Chude Qian [95%]-[100% Live]
Francis O'Brien [5%]-[0% Live]

EECS 233 Programming Assignment 4

Basically I did the whole program set myself… ☹.

Overall:

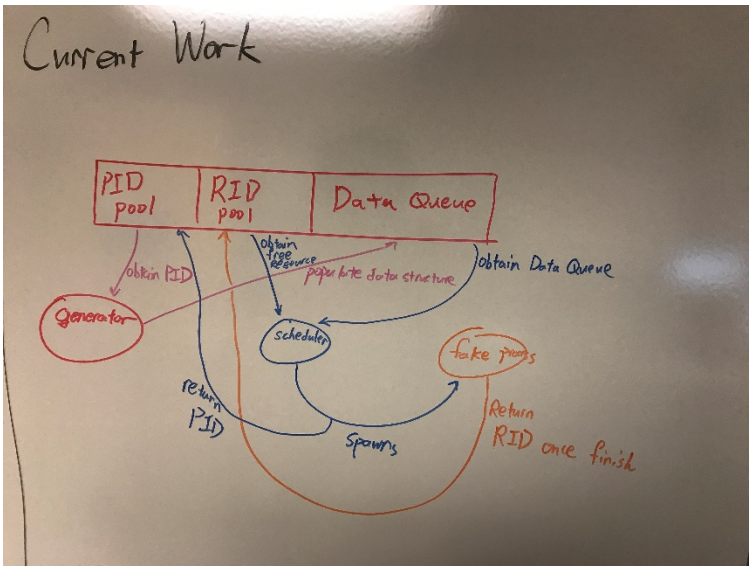
Overall, I have designed the program to be easily expandable and easily adaptable with only minor changes needed in the wrapper class for the data structure. Code Reusability is maximized.

In the assignment request, I Ire asked to implement with 4 different data structures. I folloId the instruction as Ill as I tested one custom designed data structure. More on that one later.

Architecture:

Overall, the code can be considered into several functional groups: fakePrcess, frankDS, Generator, Scheduler, and Main Runner. While I could use a master file to combine these together in one launcher file, hoIver, I feel like time is more worth it focusing on the more specific implementations. Everything other than frankDS should be the same. Again, with Java's advantage of being OOP, this could easily happen.

The overall process can be concluded in the following picture:



The common data structure has three pools. A PID pool, and RID pool, and the actual data queue. The PID pool and RID pool is basically a queue. HoIver, per request, the queue is implemented differently with the requested data structure.

The generator will dequeue a PID pool, to get a valid PID. The PID will be collected and used in the data structure. The next time the PID will get replenished is when the scheduler has spawned a fake thread. More on that later. If PID pool is exhausted, the generator will just take a nap and try again.

The resource ID pool is done similarly as the PID pool. The scheduler has access to dequeue the next available resource ID. If it gets nothing (the RID pool is empty), it will go into protective wait mode. Once it gets an RID, it will pass the RID with to the newly spawned fake ID. Once the fake ID is done sleeping, it will return the RID back to the RID pool (Enqueue).

Since the queue enqueue and dequeue operation are basically real time compared to later operation, I think this is the most efficient implementation.

As for the full Queue, this is the most common part where It is organized with priority, and look up RID on demand. The first few are always the first in ones. With all different data structure type, the implementation is basically similar that I am trying to implement a queue.

As I have stated in the beginning of the class, the generated fake process are a bit too long that this program cannot be considered will run in amortized realtime. Consider the fact that, a generated thread will have an expectation of 500ms. Parallelly, I can process 5 thread at a time. A batch size of 1e5 will take estimate about 10000 seconds, which is about 2.77778 hours. This is a lot! And this is a more ideal analysis. I do not consider any code inefficiency, any potential computer slows down. Therefore, I didn't run any result that is longer than 1e5.

