CIS 3207 Assignment 1
Giorgio's Discrete Event Simulator

Due: February 6, by the end of the day

description

The programs that we execute require different services from our machines at different times: sometimes they're crunching numbers using the CPU. Other times, they read from a disk or network device. Still other times they're doing nothing but waiting for something to happen, for example: a disk read to finish, a packet to arrive from a network device, a user to click on something with a mouse, or simply waiting for some other running program to finish using some needed component. In this assignment, you'll create a discrete event simulation in C that models the very simplified machine pictured.

The purpose of the assignment is to gain some understanding of how processes move through a computing system and to put into practice some things that you've learned in 2107 and 2168, but might not have had a chance to use in a while.

When a job arrives (perhaps because a user double-clicked on an icon to start a program), it needs to spend some amount of time executing using the CPU. It then either performs I/O on one of two disks and returns to use the CPU, or the job is finished. Jobs alternate between the CPU and I/O devices in this manner until they're finished.

It's possible that when a job arrives to use a device, the device is busy servicing another program. In this case, the job must wait in a queue until the device is no longer busy. In this assignment, all of your device queues are FIFO, however, this is not necessarily the case in real systems.

For this assignment, we'll frequently note or record the time; however, we'll use a "logical clock" and think of things in terms that sound like, "at time 37, job 6 spent 5 units of time at the CPU" rather than "at 2:30:01 PM, job 6 spent 300 microseconds at the CPU".

In your simulation, the amount of time that a job spends using the CPU, whether or not it's finished (has more computing), which disk contains the file a job needs, and many other properties are determined at runtime, at random. To generate random numbers in C, take a look at the Standard C Library functions `rand( )` and `srand( )` (so that your program doesn't use the same pseudo-random numbers in each run).

Basic Mechanism

The system behaves as follows:

- The system runs from `INIT_TIME` (usually 0) to `FIN_TIME`.
- Jobs enter the system with an interarrival time that is uniformly distributed between `ARRIVE_MIN` and `ARRIVE_MAX`.
- Once a job has finished a round of processing at the CPU, the probability that it exits the system (instead of doing a disk read and then further computation) is `QUIT_PROB`.
- When a job needs to do disk I/O, it uses the disk that's the least busy, *i.e.*, the disk whose queue is the shortest. (This might seem a bit silly, but we can pretend that each disk has the same information.) If the disk queues are of equal length, choose one of the disks at random.
- When jobs reach some component (CPU, disk1, or disk2), if that component is free, the job begins service there immediately. If, on the other hand, that component is busy servicing someone else, the job must wait in that component's queue.

- The queue for each system component is FIFO.
- When a job reaches a component (a different job leaves a component or the component is free, and this job is first in the queue), how much time does it spend using the component? This is determined randomly, at runtime, for every component arrival. A job is serviced by a component for an interval of time that is uniformly distributed between some minimum and maximum defined for that component. For example, you'll define: `CPU_MIN`, `CPU_MAX`, `DISK1_MIN`, `DISK1_MAX`, etc.
- At time `FIN_TIME`, the job simulation terminates. We can ignore the jobs that might be left receiving service or waiting in queue.

Implementation

This might all seem to make sense in theory, but how is it implemented? Your program will have 4 non-trivial data structures: one FIFO queue for each component (CPU, disk1, disk2), and a priority queue used to store events, where an event might be something like "a new job entered the system", "a disk read finished", "a job finished at the CPU", etc. Events should be removed from the priority queue and processed based on the time that the event occurred. This can be implemented as a sorted list or minheap.

