Design and Implementation of a Multi-Threaded Scheduler

Introduction:
The object of this project is to design a thread scheduler which can manage multiple threads, and it also supports First-In-First-Out (FIFO) and Shortest Remaining Time First (SRTF) scheduling policies. It should include a single CPU core system and two I/O devices. The scheduler should manage the threads to execute operations and ending threads. Those processes and results will be demonstrated as Gantt Chart based on the given expected output.
The scheduler is designed to be consisted by following components:
interface.c: Contains functions 'cpu_me()' and 'end_me()', the functions are called by the threads.
scheduler.h: We struct queue variables to schedule tasks such as 'QueueNode' and 'Queue' and functions like 'enqueue()' and 'init_queue()'. Those can help the code to manage the thread, other data structures and function prototypes.
scheduler.c: Implements main scheduling logic and functions. The queue management functions area used to manage tasks. Functions such as 'get_earliest_arrival_tid()' and 'get_shortes-t_remaining_tid()' are used to handle scheduling of threads and check if a thread is in the queue.

In task 1, the FIFO scheduling policy means that all tasks should be processed based on their arrival time. Thus, the arrival time should decide the order. In our thought, the scheduler maintains a ready queue, and each task is executed in the order it arrives. Thus, the first-in-first-out queue 'ready' is used to manage the tasks in the order they arrive. When a new task arrives, it is enqueued at the end of the queue by function 'enqueue()'. About how to schedule it, we thought about that when CPU becomes available, the scheduler dequeues the task at the head of the queue and starts executing it. Since the implementation of this policy is straightforward, compared with SRTF, as tasks are executed in the order they arrive, we don't need to consider their burst time or remaining time.
In specific, the scheduler assigned a queue of tasks using the structure Queue and QueueNode. The tasks were executed in the order of when they arrived. Then the scheduler will put the task at the head of the queue and start the execution of that first task when the CPU becomes available. This was implemented in the 'cpu_me()' function in interface.c using the 'enqueue()' (lines 18 - 34) and 'dequeue()' (line 36 - 56) functions defined in scheduler.c.

In task 2, the tasks should be processed based on their remaining time due to SRTF policy, which means the task with the shortest remaining time being prioritized. This policy requires the code to construct another queue of tasks, sorted by their remaining time. Task priority queue: A priority queue is used to manage the tasks based on their remaining time. When a new task arrives, it is inserted into the priority queue according to its remaining time. Due to different policy, when the CPU becomes available, the code dequeues the task with the shortest remaining time from the queue and starts executing it. If a new task arrives with a shorter remaining time than the current task which is executing right now, the code will change the current task to the new one and starts executing the new task. The implementation of this policy is more complex than the FIFO policy, as it requires the scheduler to manage tasks based on their remaining time and support preemption. That means the code needs to record the remaining time of all tasks and the new task, then compare two times to get the shorter one. The conceptual strategy of this task is we need to maintain a priority queue of tasks sorted by the remaining time of each task, and it should support preemption when a new task arrives, and

its remaining time is shorter than the task that is currently executing. The queue used Queue and QueueNode date structures, which is defined in 'scheduler.h'. the policy was implemented by function'get_shortest_remaining_tid()' (line 57 - 81) in 'scheduler.c' and the function 'cpu_me()' (line 13 - 49) in 'interface.c'. The scheduler also support ending thread using the 'end_me()' function in 'interface.c', which marked the thread as ended in the queue and signaled other threads waiting on the condition variable to continue execution. In our code, that was implemented using the 'pthread_mutex_lock()', 'pthread_mutex_unlock()', and 'pthread_cond_broadcast()' functions in interface.c(lines 57 - 66).

However, our code still has some issues. Intuitively speaking, we can't pass the consistency test since sometimes the output will be different from the expected one for task 2. We guess that may cause by reasons below:

1. the code in 'interface.c' includes some structural failure, and those causes the queue can't work correctly.
2. One or more functions in 'scheduler.c' includes some mistakes, so that the queue can't work correctly to manage tasks, or the scheduler can't assign different tasks based on their remaining time properly.
3. Our approach doesn't include anything about contention for devices and that may make significant changes to the code and how it works.

About other unimplemented Issues, we think that the current implementation of the scheduler still contains the following issues that we may not be able to solve:

1. We didn't finish task 3 and task 4, which decides that the scheduler will not support I/O operations and it can't manage I/O devices.
2. The current scheduler we have in the code doesn't support preemptive scheduling policies so the time on a thread for tasks may be limited.

In conclusion, we worked together in parallel on task 1 and task 2. Ran Teng worked about some details in both tasks, such as modifying the 'scheduler.h' to utilize the date structure 'Queue' and 'QueueNode', and modified task 1 to pass the consistency test. Zhengbang Fan worked on the main body of the code in 'scheduler.c' and 'interface.c' such as functions 'cpu_me()', 'enqueue()', 'dequeue()' and 'get_shortest_remaining_tid()'.

Also, we designed and implemented a scheduler that supports FIFO and SRTF policies. The scheduler manages the execution of CPU bursts, I/O operations, and thread termination. The current code and function have been discussed, along with the unimplemented issues that could be addressed in future work.