ACS

An airport queue simulator

Running

running `make` should create a file called `ACS` which you can run with `ACS <input file>` where `<input file>` is of the form specified in p2.pdf.

Design FAQ

- Q: How many threads are you going to use? Specify the task that you intend each thread to perform.
- A: There are only 6 threads + one for each customer which is input file dependant. The main thread handles the queues and the other 5 (from here on in referred to as clerk threads) all simulate individual clerks. The customer threads are glorified timers for when they show up to the airport.
- Q: Do the threads work independently? Or, is there an overall “controller” thread?
- A: The main thread can be thought of as a controlling thread whereas the clerk threads loosely follow an actor model. The customer threads again, just act as timers. The main thread controls the actors by putting customers into the queue as soon as the customer thread times out and joins with the main thread. After adding the customer notifying the clerks that someone has entered the queue. The clerks then handle the customer and go back to sleep. The clerks are not pure actors however as they do record keeping on how many customers they have served as well as how long the customers they did serve have been in line. This information is then collected and aggregated after the simulation has run its course by the main thread.
- Q: How many mutexes are you going to use? Specify the operation that each mutex will guard.
- A: There is only a single mutex. It guards the read and write end of both the economy queue and the business queue.
- Q: Will the main thread be idle? If not, what will it be doing?
- A: Sometimes! Despite handling the queues the main thread does wait when there are no customers entering the queues at the moment via trying to join with the customer threads.
- Q: How are you going to represent customers? what type of data structure will you use?
- A: The customers are a simple struct of their arrival time, id, class and service_time.
- Q: How are you going to ensure that data structures in your program will not be modified concurrently?
- A: there are 2 mutable things that are shared across threads. Everything else is immutable. The first shared mutable thing (and the most important) is the queue structure and its indexes. This is guarded by a mutex it carries with it. The second thing shared across thread bounties are the clerks' info. This contains a reference to the shard queue (who's safety has already been discussed), the clerk_id (which is immutable and set before the thread is created), and some record keeping: wait_time_ledger and performance_numbers. These are initialized before the threads are created then "moved" into the worker threads where they are modified. After the main thread has joined all the workers these values are safe to read and modify again as there is no longer any concurrency to cause errors.
- Q: How many convars are you going to use?
- A: One!
    - a)
        - Q: Describe the condition that the convar will represent.
        - A: Someone has entered one of the queues.
    - b)
        - Q: Which mutex is associated with the convar? Why?
        - A: The only mutex! Because it just makes sense the mutex that guards the queue should be associated with the condvar that represents queue mutation
    - c)
        - Q: What operation should be performed once pthread cond wait() has been unblocked and re-acquired the mutex?
        - A: The clerk threads all look for work to do. They first check the business line, and if they find a customer they service them. If they don't find one they go to the economy line and try to service someone there, if the work has already been stolen by the time the thread obtains the lock to check the line for work, they go back to sleep waiting for the next customer to join the line.
- Q: Briefly sketch the overall algorithm you will use
- A: The program starts by sorting the customers by arrival time and creating a thread per customer that has a timer, the main thread then joins with each customer thread in turn, immediately sending it to the SyncQueues after joining. The SyncQueues struct should be thought of as a single producer multiple consumer channel. This is inaccurate in that it actually holds two message queues buts it's close enough. The actors (clerks) take messages (customers) off the channels (queues) and processes them (printing and sleeping). The clerks know they are done when the main thread has put all the customers in and all the remaining customers are being handled elsewhere. The part that breaks my nice pure actor model is that I need to accumulate stats on wait time and such. this is done by the clerks and aggregated after the simulation by the main thread. Which the main thead then prints.