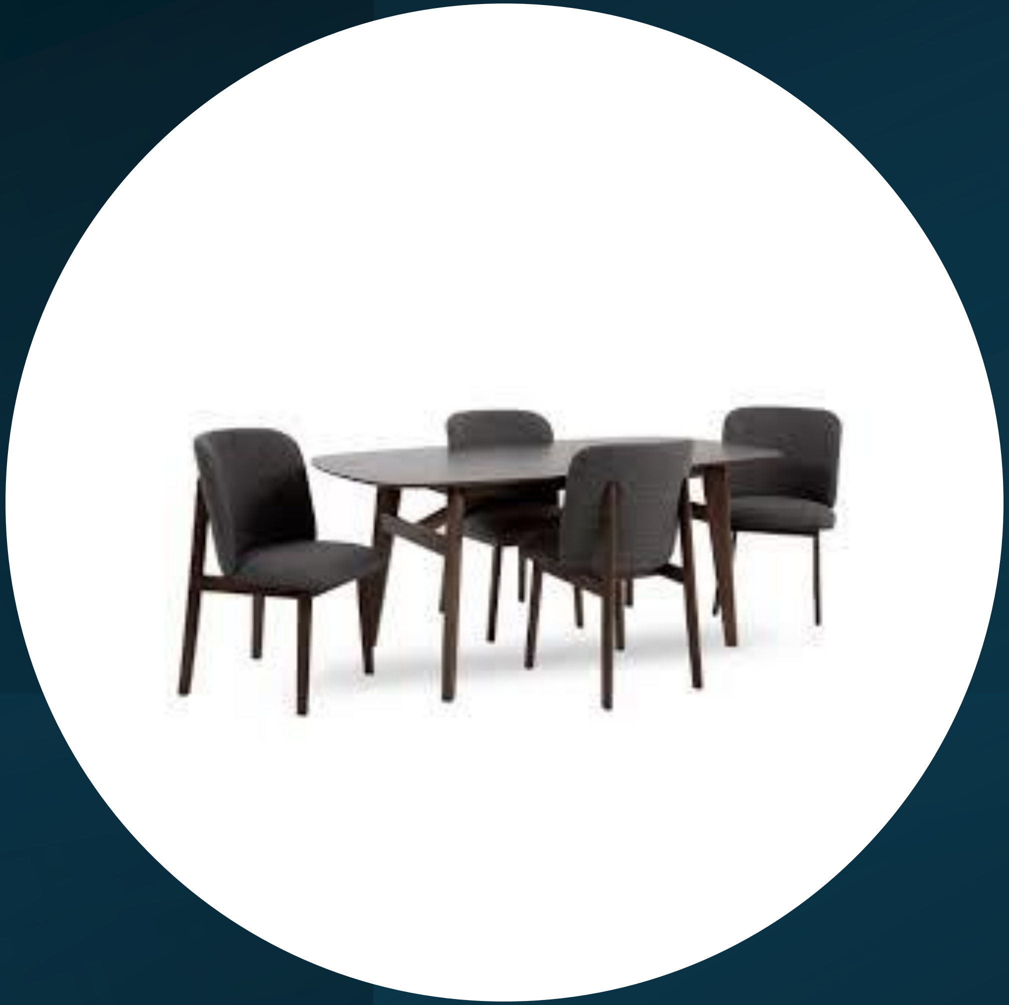
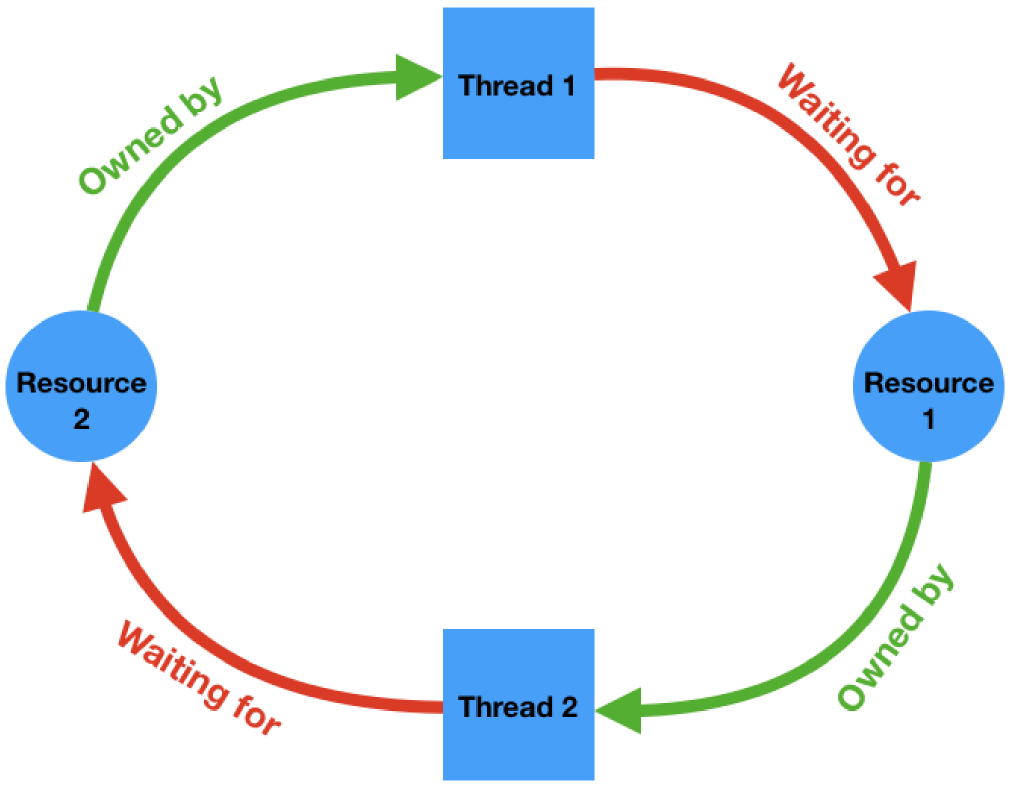
The Dining Philosophers problem

