OS Project

Team Report (Final) Presentation

ThreadMentor:
Dining Philosophers Problem Solution

1 Introduction

1.1 Background

1.2 The Dining Philosophers Problem

1.3 Outline/Layout of our Report

Introduction

Background

1.1 Concurrency and Synchronisation
1.2 Mutex Locks
1.3 Semaphores
1.4 ThreadMentor

Introduction

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.

Introduction

Outline/Layout of our Report

Section 1 is this, the Introduction

Section 2 covers the introduction of the Mutex Lock Solution

Section 3 provides a thorough Results and Analysis

In **Section 4**, we state our Conclusions

2 Mutex Lock Solution

2.1 Theory/How it works

2.2 ThreadMentor

Mutex Lock Solution

We chose the 1st solution on the ThreadMentor e book website.

Why?

Mutex Locks seemed interesting & there are many talking points.

Mutex Lock Solution

How it works

Thread = Philosopher

Mutex Lock = Chopstick

Shared Resource = Food

Thread needs 2 Mutex locks, and that's it.

Mutex Lock Solution

Philosopher.h

```
#include "ThreadClass.h"

#define PHILOSOPHERS     5

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

Mutex Lock Solution

Philosopher.cpp

```
#include <iostream>
#include "Philosopher.h"

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

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

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

     return Space;
}

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

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();
}
```

Mutex Lock Solution

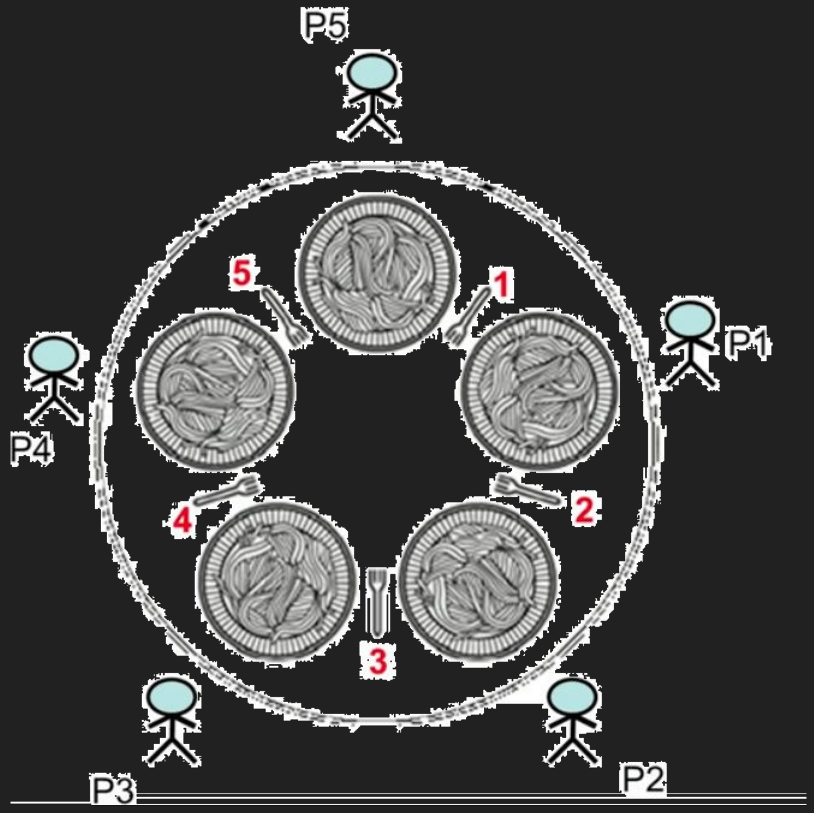
Philosopher-main.cpp

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

#include "Philosopher.h"

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

int main(int argc, char *argv[])
{
     Philosopher *Philosophers[PHILOSOPHERS];
     int i, iter;
     strstream name;

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

     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());
     }

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

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

     Exit();

     return 0;
}
```

