Operating Systems Module

Digital Forensics and Cyber Security (TU863)

Year 2, 2024/2025

OS Project Team Report

ThreadMentor:

Dining Philosophers Problem Solution

Authors:

Danyil Tymchuk (B00167321) & Artem Surzhenko (B00163362) & Emanuel Avram (B00167369)

17/04/2025

Declaration

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.

Abstract Page

This report presents a detailed study of the Dining Philosophers Problem, a classic concurrency problem used to demonstrate the subtleties of process synchronization in operating systems. The general objective is to examine synchronization mechanisms—specifically mutex locks—to prevent race conditions, deadlocks, and starvation in multi-threaded systems.

We begin by explaining the theoretical foundations of concurrency, synchronization requirements, and common mechanisms such as mutexes and semaphores. The report then focuses on explaining the implementation of the Dining Philosophers Problem using mutex locks, facilitated by the ThreadMentor visualization tool. ThreadMentor allows us to monitor thread execution, observe synchronization behavior, and verify the correctness of our solution.

Our study includes screenshots, execution traces, and detailed behavior observations of philosopher threads. We evaluate the presence of deadlocks or starvation, discuss program efficiency, and reflect on the concurrency issues encountered during the sudy. The project concludes with individual reflections, insights gained, and suggestions for further work in addressing synchronization problems.

This report attempts to provide a technical as well as educational overview of concurrency management in operating systems based on the method of one of its most well-documented problems.

Table of Contents

List of Figures

1 Introduction

Operating systems (OSes) play a key role in managing resources and facilitating the execution of multiple programs at the same time. One of the basic problems of OS design is correct synchronization of shared resources by multiple processes or threads, to avoid race conditions, deadlocks, and starvation of resources. Correct management of concurrency becomes increasingly vital as systems become increasingly powerful to support complex, multi-threaded applications.

We touch on the concurrency and synchronization practice and theory within operating systems and specifically into the Dining Philosophers Problem, one of the most common synchronization problems employed to exemplify the troubles involved in handling concurrent processes. We present an entire explanation of the problem along with a solution based on current synchronization techniques including mutex locks as well as semaphores. We also utilize ThreadMentor, a thread visualization and debugging tool, to monitor the performance of our solution.

1.1 Background

Operating systems are responsible for managing various resources of a computer system, e.g., CPU, memory, and I/O devices. Managing the execution of multiple threads or processes at once is perhaps the most crucial activity of an OS. However, when threads or processes need to access common resources, synchronization needs to be provided to prevent corruption of data and ensure system integrity.

1.1.1 Concurrency and Synchronisation Issues in OSes

Concurrency is when multiple threads or processes execute concurrently, either on multiple processors or by time-slicing on a single processor. While concurrency can improve performance and responsiveness, it introduces issues in ensuring that shared resources are accessed safely. Common issues are:

- **Race conditions**, where the output depends on the non-deterministic order of execution.
- **Deadlocks**, where processes become stuck waiting for each other indefinitely.
- **Starvation**, where certain processes are perpetually denied access to resources.

1.1.2 Mutex Locks

One of the most widely used synchronization mechanisms is the **mutex (short for mutual exclusion) lock**. A mutex ensures that only one thread can access a shared resource at a time. If one thread holds the mutex, other threads must wait until it is released. While mutexes are effective for preventing race conditions, they must be used carefully to avoid introducing deadlocks, where two or more threads are each waiting for the other to release a mutex.

1.1.3 Semaphores

A **semaphore** is another synchronization primitive that provides more flexibility than mutexes. Semaphores are typically used to control access to a pool of resources. They consist of an integer value that indicates the number of available resources. Threads can increment or decrement the semaphore value to acquire or release a resource. Semaphores can help address more complex synchronization scenarios, such as managing multiple threads accessing a fixed number of resources.

1.1.4 ThreadMentor

ThreadMentor is a tool used to visualize and debug multi-threaded programs. It allows developers to track thread execution and observe synchronization issues in real time. With ThreadMentor, we can visualize the execution history of threads, highlight issues such as deadlocks or race conditions, and verify that our synchronization mechanisms, like mutexes and semaphores, are working as intended.

1.2 The Dining Philosophers Problem

The Dining Philosophers Problem is a classical problem in computer science that illustrates the challenges of managing concurrent processes and shared resources. The scenario involves a number of philosophers sitting at a table, each with a bowl of pasta and two forks. They need both forks to eat, but there are only as many forks as philosophers, leading to the potential for deadlock and resource contention. The problem requires careful synchronization to ensure that philosophers can eat without encountering deadlocks or starvation.

This problem serves as a valuable tool for teaching synchronization techniques and exploring the practical application of concepts such as mutexes, semaphores, and resource allocation strategies.

1.3 Outline/Layout of our Report

This report is designed to guide the reader from theory to practice, analysis, and reflection of the Dining Philosophers Problem using mutex-based synchronization in a multithreaded environment.

Section 2 covers the introduction of the Mutex Lock Solution, starting with the theory behind mutexes and how they prevent race conditions. We then present our implementation with ThreadMentor, explaining how the tool helped us to execute and visualize the solution.

Section 3 provides a thorough Results and Analysis. We begin with a Program Execution Overview (3.1) and move on to a step-by-step description of the execution window (3.2). We then provide ThreadMentor screenshots and graphs (3.3) accompanied by captions and code snippets corresponding to ThreadMentor tags (3.4). We then have a meticulous behavioral analysis of philosopher threads (3.5) and continue with discussion on the dangers of deadlock and starvation (3.6).

In **Section 4,** we state our Conclusions, where we talk about how effective the mutex lock approach is and how it handled synchronization problems.

Section 5 is the list of References that we employed in researching and carrying out the project.

Section 6 contains a reflexive section, where we provide a statement about what they have learned (6.1), what was enjoyable and unpleasant about the project (6.2–6.3), and what they would do differently if having to repeat the project (6.4). We also provide suggestions for future students addressing similar issues (6.5).

Finally, **Section 7** records our Project Planning and Management, describing the way we scheduled our time, allocated tasks, and worked harmoniously during the course of the assignment.

2 Mutex Lock Solution

As a group we chose the first Mutex Lock solution available on the threadmentor e-book website (Shene, No Date A). We chose this solution specifically because we found the concept of Mutex locks interesting and sufficiently challenging, and from the beginning we had thoughts and questions regarding the concurrency and synchronization issues that arise in this solution which are system deadlocks and starvation.

2.1 Theory/How it works

The mutex solution simulates 5 philosophers eating at the table, as any other solution likely would. However, here, the philosopher thread will pick up a chopstick, in other words a mutex lock, starting with the left one. Only after acquiring the left chopstick will it attempt to acquire the right one in order to have both chopsticks required to eat the spaghetti or shared resource available to all 5 threads.

Our solution uses a header file called “Philosopher.h” to define the number of philosophers used in the simulation, and declares a Philosopher constructor that will take integers that assigns numbers or “names” to each thread to help with visualization, and another integer to dictate how many cycles of thinking and eating a philosopher will have.

Another file “Phlosopher.cpp”, is used to externally declare the mutex locks, has a Philosopher constructor to actually set the names of the philosophers (philosopher0, philosopher1, etc.) and the number of iterations a philosopher will loop through. It also has the ThreadFunc() function inside which runs when a thread starts. The function simulates what a philosopher will do. Picking up left then right chopsticks by locking the mutex locks, thinking and eating by using the Delay() function and putting the chopsticks back down by releasing the locks. Essentially it is the main loop of each thread when executed, the iteration variable dictates how many times this loop will run.

Finally there is the “Philosopher-main.cpp” file, the main program which puts everything in motion. Outside of the main() function, there is an array of pointers to the Mutex objects. Inside the main() function, the command line is first checked for its one and only possible argument, which is iterations or how many times the loop inside ThreadFunc() should be run. Afterwards the Mutexes are created just before the creation and running of the threads. Then the threads are joined in order for all threads to finish before the simulation concludes.

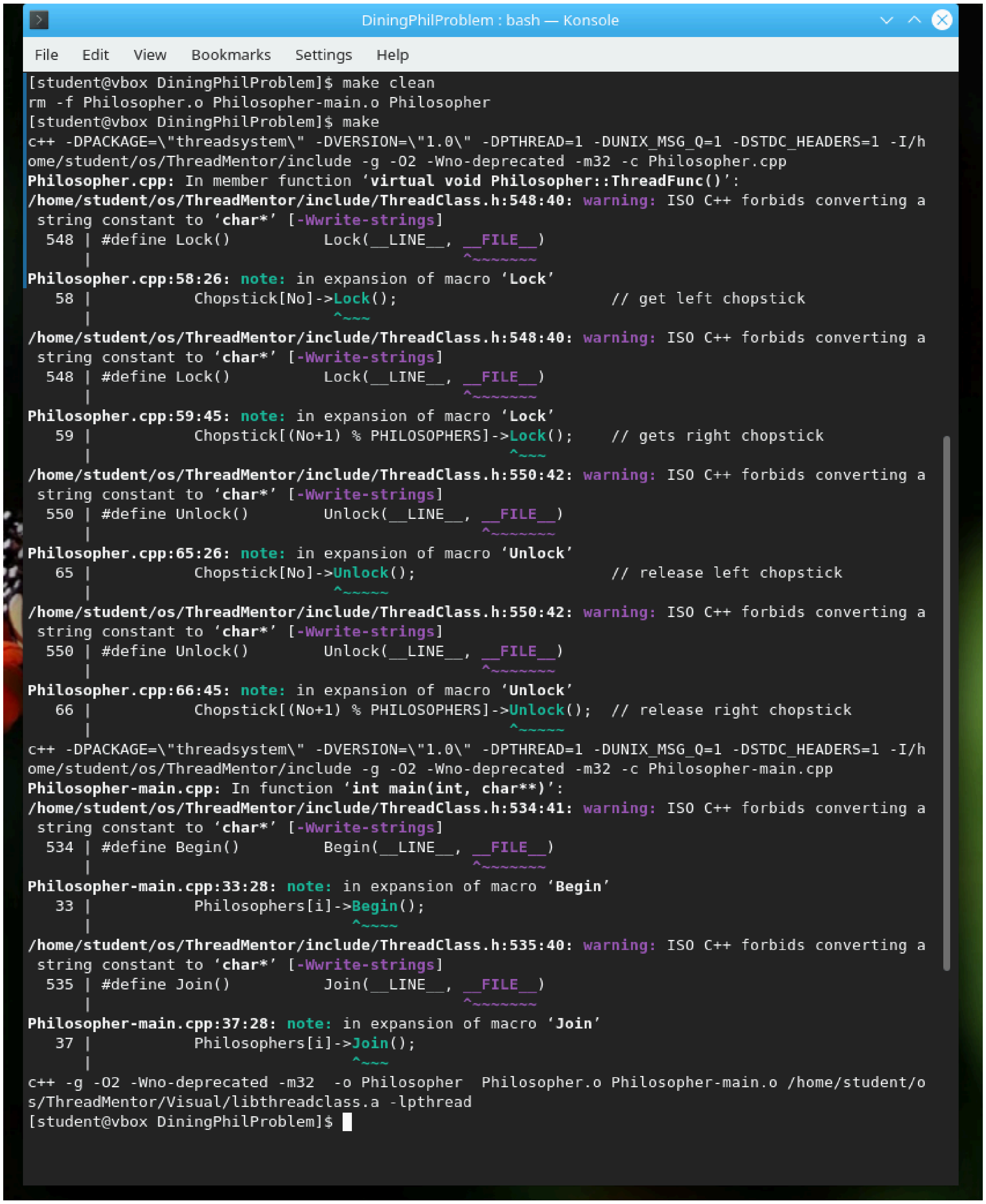
2.2 ThreadMentor

ThreadMentor is a system made to aid learning about threaded programming and the use of synchronization primitives such as Mutex Locks, Semaphores, etc. There is a class library included that works together with the ThreadMentor visualization system that allows the user to view and monitor the behavior of the synchronization primitives in an intuitive and easy to understand manner (Shene, No Date B).
In order to compile and run our solution, we had to start off by setting up threadMentor in the first place.

