Thread Scheduler

1. Design of Scheduler

- We begin by first implementing a queue data structure to manage the queues. We initially thought that we might require this, as said by the Professor. We added functionalities according to our needs.
- Whenever we must decide about which thread should be executed on the CPU, we call a 'select_next_thread' function. Internally, this function calls a specific function written for the respective scheduling algorithm decided using 'scheduler_type'.
- We used condition variables. In some cases, we used individual variables for every thread, and in some cases, we used global variables.
- We used 2-time variables: global_time and global_io_time. Global_time is for CPU_me, P, and V. Global_io_time is for IO_me.
- Used a mutex_lock named scheduler_mutex to avoid race conditions in functions that are called by multiple threads. Also, used an array of semaphores, which saves information about the number of threads blocked on this semaphore and has a condition variable to signal thread which is blocked for this semaphore, and has a mutex lock to avoid race conditions between P and V.

2.1 Common Architecture

- We wrote a function called 'wait_till_all_threads_arrive'. This function waits till all the threads arrive. As soon as a thread arrives, it increases the count of arrived_threads by 1. Once arrived_threads == active_threads (number of threads which didn't call end_me), the last thread broadcasts a signal to all the threads waiting.
- In CPU_me, we are adding all the threads that called the function before the barrier written above, so that we have all the threads required to decide. In all other functions: P, V, io_me, we have added the above barrier at the start.
- In CPU_me, we decide which thread needs to be executed. If the thread in which the decision is taken is different from the next scheduled thread, we make the current thread sleep and wake the next scheduled thread. Otherwise, the current thread returns.
- In IO_me, if the current thread is on the CPU, it must give it up before blocking for I/O. Furthermore, if the thread in which the decision is taken is different from the next scheduled thread, we make the current thread sleep and wake the next scheduled thread by signaling. The current thread enqueues itself for future scheduling and blocks itself for now. When it gets a signal, it returns from the function.
- In P, if the semaphore is available, it immediately returns. If not, it signals the next thread to be scheduled and itself goes to sleep till the V signals it.
- In V, if any thread is waiting, then it signals it and provides the wake-up time. If no thread is waiting, it increases the semaphore for future use and returns.
- In end_me, the thread terminates and signals the thread to be scheduled next.

2.2 FCFS

- For this scheduling algorithm, we check which thread has the earliest ready_arrival_tick. We then set current_cpu_thread to this thread and schedule it.
- In case the threads have the same ready_arrival_tick, then the algorithm decides based on the lower tid.

2.3 SRTF

- For this scheduling algorithm, we first create a temporary ready_queue in which we filter out according to 'ready_arrival_tick <= global_time'. Here, ready_arrival_tick is the time when the thread is added to the ready_queue.
- If no thread passes this filter, it means that global_time is lagging and needs to be advanced. So, we find out which thread arrived first in the ready_queue using the FCFS algorithm. We advance the time to the ready_arrival_tick of the selected thread and schedule it.

2.4 MLFQ

- For this scheduling algorithm, multiple queues are required, in which we can add threads according to the time quantum left for that thread. So, we created a set of 5 queues.
- Here again, we filter the threads for every queue, like the SRTF. We try to schedule the thread that has the highest priority.
- Like the SRTF algorithm, if no thread passes the filter, then we find out the first arrived thread, advance the signal, and schedule it.

4. Key Implementation Choices / Restrictions

- Implemented all queues as arrays because the maximum number of threads was given as 2^12. It also helped in the easy addition of the functionality of enqueue/dequeue. We were first thinking of using a linked list, but the implementation was tricky, and considering the given constraints, an array made more sense.
- We tried using a per-thread condition variable for blocking all threads at the start of functions and before taking the scheduling decision. And then using 'pthread_cond_signal' to signal them. However, it was interfering with our wait and signal algorithm between different functions in the interface.c file. Therefore, we created a global condition variable and used broadcast. We further managed the thread handling once decisions were taken. As this worked for FCFS, we further implemented the same for SRTF and MLFQ.
- We used a per-thread condition variable to manage wait and signal in all the functions. For example, if T0 is running and it decides that T1 should get executed, then we make T0 sleep till it gets a signal, and then T1 proceeds.
- Maintained separate global time variables for CPU and I/O to emulate asynchronous device behavior.
- Also, maintained different queues for IO from ready_queue and mlfq.
- Designed temporary per-level queues in MLFQ to handle ready threads (with ready_arrival_tick <= global_time) for realistic scheduling. For FCFS and SRTF, we used a queue with the name ready_queue, and for MLFQ, we created a set of 5 queues with different quantum sizes.
- Used semaphores for careful synchronization between P and V to ensure that V completes before P resumes.
- For debugging, we used print state to track the thread conditions and states. I think we were more comfortable with print statements compared to the GDB debugger. But we tried both.

5. Known Limitations / Not Implemented

As much as we know, we are not aware of any limitations. We tried to cover all the edge cases.

6. Testing Summary

- We tried running all the inputs given in the GitHub repo for FCFS, SRTF, and MLFQ. All the cases are passing, and we also tried running multiple times for all inputs. The outputs are consistent.
- Further, we tried creating inputs for 1000 and 5000 threads. We are getting outputs.
- Also, we tried to run inputs from the project details and from recitations, where we were successful in running all the edge cases.