Josiah Norton
20251022

I started this assignment by reading through the provided base code for scheduler.c and identifying the pieces already written by the instructor. My first step was to make sure the teacher's code compiled and ran, then I began making my part.

The instructor specified that we must include a global constant seed value for our random number generator in an announcement, so I added the constant SEED_VALUE = 200. This constant ensures that the randomOS function always reads from the correct position in the random-numbers file.

For process representation, the _process struct came pre-made with some stuff and I used it to hold all of the attributes needed for each job. Beyond the given fields A, B, C, M, and processID, I extended it to track state (UNSTARTED, READY, RUNNING, BLOCKED, TERMINATED), burst times, waiting times, and finish times. These additions allow the simulator to measure turnaround time, waiting time, CPU utilization, and I/O utilization directly as required.

For input handling I reused the provided getnext helper to parse numbers from the input files. I then wrote the readinp function, which reads all process definitions, initializes their fields to default values, and stores them in an array. This ensures that processes are zero-initialized with good defaults for timers and burst counters, which I used in all my functions.

For the ready queue implementation I wrote a Queue type with functions for initialization, push, pop, and resizing. I chose to use a circular buffer, which ensures FIFO order is preserved even as elements are added and removed. I figured I would need this for First-Come-First-Serve and Round Robin scheduling to work correctly.

I then implemented the three required schedulers. First-Come-First-Serve adds processes to the ready queue in arrival order. When the CPU is free, the next process is popped and begins execution. CPU bursts are generated with randomOS. If the process finishes its CPU time, it terminates. Otherwise, it blocks for I/O equal to lastBurst times M. Waiting times are incremented each cycle for whichever processes in the ready queue.

Round Robin is built on top of the FCFS logic but with time slices. A running process is yoinked by the cpu if it uses up its slice before finishing its CPU burst, and is re-queued in the ready list. This required tracking both curCPUBurstLeft and the remaining slice. I made sure to handle three possible cases each time: process terminates, blocks for I/O, or is yoinked.

Shortest Job First uses a similar cycle-based simulation, but the ready queue selection is different. Instead of FIFO order, I always select the process with the smallest remaining CPU time. I scan the ready queue each time the CPU is free, pick the one with the smallest remaining CPU time, and rebuild the queue without that one. Also it included a tie breaker instead of just pushing in order since the assignment asked for it.

The random number logic stayed very close to the teacher's provided code. I used randomOS(upper_bound, process_indx, file_ptr), which reads from the random-numbers file and returns a CPU burst between 1 and the given bound. The only change was ensuring the global seed offset (SEED_VALUE = 200) was honored.

I did not really change any of the print statements since the teacher provided those and they worked well enough with the code that I wrote. As a side note, I noticed that the given examples never actually change the upper limit of the burst so they have really similar outputs for almost every test case. I made another test case where I made sure they would get different outputs, but I am not including that in the files since it was not asked for.

Finally, in main, I parsed the input file into process structs, printed the original and sorted input, opened the random-numbers file, and selected which scheduler to run based on the second command-line argument. The default run is fcfs. After running the simulation, I printed both process-specific and summary statistics, as the assignment asked. I also feed pointers, because that is a good thing to do.

The default run is fcfs, and adding rr or sjf at the end of the command line runs the other cases. I tested my code on the example files and it ran correctly