CS201 Graduate Project

For this project, you will be developing a simulator to be used to study CPU scheduling.

You'll use this simulator to peform experiments with CPU scheduling.

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 this kind of 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 advanced immediately to that event's timestamp. We know that we didn't miss anything interesting during the time we skipped over, 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.

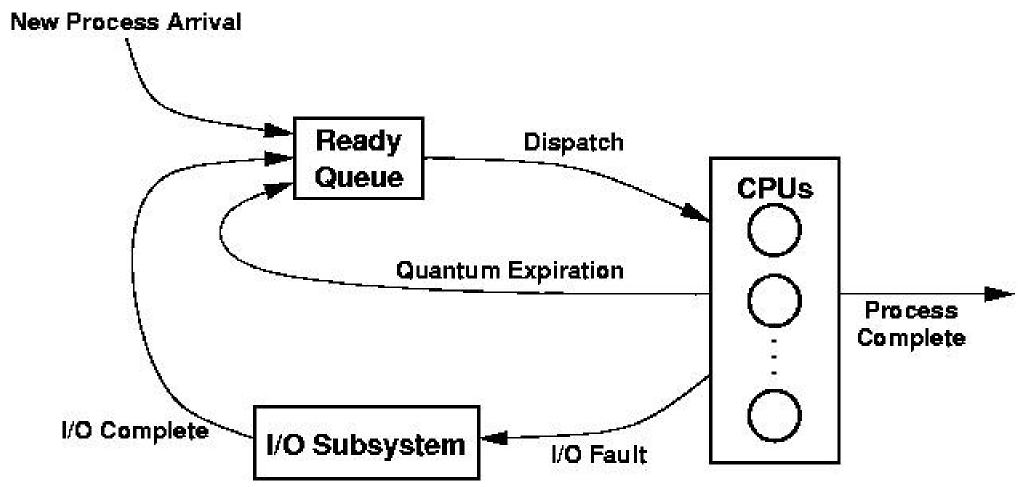
Multiple events may have the same timestamp. This models events that happen concurrently. Even though the events are processed sequentially by the simulator, they occur at the same logical time. It does not matter which order you process such events in your simulator. Any tiebreaking ordering results in an equally valid simulation result.

We will 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 represented 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: specifying 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.

- 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.
- 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.

- 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. (In other words, we won't use put processes in a queue to wait for an I/O device.)

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.

If you're using the C language, you'll find the getopt_long(3) library function useful for parsing your command-line parameters.

Switch	Short Form	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
name0 c0 b0 a0 i0
name1 c1 b1 a1 i1
…
namentypes-1 cntypes-1 bntypes-1 antypes-1 intypes-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. namej is a label for the process type, cj is the average CPU time requirement for processes of this type, bj is the average burst time for processes of this type, aj is the average interarrival time for processes of this type, and ij 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 mean. 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 mean 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 my gitlab site (https://gitlab.uvm.edu/Jason.Hibbeler/ForStudents/tree/master/CS201/Assignments). 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

While it's obviously true that any excuse to build a discrete event simulator is a good excuse to build a discete event simulator, 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.

Experiments

Conduct two 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.

Tell me what your two studies will be ahead of time so that we can discuss them (see Deliverables section).

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 you can 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.

Additional Notes

You can use any language you want. It might make sense to use Java or object-oriented Python so that you can create classes and take advantage of encapsulation and abstraction.

Maintain stats for each CPU.

Assign a unique PID for each new process object in the system.

Deliverables

Get started early. The first thing you do should be a design of the system: what data structures or classes will you need, and how will they interact? What are the major program components?

Turn in, by Monday, Oct. 22nd, a description of your design, showing your class hierarchy or structures, the program components, and a desciption of how these will interact.

Turn in, by Monday, Nov. 12th, a description of the experiments you will perform using your simluator.

The full project will be due Monday, Dec. 3rd. You should submit:

- your working code, with a description of how to compile your code and run your system
- a short (3-4 page) write-up describing your system
- results of your two experiments, with complete description, graphs, and analysis

This large project will consist of 25% of your grade.