Credit Scheduler based on GTThreads

Zeyu Chen

February 6, 2018

1 Introduction

The goal of the project is to realize a credit scheduler using GTThreads library. The credit scheduler is used as the default scheduler of Xen. The GTThreads implements a O(1) priority scheduler and a priority co-scheduler. In this report, the structure of the GTThreaads and how credit scheduler is implemented based on the GTThreads will be discussed. Also, the matrix multiplication bench mark will be shown as the result.

2 Approach

2.1 GTThreads - Priority Scheduler

GTThreads is the library which implements a O(1) priority scheduler. The algorithm can be briefly summarized as follow:

Under priority scheduling, each thread is assigned a priority. Thread with the highest priority is to be executed first and so on. Thread with the same priority are executed on first come first served basis. Priority is determined at the beginning of the execution.

The implementation of the library is using three different queues storing three kinds of threads with different states.

Three queues are:

- Active - Store the threads which are runnable in this cycle.
- Expire - Store the threads which are already ran in this cycle.
- Zombie - Store the threads which are done or canceled.

Each queue maintains different groups of threads with different priority.

First of all, all threads will be added to the active queue according to priority and be ready to execution.

Once the scheduling function calls, the active queue will first find the highest priority group and then pop out the head of this group to the CPU for execution. The priority is determined by a priority mask(The lowest bit indicates highest priority and so on). The search time is constant since the priority level is constant, that is how O(1) scheduler works.

Before execution, the scheduler will put the current thread to the expire queue. Then, the scheduler will check both active and expire queue. If the active queue is empty, the program will check if the expire queue is empty. If not, the program will switch two queues. If both active ans expire queue are empty, the program will exit since there are no more unfinished threads.

The chart shows the flow of the function calls.

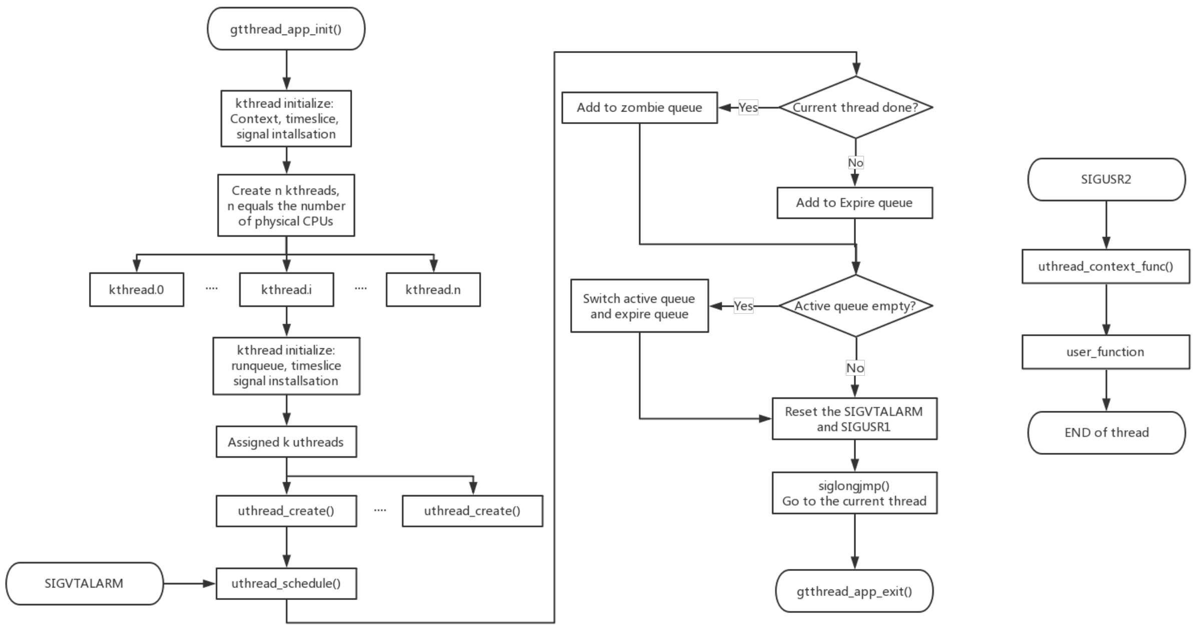

Figure 1: Function call flow

2.2 Credit Scheduler

Credit scheduler is first used in Xen. The priority of the scheduling is based on the credits of the thread [1].

