Homework Wan Huzaifah bin Wan Azhar

Answer:



There are two possible outputs:

* First thread runs first before second thread. Both threads are able to add vector successfully.
* Second thread runs first before first thread. Both threads are able to add vector successfully.



* It is very rare for the program to deadlock
  + As it requires that pthread\_mutex\_lock(v\_dst) to run in the first thread before quickly pre-empting to pthread\_mutex\_lock(v\_src) on second thread.
  + This make it happens so that both threads are waiting for each other to release the lock so they can take it to proceed.
  + This can happen on -l 100,000



* Increasing the number of threads increase the chance of deadlock occurring.
  + Since there are more threads competing for lock.
* The only number of threads that will guarantee that no deadlock will happens is -n 1, since there is only one thread that will get and release the lock.
  + This will incur performance cost as there will be only one threads doing the job.



* Vector-global-order.c enforce global ordering of locks.
  + This means a thread must acquire lock A before getting lock B but not vice-versa.
  + This make it so that thread 0 gets both lock A and B and thread 1 will not get any lock before thread 0 released both of the lock.
* The special case lock A = lock B is so that any thread that get the same lock will not be deadlocked by itself.
  + If we look at the code, lock A must be acquired before lock B. But if lock A and lock B is the same, then the thread will try to acquire lock A and lock A, which will leads to deadlock.



* On average, it took 0.02 seconds for the code to complete.
* Increasing the number of threads increase the time it took to complete.
* Increasing the loop obviously increase the time as number of operation increased.



* Running the same code on previous exercise does not change the timing very much.
* However, increasing the number of loop with -p argument with and no -p argument shows that parallelism decrease time to complete by 1/3.
  + -l 1,000,000 with -p: average 0.15 seconds
  + -l 1,000,000 without -p: average 0.22 seconds
  + This happens because the vector can only hold 100 data, judging by VECTOR\_SIZE in vector-header.h.
  + Enabling parallelism, which uses different vector for different threads decrease time because each thread can now work on their own vector with size 100, instead of global vector.
* Increasing the number of threads with -p argument also decrease the time to complete.
  + It decreases time by almost half.
  + -n 4 with -p: average 0.05 seconds
  + -n 4 without -p: average 0.10 seconds
  + Enabling parallelism, which uses different vector for different threads decrease time because each thread can now work on their own vector with size 100, instead of global vector. This is much faster if threads are increasing.



* The first call to pthread\_mutex\_trylock is not needed as if it is replaced normal lock function, it will wait until there is a lock available so there is no need to retry.
* It runs 3x slower than vector-global-order with argument -t -n 2 -l 100,000 -d
  + On average it took 0.11 seconds to finish on vector-try-wait
  + This is understandable as the threads did not wait, but retry if it fails to get the lock, thus increasing time.
* Retries increased massively if the number of threads increase.
  + More threads try to get to lock and retry.



* The main problem with this approach is that a thread needs to get all lock at the beginning instead of when it is needed.
* On average, using -p argument, it is 2x slower than vector-global-order but much faster than vector-try-wait.
  + On average, it took 0.04 seconds to finish.
  + It is faster than try-wait because it does not have to retry each time a thread fails to get the lock.



* It uses different semantics. Specifically, fetch\_and\_add is an atomic function.
* Looking at the implementation of fetch and add, it uses assembly function to execute assembly instruction atomically.
* It also use xaddl of assembly, which is exchange-and-add.



* Vector-nolock with argument -t -n 2 -l 10,000 -d:
  + Finish on average 0.40 seconds without -p
  + Finish on average 0.05 seconds with -p
  + This happens because fetch and add is atomic instruction
    - If many threads try to add the value on only one vector (without -p), only one thread can add it because only one thread can hold the vector.
    - With -p, each threads uses their own vector, therefore fetch-and-add can be done simultaneously.