THE DINING PHILOSOPHERS' PROBLEM (Four Chairs Solution)

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

INTRO:

The Dining Philosophers Problem is a concurrency and synchronization problem created by E. W. Dijkstra in 1965, to show the difficulties of resource allocation and deadlock avoidance in concurrent systems (Sari, 2022). In this report, we are going to be discussing the dining philosopher's problem in detail including its background, our solution chosen, and what we have learned from tackling this problem.

BACKGROUND:

There are many issues with concurrency and synchronization in OS. One of these issues are data races. Data races occur when more than one process/ thread use shared data at the same time and try to modify it (Lutkevich, 2021). Deadlock is another huge problem; this is a state that can happen where processors can get stuck and start waiting for resources that are never released. This brings the OS to a halt and not work. Starvation occurs when a process cannot get the service to continue.

Mutex Locks and Semaphores are the two commonly used synchronization methods. A semaphore is a variable that controls access to the shared resources in the OS. A Mutex Lock allows only one of the processes to access a resource at a time. Mutex is good for preventing data races, but it can lead to starvation. Semaphores give efficient allocation; they can control multiple processes, yet they are susceptible to programming errors.

ThreadMentor is a system designed to help visualize multithreaded programming and synchronization. Threads are sequential flows of tasks within a process (Shene, 2001). For our dining philosophers' problem, we will be using ThreadMentor.

THE DINING PHILOSOPHERS' PROBLEM:

The dining philosopher's problem is a very often used example problem to illustrate synchronization issues and techniques for fixing them. The problem goes like this: a circular table with five chairs, each occupied by a philosopher. Each philosopher has their own plate of food but there are only 5 chopsticks on the table in between the food. When a philosopher gets hungry, they attempt to pick up the two chopsticks that are closest to them. A philosopher must pick up both chopsticks to be able to eat. They pick the chopstick on their left first followed by the right. After a philosopher has finished eating, he puts down the chopsticks and starts to think (Tanenbaum, 2006). Each philosopher can only alternately think and eat. If each philosopher picks up their left chopstick at the same time, this will lead to deadlock and starvation as every philosopher will only have one chopstick. There are multiple solutions to the dining philosopher's problem such as the Lefty-Righty mutual exclusion locks solution, the basic mutual exclusion locks solution and then the solution that we have chosen: The Four Chairs Semaphores solution.

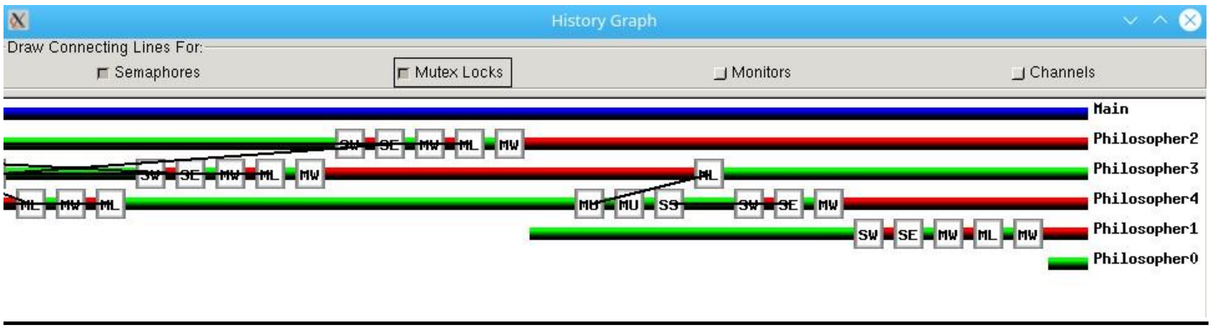
OUR SOLUTION:

The Four Chairs Solution is when a single chair is taken away from the table and there are only four chairs left. The number of chopsticks and philosophers stays the same. This solution utilizes the countdown and lock technique and completely removes the possibility of a deadlock occurring by blocking the 5th philosopher after the 4th philosopher has sat down. Each chair has a semaphore that has an initial value of 4 that counts down and locks and the Four Chairs solution does not have to initialize locks (Shene, 2001).

