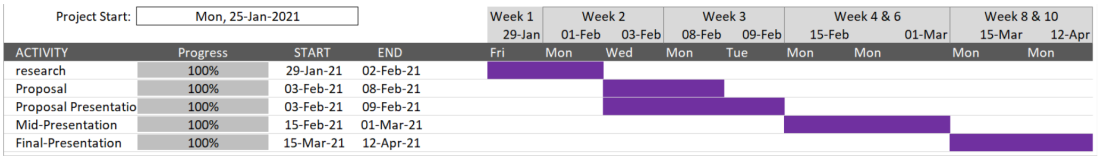

FIGURE C.1: Project Gantt Chart

Appendix D

Semaphore Solution Code

D.1 Philosopher-4chairs-main.cpp

```
//-------------------------------------------------------------------------
// Filename:
//     Philosopher-4chairs-main.cpp
// PROGRAM DISCRIPTION
//     Main program, this program uses mutex locks and semaphores to
//     implement the philosopher problem.  It allows at most four
//     philosophers to sit down and eat.
//-------------------------------------------------------------------------

#include <iostream>
#include <stdlib.h>

#include "Philosopher-4chairs.h"

//-------------------------------------------------------------------------
// main() function
//-------------------------------------------------------------------------

int main(int argc, char *argv[])
{
     Philosopher *Philosophers[NUM_OF_PHILOSOPHERS];
     int i, Iteration;

     if (argc != 2) {
        cout << "Use " << argv[0] << " #-of-iterations." << endl;
          exit(0);
        }
     else
          Iteration = abs(atoi(argv[1]));

     // create all the philosopher threads and fire them up
     for (i = 0; i < NUM_OF_PHILOSOPHERS; i++) {
        Philosophers[i]= new Philosopher(i, Iteration);
```

```
          Philosophers[i]->Begin();
     }

        // wait for philosopher thread
     for (i = 0; i < NUM_OF_PHILOSOPHERS; i++)
        Philosophers[i]->Join();

     Exit();   // main thread exits

     return 0;
}
```

D.2 Philosopher-4chairs.cpp

```
//----------------------------------------------------------------------
// Filename:
//     Philosopher-4chairs.cpp
// PROGRAM DISCRIPTION
//     Philosopher class implementation file.  This program uses mutex
//     locks and semaphores to implement the philosopher problem.
//     At most four philosophers can be sitting and eating.
//----------------------------------------------------------------------

#include <iostream>

#include "Philosopher-4chairs.h"

// locks and semaphores

static Semaphore FourChairs("FourChairs", NUM_OF_PHILOSOPHERS - 1);
static Mutex Chopstick1("Chopstick1");
static Mutex Chopstick2("Chopstick2");
static Mutex Chopstick3("Chopstick3");
static Mutex Chopstick4("Chopstick4");
static Mutex Chopstick5("Chopstick5");
static Mutex *Chopstick[NUM_OF_PHILOSOPHERS] =
    { &Chopstick1, &Chopstick2, &Chopstick3, &Chopstick4, &Chopstick5 };

// ---------------------------------------------------------------------
// FUNCTION  Filler():
//    This function fills a strstream with spaces.
// ---------------------------------------------------------------------

strstream *Filler(int n)
{
     int  i;
     strstream *Space;

     Space = new strstream;
     for (i = 0; i < n; i++)
          (*Space) << ' ';
```

```
     (*Space) << '\0';
     return Space;
}

//--------------------------------------------------------------------------
// Philosopher::Philosopher()
//      constructor
//--------------------------------------------------------------------------

Philosopher::Philosopher(int Number, int iter)
            :No(Number), Iteration(iter)
{
     ThreadName.seekp(0, ios::beg);
     ThreadName << "Philosopher" << Number << '\0';
};

//--------------------------------------------------------------------------
// Philosopher::ThreadFunc()
//      implement a Philosopher thread
//--------------------------------------------------------------------------

void Philosopher::ThreadFunc()
{
     Thread::ThreadFunc();
     strstream *Space;
     int i;

     Space = Filler(No*2);
     for (i=0; i < Iteration; i++) {
          Delay();
          FourChairs.Wait();       // allows 4 to sit down
               Chopstick[No]->Lock();
               Chopstick[(No + 1) % NUM_OF_PHILOSOPHERS]->Lock();
               cout << Space->str() << ThreadName.str()
                    << " begin eating." << endl;
               Delay();
               cout << Space->str() << ThreadName.str()
                    << " finish eating." << endl;
               Chopstick[No]->Unlock();
               Chopstick[(No+1)%NUM_OF_PHILOSOPHERS]->Unlock();
          FourChairs.Signal();     // release the chair
     }
     Exit();
}

// end of Philosopher.cpp file
```