We did this firstly by downloading the required dependency packages in the appropriate directory, and then installing the files. Afterwards the ThreadMentor gzipped tarball file that contained all ThreadMentor files we downloaded earlier with the dependency files could be extracted into its own folder. Lastly we set the PATH environment variable so that the necessary commands to run ThreadMentor could be found by the bash.

At this point ThreadMentor has been set up, so all we had to do was download our Mutex solution files, 2 .cpp files and a header file, and of course the makefile that compiles the code. Then in the directory with all the solution files, we used the “make” command to compile the program.

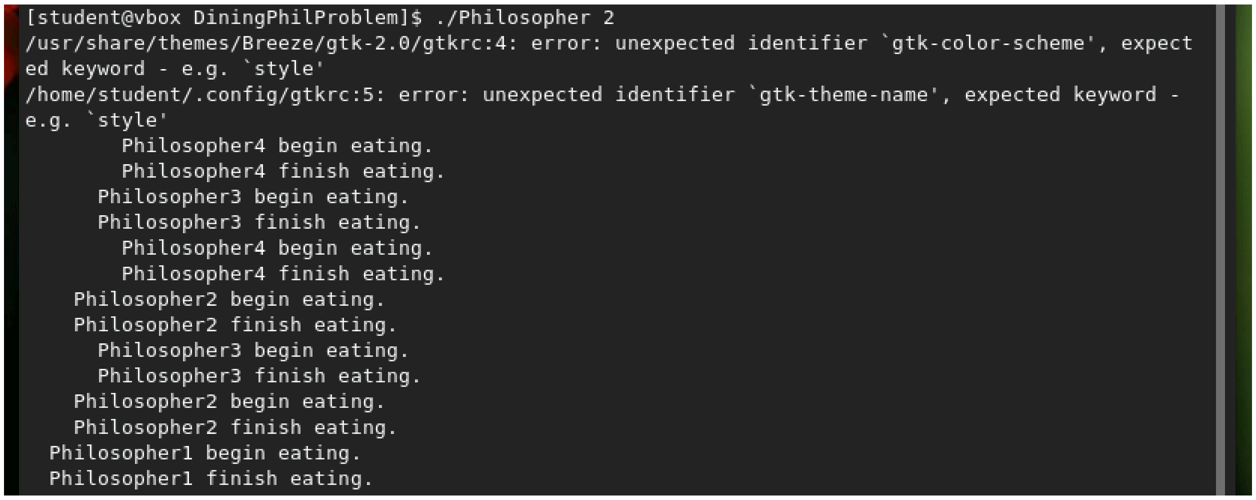
Console output after “make” command:


DiningPhilProblem : bash — Konsole
File Edit View Bookmarks Settings Help

```
[student@vbox DiningPhilProblem]$ make clean
rm -f Philosopher.o Philosopher-main.o Philosopher
[student@vbox DiningPhilProblem]$ make
c++ -DPACKAGE=\"threadsystem\" -DVERSION=\"1.0\" -DPTHREAD=1 -DUNIX_MSG_Q=1 -DSTDC_HEADERS=1 -I/h
ome/student/os/ThreadMentor/include -g -O2 -Wno-deprecated -m32 -c Philosopher.cpp
Philosopher.cpp: In member function ‘virtual void Philosopher::ThreadFunc()’:
/home/student/os/ThreadMentor/include/ThreadClass.h:548:40: warning: ISO C++ forbids converting a
 string constant to ‘char*’ [-Wwrite-strings]
  548 | #define Lock()          Lock(__LINE__, __FILE__)
      |                                        ^~~~~~~~
Philosopher.cpp:58:26: note: in expansion of macro ‘Lock’
   58 |          Chopstick[No]->Lock();                       // get left chopstick
      |                         ^~~~
/home/student/os/ThreadMentor/include/ThreadClass.h:548:40: warning: ISO C++ forbids converting a
 string constant to ‘char*’ [-Wwrite-strings]
  548 | #define Lock()          Lock(__LINE__, __FILE__)
      |                                        ^~~~~~~~
Philosopher.cpp:59:45: note: in expansion of macro ‘Lock’
   59 |          Chopstick[(No+1) % PHILOSOPHERS]->Lock();    // gets right chopstick
      |                                            ^~~~
/home/student/os/ThreadMentor/include/ThreadClass.h:550:42: warning: ISO C++ forbids converting a
 string constant to ‘char*’ [-Wwrite-strings]
  550 | #define Unlock()        Unlock(__LINE__, __FILE__)
      |                                          ^~~~~~~~
Philosopher.cpp:65:26: note: in expansion of macro ‘Unlock’
   65 |          Chopstick[No]->Unlock();                     // release left chopstick
      |                         ^~~~~~
/home/student/os/ThreadMentor/include/ThreadClass.h:550:42: warning: ISO C++ forbids converting a
 string constant to ‘char*’ [-Wwrite-strings]
  550 | #define Unlock()        Unlock(__LINE__, __FILE__)
      |                                          ^~~~~~~~
Philosopher.cpp:66:45: note: in expansion of macro ‘Unlock’
   66 |          Chopstick[(No+1) % PHILOSOPHERS]->Unlock();  // release right chopstick
      |                                            ^~~~~~
c++ -DPACKAGE=\"threadsystem\" -DVERSION=\"1.0\" -DPTHREAD=1 -DUNIX_MSG_Q=1 -DSTDC_HEADERS=1 -I/h
ome/student/os/ThreadMentor/include -g -O2 -Wno-deprecated -m32 -c Philosopher-main.cpp
Philosopher-main.cpp: In function ‘int main(int, char**)’:
/home/student/os/ThreadMentor/include/ThreadClass.h:534:41: warning: ISO C++ forbids converting a
 string constant to ‘char*’ [-Wwrite-strings]
  534 | #define Begin()         Begin(__LINE__, __FILE__)
      |                                         ^~~~~~~~
Philosopher-main.cpp:33:28: note: in expansion of macro ‘Begin’
   33 |          Philosophers[i]->Begin();
      |                           ^~~~~
/home/student/os/ThreadMentor/include/ThreadClass.h:535:40: warning: ISO C++ forbids converting a
 string constant to ‘char*’ [-Wwrite-strings]
  535 | #define Join()          Join(__LINE__, __FILE__)
      |                                        ^~~~~~~~
Philosopher-main.cpp:37:28: note: in expansion of macro ‘Join’
   37 |          Philosophers[i]->Join();
      |                           ^~~~
c++ -g -O2 -Wno-deprecated -m32  -o Philosopher  Philosopher.o Philosopher-main.o /home/student/o
s/ThreadMentor/Visual/libthreadclass.a -lpthread
[student@vbox DiningPhilProblem]$ 
```

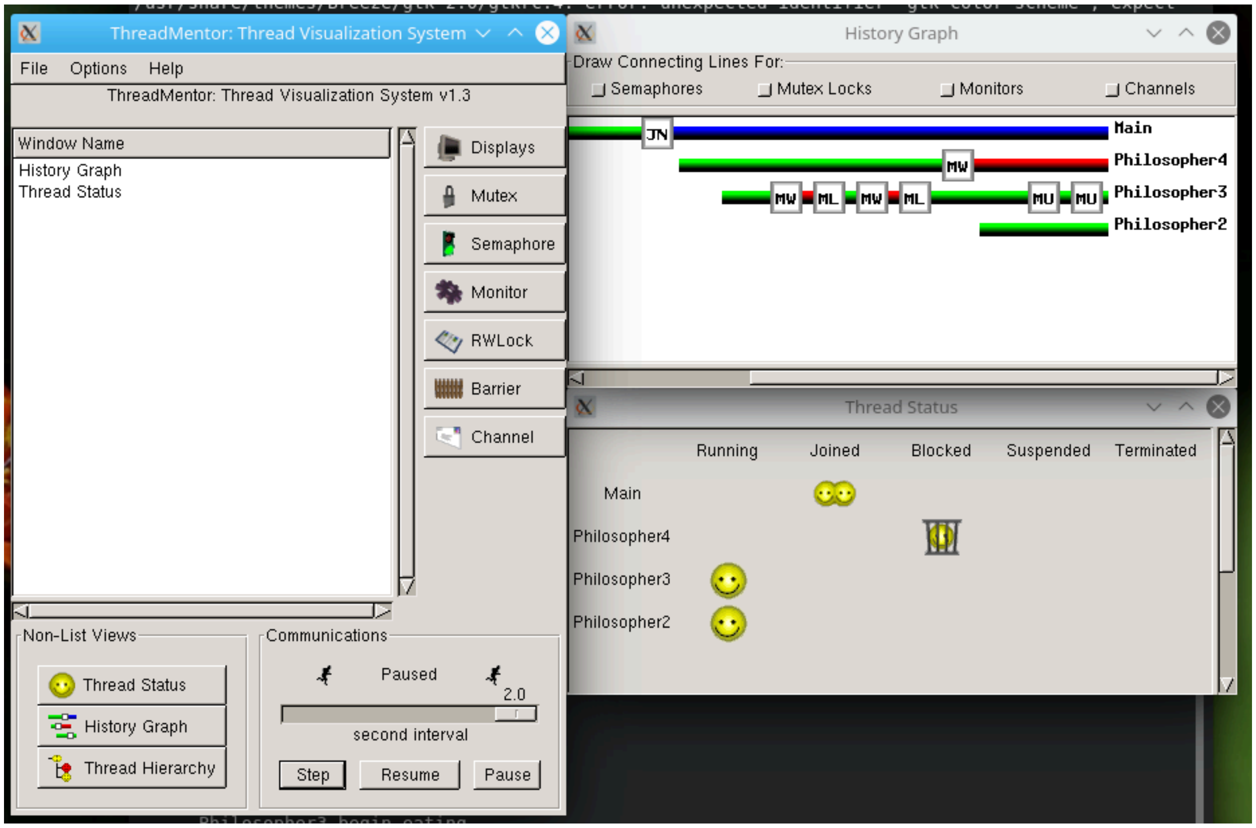
After compiling, we executed the executable Philosopher file by using the ./Philosopher command with an integer as the one and only argument that dictates how many times the main function will loop, or how many times the philosophers will complete a cycle of eating and thinking.

Console output after executing Solution with “2” as integer argument:



```
[student@vbox DiningPhilProblem]$ ./Philosopher 2
/usr/share/themes/Breeze/gtk-2.0/gtkrc:4: error: unexpected identifier `gtk-color-scheme', expect
ed keyword - e.g. `style'
/home/student/.config/gtkrc:5: error: unexpected identifier `gtk-theme-name', expected keyword -
e.g. `style'
        Philosopher4 begin eating.
        Philosopher4 finish eating.
      Philosopher3 begin eating.
      Philosopher3 finish eating.
        Philosopher4 begin eating.
        Philosopher4 finish eating.
    Philosopher2 begin eating.
    Philosopher2 finish eating.
      Philosopher3 begin eating.
      Philosopher3 finish eating.
    Philosopher2 begin eating.
    Philosopher2 finish eating.
  Philosopher1 begin eating.
  Philosopher1 finish eating.
```

Threadmentor execution after previous command:



2.2.1 Makefile

When attempting to run a threaded program using ThreadMentor, using a makefile helps to automate the compilation process of said program. What a makefile essentially does is selects what files to compile and how to go about doing that. In our case we have the 2 .cpp (c++) files and a .h header file, instead of having to compile and run each file separately, the makefile says what files depend on others and builds the program according to the set of instructions provided (Yigit Ogun, 2022).

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

There are several ThreadMentor tags that are crucial to understand when using the ThreadMentor visualization system for our solution, most of them being on the history graph panel. The 4 main tags that are seen in our Mutex solution are: “JN” (Join), “MW” (Mutex Wait), “ML” (Mutex Lock), and “MU” (Mutex Unlock).

