SIENA*college*
Computer Science

Computer Science 330
Operating Systems
Siena College
Fall 2022

Programming Project 1: CPU Scheduler Simulation

Due: 11:59 PM, Thursday, September 29, 2022

In this programming project, you will be developing a program to simulate the flow of processes through a model operating system.

You may work alone but are encouraged to work in groups of 2 or 3. Collaboration within a group is, of course, unrestricted. You may discuss the project with members of other groups, but what you turn in must be your own group's work. Groups must be formed and those choosing to work alone must declare this by no later than 11:59 PM, Thursday, September 22, 2022.

This project requires a significant design and programming effort and you will need to dedicate some time to it consistently throughout the time from the assignment's release until the due date.

You may develop your programs in any environment, as long as they can be compiled and executed on noreaster. This should not be a problem for Java or Python 3 programs, but could be for C or C++. If you wish to work in a programming language other than those, please ask first.

Getting Set Up

You will receive an email with the link to follow to set up your GitHub repository, which will be named `cpusched-proj-yourgitname`, for this programming project. Only one member of the group should follow the link to set up the repository on GitHub, then others should request a link to be granted write access.

All GitHub repositories must be created with all group members having write access and all group member names specified in the `README.md` file by 11:59 PM, Thursday, September 22, 2022. This applies to those who choose to work alone as well!

Discrete Event Simulation

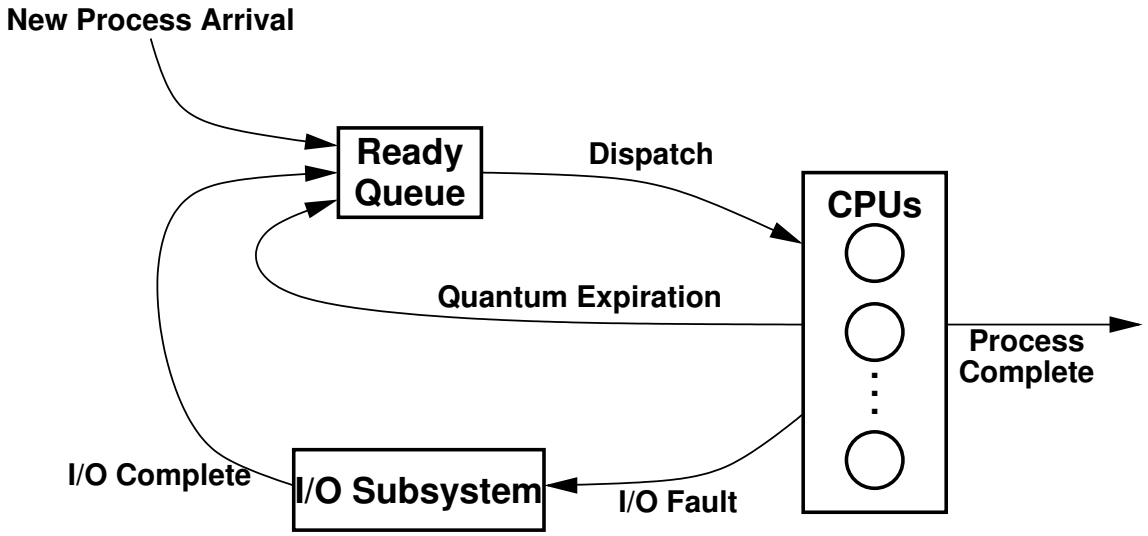
Discrete event simulation (DES) is a method used to model real world systems that can be decomposed into a series of logical *events* that happen at specific times. The main restriction on this kind of system is that an event cannot affect the outcome of a prior event. When an event is generated, it 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 CPU scheduling in 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

Your program should implement the DES to model CPU scheduling using 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 become blocked waiting 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. To simulate a non-preemptive scheduling system, the quantum parameter should be specified as 0.

When a process is assigned to a CPU, an event should immediately 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 length, the process will run to completion before it is required to leave the CPU, so 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, or scheduling is non-preemptive, the process will leave the CPU due to its next I/O fault, so an event is generated that will cause the process to leave for I/O service.
3. Else the process' quantum will expire before it completes or requires I/O. 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.

Those writing in C will find the `getopt_long(3)` library function useful for parsing your command-line parameters. See: `https://github.com/SienaCSISOperatingSystems/getopt-example`

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
...
name_{ntypes-1} c_{ntypes-1} b_{ntypes-1} a_{ntypes-1} i_{ntypes-1}
```

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 `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 `random.c` in your repository. 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 be able 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
  advance time to next event's timestamp
  process event (move job among queues or CPU, create new job,
                 add new events, update statistics, etc)
report statistics
```

Notes

- You may develop your code anywhere, but make sure it works on noreaster or on a Mac.
- You can run the reference solution on noreaster. It can be found in `/home/cs330/cpusched`, with the executable named `cpusim` and a few example process generation files. You should be able to run the executable, but you will not be able to copy it.

Frequently Asked Questions

Additional questions and their answers will be added here as they are asked (and answered).

- Q: Does there need to be a structure to represent each CPU?

  A: You don't necessarily need a data structure for the CPU, but at a minimum you'll need a place keep per-CPU stats. A struct might be appropriate there.

- Q: Your sample executable does not actually stop at the specified simulation stop time. Why not?

  A: The simulation can't stop at the exact time unless there happens to be an event in the event queue that will advance the global clock to precisely that time. It's up to you to decide if you want to take extra care to stop the gathering of statistics at the specified simulation stop time or at the time of the first event that happens at or after that time.

- Q: If we generate our own PIDs for the "process" objects, should we use some mechanism to make them unique? If they go all the way to `LONG_MAX` or w/e they won't be.

  A: I've never worried about running out of process ids - I use a long and just let it grow. I suppose it could go negative eventually and the program likely wouldn't care.

- Q: Do you have any objection to the use of global variables to store several important things?

  A: No objection to globals, and my reference solution makes use of them to avoid passing around a lot of pointers.

- Q: Is it OK to use examples and code from other classes in this project?

  A: Yes, with appropriate attribution.

- Q: Should I use threads to represent the concurrency in the system?

  A: No! This should be a single-threaded program. The concurrency is totally logical, and happens through the timestamps on the events.

Simulator Submission and Evaluation

Make sure that all group members' names appear in all files. Evidence of significant contributions for all group members should be included in commit messages.

By 11:59 PM, Thursday, September 29, 2022, commit and push your documented source code, with a brief entry in the `README`.md file that describes how to build and run your simulator. Include a `Makefile` or an appropriate equivalent 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, or the equivalent in the implementation language.

Grading

This assignment will be graded out of 75 points.

Feature	Value	Score
`Makefile`, build instructions, `README`	2	
`-h` (help output)	1	
Command-line parameter parsing	3	
Basic DES setup: event queue-based processing	10	
Process generation file	5	
CPU Scheduling simulation correctness	25	
Gathering and reporting of statistics	15	
Design and Style	5	
Documentation	7	
Simulation efficiency	2	
Total	75	