Project 1 : Buzzing with Threads!

Sikai Zhao
szhao326@gatech.edu (903517007)

Goal

The goal of Project1 Buzzing with Threads is to implement a credit-based scheduler in a user-level threads library, GTThreads Library, which implements O(1) scheduler and co-scheduler algorithms.

1 Overview

Credit scheduler is a proportional fair share virtual CPU scheduler. It is the default scheduler of Xen.[1]

There are two important parameters in credit scheduler:
(1) Weight, is the weight coefficient of the CPU time a virtual machine can get. A domain with a larger weight will get more CPU on a contended host.
(2) Cap, fixes the maximum amount of CPU a domain will be able to consume. A cap of 0 puts the VM in work-conserving mode

Each VM is assigned a weight and a cap. The project 1 works in the work-conserving mode, so the value of cap will be 0 in this project. As for weight, initial credits represent the weights in this project, that is the more weight a thread has, the more credits it will get every time, and the more CPU it will get. The detailed implementation of credit scheduler will be discussed in next section.

GTThread Library is is a user-level thread library that can provide Multi-Processor support, Local runqueue and O(1) scheduler and co-scheduler. The workflow of GTThread is roughly diplicted in Figure 1. The GT-Threads Library contains two kinds of threads, kthreads and uthreads. The former generated in the physical CPU and the latter simulates the virtual CPU and runs on the kthreads. Also, there are 3 kinds of signals, VTALRM, SIGUSR1 and SIGUSR2 connecting each part of the library. Kthreads, uthreads and signals are

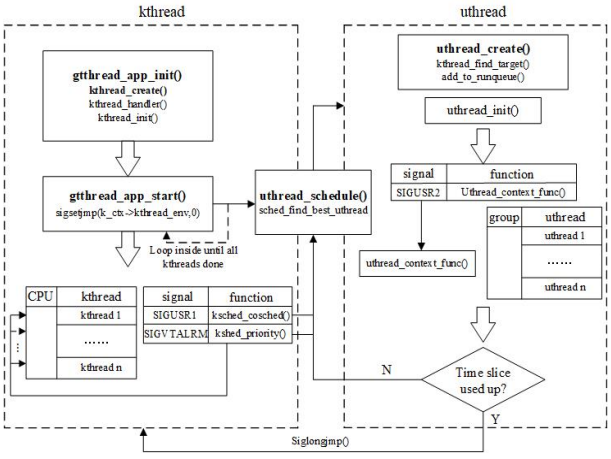

Figure 1: Machanism of GTThread

important concepts in this project and in credit scheduler.

In GTThreds, the initial kthread is created as structure of *kthread_context_t* and then be duplicated by other processes using *CLONE()*. There are two signal handlers installed in kthreads, *kthread_sched_timer* and *ksched_sched_relay*. An kthread will catch the VTALARM signal at the end of each timeslice using *kthread_sched_timer* and then make scheduling. And this kthread relays the scheduling signal to other kthreads using signal of SIGUSR1 , which will be caught by other kthreads by *ksched_sched_relay*. Then all the kthreads can be scheduled. As for uthreads, they are running under kthreads, and are chosen by *uthread_schedule* at the end of every time slice to be held by the current kthread. There is also a signal hander in uthread, *uthread_context_func*. Once a uthread is initialized, SIGUSR2 signal will be generated, which triggers the *uthread_context_func* to complete the task of the uthread. *Uthread_schedule* will be called again if the task is done within a timeslice.

Besides implementing credit scheduler based on GT-Threads Library, migration of urthreads between the kthreads when a kthread is idle is performed in this project. The migration maintains the principles of work-conserving and load balancing. Also, when a user-level thread executes this function, it should yield the CPU to the scheduler, which then schedules the next thread. A command line method for choosing O(1) or credit scheduler, choosing if load balancing or yiele is needed is implemented.

