Assignment 2 – Deadlock, Starvation and Mutual Exclusion Testing

By Jacob Boyce
19/08/2022

Testing Methods, Specific Tricks Applied

Testing for all three problems was conducted using the following methods

Debug Statements. This is done by using `System.out.println` debug statements to output key information about the state of the program at key moments. By inspecting the console output, information can be quickly derived about what is causing a bug. This is particularly useful for detecting mutual exclusion errors, for example, with P1 if two farmers begin crossing the bridge after the ‘`NEOM`’ line is printed then it is clear, that mutual exclusion had not been enforced correctly. Similarly, if a farmer is not producing any debug statements, then either starvation or deadlock has occurred, depending on whether the program has properly terminated or not

Automatic Testing. This is done by using additional code within the program to assist with testing. This is very useful for testing for starvation, deadlock, and mutual exclusion. For example, to ensure that bridge is starvation free a piece of code was written to record statistics on the amount of bridge crosses each farmer completed. These statistics were then outputted after the program execution. To test for deadlock and mutual exclusion issues, high-volume test cases are used, for example in Test Case 5 for P2 and P3, 10000 customers arrive at the parlour. As race conditions are probabilistic in nature, by simply having a high-volume test case it is more likely for any that exist to surface. A small testing script can be made to enhance these high-volume tests, for example when running Test Case 8, for P2 and P3 which had 5000 concurrent customers, a loop at the end checked if the output given was equal to the expected output by looping through arrival and leave times.

Edge Cases

Note. all edge case tests can be found in ‘`P1.java`’ and ‘`ParlorProblem.java`’ for P1 and P2/P3 respectively at the bottom of each file. Notable edge cases are described below

P1

- 10 Farmers start northbound. As expected, all 10 farmers cross from south to north, before they proceed to cross from south to north again.
- 10 Farmers Southbound, 1 Northbound. This test is to ensure the southbound farmers do not starve the northbound farmers. As expected, all farmers cross the bridge at least once every 11 crosses.
- 1 Northbound farmer. This is a simple bounds test. As expected, the singular farmer crosses back and forth by itself.
- 100 Farmers. This test ensures that none of the farmers deadlock or are starved. As expected, each farmer crosses the bridge at once before the simulation terminates.

P2/P3

- 1 Customer enters the parlour at t=5 with an eating time of 5. This is another simple boundary edge case. As expected, the customer seats at t=5 and leaves at t=10
- 11 Customers with 0 eating time. This is to ensure that the program works as expected even if customers have 0 eating time. The program worked at expected with all the customers arriving at t=0 and leaving at t=0.
- 6 Customers arrive at t=0 with an eating time of 5, another 6 arrive at t=20 with an eating time of 5. This test is to ensure that even if there is a gap in customer arrivals the program still works correctly. The program works as expected admitting 5 customers in the parlour at t=0, then the final at t=5, then the next 5 at t=20, then the last one at t=25
- 100 Customers arrive at the same time. This is to ensure that the parlour correctly processes all the arrivals without deadlock or starvation. This works as expected with the parlour taking 5 customers out of the queue at a time over and over until the waiting queue is empty.

Specific Issues

One issue that occurred in P2/P3 was that the parlour would continue to the next timestamp even if not all customers had finished their events. The presence of this issue was found using the testing function which automatically checked if the output was correct when running Test Case 4. The race condition was subsequently found when reviewing debug statements on the console.

Another issue found in P3 was deadlock. This would occur because ‘`notify()`’ would be called before ‘`wait()`’ for customers. As a result, the customer would be waiting forever. This was found when running Test Case 8 over 10000 customers and seeing that the program did not terminate and then once again reviewing debug statements to find the root cause.