Lab 2 – CPU Scheduling Discrete Event Simulation

Due: 9:55 AM, Thursday, February 24, 2005

This week's lab consists of questions to look at on your own (not to be turned in) and a significant programming task. The program for this lab is considered a "team program" (groups of size 2 or 3 permitted) for honor code purposes. Collaboration within a group is, of course, unrestricted. You may discuss the program with members of other groups, but what you turn in must be your own group's work. Groups must be formed no later than 4:00 PM, Friday, February 18, 2005, and be confirmed by all group members by electronic mail to *terescoj@cs.williams.edu*. All group members will be assigned the same grade for the lab.

Warning: Significant design and programming effort will be necessary to complete this lab. You may use any programming language you wish. If you are not a C expert, you may choose to use this as an opportunity to become an expert or maybe you will think better of it and become a C expert when we write the Unix shell. The case studies and writeup are worth 25% of the lab grade, so keep that in mind as you plan your time.

Practice Questions

You do not need to turn in answers to these questions. But they could certainly form the basis for potential exam questions.

SG&G 4.4, 5.1, 5.4, 5.5, 5.6, 5.8, 5.10

Discrete Event Simulation

Discrete event simulation is a method used to model real world systems that can be decomposed into a series of logical *events* that happen at specific times. The main restriction on the system is that an event cannot affect the outcome of a prior event. When an event is generated, it is assigned a *timestamp*, and is stored in an *event queue* (actually, a priority queue). A *logical clock* is maintained to represent the current simulation time. At any point in the simulation, the next event to take place is at the head of the event queue. Since nothing of interest can happen between the present simulation time and the timestamp of the event at the head of the event queue, removal of an event for processing allows the logical clock to be incremented to that event's timestamp.

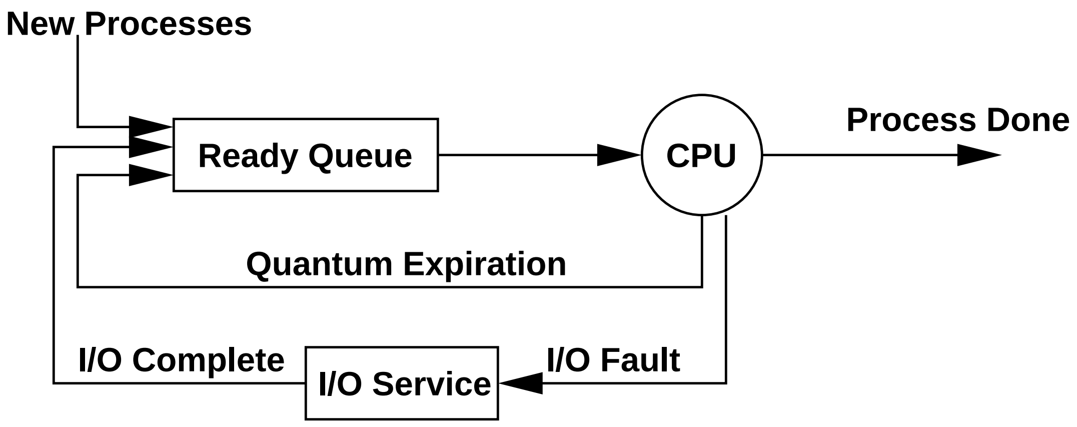
Multiple events may have the same timestamp. This models events that happen concurrently. Even though the events are processed sequentially by the simulator, they occur at the same logical time.

We use this model to simulate a simple operating system. Processes arrive in the system and take turns on the CPU, possibly spending some time waiting for I/O service. Each process remains in the system until its predetermined computational needs are met. There are a small number of events that can occur that will affect the system, and it is these events that drive the simulation.

System Description

You are to design and implement a discrete event simulator to model a CPU scheduler. You will then use your simulator to conduct studies of the performance of round robin (RR) and related scheduling algorithms.

The basic version of your program should implement a queueing system, as shown here:



Processes enter the system and wait for their turn on the CPU. They run on the CPU, possibly being pre-empted by the scheduler or for I/O service, until they have spent their entire service time on the CPU, at which point they leave the system.

- **Time:** A logical clock is used to coordinate all events in the system; the "ticks" of the clock are measured in (arbitrary) "time units." The total simulation time is a parameter supplied by the user.
- **Ready queue:** The ready queue is FIFO (first-in, first-out). A process is selected from the front of the ready queue when the CPU becomes available. The CPU becomes available when a process leaves the CPU because it has been completed, because it has been pre-empted, or because it requires I/O service. If the ready queue is empty, a process entering the ready state will be assigned to the CPU immediately.
- **New process entry:** The time between the arrival of new processes entering the system's ready queue is a random number of ticks; these random numbers have an exponential distribution about a user-supplied mean value. Each time a process enters the system, an event is generated that will result in the creation of the next process to enter the system.
- **Process CPU requirements:** The amount of CPU service time (total number of ticks elapsed while the process is in the CPU) required by the process will be a random number of ticks selected at the time the process first enters the system. This random value has an exponential distribution whose average is a user-supplied parameter.