2 Design and Implementation

2.1 Design of Credit Scheduler

In the credit scheduling algorithm, the main idea is that the virtual CPUs should be given CPU time according to their credits. Each uthread, which represents the virtual CPU(vCPU) has two parameters, weight and cap. In this project, the weight can be implement by the initial credits, which shows the ”importance” of a uthread. Every time when the uthreads be initialized or reassigned credits, the value will be the initial credits(weight). And the parameter of cap set the maximum of credits a uthread can have. Because this project works in the work-conserving mode, the value of cap will be 0 for all the uthreads.

Each kthread, which represents the physical CPU, have two runqueues, UNDER queue and OVER queue. The former contains the uthreads that have positive credits and will be scheduled on the kthread in current round, while the uthreads on the latter queue will not be scheduled in current round. An operation of switch the OVER and UNDER queues is critical to the credit scheduler. Another key point is that the credits of the uthreads that are running should be deduced in proportion to the CPU time that get. It is the main idea of credit.

Therefore, the working flow of credit scheduler in this project is as follow(Figure 2 shows the flowchart):

(1) When the uthreads are initialized, they are assigned the initial credits;
(2)As the uthreads run, they consume their own credits in proportion to the CPU time it get;
(3)At the end of a time slice, if the current uthread has negative credit, which means it consumed more CPU time than it should have, it will be added to the OVER queue and will not be scheduled in this round; and if the credit is positive, the uthread will be put at the tail of UNDER queue and wait to be scheduled;
(4)If the UNDER queue is empty, the UNDER and OVER queues will be switched and the uthreads will be reassigned credits for the next round;
(5)If both the UNDER and OVER queues of a kthread are empty, a load balancing method will be used to ”steal” uthread from other kthread’s UNDER queue to balance the load.

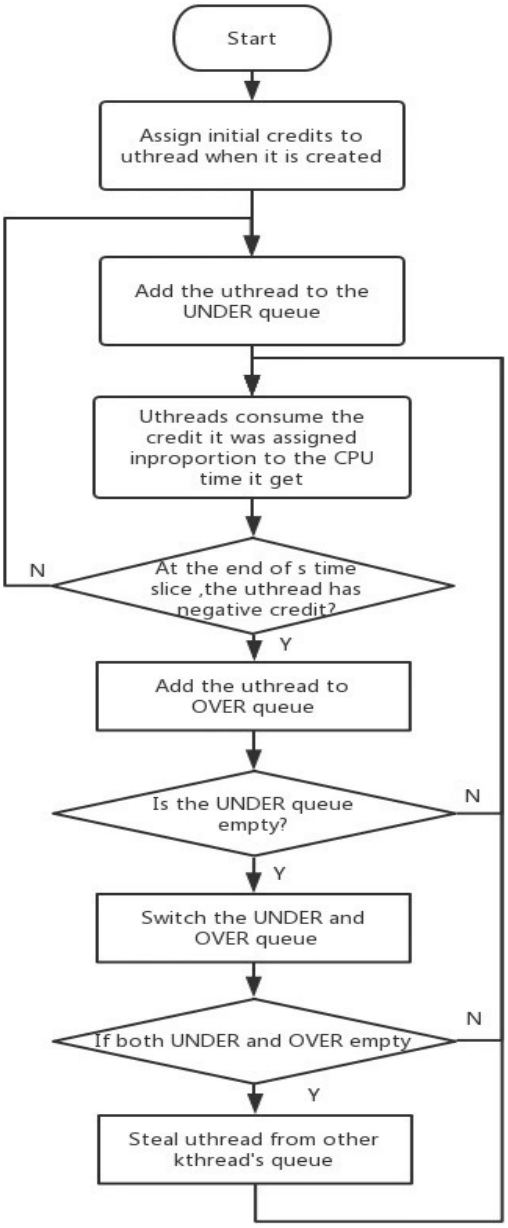

Figure 2: Flowchart of Credit Scheduler

2.2 Implementation of Credit Scheduler

In this project, credit scheduler algorithm should be implemented in the GTThreads Library that originally implements O(1) scheduler, so a command line argument choosing algorithm, like *uthread_schedule_algo* in this project, is needed. More importantly, another function for credit scheduling is also needed.

