Assign 5 Report

Aaron Perez

August 2, 2020

The language selected for assign5 was java, threads were implemented using the runnable interface. Three threads were created to handle program execution: a file read thread, a CPU thread, and an io thread. The organization of the process was handled by the process control block object in PCB. Java Synchronization was accomplished through the OS object which all the created threads access. Overall, the assignment proved an interesting challenge, and it was refreshing to take system-level principles from c and apply them in a different language and framework.

At the design level, the thread objects handled all vital aspects of execution in and of themselves using PCB and OS only as containers for shared elements to simplify synchronization in the critical sections. Critical section syn- synchronization is handled with synchronized blocks in unsynchronized methods to create mutex locks on the object being used. This prevents two threads from using the object, ensuring a smooth program ow. The critical total processes counter, which increments as file read creates new PCBs, is additionally set to volatile to make sure that it's stored in the jvm main memory instead of one of its virtual CPU caches. This is done to ensure that CPU and io only stop processing data when there are no more processes to operate upon. Problems could, admittedly, still exist with this type of synchronization especially in the addition of PCBs to the ready queue. By only adding a new process to the queue in the file read thread, and not interrupting the methods in the other two threads on this event, processes may have lower wait times than they waited for.

Through analysis of the processing, with the -debug argument, shows that the algorithms are all correctly implemented, there is little change from one to another with the test data. This

upsetting result is probably due to the simplicity of input2. Txt and the speed at which file read processed all the data. Variations in the results are likely due to the different speeds at which it creates new PCBs. This causes the waiting time of processes other than the _rst to change, as they will not be in the queue - and therefore wait - only after they've been added by file read. A sampling of results is available in the directory, generated with the sample input given. More accurate, and realistic, data could have been created with more varied inputs but such testing was not able to do due to time constraints.

While cpu scheduling was implemented to the speciation's requested, io scheduling was left with a simple fifo pipe. On a design level, object-oriented principles were attempted with popularization. Cpu bursts were given general use method for non-preemptive procedures - fifo, sjf, pr - and another for preemptive - rr. This allowed the codebase to be kept down to size, and errors to be noticed and _xed far easier. The correctness of these implementations can be checked by appending -debug to the end of command-line arguments, which provides detailed information on burst ordering within the synchronized blocks.

After waiting for all threads to _nish execution, the initial thread then begins to process the data. Looking through the list of completed processes, the main gather data from the pcb objects and then calculate the statistics for the run. All numbers generated at the end of the execution report are, of course, subject to rounding errors.

Thanks for the help on the implementation go to Herman Miller, who helped with several conceptual problems. Most notably, the idea to encapsulate critical data in the os object to allow easy passing and communication between the executing threads were due to his critique after attempts were made to store everything in progressively longer lists of constructor arguments.

Less critical to execution, but still of great help, was logic in parsing the command linear-arguments before thread execution.