In order to address that, I have scaled down the operation time to 50 ms, which will help the operation time a bit. Since this is considered a sort of scale down. Therefore I don't think there should be any different on the result plotting.

FakeProcess:

The purpose of the fake process class is basically help us unify what needs to be spawned. The scheduler will spawn the fakeprocess as a runnable, however not joined. This is a fairly interesting issue because if the scheduler joins the fake process thread, the scheduler will have to wait for the fake process to end until able to spawn a new one. It feels that the mechanisms of threading in java should be introduced more.

Once the thread is spawn, it sleeps for the given time. Once its done, it will return the resource ID as well as log when it finish.

Generation:

The generation node is responsible for generating the random resource. There are five helper function to help generate the result.

The runner function is responsible for guarding as well as overall calling. Once a sequence is allowed to generate (given by is there available PID and is there available data structure space), it then basically calls the data structure add function, which will be included later.

Scheduler:

The scheduler node is responsible for getting the scheduling correct. The void run function basically guards for completeness as well as guards for if there are resource or not, or is there any available

queue. It involkes the data structure provide next function where it tells the data structure that it is looking for a specific RID, and then the data structure will provide that back. The scheduler is also responsible for spawning new threads that just sleeps. It will pass along the datastructure pointer as well as will tell the thread which resource ID it is pretending to be.

Datastructure

The datastructure, as mentioned previously, is mainly considered as two resource pool and one queue. Although each question's implementation might be different, but eventually what they do is a queue.

The datastructure provide two critical function, that is crucial to the operation. One is provide the next task. The reason why I set the provide next inside data structure is for better code reuse. With this function, we can just change the datastructure and not worry about implementation in generation and scheduler, knowing that if written correctly, the scheduler will always get a task that it requested with resource ID, and it is the most urgent one. The method is basically checking first the highest priority, see if the specific resource ID exists or not. If not, then it looks the next prirority. This is the relatively time consuming part. But considering how long a thread executes, it is relatively realtime.

The add function basically gets the generated sequence, check its priority, and then pass it to the correct location. All of my implementation basically did something like a hashing based on priority first, as I think this is the most efficient way of organizing it. However, in java priority queue, it was handled automatically (sorted by priority).

Result:

To better illustrate, the operation are all in the logarithmic increase.