In the original O(1) scheduler, the main scheduling of next uthread is implemented in the function of *sched_find_best_uthread*, which is an input argument of *uthread_schedule* function, where the kthreads interact with uthreads. Therefore, a new funtion equal to *sched_find_best_uthread* can be implemented to serve as the function to find the next uthread based on credit scheduler. I added a pointer function called *sched_get_head* to the *gt_pq.c*, which is used to return a pointer to the first uthread in the UNDER queue of the current kthread. In this way the next uthread to be scheduled can be find in an easy way.

In addition, the switch of UNDER and OVER queues is also implemented in the function *sched_get_head*. In the O(1) scheduler of GTThreads, the uthreads in the kthreads are in a structure of *kthread_runqueue_t*, which consists of three queues: *active_runq*, *expires_runq* and *zombie_uthreads*. *active_runq* contains the uthreads that are to be scheduled, *expires_runq* contains the uthreads that have already been scheduled in current round, and *zombie_uthreads* contais the uthreads that have been completed or cancelled. When O(1) is working, the uthreads in the *active_runq* will be selected according to priority and put in *expires_runq* after executing. When the *active_runq* is empty, they will be switched[2]. If *expires_runq* is also empty, the kthread is done. Hence, the operation of queues of credit scheduler and O(1) scheduler has something in common. In the function *sched_get_head*, I use the *active_runq* of kthread as UNDER queue and *expires_runq* as OVER queue. The difference is that in the credit scheduler, there is a parameter of credit, so when the queues are switched, the values also need to be reassigned based on some methods. In this project, I chose to add the initial credits to the current negative values of the uthreads to give them the remaining credits to use in next round.

There are some others changes to support the credit scheduler implementation. In the *gt_kthread.c*, I changed the calls for function *uthread_schedule* in the functions by adding an *if* operation to judge what is the scheduler algorithm needed and use the right function as the input to *uthread_schedule*. And in the *kthread_shared_info*, I also add an argument to show the algorithm that the project is working on. In the structure of *uthread_struct_t*, I add several arguments to store the information related to credit scheduler, *credit*, *initial_credit* and *cap*.

2.3 Implement of load balancing

As stated before, load balancing can maintains the principles of work-conserving of credit scheduler. To implement it, I add some code in the *uthread_schedule*. If the *uthread_sched_algo* and *load_balancing* arguments are both set 1, the function will search for the first kthread that has not done and take a uthread from the tail of its UNDER queue. To avoid the contention with the function that select uthread from the head, when load balancing is working, the tail will be selected as the current uthread of current kthread. If the load balancing happens, we print the infomation to the screen and also print the UNDER queue of the target kthread before and after the balancing.

Also, same as algorithm choosing, a command line argument called *load_balancing* is implemented to choose if load balancing is needed.
./bin/matrix -c 1 -l 1 for load balancing

2.4 Implement of Yield

When a user-level thread executes the function of *gt_yield*, it yields the CPU to the scheduler, which then schedules the next thread. The *gt_yield* is a kind of voluntary preemption and can be implemented by calling *uthread_schedule* to yield to the scheduler and let the next uthread to be scheduled.

A command line argument called *yield* is implemented to choose if yield will be used:
./bin/matrix -c 1 -y 1 for yield

2.5 Implementation of test code, *gt_matrix.c*

The *gt_matrix.c* is implemented to test the credit scheduler. In *gt_matrix.c*, 128 matrices are generated and work on the uthreads. In this project, the matrices are of the size of 32, 64, 128 and 256, and the credits for the uthreads that execute them is 25, 50, 75, 100,so there are 16 kinds of matrices with 8 for each kind. The function to generate, print the matrices are implemented. And the code to deal with the command line arguments is also implement in *gt_matrix.c*. I also create some arrays of *CPU_time*, *execution_time* and *wait_time* to help evaluate the code.

