Homework 5
CS 345 Operating Systems

Members: Christy Keinsley, Courtney Arrowood, and Lauren Dickman

Chapter 31 CodingQuestions:

1. The first problem is just to implement and test a solution to the fork/join problem, as described in the text. Even though this solution is described in the text, the act of typing it in on your own is worthwhile; even Bach would rewrite Vivaldi, allowing one soon-to-be master to learn from an existing one. See fork-join.c for details. Add the call sleep(1) to the child to ensure it is working.

Code implemented.

2. Let's now generalize this a bit by investigating the rendezvous problem. The problem is as follows: you have two threads, each of which are about to enter the rendezvous point in the code. Neither should exit this part of the code before the other enters it. Consider using two semaphores for this task, and see rendezvous.c for details.

Code implemented.

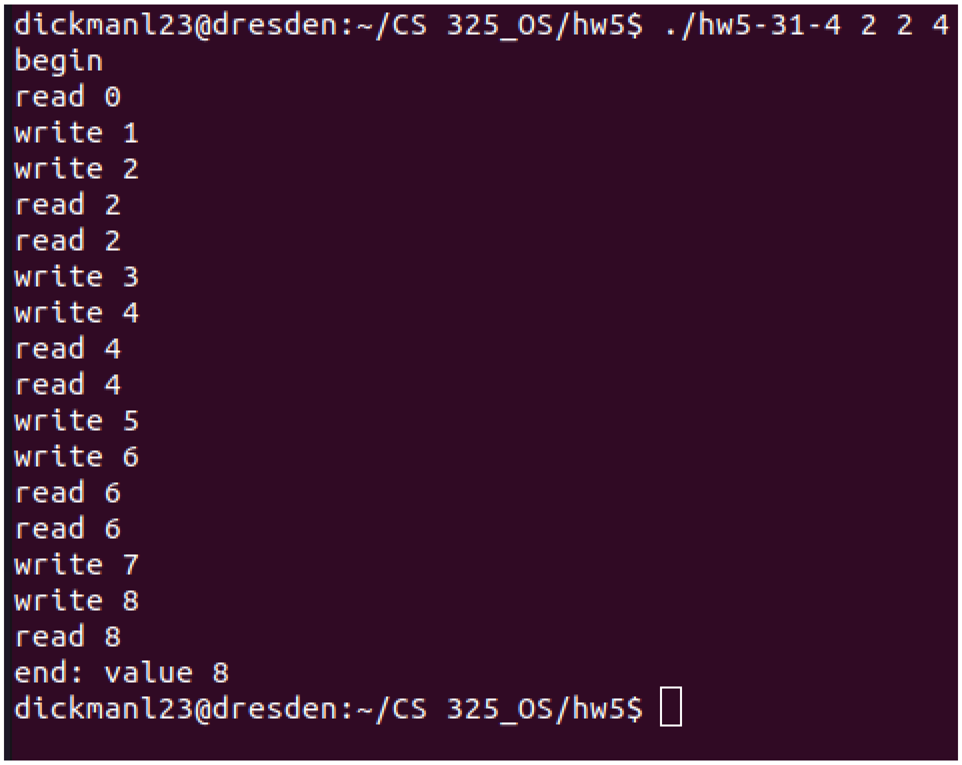
3. Now go one step further by implementing a general solution to barrier synchronization. Assume there are two points in a sequential piece of code, called P1 and P2. Putting a barrier between P1 and P2 guarantees that all threads will execute P1 before any one thread executes P2.

Your task: write the code to implement a barrier() function that can be used in this manner. It is safe to assume you know N (the total number of threads in the running program) and that all N threads will try to enter the barrier. Again, you should likely use two semaphores to achieve the solution, and some other integers to count things. See barrier.c for details.

Code implemented.

4. Now let's solve the reader-writer problem, also as described in the text. In this first take, don't worry about starvation. See the code in reader-writer.c for details. Add sleep() calls to your code to demonstrate it works as you expect. Can you show the existence of the starvation problem?

