Programming Project 2: CPU Scheduler Simulation
Program Due: 9:20 AM, Friday, February 17, 2012
Writeup Due: 10:00 AM, Tuesday, February 21, 2012

For this project, you will be developing a simulator in C to be used to study CPU scheduling. You may work alone or in groups of 2 or 3. 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 Monday, February 6, 2012, and be confirmed by all group members by electronic mail to *jteresco@siena.edu*. All group members will be assigned the same grade for the project.

Warning: Significant design and programming effort will be necessary to complete this project. You may develop your programs in any environment, as long as I can compile and execute your code on either the Roger Bacon 306 Linux systems or on winterstorm. If you are not yet a C expert, this is an opportunity to become an expert. This is strongly encouraged, but you may choose to program in another language at the cost of a 10% penalty on the programming part of this assignment.

The case studies and writeup account for a significant part of the grade, so keep that in mind as you plan your time. Although the programming is not due until 9:20 AM, Friday, February 17, 2012, I recommend finishing up the code sooner to leave extra time for the writeup.

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* (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 interesting can happen between the current simulation time and the time 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. We know that we didn't miss anything interesting, because if something else was going to happen before that event, an event would have been in the system that would come out of the event queue before this one.

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. It does not matter which order you process such events in your simulator.

We use this model to simulate a simple operating system. Processes arrive in the system and take turns executing on one or more CPUs, 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 CPU scheduling algorithms.

Your program should implement a queueing system, as shown here:
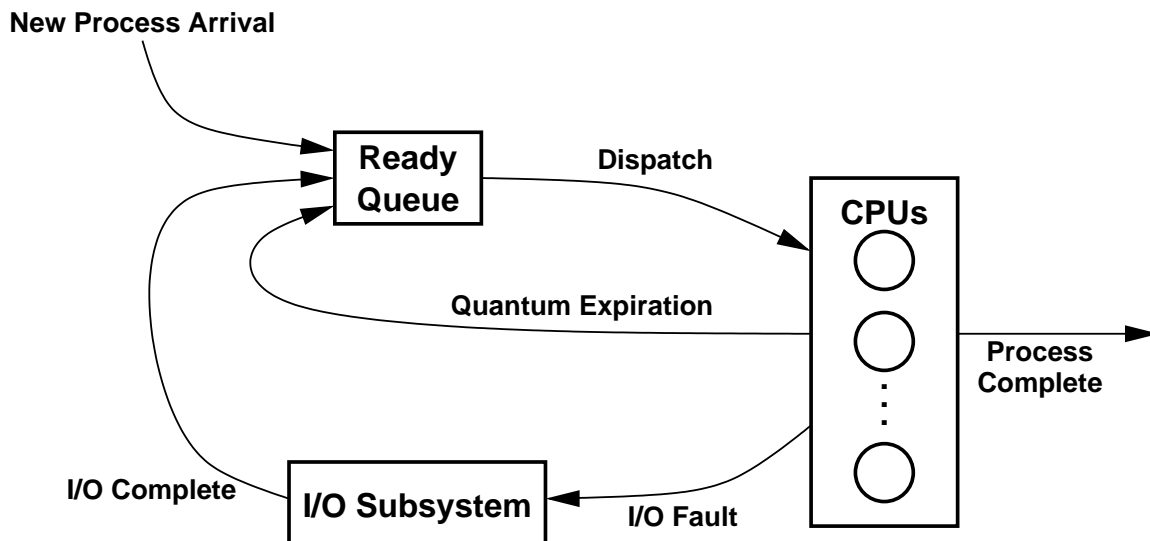
Processes enter the system and wait for their turn on a CPU. They run on a CPU, possibly being pre-empted by the scheduler or for I/O service, until they have spent their entire service time on a CPU, at which point they leave the system.

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.

An input file specifies the frequency and length of new processes to be introduced into the system. Each time a new process enters the system, an event is generated that will result in the creation of the next process of the same type to enter the system. The format of the input file and the meaning of its entries are described later in this document.

The *ready queue* contains the processes in the system that are ready to run on a CPU when no CPU is available to run them. A process is selected from the front of the ready queue when a CPU becomes available. If the ready queue is empty, a process entering the ready state should be assigned to any idle CPU immediately.

Each process can spend up to, but not more than, one *quantum* on a CPU before it is switched out. The quantum is a system wide constant value, entered as a parameter by the user. Note: setting a quantum of 0 should result in a non-preemptive scheduler.

When a process is assigned to a CPU, an event should be generated that will remove it from a 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.

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 to 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 cost is entered as a parameter by the user and should be accounted for when generating events and gathering statistics.

A process executes for a set number of clock cycles (its *burst time*) 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.

The time needed to service an I/O fault is determined for each process, and remains constant for the lifetime of that process. We make the unrealistic assumption 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

Your simulator should be controlled by a number of command-line parameters and an input file describing the creation of new processes.

You may find the `getopt_long(3)` library function useful for parsing your command-line parameters.

