Project 2 - Thread Scheduler

CMPSC 473

Xinkai Wu, Kaile Ying

Xinkai Wu: Scheduler

Kaile Ying: Queue

This is a report for Project 2 in CMPSC 473. Our goal is to construct different schedulers and use pthread library in C to implement the multithreaded programs. In this project, we spent much time and effort completing different kinds of functions. So far, our code has passed all the test cases in all Tasks.

First, we constructed the function *init_scheduler()*. In this function, we initialized all global variables for the other functions, including three mutex locks, three tids, three queue counts, three global times (For one CPU and two IOs), total thread count, and policies (FCFS and SRJF). Also, we allocated three queues for CPU, IO_1, and IO_2. (./interface: line 13-36)

Next, we constructed *cpu_me()* and *io_me()*. These two functions are almost the same. *io_me()* needs to check which IO device the certain thread is using first. Every time when we call these functions, we need to trace the number of threads. If the task of all threads is not known, there is a possibility that the task of a thread that has not yet arrived is earlier than the scheduled task. Therefore, before each *schedule_next()* (will describe later), you need to determine if the number of threads with scheduled tasks is equal to the number of threads. So, *queued_thread_count* will add 1 (./interface: line 54) and check whether it is equal to the *total_thread_count* (./interface: line 56). In the functions *cpu_me()* and *io_me()*, we use *pthread_mutex_lock()* and *pthread_mutex_unlock()* to prevent order confusion. When a thread gets the lock, the function *enqueue()* will be called. This thread will be put in a CPU queue or IO_1 queue or IO_2 queue, which depends on the operations of threads (./interface: line 53 & 125). Then wait for all threads coming in (We use *pthread_cond_wait()* to let threads wait for others). After all the threads are in the queue, we use *pthread_cond_boardcast()* to wake up all of them and call the function *schedule_next() (schedule_next() will be only called one time at this time)*. (./interface: line 64 & 67 & 136 & 139)

schedule_next() is a very complicated function. We put lots of effort into it. The feature of this function is to call *dequeue()* to get the task in each queue that should be scheduled first according to the policy. Each task has its own global time: *global_time, global_IO_1_time and global_IO_2_time.* We sort three global variables and find the smallest one (./scheduler: line 31 - 110). Then put this task to the function *schedule_cpu()/schedule_IO()*. If there is a conflict, CPU > IO_1 > IO_2, because 1 tick < 2 tick < 5 tick, the next task of the thread with the shortest task burst duration must also be the first to arrive, provided the arrival times are the same. After choosing, the other tasks which are not selected will be enqueued again by call *enqueue().* The most difficult part in *schedule_next()* is how to make *cpu_me()* and *io_me()* returns the correct result. This correct result will be returned by three factors: 1. following the specified policy. 2. After *schedule_next()* has made its decision, no task will arrive earlier than the scheduled task (and thus should be scheduled earlier). 3. Only the *cpu_me()/ io_me()* of the thread selected by s*chedule_next()* will return. To reach these three goals, we create many functions. First, parameters and logical guarantees in *enqueue()* and *dequeue()* help *schedule_next()* know which policy the certain thread has (./scheduler: line 264 & 276 & 332 & 336). The second factor is guaranteed by two components, including keeping track of the number of arrived threads and maintaining the global time for each queue. Third, make sure that only *schedule_next()* can decide which thread is chosen by using mutex lock and signals(./scheduler: line 119 & 129). After constructing these features, *schedule_next()* can finish all tasks.

In *schedule_cpu() and schedule_IO()*, the selected task will be scheduled finally. The *global_time/global_io_1_time/global_io_2_time* will be converted to int type and returned by *cpu_me()/io_me().* (./scheduler: line 145 – 149 & 201 - 206)

When we call the function *end_me(), total_thread_count* will be minus 1. Then, the function will check whether all other threads arrive. If they arrived, use *pthread_cond_boardcast()* to wake them up.

This is the whole process of project 2. It completes the various functions of multi-threaded synchronization.