Paul Fretz
Philron Hozier
Ethan Miller
CMPSCI 377 Operating Systems
Lab 02 Part 01
March 16, 2016

Overview
The design of our scheduling solution is implemented across two languages – JavaScript (Node.js) and Python. The choice for a fast yet elegant I/O implementation arose from Node.js's ability to carry out file processing in a few short lines without the need for boilerplate code. Coupling JavaScript's conciseness with its very own lightweight format for storing data known as JSON enabled us to write a parser which churned out JSON for consumption by our Python solution file. Our resulting implementation can be viewed in three parts: parse, pass and compute.

Parse and pass is carried out by `solution.js` and produces JSON files which `algorithms.py` will read and process *directly*. Compute is carried out by `algorithms.py` and executes both FCFS and RR scheduling algorithms.

Usage
From the command line, `cd` into the `Part1/lib/` directory and enter the command `node solution.js`. Results are written to `solution.txt`.

Implementation Details
File: `solution.js`
Central to the Node.js solution is the File System object. `fs` provides simple wrappers around standard POSIX functions which carry out I/O. Three object methods from the Node.js framework which gave us the ability to locate, read and execute asynchronously are `fs.readdir`, `fs.readFile` and `execFile` of the child process library. Within our implementation, these methods sit inside specialised functions and solve bits of the parse, pass and compute problem. `exec` uses `fs.readdir` to collect information on the content of the current working directory. `exec`'s callback passes each tracefile within the directory to `construct_json`. `construct_json` uses `fs.readFile` to read the tracefile at the cwd path. `construct_json`'s callback takes the parsed json and the tracefile's name. This callback is known as `pass`. `pass` uses a method of Node's child process library, `execFile,` to spawn a new Python process. It invokes the Python binary with two arguments -- `algorithms.py` and the parsed JSON. As soon as `solution.js` hands off JSON to `algorithms.py`, `algorithms.py` begins execution. The programs will run concurrent until execution of either parsing or computing is complete. The asynchronous nature of the script allowed us to write functions in a way that didn't rely on linear executions. Scenarios such as, "if a trace file within the current working directory is currently being parsed, begin reading a new trace file from the directory" is where Node.js excels**. The event-driven framework allows functions to continue

under I/O so the process never blocks. In essence, once JSON is created and passed as an argument parameter, `algorithms.py` automatically begins*.

*Our implementation internally bridges the two solutions together. To run, enter `node solution.js` on the command line.
Note: Dependencies include Node.js (v0.12.5) and Python (2.7).
Tested both under Windows 10 (using Cygwin) and *NIX (Mac OS X 10.11.2)

File: `algorithms.py`
The file `algorithms.py` holds our python implementation of the First-Come-First-Serve and Round Robin algorithms. Simply put, it reads in job data from the JSON files that solution.js created, then carries out the algorithms on the given jobs and records its findings. For both FCFS and RR, the jobs are stored in multiple queues (two queues for FCFS, three for RR). We also keep track of a variable called "processing_time" which represents the system timer, and is used to compute wait time.
The FCFS approach is as follows: dequeue a job. **If** processing_time is greater than the job's start time (this job has arrived and been waiting), record how long the job has been waiting, then increment processing_time by the job's length and add the job to the finished_jobs queue. **Else** the processing_time is *less than* the current job's start time, and in a real scheduler, this job would not have arrived yet. Thus, its wait time is zero. In moving on to the next job, we increment processing_time by the job's length *and* by the difference between processing_time and the job's start time.
This is perhaps best illustrated by an example: let's say processing_time is only 1. The current job's start time is 3, and its processing time is 5. We must increment processing_time by 5 for the job's length, plus 2 because two time slices go by before the job gets processed. Thus, after processing the current job, the processing_time variable will be 8.
Next is our RR algorithm. This one is fairly straightforward and performs stepwise operations in the same way a real scheduler would. It utilizes three queues: waiting_queue, current_jobs, and finished_jobs. They are: all jobs which haven't arrived, the jobs currently being processed, and the jobs that are finished processing, respectively.
The algorithm operates within a loop, which can be simplified as follows: while there are still jobs to process, check to see if the next job in waiting_queue is ready (its start time matches the timer). If so, append it to the current_jobs queue. Then, decrement the length of the job in the front of the current_jobs queue (process it by one time slice). If it is finished processing, dequeue it and add it to finished_jobs. Next, increment the wait time of all other jobs in the current_jobs queue. Finally, rearrange the queue in accordance with the round robin idea: current_jobs.enqueue(current_jobs.dequeue). In other words, we dequeue the head element off of the queue and append it to the end of the same queue, in order to rotate the jobs as they are processed.
The output of these two functions is the average wait time of each file under both algorithms, printed to `solution.txt`.

Implementation Highlights

One highlight of our implementation detail was in how we solved applying an arbitrary scheduling algorithm to a current trace file such that the method did not contain any hard-coded information to run `fcfs()` or `rr()`. The interface of the selection method is as follows: `def select_algorithm(self, file, algorithm)` where `algorithm` is a callback method and `file` is json received from solution.js to which the callback method is applied.