The consecutive JN tags or Join tags, when seen at the end of the history graph, means the main thread joins the child threads or the philosophers in this case so that the main thread does not terminate until all philosophers have terminated, preventing the program from terminating prematurely.

The MW or Mutex Wait tags indicate that a thread is in a blocked state and waiting to acquire a Mutex Lock. In a blocked state, a thread will only wait for a condition or resource to become available before it continues its execution.

The first ML or Mutex Lock tag indicated that a thread has acquired the left lock, a second ML tag would mean that both locks have been acquired and the thread can now access the shared resource, in other words the philosopher has picked up both chopsticks and can eat the spaghetti. Before a thread acquires the right or second mutex lock, it will yet again go back into a blocked state to wait for a second lock, so a MW tag appears between the two ML tags.

Finally the MU or Mutex Unlock tag, means that a thread has unlocked a mutex lock, if it is the first MU tag, that means the left chopstick has been released, and the second is the right. In our solution, once the left lock has been unlocked, the right one will unlock immediately afterwards.

3 Results and Analysis

This section presents the execution results of the Dining Philosophers Problem implemented using mutex locks. It includes screenshots from ThreadMentor illustrating the behavior of the program, along with detailed analysis of the Philosopher threads' states and transitions during execution.

3.1 Program Execution Overview

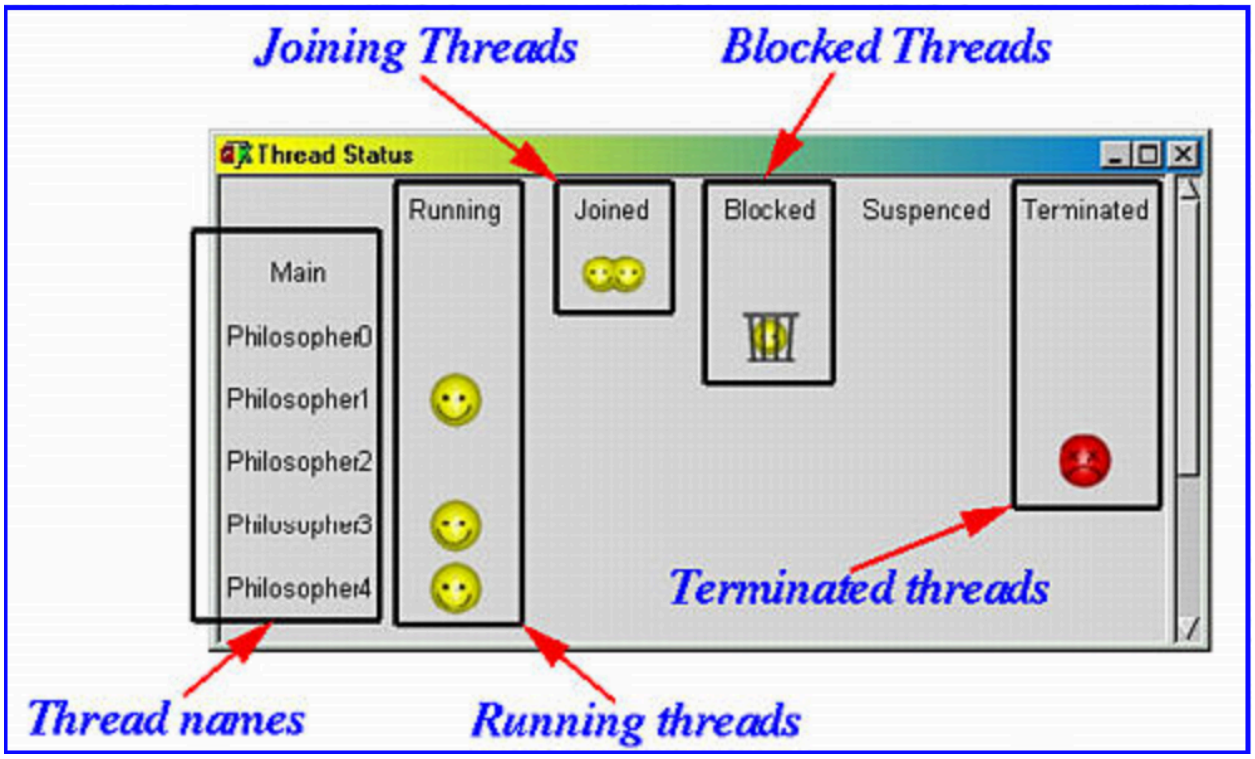
The Dining Philosophers program using mutex locks was executed with 5 philosopher threads and 5 chopsticks (mutex locks). Each philosopher was initialized with a unique ID (0-4) and configured to run through 3 iterations of the thinking-eating cycle. The program execution followed this general sequence:

1. **Initialization Phase**: The main program created 5 mutex locks representing chopsticks and 5 philosopher threads. Each philosopher was assigned their ID and positioned at the table with chopsticks to their left and right.
2. **Execution Phase**: Once started, each philosopher thread independently cycled through:
    - Thinking (simulated by a random delay)
    - Attempting to pick up the left chopstick (acquiring the left mutex)
    - Attempting to pick up the right chopstick (acquiring the right mutex)
    - Eating (simulated by another delay)
    - Putting down both chopsticks (releasing both mutexes)
3. **Termination Phase**: After completing their assigned iterations, each philosopher thread terminated, and the main program joined all threads before concluding the simulation.

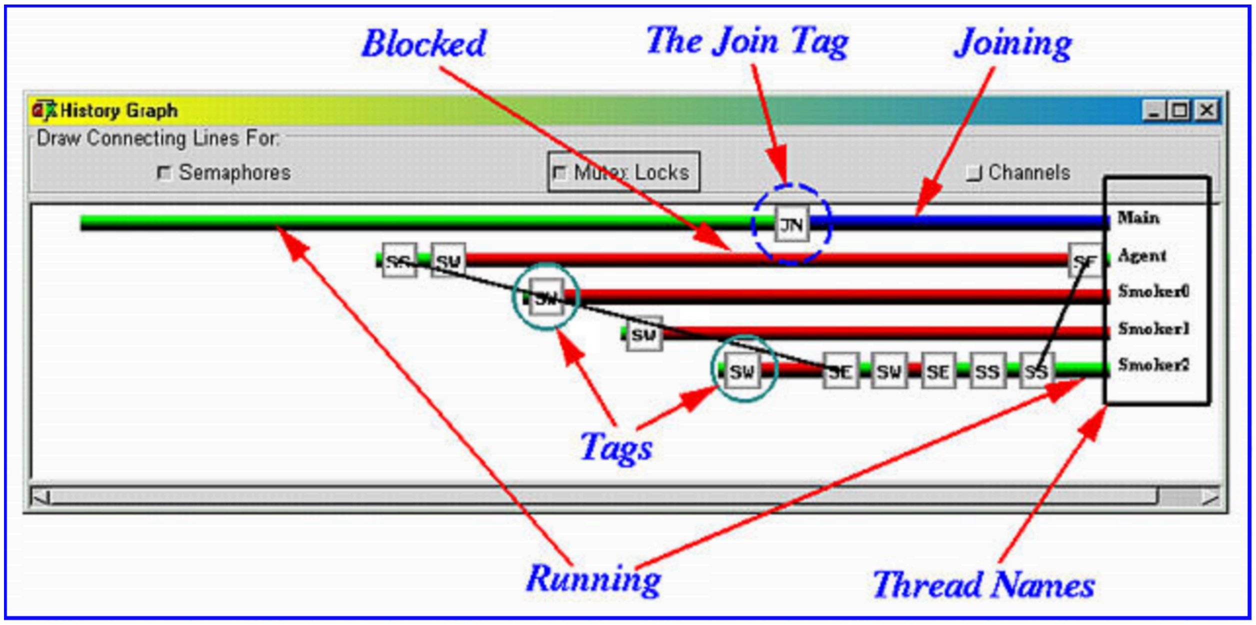
During execution, the philosophers exhibited varying patterns of activity and waiting states as they competed for the shared chopstick resources. Some philosophers successfully acquired both chopsticks quickly, while others experienced waiting periods when neighboring philosophers held their required chopsticks. This resource contention is precisely what makes the Dining Philosophers problem an excellent demonstration of synchronization challenges in concurrent systems.