3 Results and Analysis

3.1 Program Execution Overview
3.2 Program Execution Window Explaining
3.3 History Graph & Thread Status Screenshots and Captions
3.4 Code Snippets Corresponding to ThreadMentor Tags
3.5 Detailed Behavior Analysis of Philosopher Threads
3.6 Discussion of Deadlock and Starvation Risks

3.1 Program Execution Overview



Key Parameters:

- 5 Philosopher Threads
- 5 Chopstick Mutex Locks
- 3 Thinking-Eating Cycles

Execution Phases

1. Initialization

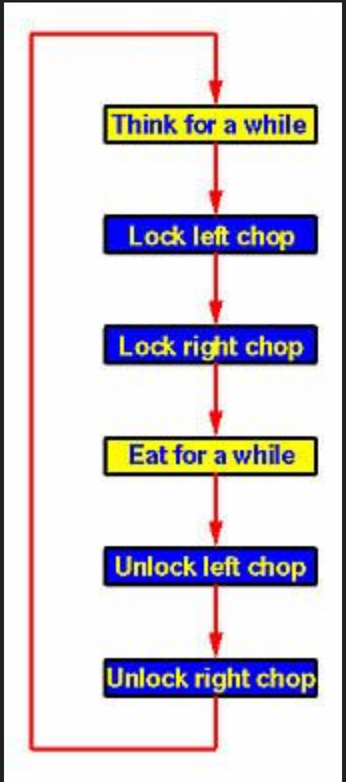
- Create 5 mutex locks (chopsticks)
- Spawn 5 philosopher threads