When your program begins, after some initialization steps, it will add to the priority queue the arrival time of the first job and the time the simulation should finish. Priority in the queue is determined by time of the event. Suppose that these are times 0 and 100000 respectively. The priority queue would look like:

when	what
0	job1 arrives
100000	simulation finished

Until we're finished, we start to remove and process events from the priority event queue. First, we remove the event `job1 arrives` because it has the lowest time (and, therefore, the highest priority). In order to process this `JOB_ARRIVAL_EVENT` we:

1. set the current time to 0, i.e., the time of the event we just removed from the queue
2. determine the arrival time for the next job to enter the system and add it as an event to the priority queue
3. send job1 to the CPU

To determine the time of the next arrival, we generate a random integer between `ARRIVE_MIN` and `ARRIVE_MAX` and add it to the current time. This will be the time of the 2nd arrival. Suppose that we end up with 15.

We also need to add job1 to the CPU. Because the CPU is idle, it can start work on job1 right away. We add another event to the event queue. This is the time at which job1 will finish at the CPU. To do this, we generate a random integer between `CPU_MIN` and `CPU_MAX` and add it to the current time. Suppose that we end up with a CPU time of 25.

The event queue is now:

when	what
15	job2 arrives
25	job1 finishes at CPU
100000	simulation finished

We proceed the same way until we remove a `SIMULATION_FINISHED` event from the queue. The main loop of your program will then be something like:

```
while the priority queue is not empty (it shouldn't ever be)
   and the simulation isn't finished:
     read an event from the queue
     handle the event
```

Each event in the event queue will be of a given type. It seems simplest to store these as named constants, e.g., something like `CPU_FINISHED=1`, `DISK1_FINISHED=2`, etc., and to write a handler function for each possible event type. We've just described what the handler for a job arrival event might do. What should a disk finished handler do? Remove the job from the disk, return the job to the CPU (determining how much CPU time it will need for the next CPU execution, seeing if the CPU is free and if not, add it to the CPU's queue just as we did previously). We should also look at the disk's queue. If the disk's queue isn't empty, we need to remove a job from its queue, determine how long the job will require using the disk, create a new "disk finished" event and add it to the event queue.

Running Your Simulator

The program will read from a text config file the following values:

- a `SEED` for the random number generator
- `INIT_TIME`
- `FIN_TIME`
- `ARRIVE_MIN`
- `ARRIVE_MAX`
- `QUIT_PROB`
- `CPU_MIN`
- `CPU_MAX`
- `DISK1_MIN`
- `DISK1_MAX`
- `DISK2_MIN`
- `DISK2_MAX`

The format of the file is up to you, but it could be something as simple as:

```
INIT_TIME 0
FIN_TIME 10000

...
```

Results

log

Your program should write to a log file the values of each of the constants listed above as well as each significant event (*e.g.*, arrival of a new job into the system, the completion of a job at a component, the termination of the simulation, along with the time of the event)

statistics

Calculate and print:

- The average and the maximum size of each queue.
- The utilization of each server (component). This would be: time_the_server_is_busy/total_time where total_time = FIN_TIME-INIT_TIME.
- The average and maximum response time of each server (response time will be the difference in time between the job arrival at a server and the completion of the job at the server)
- The throughput (number of jobs completed per unit of time) of each server.

Run the program a number of times with different values for the parameters and random seed. Examine how the utilizations relate to queue sizes. If for a given choice of parameters by changing the random seed we obtain utilization and size values that are stable (*i.e.*, they do don't change much (maybe 10%)), then we have a good simulation.

Your program should process a reasonable number of jobs, at least one thousand.

As part of the homework submit a document, README.txt, from two to three double-spaced pages plus (may be) diagrams, describing your program. Your description as addressed to a technical manager whom you want to understand what you have done, the alternatives that you had, why you made the choices you made, and how you tested your program.

Include also a second document, RUNS.txt, describing the data you have used to test your program and what you have learned from it. You should choose reasonable values for the interarrival times and for the CPU service times. For simplicity, choose a `QUIT_PROB` defaulted to 0.2, and use a service time at the disks equal to the service time of real disks. If you can, determine what is the smallest reasonable interarrival time (and for that matter, how would we even go about deciding on what reasonable would be)?