3.2 Program Execution Window Explaining

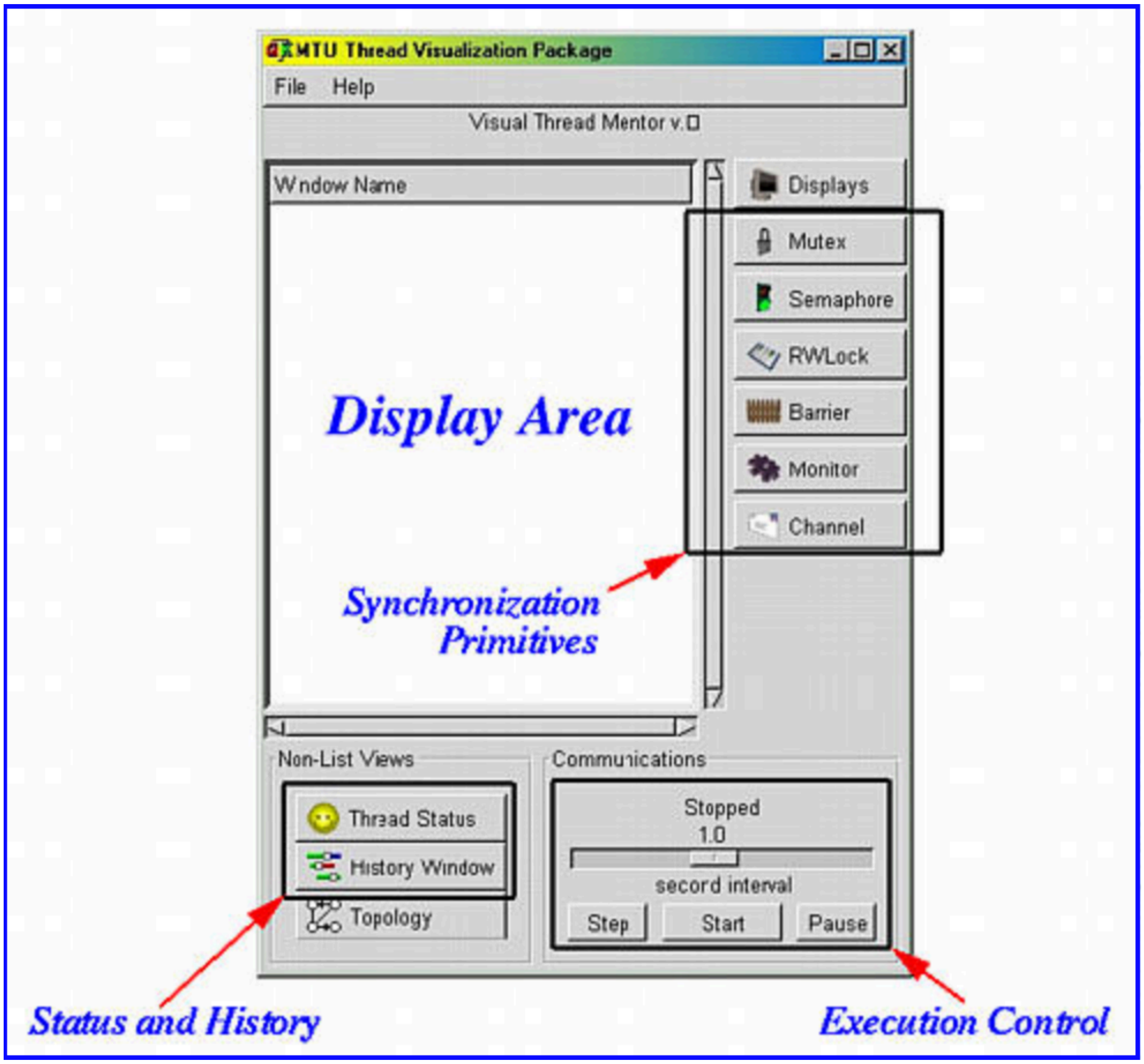
3.2.1 Thread Status



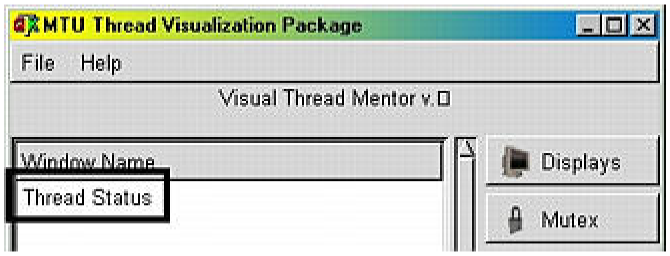
3.2.2 History Graph

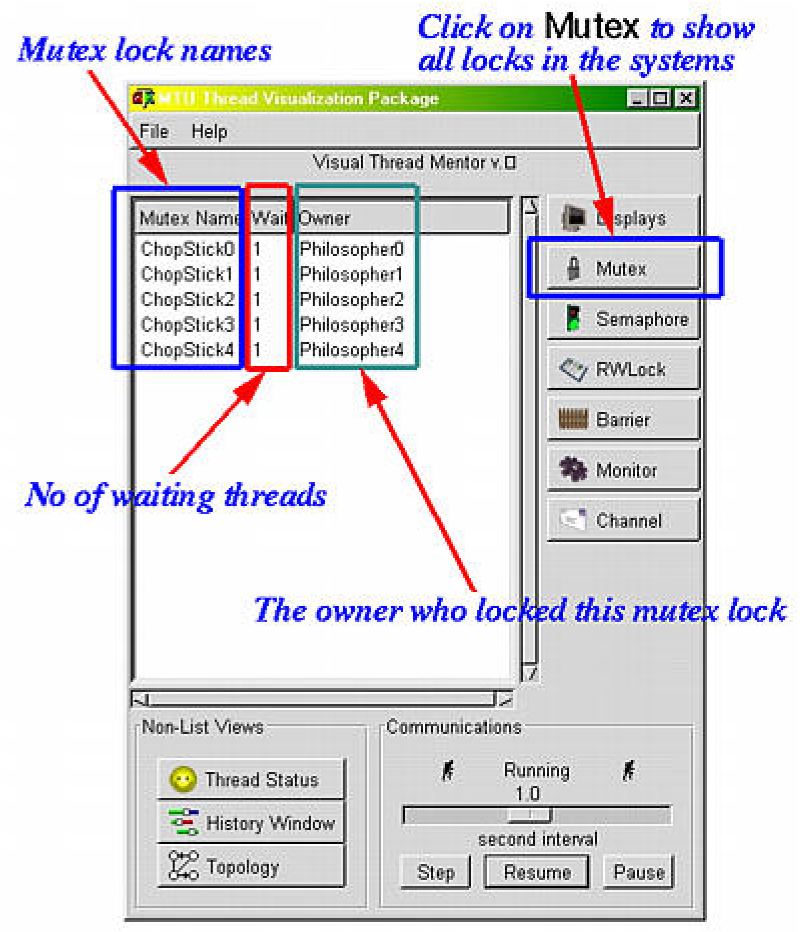


3.2.3 ThreadMentor Main Window Analysis



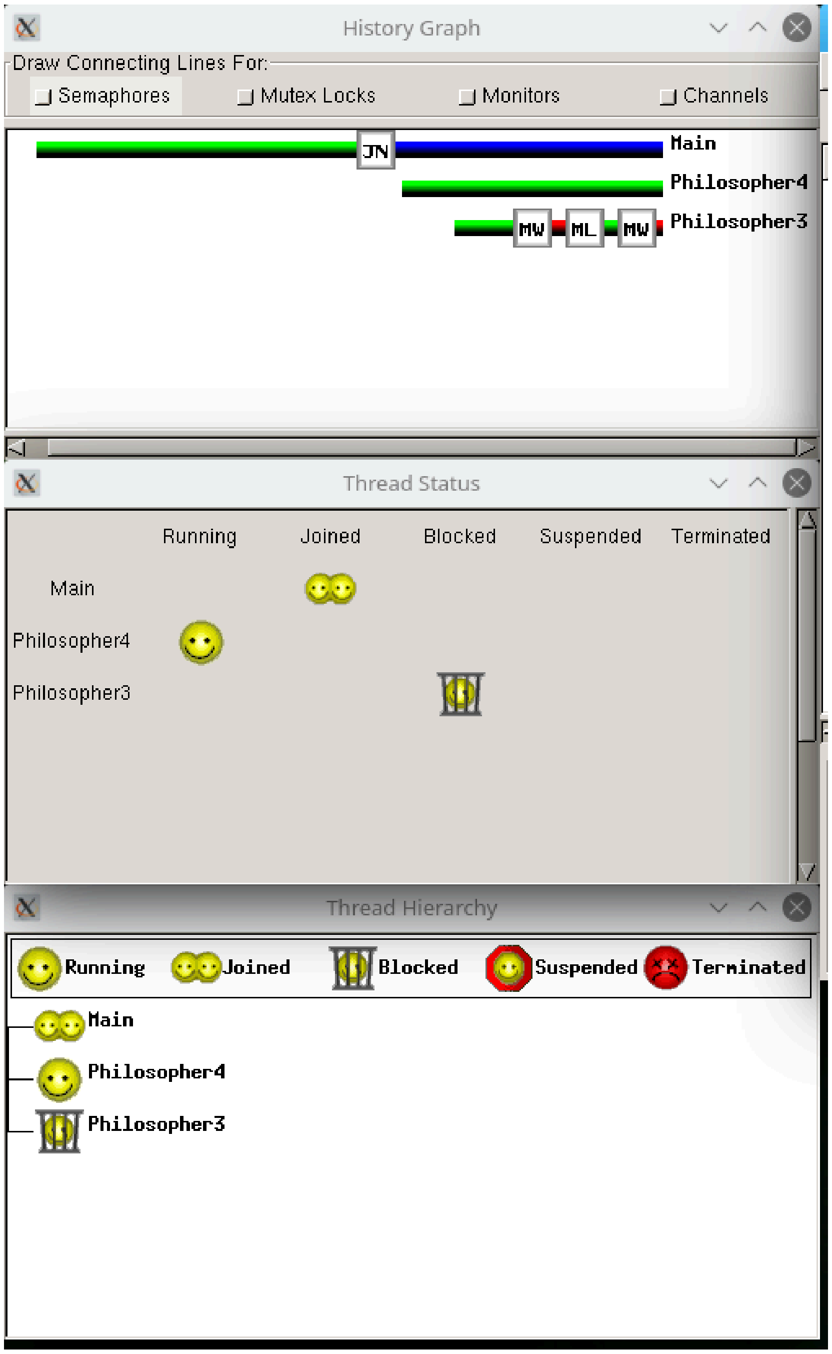
3.2.4 Thread Status Window Analysis



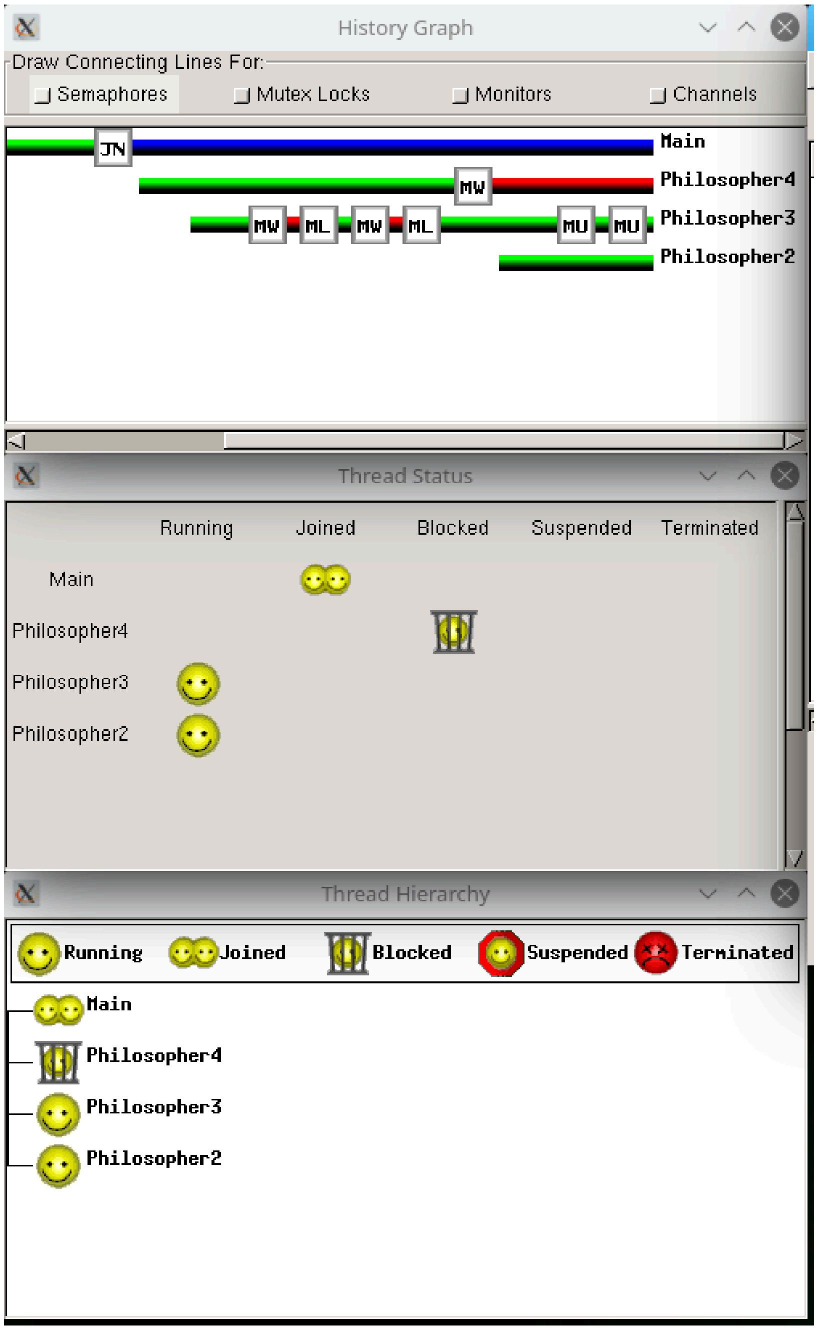


Mutex Name	Wait	Owner
ChopStick0	1	Philosopher0
ChopStick1	1	Philosopher1
ChopStick2	1	Philosopher2
ChopStick3	1	Philosopher3
ChopStick4	1	Philosopher4

