Project 1: Thread Scheduler

Benjamin Zydney & Yifan Lu
School of Electrical Engineering and Computer Science, Pennsylvania State University

CMPSC 473: Operating Systems Design & Construction

October 10, 2022

PennState College of Engineering | ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Design of the Scheduler + Implementation Choices / Restrictions:

When we began this project, we initially tried to implement FCFS and `cpu_me()` without using any queue data structures, instead opting to use a `get_max_priority()` function. Though this design worked for simple CPU-only inputs, such as `input_0` and `input_1`, it became clear as soon as we began implementing `io_me()` that this would be extremely difficult and inefficient. `get_max_priority()` was highly dependent on the global clock only changing with CPU inputs, which is not the case for more complex examples.

As such, we decided to have the framework of our scheduler be a priority queue, implemented as a singly-linked list. In addition to the standard `push()`, `pop()`, `peek()`, and `init()` functions, which were implemented exactly as we've done in previous classes, with the addition of mutex locks, we used a `schedule()` function to decide which thread should be scheduled first, allowing us to re-use the same code for all three scheduling algorithms. For FCFS and SRTF, `schedule()` organizes threads by `arrival_time` and `remaining_time`, respectively, with the tiebreak being `tid` (as specified in the project description). For MLFQ, we used additional helper functions `schedule_mlfq()` and `update_mlfq_info()`, which allowed us to organize threads into a queue of queues.

Then, as to not clutter the `interface.c` class and to allow the reusing of code, we implemented the bulk of the signaling operations in helper functions in `scheduler.c`. `signal_cpu()` checks the thread to be ran by using `peek()`, and then `pop()`s it if exists. Then, if there's at least one thread in the CPU queue, it signals the CPU using `pthread_cond_signal()` and waits for its turn. `signal_io()` is essentially the same implementation, but since I/O is only implemented using FCFS, only that case is considered. To handle time, we used a `global_clock()` function that looped over a `global_time` global variable, calling the threads when it was their turn. Finally, the `wait_until_turn()` function has the given thread wait on a condition variable that will be triggered by `global_clock` when the thread should be run.

The main functions in `interface.c` are pretty straightforward, as they just perform calls to the aforementioned helper functions, with necessary mutex locks to avoid race conditions. More importantly, we used an array of active threads and tracked the total number of threads vs. the number of threads remaining, which allowed us to handle threads arriving out of order. To track semaphores, we decided to initialize an array of semaphores of size 10 (= `MAX_NUM_SEM`). The `P()` operation waits until `s == 0`, and the `V()` operation signals the semaphore (i.e. `s = s + 1`). In both functions, we locked the semaphore critical section to avoid any race conditions.

Unimplemented Issues / Problems with Implementation:

After testing the program with the sample inputs and with some custom test cases that we created, which included multiple corner cases, we were unable to find any unimplemented issues. However, there were a handful of problems that we encountered in the process of designing the project, but have been fixed since then.

As mentioned in the previous section, we initially decided to implement FCFS and `cpu_me()` without using any queue data structures. This heavily restricted our design, since it would be extremely difficult to implement I/O operations. We were able to fix this problem by switching our design to a priority queue framework.

The most challenging problem we dealt with was a race condition within our semaphore implementation. While testing input 11, we noticed that, sometimes, we would get the wrong output, with the Gantt chart output outputting extremely large time values (i.e. in the millions). We wrote a bash script (see `tester.sh`) that ran each input 1000 times to confirm this issue. Essentially, when a pause was issued, it would signal being ready before checking the wait condition. Because it didn't have to wait, it would sometimes manage to be slower than the global clock, since they weren't mutually exclusive. As such, the global clock would think that all of the threads were still active, since `P()` hadn't declared itself inactive (while waiting for a signal it was active). Once we were able to identify the underlying cause of the problem, we fixed this by checking if the thread was going to wait, and if it wasn't, setting the thread to inactive before signaling.

The final problem we ran into was with custom inputs input_d, input_f, and input_g outputting the wrong Gantt charts. For input_d, T1 and T0 were swapped, so we initially suspected that the problem had to do with arrival times being out-of-order. However, our implementation accounted for this in other inputs. As it turns out, the problem was simply that we accidentally used `ints` for our arrival times instead of `floats`. Inputs f and g had a similar problem, but with semaphores. We simply changed the return time in `V()` to return the ceiling of the current time, rather than the current time (which rounded down).

Distribution of Workload:

The workload for this project was divided fairly evenly. Most of the work was done either together in-person or over text, which ensured that only one of us was committing work at a time so that no work would be lost or overridden by a merge. When we ran into errors that we needed to debug, such as the ones mentioned above, we would both talk about the potential causes of the errors and possible solutions. We would then both try to fix the errors, and if either of the proposed fixes worked, we would commit that change.