P2 - Thread Scheduling Report - Travis Stauffer and Michael Sullivan

Conceptual Strategy

The bulk of the work done in P2 was done in scheduler.c and interface.c with additional function and struct declarations in scheduler.h.

Scheduler.c/.h

Early on in our design process, we decided that our implementation of thread scheduling was to be done using a queue in the form of a linked list (this linked list and its functions all have "fifo" in the name even though it performs SRTF calculations now. This is because we were only focused on task 1 when initially naming these functions). The data structures used in this queue can be found in the scheduler.h file in lines 16 to 32 where we declared structs for the queue itself and its entries. Additionally, we decided to design a few queue operation algorithms in scheduler.c, the two most important ones being enqueue and dequeue. The enqueue function found at line 35 simply places an incoming entry at the tail of the queue. The dequeue operation found on line 52 works by finding the entry with the specified tid and proceeding to evict the entry while maintaining the integrity of the queue pointers.

Outside of these queue altering operations, our implementation of interface.c led us to recognize the need to implement a few auxiliary functions. The function "init_fifo" is used in line 25 of interface.c in init_scheduler to initialize the queue structure by setting the head and tail to NULL. The function "isEmpty" is used within the enqueue function in scheduler.c at line 42 to check if the queue is empty in order to set the head and tail pointers properly. The function "find" will return 1 if the specified TID is found in the queue and 0 otherwise to check if a thread is enqueued yet or not.

Next, we implemented two search algorithms for FIFO and SRTF. find_fifo, found on line 83, searches through the queue, storing the thread with the earliest arrival time inside of the "shortest" fifo entry. After the while loop terminates, the stored fifo entry is returned and used to find the next thread to be scheduled for task one. A similar function, find_srtf, was created to find the next thread to be scheduled for our implementation of task 2. This function works in a similar way to find_fifo, except instead of finding the thread with the earliest arrival time, it finds the thread with the shortest remaining time that is able to be scheduled by checking the global time and the thread's arrival time. Since we finished task 2, we only use "find_srtf" in cpu_me.

We partially implemented task 3 of the project and have multiple functions and structs specifically for task 3. In scheduler.h we have two more structs between lines 50 and 66 for a new queue for IO bursts only. We decided to make another separate queue for IO bursts because of the fact that IO can happen at the same time as CPU if it is with two different threads. These structs have basically the same format as the ones stated above for CPU bursts, but the entries have different parameters, such as "device_id". Since this queue is very similar, many of the functions are too. The functions init_io_1, isEmpty_io_1, enqueue_io_1, dequeue_io_1, and find_io work exactly like the ones stated above for CPU bursts and have the same overall layout and purpose. The only differently named function for IO is "find_first", defined at line 229, which is very similar to "find_fifo" in that it finds the first IO burst to deal with since IO is dealt with on a FCFS basis.

Interface.c

The biggest challenges with our implementation of interface.c were maintaining consistency of implementation, as the functions in this file were being continuously called by each thread simultaneously, and dealing with weird corner cases once we got the basic cases working.

Globally, we decided to initialize a fifo queue, define global_time, global_thread count (which is a variable used to track if all threads have arrived yet), and a variable called threads_to_complete to specify the remaining threads to be completed. Additionally, we initialized two variables from the pthread library cond, and lock. We also initialized an IO queue, variables to keep track of the amount of threads in IO and their time, and a condition variable for IO only.

The initialization function is fairly simple. Here we initialize and allocate memory for our scheduling queues, and set the rest of our global variables. Please note that while global_thread_count, and threads_to_complete are initially set to the same value, the former is used to keep track of the threads that have not been put in the queue, and the latter is used to keep track if the threads that have not completed their run on the cpu. Global time is also set to 0 here to start.

Despite the minimal amount of code, the implementation of cpu_me occupied the bulk of our time. This is the function that is being simultaneously called by multiple threads, so we had to make sure that we were correctly blocking threads from interfering with each other's data. We decided to lock and unlock at lines 43 and 121 respectively. This put the entire function within the a lock. We started by waiting for all the threads to arrive between lines 44 and 53. We thought to do this because the threads could arrive in any order, so we need to start by waiting (cond_wait). When global_thread_count hits 0, we know to stop waiting and all threads are enqueued. The if statement at line 59 checks for the corner case of the first thread not starting at time 0 and sets global time accordingly. We then have to find the SRTF entry to deal with first. We call the find_srtf function at line 68 and compare it to the TID of the thread currently trying to use cpu_me. Once they are the same, we break out of the loop. While they aren't the same, we keep waiting until they are and signaling for other threads to try and see if they are the one we are looking for. Once we break out, we have found the entry to work with. The if-else statement from lines 98 to 110 deals with incrementing global time and another corner case. The if statement will increment global time when the remaining time is not 0 since we ignore the return when remaining time is 0. If we added 1 every time, the global time would be 1 too high at the end of every thread. When the remaining time is 0, we choose to dequeue it here and add 1 to the global thread count. This takes care of the corner case of multiple CPU bursts across multiple threads. If we kick out the thread and tell it that we aren't done yet and it needs to be added back in for more CPU bursts, it will work for this scenario. On the next loop, it'll see we need to wait to add a new burst to the queue and it will have the proper times for the next burst to compare to the others in the queue. After this, we decrement the remaining time of the current thread at line 113 to keep SRTF working properly since it checks this every loop. We then unlock and return the global time variable. This works for every corner case provided.

Before explaining io_me, I will explain the end_me function since it pertains to cpu_me's functionality. We originally did a dequeue within end_me, but we realized this wouldn't allow for all corner cases to work in our implementation, which is why we moved the call to dequeue to cpu_me. However, end_me still has a purpose. When we hit the "E" at the end of a thread, we decrement the global thread count by 1. This is necessary because we increment it by 1 within cpu_me and it would deadlock and wait forever in cpu_me if we didn't decrement it when the thread was completely done since it would think a thread needed to be added. We also lock around this function for safety of the program. The threads_to_complete variable doesn't have a use anymore, but it kept track of the threads when we did a dequeue in end_me.

The io_me function doesn't work 100%, but we got some functionality working. If the IO device is 1, we set io_complete to the global time + 2 since the length of device 1 is 2 ticks. We then enqueue the IO and have a function similar to that in cpu_me where we look to see if the thread currently using io_me is the one we want to be dealing with on a FCFS basis. If it is, we dequeue it and return io_complete. We also lock the whole inside of io_me to keep it safe from multiple threads accessing it. We know that io_me doesn't work in all scenarios, but we were able to start it and get it to work for some scenarios.

In the end, task 2 works as it should and we got a start to task 3 to make it partially work, as our focus switched to task 2 with the rubric change.

Breakdown of Effort

We both worked extensively on this project. One of us didn't work significantly more than the other. It is hard to pinpoint who worked on what section because we worked as a team on all of the sections that we were able to complete. We went to office hours frequently together to work on task 1 and 2, mostly working off of one computer and pushing from that one most of the time. We both agree there was an even amount of effort and work put in by both of us throughout this project, as we both wrote many functions individually and as a group, and came up with many ideas individually and as a group when we met up between classes.