CS 6210: Advanced Operating Systems Feb 15, 2023

Project 1 Report

Instructor: Ada Gavrilovska *Aditya Vikram*

1 Abstract

This report provides an overview of the GTThreads package, a design of the Credit Scheduler along with implementation details, and concepts like load-balancing and yield. I also show the experimental results and draw inferences from them.

2 GTThreads Package

The diagram below provides a broad overview of the various modules involved.

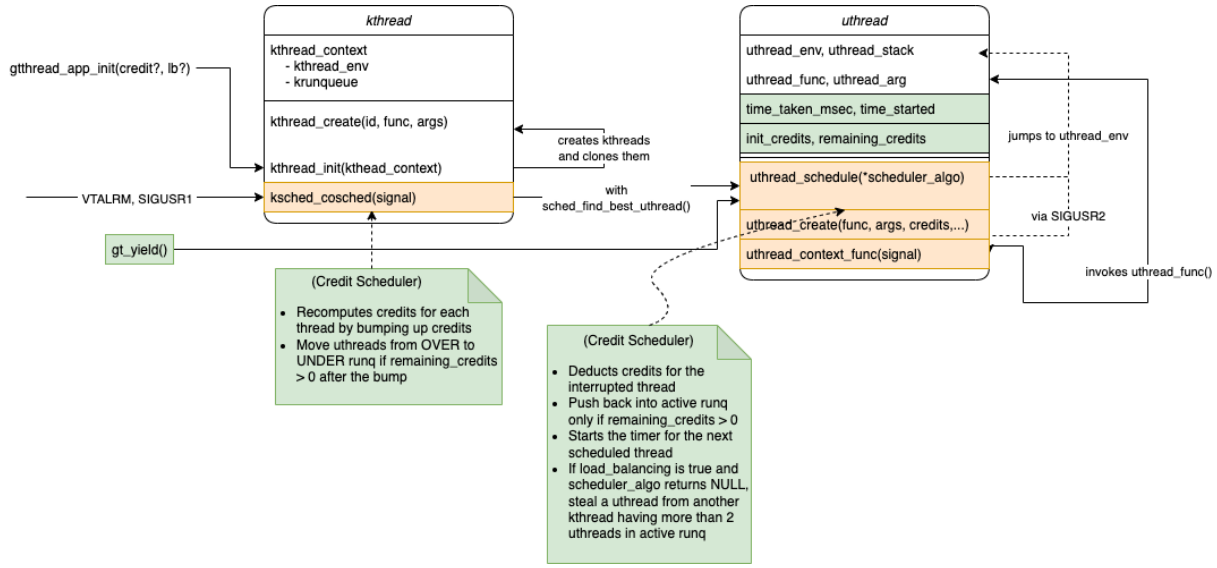

Figure 1: GTThreads package with Credit scheduler
Components colored green are added specifically for the credit scheduler implementation, while the orange components were modified to accommodate the credit scheduler and load balancing

Let's go through each module of the GTThreads package briefly.

2.a kthreads

Each kthread "object" (used loosely for structs throughout the report even though there are no classes here) represents a physical CPU core and maintains its own runqueue(s). When the client app starts, it calls `gtthread_app_init()`, which internally determines the number of CPU cores (n) on the machine, and clones the current thread $(n-1)$ times to create n kthreads. This function also installs signal handlers for `SIGVTALRM` and `SIGUSR1`. `SIGVTALRM` is signalled on a timer every 30ms (to be thought of as the time slice), and on every interrupt per kthread, the scheduler is invoked. `SIGUSR1` is used to relay messages between the kthreads.

Each kthread maintains its own runqueue object `kthread_runqueue_t`, which is used for the scheduling decisions for that kthread. This will be talked about in the later subsections again.

2.b uthreads

User threads are created by the client app for parallelizing its work using `uthread_create()`, where a function pointer and function arguments are passed as arguments. I have modified this function to also take as input the initial credit assigned to that uthread. Internally, `uthread_create()` creates the uthread object, initializes it and schedules it onto the available kthreads in a round-robin fashion. A uthread object also stores the thread context and stack in `uthread_env` and `uthread_stack` members respectively. These are used for jumps while context switching to and from the given uthread.

It is important to note that the package also supports priority and groups for uthreads, but those are not used in this project. Priority and groups will be discussed briefly in the scheduler section next.

2.c Runtime workflow

Once `SIGVTALRM` is fired, one of the kthreads handles this and relays the message to other threads. In response, all kthreads invoke their scheduler via `ksched_cosched()`, which then calls `uthread_schedule()`. `uthread_schedule()` takes a scheduler function as an argument and first interrupts the currently running uthread on the kthread, then calls the scheduler function (`sched_find_best_uthread` in this case) to get the next best uthread in this kthread's runqueue(s). Finally we jump to the new uthread's context and finally call that uthread's executable function. This may again by interrupted later by VTALRM, but execution would resume once the uthread gets scheduled again.

