CSCI 4210 — Operating Systems
Simulation Project (document version 1.0)
CPU Scheduling

Overview

- This assignment is due in Submitty by 11:59PM EST on Friday, July 15, 2022, **but plan to start this project early**
- This project is to be completed either individually or in a team of at most three students (in either section); 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 20 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
- Given that your simulation results might not entirely match the expected output on Submitty, we will cap your auto-graded grade at **50 points** even though there will be more than 50 auto-graded points available in Submitty; this should help you avoid the need to handle unusual corner cases, which are not an important aspect of this project
- All submitted code **must** successfully compile and run on Submitty, which uses Ubuntu v20.04.4 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 `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_312`); 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

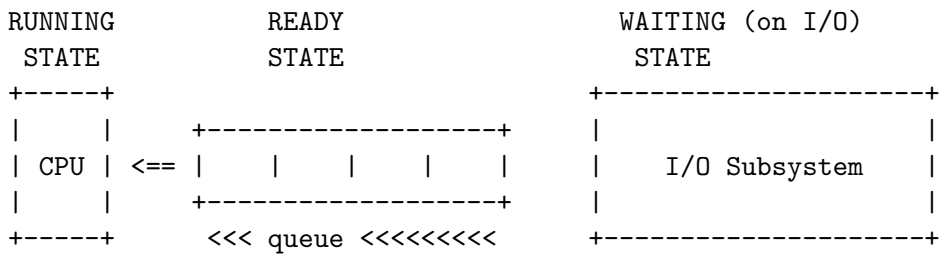
Project specifications

For our simulation project, you will implement a simulation of an operating system. The 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 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. There are **four** algorithms that you are to implement: first-come-first-served (FCFS); shortest job first (SJF); shortest remaining time (SRT); and round robin (RR).

All four of these algorithms will be simulated for the **same set of processes**, which will therefore support a comparative analysis of results. As such, when you run your program, all four algorithms are to be simulated in succession.

In general, 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 the SRT and RR algorithms. Each algorithm is described in the pre-recorded lecture videos (also summarized on the next page).

First-come-first-served (FCFS)

The FCFS algorithm is a non-preemptive algorithm in which processes simply line up in the ready queue, waiting to use the CPU. This is your baseline algorithm (and could be implemented as RR with an "infinite" time slice).

Shortest job first (SJF)

In SJF, processes are stored in the ready queue in order of priority based on their anticipated CPU burst times. More specifically, the process with the shortest predicted CPU burst time will be selected as the next process executed by the CPU.

Shortest remaining time (SRT)

The SRT algorithm is a preemptive version of the SJF algorithm. In SRT, when a process arrives, as it enters the ready queue, if it has a predicted CPU burst time that is less than the remaining predicted time of the currently running process, a preemption occurs. When such a preemption occurs, the currently running process is added back to the ready queue.

Round robin (RR)

The RR algorithm is essentially the FCFS algorithm with time slice t_{slice}. Each process is given t_{slice} amount of time to complete its CPU burst. If the time slice expires, the process is preempted and added to the end of the ready queue.

If a process completes its CPU burst before a time slice expiration, the next process on the ready queue is immediately context-switched in to use the CPU.

Skipping preemptions in RR

For your simulation, if a preemption occurs and there are no other processes on the ready queue, do not perform a context switch. For example, given process `G` is using the CPU and the ready queue is empty, if process `G` is preempted by a time slice expiration, do not context-switch process `G` back to the empty queue; instead, keep process `G` running with the CPU and do **not** count this as a context switch. In other words, when the time slice expires, check the queue to determine if a context switch should occur.

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 following simulation parameters as tunable constants within your code, all of which will be given as command-line arguments:

- `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]`: 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[3]`: 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[4]`: 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.
- `argv[5]`: Define t_{cs} as the time, in milliseconds, that it takes to perform a context switch. Remember that a context switch occurs each time a process leaves the CPU and is replaced by another process. More specifically, the first half of the context switch time (i.e., $\frac{t_{cs}}{2}$) is the time required to remove the given process from the CPU; the second half of the context switch time is the time required to bring the next process in to use the CPU. Therefore, expect t_{cs} to be a positive even integer.
- `argv[6]`: For the SJF and SRT algorithms, since we cannot know the actual CPU burst times beforehand, we will rely on estimates determined via exponential averaging. As such, this command-line argument is the constant α. Note that the initial guess for each process is $\tau_0 = \frac{1}{\lambda}$. When calculating τ values, use the "ceiling" function for all calculations.
- `argv[7]`: For the RR algorithm, define the time slice value, t_{slice}, measured in milliseconds.

[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 is given the same set of processes, carefully follow the algorithm below to define the set of processes. This algorithm should be fully executed before applying any of the scheduling algorithms.

Define your exponential distribution pseudo-random number generation function as `next_exp()`. Then, for each of the n processes, in order `A` through `Z`:

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 100; this should obtain a random integer in the inclusive range $[1, 100]$
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 the last CPU burst, do not generate an I/O burst time (since each process ends with a final CPU burst)

After you simulate each scheduling algorithm, you must reset the simulation back to the initial set of processes and set your elapsed time back to zero. More specifically, you must re-seed your random number generator to ensure the same set of processes and interarrival times.

