OSC Project 2 Report

Dieks Scholten - s1098110

October 2024

1 Implementation

I created two classes: one for the buffer (class Buffer) and a class for the logger (class Log). The buffer class has a buffer vertex, an integer bound and two mutexes (one for the buffer vertex and one for the bound integer). It also has a pointer to an instance of the Log class. I used a separate lock for the bound variable (which indicates the buffer vector's max size) to allow for more concurrency. When changing the bound variable you can release its lock earlier then the lock for the buffer.

My implementation does not suffer from race conditions because each-time a buffer function is called which changes either the buffer vertex or the bound integer the appropriate lock is called. The same holds for the logger class.

2 Tests

I designed 5 tests for my code. These are called in the main function and will be run immediately when executing main.cpp.

Test Explanation and Purpose

1. Test 0: This test writes ten numbers to the buffer on the main thread. The function of this test is test the functionality of the buffer, disregarding any thread handling. Since there are few buffer actions called, the log will be quite short, so the log is printed and can be validated to ensure it is functioning properly.

2. Test 1: This test creates 10 threads and lets each thread write their thread ID (0 to 9) 50 times to the buffer. After this, the buffer and the count for how often each ID occurs in the buffer are printed. The purpose of this test is to check whether writing to the buffer concurrently works, and whether any values are lost or duplicated.

3. Test 2: This test is exactly the same as Test 1 but writes the thread ID 500 times instead of 50. This ensures that the threads work concurrently and that no threads finish before the others are created.

4. Test 3: This test again creates 10 threads and lets each thread write its thread ID 50 times to the buffer. In addition, two threads are created that will simultaneously read from the buffer. Then, the buffer is printed, and the occurrences are validated, as well as the total length of the buffer. The purpose of this test is to ensure that the writing and reading operations can work concurrently and that no values are read without being removed from the buffer.

5. Test 4: This test creates 10 writing threads, exactly like in Test 1 and Test 3, but one of these threads (thread 6, about halfway through) bounds the buffer to a size of 80. We expect the buffer to already be bigger than that, so this test ensures that reducing the size of an oversized buffer and bounding it works as expected.

Test Expectation and Result

1. Test 0:The expected result is a buffer containing *0123456789*, followed by the log showing "write successful" ten times and "read successful" ten times. This is exactly the result produced by the test in the program.

2. Test 1: The expected result is a buffer of size 500 with 50 occurrences of each number from 0 to 9, and the buffer is not really ordered. The result varies slightly. The buffer is of size 500, and each number occurs 50 times without fail. However, the buffer is sometimes nearly sorted, which happens because thread creation takes time, and writing 50 numbers can sometimes occur within this thread-creation time. This is why Test 2 was created to ensure the results are proper.

3. Test 2: The expected result is the same as in Test 1 but with a buffer size of 5000 and 500 occurrences of each number. The results are as expected. There is little structure in the buffer, but sometimes there is a long string of the same number. This seems to be the result of the nondeterminism of correctly functioning threads. Overall, Tests 1 and 2 have results as expected.

4. Test 3: Based on the results of Test 1, the expected result here is that the buffer will have a size of 480 and will be almost sorted. Specifically, the expected result is around 30 occurrences of 0, with any extra occurrences likely missing from numbers 1 or 2. Numbers 3 to 9 should occur 50 times. This is exactly the result produced by the test.

5. Test 4: The expected result for this test is a buffer length of 80, with most likely 50 occurrences of 0, and the other 30 filled with low digits (between 1 and 4). The result may vary slightly with different runs, but this is the most likely outcome. This is exactly the result provided when running Test 4.

3 Deadlocks and Starvation

Deadlocks

Let's assume my program is in a deadlock. First, this could not occur in the Log class. The Log class only has one lock, and each operation following the locking of the mutex is a simple, single operation. Additionally, if there is an exception, it will always be caught, and the lock will still be released. Thus, is there is a deadlock, it means it has to occur in the buffer class.

The buffer class contains no loops, and each function always releases to lock (also when there is an exception), so deadlocks can only occur in the style of the "five philosophers" problem. This is however, impossible since the only functions which need two locks – buffer.write() and buffer.set_bound() – both acquire the bound mutex first and then the buffer mutex. It is therefore easy to see that the Buffer class cannot suffer from deadlocks.

In conclusion, my program cannot suffer from deadlocks.

Starvation

Starvation cannot occur since all operations require locks before executing, so there will be no "skipping the line" as can happen in the reader-writer problem discussed in lecture 4. Starvation occurs when one operation holds a resource indefinitely. In this case, this never happens because all operations always release their resources at the end of the computation. Thus, starvation is not possible.