CSCI 4210 — Operating Systems
Simulation Project Part I (document version 1.1)
Processes and CPU Scheduling

Overview

- This assignment is due in Submitty by 11:59PM EST on **(v1.1)** Friday, July 21, 2023
- This project is to be completed either individually or in a team of at most three students; teams are formed in the Submitty gradeable, but do **not** submit any code until we announce that auto-grading is available
- Beyond your team (or yourself if working alone), **do not share your code**; however, feel free to discuss the project content and your findings with one another on our Discussion Forum
- To appease Submitty, you must use one of the following programming languages: C, C++, Java, or Python (be sure you choose only **one** language for your entire implementation)
- You will have 16 penalty-free submissions, after which a small penalty will be applied
- You can use at most three late days on this assignment; in such cases, each team member must use a late day
- All submitted code **must** successfully compile and run on Submitty, which currently uses Ubuntu v20.04.6 LTS
- If you use C or C++, your program **must** successfully compile via `gcc` or `g++` with no warning messages when the `-Wall` (i.e., warn all) compiler option is used; we will also use `-Werror`, which will treat all warnings as critical errors; **(v1.1)** the `-lm` flag will also be included; the `gcc`/`g++` compiler is currently version 9.4.0 (`Ubuntu 9.4.0-1ubuntu1~20.04.1`)
- For source file naming conventions, be sure to use `*.c` for C and `*.cpp` for C++; in either case, you can also include `*.h` files
- If you use Java, name your main Java file `Project.java`, and note that the `javac` compiler is currently version 8 (`javac 1.8.0_362`); do **not** use the `package` directive
- For Python, you must use `python3`, which is currently Python 3.8.10; be sure to name your main Python file `project.py`
- For Java and Python, be sure no warning messages or extraneous output occur during compilation/interpretation
- Please "flatten" all directory structures to a single directory of source files before submitting your code

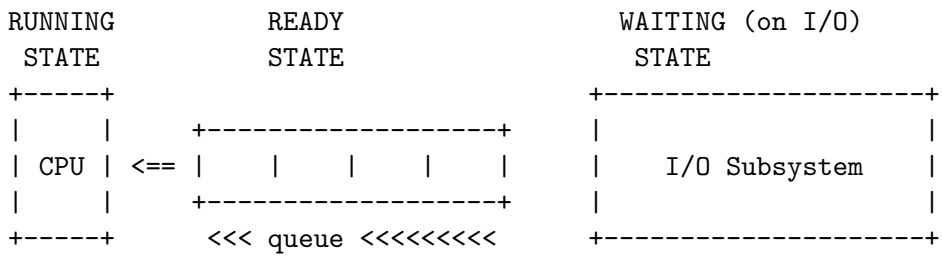
Project specifications

For our simulation project, which encompasses **both** Part I and Part II, you will implement a simulation of an operating system. The overall focus will be on processes, assumed to be resident in memory, waiting to use the CPU. Memory and the I/O subsystem will not be covered in depth in either part of this project.

Conceptual design

A **process** is defined as a program in execution. For this assignment, processes are in one of the following three states, corresponding to the picture shown further below.

- **RUNNING:** actively using the CPU and executing instructions
- **READY:** ready to use the CPU, i.e., ready to execute a CPU burst
- **WAITING:** blocked on I/O or some other event



Processes in the **READY** state reside in a queue called the ready queue. This queue is ordered based on a configurable CPU scheduling algorithm. You will implement specific CPU scheduling algorithms in Part II of this project.

All implemented algorithms (in Part II) will be simulated for the **same set of processes**, which will therefore support a comparative analysis of results. In Part I, the focus is on generating useful sets of processes via pseudo-random number generators.

Back to the conceptual model, when a process is in the **READY** state and reaches the front of the queue, once the CPU is free to accept the next process, the given process enters the **RUNNING** state and starts executing its CPU burst.