The Four Chairs Solution

By: Hayden Carr, Kenert Salm, Ramazan Iskandarov, Joshua Hand





Introduction

- There are many issues with synchronization for the Dining Philosophers problem, such as Data races, Deadlock and Starvation.
- Mutex Locks and Semaphores are commonly used to achieve synchronization.
- Mutex is good for preventing Data Races, but they can lead to starvation.
- Semaphores can control multiple processes, yet they are susceptible to programming errors.

Our Solution (The Four Chairs Solution)

- For this solution, we take away one chair.
- Each chair is associated with a semaphore – the initial value for the semaphore is set to 4.
- After a philosopher has finished eating, the value of the semaphore will go back up by 1.

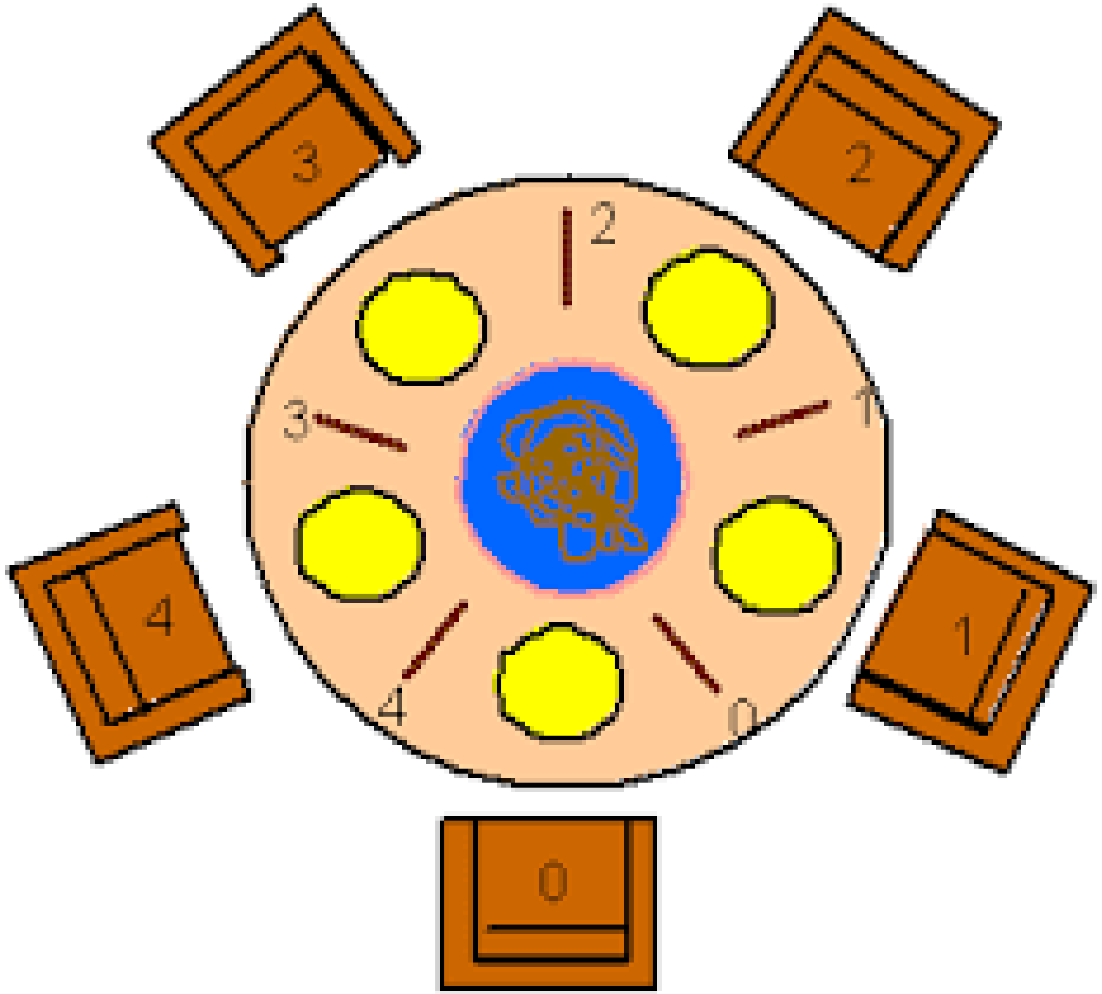
The Four Chairs Solution Overview

- **Key Features:**
  - **Chairs Representation:** Instead of focusing on the chopsticks, this solution focuses on representing the availability of chairs using semaphores.
  - **Semaphore Usage:** Semaphores are used to control access to the chairs, ensuring that philosophers can only sit on available chairs.
