Assign 5 Report

Sean Hassan

April 29, 2017

The language selected for assign5 was Java, threads were implemented using the Runnable interface. Three threads were created to to handle program execution: a FileRead thread, a CPU thread, and an IO thread. Organization of the process was handled by the Process Control Block object in PCB.java. Synchronization was accomplished through the OS object which all of the created threads access. Overall, the assignment roved 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 in order to simplify synchronization in the critical sections. Critical section 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 smooth program flow. The critical totalProcesses counter, which increments as FileRead 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 FileRead thread, and not interrupting the methods in the other two threads on this event, processes may have lower wait times than they actually waited for.

Though analysis of the processing, with the -debug flag, 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 input1.txt and the speed at which FileRead processed all of the data. Variations in the results is likely due to the differing speeds at which it creates new PCBs. This causes the waiting time of processes other than the first to change, as they will not be in the queue - and therefore wait - only after they've been added by FileRead. A sampling of results are 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 done due to time constraints.

While CPU scheduling was implemented to the specifications requested, IO scheduling was left with a simple FIFO pipe. On a design level, Object-Oriented principles were attempted with modularization. CPU bursts were given a general use method for non-preemptive procedures - FIFO, SJF, PR - and another for preemptive - RR. This allowed the code base to be kept down to size, and errors to be noticed and fixed far easier. 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 finish execution, the initial thread then begins to process the data. Looking through the list of completed processes, main gathers data from the PCB objects and then calculates the statistics for the run. All numbers generated at the end of execution report are, of course, subject to rounding errors.

Thanks for help on the implementation go to Andrew Sanetra, 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 line arguments before thread execution.