After each CPU burst is completed, if the process does not terminate, the process enters the **WAITING** state, waiting for an I/O operation to complete (e.g., waiting for data to be read in from a file). When the I/O operation completes, depending on the scheduling algorithm, the process either (1) returns to the **READY** state and is added to the ready queue or (2) preempts the currently running process and switches into the **RUNNING** state.

Note that preemptions occur only for certain algorithms.

Simulation configuration

The key to designing a useful simulation is to provide a number of configurable parameters. This allows you to simulate and tune for a variety of scenarios, e.g., a large number of CPU-bound processes, differing average process interarrival times, multiple CPUs, etc.

Define the simulation parameters shown below as tunable constants within your code, all of which will be given as command-line arguments. In Part II of the project, additional parameters will be added.

- `argv[1]`: Define n as the number of processes to simulate. Process IDs are assigned in alphabetical order `A` through `Z`. Therefore, you will have at most 26 processes to simulate.
- `argv[2]`: Define n_{cpu} as the number of processes that are CPU-bound. For this project, we will classify processes as I/O-bound or CPU-bound. The n_{cpu} CPU-bound processes, when generated, will have CPU burst times that are longer by a factor of 4 and will have I/O burst times that are shorter by a factor of 8.
- `argv[3]`: We will use a pseudo-random number generator to determine the interarrival times of CPU bursts. This command-line argument, i.e. *seed*, serves as the seed for the pseudo-random number sequence. To ensure predictability and repeatability, use `srand48()` with this given seed before simulating **each** scheduling algorithm and `drand48()` to obtain the next value in the range $[0.0, 1.0)$. For languages that do not have these functions, implement an equivalent 48-bit linear congruential generator, as described in the `man` page for these functions in C.[1]
- `argv[4]`: To determine interarrival times, we will use an exponential distribution; therefore, this command-line argument is parameter λ. Remember that $\frac{1}{\lambda}$ will be the average random value generated, e.g., if $\lambda = 0.01$, then the average should be appoximately 100. See the `exp-random.c` example; and use the formula shown in the code, i.e., `-log( r ) / lambda`, where `log` is the natural logarithm.
- `argv[5]`: For the exponential distribution, this command-line argument represents the upper bound for valid pseudo-random numbers. This threshold is used to avoid values far down the long tail of the exponential distribution. As an example, if this is set to 3000, all generated values above 3000 should be skipped. For cases in which this value is used in the ceiling function (see the next page), be sure the ceiling is still valid according to this upper bound.

[1] Feel free to post your code for this on the Discussion Forum for others to use.

Pseudo-random numbers and predictability

A key aspect of this assignment is to compare the results of each of the simulated algorithms with one another given the same initial conditions, i.e., the same initial set of processes.

To ensure each CPU scheduling algorithm runs with the same set of processes, carefully follow the algorithm below to create the set of processes.

Define your exponential distribution pseudo-random number generation function as `next_exp()`. Then, for each of the n processes, in order `A` through `Z`, perform the steps below, with CPU-bound processes generated last. Note that all generated values are integers.

1. Identify the initial process arrival time as the "floor" of the next random number in the sequence given by `next_exp()`; note that you could therefore have a zero arrival time
2. Identify the number of CPU bursts for the given process as the "ceiling" of the next random number generated from the **uniform distribution** (i.e., obtained via `drand48()`) multiplied by 64; this should obtain a random integer in the inclusive range $[1, 64]$
3. For *each* of these CPU bursts, identify the CPU burst time and the I/O burst time as the "ceiling" of the next two random numbers in the sequence given by `next_exp()`; multiply the I/O burst time by 10 such that I/O burst time is generally an order of magnitude longer than CPU burst time; for CPU-bound processes, multiply the CPU burst time by 4 and divide the I/O burst time by 8; for the last CPU burst, do not generate an I/O burst time (since each process ends with a final CPU burst)

Measurements

To start planning for Part II, there are a number of measurements you will want to track in your simulation. For each algorithm, you will count the number of preemptions and the number of context switches that occur. Further, you will measure CPU utilization by tracking CPU usage and CPU idle time.