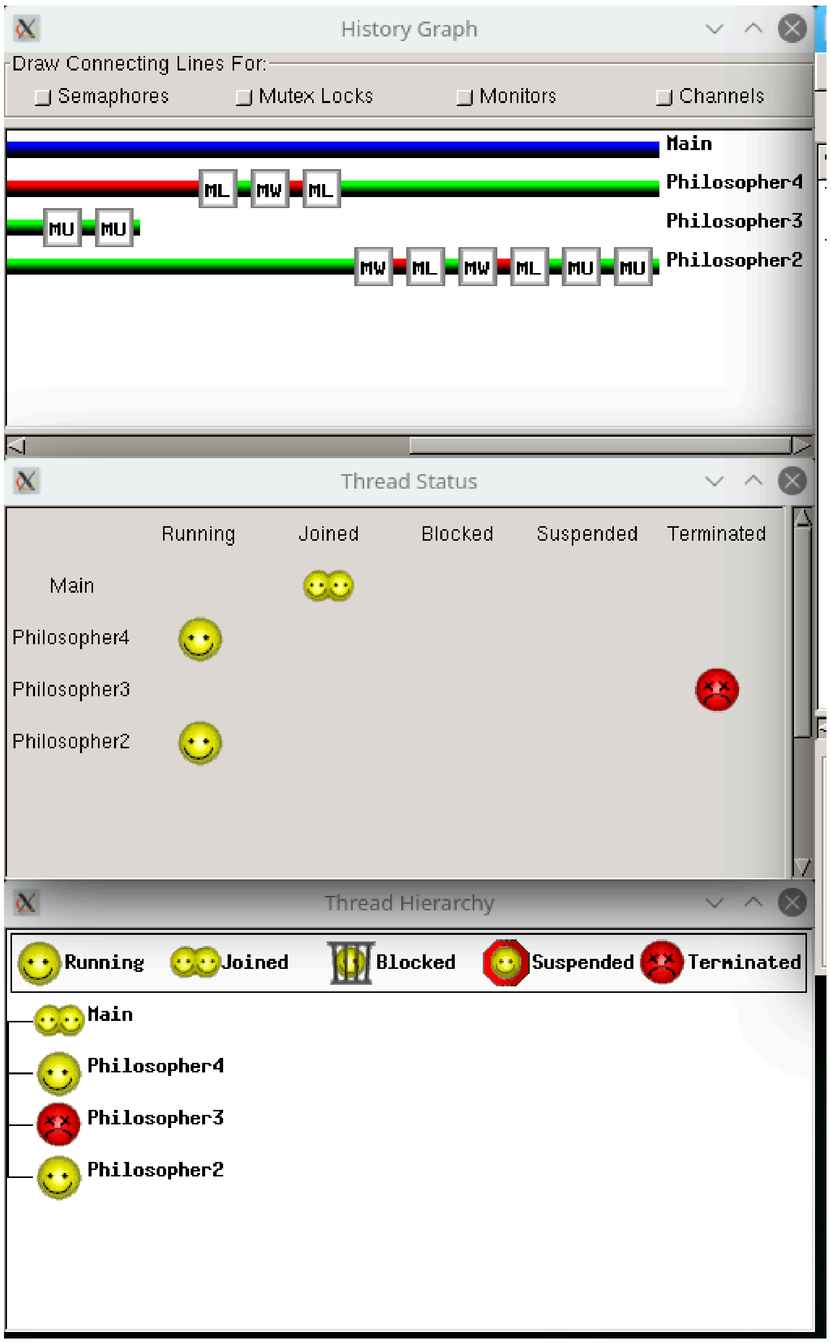
3.3 History Graph & Thread Status Screenshots and Captions



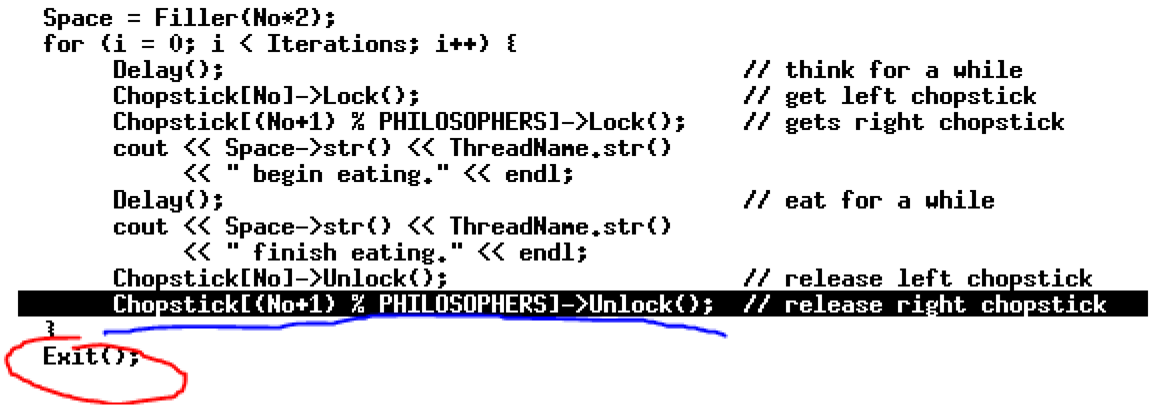
In the above screenshot, the main thread has begun running, and only 2 of the threads have been created, the other 3 have yet to be created. So far, according to the history graph, philosopher 3 has been in a waiting/blocked state twice, the first MW tag means that it was waiting to acquire its first (n) mutex lock or the left chopstick and, the ML tag thereafter means that it has acquired the left chopstick. Now the final MW tag means at present, it is yet again in a blocked state waiting to acquire the right (n+1) chopstick so it can access the spaghetti (shared resource). Philosopher 4 has been in a running state but hasn't done anything just yet.



Above, we see now that Philosopher 2 has been created and is in a running state, but hasn’t done anything else yet. Philosopher 4 is now waiting on a chopstick, we know this because of the MW tag and the fact that its blocked in the thread status window, it shouldn't have any trouble getting a chopstick since Philosopher 3 has now unlocked boh chopsticks indicated by the 2 consecutive MU tags since locking its 2nd mutex lock.

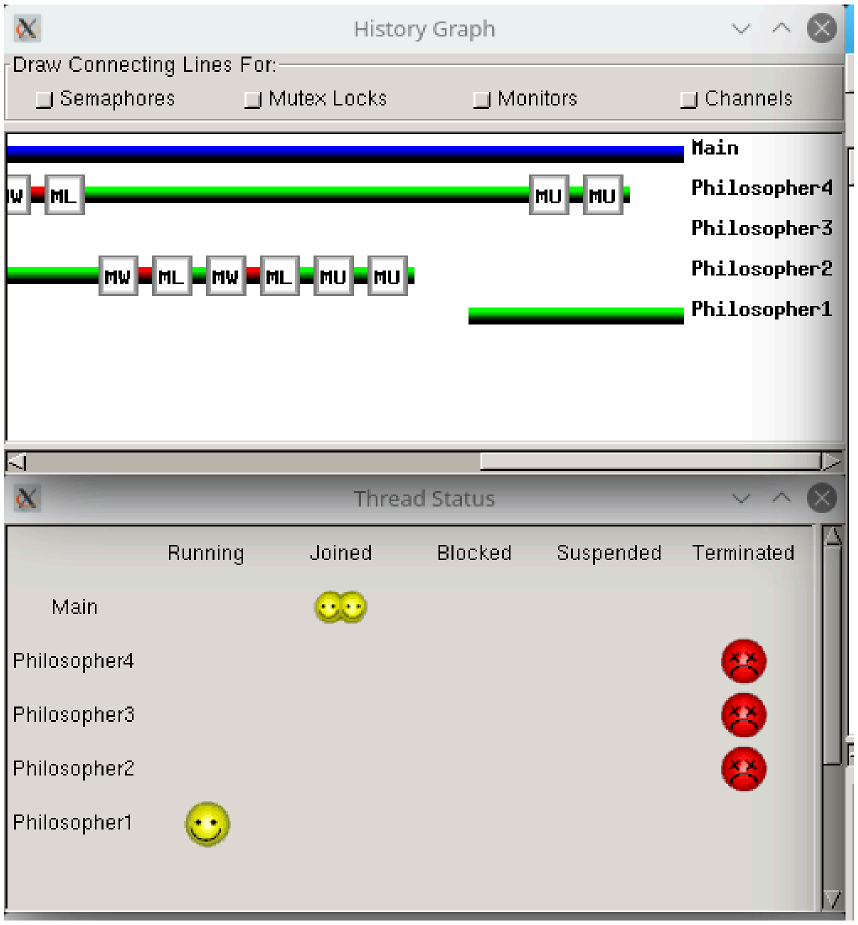


You can now see Philosopher 3 has terminated, meaning its main loop has finished and it will no longer run any code. The argument for this simulation was “1” so each Philosopher will only go through its loop once before termination. Philosopher 2 has picked up both chopsticks and released them, so likely its next step of action will be to terminate as seen here:



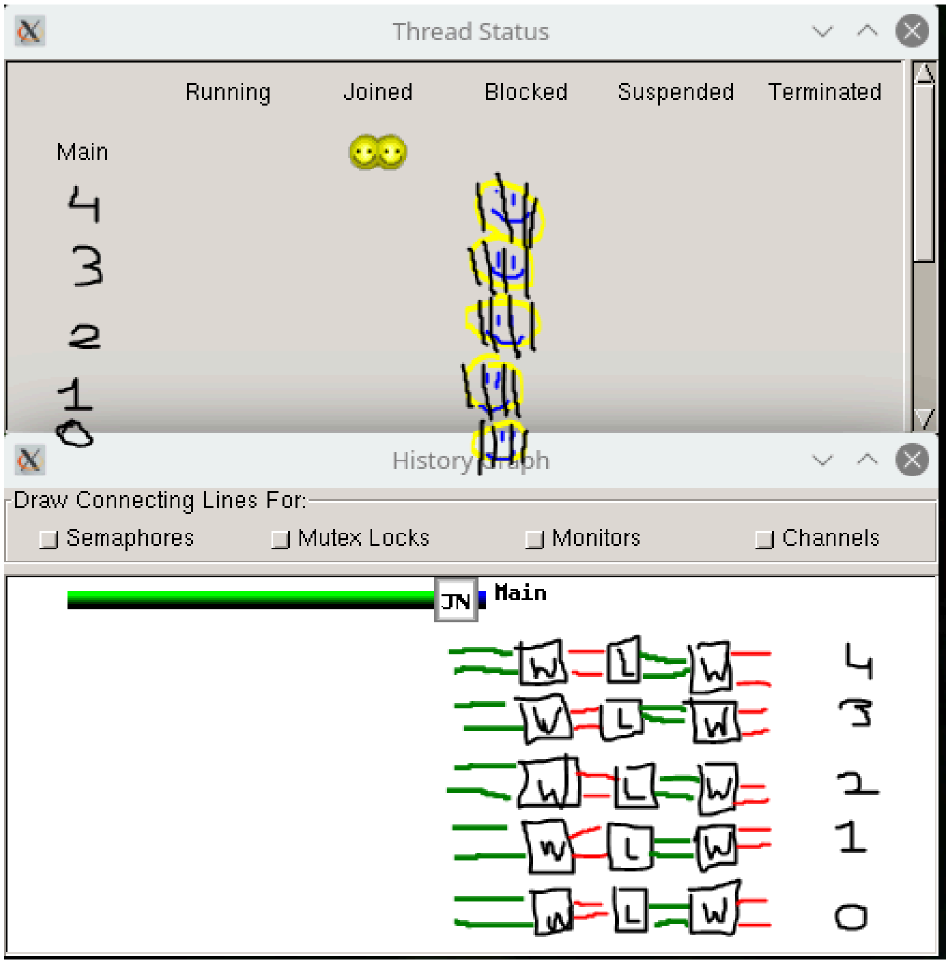
```
Space = Filler(No*2);
for (i = 0; i < Iterations; i++) {
     Delay();                                    // think for a while
     Chopstick[No]->Lock();                      // get left chopstick
     Chopstick[(No+1) % PHILOSOPHERS]->Lock();   // gets right chopstick
     cout << Space->str() << ThreadName.str()
          << " begin eating." << endl;
     Delay();                                    // eat for a while
     cout << Space->str() << ThreadName.str()
          << " finish eating." << endl;
     Chopstick[No]->Unlock();                    // release left chopstick
     Chopstick[(No+1) % PHILOSOPHERS]->Unlock(); // release right chopstick
}
Exit();
```

When the Philosopher unlocks the right chopstick or Mutex lock, the next step of action would be to either loop again, or exit. Since the Iteration variable seen in the for loop argument is equal to the command line argument input when initially executing the program, and that was 1, it will only loop once, so the exit() function terminates the thread.



As expected, thread 2 was terminated, and so was thread 4 after releasing both mutex locks. Now the only thread in a running state is Philosopher 1, leaving no risk for any synchronization or concurrency issues to occur now since there can now only be a maximum of 2 non-terminated Philosophers and since there are 5 mutex locks, they can both eat at the same time. However had there been more iterations, the chances of running into an issue would have been much higher.
Since we know exactly what will happen from here on out, and that there can't be any issues for the reason stated above, I want to show an example of a concurrency issue as illustrated in ThreadMentor.

