# **Task 1: Dining Philosophers (Busy Waiting)**

1. When implementing and testing your solution, did you notice any deadlock? Deadlock occurs when threads cannot progress due to improperly controlled access to critical resources. (In this case, the critical resources are the chopsticks.) How did you solve any deadlock problems?
   1. There were some deadlock issues encountered. I solved these problems by using a multidimensional array of bool's for the chopstick, having philosopher i try to access chopstick[i][i+1] if chopstick[i][i+1], chopstick[i-1][i], and chopstick[i+1][i+2] aren't being used.
2. Make the philosophers yield between attempting to pick up the left and right chopsticks. Run at least 3 tests with various parameters and -rs seeds. Record your findings and explain your results. Undo any changes made to accommodate this question before submitting your assignment.
   1. Our solution to resolving the problem does not allow yielding between picking up chopsticks, as it picks up both chopsticks simultaneously. However, if we were able to do it, we suspect that other philosophers would try to pick up the chopstick that is just about to be picked up, but would still not be able to because it would be waiting in a busy waiting loop. We may be completely wrong.

# **Task 2: Dining Philosophers (Semaphores)**

1. Run your solutions to Task 1 and Task 2 with the same parameters, including -rs seeds. Note the number of ticks that NachOS runs. Do this for at least 3 different sets of parameters. Record your results in a table, including -rs seeds, number of philosophers, and number of meals. Explain your findings.

Task 1

|  |  |  |  |
| --- | --- | --- | --- |
| **-rs Seeds** | **# of Ticks** | **# of Philosophers** | **# of Meals** |
| 123 | 5651 | 4 | 50 |
| 87654 | 3060 | 9 | 20 |
| 6666 | 13784 | 20 | 100 |

Task 2

|  |  |  |  |
| --- | --- | --- | --- |
| **-rs Seeds** | **# of Ticks** | **# of Philosophers** | **# of Meals** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

1. As with Task 1, make the philosophers yield between attempting to pick up the left and right chopsticks. Run the same tests used in Task 1 and record the results. Were the results the same or different? Why? Undo any changes made to accommodate this question before submitting your assignment.
   1. After putting a yield in between the text in our test() function that states that the philosopher has picked up a chopstick, nothing has changed in the output. With the state system we use, the philosopher is already set to eating at this time, so this will protect his chopsticks from being taken or an interruption occurring, as each time a philosopher attempts to pick up a chopstick, it checks to see if the philosophers next to him are eating first.

# **Task 3: Post Office Simulation (Busy Waiting)**

1. Explain the method you used to resolve the deadlock problem. Why did you choose this particular method?
   1. To solve the deadlock problem we have the thread yield for three times and if it can't send a message it goes and read all mail in his box. We choose this way to let the thread read all his message just in case someone one was trying to send something in their mailbox.

# **Task 4: Post Office Simulation (Semaphores)**

1. Did you experience any deadlock when testing this task? How was it different from Task 3?
   1. When working on the project I didn't have to experience any deadlock. Task four is different from Task three because you have to make sure that no deadlock happen and dealing with deadlock.

# **Task 6: Report**

1. In your own words, explain how you implemented each task. Did you encounter any bugs? If so, how did you fix them? If you failed to complete any tasks, list them here and briefly explain why.
   1. Task 1: The user is asked for input on how many philosophers and meals there are to be. A thread is created for each philosopher, and is forked to a dine function. In here, each philosopher enters and then sits by using yields. The function goes to a take chopsticks function, which sets the state to HUNGRY, and tests if they can eat. The test function will check if the philosopher is hungry, and the philosophers to the right and left aren't currently eating. If these all pass, then the function will check a multidimensional array of booleans used to represent the chopsticks. It will check if chopstick[i][i+1], chopstick[i-1][i], and chopstick[i+1][i+2] are available, and set chopstick[i][i+1] to true, allowing them to eat. The number of meals is decremented by 1, and they eat for 2-5 cycles. If they aren't hungry, or their neighbors are eating, they will yield until they can. The take chopsticks function will then call the put chopsticks function, which will set the chopsticks back to false, and set the philosopher to thinking, and then will call the test function on the left and right neighbors. All of this will continue until there are no more meals left.

The task of getting all of the philosophers to leave at the same time was not able to be fully completed, and so there will on certain runs still be a few philosophers remaining waiting to leave. Hence why the table above for task two is empty.

* 1. Task 2: This is essentially the same as Task 1, except when checking for chopstick availability, a semaphore is used instead of a busy/waiting loop. A philosopher will attempt to take his chopstick by setting a state corresponding to that philosopher to hungry, then testing to see if the philosophers next to him are currently set to eating, all in a mutex semaphore that ensures only 1 philosopher can be testing for this at the same time. If the philosopher can pick up his chopsticks, he is set to eating, the number of total meals is decremented, and that information is reported. The philosopher then puts down his chopsticks, setting his own state to thinking and calling the test on the philosophers next to him. This continues until there are no meals left.

The task of getting all of the philosophers to leave at the same time was not able to be fully completed, and so there will on certain runs still be a few philosophers remaining waiting to leave. Hence why the table above for task two is empty.

1. What sort of data structures and algorithms did you use for each task?
   1. Task 1: Enum used to dictate philosopher’s state, multidimensional array of Boolean values for the chopsticks, Semaphore for mutex, several Integers for variables. Algorithms used include the dining philosophers algorithm found on Moodle.
   2. Task 2: Enum used to dictate philosopher’s state, multidimensional array of Semaphores for the chopsticks, Semaphore for mutex, several Integers for variables. Algorithms used include the dining philosophers algorithm found on Moodle.
2. Task 3
   1. For the task I used a colaboration of many functions and global variables including pointers pointing to pointers of arrays containing arrays. The task does not 100% work correctly. The correct number of messages still gets sent and the people read and compose messages, but some people will enter the post office and leave without composing or reading. These people still complete the tasks while entering and leaving differently than instructed. I ran short on time to implement the program due to an unexpected family matter I had to take care of.
   2. I used double pointer boolean array as the main structure of my program. It determined whether or not there was mail in a person's box.
3. Task 4
   1. For the task I use the semaphore to watch over the mail box size. While doing this I have trouble know how the semaphore work and figure out how to get it to work with my program.
   2. For this task I use array of Semaphore as my main data structure.