Common Concurrency Problems

Annalise Tarhan

March 8, 2021

1 Study the code in vector-deadlock.c and main-common.c. Run vector-deadlock with the following flags. How does the output change from run to run?

The thread ordering changes from run to run, but they are never interleaved, since locks protect the critical section and the arguments are always passed in the same order, preventing deadlock.

2 Now add the -d flag, and increase the number of loops. What happens? Does the code (always) deadlock?

No, the code does not always deadlock. Even with five threads and 10,000 loops, it succeeds most of the time. The reason it deadlocks some of the time is because the arguments are passed in different orders, so sometimes one thread's source will be another's destination, and with an unlucky interleaving they will deadlock after their respective destinations' locks are acquired.

3 How does changing the number of threads change the outcome of the program? Are there any numbers of threads that ensure no deadlock occurs?

The more threads there are, the more likely deadlock will occur. The only number of threads that guarantees there will be no deadlock (when -d is used) is one.

4 Now examine the code in vector-global-order.c. Why does the code avoid deadlock? Also, why is there a special case in this vector_add() routine when the source and destination vectors are the same?

Deadlock is avoided by imposing a global order on the vectors, based on their location in (virtual) memory. The special case is necessary to prevent the thread from deadlocking with itself when the source and destination point to the same vector with the same lock.

5 Now run the code with the following flags. How long does the code take to complete? How does the total time change when you increase the number of loops, or the number of threads?

With the given flags, it takes 0.02 seconds. The time increases linearly with the number of loops, but less smoothly with the number of threads. Adding one more thread increases the time significantly, but another less so. The pattern continues, with odd numbered threads being costly and the even ones cheap.

6 What happens if you turn on the parallelism flag? How much would you expect performance to change when each thread is working on adding different vectors versus working on the same ones?

Increasing the number of loops has the same effect as without the parallelism flag, but adding more threads makes a much smaller difference, since they aren't waiting for each other to finish.

7 Now study vector-try-wait.c. Is the first call to pthread_mutex_trylock() really needed? How fast does the code run compared to the global order approach? How does the number of retries, as counted by the code, change as the number of threads increases?

The first call to trylock is a bit wasteful, since all it does is repeatedly try to acquire the lock. A more efficient version would simply call pthread_mutex_lock() instead of spinning. As is, the code is much, much slower than the global order approach, which is around five times faster. The number of retries is enormous, starting with about 10 retries per thread per loop when there are only two threads and increasing to 30 retries per thread per loop at ten threads.

8 Now look at vector-avoid-hold-and-wait.c. What is the main problem with this approach? How does its performance compare to the other versions, when running both with -p and without it?

The main problem is that it is too blunt of a solution, because it prevents other threads from operating on unrelated vectors while it holds the general lock to acquire its own vectors' locks. Compared to the global order approach, it is slower and scales poorly. The biggest difference is when -p is set and more concurrency should be possible. The global order version takes advantage of this difference and scales very well. This version doesn't, and ends up taking ten times as long as the global order version when the number of threads reaches twenty.

9 Finally, look at vector-nolock.c. This version doesn't use locks at all; does it provide the exact same semantics as the other versions? Why or why not?

This version uses a hardware instruction to achieve the same result as the others, but atomically. Performing the calculation and store atomically has the same effect as locking the code, so I would argue that the semantics are the same, even though the work is performed in hardware instead of in software.

10 Now compare its performance to the other versions, both when threads are working on the same two vectors and when each thread is working on separate vectors. How does this no-lock version perform?

For smaller numbers of threads, it is slower than the global order version, but it scales better, breaking even between seven and eight threads. When the threads are working on separate vectors, though, which is where global order really shines, the no-lock version doesn't do nearly as well.