3 Evaluation

The most important thing to be evaluate for the thread scheduling algorithm is how the threads are scheduled. It's hard to directly test the details of the order or other factors of scheduling, but we can evaluate with time. In this project, three kinds of time are used to evaluate the performance of the code:

(1) CPU Time - time spent by a uthread running every time it was scheduled (i.e. excluding the time it spent waiting to be scheduled).
(2) Wait Time - time spent by a uthread waiting in the queue, to be scheduled.
(3) Execution Time = CPU Time + Wait Time

To calculate the time, I add some variables to the structures in the code. In *kthread_shared_info*, I add an array of *cpu_time* to store the cpu time of the uthreads run on it. In textituthread_struct_t, I add *start_time* to store the time that the uthread starts to run on the CPU for every time. And in the *uthread_schedule* function, the CPU time that the uthread got this time can be calculated by the actual time and its *start_time* and update its *cpu_time*. Then we can finally get its precise CPU time. In *gt_matrix.c*, I calculate the total time of the operation of multiply of the matrices using the time the uthread created and the time the operation ends, then we have the execution time. The wait time can be obtained through execution time and cpu time. The three kind of time are printed to the files and screen in *gt_matrix.c*.

group_name	mean_cpu_time	mean_wait_time	mean_execution_time
c_25_m_32	1945	1438445	1440391
c_25_m_64	21756	1442181	1463938
c_25_m_128	181006	2391717	2572723
c_25_m_256	1442213	10121200	11563414
c_50_m_32	1450	1439454	1440905
c_50_m_64	25745	1462123	1487868
c_50_m_128	162547	2236230	2398778
c_50_m_256	1446830	10073072	11519902
c_75_m_32	511	1441866	1442377
c_75_m_64	18344	1489641	1507985
c_75_m_128	165862	1990845	2156707
c_75_m_256	1457135	9674344	11131479
c_100_m_32	1102	1442071	1443174
c_100_m_64	23275	1508705	1531980
c_100_m_128	176708	2788143	2964852
c_100_m_256	1456363	8724668	10181032

Figure 3: Result of time (mean)

The table above shows the mean of the CPU time, wait time and execution time.

group_name	sd_cpu_time	sd_wait_time	sd_execution_time
c_25_m_32	3792.59	1444114.02	1442663.38
c_25_m_64	9844.75	1443905.16	1439595.68
c_25_m_128	40626.29	24015.22	638940.82
c_25_m_256	358916.72	2478398.22	2792120.33
c_50_m_32	2537.59	1442037.71	1442622.66
c_50_m_64	11785.01	1440857.45	1447162.69
c_50_m_128	36727.87	181977.53	2187808.42
c_50_m_256	356545.91	2594493.17	2884950.16
c_75_m_32	78.13	1443248.05	1443209.54
c_75_m_64	11693.34	1445922.98	1444104.03
c_75_m_128	38898.64	1365405.42	1375145.84
c_75_m_256	384856.31	2253842.7	2513340.17
c_100_m_32	1626.11	1442993.53	1443361.44
c_100_m_64	6625.33	1444079.17	1443745.41
c_100_m_128	44912.4	1789448.41	1812456.95
c_100_m_256	411462.6	2358555.97	2676192.37

Figure 4: Result of time (standard variation)

The table above shows the standard variation of the CPU time, wait time and execution time.

3.1 Results

The code is test using my local computer(Ubuntu 16.04) and virtual machine advos-02.cc.gatech.edu.