When running the program read and write appear to be printing in order from each other with there being a read 4 and a write 4 displaying on the terminal. However, we do suspect that there is code starvation within this program because there is at one point where the write value does not have a read of the same value as one of the write values. For example, we ran the command ./hw5-31-4 2 2 4 in the terminal and it displays:



```
dickmanl23@dresden:~/CS 325_OS/hw5$ ./hw5-31-4 2 2 4
begin
read 0
write 1
write 2
read 2
read 2
write 3
write 4
read 4
read 4
write 5
write 6
read 6
read 6
write 7
write 8
read 8
end: value 8
dickmanl23@dresden:~/CS 325_OS/hw5$
```

As it can be seen in the image, write 7 does not have a read 7. With this in mind we can assume that startation within the code is causing certain writes and reads to be without a matching value. There also seems to be issues with certain reads and writes having the lock for longer and not sharing with others. Overall, in this code any number of readers can be in the critical section simultaneously. To avoid starvation writers, or the writers, must have exclusive access to the critical section.

5. Let's look at the reader-writer problem again, but this time, worry about starvation. How can you ensure that all readers and writers eventually make progress? See reader-writer-nostarve.c for details.

To ensure that all readers and writers eventually make progress by creating another lock, that controls when to release the writer lock. In other words, the lock prevents a writer from entering the critical section while any other thread (reader or writer) is there, and while the writer is there, no other thread may enter.

6. Use semaphores to build a no-starve mutex, in which any thread that tries to acquire the mutex will eventually obtain it. See the code in mutex-nostarve.c for more information.

<u>Code implemented.</u>

7. Liked these problems? See Downey's free text for more just like them. And don't forget, have fun! But, you always do when you write code, no?
These problems were interesting and confusing to complete. It is satisfying to get the code to work in the way that you want after spending long periods of time coding the program.

Chapter 32 Coding 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 -n 2 -l 1 -v, which instantiates two threads (-n 2), each of which does one vector add (-l 1), and does so in verbose mode (-v). Make sure you understand the output. How does the output change from run to run?

According to the ostephomework overview, the vector-deadlock blithely grabs the locks in a particular order (dst then src). By doing so, it creates an "invitation to deadlock", as one thread might call vector_add(v1, v2) while another concurrently calls vector_add(v2, v1). This implies that the order of the threads might change.

2. Now add the -d flag, and change the number of loops (-l) from 1 to higher numbers. What happens? Does the code (always) deadlock?

In running the command ./vector-deadlock -2 3 -l 10000 -v -d, with a higher number of loops, it can be seen that the code does not always deadlock with there being a change from thread 0 to thread 1. Though, it can be seen that the code experiences deadlock very often. If there were a smaller number of loops it would take multiple times of running the same command to see a switch or change of threads being used in the program.

3. 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?

Changing the number of threads (-n) changes the outcome of the program and affects the number of times deadlock is not experienced in the code to a few times, to always experiencing deadlock with two threads accessing the vector. To avoid deadlock, would be if we made -n equal to 1.This means that only 1 thread is accessing the vector, therefore deadlock is avoided.

4. 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?

According to the overview, the vector-global-order.c is a version of vector_add() that grabs the locks in a total order, based on the address of the vector. The vector-global-order.c has a if-else statement that includes the special case that enforces the locking order by the lock address. The if-else and the special case within that if-else statement ensures that the program cannot lock the same lock twice.

5. 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?

In running the command ./vector-global-order -t -n 2 -l 100000 -d, the time is 0.07 seconds. The total time increases when the number of loops increases. While the total time decreases when the number of loops is smaller. For example, when running the command of ./vector-global-order -t -n 2 -l 10000 -d the total time is 0.01 seconds.

Changing the amount of threads has the same effect in increasing the total time. The higher number of threads the higher the total time. While the smaller number of threads used results in a lower amount in the total time. For example in running the command ./vector-global-order -t -n 3 -l 400000 -d and ./vector-global-order -t -n 4 -l 400000 -d it is noticeable that the 4 threads take longer with 1.18 seconds compared to the threads having 0.51 seconds in total time. Though we did notice that there are times that the total times of different numbers of threads match. For example, both this command ./vector-global-order -t -n 2 -l 200000 -d and this command ./vector-global-order -t -n 3 -l 100000 -d are both 0.14 seconds.

6. 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?

The -p flag gives each thread a different set of vectors to call add upon, instead of just two vectors. In using the -p flag there isn't contention for the same set of vectors. Thus, the threads don't need to wait for the same two locks. This results in the total time decreasing when adding the -p flag. For example when running the command ./vector-global-order -t -n 3 -l 400000 -d (without the -p flag) the total time is 0.51 seconds. While running the command ./vector-global-order -t -n 3 -l 400000 -d -p (with the -p flag) the total time is 0.17 seconds.

