**Program Instructions:**

App.java holds the main method. In the main method, you will find 4 methods within if-statements that can be run. Beware that only 1 method can be run each time you start the program. If you want to run all of them at once, make sure to reset the time variable to 0 between each method. Timeout can also be changed if you want to shorten/extend the trial run.

To go over all the files briefly: App is where the new processes are generated and where the running logic of each scheduling method is housed. Each of these methods holds a FCFS object (poor name, I know), which keeps track of 2 queues, an overall throughput calculator, and a printout method. The 2 queues hold (respectively) a list of upcoming processes, and a list of completed processes (for the report). The queues are implemented with nodes rather than an array, so you can change the limit to how many processes you want to feed ‘scheduler’ at will. The queue class also holds 2 unique methods: the first is for finding the smallest service-time & disk-time of all the items in the list (and bumping it to the front), the other does the same but instead uses the HRRN algorithm for finding the largest ratio. Last but not least, the Process class holds a bunch of ints. Num is simply a ID number (unique to each process), service time is randomly generated between 4-12, disk time is RNG’d too between 0-2, and then we have the timestamp vars that are calculated as the App.java logic runs. Finally we have a few calculator methods that are used in our FCFS report method to give you the readout after the simulator finishes.

*P.S. – To see the throughput of the entire list, you’ll need to scroll up past the 70-80 processes to the very top.*

**First Come First Serve**

Compared to other algorithms, the throughput is quite poor at about 0.11 processes/second. Starvation is not possible in FCFS as long as a process is completable (i.e. doesn’t have an infinite loop). This is because all processes are computed in the order they arrive, meaning any queue’d up process will be computed so long as every process in front of it can be successfully completed. Context switching is minimal in FCFS, as all processes are loaded into memory only once, and then restored into memory only once. The algorithm is not particularly fair. Large processes will get a disproportionate amount of CPU scheduling, and due to it’s non-preemptive nature, important processes cannot be bumped into the CPU if necessary. Other than that, the turn-around time is roughly between 24-36 seconds, with the turnaround to service ratio averaging between 3.0-5.0.

**Round Robin**

The throughput for Round Robin is slightly better at about 0.12 processes/second (using an 8 second quanta). Starvation is not possible regardless of the inputs since each process is allocated an equal amount of time to run before automatically switching to the next process. Context switching has a higher impact here than in FCFS, since processes have to unpack/repack every single quanta (and most processes take 2 quanta to complete). This algorithm gives more of an equality of outcome to the processes. It allows smaller processes more of an opportunity to complete themselves at the expense of the completion times of larger processes. Response time varies heavily, with some (assumably smaller processes) having response times in the teens, with others jumping into the high 30s and low 40s. The turnaround time varies likewise between the low 20s and mid 40s. Finally, the ratio seems to be somewhere around 4.0-5.0.

**Shortest Remaining Time**

For SRT, throughput seems to have increased pretty well, to 0.15 processes/second. Starvation (of larger processes) is possible with SRT due to the fact that it searches for the smallest process every time a new process is completed or created, meaning that a stream of smaller processes could block a large but important process for a long time (one process had a 358s response time, and a 368s turnaround time; resulting in a ratio of 40.9). Ratios overall are quite low for SRT, with most between 1-4 (albeit with some **significantly** higher than that). This makes the algorithm incredibly unequal, as small processes will always be favored over large processes. Context switching is certainly a big issue here (as in Round Robin), however in terms of it’s effects on overall performance, it seems not due to the algorithms insistence on clearing out as many (small) processes as possible. For response times for small processes can be in the single digits, with turnaround times not far behind. However the response and turnaround times for large processes are greatly exacerbated, resulting in 2 bell curves: one very quick, and the other absurdly slow.

**Highest Response Ratio Next**

For HRRN, throughput seems to be in the middle, at 0.13. The algorithm was explicitly created to avoid starvation, by taking into account the time on the scheduling list when considering what process to run next. Response & turnaround times seems to be pretty consistently between the high teens and low 30s throughout, and the ratio is very tight at between 4.0-4.5, with only a few outliers. This means the algorithm is remarkably equal, with all processes returning within a reasonable amount of time. Overall, HRRN seems to be the most reliable (in terms of starvation) & equal algorithm of the four given.

**Proposal for Improvement with SRT**

Since SRT seems to be the weakest algorithm, we could easily improve it by adding in a simple weight equation (similar to HRRN). This equation would be designed to as to prefer smaller runtimes, however it would provide **some** weight to old processes to ensure they get ran within a reasonable timeframe. For instance, since we came across some massive processes that literally took 5 minutes to run, we could say that any process **must** be completed within 60 seconds of being created. Meaning a (for example) 20-second-long process must start at second 40 at the furthest, and therefore must be completed, before we reach our set time limit for each process in the list. This would improve the equality only somewhat, while still retaining our preference for small processes.