Introduction

The objective of this assignment is to load balance the firewall we created in 2. Since we are using lock-free lamport queues to do this, we have to be very careful about our methods. The dispatcher thread is the only one that enqueues packets onto each lamport queue. The worker threads, however, can now dequeue packets from any worker thread instead of just their designated one in 2. As such, we need to use the locks we created in 3a.

Modules

Module 1: Our first module is going to be Serial where it creates the firewall structure that we are used to seeing in project 2. Effectively, it should feature a single worker that calculates checksums for each source serially. The only difference between this and project 2 is that there is an additional parameter: M, that basically dictates how long the experiment runs. Effectively, we can do this part by reading in the command line parameter M, and then using the stopwatch data structure to start the time at the beginning, and then in our double loop structure, keep checking the time, and if it exceeds M, then we stop the experiment.

Module 2: Our next module is dealing with parallel that effectively dictates how the parallel computation works. This part is configurable by three parameters, Queuedepth, Locktype, and Strategy. Effectively, to do this part, we need to write 12 different functions, one for each strategy and locktype combination. The LockFree part can be directly copied from homework 2, so that should be straightforward. Home Queue is pretty straightforward too, as all we need to add is that each worker grabs the associated lock with each lamport queue. Awesome is the most complicated part of this project. My design idea is to create an array, the number of workers long, and that array stores the amount of work each thread is working on. So every time,

we have the option to dequeue from a lamport queue, we are going to choose the thread that is undergoing the least current work to grab the lock, and dequeue a packet. Then, it will add the expected work to its corresponding entry in the array. Then, soon as it computes the checksum, the thread subtracts said expected work from its entry in the array. This is all happening concurrently, so each worker thread is working in sync with each other.

Module 3: Main. Module 3 is going to deal with the main functionality of this project, so we are just going to combine every function, and make it configurable by each associated parameter. We can actually use a dispatcher table to choose which function to run, and what parameters to provide it.

What data structures will the serial program use internally?

The serial program does not need any data structures. It is simply going to take in an input and just count to that. The serial implementation for this assignment should be very simple.

What additional data structures will your parallel program use internally?

Like Project 2, we need to use our lamport queue data structure. Additionally, we need an array that signifies the amount of work each thread is working on currently. I suggested a fine grained approach above, but if we do not see a speedup, we need to change it to an optimistic approach.

How will work be divided amongst the threads?

Work is going to be divided based on that array data structure. It is going to signify the amount of work each thread is doing, so each time we compute the thread working on the least

work to compute the checksum of the nex packet that is dequeued. Then after computing the checksum, it can subtract the expected amount of work from its entry in the array

What data if any must be communicated between threads?

We need to keep computing the minimum array entry between the threads, and use that entry to find the associated worker thread. This thread is going to take in the next packet that we dequeue.

What synchronization (if any) is needed to ensure correctness?

We need to be very careful with synchronization in this assignment to ensure correctness. We have to grab the associated lock before we take a packet from said lamport queue.The reason is that we do not want two threads grabbing the same packet. This strategy should ensure correctness.

What invariants must be maintained to ensure correctness?

Essentially, we have to utilize our locks to ensure correctness. The reason for this is that we need to lock access to the lamport queues, so that only one thread is obtaining the lock at a specific time. As always, we have to ensure that we have a loop structure that is constantly checking each lamport queue to see if we can dequeue anything.

Correctness Testing Plan

Once again, our hypothesis is that no invariants will be violated in the construction of this experiment. We can use the same strategy to test this as in homework 2. We are very sure at this

point that homework 2 is working well, so we can do the same strategy: create a long and add every checksum computed to that, and then printf it, and if we enter in the same parameters, we should get the same printf. To test whether we are timing out at the right time, we can also printf the time, and compare it to the input parameter.

Performance Testing Plan

Experiment 1 Hypothesis: I am expecting to see the same results, as in homework 2, with the exception of the S = HomeQueue, as it requires obtaining the lock every time, so that will take an amount of overhead, to both set up, and then acquire, and release. In the Homequeue case, I am expecting each trial to take much longer than I would have expected. The speedup decrease I am expecting to see is ~25% for all of them.

Experiment 2 Hypothesis: We are still testing uniform loads, so any bad load balancing should hurt us minimally. For S = Lockfree, I am expecting the same thing as in homework 2, as that is essentially what I did there. When S = Homequeue, however, the speedup of each trial should be hurt a bit, as we need to set up the locks, and also keep locking and unlocking them, which should add quite a bit of overhead. This overhead should constitute a smaller percentage with higher values of W such as 8000. I think the overhead should result in a slowdown of 30% when W is smaller, but decrease to about 15% with higher values of W like 8000.

Experiment 3 Hypothesis: We are finally testing exponential packets, so any bad load balancing is going to severely compromise performance. As such I am still expecting to see the same thing as in homework 2 when S = Lockfree. When S = Homequeue, however, the speedup of each trial should be hurt in a comparative way to experiment 2, as locks need to keep being acquired and released.speedup decrease, as we increase W due to there being more work, but I

am expecting to see the slope be much higher, so the speedup should be much better than the corresponding experiment in homework 2. If it isn't, that shows us that our load balancing scheme is not effective, and we need to improve it. I think the overhead plus bad load balancing should result in a slowdown of 40% when W is smaller, but decrease to about 20% with higher values of W like 8000. This is worse than last experiment due to exponential packets having worse load balancing.

Experiment 4: We are load balancing our project finally!!!! So we finally get to see the fruit of our idea. The easiest way to demonstrate that we have done this is to compare it with Experiment 3, and see whether we see a higher speedup for each W, as we have done better load balancing. We should see higher speedups particularly for later values of W such as 8000. If we do not, that means our load balancing scheme was ineffective, and we should try some kind of optimistic graining strategy. That 40-20% slowdown I am expecting to see in each of the last two experiments should go down to about 10%, as we still need to set up the data structures, but load balancing should no longer severely hurt us.