**Deadlocks**

**Submission:**

* Deadline: Wednesday, November 8, 2023 8:00 pm HKT.
* Answer ALL questions. Add additional pages if necessary.
* Submit this answer sheet via Canvas->Assignments->Tutorials->Tutorial 5.

**Questions**

1. First let’s make sure you understand how the programs generally work, and some of the key options. Study the code in vector-deadlock.c, as well as in main-common.c and related files. Now, run ./vector-deadlock -d -n 2 -l 1 -t, which instantiates two threads (-n 2), each of which does one vector add (-l 1). Change the number of loops (-l) from 1 to higher numbers. What happens? Does the code (always) deadlock?

**Answer:**

**Yes, the -d flag causes the threads to acquire locks on the vectors in different orders, which can lead to a deadlock, and if we increase the number of loops (-l), each thread will call vector\_add() more times, increasing the likelihood of a deadlock.**

1. How does changing the number of threads (-n) change the outcome of the program? Are there any values of -n that ensure no deadlock occurs?

**Answer: Changing the number of threads (-n) affects the concurrency of the program, which can influence the likelihood of a deadlock. This is because multiple threads are trying to acquire locks on the same resources (the vectors) in different orders. However, if we have only one thread (-n 1), there will be no deadlock because a single thread can’t wait for itself to release a lock.**

1. Now examine the code in vector-global-order.c. First, make sure you understand what the code is trying to do; do you understand why the code avoids deadlock? Also, why is there a special case in this vector\_add() routine when the source and destination vectors are the same?

**Answer: In the vector\_add function, before acquiring the locks, it first compares the addresses of the source and destination vectors (v\_src and v\_dst). It then acquires the locks in a consistent order based on their addresses. This ensures that all threads always acquire the locks in the same order, regardless of the order of the vectors passed to the function, which helps prevent deadlocks. The special case where v\_src and v\_dst are the same is handled separately because there’s no need to acquire the same lock twice.**

1. Now run the code with the following flags: -t -n 2 -l 100000 -d. 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?

**Answer: With -t -n 2 -l 100000 -d flags, the code took 0.06 seconds to complete. If we increase the number of loops (-l), each thread will perform more vector additions, which will likely increase the total time. if we increase the number of threads (-n), more threads will be performing the vector additions concurrently, which could also increase the total time too. As I have tested with –t –n 20 –l 100000 –d , it took 1.54 seconds to complete, and with –t –n 2 –l 1000000 –d, it took 0.42 seconds to complete.**

1. What happens if you turn on the parallelism flag (-p)? How much would you expect performance to change when each thread is working on adding different vectors (which is what -p enables) versus working on the same ones?

**Answer: When the parallelism flag (-p) is enabled, each thread works on different vectors, as we should expecting that with –p enabled can reducing contention and potentially improving performance.**

1. Now let’s study vector-try-wait.c. First make sure you understand the code. Is the first call to pthread\_mutex\_trylock() really needed? Now run the code. How fast does it 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?

**Answer: With the same parameters above –t –n 2 –l 10000 –d, vector-try-wait.c took 0.25 seconds with 185348 retries. In general, the global order approach tends to perform better in terms of speed because it avoids the overhead of retrying. The number of retries in the vector-try-wait.c approach is likely to increase as the number of threads increases. This is because more threads are competing for the same locks, leading to more cases where a thread needs to release a lock it holds and retry because it couldn’t acquire the other lock it needed.**

1. Now let’s 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?

**Answer: The main problem with the approach in vector-avoid-hold-and-wait.c is that it uses a global lock to ensure atomicity of lock acquisition. This means that only one thread can acquire locks at a time, even if they are working on different vectors. This approach effectively serializes the lock acquisition, which can lead to significant performance degradation, especially when there are many threads and vectors. When running with the -p flag, each thread gets a different set of vectors to work on. The performance of vector-avoid-hold-and-wait.c would be worse than vector-global-order, but better than vector-try-wait. With the same parameters we tested before, -t -n 2 -l 100000 –d, vector-avoid-hold-and-wait.c took 0.08 seconds, and with –p enabled, it took 0.07 seconds. As we can know that with –p flag enabled, the performance is better.**

1. Finally, let’s 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?

**Answer: The vector-nolock.c version uses atomic fetch\_and\_add operation instead of locks. This allows more concurrency, but the final vector state can vary based on thread scheduling. It’s faster but provides weaker operation order guarantees.**

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

**Answer: With the same parameters, -t –n 2 –l 100000 –d, it took 0.75 seconds. With –p flag enabled, it only took 0.08 seconds to complete. It shows that with –p flag enabled, vector-nolock.c is slightly better than other approaches. This is because it allows more concurrency and there’s no contention for locks.**