2. Execution

- Think → Grab Left → Grab Right → Eat → Release Both

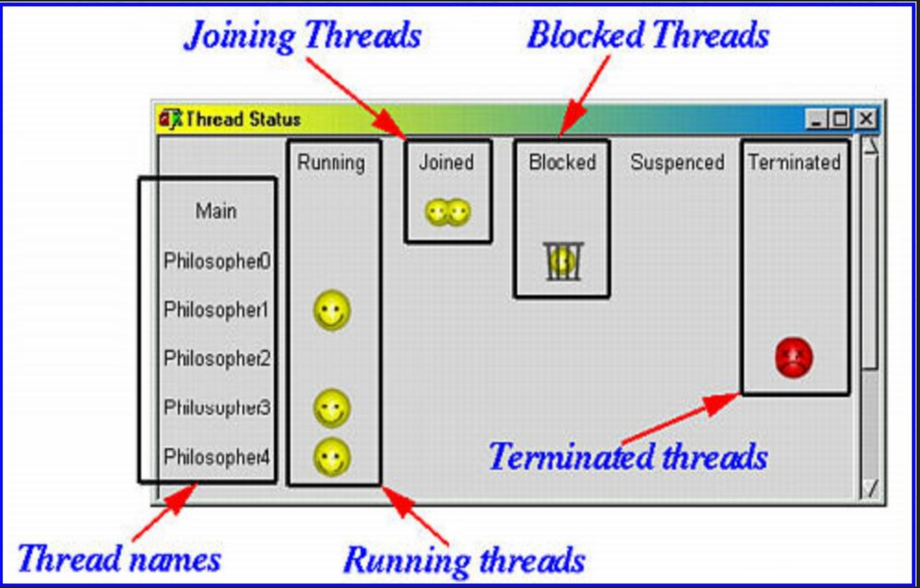
3. Termination

- Threads complete 3 cycles
- Join all threads
- End simulation

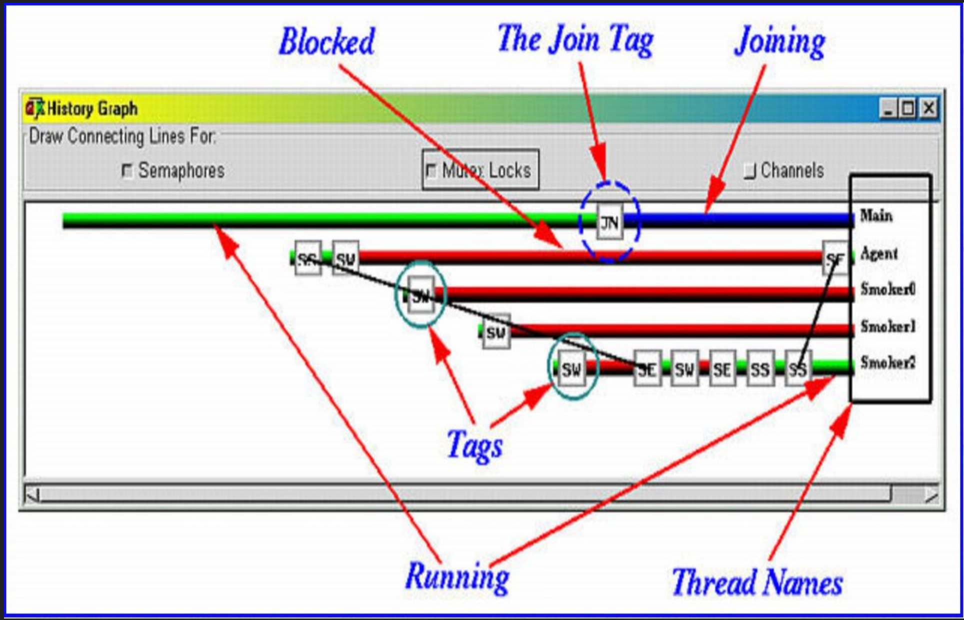


3.2 Program Execution Window Explaining