The features of the credit scheduler can be summarized:

- Each thread is assigned certain credits. The credits is calculated based on the weight and cap of the thread.
- Once the thread is executing, the credits are burned by execution time. When credits are positive, the thread is assigned "UNDER" priority, otherwise, it is assigned "OVER" priority.
- Weight: The thread with higher weight will have more chance to occupy the CPU. For example, a thread with a weight of 128 will get twice as much CPU as a thread with a weight of 64.

- Cap: If set, fixes the maximum amount of CPU will be able to use. For example, 100 is 1 physical CPU, 400 is 4 physical CPUs, etc. However, 0 means no cap.
- The default time slice is set 30ms. However, the user can change the time slice based on the specific situation.

Credits are calculated by the weights and caps, however, there is no specific equations. Basically, the core of the algorithm is that the threads with higher credits will occupy more computing resource during the whole process.

2.3 Implementation

2.3.1 Algorithm

In this article, credits are ranging 25, 50, 75 and 100. Cap is not considered. In other words, the thread with higher credits will have more chance to get the CPU resource which leads to the fact that these threads with higher credits will have less lifetime.

The implementation of credit scheduler here is based on the priority scheduler implemented in GTThreads library. The active queue represents "UNDER" priority while the expire queue representing "OVER". Also, the zombie still store the finished and canceled threads.

Once the thread is created, it will be assigned a certain credits. When the thread is selected to be scheduled, the program first charge it 25 credits for the current execution and then check its priority(simply check whether the credits is positive or not). If it is "UNDER", it will be added to the active for the next scheduling. Otherwise, the thread will go to the expire queue and the credits of the thread will be reset.

The priority is also taken into consideration. Since once the thread is created, it will starts automatically(based on the implementation of GTThreads). Thus when a thread is being initialized, the priority mask will be set according to the credits.

2.3.2 Technical Details

The implementation is based on the priority scheduler, some of the functions are still remained. The main modification is shown below:

- Each thread maintains its own current credits, assigned credits and born time.
- The uthread scheduling function is modified to realize the algorithm mentioned above.
- The function in priority queue is updated that the done kthread will try to find active uthread from other undone kthread since the credit scheduler is a work conserving scheduler.
- A yield function is added calling uthread schedule function to realize the voluntary preemption.
- A new structure and a series of regarding functions are added to collect the running time and the lifetime of the thread.

3 Results and Analysis

There are 16 combinations with 8 threads in each group. The time slice used is 30 ms. The program calculates the mean time of each group and the standard deviation.

The result has been tested both on 2 core a non-shared Linux machine and 4 core shared Linux machine. The time accuracy is micro seconds. Here we only show the mean value, standard deviation can be checked in the output file.

The mean lifetime on a 2 core non-shared machine:

Size / Credits	32	64	128	256
25	3795.674	3811.352	3893.143	3616.396
50	2976.492	2988.120	3002.523	3053.638
75	1845.409	1857.014	1870.886	1982.424
100	681.051	692.674	703.260	838.109

The mean lifetime on a 4 core shared machine:

Size / Credits	32	64	128	256
25	1014.878	1067.464	1023.836	1491.382
50	1071.820	1077.570	1081.427	1388.574
75	460.743	466.432	470.296	576.531
100	407.204	412.978	417.096	531.587

The mean execution time on a 2 core non-shared machine:

Size / Credits	32	64	128	256
25	0.221	2.987	20.277	170.700
50	0.229	2.077	19.244	103.010
75	0.233	2.701	27.863	161.122
100	0.227	5.001	24.560	161.340

The mean execution time on a 4 core shared machine:

Size / Credits	32	64	128	256
25	0.418	2.373	21.710	37.606
50	0.297	2.265	16.730	25.545
75	0.343	2.171	16.653	37.190
100	0.278	1.963	15.162	44.596