3.1.1 Credit scheduling without load balancing

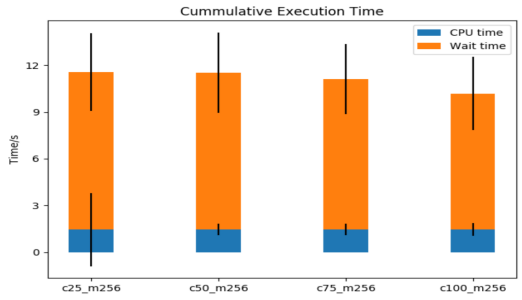

Figure 5: Cummulative time for M256

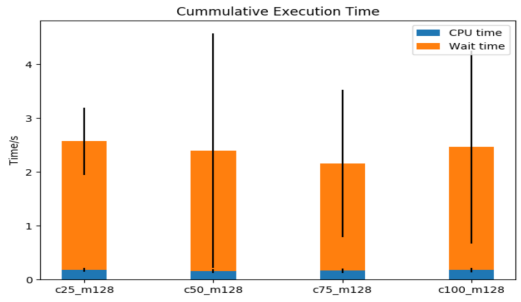

Figure 6: Cummulative time for M128

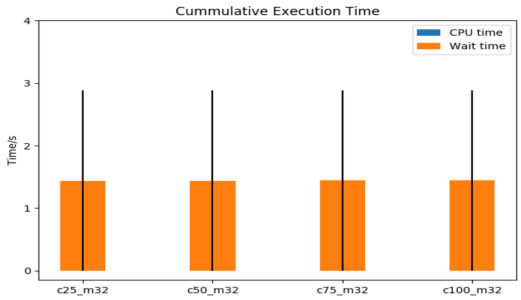

Figure 7: Cummulative time for M64

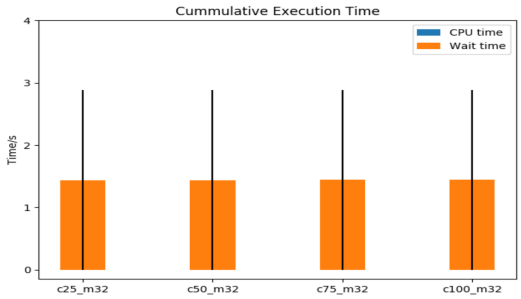

Figure 8: Cummulative time for M32

The figures above show the CPU time, wait time and execution time of the four weights and four matrix sizes in credit scheduler clearly without load balancing.

3.1.2 Credit scheduling with load balancing

When the load balancing happens, a message of ”Stealing uthread from another kthread to achieve load balancing” is printed, and the UNDER queue of the target kthread is printed twice before and after the load balancing. Here is an example:

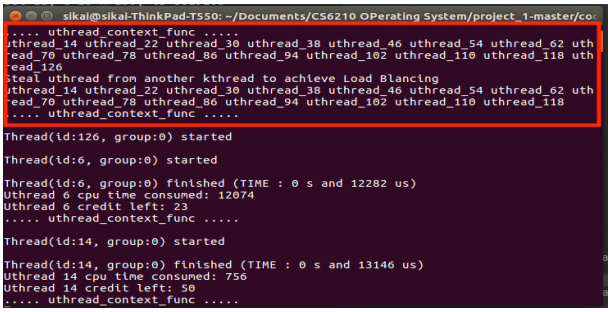

Figure 9: Screenshot of load balancing

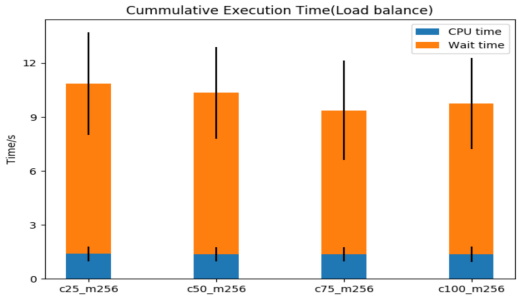

Figure 10: Cummulative time for M256

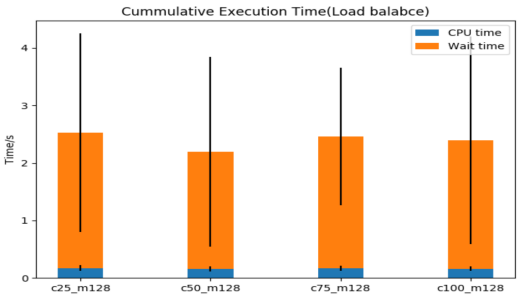

Figure 11: Cummulative time for M128

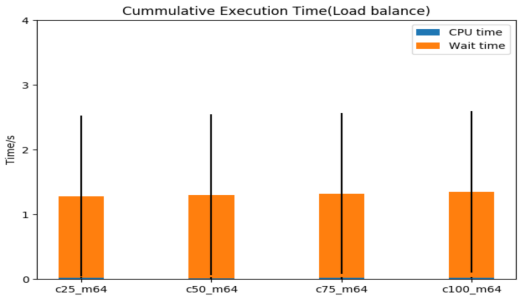

Figure 12: Cummulative time for M64

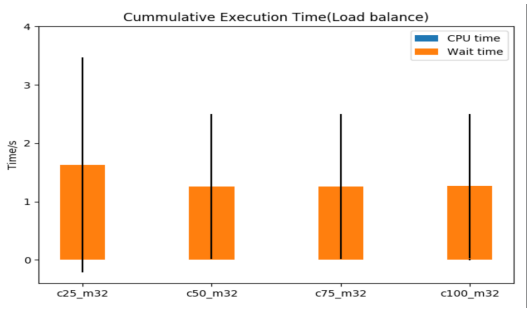

Figure 13: Cummulative time for M32

The figures above show the CPU time, wait time and execution time of the four weights and four matrix sizes in credit scheduler with load balancing.

3.2 Observations

1. Theoretically, for the same size of matrices, the CPU time of the uthreads of the four credits will be equal in credit scheduler. And the wait time and execution time is larger for the ones with low credits or to say, weights. According to my results, the CPU time for the matrices of the same size is basically the same. However, as for wait time, the results shows the regularity when the size is 256 or 128, while it is not so clear for sizes of 64 and 32. And for 128, there is a abnormal wait time for credit 100.

Anlysis: The first possibility is that the time calculation method for execution time in my code does not work so well. I stored the start time for all the uthreads, and the end time for each of them. The process of creation of uthreads might take some time. The second possibility is that the credit scheduler algorithm is not so good. There are some details maybe I didn't deal with or do not treat well. Such as the use of spinlock or the method to prevent deadlock.

2. The load balancing works, and can take uthread from the UNDER queue of other kthread to the current kthread, which will make the time evener and shorter. In my experiments, I found that load balance happens not so frequently. It happens only one or two time each time or even not happen.

Analysis: The load balancing can allocate the uthreads more evenly to improve the performance of the algorithm. As for the frequency, the possible reason is that the uthreads runs on four kthreads within a short time, so when the UNDER and OVER queue are both empty, the other kthreads might also be empty. Therefore, load balancing not happen frequently.

3. The results of time for each experiment vary, sometimes a lot, especially for the small size matrices.

Analysis: The reason for this might be the operations for calculating small size matrices is fewer than for large size ones. So the execution time of them are easier to be influenced and have large variations. What's more, the load, deadlock and other factors might also influence that.

4 Conclusion

This project give me a chance to learn plenty of knowledge of Operating System. As an ECE student, although I have learned something about OS, this is the first time I try to write a code to implement some basic ideas of OS. From the Project1, I learned how to schedule the processes and threads, how important the scheduling is and how does O(1), credit and GTThreads work, how to use some system calls, how to deal with the signals in code and lots of other things. The project also gives me a good chance to practice programming in C and deal with large amount of code. Thanks the professor and TAs who answer questions and help me understand OS.

References

[1] Credit Scheduler. https://wiki.xen.org/wiki/Credit_Scheduler. June 28, 2018

[2] O(1) Scheduler.https://en.wikipedia.org/wiki/O(1)_scheduler. August 11, 2019