**CSCI 4210 — Operating Systems**
**CSCI 6140 — Computer Operating Systems**
**Project 1 (document version 1.3)**
**CPU Scheduling Simulation**

# Overview

- This project is due by 11:59:59 PM on Tuesday, March 19, 2019.
- This project is to be completed either individually or in a team of at most three students. Teams may consist of both undergraduate and graduate students. Do not share your code with anyone else.
- Note that students registered for CSCI 6140 will be required to submit additional work for this project, as described on page 10.
- You **must** use one of the following programming languages: C, C++, Java, or Python.
- As per usual, your code **must** successfully compile/run on Submitty, which uses Ubuntu v18.04.1 LTS.
- If you use C or C++, your program **must** successfully compile via `gcc` or `g++` with absolutely 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.
- Note that the `gcc`/`g++` compiler is version 7.3.0 (`Ubuntu 7.3.0-27ubuntu1~18.04`). For source file naming conventions, be sure to use `*.c` for C or `*.cpp` for C++. In either case, you can also include `*.h` files.
- If you use Java, name your main Java file `Project1.java`. And note that the `javac` compiler is version 8 (`javac 1.8.0_191`).
- For Python, you must use `python3`, which is Python 3.6.7. Be sure to name your main Python file `project1.py`.
- For Java and Python, be sure no warning messages occur during compilation/interpretation.
- Finally, keep in mind that Project 2 may build on this initial project. Therefore, be sure your code is easily maintainable and extensible.

## Project specifications

In this first project, you will implement a rudimentary simulation of an operating system. The initial 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:

- **READY:** in the ready queue, ready to use the CPU
- **RUNNING:** actively using the CPU
- **BLOCKED:** blocked on I/O

Processes in the **READY** state reside in a queue called the ready queue. This queue is ordered based on a configurable CPU scheduling algorithm. In this first assignment, there are four algorithms to implement: shortest job first (SJF); shortest remaining time (SRT); first come first served (FCFS); and round robin (RR). Note that if you use a large enough time slice, RR essentially becomes the FCFS algorithm.

All four of these algorithms will be simulated for the same set of processes, which will allow for a comparative analysis. As such, when you run your program, all four algorithms are simulated.

And in general, when a process reaches the front of the queue and the CPU is free to accept the next process, the given process enters the **RUNNING** state and starts executing its CPU burst.

After the CPU burst is completed, if the process does not terminate, the process enters the **BLOCKED** 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 some algorithms, i.e., for SRT and RR. Each algorithm is summarized on the next page.

### Shortest Job First (SJF)

In SJF, processes are stored in the ready queue in order of priority based on their CPU burst times. More specifically, the process with the shortest 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, before it enters the ready queue, if it has a CPU burst time that is less than the remaining 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.

### First Come, First Served (FCFS)

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

### Round Robin (RR)

The RR algorithm is essentially the FCFS algorithm with predefined time slice $t_{slice}$. Each process is given $t_{slice}$ amount of time to complete its CPU burst. If this time slice expires, the process is preempted and added to the end of the ready queue (though see the $rr_{add}$ parameter described below).

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

### Skipping preemptions in RR

For your simulation, if a preemption occurs but there are no other processes on the ready queue, do not perform a context switch. For example, if 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 to the user. This allows you to simulate and tune for a variety of scenarios, e.g., a large number of CPU-bound processes, a variety of average process interarrival times, multiple CPUs, etc.

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

- `argv[1]`: We will use a random number generator to determine the interarrival times of CPU bursts. Since we can only generate pseudo-random numbers, the first command-line argument, $s$, serves as the seed for the random number generator. To ensure predictability and repeatability, use `srand48()` with this given seed before each scheduling algorithm and `drand48()` to obtain the next value in the range $[0.0, 1.0)$. For other languages, implement an equivalent 48-bit linear congruential generator, as described in the `man` page for these functions. **(v1.1)** See Piazza for code to use.

- `argv[2]`: To determine interarrival times, we will use an exponential distribution; therefore, the second command-line argument is parameter $\lambda$. 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. **(v1.1)** Use the formula in the code, i.e., `-log( r ) / lambda`.

- `argv[3]`: As part of the exponential distribution, the third command-line argument represents the upper bound for valid pseudo-random numbers. Remember that 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.

- `argv[4]`: Define $n$ as the number of processes to simulate. Process IDs are assigned in alphabetical order `A` through `Z`. Therefore, at most you will have 26 processes to simulate.

- `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. Note that 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 discussed in class on 2/11). As such, this command-line argument is the constant $\alpha$. And note that the initial guess for each process is $\tau_0 = \frac{1}{\lambda}$. **(v1.3)** to calculate $\tau$ values, use the "ceiling" function for all calculations.

