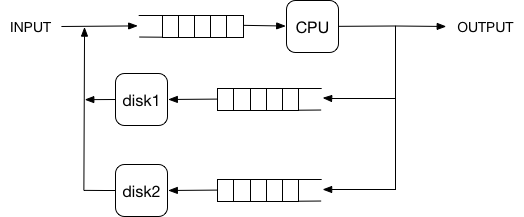
Cameron Zach

Lab 1

This program simulates a computer scheduling processes and moving them between the CPU and the two available disks. The operating parameters can be modified in the config file. Values should be separated from the parameter name by a single space. If any parameters are missing, or the file is not formatted correctly, the program may not function correctly.

This program operates with a few basic data structures. The first is a priority queue, which is the central component of the whole program. This queue is made of a series of event structs, which each have a location, an event type, a process ID, and an event time. The priority queue is sorted based on event time, so the next event to occur will always be easily accessible. The program will run a loop continuously pulling the top element from the priority queue and handling it appropriately.



There are three devices where an event can be sent: the CPU, and two disks. All three of these devices have a similar internal structure: They have a queue that stores events in FIFO order, and they are accompanied by a collection of functions to help them operate.

Firstly, the config file is read. All values are stored into the appropriate variables, which define the operating parameters for the simulation.

We set up the simulator by adding events representing the arrival of the first job, and the end of the simulation. When an event is pulled off the queue, it is sent to the handler of the proper device. From there, one of these things happens.

CPU:

An event arriving at the CPU will be moved into the CPU’s internal queue. We log the arrival of the event, noting the process ID and current time. Then, if it is the first time this job has arrived, we need to add a new job to the priority queue. We can check if it is the first time by checking if the process ID of the event is the same as the job index counter. This counter is incremented by one every time a job is created, so its value should always be equal to the newest process ID. When we have added the event to the CPU’s internal queue, we then should check to see if the CPU is working or not. If it is, we tell it to start work.

When we start work at the CPU, we simply set the internal state of the CPU to processing. We can leave the job in the queue for now. We also set the event’s “startTime” property to the current time for logging purposes. We can also log the response time for this event by checking the event’s time and comparing it to the current time. The event’s time will be the time it was added to the CPU’s queue, so subtracting that from the current time gives the response time. We then generate a “job finished” event to tell the CPU when to stop processing the current job.

When we finish a job, we first remove the job from the CPU’s internal queue. We can now log its processing time by subtracting the start time from the current time. We then decide randomly whether to send this job to disk. If we decide to do so, we create a new arrival event in the priority queue, and set the location to Disk 1. We won’t worry about which disk it should go to until later. Now that we’re done with the current job, we can free it from memory.

Disk:

Since all arrival jobs arrive at Disk 1, we need to first check if it’s a newly arriving job or if it’s a job about to be completed. If it is a new job, we determine which disk has a shorter queue, and send it there. Otherwise, we just let it go to the disk it is labelled for. We update the event to reflect its proper destination.

When a job arrives at a disk, we follow a process identical to the CPU, except we don’t worry about generating a new job in the priority queue when it arrives.

When a job finishes, we again follow a process identical to the CPU, except we don’t worry about randomly sending it to disk. Instead, we send it directly back to the CPU. We again free the event from memory.

When the priority queue finally reaches the “end simulation” event, we terminate the main program loop. Then we can run our statistics function and save that out to a statistics file. The log file is saved as a .csv so that it can be easily viewed and sorted to examine subsets of data. The statistics are simply saved in a .txt.

In testing, I used a start time of 0 and an end time of 100,000, to ensure that I would have many events run. I set the CPU time between 10 and 15 units of time, since it should operate very rapidly. I set the interarrival time between 100 and 200 units of time, since new jobs will be created slower than they are processed. The disk times are set between 500 and 800 units for disk 1, and between 600 and 900 for disk 2, since not all disks are the same speed. Additionally, they are much larger than the CPU time because a disk is extremely slow compared to a CPU.

After testing with these parameters, I found a CPU utilization of around 11%, which seemed to reflect what I measure on my own computer. Because the quit probability was set to 20%, this simulation was a heavily I/O bound process, like what may be seen in a web server. Accordingly, my disk usage was up near 99%. The average queue sizes for the CPU and the Disks reflected this as well.