  - **Concurrency Management:** The solution ensures that philosophers take turns to access chairs and eat without causing conflicts or deadlocks.
- **Implementation:** Philosophers (threads) wait on semaphores representing the chairs they need. When a chair becomes available, a philosopher sits down, eats, and then releases the semaphore (chair).

Categories of Solutions

The Dining Philosophers problem has been addressed through various solutions, each employing different synchronization techniques. We mainly focus on the use of Semaphores.

- **Semaphores**
  - **Explanation:** Semaphores are synchronization primitives used to control access to shared resources in concurrent programming. They act as counters, allowing a specified number of threads to access a resource concurrently.



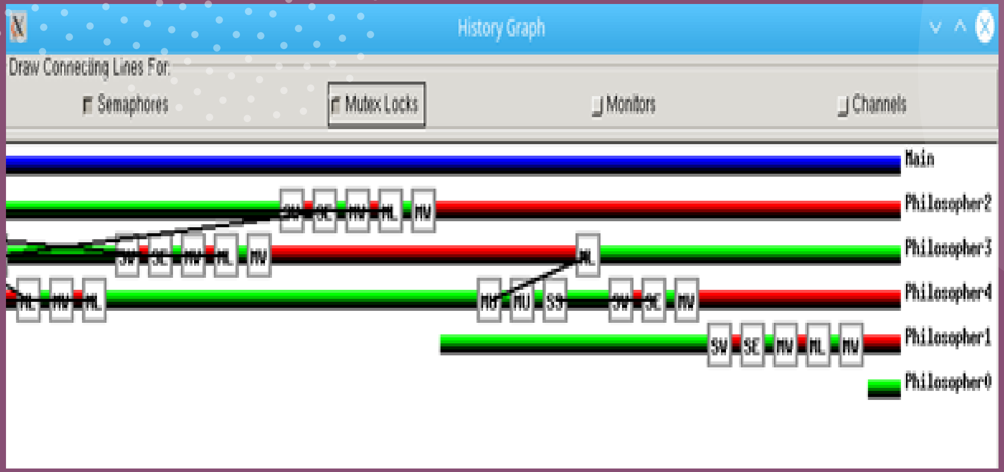
Code Snippet

- This function allows for picking up chopsticks, eating(delay) and putting down chopsticks. The releasing of each chair is controlled by a semaphore.
- fourChairs.Wait(); is responsible for allowing four philosophers to sit down at a table.
- FourChairs.Signal(); is responsible for releasing a chair.
- Delay(); creates a delay so that the philosophers can eat.