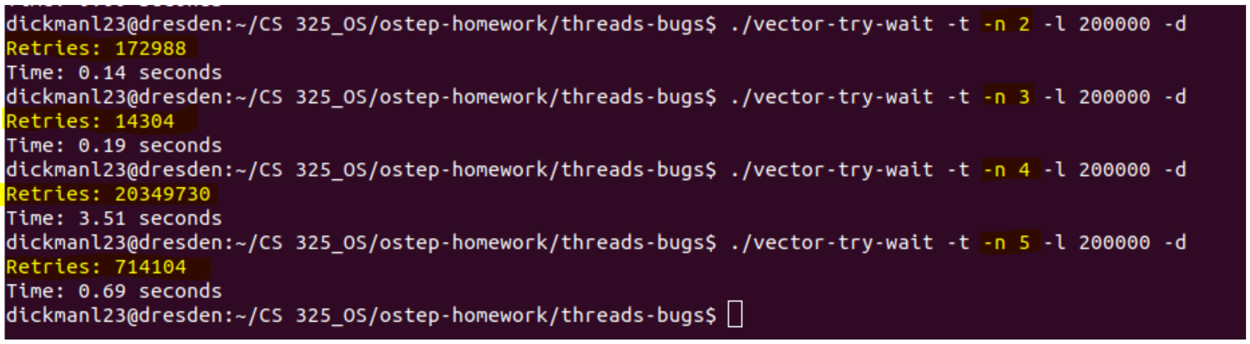
7. Now let's study vector-try-wait.c. First make sure you understand the code. Is the first call to pthread mutex trylock() really needed? Now run the code. How fast does it run compared to the global order approach? How does the number of retries, as counted by the code, change as the number of threads increases?

The vector-try-wait.c is a version of vector_add() uses pthread_mutex_trylock() to attempt to grab locks; when the try fails, the code releases any locks it may hold and goes back to the top and tries it all over again. The vector-global-order.c is faster compared to the vector-try-wait.c, though not by much.

For example, in running ./vector-try-wait -t -n 2 -l 200000 -d the total time is 0.15 seconds and in running the command ./vector-global-order -t -n 2 -l 200000 -d the total time is 0.13. The difference between the two program's time is 0.02 seconds in total. Though, oddly enough both these programs can have the same time. However, it depends if the vector-try-wait was ran only once and then running the vector-global-order to have the same total time.

There does not seem to be a need for the first pthread mutex trylock() because it is only stating that if the number of the argument is 0 then go back to the beginning of the code. The goto statement is a jump statement which is sometimes also referred to as an unconditional jump statement. The goto statement can be used to jump from anywhere to anywhere within a function. In the vector-try-wait program the top is defined at the after the intilzing of the vector_add(), in the beginning of the function.

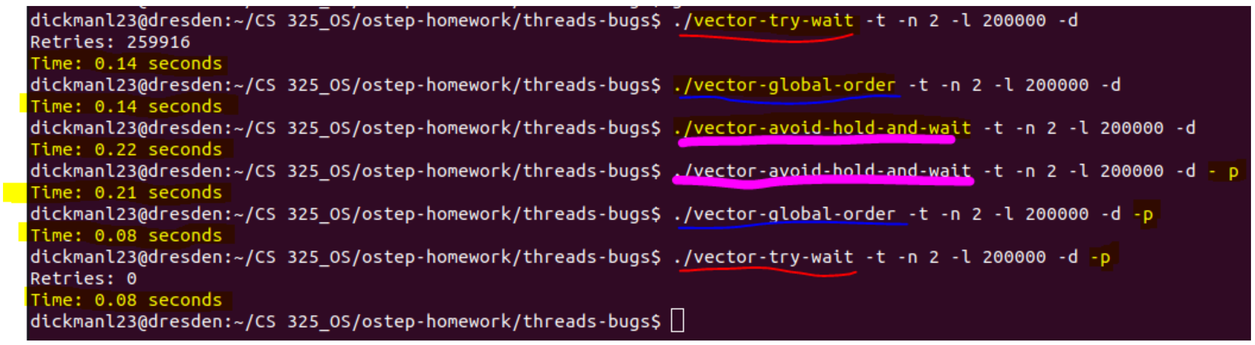
There is no ordering or exact increase in the retries when changing the number of threads. There are times that increasing the number of threads does increase the number of retries, however, that is not always the case as can be seen in the image below. When first running the command ./vector-try-wait -t -n 2 -l 200000 -d the retries are 172988, but then changing the number of threads to 3 the number of retries changes to 14304. The number of retries appear to be increasing and decreasing at random regardless of the number of threads used.



```
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-try-wait -t -n 2 -l 200000 -d
Retries: 172988
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-try-wait -t -n 3 -l 200000 -d
Retries: 14304
Time: 0.19 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-try-wait -t -n 4 -l 200000 -d
Retries: 20349730
Time: 3.51 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-try-wait -t -n 5 -l 200000 -d
Retries: 714104
Time: 0.69 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$
```

8. 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?

