Steven Babineaux smb0007
Jarret Donne jad8149
Deandre Metoyer ddm1025
Ian Beatty imb4769

Task 1:
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?

- We never had any deadlocks. But we wish we did because that way we could have tested our solution in a scenario we knew created a deadlock. That being said, we are still confident that we handle deadlocks appropriately.

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.

- When I run with the extra Yield all that this is doing is changing threads after picking up the left chopstick, regardless of whether or not the right chopstick is available. But this section of the code isn't protected. There can be an interrupt in between picking up the left and right chopstick. This Yield does not change the output.

Task 2:
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	Task 2	Philosophers	Meals
2651	3132	10	10
5574	7620	20	20
15520	35322	50	50

- This is not what I expected. I would have expected the semaphore solution would have been more efficient. However either do to lousy coding or bad intention it seems like the busy waiting loop solutions works faster.

2. 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.

- In our semaphore solution we don't allow interrupts in between picking up the left and right chopsticks. So by adding a yield in between the left and right chopstick it forces the thread to change. So other threads can continue as long as they don't try to pick up a chopstick because that is still protected.

Task 3:
1. Explain the method you used to resolve the deadlock problem. Why did you choose this particular method?

- When a person would try to send a message a certain amount of times while that person's mailbox was full, the sender would then give up and leave the post office to prevent all mailboxes being full with no one else to send to.

Task 4:
1. Did you experience any deadlock when testing this task? How was it different from Task 3?

- We did not because the semaphore protecting the sending station would prevent it.

Summary:
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.

- For each task, we looked at the procedure document and wrote out the pseudo code to have a basis. We then coded the pseudo code with some tweaks and changes. We ran the code and fixed everything accordingly. There were many bugs throughout and to fix them, we would debug using print statements, follow the print statements, and find where they did not make sense. We would fix that problem, then do it again until there were no more lines that did not make sense or were out of order. We did not fail to complete any tasks.

2. What sort of data structures and algorithms did you use for each task?

- We used semaphore arrays and if conditionals. While loops were used for the busy waiting loops. Inside of these, we had global variables that all threads could access.