COP4520 Project Rough Draft

Carlos Sanchez Ruiz Yosha Riley Kariel Sanchez Ruiz Andrew Ballen

***Abstract*—The dining philosopher's problem is a well explored riddle that is often used to represent and teach a deadlocking problem in parallel computing. While the simple and solved nature of this problem would normally not be rigorous enough for a project like this, we are using it to aid us in learning more about an unfamiliar programming language for us, Rust. The problem goes like this: Five philosophers sit at a dining table, there are the same amount of forks placed between them. The philosophers can do two actions eat or think. They need to pick up the two forks around them to eat, then when they are done eating they will replace the forks and go back to thinking. The philosophers represent threads and the forks represent locks. Our objective is to develop an algorithm that efficiently allows the philosophers to eat and think without causing any deadlocks.**

I. Introduction

What we hope to gain from our project is to use concepts we learned in class which involve concurrent objects, mutual exclusion and learn the language Rust. Rust is a coding language (a systems programming language) that was designed to be safe, fast and allow for concurrency. It has a user-friendly compiler that with it brings integrated package managers and also prevents crashes in the code. To put it in simpler terms it wont allow the code to run if it sees that it will cause a crash instead it will show in the compiler to the user what changes to make to prevent a crash and allow the code to run. We decided to do a simple problem first that would allow us to learn the language while tackling parallel concepts and chose the dining philosophers problem.

II. Problem Statement

The Dining Philosophers problem states that there are N number of philosophers/threads around a table that have an infinite appetite and will never be full. They all have a fork to their left and to their right and must have both picked up to be able to eat. They can either think or eat, and they must not starve which is represented by reaching a deadlock.

III. Related work, evaluation and discussion

Rust is becoming a more common programming language for systems programming and parallel computing tasks. There are several features of the language that make it a natural choice for these problems and elevate it above the closest competition; C++ and Java. Rust offers a pseudo garbage collector which makes it feel like Java or any other garbage collected language to the user, but has the consistency and memory safety that comes with advanced manual memory management that you would find in C or C++. Rust has a large set of tools which helps with parallel and concurrent programs that the creators have dubbed "fearless concurrency". This is built up by the strictness type of the type system and using an ownership system over raw pointers. The ownership system allows for sharing memory and variables in a more nuanced way than with only pointers. The ownership rules force us to move around or clone values rather than just referencing them, not only does this allow for the rusts memory management system to exist, but it also saves us the headache of dealing with multiple threads having the same pointer, as its just not possible in Rust. The other part large part in Rust's tool kit is the compiler itself. The compiler will not compile unsafe code or code with common concurrency bugs, it will instead fail and then tell you what is wrong with the code. Rust's compiler has much more advanced error reporting most programming languages and is does a unique thing where it is actually helpful. We decided to use this language to take advantage of the many features it has while exploring the dining philosopher's problem and hopefully have consistent solution using Rust.
Advantages of using Rust

1) Allows for easy and quick debugging as the compiler helps solve common concurrency issues.
2) Compared to Java it is much faster, and C++ it is as fast or faster.
3) It is safer than C++ and Java, the language will not compile unsafe code.

Disadvantages of using Rust

1) Rust is more complicated than other languages to learn
2) Due to the learning curve it may take longer to solve a problem in Rust
3) Rust is less commonly used, so there are less online resources to it

Overall, Rust is good at solving concurrency problems which is why we chose to do the Dining philosophers problem for our project as it is a complex concurrency problem.

Common issues of the problem

1) Two philosophers pick up the same fork
2) One philosopher picks up a fork and another one picks up the only other available fork meaning the philosopher starves (causes a deadlock).

To solve these common issues, we will have to use locks to make sure once a thread uses a fork another one cannot which in theory should solve issue number 1. To solve issue number 2, we will have to design our code so that the other fork available to a philosopher once they pick one up is inaccessible. This will require for

the variable storing forks to be accessible by all threads.

How using Rust positively affects this problem

1) We can share data between the threads using Arc and mutex
    a) This will let us tell a philosopher if a fork is available for them or not
2) We can spawn child threads in Rust which can make on thread relate to another
3) Rust can lock data meaning that once a thread/philosopher picks up a fork we can lock it until the philosopher drops the fork which then will make it accessible to other threads.

Concept of how we want to solve

In our solution, we would prefer to make an algorithm that requires minimal locking. There is a fairly simple solution we could use, where each philosopher in turn is the only one able to eat, and thus guarantees their access to the forks, but this solution allows much less work to be performed over all. It would be best if our solution allowed multiple philosophers to eat at the same time. To accomplish this, we are considering the use of semaphores. In our case, we are considering having an array that represents how many forks are available for each philosopher at any given time. when a philosopher ”picks up” their forks, they update the the available forks values for the philosophers to the left and right, and the same goes for when they place their fork down. This will keep philosophers from attempting to pick up forks when the fork is not available.

IV. Technique

We have it so that we have 5 threads representing the 5 philosophers. Then after this we decided to have boolean sate variables for the forks that will be to the left and to the right of each philosopher.At first all 5 philosophers will run parallel to see which one grabs a fork first and it will be to their left hands. For example lets say philosopher 3 grab left hand fork first the others philosophers would now contemplate and not pick up a fork to ensure that a scenario like all philosophers picking up the fork to their left hand making no forks available to each other occurring. So back to philosopher 3 he picks up his left fork everyone else contemplates then he grabs fork to his right, then eats, then states that he is done. After this the philosophers would race again parallel wise for same rotation.

V. Conclusion

We were able to become more familiar with rust by putting the language into practice a simple concurrent/parallel problem. We became familiar with how to make threads in the language, we worked with mutex and at the end figured out how to join the threads and run the program. For the conclusion for the problem we were able to solve the problem by making it so that each thread can represent either, ”think”, ”pick up forks”, ”eat”, then ”put forks down”. For think we just have the threads state that they are contemplating, for picking up forks we have if one thread picks up a fork they will pick up the other fork they need to eat while other threads contemplate to not run into deadlock then they will eat and after state that they are done which will basically let all other threads know a new rotation has started.