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)** Thursday, July 25, 2024
- This project is to be completed either individually or in a team of at most three students; form your team within 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++, or Python (be sure you choose only **one** language for your entire implementation)
- You will have **five** penalty-free submissions on Submitty, after which points will slowly be deducted, e.g., `-1` on submission #6, etc.
- You can use at most three late days on this assignment; in such cases, each team member must use a late day
- You will have at least **three** days before the due date to submit your code to Submitty; if the auto-grading is not available three days before the due date, the due date will be 11:59PM EDT three days after auto-grading becomes available
- All submitted code **must** successfully compile and run on Submitty, which currently uses Ubuntu v22.04.4 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; the `-lm` flag will also be included; the `gcc`/`g++` compiler is currently version 11.4.0 (`Ubuntu 11.4.0-1ubuntu1~22.04`)
- 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
- For Python, you must use `python3`, which is currently Python 3.10.12; be sure to name your main Python file `project.py`; also be sure no warning messages or extraneous output occur during interpretation
- Please "flatten" all directory structures to a single directory of source files before submitting your code

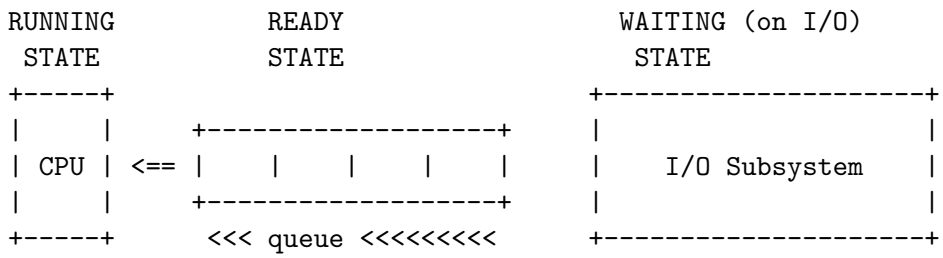
Project specifications

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 a two-character code consisting of an uppercase letter from `A` to `Z` followed by a number from `0` to `9`. Processes are assigned in order `A0`, `A1`, `A2`, ..., `A9`, `B0`, `B1`, ..., `Z9`.
- `*(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 $[0.0, 1.0)$. Since Python does 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, as illustrated in the `exp-random.c` example. 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. In the `exp-random.c` example, use the formula shown in the code, i.e., $\frac{-\ln r}{\lambda}$.
- `*(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.

For each of the n processes, in order `A0` through `Z9`, perform the steps below, with CPU-bound processes generated first. Note that all generated values are integers.

Define your exponential distribution pseudo-random number generation function as `next_exp()` (or another similar name).

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** obtained via `drand48()` multiplied by 32; this should obtain a random integer in the inclusive range $[1, 32]$
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 8 such that I/O burst time is close to an order of magnitude longer than CPU burst time; as noted above, 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 track 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 summarizes the set of generated processes.

The output format must follow the example shown below.

```
bash$ ./a.out 3 1 32 0.001 1024
<<< PROJECT PART I
<<< -- process set (n=3) with 1 CPU-bound process
<<< -- seed=32; lambda=0.001000; bound=1024
CPU-bound process A0: arrival time 319ms; 25 CPU bursts:
==> CPU burst 1448ms ==> I/O burst 608ms
==> CPU burst 316ms ==> I/O burst 474ms
==> CPU burst 3556ms ==> I/O burst 964ms
==> CPU burst 2516ms ==> I/O burst 14ms
==> CPU burst 732ms ==> I/O burst 669ms
==> CPU burst 1872ms ==> I/O burst 82ms
==> CPU burst 1020ms ==> I/O burst 486ms
==> CPU burst 228ms ==> I/O burst 347ms
==> CPU burst 2092ms ==> I/O burst 222ms
==> CPU burst 3380ms ==> I/O burst 59ms
...
==> CPU burst 820ms ==> I/O burst 1005ms
==> CPU burst 920ms ==> I/O burst 448ms
==> CPU burst 1876ms
I/O-bound process A1: arrival time 506ms; 5 CPU bursts:
==> CPU burst 971ms ==> I/O burst 432ms
==> CPU burst 129ms ==> I/O burst 1264ms
==> CPU burst 188ms ==> I/O burst 1496ms
==> CPU burst 302ms ==> I/O burst 1328ms
==> CPU burst 73ms
I/O-bound process A2: arrival time 821ms; 15 CPU bursts:
==> CPU burst 408ms ==> I/O burst 5512ms
==> CPU burst 182ms ==> I/O burst 3744ms
==> CPU burst 89ms ==> I/O burst 8ms
==> CPU burst 781ms ==> I/O burst 256ms
==> CPU burst 107ms ==> I/O burst 656ms
==> CPU burst 65ms ==> I/O burst 5872ms
==> CPU burst 69ms ==> I/O burst 4344ms
==> CPU burst 232ms ==> I/O burst 5120ms
==> CPU burst 225ms ==> I/O burst 4688ms
==> CPU burst 42ms ==> I/O burst 7504ms
==> CPU burst 335ms ==> I/O burst 4992ms
==> CPU burst 247ms ==> I/O burst 5960ms
==> CPU burst 66ms ==> I/O burst 64ms
==> CPU burst 155ms ==> I/O burst 5760ms
==> CPU burst 280ms
```

Required output file

In addition to the above output (which should be sent to `stdout`), generate an output file called `simout.txt` that will ultimately contain statistics for each simulated algorithm. For Part I, The file format is shown below (with `#` as a placeholder for actual numerical data). Use the "ceiling" function out to exactly three digits after the decimal point for your averages.

```
-- number of processes: #
-- number of CPU-bound processes: #
-- number of I/O-bound processes: #
-- CPU-bound average CPU burst time: #.### ms
-- I/O-bound average CPU burst time: #.### ms
-- overall average CPU burst time: #.### ms
-- CPU-bound average I/O burst time: #.### ms
-- I/O-bound average I/O burst time: #.### ms
-- overall average I/O burst time: #.### ms
```

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.