The Dining Philosophers Problem

CS 444: Operating Systems II

Fall 2017

Abstract

In our second concurrency assignment, we solved the Dining Philosophers Problem. Five philosophers eat infinite spaghetti and think around a dining table, but each needs two chopsticks to ensnare the slippery noodles. Our challenge is to ensure that all philosphers can eat and none dies of starvation. The solution we implemented makes use of a arbitrator (waiter) function to give permission to each philospher to eat. Our waiter manages locking and unlocking of global mutexes to ensure that no philosopher goes hungry.

Jacob Mahugh, Alannah Oleson

October 27, 2017

1 Solution Description and Design Decisions

First, we created a philosopher struct to hold information such as the philosopher's name and the numbers of chopsticks on their left and right. We also created an array of integers to represent the positions of the chopsticks on the table. As defined in the assignment, we created functions that the philosophers could use to eat and think.

After we looked at various ways to solve this concurrency problem, we decided to implement what is called the arbitrator solution. To do this, we have a global mutex that each of the threads checks the status on when it wants to eat. When a thread requests to pick up the two chopsticks next to it on either side, it must first receive permission to do so from a function that checks if the two chopsticks next to the asking philospher are available to be picked up. If they are not, the querying philospher simply asks again until he gets permission to pick up both sticks. If both can be picked up, however, the philosopher (which, of course, is represented by a thread) locks a global mutex corresponding to the chopsticks that he wants to pick up. If this lock is successful, he can pick up both chopsticks, eat for a random amount of time, then set them back down (unlocking the mutexes). As philosphers do, he will then think for while before he attempts to eat again.

In the main function, initialize the philospher structs and spawn five threads to represent these five philosophers. Each of these threads calls the philosphize() function, which kicks off an infinite loop of thinking, grabbing, eating, and setting. This loop will continue until the user kills the process.

2 Testing Details

To test our code, run the "make" command on the files enclosed in the tarball and type "dinnertime" to run the program.

The program will start running and outputting the various statuses of the philosphers to the screen. You will see four messages:

- ¡Name¿ is thinking for X seconds! Go ¡Name¿!
- ¡Name¿ is grabbing chopsticks A, B! Dinner time!
- ¡Name¿ is eating for X seconds! Delicious!
- ¡Name¿ is letting go of chopsticks A, B. He ate about X noodles!

These messages correspond to the philosopher thinking, grabbing, eating, and setting, respectively.

To check for correctness, simply check the output for the following conditions:

- Philosophers should think, grab, eat, and set, in that order.
- Philosophers should only grab the chopsticks that are nearest to them.
- A philospoher should not pick up a chopstick that another philosopher is currently using.
- There should be at least one point in the output where multiple philosophers are eating at the same time, given that the process runs until it is killed.
- The process should not hang (become deadlocked).

3 What We Learned

This assignment was a valuable lesson in race conditions. Once we got our code to the point where we were fairly sure it was correct, we encountered a weird bug with the output that seemed to indicate that philosophers were picking up chopsticks right before the guy before them put them down. This caused us to inspect our mutex locking and unlocking code to make sure that we were doing that correctly. However, we eventually realized that the placement of our print statements was causing the philosopher holding the chopstick to set it down (allowing the next philosopher to snatch it up and print his statement) and print his statement only after doing some tear-down work. This made it appear that the mutexes were stepping over each other, while in reality everything was going according to plan and the statements were just printing at inopportune times. To fix this, we added a slight waiting time before the second philosopher prints his statement and rearranged our print code slightly.

This example demonstrates the importance of testing for and handling race conditions in code - if this application were to be more critical, this could have easily caused a failure.