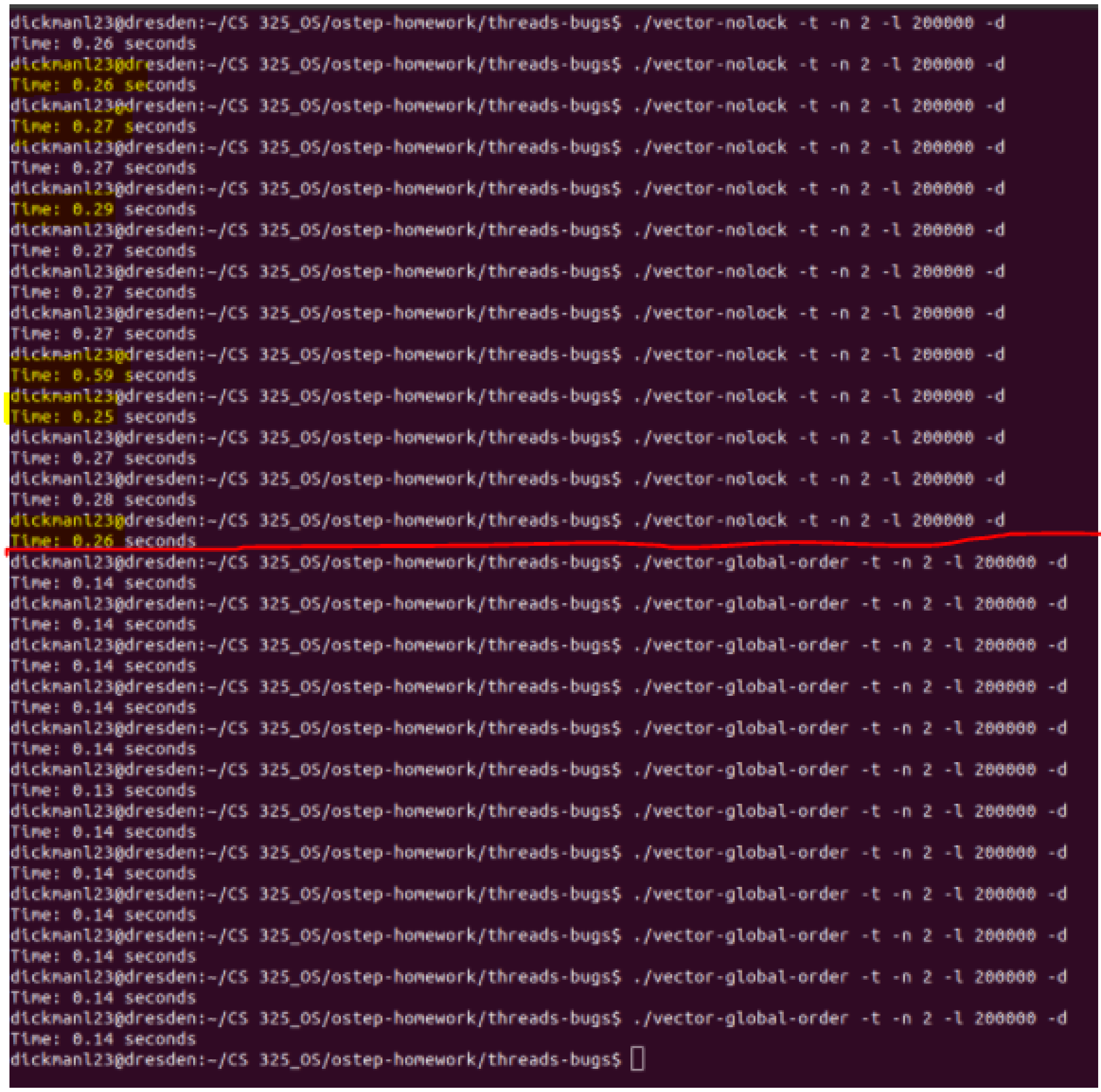
The vector-avoid-hold-and-wait.c version ensures it can't get stuck in a hold and wait pattern by using a single lock around lock acquisition. While only one thread is allowed to add at one time, this causes the performance to be the worst out of all the other versions, in terms of the total time being greater than the rest of the version. Even including the -p flag to the vector-avoid-hold-and-wait, the program is still greater in total time compared to all the other versions, as it can be seen in the image below. The vector-avoid-hold-and-wait continues to have the total time between 20 to 22, regardless of using the -p flag.



```
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-try-wait -t -n 2 -l 200000 -d
Retries: 259916
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-avoid-hold-and-wait -t -n 2 -l 200000 -d
Time: 0.22 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-avoid-hold-and-wait -t -n 2 -l 200000 -d -p
Time: 0.21 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d -p
Time: 0.08 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-try-wait -t -n 2 -l 200000 -d -p
Retries: 0
Time: 0.08 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$
```