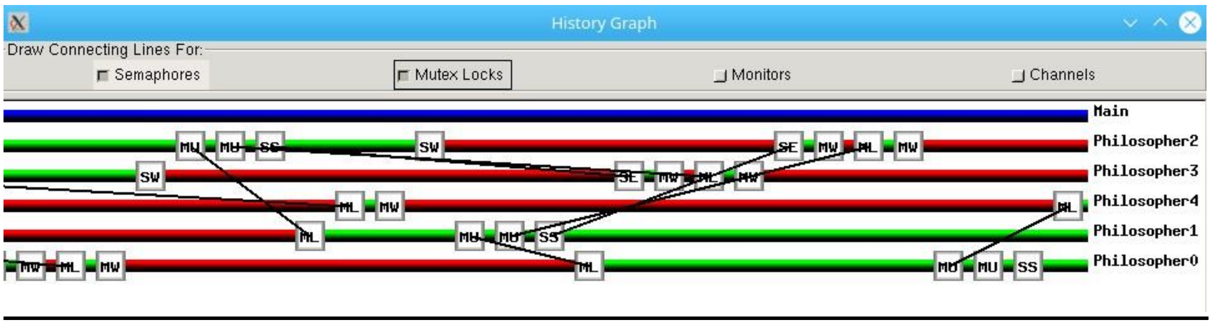
In our implementation, each chair is associated with a semaphore. The initial value of each semaphore is set to 4, indicating that only one philosopher can occupy the corresponding chair at any given time. When a philosopher wants to sit down and eat, they must first acquire the semaphore associated with the chair they intend to occupy. If the semaphore's value is 4 (indicating the chair is available), the philosopher decreases the semaphore value to 0 (indicating the chair is now occupied) and proceeds to eat. After finishing their meal, the philosopher releases the semaphore, incrementing its value back to 4 to indicate that the chair is available for the next philosopher.)

RESULTS AND ANALYSIS:

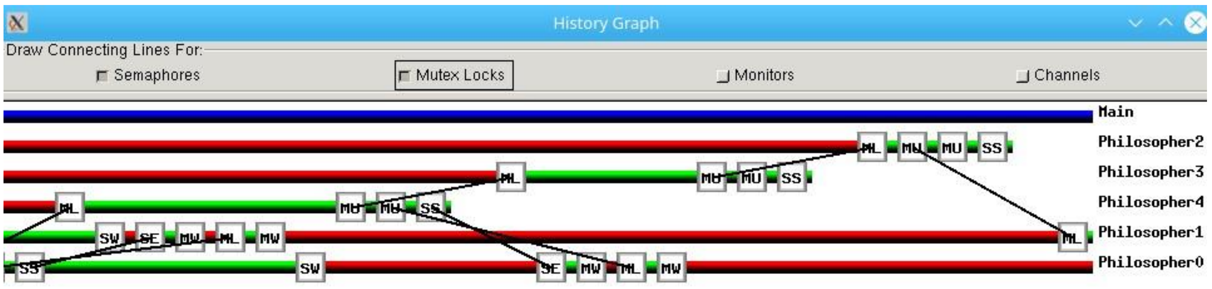
ThreadMentor History Graphs:



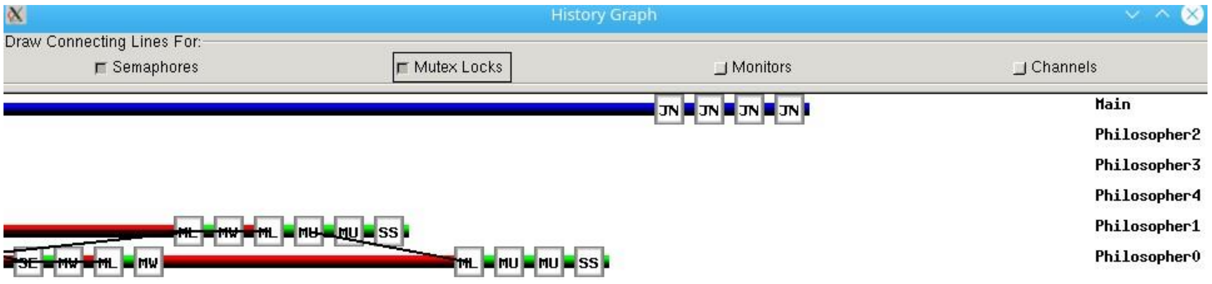
This screenshot of the History Graph shows that Philosophers 3 and 1 are eating with Philosopher 4 picking up a chopstick but realising he can’t eat so he puts it down. Philosopher 0 proceeds to eat when Philosopher 1 finishes.