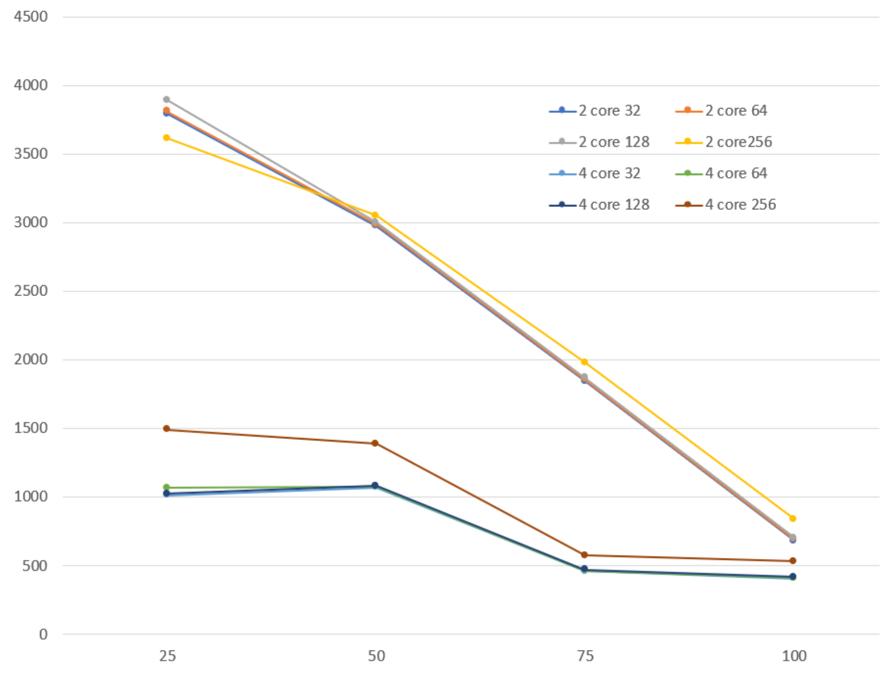

Figure 2: Mean Thread Life Time

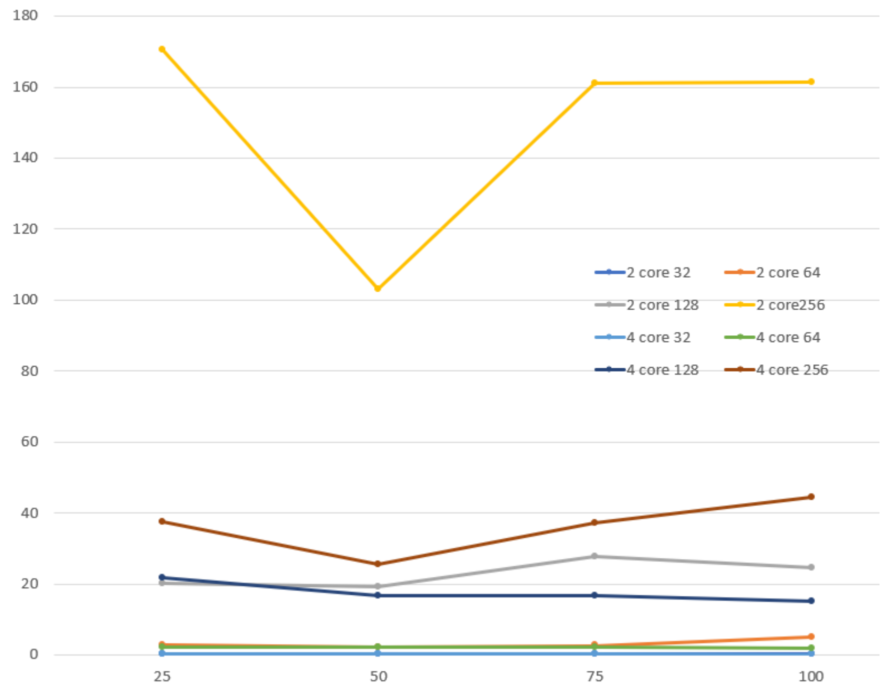

Figure 3: Mean Thread Execution Time

According to the results, some conclusions can be briefly summarized:

- The mean life time is related to both size of the matrix and the credits of the thread. The larger size will cost more time.
- In the same size group, threads with larger credits has less life time. This verifies the rule of the credit scheduler. A thread with higher credits has more chance to occupy CPU that it wastes less time in waiting.
- The mean execution time is only related to the size of the matrix. Execution time of same size with different credits are in the same level. This is because the actual on CPU execution time is only related to the computing complexity of the user and the result verifies the reason.
- Performance are different upon different machines. The non-shared machines turns out to have a clearer descending life time by credits since there are less other processes disturbing the program. Also, some time collection is quite different with the expectation dues to the same reason.

In short, the result of the test meets the expectation. During the implementation, small bug is found such as non-safe thread communication. The program in more cores machine will sometimes hang during the kthread exit function. The kthread waiting will be interrupt by the signal and go to an invalid address in memory. One quick way to solve this problem is using a barrier method instead of the original method, which is more reliable.

References

[1] Dario Faggioli, “Credit scheduler,” 2018.