Specifically, for **each CPU burst**, you will measure CPU burst time (given), turnaround time, and wait time.

CPU burst time

CPU burst times are randomly generated for each process that you simulate via the above algorithm. CPU burst time is defined as the amount of time a process is **actually** using the CPU. Therefore, this measure does not include context switch times.

Turnaround time

Turnaround times are to be measured for each process that you simulate. Turnaround time is defined as the end-to-end time a process spends in executing a **single CPU burst**.

More specifically, this is measured from process arrival time through to when the CPU burst is completed and the process is switched out of the CPU. Therefore, this measure includes the second half of the initial context switch in and the first half of the final context switch out, as well as any other context switches that occur while the CPU burst is being completed (i.e., due to preemptions).

Wait time

Wait times are to be measured **for each CPU burst**. Wait time is defined as the amount of time a process spends waiting to use the CPU, which equates to the amount of time the given process is actually in the ready queue. Therefore, this measure does not include context switch times that the given process experiences, i.e., only measure the time the given process is actually in the ready queue.

CPU utilization

Calculate CPU utilization by tracking how much time the CPU is actively running CPU bursts versus total elapsed simulation time.

Required terminal output

For Part I, required terminal output simply summarizes the set of generated processes.

The output format must follow the example shown below.

```
bash$ ./a.out 3 1 1024 0.001 3000
<<< PROJECT PART I -- process set (n=3) with 1 CPU-bound process >>>
I/O-bound process A: arrival time 136ms; 17 CPU bursts:
--> CPU burst 1330ms --> I/O burst 5740ms
--> CPU burst 735ms --> I/O burst 660ms
--> CPU burst 24ms --> I/O burst 450ms
--> CPU burst 683ms --> I/O burst 1850ms
--> CPU burst 1100ms --> I/O burst 14220ms
--> CPU burst 1000ms --> I/O burst 6950ms
--> CPU burst 1166ms --> I/O burst 5290ms
--> CPU burst 1561ms --> I/O burst 2820ms
--> CPU burst 149ms --> I/O burst 720ms
--> CPU burst 388ms --> I/O burst 7370ms
--> CPU burst 34ms --> I/O burst 9540ms
--> CPU burst 954ms --> I/O burst 9960ms
--> CPU burst 236ms --> I/O burst 17010ms
--> CPU burst 734ms --> I/O burst 5850ms
--> CPU burst 1124ms --> I/O burst 19640ms
--> CPU burst 62ms --> I/O burst 2350ms
--> CPU burst 647ms
I/O-bound process B: arrival time 929ms; 12 CPU bursts:
--> CPU burst 2438ms --> I/O burst 880ms
--> CPU burst 1173ms --> I/O burst 540ms
...
--> CPU burst 2245ms --> I/O burst 27510ms
--> CPU burst 615ms
CPU-bound process C: arrival time 26ms; 60 CPU bursts:
--> CPU burst 2920ms --> I/O burst 368ms
--> CPU burst 2328ms --> I/O burst 3112ms
...
--> CPU burst 1784ms --> I/O burst 626ms
--> CPU burst 1704ms
```

Error handling

If improper command-line arguments are given, report an error message to `stderr` and abort further program execution. In general, if an error is encountered, display a meaningful error message on `stderr`, then abort further program execution.

Error messages must be one line only and use the following format:

```
ERROR: <error-text-here>
```

Submission instructions

To submit your assignment and also perform final testing of your code, please use Submitty.

Note that this assignment will be available on Submitty a minimum of three days before the due date. Please do not ask when Submitty will be available, as you should first perform adequate testing on your own Ubuntu platform.

Relinquishing allocated resources

Be sure that all resources (e.g., dynamically allocated memory) are properly relinquished for whatever language/platform you use for this assignment. Sloppy programming will likely result in lower grades. Consider doing frequent code reviews with your teammates if working on a team.