Philosophers are switching between eating stages here. Philosopher 4 does get one chopstick but he is not able to eat due to only having one chopstick. Philosophers 1 and 0 are able to eat after 2 and 3 put their chairs and chopsticks down.

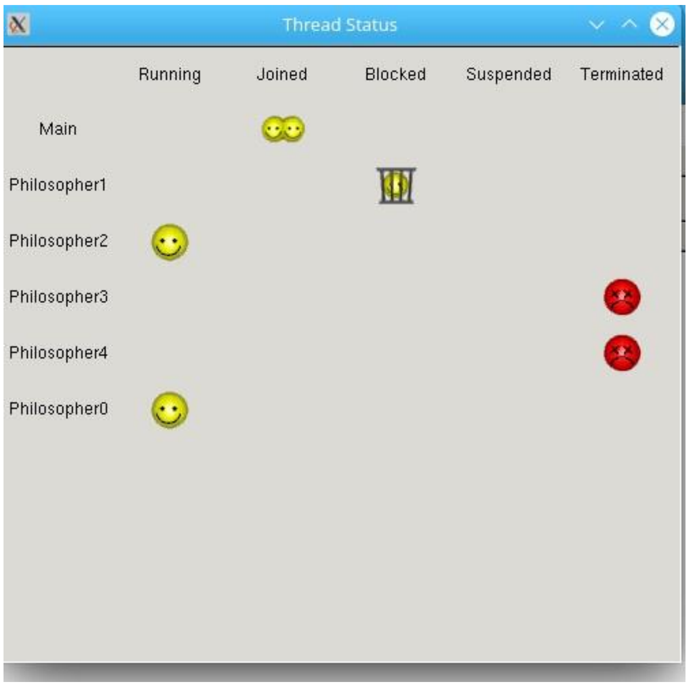


Philosopher 2, 3 and 4 have finished eating. After philosopher 3 finished, he passed the chopstick to philosopher 2. Which then leaves philosopher 1 and 0 to finish eating.



The History Graph ends with Philosopher 1 and 0 switching between each other and Philosopher 0 finishes eating last.

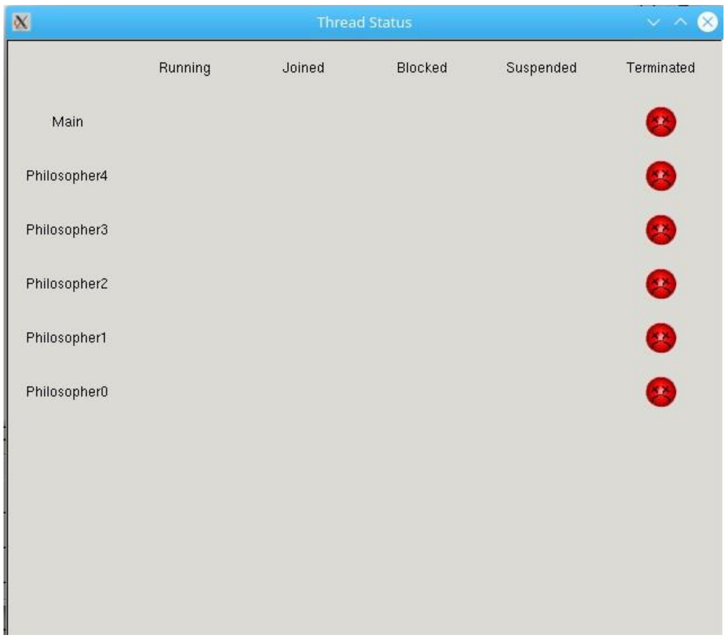
Status Window:



Thread Status

	Running	Joined	Blocked	Suspended	Terminated
Main		🙂🙂			
Philosopher1			▥		
Philosopher2	🙂				
Philosopher3					🔴
Philosopher4					🔴
Philosopher0	🙂				

Here we can see that Philosophers 3 and 4 have both left the table with Philosophers 2 and 0 are still eating their food. Philosopher 1 is currently waiting.