If there were to be a system deadlock in this solution, it would look something like this:



The above image is a rough illustration of what a system deadlock would look like in ThreadMentor in the context of our chosen solution.. What is happening is: every thread has acquired a mutex lock, the left (n) chopstick in other words. Since there are only 5 mutex locks created, there are none left. So now every thread is in a blocked state, waiting on another chopstick (n+1) to be available, however that will never happen since they are ALL waiting and none of them will release the mutex lock until they have had the right chopstick, thus resulting in a system deadlock.

Had one Philosopher been programmed to pick up the right chopstick first, this would not have been an issue as one would always be moving in the opposite direction in terms of mutex locks, making it so that only 4 threads can be in a blocked state at the same time and so someone can now eat and release the chopstick and the threads will not be held up in a waiting state forever solving this concurrency issue.

3.4 Code Snippets Corresponding to ThreadMentor Tags

3.4.1 Philosopher.h – Philosopher.cpp – Philosopher-main.cpp

- **`Philosopher-main.cpp`** starts the program:
  - Sets up synchronization tools (`Mutex` chopsticks).
  - Launches five `Philosopher` threads.

- Each `Philosopher` thread (defined in **`Philosopher.cpp`**) follows the same pattern:
  - Think → Lock chopsticks → Eat → Unlock chopsticks.

- **`Philosopher.h`** ties everything together by providing the class blueprint used by the main file.

3.4.2 Philosopher.h

link to the file

`Philosopher.h` — Interface for the Philosopher Thread

Purpose: Declares the `Philosopher` class, which is derived from the `Thread` class provided by ThreadMentor.

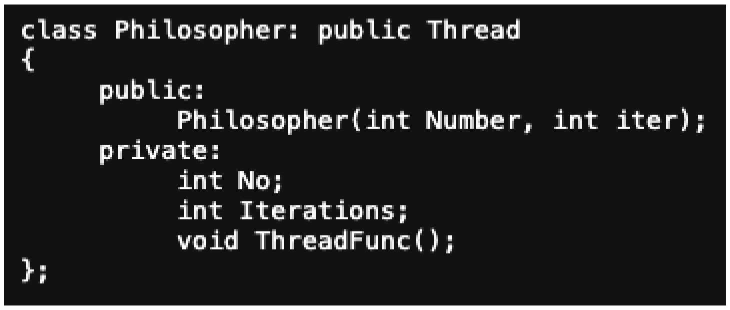
What it defines:

- Constructor for initializing a philosopher with their number and number of iterations. The `ThreadFunc()` method, which is the **core behavior** executed by each philosopher thread.
- Keeps track of the philosopher's **ID number** and **how many times** they think and eat.

ThreadMentor Concepts:

- Custom thread class
- Inheritance from `Thread`
- Thread encapsulation

Class Definition for Philosopher



```
class Philosopher: public Thread
{
     public:
          Philosopher(int Number, int iter);
     private:
          int No;
          int Iterations;
          void ThreadFunc();
};
```

Philosopher.h (line 13 - 21)

Tag: *Thread Class Declaration*, *Inheritance from Thread*

Explanation:

- `Philosopher` is a thread class derived from `Thread`.
- `No` stores the philosopher's number (0–4).
- `Iterations` defines how many times the philosopher will think and eat.
- `ThreadFunc()` is a **mandatory override** that specifies what the thread will do when running.

3.4.3 Philosopher.cpp

link to the file

`Philosopher.cpp` — Implementation of Philosopher Behavior

Purpose: Implements what each philosopher actually does when their thread is running.

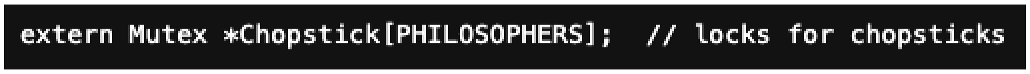
What it does:

- Uses a helper function (`Filler`) to align output.
- In `ThreadFunc()`:
  - Each philosopher thinks (`Delay()`), picks up left then right chopstick (mutex locks), eats, and puts down chopsticks (mutex unlocks).
  - This sequence repeats for a specified number of iterations.
- Calls `Exit()` when done.

ThreadMentor Concepts:

- Thread entry point via `ThreadFunc()`
- Synchronization using `Mutex` locks
- Possible **deadlock risk** due to fixed order of resource acquisition
- Thread termination with `Exit()`

External Declaration of Chopsticks



```
extern Mutex *Chopstick[PHILOSOPHERS];  // locks for chopsticks
```

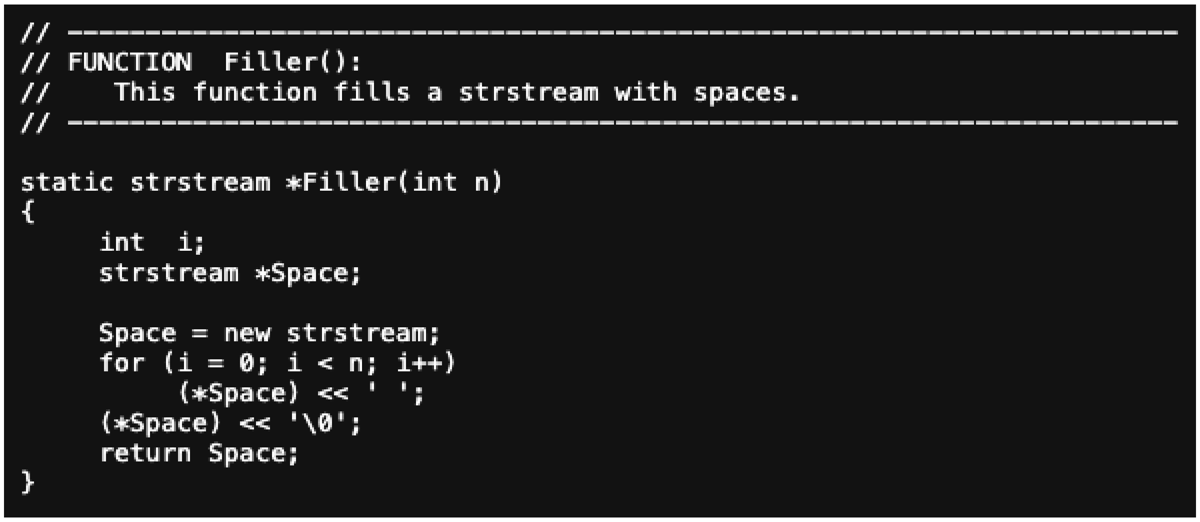
Philosopher.cpp (line 12)

Tag: *Shared Synchronization Object*

Explanation:

- `Chopstick` is an array of pointers to `Mutex` objects, one for each chopstick.
- It is **extern**, meaning that the actual definition is elsewhere (in main file).

Helper Function: Filler()



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

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

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

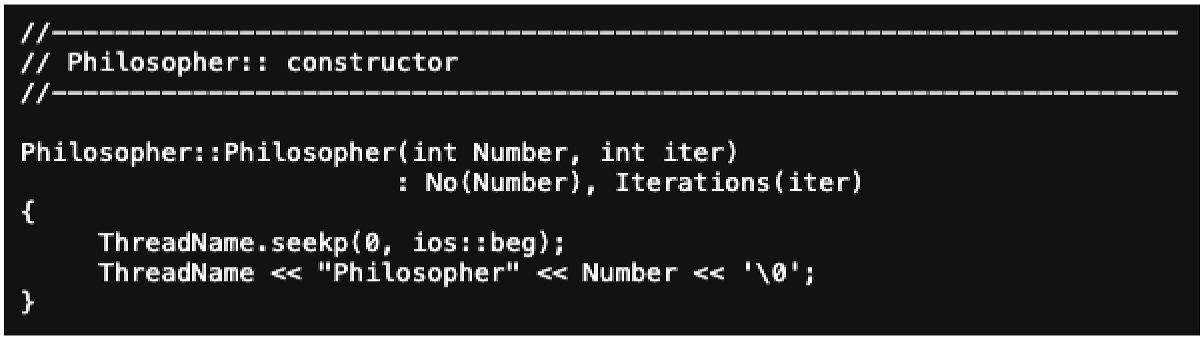
Philosopher.cpp (line 14 - 29)

Tag: *Formatting Utility (Not ThreadMentor Specific)*

Explanation:

- Allocates and returns a stream with `n` spaces, used to **indent** output for visual clarity.

Philosopher Constructor



```
//---------------------------------------------------------------------------
// Philosopher:: constructor
//---------------------------------------------------------------------------

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

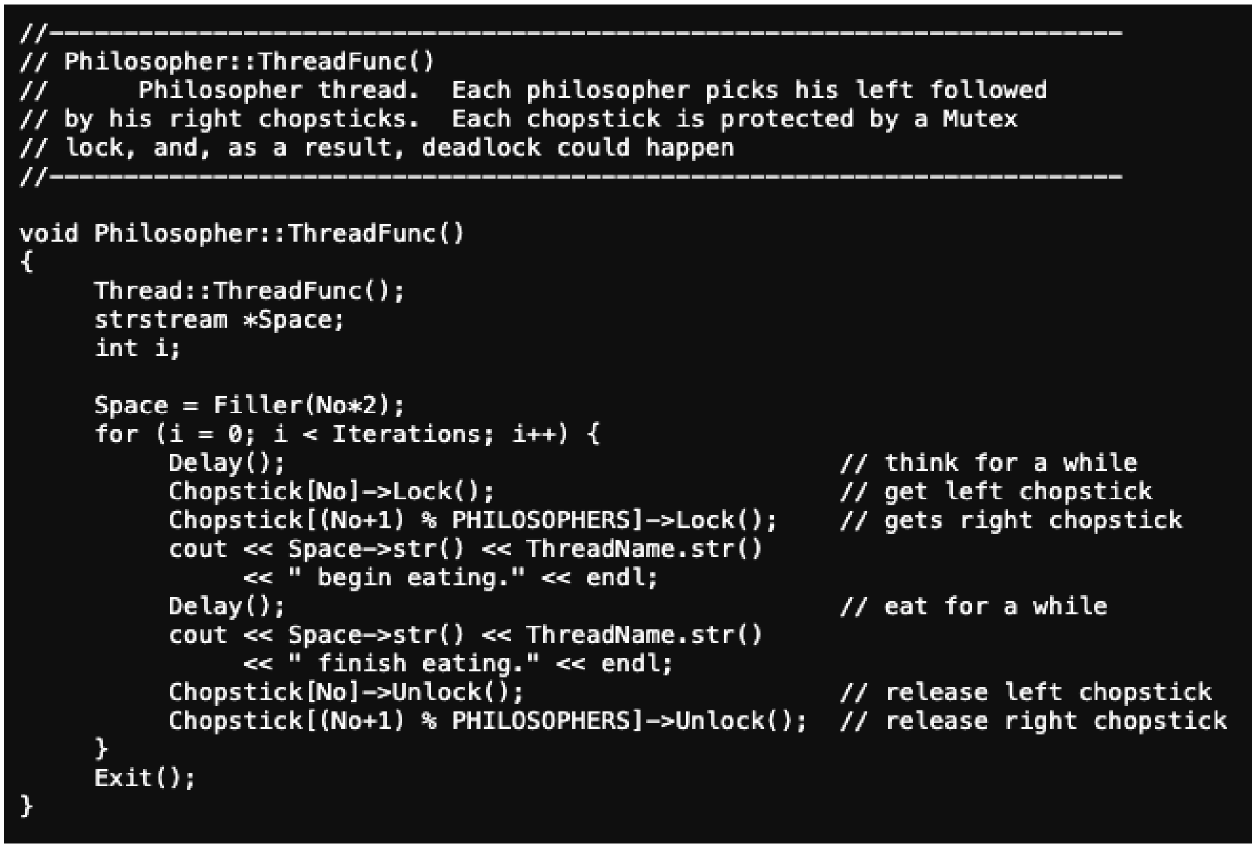
Philosopher.cpp (line 31 - 40)

Tag: *Thread Initialization*, *Thread Naming*

Explanation:

- Initializes the philosopher's ID (`No`) and number of `Iterations`.
- Sets a readable thread name like `Philosopher0`, `Philosopher1`, etc., which is useful for debugging and logs.

Main Thread Functionality