9. 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?

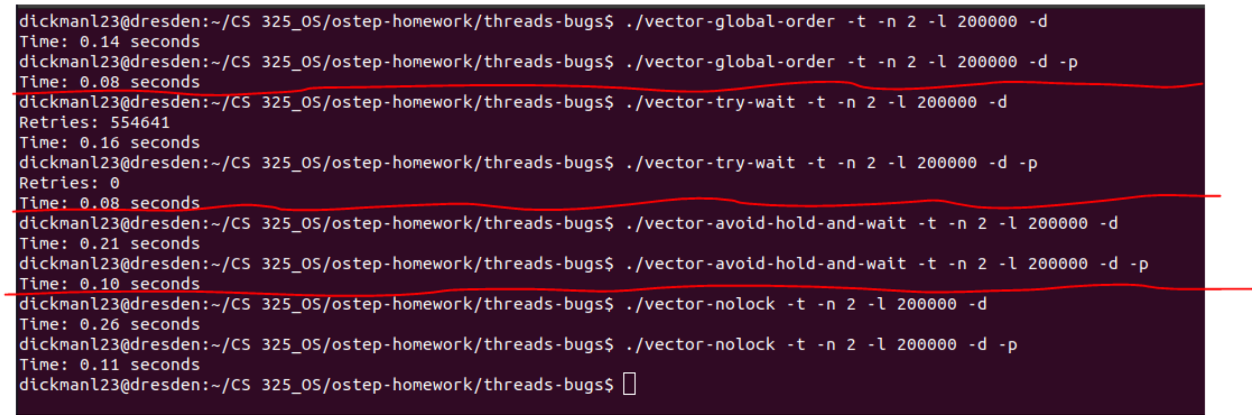
The vector-nolock.c is a version that doesn't even use locks; rather, it uses an atomic fetch-and-add to implement the vector_add() routine. Thus, it does not provide the exact same semantics as the other versions. The vector-nolock actually changes the total time when running the same command repeatedly, as seen in the image below when running ./vector-nolock -t -n 2 -l 200000 -d repeatedly. Unlike in the other versions, the total time does not change when running the same command repeatedly. In running ./vector-global-order -t -n 2 -l 200000 -d repeatedly the total time still remains at 0.14 seconds. While running ./vector-nolock -t -n 2 -l 200000 -d repeatedly the total time changes from 0.25, 0.27, to 0.56 seconds.



```
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.26 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.26 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.27 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.27 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.29 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.27 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.27 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.27 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.59 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.25 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.27 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.28 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.26 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.13 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ 
```

10. 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?

The performance of vector-nolock.c seems to be even worse compared to all the other versions, even worse than the vector-avoid-hold-and-wait version. There is also the lack of reliability with the vector-nolock because of how the values change when running the same command, while the other versions continue to have the same value regardless of common repeats. As seen in the image below, performance wise vector-nolock performs poorly compared to all other versions.



```
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d
Time: 0.14 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-global-order -t -n 2 -l 200000 -d -p
Time: 0.08 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-try-wait -t -n 2 -l 200000 -d
Retries: 554641
Time: 0.16 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-try-wait -t -n 2 -l 200000 -d -p
Retries: 0
Time: 0.08 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-avoid-hold-and-wait -t -n 2 -l 200000 -d
Time: 0.21 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-avoid-hold-and-wait -t -n 2 -l 200000 -d -p
Time: 0.10 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d
Time: 0.26 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$ ./vector-nolock -t -n 2 -l 200000 -d -p
Time: 0.11 seconds
dickmanl23@dresden:~/CS 325_OS/ostep-homework/threads-bugs$
```