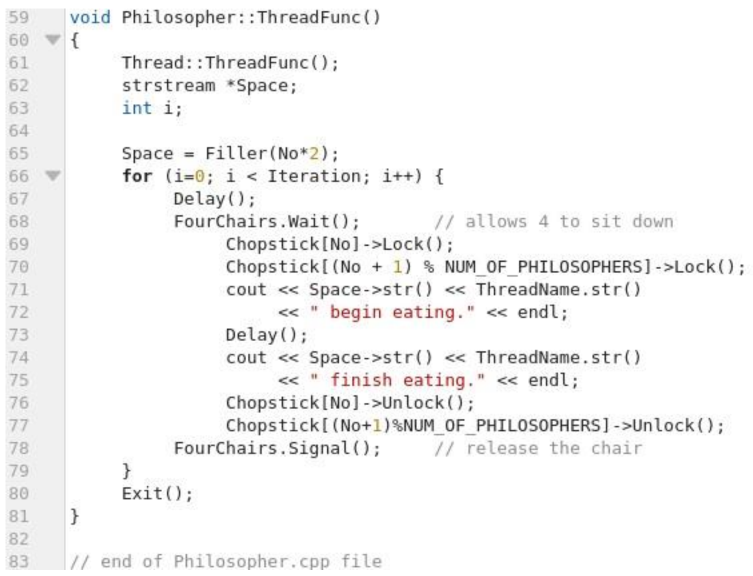


Thread Status

	Running	Joined	Blocked	Suspended	Terminated
Main					🔴
Philosopher4					🔴
Philosopher3					🔴
Philosopher2					🔴
Philosopher1					🔴
Philosopher0					🔴

This status window shows that every Philosopher has finished their food and have left the table to go think somewhere else.

<u>Screenshot of the 4 Chair Solution:</u>



```
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
```

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.

CONCLUSIONS

Our chosen Solution of the Dining Philosophers' Problem was to use the Four Chairs Solution this has demonstrated an effective approach to mitigate deadlock scenarios commonly encountered in other solutions. By reducing the number of chairs available at the table to four while maintaining five philosophers and chopsticks, we have successfully prevented the occurrence of deadlocks.

Advantages:

- Completely Avoids deadlocks: The use of semaphores associated with each chair ensures that only one philosopher can occupy a chair at a time, preventing situations where all philosophers are waiting for their neighbouring philosopher to release a chopstick.
- Simplicity: The solution is simple to implement and understand, making it accessible for educational purposes and practical applications. It also makes it easy to explain in a short demonstration.
- Scalability: The Four Chairs Solution can be extended to accommodate a larger number of philosophers by adjusting the number of chairs, accordingly, offering scalability in solving similar synchronization problems.

Disadvantages

- Resource limitation: although the 4 chairs solution effectively prevents deadlocks, it introduces competition for resources as philosophers may need to wait for an available chair, potentially leading to starvation if not managed properly. (To avoid this, you could implement a Timeout Mechanism where philosophers waiting for a chair are not kept waiting forever so if they cannot get a chair in a certain time they are removed from the queue and given priority)

- Increased complexity with more philosophers: As the number of philosophers grows, managing the access to chairs and ensuring fair access to resources becomes a lot more challenging.

APPENDIX: PERSONAL REFLECTIONS (Josh Hand)

Summary of Learnings: Through this project, I gained a deeper understanding of concurrency and synchronization challenges in operating systems (*when multiple processes or threads attempt to access shared resources simultaneously*), through the project on the Dining Philosophers' Problem. The Four Chairs Solution provided valuable learning into resource allocation and deadlock avoidance strategies.

Likes: I liked creating the presentations as I find it very engaging to re-trace the steps you have taken to get to the end of a project, especially with a group as it allows you to see different points of views on the work.

Dislikes: I dislike how little % of your grade you get for the project considering how time-consuming it is. A lot of other modules would give a much larger % for similar projects.

Areas for Improvement: I feel like I did not communicate very well with my team and would aim to improve on that in future team projects.

Recommendations: Nope

(Hayden Carr)

Summary of Learnings: During the process of this module, I Learned about the Dining philosopher's problem in deep depth by heavily researching our chosen solution called “The Four Chairs” which brought us down the interesting rabbit hole of semaphores as they are the method in which our solution avoids deadlock.

Likes: I enjoyed watching the code run in ThreadMentor as it made reading the code at the same time engaging as I was able to find which line caused each change in the ThreadMentor visual showcase.

Dislikes: I disliked how little the reliance on using linux as it did not run too well on my laptop.

Areas for Improvement: I believe that being more consistent with tests using ThreadMentor would have led to better understanding of the solution rather than doing most of our tests within a few days that we set.

Recommendations: Building upon my areas for improvement for students I would recommend that they try to run the ThreadMentor multiple times a week while reading the code as I believe it would help tremendously with making them understand it. As for the teacher I would recommend more checkups on the groups during the semester as i believe it would keep teams from slacking on work during slow times.

