Giorgio’s Discrete Event Simulator

Readme.txt

This program simulates an Operating System’s event handler, where jobs arrive and must be redirected to CPU, or disks to perform IO operations.

To achieve this, I’ve used a priority queue in which events are stored by time to simulate job arrivals, a FIFO queue for jobs waiting to be executed by the CPU and another two FIFO queues for jobs waiting to execute IO operations in the disk.

In the beginning three jobs are inserted to the priority queue. The first one signalizes the start of the simulation, the second is job1, the first job to be sent to the CPU, and a third one with time 10000 that once popped from the priority queue indicates that the simulation is over.

A while loop is responsible for keeping the simulation running until the time 10000 is reached, or until the priority queue is empty (which should never happen).

Once a job is popped from the priority queue, it is analyzed and depending on its type is redirected to the proper handler function. Jobs of type ARRIVAL and FINISH are sent to the process\_cpu function, and types DISK\_ARRIVAL, DISK\_I\_FINISH, and DISK\_II\_FINISH are sent to process\_disk function. Here I’ve used enums for code readability. At this time, the current event’s time is used to set the overall time.

The process\_cpu function first checks the event type. If it is a CPU\_ARRIVAL, it creates a new job with time determined by the current time plus a random number from ARRIVAL\_MIN and ARRIVAL\_MAX. It inserts it in the priority queue and inserts the current job in the CPU queue. If event is of type FINISHED, it sets the CPU to idle, indicating that the current job is done at CPU, it calls the quitProb function to determine the probability that the current job would exit the simulation. If it doesn’t exit, the job is sent to the priority queue as a DISK\_ARRIVAL type, and the event time set as current time. If it exits, it is sent to a done vector, which is checked in a later time time to make sure that jobs of that jobs sequence are not included in the simulation anymore. The process\_cpu function then checks if the CPU queue is not empty and if the CPU is idle. If both conditions are true, it pops a job from the CPU queue, sets its type to FINIHSED and its time to current time + a random number between CPU\_MIN and CPU\_MAX to illustrate time taken to be processed by CPU. CPU is set to busy.

The process\_disk function checks for the current job type. If it is a DISK\_ARRIVAL, it checks which disk queue is emptier and pushes the event to this queue. The function then, sets the job type to DISK(I orII)\_FINISH, with times of current time + a random number between DISK\_MIN and DISK\_MAX, and inserts it to the priority queue. At this time disk 1 or 2 is set to busy. If the job type is of DISK\_FINISH it finds the job in the disk queue, pops it, sets its type to CPU\_ARRIVAL and pushes it to the CPU queue. At this time disk 1 or 2 is set to be idle again.

The values of INIT\_TIME, FIN\_TIME, QUIT\_PROB, ARRIVE\_MIN, ARRIVE\_MAX, SEED, CPU\_MIN, CPU\_MAX, DISK\_I\_MIN, DISK\_I\_MAX, DISK\_II\_MIN and DISK\_II\_MAX, are read from the file textConfig.txt, stored in a vector, and assigned the respective variables.

The results are printed to the file output.txt. The results include the simulation, number of jobs created, throughput, max size reached by all queues and also the time spent in each task.

For testing I used a technique called “Unity testing” where the code is tested constantly in small units. Every functionality written was tested in isolation before it was added to the code. During this part of the test I used edge cases to make sure the code would still function. After the functionality is added to the main code, I test it again, with the same edge cases to make sure it would work entirelly.