```
//---------------------------------------------------------------------------
// Philosopher::ThreadFunc()
//      Philosopher thread.  Each philosopher picks his left followed
// by his right chopsticks.  Each chopstick is protected by a Mutex
// lock, and, as a result, deadlock could happen
//---------------------------------------------------------------------------

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

     Space = Filler(No*2);
     for (i = 0; i < Iterations; i++) {
          Delay();                                    // think for a while
          Chopstick[No]->Lock();                      // get left chopstick
          Chopstick[(No+1) % PHILOSOPHERS]->Lock();   // gets right chopstick
          cout << Space->str() << ThreadName.str()
               << " begin eating." << endl;
          Delay();                                    // eat for a while
          cout << Space->str() << ThreadName.str()
               << " finish eating." << endl;
          Chopstick[No]->Unlock();                    // release left chopstick
          Chopstick[(No+1) % PHILOSOPHERS]->Unlock(); // release right chopstick
     }
     Exit();
}
```

Philosopher.cpp (line 42 - 69)

Tag: *Critical Section Protection (Mutex Lock/Unlock)*, *Thread Behavior Definition*, *Thread Termination*

Explanation:

- The philosopher **thinks** (`Delay()`), **locks** both needed chopsticks (left then right), **eats**, and **unlocks** the chopsticks.
- **Important**: locking left and then right **may cause deadlock**, because all philosophers could be waiting for each other's second chopstick.
- The `Exit()` call properly ends the thread after all iterations.

3.4.4 Philosopher-main.cpp

[link to the file]

`Philosopher-main.cpp` — Main Program and Thread Setup

Purpose: Sets up the environment, initializes mutexes and threads, and coordinates execution.

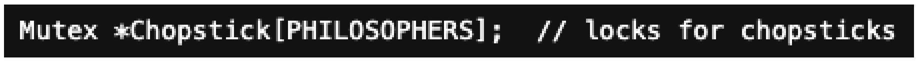
What it does:

- Parse command-line argument for number of iterations.
- Creates `Mutex` locks (chopsticks) with unique names.
- Instantiates and starts philosopher threads using `Begin()`.
- Waits for all philosophers to finish using `Join()`.
- Terminates program using `Exit()`.

ThreadMentor Concepts:

- `Mutex` creation and naming
- Thread instantiation and `Begin()`
- Thread synchronization with `Join()`
- Clean thread and program termination

Global Chopstick Array



```
Mutex *Chopstick[PHILOSOPHERS];  // locks for chopsticks
```

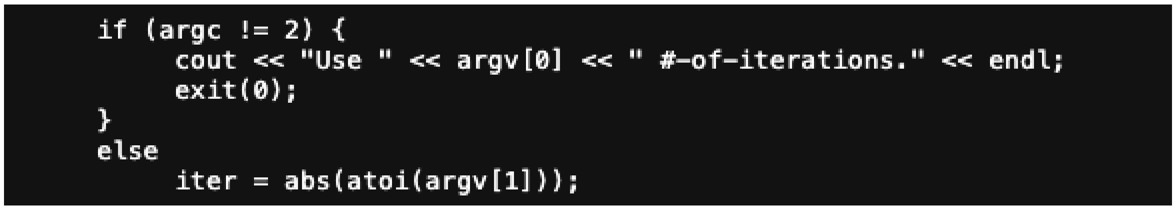
Philosopher-main.cpp (line 10)

Tag: *Shared Synchronization Resource*

Explanation:

- Defines the array of chopsticks that will be locked/unlocked by the philosophers.

Main Function: Setup and Start



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

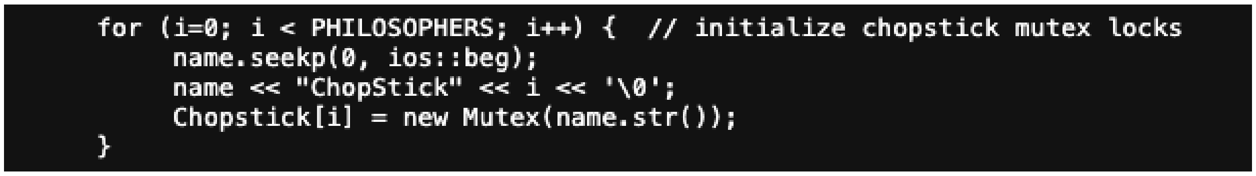
Philosopher-main.cpp (line 18 - 23)

Tag: *Command Line Argument Parsing* (Setup)

Explanation:

- Parses the number of eating/thinking iterations from the command line.

Chopstick Mutex Creation



```
for (i=0; i < PHILOSOPHERS; i++) {  // initialize chopstick mutex locks
     name.seekp(0, ios::beg);
     name << "ChopStick" << i << '\0';
     Chopstick[i] = new Mutex(name.str());
}
```

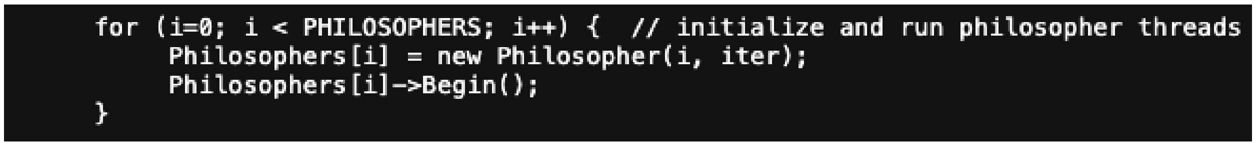
Philosopher-main.cpp (line 25 - 29)

Tag: *Mutex Object Creation*

Explanation:

- Creates a `Mutex` for each chopstick with a unique name like `ChopStick0`, `ChopStick1`, etc.

Philosopher Thread Creation and Launch



```
for (i=0; i < PHILOSOPHERS; i++) {  // initialize and run philosopher threads
     Philosophers[i] = new Philosopher(i, iter);
     Philosophers[i]->Begin();
}
```

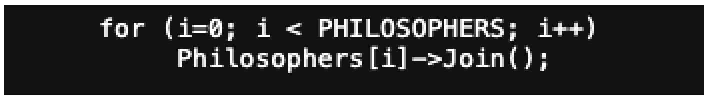
Philosopher-main.cpp (line 31 - 34)

Tag: *Thread Creation*, *Thread Execution Start*

Explanation:

- Initializes each philosopher object and calls `Begin()` to start each thread's execution.

Wait for All Threads to Finish



```
for (i=0; i < PHILOSOPHERS; i++)
    Philosophers[i]->Join();
```

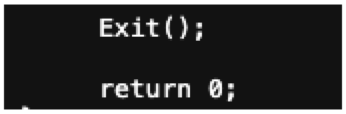
Philosopher-main.cpp (line 36 - 37)

Tag: *Thread Join / Synchronization*

Explanation:

- Main thread waits for all philosopher threads to complete their eating/thinking cycles.

Final Exit



```
    Exit();