- `argv[7]`: For the RR algorithm, define the time slice value, $t_{slice}$, measured in milliseconds

- `argv[8]`: Also for the RR algorithm, define whether processes are added to the end or the beginning of the ready queue when they arrive or complete I/O. This optional command-line argument, $rr_{add}$, is set to either `BEGINNING` or `END`, with `END` being the default behavior.

## 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 conditions. To ensure each CPU scheduling algorithm is given the same set of processes, you will need to carefully follow the algorithm below to identify the set of processes. This algorithm should be fully executed before applying any of the scheduling algorithms. For each of the $n$ processes, in order `A` through `Z`:

1. Identify the initial process arrival time as the next random number in the sequence; more specifically, **(v1.2)** use the "floor" of the next random number generated as shown in the posted `exp-random.c` example (note that you could have a `0` arrival time)
2. Identify the number of CPU bursts for the given process as the next random number multiplied by 100 and truncated (e.g., 0.454928 becomes 45, 0.087188 becomes 8, etc.), **(v1.1)** then incremented by 1; this should obtain a random integer in the range $[1, 100]$
3. For each of these CPU bursts, identify the actual CPU burst time and the I/O burst time as the next two random numbers in the sequence, **(v1.2)** obtained by using the ceiling (i.e., `ceil()`) of the next random number generated as shown in the `exp-random.c` example; 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, when all processes terminate, your simulation ends.

## Handling "ties"

For events that occur at the same time, use the following order to break the "tie": (a) CPU burst completion; (b) I/O burst completion (i.e., back to the ready queue); and then (c) new process arrival.

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

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 in this first project.

### CPU burst time

CPU burst times are randomly generated for each process that you simulate (see algorithm above). 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 process that you simulate. 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).

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 $t_{cs}$.

### 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.

**(v1.3)** 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. See example output files posted on 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>]
```

And the "interesting" events are:

- Start of simulation
- Process arrival
- Process starts using the CPU
- Process finishes using the CPU (i.e., completes a CPU burst)
- **(v1.3)** Process has its $\tau$ 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, i.e., based on the initial arrival time and when a process completes I/O. In other words, processes "arrive" within the subsystem that consists of the CPU and the ready queue.

The "process preemption" event occurs every 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 (unless for RR there are no available processes in the ready queue).

Note that when your simulation ends, you must display that 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). Round to exactly three digits after the decimal point for your averages.

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

Note that averages are averaged over all executed CPU bursts. Also note that 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, the homework submission server. Also use Submitty to define teams, including a team of one if you decide to work alone on this project.

Be sure to include all names and RCS IDs in comments at the top of each source file submitted.

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

That said, to make sure that your program does execute properly everywhere, including Submitty, consider using the techniques below.

First, as discussed in class (on 1/10), use the `DEBUG_MODE` technique to make sure you do not submit any debugging code. Here is an example in C:

```
#ifdef DEBUG_MODE
    printf( "the value of x is %d\n", x );
    printf( "the value of q is %d\n", q );
    printf( "why is my program crashing here?!" );
    fflush( stdout );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE p1.c
```

Second, as discussed in class (on 1/14), output to standard output (`stdout`) is buffered. To disable buffered output for grading on Submitty, use `setvbuf()` as follows in C:

```
setvbuf( stdout, NULL, _IONBF, 0 );
```

You would not generally do this in practice, as this can substantially slow down your program, but to ensure correctness on Submitty, this is a good technique to use.

### 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 potentially lead to grading penalties. Consider doing frequent code reviews with your teammates.

## Graduate section requirements

For students registered for CSCI 6140, additional analysis is required.

Please answer the questions below by submitting a PDF file called `<userid>-p1-analysis.pdf` (e.g., `goldsd3-p1-analysis.pdf`). Answer all questions below in no more than five pages.

Note that each student registered for CSCI 6140 must write up his or her own answers even if you are working on a team.

1. Of the four simulated algorithms, which algorithm is the "best" algorithm? Which algorithm is best-suited for CPU-bound processes? Which algorithm is best-suited for I/O-bound processes? Support your answer by citing specific simulation results.
2. For the RR algorithm, how does changing `rr_add` from `END` to `BEGINNING` impact your results? Which approach is better?
3. For the SJF and SRT algorithms, what value of $\alpha$ seemed to produce the best results? Support your answer by citing specific simulation results.
4. For the SJF and SRT algorithms, how does changing from a non-preemptive algorithm to a preemptive algorithm impact your simulation results?
5. Describe at least three limitations of your simulation, in particular how the project specifications could be expanded to better model a real-world operating system.

If you are registered for CSCI 4210, feel free to review these questions, but do not submit an analysis on Submitty.