- **Quantum:** Each process can spend up to, but not more than, one quantum in the CPU before it is switched out. The quantum is a system wide constant value, entered as a parameter by the user.
- **Removal of processes from the CPU:** When a process is assigned to the CPU, an event should be generated that will remove it from the CPU at a later time.
  1. If the time remaining for the process to complete is less than the time to the next I/O fault or the quantum, an event is generated that will cause the process to leave the system. Performance statistics for the process should be collected when this event is processed.
  2. Else, if the time remaining for the process to I/O fault (see next item) is less than the quantum, an event is generated that will cause the process will leave for I/O service.
  3. Else the process' quantum will expire. An event is generated that will return the process to the ready queue.

  Context switch time is entered as a parameter by the user and should be accounted for.
- **I/O faults:** The process executes for a set number of clock cycles between each I/O fault. This number is constant on a per-process basis - it is determined for each process when it is created, and remains the same for the duration of the process. It will be a uniformly distributed random number whose average is $\frac{1}{2}$ the time needed to run the process.
- **I/O Service time:** The time needed to service an I/O fault is constant throughout the system. This is a user-supplied parameter. Assume that the I/O subsystem can handle an infinite number of requests at the same time with no loss of turnaround time. When each process enters I/O service, an event is created to take place at the time of the service's completion.

User-supplied parameters

- mean inter-arrival time between processes (`-a` flag)
- mean CPU time per process (`-c` flag)
- constant length of the CPU quantum (`-q` flag)
- constant context switch time (`-s` flag), defaults to 0
- constant I/O service time (`-i` flag)
- total simulation time (`-t` flag)
- whether to "shut off" I/O faults (`-n` flag)

Statistics to Gather

- value of system clock
- number of processes completed
- throughput (processes completed per unit time)
- ready queue length
- event queue size (this is a simulation artifact rather than one of the system being simulated)
- turnaround times:
  - turnaround time for the average process
  - turnaround time for the longest process
  - turnaround time for the last process
- CPU utilization

Implementation Notes

- You may use any programming language, as long as it can be compiled and run on (at least one of) the Williams CS Lab's x86/FreeBSD, x86/Linux, Sparc/Solaris or Macintosh systems.
- If you work in a group and want to be able to work concurrently, consider requesting a Unix group for your files and consider keeping them in a CVS repository for source code control. Whatever you do, please do not share passwords or create world-readable (or worse, world-writeable) directories for your work on this lab.
- A graphical user interface is not required, but you are welcome to provide one. Remember to design your simulator to facilitate the gathering of statistics for the studies (scripts are good, copy and paste is bad). Thus, you may wish to include an option to run without a GUI even if you provide one.
- The directory `/home/faculty/terescoj/shared/cs432/labs/lab2` contains some files you may find useful. These include a sample `Makefile`, C source code and header files for functions that generate uniform and random distributions and do screen management. Also, a working executable, compiled for the Williams CSLab FreeBSD systems, is available in the file `sim` in that directory. You should be able to run the executable, but you will not be able to copy it.
- If you choose to program the simulation in C, you may find the `getopt(3)` library function useful for parsing your command-line parameters.

Simulation Case Studies

Conduct two (three if you are in a group with three members) studies of your own choosing using the simulator. These should involve comparisons of statistics such as CPU utilization, turnaround times, and queue lengths, as system parameters are varied. You are encouraged to e-mail me with your ideas to make sure they are appropriate before you go too far.

The most meaningful statistics are collected after the system stabilizes, that is, after the system has been under "load" for a while. The first processes to arrive enter an unloaded system and their behavior may not be typical of long-term trends. Wait until hundreds of processes have entered the system before gathering statistics, or run your simulation long enough that the behavior of these early processes will not have a significant effect on the long-term trends you are studying. Choose parameters that will result in several thousand (or more) processes passing through the system before the simulation ends. Some combinations of parameters produce mostly meaningless results, such as when processes arrive much more quickly than the CPU can process them. Avoid these situations in your studies.

If your simulator is not finished in time to use it for this part of the lab, you may generate your results using mine.

You should submit a writeup that includes a description of each of your two (or three) studies. Your discussion of each study should include what you expected would happen and what actually happened, conclusions you can draw from what happened, a critique of the model, and how your studies might apply to a "real world" situation. Include graphs to show trends as the key parameter(s) change.

Include this writeup in your lab submission as a single postscript or PDF file `lab2-studies.ps` or `lab2-studies.pdf`.

Since real computer scientists don't use Office, you are encouraged but not required to generate your writeup using $\LaTeX$. You might also want to consider using `gnuplot` to generate your graphs. Both packages are available on the CSLab FreeBSD systems. There is a sample $\LaTeX$ document in `/home/faculty/terescoj/shared/latexexample`.

Submission and Evaluation

- Submit documented source code, with a brief `README` file that describes how to build and run your simulator. All necessary files should be submitted using `turnin` as a single "tar" file, `lab2.tar`. Include a `Makefile` to allow easy compilation. Don't forget to include your postscript or PDF file that describes your simulation studies. Make sure that all group members' names appear in all files. Only one group member should submit.
- The simulation software and its documentation is worth 30 points. Correctness, design, documentation, style, and efficiency will be considered when assigning a grade.
- The writeup of the studies is worth 10 points. Both content and writing style will be considered when assigning a grade.