Switch	Description	Default
`--procgen-file, -f`	File name for the process generation description file	`pg.txt`
`--num-cpus, -c`	Number of CPUs	1
`--quantum, -q`	Quantum for pre-emptive scheduling	0
`--stop-time, -t`	Simulation stop time	(none)
`--switch-time, -w`	Context switch cost	0
`--no-io-faults, -n`	Disables I/O faults	
`--verbose, -v`	Enable verbose output	
`--batch, -b`	Display parseable batch output	
`--help, -h`	Display help message	

The remaining parameters are described in the process generation file. The format of the file is

```
ntypes
name_0 c_0 b_0 a_0 i_0
name_1 c_1 b_1 a_1 i_1
...
```

$\texttt{name}_{ntypes-1}$ $\texttt{c}_{ntypes-1}$ $\texttt{b}_{ntypes-1}$ $\texttt{a}_{ntypes-1}$ $\texttt{i}_{ntypes-1}$

The `ntypes` line specifies the number of process types to be used in the simulation. Each successive line defines the parameters for one of those types. $\texttt{name}_j$ is a label for the process type, $\texttt{c}_j$ is the average CPU time requirement for processes of this type, $\texttt{b}_j$ is the average burst time for processes of this type, $\texttt{a}_j$ is the average interarrival time for processes of this type, and $\texttt{i}_j$ is the average time required to service each I/O fault for processes of this type.

For example,

```
2
interactive 20 10 80 5
batch 500 250 1000 10
```

specifies two process types, one called "interactive" with average CPU service time 20, average burst time 10, average interarrival time 80, and average I/O service time per fault of 5, and a second called "batch" with average CPU service time 500, average burst time 250, average interarrival time 1000, and average I/O service time per fault of 10.

During a simulation, you will need to generate random times with a given average. For example, you might want to generate process interarrival times that average out to 100. In order to allow a wide range of possible values and to simulate realistic situations more accurately, use an *exponential distribution* of values with the given average value. This will result in a larger number of smaller values and a smaller number of larger values. In other circumstances, it may make more sense to use a *uniform distribution*, where values are randomly selected within an interval. C functions that provide random values for both exponential and uniform distributions are available in `~jteresco/shared/cs330/labs/cpusched`. You are welcome to use these functions or write your own.

For each process type, use an exponential distribution to generate process interarrival times averaging the given value. When generating new processes, use an exponential distribution to generate the new process' CPU service time and I/O service time and a uniform distribution to generate the burst time (in the range between 1 and twice the average burst time specified).

Statistics to Gather

The whole point of the simulation is to gather statistics to see how a given system performs over time. You should gather and report the following statistics:

- Length of the simulation in time units and number of events processed.

- Final and average length of the event queue.

- Final and average length of the ready queue.

- For each process type:

- the number of processes of this type completed.

- the throughput (number of processes of this type completed per unit time).

- last, longest, and average turnaround times for processes of this type.

- For each CPU in the system:

  - active time (time spent running jobs), context switch time, and idle time.

  Report each as a raw amount of time and as a percentage of simulation time.

Program Outline

Your program's execution should follow this approach:

```
read and set parameters
create an empty event queue, empty ready queue, initialize stats
insert initial process creation events into event queue
while (simulation time not expired)
  take next event from event queue
  process event (move job among queues or CPU, create new job,
                 add new events, update statistics, etc)
report statistics
```

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. You can 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 many thousands of 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.

Implementing an appropriate `--batch` output mode will make it easier for you to parse the statistics you gather into meaningful and plottable data.

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

Notes

- You may develop your code anywhere, but make sure it works on winterstorm or the Roger Bacon 306 Linux 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 Subversion repository for source code control. Whatever you do, please do not share passwords or create world-readable (or worse, world-writeable) directories.

- Since real computer scientists don't use Office, you are encouraged (but not required) to generate your writeup using LaTeX. Also consider using gnuplot to generate your graphs. Both packages are available on the lab Linux systems. There is a sample LaTeX document in my shared area on the Roger Bacon 306 Linux systems.

Simulator Submission and Evaluation

This assignment is graded out of 50 points, 40 of which are based on the simulation software.

Make sure that all group members' names appear in all files. Only one group member should submit. Group submissions must include a rough breakdown of the work done by each group member.

By 9:20 AM, Friday, February 17, 2012, submit documented source code, with a brief README file that describes how to build and run your simulator. All necessary files should be submitted by email to *jteresco@siena.edu*. Include a Makefile to allow easy compilation.

Correctness, design, documentation, style, and efficiency will be considered when assigning a grade. All files should compile without warnings when using gcc's -Wall option.

Grading Breakdown	
Makefile, build instructions, README	2 points
-h (help output)	1 point
Command-line parameter parsing	2 points
Basic DES setup: event queue-based processing	5 points
Process generation file	3 points
CPU Scheduling simulation correctness	10 points
Gathering and reporting of statistics	5 points
Design and Style	5 points
Documentation	5 points
Simulation efficiency	2 points

Simulation Studies Submission and Evaluation

The writeup of the studies is worth 10 points.

By 10:00 AM, Tuesday, February 21, 2012, you should submit a writeup, by email to *jteresco@-siena.edu*, that includes a description of each of your two (or three, for groups of three) studies. Please submit as a PDF file. 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" (OK, a "real OS") situation. Include graphs to show trends as the key parameter(s) change.

Both content and writing style will be considered when assigning a grade.