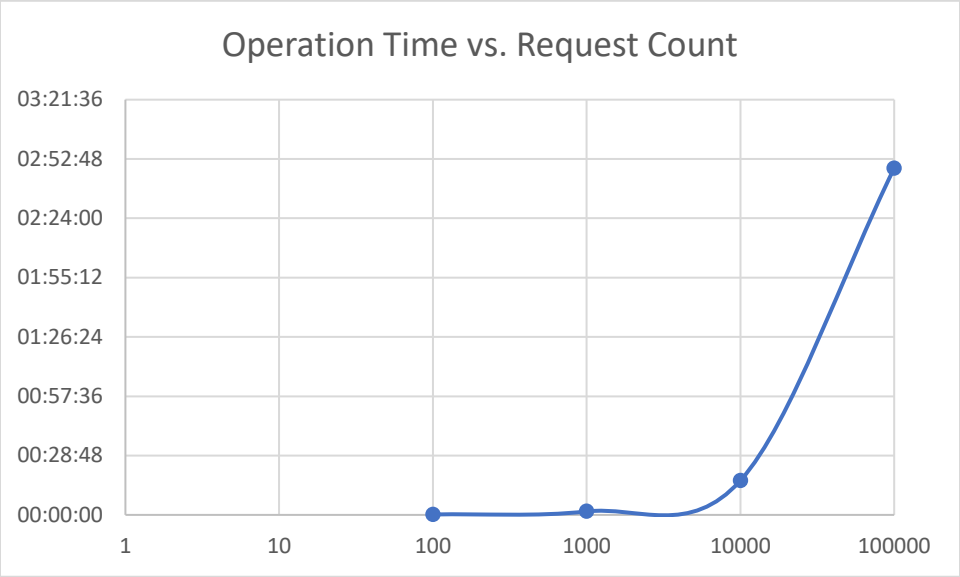


Figure 1: Custom Data Structure, Log Scale

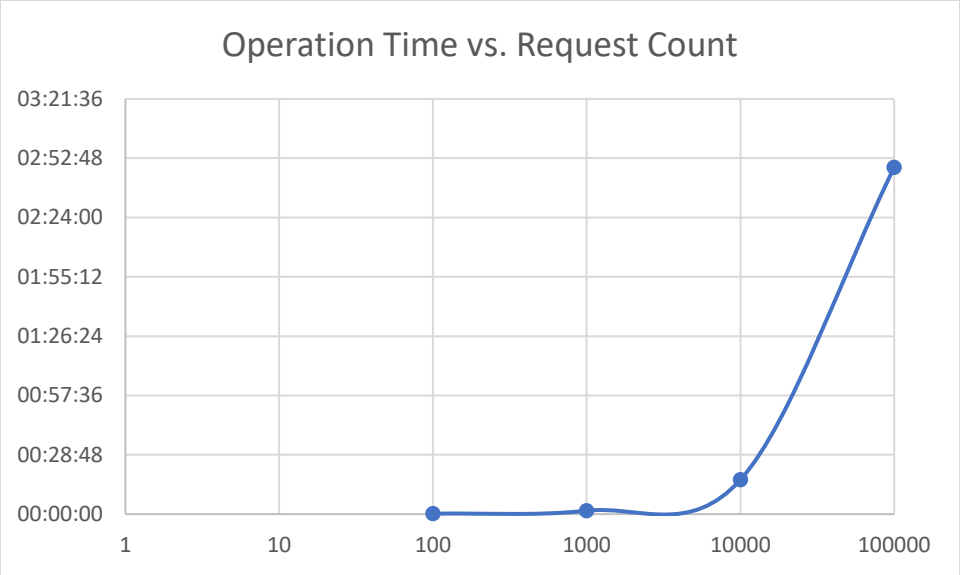


Figure 2: Hash table Implementation

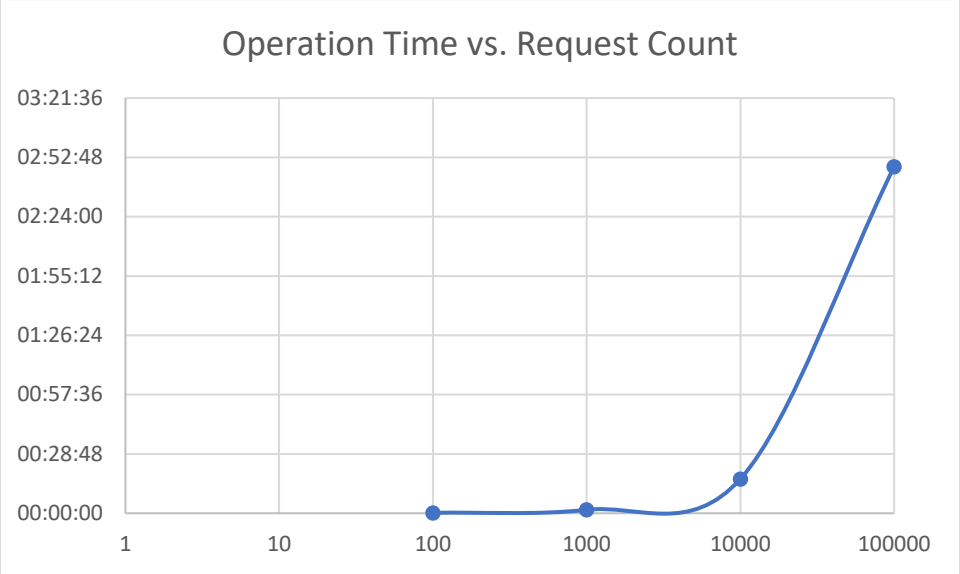


