Bo Broyles

Operating Systems CS475

Lab 6

Dining Philosophers Problem

Creating Deadlock:

In order to create deadlock in my Lab 5 code I made sure every philosopher picked up their left chopstick first, proceeded to wait for 20 seconds, and then attempted to pick up their right chopstick before eating. Because the philosophers would not put down their left chopstick until they were able to pick up their right chopstick and eat, it resulted in deadlock. The thinking time was less than the time they waited to pick up their right chopstick after picking up their left and eventually every philosopher picked up their left chopstick and was waiting for their right chopstick. The only code required for this was ‘sleep(20)’ placed after picking up the left chopstick and before picking up the right chopstick.

Attempted Solution:

To solve the deadlock problem I found an algorithm created by some Penn State professors (<https://arxiv.org/pdf/cs/0109003.pdf>). The link is to their paper which I found using google scholar. The idea of the method was that the chopsticks were all given a random integer value, assigned by each philosopher. The chopstick with the higher value was picked up first. Next the philosopher would wait until that chopstick was available to be picked up, then (because I was trying to force deadlock) the philosopher went to sleep for 10 seconds. After awaking the philosopher attempted to pickup the other chopstick, if it was available it was picked up and the philosopher would eat. If it was not available (already being held) then the philosopher would drop their current chopstick and the process would occur again. This way the philosophers are not holding on to a chopstick if the other one is not available and in turn not causing a situation where deadlock can occur. One thing I noticed about the solution is that occasionally the printing out of the statements to the terminal can get interrupted but I think that has more to do with the threads themselves than the algorithm.

Analyze the Method:

The program in terms of concurrency is very effective and I would say this is one of the strengths. This is because the threads are able to work at the same time when waiting for the first choice chopstick to become available. Using the semaphores the threads are able to check whether the chopstick they want to pick up has become available with the other threads are eating or asleep. By doing this the chopsticks are able to be picked up and locked right when they are made available by the other thread. Another advantage of this algorithm is that there wont be an instance of deadlock. The biggest reason for deadlock being prevented is the fact that once one chopstick is picked up the other chopstick’s availability is checked, and if it is not available then the first chopstick is put down. By making sure to put down the first chopstick no philosopher will be holding one chopstick while waiting for the other, other than when they are put to sleep which is only being done to try and ensure that deadlock was occurring in my lab 5 code. As the philosophers will put down the first chopstick if the second isn’t available then deadlock won’t occur. This is one of the major advantages of the solution. One of the disadvantages is that there is a very small possibility of starvation occurring. This could occur when one of the threads assigns a nr value to the chopsticks and keeps trying to pick up the same chopstick first. When this occurs then as one chopstick is put down the same chopstick will be picked up again. This can happen for multiple threads and repeat in a cycle causing starvation for some of the threads. The likelihood of this occurring however is very slim because the priority is randomly assigned using the rand() function and the rand function is seeded for time meaning the numbers that are being generated should change every time the function is called. When I first ran the program starvation was routinely happening but I was able to change the assignnum function to ensure that the random nr values were being assigned to the chopsticks and the starvation issues began to subside. If starvation occurs now it is only temporary and the philosopher thread should be able to eat eventually. I would say in terms of simplicity for the algorithm the logic is fairly straightforward. The two major changes were the priority assigning of the chopsticks and the putting down of a chopstick if the pairing chopstick wasn’t available to be picked up. Because I already had created a struct for the chopsticks and given them an available field in lab 5 this implementation was fairly straightforward. I think it is a solution that would work for many other set ups of the problem and one that many people could look at and put it into place quickly. The whole solution consisted of only two functions, one simply generated a random number and the other (to put it simply) picked up one chopstick and then checked if the other necessary chopstick was available. I think the workload on the operating system is fairly minimal because the only deadlock detection being done is checking an integer value for the chopsticks. The locking and unlocking of the semaphores also only occurs one more time (potentially) than it did in lab 5. The only additional strain on the operating system would be the locking and unlocking of the semaphores occurs more frequently in this solution than in the original solution. However, I am not sure if this is more taxing on the operating system than holding a semaphore locked for long periods of time.

Comparison:

<https://dl.acm.org/doi/pdf/10.1145/567532.567547>

One of the other solutions I looked at was the one above published by academics at the Mathematics Institution at Hebrew University in Jerusalem. There were many similarities between the two solutions, namely the idea of randomly choosing which chopstick to pick up first and making sure the probability of picking up either chopstick first is equal. This is important because one of the original causes of deadlock was occurring if all the philosophers picked up either the right or the left chopstick first. Another similarity between the two solutions was the placing down of the chopsticks if the other necessary chopstick wasn’t available. The major difference was in my solution I would keep looping back trying to get both chopsticks picked up before going back to think. In this solution the program called for the philosophers to think again if both of their chopsticks werent available. I think this solution is much more likely to result in starvation because it is more likely that the philosophers won’t be able to pick up both of their chopsticks at the same time and in turn will spend a longer time thinking. I think it would be easier on the operating system however because there will be less locking and unlocking as the most locking occurs when the philosophers eat because both chopsticks are picked up. I also think that many of the programs have the philosophers “sleep” when they are thinking so this solution would result in the threads being asleep for longer periods of time. The odds of deadlock in this case seem to be zero however because anytime the chopsticks aren’t available to be picked up they are simply set down so no philosopher is ever holding a chopstick and preventing another philosopher from picking it up indefinitely.