Assignment 2 - Report

Philipp Küppers Sofie Vos

October 2022

1 Implementation

For our buffer we use a vector of integers as described in the assignment. However a vector is not bounded. Therefore we introduce the variable `bound` to simulate a bound for all of the operations. The variable represents the size of the buffer, with -1 representing an unbounded buffer. Note that we always log the success or failure of an operation before we unlock the buffer for that operation, this is done to add the log messages in the correct order. Otherwise we would unlock the buffer, and another operation writes its message before the original message was written.

Removing and adding from the buffer is a case of the consumer-producer problem that is solved with the m_buf mutex and two if-statements: for adding check if the buffer is not full, for which we use the bound variable, and for removing check if the buffer is not empty. Otherwise we write the failure to the log. In case of adding an element to a full buffer the element will be discarded. This was easier to implement than reading back from the log if adding was a success and if not try to add the element again later.

Writing to and reading from the log is a case of the reader-writer problem. When a thread reads it reads the whole log to see what happened during the execution (useful in the tests). When a thread writes, it pushes back the success or failure such that it will never overwrite an element. Hence it will never overwrite an element read.

Race conditions occur when two or more processes try to modify data at the same time. However in our implementation we lock every critical section, such that only one process is updating or accessing shared data at the same time. Hence we guarantee mutual exclusion.

2 Tests

In general, it is nearly impossible to test every possible input and situation to be absolutely sure, that you do not suffer from any deadlocks, starvation or race conditions. However we tried to design some tests, to test as many edge cases as possible. Here are the tests we used:

run_test1(): tests for possible deadlocks or starvation when adding to and removing from a buffer
if no starvation or deadlock occurs all 100 threads will have written to the log, and that is the expected result. After running a few times our expected result was never unequal to the actual result.

run_test2(): tests if one reader thread will get a chance to read (no starvation) when 50 threads are writing.
Because reading from the empty log also counts as a read, sth should be showed when this is the case (uncomment line 55)
The reader was able to read in every run which is what was expected.

run_test3(): tests if one writer thread will get a chance to write (no starvation) when 50 threads are reading.
Because reading from the empty log also counts as a read, sth should be showed when this is the case (uncomment line 55)
The reader was able to read in every run which is what was expected.

run_test4(): tests if and equal amount of readers and writers succeeds. If a deadlock occurs, the program should not terminate.

run_test5(): This test is designed to check if the bounded buffer works. We ran this test multiple times and observed that we never had the situation where more elements were added than there should be and no elements were removed from an empty buffer.

3 Deadlocks and starvation

In general, we can say that for the functions where we lock and then unlock, without using another lock in between, we are very unlikely to get a deadlock, because the first lock can not be stuck waiting for another lock.

We know that for the buffer we have a producer-consumer situation, where we add or remove elements. Because each thread asks only one time for the lock, every thread will be able to execute and thus no starvation occurs. Also for our buffer, we do not have circular wait because we use one lock, meaning that it will be released without waiting for another lock or that same lock again.

Due to the m_log mutex being used as a turnstile, we get that when no writer attempts to enter the critical section, the readers can each pass through this mutex. Hence the writers threads will not starve due to there being multiple readers. Without this mutex the readers did not starve and with the mutex they still won't. Also in this case, there will be no circular wait because every time a thread acquires a mutex, another one that has it will unlock it.