Figure 3: Linked List Implmentation

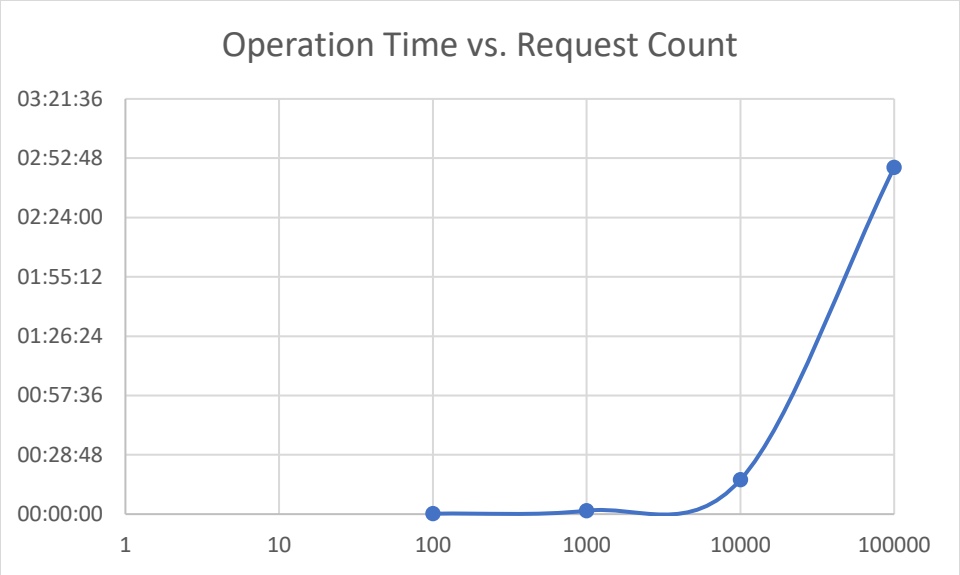


Figure 4: Sorted Array Implmentation

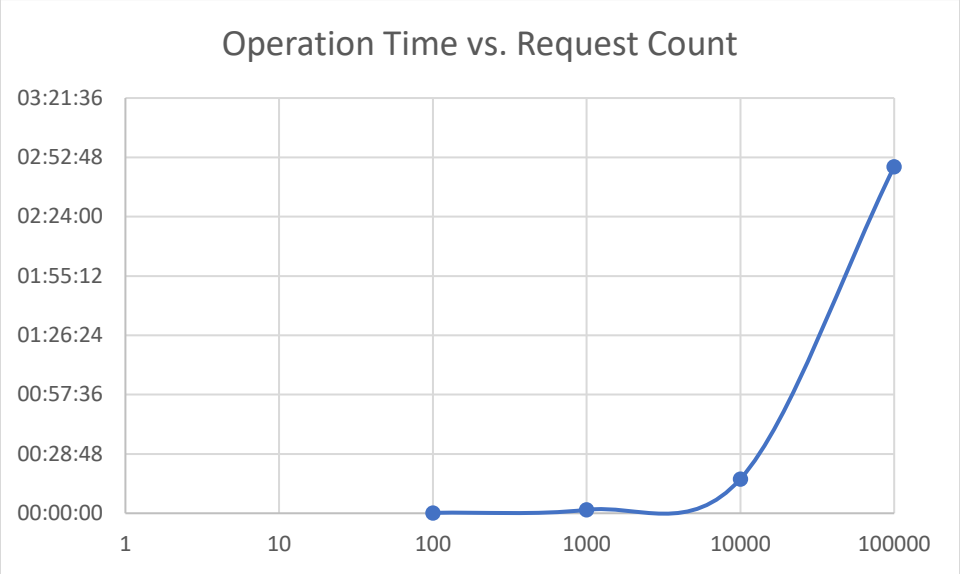


Figure 5 Priority Queue Implmentation