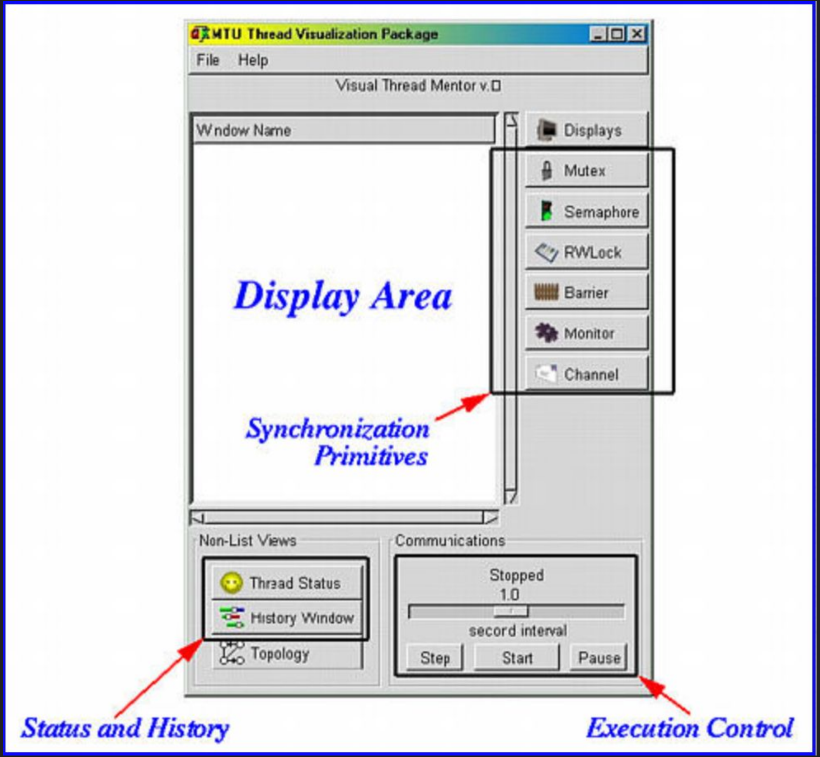
Thread Status Window



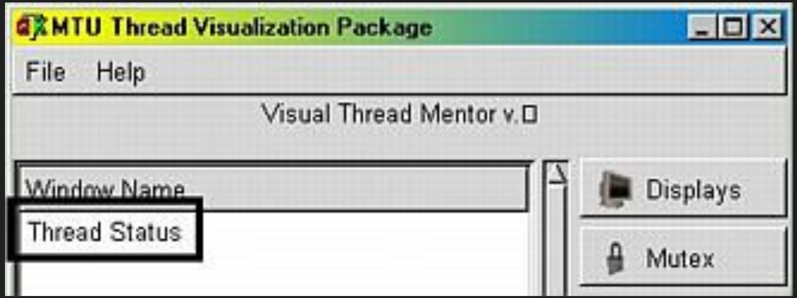
History Graph



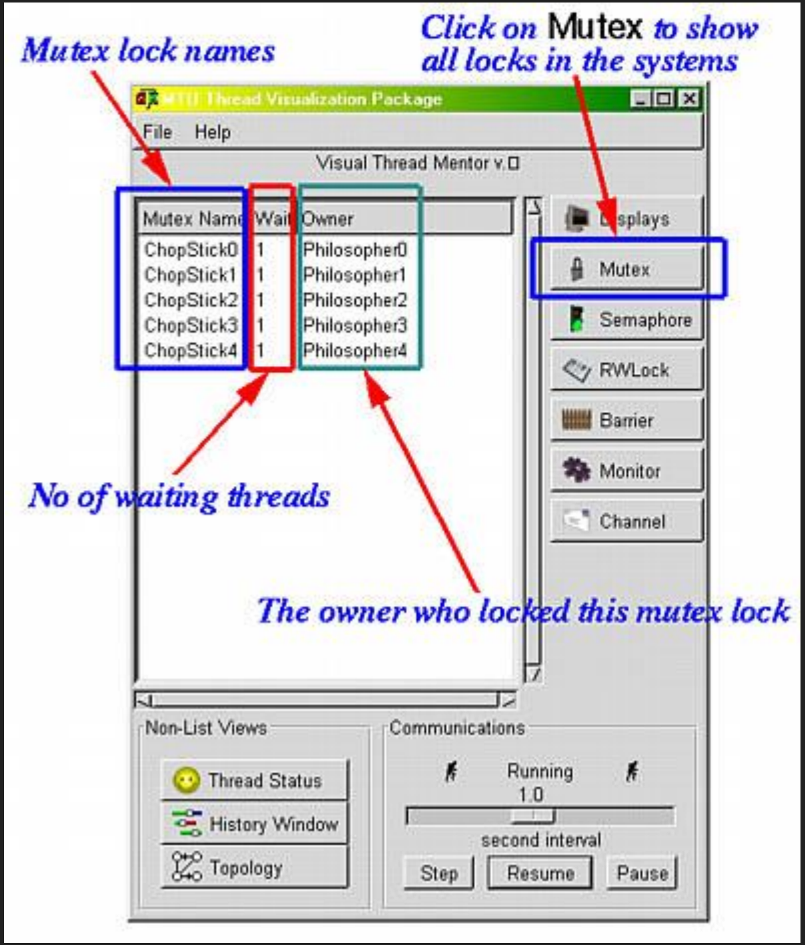
ThreadMentor Main Window



Thread Status Window Details



Mutex Locks Window