3 O(1) Priority Scheduler

The default scheduler implemented in GTThreads package is the O(1) priority scheduler. It is a multilevel scheduler, where each `kthread_runqueue_t` stores an active and an expires run queue. All uthreads are scheduled on the active runqueue first. Once a uthread executes, it may be interrupted by the scheduler and the scheduler would get to choose the next uthread which runs. In this case, the previously running uthread gets scheduled onto the expires runqueue, where it'd get a chance to run again once all the active runqueue tasks have had a chance to run at least once.

Each runqueue in itself is multi-level task queues indexed first by priority, and then by groups. So all uthread groups with the highest priority are picked first by the scheduler (in order of the first group to be added for that priority). Subsequently, the scheduler goes in decreasing order of priority and keeps on popping uthreads from the runqueue onto the CPU. For each given priority and group, it simply maintains a tail queue of all uthreads and chooses the head (FIFO) when deciding the best thread to schedule.

4 Credit Scheduler

The credit scheduler implementation follows the design described in Xen Credit Scheduler([2]). For more details and possible improvements, I referred to [1]. In a credit scheduler, each thread gets an initial credit value when created. Credits can be thought of as importance of that thread, because the idea is to give threads CPU time proportional to their allotted credits. There is also a concept of caps for each thread, which limits the percentage of CPU time a thread can get. Due to time constraints, I have not implemented caps in this project.

As a thread runs on the CPU, it consumes credits. The scheduler maintains two runqueues: UNDER and OVER, which will be the active and expires runqueues from `runqueue_t` for our implementation. When the thread is interrupted or finishes, we compute `remaining_credits` for that thread based on the time elapsed while it used the CPU. If the thread still has positive credits remaining, it is added back to the UNDER runqueue. Otherwise, it goes to the OVER runqueue, denoting that it has overshot its allowed time on the CPU. *This is the major difference between O(1) and Credit scheduler. In O(1) priority scheduler, regardless of the time a uthread takes on the CPU, it gets pushed to the expires queue when it is interrupted/finished. For the Credit scheduler, we reward threads that consume CPU time based on their allotted credits.* The scheduler keeps processing uthreads in the UNDER queue until it is empty, and only then, it processes the OVER queue.

Now as the scheduler runs, it keeps deducting credits from the threads. So all threads would eventually have negative remaining credit (unless they finish early) and would get pushed to the OVER queue. To avoid this, periodically, the scheduler also recomputes `remaining_credits` for each thread. It would take into account the initial credits that the thread had and its `remaining_credits` currently to bump `remaining_credits` based on an appropriate algorithm. Once this is done, if `remaining_credits` > 0 now, then that thread is also bumped up to the UNDER queue.

4.a Yielding

Yielding the CPU is one of the ways a thread can get maximal CPU time. If a thread voluntarily yields the CPU at an appropriate time, its credits would stay positive (considering the credit bump cycles too) and it would remain in the UNDER queue for a longer period of time. This is especially helpful for longer running tasks since they would be heavily penalized at each interruption unless their allotted credits are proportionally larger.

4.b Load balancing

In a real-world system, it is unlikely that all kthreads get an equal amount of work (loosely the sum of work for each uthread scheduled on it). So it is highly likely that a kthread becomes idle after running all its assigned uthreads, while another kthread has a longer queue to process. This ends up in higher wait times for the loaded kthread's uthreads while another kthread could've been used to finish them earlier. So we use load-balancing to reduce the wait times.

Specifically, we use a pull model of load-balancing, where an idle kthread with empty active and expires runqueues would steal a uthread from another kthread with more load. There also exists a push model where a central task would periodically load-balance the system by rearranging uthreads between the kthreads. The implementation details for this are discussed in the next section.

4.c Implementation

I plug into the GTThreads package, using it as a base to build the credit scheduler. All the components added for the credit scheduler, load balancing and yielding functionalities are highlighted in orange or green in Figure 1. First I've added four members in uthread struct to keep track of initial credits allotted, `remaining_credits`, timespec when the uthread last started on the CPU and the total time consumed on the CPU by this thread. The last member is solely for analyzing the results and is returned as an array of length `NUM_UTHREADS` (=128) in `gtthread_app_exit()`. I also updated the `SIGVTALRM` timeslice to $30ms$ as recommended by the designers in [2]. I use the active and expires runqueues of the kthreads's `krunqueue` for UNDER and OVER queues respectively.

Core of the credit scheduler implementation is in `uthread_schedule()`. When a new schedule cycle is invoked, this function finds the current running uthread and interrupts it. At this point, I calculate the time elapsed since this uthread started running on the CPU and deduct credits proportional to that. Based on my experimentation, I found that deducting 2 credits/ms worked out best to keep the longer running tasks in check, while also somewhat penalizing the shorter tasks. While deducting credits, there is also a scenario where a thread might run for too long and int underflow could cause it to have positive credits again. So I capped the lowest `remaining_credits` to `INT_MIN`.