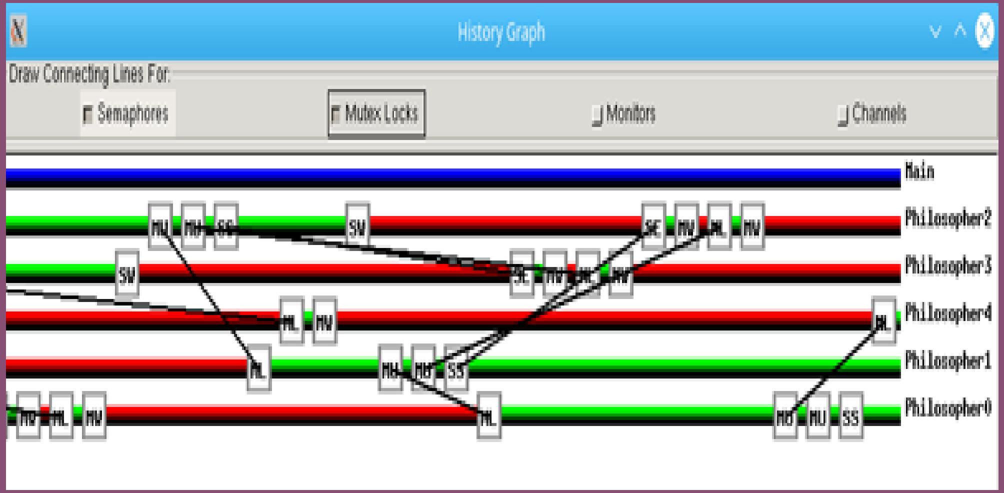
```cpp
void Philosopher::ThreadFunc()
{
     Thread::ThreadFunc();
     strstream *Space;
     int i;

     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
```

ThreadMentor Visualization

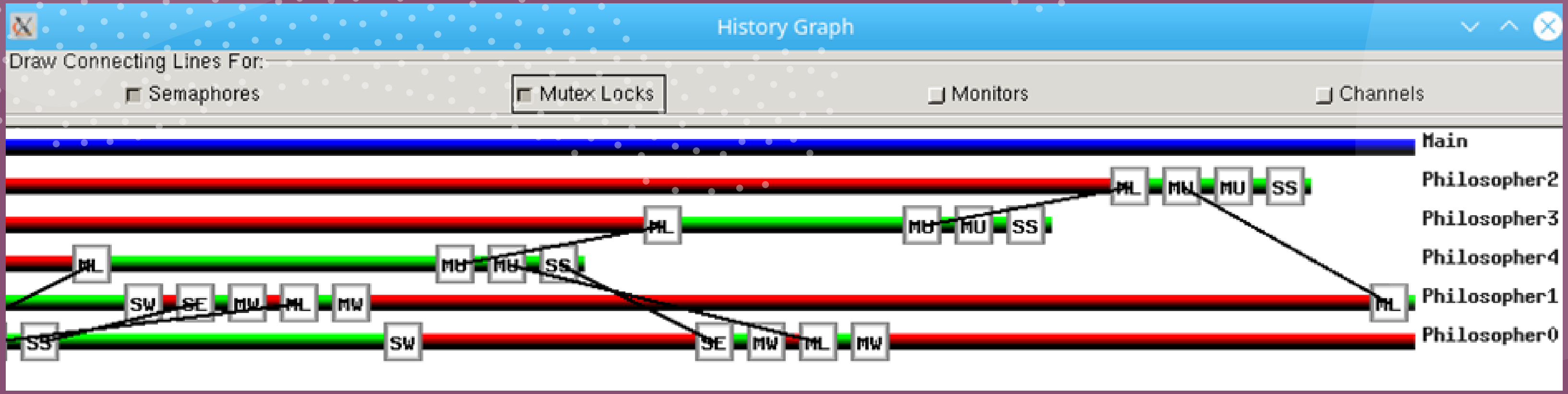
- The top screenshot of the History Graph shows that Philosophers 3 and 1 are eating with Philosopher 4 picking up a chopstick but realising he can't eat so he puts it down. Philosopher 0 proceeds to eat when Philosopher 1 finishes.

