

CS5541 - Dining Philosophers

Brandon Rodriguez

April 17, 2020

In all of the following implementations, each thread is considered a "Philosopher". Each thread will attempt to grab forks, and when it has two forks, it will take a single "bite". After 1000 bites, the thread is considered "full" and will not eat or access forks further.

1 Correct Implementation

Compiled and executed via "**make correct**" in terminal.

It's easiest to start by describing how to implement the "correct" (no starvation or deadlock) implementation. For this, I started by making threads that are given a unique thread number, starting at 0.

Once initialized, all threads are set up and ready, they start attempting to "eat". This is done by each thread attempting to get a left and right fork. After the attempt, the thread checks if it was able to grab both forks. If it wasn't, it puts down any forks it currently holds. If it did grab both forks, then it "takes a bite" and then puts the forks back down. Each fork is guarded by a unique mutex variable to make accessing and releasing threadsafe.

This repeats until all threads "are full" and thus done eating. As more threads finish, it becomes easier for the remaining threads to grab forks and also finish.

2 Deadlock Implementation

Compiled and executed via "**make deadlock**" in terminal.

The deadlock implementation will occasionally deadlock. This is accomplished by copying the original "correct implementation" code, and then adjusting it

slightly.

In this version, the program will work the same up until threads pick up a fork. In this version, they will hold onto the fork unconditionally, regardless of if they were able to grab two or not.

Then, if the thread had two forks, it will eat and put the forks back down. If the thread did not have two forks, it will just keep trying to grab two forks until it succeeds.

This can easily result in all threads picking up a single fork. They will each hold onto their single fork while attempting to grab more. But since no threads are putting the forks back down, there are no more forks to grab. Deadlock occurs.

3 Starvation Implementation

Compiled and executed via "**make starvation**" in terminal.

The starvation implementation will basically always starve out one or more threads until some others complete. This is accomplished by copying the original "correct implementation" code, and then adjusting it slightly.

In this version, the program will work the same up until the threads eat. In this version, the threads will hold onto the forks even after taking a bite. They don't put the forks back down until 100% done eating. Thus the threads directly adjacent will be unable to eat for the entire duration of the thread's life. The result is that some threads will be very low in bite count while others are finishing up.

Note that the "correct" implementation seems to generally have all threads at roughly the same bite count (at least on my machine), and thus is "starvation free". However, in some runs, thread execution order does still result in some threads being as low as 500 bite count while others are finishing up. But not usually.

To correct this, the program would need to have an additional threading metric that tracks how many bites each philosopher has. Then fork priority can be given to some threads over others, based on this metric.

Having said that, the "starvation" implementation is still much worse. It has

"true starvation" where a thread won't get any bites at all until another one is fully complete.