Note that there may be times during your simulation in which the simulated CPU is idle because all processes are busy performing I/O. Also, your simulation ends when all processes terminate.

Handling "ties"

If different types of events occur at the same time, simulate these events using the following order: (a) CPU burst completion; (b) process starts using the CPU; (c) I/O burst completions (i.e., back to the ready queue); and (d) new process arrivals.

Further, any "ties" that occur *within* one of these three categories are to be broken using process ID order. As an example, if processes `G` and `S` happen to both finish with their I/O at the same time, process `G` wins this "tie" (because `G` is alphabetically before `S`) and is therefore added to the ready queue before process `S`.

Be sure you do not implement any additional logic for the I/O subsystem. In other words, there are no I/O queues to implement.

Measurements

For each algorithm, count the number of preemptions and the number of context switches that occur. Further, measure CPU utilization by tracking CPU usage and CPU idle time.

For **each CPU burst**, measure CPU burst time (given), turnaround time, and wait time. These will simply be averaged together for each algorithm.

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** and 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.

More specifically, a process leaves the ready queue when it is switched into the CPU, which takes half of context switch time t_{cs}. Likewise, a preempted process leaves the CPU and enters the ready queue after the first half of t_{cs}.

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

Your simulator should keep track of elapsed time t (measured in milliseconds), which is initially zero for each scheduling algorithm. As your simulation proceeds, t advances to each "interesting" event that occurs, displaying a specific line of output that describes each event.

Your simulator must display results for each of the four algorithms you simulate. For each algorithm, display a summary of the "pseudo-randomly" generated processes (which should be the same for each algorithm), then the "interesting" events from time 0 through time 999, followed only by process termination events and the final end-of-simulation event. Example output files will be posted to Submitty.

Your simulator must display a line of output for each "interesting" event that occurs using the format shown below. Note that the contents of the ready queue are shown for each event.

```
time <t>ms: <event-details> [Q <queue-contents>]
```

The "interesting" events are:

- Start of simulation
- Process arrival (i.e., initially and at I/O completions)
- Process starts using the CPU
- Process finishes using the CPU (i.e., completes a CPU burst)
- Process has its τ value recalculated (i.e., after a CPU burst completion)
- Process preemption
- Process starts performing I/O
- Process finishes performing I/O
- Process terminates by finishing its last CPU burst
- End of simulation

The "process arrival" event occurs every time a process arrives, which includes both the initial arrival time and when a process completes an I/O burst. In other words, processes "arrive" within the subsystem that consists only of the CPU and the ready queue.

The "process preemption" event occurs each time a process is preempted by a time slice expiration in RR or by an arriving process in SRT. When a preemption occurs, a context switch occurs, except for the RR algorithm when the ready queue is empty.

When your simulation ends, display that final event as shown below.

```
time <t>ms: Simulator ended for <algorithm> [Q empty]
```

Be sure that you still include the process removal time (i.e., half the context switch time) for this last process.

Required output file

In addition to the above output (which should be sent to `stdout`), generate an output file called `simout.txt` that contains statistics for each simulated algorithm. 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.

```
Algorithm FCFS
-- average CPU burst time: #.### ms
-- average wait time: #.### ms
-- average turnaround time: #.### ms
-- total number of context switches: #
-- total number of preemptions: #
-- CPU utilization: #.###%
Algorithm SJF
-- average CPU burst time: #.### ms
-- average wait time: #.### ms
-- average turnaround time: #.### ms
-- total number of context switches: #
-- total number of preemptions: #
-- CPU utilization: #.###%
Algorithm SRT
-- average CPU burst time: #.### ms
-- average wait time: #.### ms
-- average turnaround time: #.### ms
-- total number of context switches: #
-- total number of preemptions: #
-- CPU utilization: #.###%
Algorithm RR
-- average CPU burst time: #.### ms
-- average wait time: #.### ms
-- average turnaround time: #.### ms
-- total number of context switches: #
-- total number of preemptions: #
-- CPU utilization: #.###%
```

The averages shown above are averaged over all executed CPU bursts per each algorithm.

To count the number of context switches, you should count the number of times a process **starts** using the CPU.

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.

Analysis questions to think about...

As you work on this project, consider the questions below. Though there is no formal write-up required for this project, you should be able to answer at least some of these questions based on specific results from your simulation. A few of these questions would require features beyond the given requirements.

1. Of the four simulated algorithms, which algorithm is the "best" algorithm for CPU-bound processes? Which algorithm is best-suited for I/O-bound processes?
2. For the SJF and SRT algorithms, what value of α produced the "best" results?
3. For the SJF and SRT algorithms, how does changing from a non-preemptive algorithm to a preemptive algorithm impact your results? Are there specific examples of processes that have improved turnaround or wait times?
4. Identify at least three limitations of your simulation, in particular how the project specifications could be expanded to better model a real-world operating system.
5. Identify other "interesting" events and measurements that you could incorporate into your simulation.
6. Describe a priority scheduling algorithm of your own design, i.e., how could you calculate priority? What are advantages and disadvantages of your algorithm?

Again, the above questions are included here for you to think about and discuss with your teammates and/or others in the course. Feel free to open a discussion about these on our Discussion Forum.