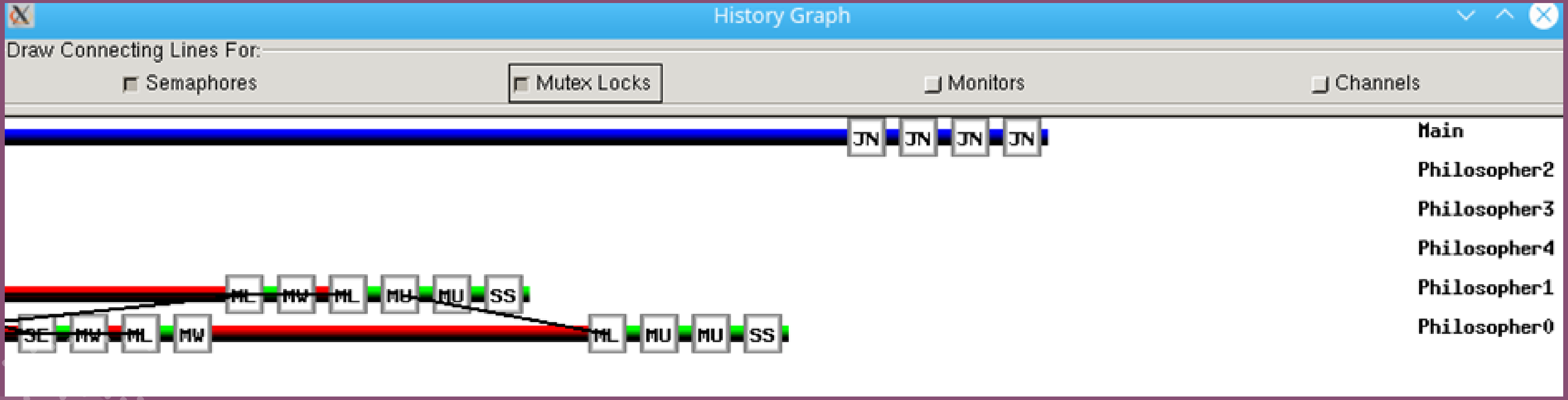


- The bottom screenshot shows the Philosophers switching between eating stages here. Philosopher 4 does get one chopstick, but he is not able to eat so he puts it down. Philosophers 1 and 0 start eating but only after 2 and 3 put their and chopsticks down and stand up.

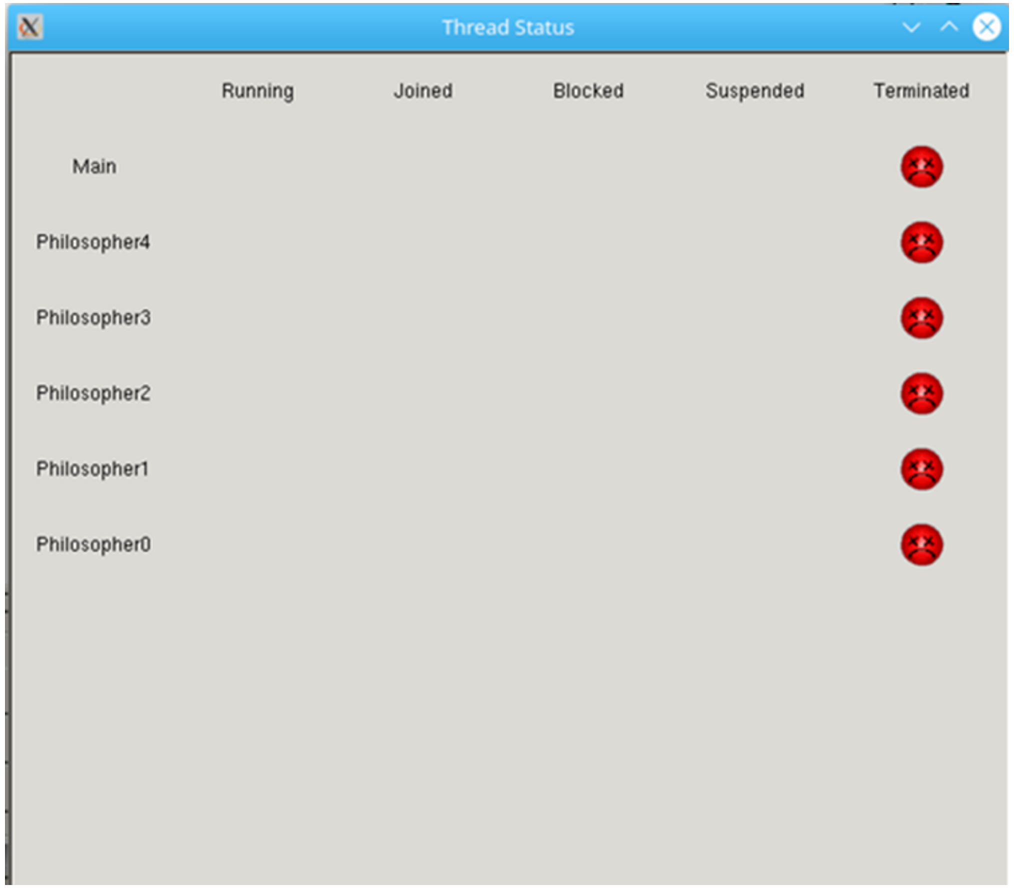


ThreadMentor Continued





Thread Status



Conclusions

- The 4Chairs solution is an effective solution to the Dining Philosopher problem.
- Advantages: no deadlock, simple, scalability.
- Disadvantage: Starvation.

The End