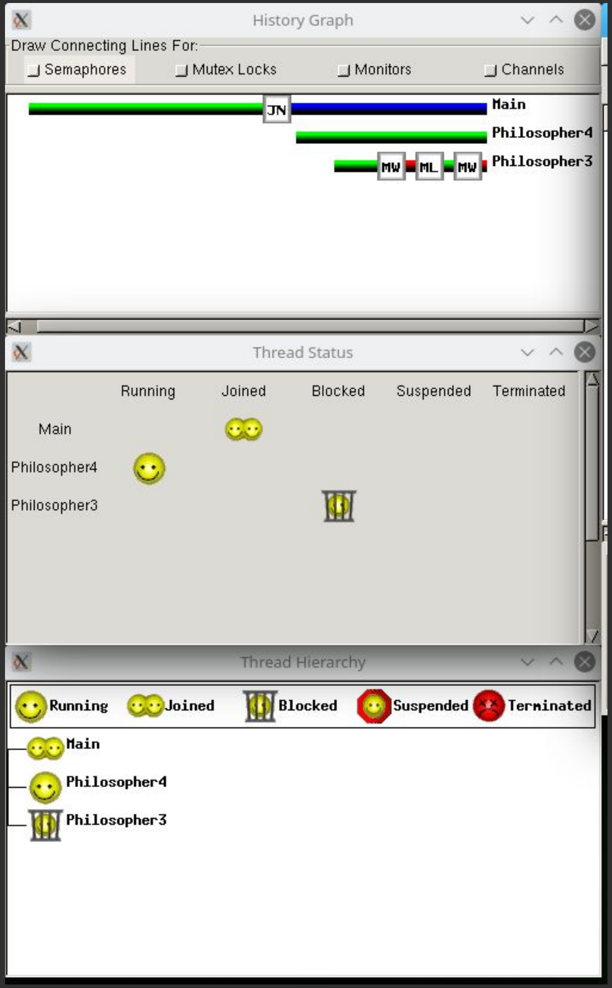


3 Results and Analysis

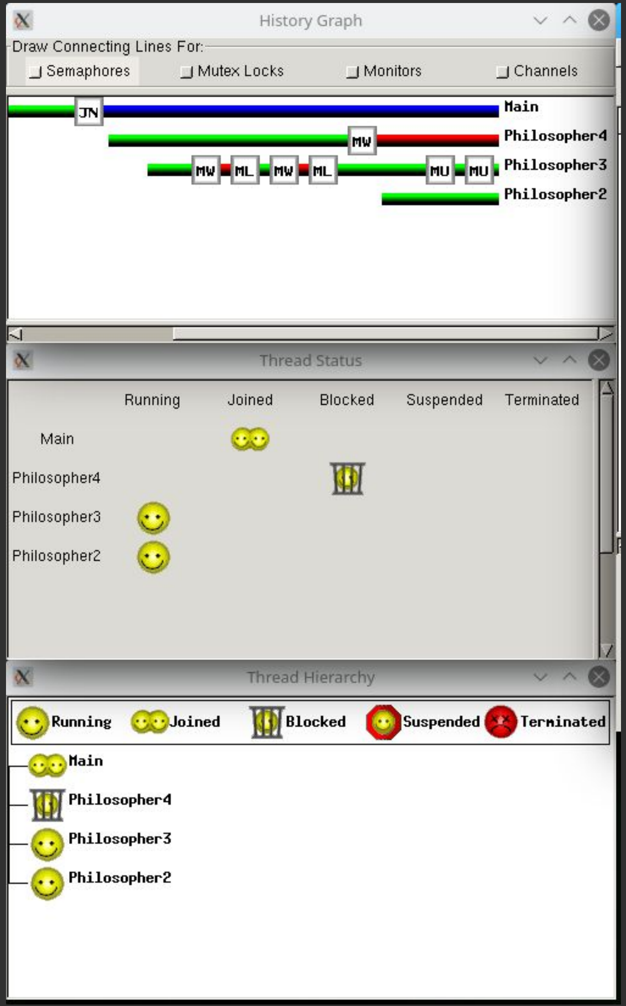
History Graph & Thread Status Screenshots and Captions

History Graph & Thread Status explanation

```
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();
```

