Name: Xuhong Lin, Zishan Wei
Professor: Trent Jaeger
CMPSC473

Project 2_Multi-Thread Programming Report

Task 1, 2, 3: Zishan Wei, Xuhong Lin

Introduction

In this project, we used the pthread library, locks, mutex, and condition variables to implement the project about multithreading.

Before the project starts, we define the variables and initializers that we might use.

First, we created a “clock” to record our global time. We then create “IO1_list” in preparation for fifo thread scheduling for Tasks 1 and 3 and create a “srtf_list” in preparation for srtf thread scheduling of Task 2.

Next, We a “create num_thread” variable to equal to “thread_count” for recording the total number of threads.

After that, we created a “curr_num_thread” variable and “io_num_thread” to record the CPU Burst and IO Device entered into their respective link list, respectively.

And an “io1_return_time” variable to record the I/O return.

Moreover, we create “num_mutex”, “cpu_mutex”, “io1_mutex”, and “clock_mutex,” 4 locks, to insure only one thread enters the critical section so that to avoid the race condition.

Here,“num_mutext” ensures that only one thread of “curr_num_thread” and (total) “num_thread” can enter the critical section. Then “clock_mutex” controls the clock time of “cpu_me” and “io_me” functions to again, avoid the race condition.

For the last variable, we create a “srtf_next_thread” variable to define the next signal thread we are going to use in task 2（by using the srtf's thread scheduling）.

After defining and initializing all the variables.

Here is a brief summary of all three tasks we wrote in this code.

First of all, in task 1, we use FIFO (First in, First out) thread scheduling to control the time when CPU Bursts into the Link list and their scheduling order （define CPU scheduling policies）.

Secondly, in task 2, we use another scheduling approach, srtf（Shortest Remaining Time First), to control the sequence of adding CPU Burst to the Linked list and their time of sheduling （defining CPU scheduling policies）.

Third, for Task 3, we modify the “io_me” function, apply the use of io_device, and use multiple locks to control the switching scheduling between cpu_me and io_me.

Task 1

In task 1, we implement the first in first out scheduler in the cpu_me function but we implement task 2 so that the following FIFO scheduler will implement in the io_me function. The overall strategy is that we create a linked list for storing all the coming thread information. The structure is shown in the scheduler. h from lines 19 to 32. When all the threads rush into the cpu_me function, one of the threads will grab the num_mutex lock and enter the critical section. In this critical section which is in the interface. c from lines 99 to 108, we will go to add the thread and its information into the linked list according to its arriving time in increasing order, and for threads that have the same arrival time, we will arrange them according to their tid in increasing order. This adding function is implemented using two while loops with multiple constraints to find the place for the new thread to insert. More details in the scheduler. c from lines 18 to 78. After one thread adds itself to the linked list, we will check if the current number thread that is already in the list is the same as the total number thread. If they are not equal, then the thread will wait and release the lock, so the next thread will come in. Eventually, the last thread that comes in will pass the condition, and we decide to that this thread to wake up the thread that should run next. Since we arrange the list at the first time, the thread that will run next will always be the head node of the linked list. After we decide which thread to run, we will go to increase the clock time depending on different situations. Then, once the thread runs out of its CPU burst, we will kick it from the linked list. We use the function in the schedule. c from lines 166 to 181. Then, we return to clock time. The overall code is implemented in interface. c from lines 112 to 157. There are two places we will go to signal the next running thread. One is after the adding thread step and the other one is in the end_me function. The reason we doing this is that once the new task comes in. It can either be CPU burst or end me. So we signal the next thread here. And in the end_me function, we decrement the total number of threads to tell all the threads that there is one thread left. The end_me function is in the interface. c from lines 159 to 173.

Task 2

In task 2, we implement the shortest remaining time first in cpu_me. The overall strategy is similar to fifo. We also first add all the threads into the linked list （not the same as the FIFO one) one by one, but the way that we add the thread into the list is in an srtf manner. We arrange the thread according to their increasing arrival time. And for those threads who have the same arriving time, we arrange them in increasing order of their remaining time. For those who have the same arriving time and remaining time, we arrange them according to their tid. This adding function is in the schedule. c from 80 to 141. And then we check the condition of the current number thread and total number thread to wait for the thread as needed. The last thread added to the list will pass the condition and then signal the next thread, we have a variable call srtf_next_thread to keep the next running thread, and we will find the next thread by using the function call find_next_thread in the scheduler. c from line 143 to 153. Then, we will go to signal the next running thread by using pthread_cond_signal(). We also have another condition to filter out the threads that should not run after we signal the next thread. The condition is simple to check is the thread’s tid is equal to the srtf_next_thread’s tid. If true, then the thread can pass. Then, we will go to increment the clock time. And kick the thread that runs out of the CPU burst by using the function remove_thread_from_srtf() in scheduler. c from 183 to 212. In the end, we return the clock time.

Task 3

In task 3, we will implement the concurrency of CPU and IO functions. We implement this by using different mutex locks. We have cpu_mutex to guide the cpu_me function and io1_mutex to guide the io_me function. To avoid the race condition, we use num_mutex to protect the current number thread and total number thread variable so that only one thread can go to the critical section to change them or access them. And also, we use the clock_mutex to protect the global time variable, clocks since there might have IO and CPU access clocks at the same time. Then, the final thing is that we are going to signal both IO and CPU next running thread in cpu_me, io_me, and end_me funtion to make sure that threads can run at the same time.