D.3 Philosopher-4chairs.h

```
//--------------------------------------------------------------------------
// Filename:
```

```
//     Philosopher-4chairs.h
// PROGRAM DISCRIPTION
//     Interface file for class Philosopher
//---------------------------------------------------------------------------

#ifndef _PHILOSOPHER_H
#define _PHILOSOPHER_H

#include "ThreadClass.h"

// macro defines

#define NUM_OF_PHILOSOPHERS   5

//---------------------------------------------------------------------------
// Philosopher class definition
//---------------------------------------------------------------------------

class Philosopher: public Thread
{
     public:
          Philosopher(int Number, int iter);  // constructor
     private:
          int No;     // position of the philosopher
          int Iteration;
          void ThreadFunc();
};

#endif
```

D.4 Makefile

```
# ------------------------------------------------------------

# Makefile
#
# Options:
# 1. TMLIB = /home/pmckeon/os/ThreadMentor/Visual/libthreadclass.a
# 2. TMLIB = /home/pmckeon/os/ThreadMentor/NoVisual/libthreadclass.a
# ------------------------------------------------------------

CC        = c++
CFLAGS    = -g -O2 -Wno-deprecated -m32
DFLAGS    = -DPACKAGE=\"threadsystem\"
-DVERSION=\"1.0\" -DPTHREAD=1 -DUNIX_MSG_Q=1 -DSTDC_HEADERS=1
IFLAGS    = -I/home/pmckeon/os/ThreadMentor/include
TMLIB     = /home/pmckeon/os/ThreadMentor/Visual/libthreadclass.a

OBJ_FILE  = Philosopher-4chairs.o Philosopher-4chairs-main.o
EXE_FILE  = Philosopher-4chairs

${EXE_FILE}: ${OBJ_FILE}
```

```
        ${CC} ${CFLAGS}  -o ${EXE_FILE}  ${OBJ_FILE} ${TMLIB} -lpthread

Philosopher-4chairs.o : Philosopher-4chairs.cpp Philosopher-4chairs.h
        ${CC} ${DFLAGS} ${IFLAGS} ${CFLAGS} -c Philosopher-4chairs.cpp

Philosopher-4chairs-main.o: Philosopher-4chairs-main.cpp Philosopher-4chairs.h
        ${CC} ${DFLAGS} ${IFLAGS} ${CFLAGS} -c Philosopher-4chairs-main.cpp

clean:
        rm -f ${OBJ_FILE} ${EXE_FILE}

# End of Makefile
```

D.5 .bashrc

```
# .bashrc

# User specific aliases and functions

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

PATH=$PATH:/home/student/os/ThreadMentor/bin
export PATH# .bashrc

# User specific aliases and functions

# Source global definitions
if [ -f /etc/bashrc ]; then
. /etc/bashrc
fi

PATH=$PATH:/home/student/os/ThreadMentor/bin
export PATH
```

Appendix E

Appendix E - Fact Sheet

E.1 Group Members

- Patricia Mc Keon - B00123338
- Dahir Mussa - B00107811
- Brian Maher - B00123312
- Robert Preston - B00128295

E.2 Meeting Minutes

- 6th March 2021 - Dahir Mussa
- 9th March 2021 - Patricia Mc Keon
- 31st March 2021 - Brian Maher
- 3rd April 2021 - Robert Preston

E.3 Report Sections

E.3.1 Chapter 1 - Introduction

- Introduction - Robert Preston
- Background - Patricia Mc Keon
- Types of Solutions to this Problem - Patricia Mc Keon

- Semaphores- Four Chair Solution - Dahir Mussa
- Deadlock - Brian Maher
- Starvation - Brian Maher

E.3.2 Chapter 2 - Running the Program

- How to Run the Program - Dahir Mussa
- Makefile - Dahir Mussa
- The PATH Environment Variable - Dahir Mussa

E.3.3 Chapter 3 - Visualization

- Thread Mentor Interface - Brian Maher
- Thread Visualization System - Patricia Mc Keon
- Thread Status - Patricia Mc Keon
- Thread Hierarchy - Patricia Mc Keon
- History Graph - Patricia Mc Keon
- Program End - Brian Maher

E.3.4 Chapter 4 – Conclusion and Future Work

- Conclusion - Robert Preston/Patricia Mc Keon
- Future Work - Patricia Mc Keon

E.3.5 Chapter 5 – Bibliography

- Bibliography - Brian Maher/Dahir Mussa

E.4 Other Information

- Appendix A: Personal Reflection - Group
- Appendix B: Meeting Minutes - Patricia Mc Keon