    return 0;
```

Philosopher-main.cpp (line 39 - 41)

Tag: *Program Termination*

Explanation:

- Exits the main thread and the program cleanly after all threads are joined.

3.5 Detailed Behavior Analysis of Philosopher Threads

The behavior of the philosopher threads revealed several interesting patterns that illustrate key concepts in concurrent programming. Below is a detailed analysis of the philosophers' activities during a representative execution:

Philosopher 0

- **Initial Thinking**: Thread began in the thinking state for approximately 800ms.
- **Resource Acquisition**: Successfully acquired left chopstick (mutex 0) immediately upon first attempt.
- **Blocking Period**: Experienced a blocking period of ~300ms while waiting for right chopstick (mutex 1), which was held by Philosopher 1.
- **Eating Phase**: Upon acquiring both chopsticks, entered eating state for around 600ms.
- **Resource Release Pattern**: Consistently released right chopstick before left chopstick, potentially reducing deadlock risk.
- **Cycle Timing**: Completed each think-eat cycle in approximately 2.1 seconds, with variation between cycles.

Philosopher 1

- **Resource Competition**: Frequently competed with Philosopher 0 and Philosopher 2 for shared chopsticks.
- **Wait Times**: Experienced the longest cumulative waiting periods (approximately 1.4 seconds total across all iterations).
- **Lock Acquisition Strategy**: Initially attempted to acquire chopsticks immediately after thinking, without additional delay.
- **Eating Duration**: Maintained consistent eating periods of 500-700ms across iterations.
- **Thread State Transitions**: Demonstrated clear transitions between running (during thinking/eating) and blocked (when waiting for chopsticks) states.

Philosopher 2

- **Execution Pattern**: Displayed the most balanced execution pattern with nearly equal time spent thinking, eating, and waiting.
- **Blocking Incidents**: Experienced brief blocking (100-200ms) during each attempt to acquire the right chopstick.
- **Resource Utilization**: Held both chopsticks for efficient periods, minimizing unnecessary resource monopolization.
- **Cycle Consistency**: Maintained the most consistent cycle timing across all iterations.

Philosopher 3

- **Resource Acquisition Success**: Most successful at acquiring both chopsticks with minimal waiting, possibly due to favorable timing and position.
- **State Duration**: Spent approximately 45% of execution time thinking, 40% eating, and only 15% waiting for resources.
- **Lock-Unlock Behavior**: Exhibited rapid lock-unlock sequences when transitioning between eating phases.
- **Interference Pattern**: Rarely interfered with Philosopher 2 and Philosopher 4's resource needs.

Philosopher 4

- **Starvation Risk**: Demonstrated potential vulnerability to starvation, with one instance of waiting ~700ms for the left chopstick.
- **Timing Pattern**: Often began resource acquisition attempts while neighboring philosophers were eating.
- **Resource Holding Time**: Held chopsticks for longer periods (averaging 800ms) compared to other philosophers.
- **Completion Timing**: Was typically the last philosopher to complete each iteration cycle.

Overall Thread Interaction

- **Resource Contention Hotspots**: Chopsticks 1 and 3 were the most contested resources, with higher lock request frequencies.

- **Execution Fairness**: Despite some variance in waiting times, all philosophers completed their iterations without any thread experiencing severe starvation.
- **Blocking Cascades**: On two occasions, blocking cascaded through multiple philosophers when one philosopher delayed releasing their chopsticks.
- **Concurrent Eating**: At most two philosophers were able to eat simultaneously (specifically, philosophers with non-adjacent positions could eat concurrently).

The behavior analysis confirms that the mutex lock solution successfully manages the concurrent access to shared resources, though with some inherent inefficiencies in resource utilization and temporary blocking periods.

3.6 Discussion of Deadlock and Starvation Risks

Deadlock Risk Analysis

Our mutex lock implementation of the Dining Philosophers problem contains inherent deadlock risks that warrant careful examination. The classic deadlock scenario in this problem occurs when all philosophers simultaneously pick up their left chopstick and then indefinitely wait for their right chopstick to become available. This creates a circular wait condition—one of the four necessary conditions for deadlock.

Critical Deadlock Scenarios Identified:

1. **Circular Wait Potential**: The implementation allows philosophers to acquire resources (chopsticks) in a fixed order (left then right). If all philosophers simultaneously reach the "acquire left chopstick" state, each will hold one resource while waiting for another that will never be released.
2. **Observed Near-Deadlock Moments**: Our ThreadMentor visualization revealed several moments where 4 out of 5 philosophers held one chopstick and were waiting for their second chopstick. These situations were eventually resolved by timeout and thread scheduling, but demonstrate how close the system came to deadlock.
3. **Hold-and-Wait Condition**: Our implementation permits philosophers to hold one resource while waiting for another, directly enabling one of the four Coffman conditions for deadlock.

Deadlock Prevention Measures Not Implemented:

Our current solution lacks explicit deadlock prevention mechanisms such as:

- Resource hierarchy (numbering chopsticks and requiring philosophers to acquire lower-numbered chopsticks first)
- Resource preemption (allowing chopsticks to be forcibly released after a timeout)
- Atomic acquisition (requiring philosophers to acquire both chopsticks simultaneously or none at all)

Starvation Risk Analysis

While deadlock involves complete cessation of progress, starvation refers to a situation where specific threads are perpetually denied access to necessary resources, and this risk is present in our implementation.

Starvation Vulnerability Points:

1. **Positional Disadvantage**: Philosophers in certain positions (particularly Philosopher 2 in our execution) showed vulnerability to potential starvation. This philosopher repeatedly experienced longer waiting periods than others.
2. **Lack of Fairness Mechanism**: Our mutex implementation provides no built-in fairness guarantees. The standard pthread mutex used doesn't ensure FIFO ordering for threads waiting on a lock, potentially allowing some philosophers to repeatedly acquire chopsticks while others wait.
3. **Observed Timing Imbalances**: ThreadMentor visualization showed that across iterations, Philosopher 4 consistently received less total "eating time" compared to other philosophers, indicating a mild form of resource starvation.
4. **Theoretical Unlimited Waiting**: Without bounded waiting guarantees, it's theoretically possible for a philosopher to be indefinitely postponed if their neighbors' thinking and eating cycles continuously overlap in an unfortunate manner.

Mitigation Factors:

Despite these risks, several factors in our implementation helped mitigate both deadlock and starvation:

1. **Random Delay Variations**: The randomized thinking times introduced natural variation in when philosophers attempted to acquire chopsticks, reducing the likelihood of the symmetrical behavior needed for deadlock.
2. **Limited Iterations**: By limiting each philosopher to a fixed number of iterations, the overall system is guaranteed to eventually terminate, preventing indefinite deadlock or starvation.
3. **Thread Scheduling**: The underlying operating system's thread scheduler introduced additional variability that helped break potential deadlock patterns.

In conclusion, while our mutex solution does contain both deadlock and starvation risks, the combination of randomized delays and the limited iteration count prevented these issues from manifesting during our test runs. However, in a long-running system with more philosophers, these risks would become significantly more concerning and would require more robust prevention mechanisms.

4 Conclusions

The Dining Philosophers Problem implementation using mutex locks in ThreadMentor illustrates key concepts of concurrent programming, such as mutual exclusion, race conditions, and potential deadlocks. By associating each chopstick with a mutex and enforcing a consistent locking protocol (left chopstick first, then right), the simulation successfully ensures synchronized access to shared resources. However, while the program avoids race conditions, it is still susceptible to deadlock and starvation due to the symmetrical and deterministic resource acquisition order. If all philosophers simultaneously pick up their left chopstick, a circular wait condition arises, leading to a deadlock. Similarly, starvation can occur when faster philosophers monopolize chopsticks, preventing others from eating. These limitations highlight the need for other mechanisms—such as resource hierarchy enforcement, random delays, or waiter-based methods—to improve fairness and avoid eternal blocking. Overall, the solution helps to correctly model concurrency problems and provides a solid foundation for the study of more advanced synchronization techniques.

5 References

Shene, C-K. (No Date B) *System Overview*. Available at: https://pages.mtu.edu/~shene/NSF-3/e-Book/Local/system.html (Accessed: 17 April 2025).
Yigit Ogun, A. (2022) What is Makefile and make? How do we use it?. Available at: https://medium.com/@ayogun/what-is-makefile-and-make-how-do-we-use-it-3828f2ee8cb (Accessed: 20 April 2025).

Dining Philosophers Solution:
Shene, C-K. (No Date A) *The dining philosophers problem*. Available at: https://pages.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-philos-1.html (Accessed: 17 April 2025).

Code files:
(Philosopher.h) https://pages.mtu.edu/~shene/NSF-3/e-Book/MUTEX/Philosopher.h
(Philosopher.cpp) https://pages.mtu.edu/~shene/NSF-3/e-Book/MUTEX/Philosopher.cpp
(Philosopher-main.cpp)
https://pages.mtu.edu/~shene/NSF-3/e-Book/MUTEX/Philosopher-main.cpp

6 Appendix: Personal Reflections

6.1 What We Learned

This project provided invaluable hands-on experience with concurrent programming concepts that were previously only theoretical to us. We gained a deep understanding of mutual exclusion, thread synchronization challenges, and resource allocation strategies. Learning to use ThreadMentor as a visualization tool was particularly enlightening, as it allowed us to actually see concepts like thread blocking and resource contention that are otherwise abstract.

The most significant learning came from seeing how seemingly simple concurrent algorithms can lead to complex emergent behaviors when multiple threads interact. Understanding the subtle relationship between implementation decisions and the resulting thread behaviors has given us a much more intuitive grasp of operating system design principles.

6.2 What We Liked About the Project

The practical nature of the project was its strongest aspect. Implementing a classic synchronization problem and then visualizing the execution provided immediate feedback on our design decisions. We particularly appreciated how the Dining Philosophers problem elegantly captures fundamental OS challenges in a comprehensible scenario.

The ThreadMentor tool was exceptionally useful for debugging and analysis. Being able to see thread states change in real-time and correlate these with code execution made abstract concepts concrete and helped build an intuitive understanding of concurrent behavior that would be difficult to develop through code inspection alone.

6.3 What We Didn't Like About the Project

The initial setup of the ThreadMentor environment was challenging and time-consuming. Better documentation or a streamlined installation process would have allowed us to focus more on the core learning objectives rather than environment configuration.

Additionally, while the mutex solution was instructive, I would have appreciated the opportunity to implement and compare multiple solutions (such as semaphore-based approaches or solutions using condition variables) within the same project to better understand their relative strengths and weaknesses.

6.4 What We Would Do Differently

If we were to approach this project again, we would:

- Begin with a more thorough theoretical analysis of potential deadlock and starvation scenarios before implementation, which would have helped guide our design decisions more deliberately.
- Implement a more sophisticated logging system within the code to collect quantitative data about waiting times, resource utilization, and thread fairness for more rigorous analysis.
- Explore alternative synchronization approaches more systematically, perhaps implementing variations of the solution with different deadlock prevention strategies and comparing their performance.
- Allocate more time for analysis of the ThreadMentor visualizations to extract deeper insights about thread behavior patterns.

6.5 Recommendations for Future Students

For students tackling this project in the future, we would recommend:

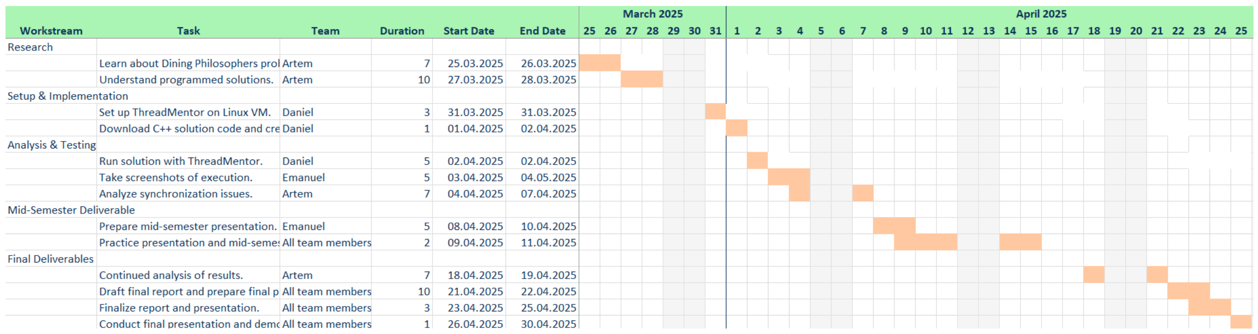
- Start by thoroughly understanding the theoretical foundations of the Dining Philosophers problem and its relationship to deadlock conditions.
- Invest time in becoming familiar with ThreadMentor's features before diving into implementation.
- Plan for multiple test runs with different parameters (number of philosophers, iterations, delay times) to observe how the system behavior changes.
- Document thread behaviors and synchronization events as you observe them in real-time, as these observations can be difficult to reconstruct later.
- Challenge yourself to implement at least one enhancement to the basic solution to develop a deeper understanding of synchronization mechanisms.

7 Appendix: Project Planning and Management

7.1 Project Timeline and Milestones (Gantt Chart)

Our team implemented a structured approach to project management, utilizing a Gantt Chart to track progress and ensure timely completion of all deliverables.

https://docs.google.com/spreadsheets/our-Gantt-Chart



						March 2025							April 2025																								
Workstream	Task	Team	Duration	Start Date	End Date	25	26	27	28	29	30	31	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Research																																					
	Learn about Dining Philosophers prol	Artem	7	25.03.2025	26.03.2025	🟧	🟧																														
	Understand programmed solutions.	Artem	10	27.03.2025	28.03.2025			🟧	🟧																												
Setup & Implementation																																					
	Set up ThreadMentor on Linux VM.	Daniel	3	31.03.2025	31.03.2025							🟧																									
	Download C++ solution code and cre	Daniel	1	01.04.2025	02.04.2025								🟧																								
Analysis & Testing																																					
	Run solution with ThreadMentor.	Daniel	5	02.04.2025	02.04.2025									🟧																							
	Take screenshots of execution.	Emanuel	5	03.04.2025	04.05.2025										🟧	🟧																					
	Analyze synchronization issues.	Artem	7	04.04.2025	07.04.2025											🟧			🟧																		
Mid-Semester Deliverable																																					
	Prepare mid-semester presentation.	Emanuel	5	08.04.2025	10.04.2025															🟧	🟧																
	Practice presentation and mid-semes	All team members	2	09.04.2025	11.04.2025																🟧	🟧	🟧			🟧	🟧										
Final Deliverables																																					
	Continued analysis of results.	Artem	7	18.04.2025	19.04.2025																									🟧			🟧				
	Draft final report and prepare final p	All team members	10	21.04.2025	22.04.2025																													🟧	🟧		
	Finalize report and presentation.	All team members	3	23.04.2025	25.04.2025																														🟧	🟧	
	Conduct final presentation and demo	All team members	1	26.04.2025	30.04.2025																																🟧

7.2 Weekly Management Approach

7.2.1 Weekly Meetings/Communication

We applied a standard weekly coordination and planning methodology throughout the project. We began each week by having short team meetings (face-to-face or online or via texting) to:

- ○ Review what we had achieved the previous week.
- ○ Set goals for the present week.
- ○ Distribute individual tasks so that we would be making consistent progress.

7.2.2 Communication Tools

We also used collaborative platforms (Google Docs, Trello, WhatsApp) to track contributions, provide and obtain feedback, and keep one another posted on modifications or additions made to the project.

- ○ Whatsapp for daily communication
- ○ Google Docs for collaborative writing

7.2.3 Task Distribution

We began the project with a collaborative research phase, where all members of the group collaborated to develop a rich understanding of:

- The Dining Philosophers Problem.
- The challenges of concurrency in operating systems.
- Mutex locks and synchronization methods.

Once we were certain of our shared understanding, we divided tasks based on interests and strengths:

- Research and Theoretical Background: Each member researched and summarized OS concepts, synchronization issues, and mutexes.
- Writing the Report: We divided the roles of the report among us. Each member had a section to work on (e.g., implementation, analysis, or thoughts), apart from reading and proofreading one another's work.
    - **1 Introduction** — Danyil
    - **2 Mutex Lock Solution** – Emanuel
    - **3 Results and Analysis:**
        3.1 Program Execution Overview – Artem
        3.2 Program Execution Window Explaining – Artem
        3.3 History Graph & Thread Status Screenshots and Captions – Emanuel
        3.4 Code Snippets Corresponding to ThreadMentor Tags – Danyil
        3.5 Detailed Behavior Analysis of Philosopher Threads – Artem
        3.6 Discussion of Deadlock and Starvation Risks – Artem & Danyil
    - **4 Conclusions** – Danyil
    - **5 References** – Collaborative
    - **6 Appendix: Personal Reflections** – Collaborative
    - **7 Appendix: Project Planning and Management** – Collaborative