# CprE 308 Laboratory 5: Process Scheduling

Department of Electrical and Computer Engineering
Iowa State University

## 1 Submission

There is no report template for this lab. Instead, submit the following items:

- 20 pts - A PDF with a summary of what you learned in the lab session. This should be no more than two paragraphs. Try to get at the main idea of the lab, and include any particular details you found interesting or any problems you encountered.
- 80 pts - The source code of your algorithm implementations (i.e. a completed scheduling.c).

## 2 Description

This project will allow you to explore four different scheduling algorithms: First-Come-First-Served, Shortest Remaining Time, Round Robin, and Round Robin with Priority. Download scheduling.c from Canvas. Read through the main function. This program simulates process scheduling. You will implement four scheduling algorithms, described in Section 3, in functions marked by TODOs. These functions are called at every timestep of the simulation (one simulated second) to determine which simulated process runs next.

The scheduling functions accept two parameters:

1. The array of `process` structures, which contains a variety of information about the parameters and current state of the processes being simulated. See the code for the list of values contained in each `process`. You may use any of these values in your scheduling algorithm, but **you may not modify any of them**. The code in main will take care of that for you.
2. The current time of the simulation. You will need to use this to determine which processes are "ready," meaning they have arrived and they have not finished. (Processes do not block for I/O in this simulation.)

### 2.1 Output

As the simulation runs, the code in main will print out the following for you:

1. The start time of each process (the time instant when the process *first* runs, not each context switch)
2. The end time of each process (the time instant when the process's remaining time drops to zero)

After all simulated processes have finished for an algorithm, the average turnaround time (time from *arrivaltime* to *endtime*) is also printed. Sample output for the random seed `0xC0FFEE` is available on Canvas. Your implementations of Round Robin and Round Robin with Priority may produce slightly different results, depending on the implementation details, but the average turnaround time should be close to the example.

### 2.2 Grading

You will be graded based on correct output for a set of random seeds. Focus on correct output; you do not need to worry about efficiency of your implementation, fancy data structures, etc. As noted above, your output for Round Robin and Round Robin with Priority may be slightly different from someone else's, depending on some choices made during implementation. You will receive 10 points for code that compiles, runs, produces some output, and terminates. You will receive 20 points for each successfully implemented algorithm.

## 3 Algorithms

### 3.1 void first_come_first_served()

This algorithm simply chooses the process that arrived the earliest and runs it to completion. If two or more processes have the same arrival time, the algorithm should choose the process that has the lowest index in the process array.

### 3.2 void least_remaining_time()

This algorithm chooses the process that has the least remaining execution time left, of those that are ready to run. When a new process arrives, it may preempt the currently running process. On a tie, choose the process with the lowest index in the process array.

### 3.3 void round_robin()

This algorithm cycles through the ready processes, running each for a 1-second quantum (i.e. a single simulation timestep) before switching to the next process. Newly-arrived processes do not necessarily need to go at the end of the "queue"—to keep your implementation simple, they can simply take their first turn according to their index in the process array.

### 3.4 void round_robin_priority()

This algorithm is the same as basic Round Robin algorithm, while also accounting for priority. Assume that a larger value of `priority` (from the `process` struct) indicates a higher priority. The algorithm cycles between only the highest-priority ready processes. For example, if there are processes with priority levels 1 and 2 that are ready, only the processes with priority level 2 are run, either until they have all completed, or until a higher-priority process arrives. If one or more higher-priority processes arrive while lower-priority processes are running, the algorithm immediately begins running the higher-priority processes.