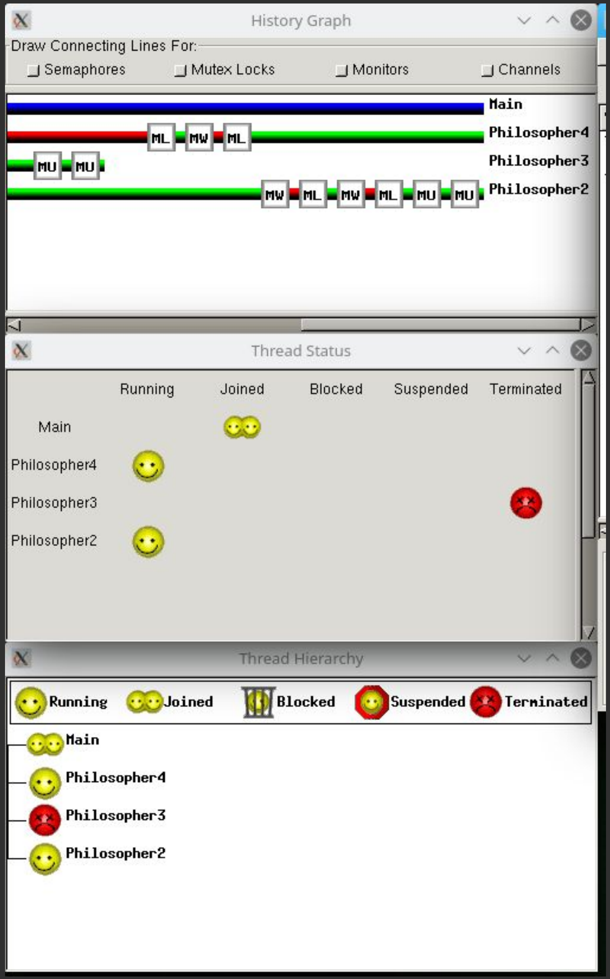


History Graph & Thread Status explanation



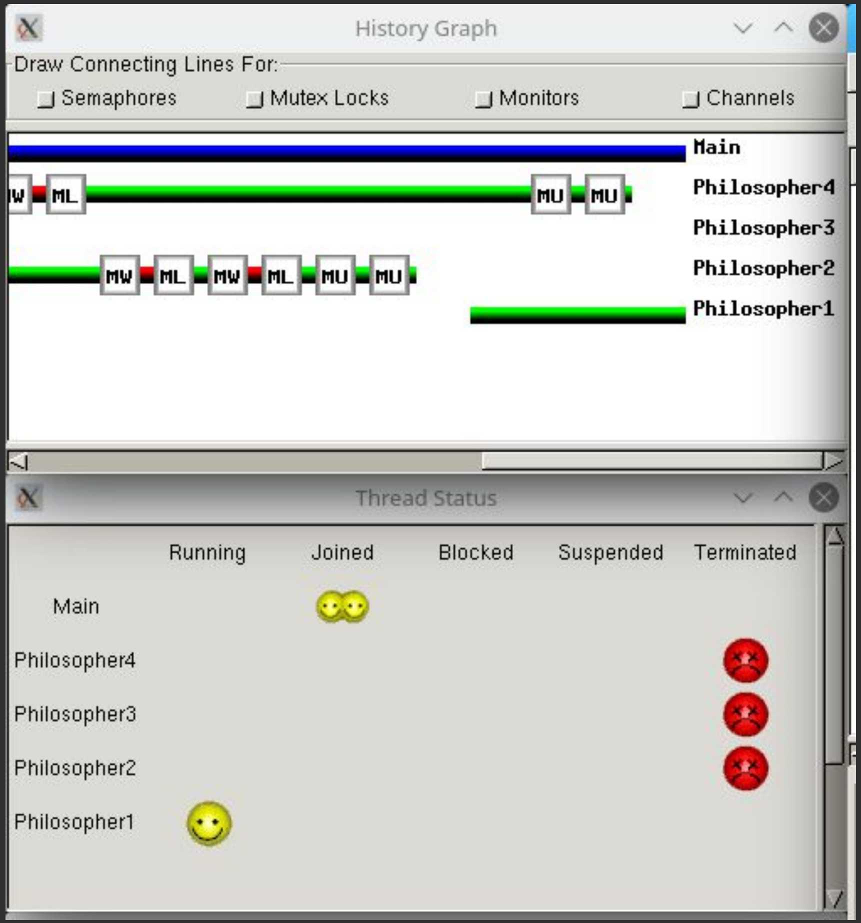
History Graph & Thread Status explanation

```
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();
```