(Kenert Salm)

Summary of Learnings: From working on this project, I have learned about synchronization and how it works. This also helped me learn about the various issues that can occur such as Deadlock or Starvation. I also learned how to read and use ThreadMentor. Overall, this project has helped me learn and improve my skill related to Linux and all related areas.

Likes: I liked seeing the visualisation of our solution for the dining philosophers problem on the ThreadMentor application. It was also great to work on a team.

Dislikes: Learning the full elements related to the 4 chairs solution was a bit confusing at first. Linux is also a pain to use sometimes.

Areas for Improvement: For our first presentation, we could have included more information about the Thread Mentor history graphs and code snippets. We also could learn more about the Dining Philosophers problem.

Recommendations: The lecturer could be more involved in the process of working on the report and presentations, something like a weekly checkup during lab to quickly see what stage everyone is at and provide any help/ feedback if needed.

(Ramazan Iskandarov)

Summary of Learnings: Working throughout this project, I learned many valuable and insightful information about synchronization during my time on working with the Dining Philosopher's Problem, the four chairs solution.

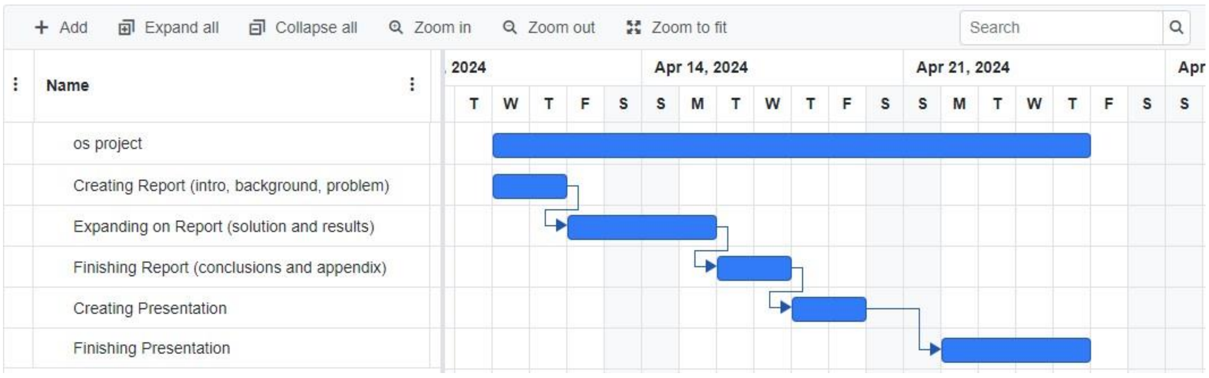
Likes: I enjoyed working with others and working together on the four chairs solution, ThreadMentor was easy to learn and use and allowed our team to better understand the solution and its challenges that it provided.

Dislikes: I did not like the set-up process for the project as there were many problems that occurred before the project had begun.

Areas for Improvement: spending more time tinkering with ThreadMentor is one area I would like to improve more on.

Recommendations: none.

APPENDIX: PROJECT PLANNING AND MANAGEMENT



os project
Creating Report (intro, background, problem)
Expanding on Report (solution and results)
Finishing Report (conclusions and appendix)
Creating Presentation
Finishing Presentation

This is our Gannt chart.

The report was started on the 10th of April. We continued working on the report until the 17th. After finishing the report, we started on the presentation.

We managed our project on a week-by-week basis by using techniques such as scrum and sprint and the Linear project management approach known as the waterfall methodology.

REFERENCES

https://pages.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-philos-1.html

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

https://www.almabetter.com/bytes/articles/process-synchronization-in-os

https://www.geeksforgeeks.org/concurrency-in-operating-system/

https://www.shiksha.com/online-courses/articles/mutex-vs-semaphore-what-are-thedifferences/#:~:text=A%20semaphore%20is%20a%20variable,for%20a%20single%20s hared%20resource.

https://pages.mtu.edu/~shene/NSF-3/e-Book/FUNDAMENTALS/TM-overview.html

https://www.scaler.com/topics/operating-system/threads-in-operating-system/

https://www.baeldung.com/cs/diningphilosophers#:~:text=In%20this%20tutorial%2C%20we'll,the%20problem%20of%20re source%20contention.

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

Process Synchronization info taken from Lecture 5 notes