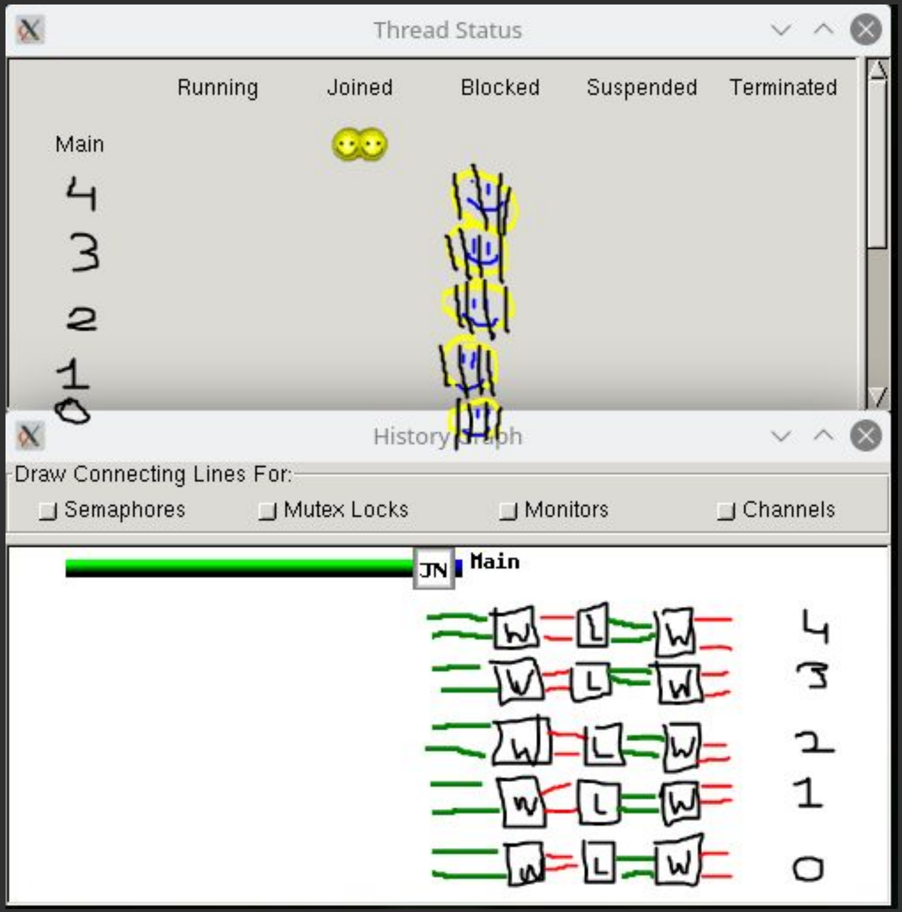


History Graph & Thread Status explanation

```
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();
```



History Graph & Thread Status explanation



3 Results and Analysis

Code Snippets Corresponding to ThreadMentor Tags

Philosopher.h Philosopher.cpp

Philosopher-main.cpp

Philosopher.h

Philosopher.h

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)

Philosopher.cpp

Philosopher.cpp

External Declaration of Chopsticks

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

Philosopher.cpp (line 12)

Philosopher.cpp

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)

Philosopher.cpp

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)*

Philosopher.cpp

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 31 - 40)

Philosopher-main.cpp

Philosopher-main.cpp

Global Chopstick Array

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

Philosopher-main.cpp (line 10)

Philosopher-main.cpp

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)*

Philosopher-main.cpp

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 10)

Philosopher-main.cpp

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 10)

Philosopher-main.cpp

Wait for All Threads to Finish

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

- *Philosopher-main.cpp (line 10)*

Philosopher-main.cpp

Final Exit

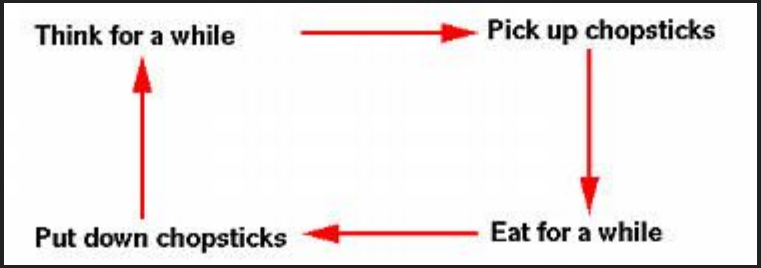
```
Exit();
return 0;
```

Philosopher-main.cpp (line 10)

3.5 Detailed Behavior Analysis of Philosopher Threads

Key Observations:

- Success of mutex lock solution
- Varying resource contention patterns
- No deadlocks occurred
- At most 2 philosophers eating concurrently



Philosopher 0 Behavior

Key Patterns:

- Initial thinking: ~800ms
- Brief blocking: ~300ms waiting for right chopstick
- Eating phase: ~600ms
- Released right chopstick before left

Philosopher 1 Behavior

Key Patterns:

- Highest resource competition
- Longest waiting periods (1.4s total)
- Consistent eating duration: 500-700ms
- Clear running/blocked state transitions

Philosopher 2 Behavior

Key Patterns:

- Most balanced execution pattern
- Brief blocking periods: 100-200ms
- Efficient resource utilization
- Highest cycle consistency

Philosopher 3 Behavior

Key Patterns:

- Minimal waiting times
- Time distribution: 45% thinking, 40% eating, 15% waiting
- Rapid lock-unlock sequences
- Low interference with neighbors

Philosopher 4 Behavior

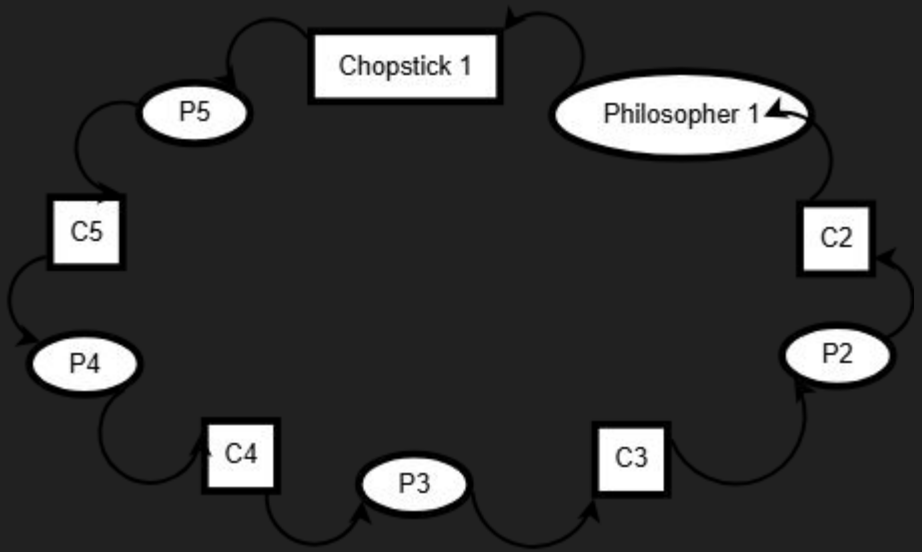
Key Patterns:

- Potential starvation risk
- One instance of ~700ms wait
- Longer resource holding (~800ms avg)
- Typically last to complete cycles

3.6 Deadlock Risk Analysis

Key Deadlock Risks:

- Circular wait potential
- Hold-and-wait condition present
- Near-deadlock moments observed



Starvation Risk Analysis

Starvation Vulnerabilities:

- Positional disadvantages observed
- No fairness guarantees in mutex implementation
- Consistent timing imbalances detected
- Theoretical unlimited waiting possible

Mitigation Factors

Factors Preventing Deadlock & Starvation:

- Random delay variations
- Limited iterations (3 cycles)
- OS thread scheduling variability
- Resource release ordering

4 Conclusions

- The Dining Philosophers Problem helped us understand real-world concurrency issues.
- Mutex locks ensured safe access to shared resources.
- ThreadMentor allowed us to visualize thread behavior and detect issues.
- We gained hands-on experience with synchronization in operating systems.

5 References

- Dining Philosophers Solution
- Code files:
  - Philosopher.h
  - Philosopher.cpp
  - Philosopher-main.cpp
- Other references:
  - System Overview
  - What is Makefile and make? How do we use it?

6 Appendix: Personal Reflections

6.1 What We Learned
6.2 What We Liked About the Project
6.3 What We Didn't Like About the Project
6.4 What We Would Do Differently
6.5 Recommendations for Future Students

7 Appendix: Project Planning and Management

- Gantt Chart
- Task Distribution

Workstream	Task	Team	Duration	Start Date	End Date
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-seme:	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 demc	All team members	1	26.04.2025	30.04.2025