After deducting the credits, I check if the remaining credits are still positive, this uthread gets queued back on the active runqueue. Else it is queued on the expires runqueue similar to O(1) scheduler. Once the scheduler algorithm returns the best uthread to run for the next timeslice, I initialize its $time_started$ member.

For recomputing credits, I chose to do this at start of each timeslice in `kthread_recompute_credits()`. This function gets called from `ksched_cosched()` (before the scheduler is invoked) which is called for each `SIGVTALRM` signal. I made the decision to bump credits at start of each timeslice based on my understanding from [1] and [3]. When recomputing credits, I first thought about only incrementing `remaining_credits` for threads in the OVER queue, but that would have been a violation of ensuring proportional CPU time for the threads. If we need to maintain CPU time proportionality, the easiest way is to ensure an increment

proportional to `init_credits` of the thread. Upon some experimentation, it turned out that it is feasible and simplest to keep this constant factor as 1.

Implementing `gt_yield()` was fairly simple, because we just need to yield the CPU. That can be done by simply calling `uthread_schedule()` with the scheduling algoritm as it interrupts the currently running thread on the CPU. I had to factor out some common code in a private function because we need to explicitly `gt_yield()` calls and finding out currently running thread's id was already done in `uthread_schedule()`.

4.c.1 Load balancing (and implementation issues)

Load balancing was the trickiest part for me, mainly because I kept facing deadlocks. I chose to steal a uthread if the scheduler algorithm returns `NULL` in `uthread_schedule()`. This means that the current kthread has empty OVER and UNDER queues. If this is the case, I iterate through all other running kthreads and call `sched_steal_active_uthread()` (which I implemented in `gt_pq.c`) until I get a non-null uthread. In `sched_steal_active_uthread()`, we check if this kthread's active runqueue has more than 2 uthreads and return the tail (not the best) of the active runqueue. This is good because if there are only 2 or less, the current kthread could easily finish both tasks soon. Note that there could be a more detailed algorithm where we go through each kthread's runqueue and find the one with maximum active tasks but I chose the simplest and fastest one for demonstrating load-balancing functionality.

I return the tail of the source kthread's active runqueue because to get the head task, we would need to acquire the lock for the source kthread's runqueue. This lead to deadlock for me in most runs because if two kthreads are empty and both are looking to steal a uthread from each other's runqueues, it would deadlock the scheduler. Ideally, we would have two spinlocks (public "stealing" spinlock and private per each kthread) and each kthread would atomically acquire both spinlocks. Alternatively, there is a way to use `try_lock()` paradigm where the stealing kthread can try to lock the source kthread's runqueue lock. If it succeeds, it could steal a uthread, otherwise it just moves on to the next kthread. I have used this paradigm in the past with C++'s mutex `try_lock` and tried to find if this is possible with basic spinlocks using assembly instructions but couldn't find anything useful.

So instead I chose to go with another "public" lock per kthread's runqueue, which would ensure that two kthreads don't steal at the same time from a given kthread. This means that only the current kthread and one other kthread could remove tasks from the active runqueue. If we make sure that the implementations are fast and there are more than 2 uthreads, we could get away with this approach. This approach does work to an extent and I did see uthreads being stolen from other threads (seen in the logs when executed with `-lb` arg) but the program always stalled at the end. I verified from the logs that all uthreads in fact finish but the kthread code that waits for all uthreads to complete remains blocked forever. *Unfortunately with limited time, I wasn't able to fix this scenario and have no experimental results for load balancing, but I'll provide my hypotheses in the Results section.*

5 Results

For my experiments, I ran each scheduler configuration (O(1), Credit scheduler and Credit scheduler with yield) with 128 uthreads on a 4 core Intel i7-7500 CPU with 8GB RAM, running Ubuntu 20.04.5 LTS. In each run, the 128 uthreads were partitioned into 16 combinations of following matrix sizes and credits:

- Matrix sizes: 32, 64, 128, 256
- Credits: 25, 50, 75, 100

so that each combination has 8 threads. All 8 threads of same configuration are stacked together when calling `uthread_create()` so that they get split evenly among the 4 CPUs in round-robin assignment. That way, all CPUs have near-equal load at least initially. To reduce system noise, each scheduler configuration is run 3 times and results are aggregated over the 3 runs. Specifically, for each thread group (having same credits and matrix size), the following aggregates are computed:

- Mean and standard deviation of CPU time
- Mean and standard deviation of Wall time

For wall time, the time difference of thread start and finish times are recorded in the client app `gt_matrix()`. For CPU time, for each uthread, I maintain a `time_taken_msec` property where I add the elapsed time since uthread being last scheduled on the CPU till when it gets interrupted or yields voluntarily or finishes. Each thread's final CPU time is then returned via `gtthread_app_exit()`, then both times are logged to `out/runTimesOutput.txt`. Runtimes are computed to nanosecond precision using the monotonic system clock using `clock_gettime(CLOCK_MONOTONIC,...)`. This is preferred over the usual `gettimeofday(..., NULL)` because the monotonic clock difference doesn't get affected on time dilations if the system has to adjust with UTC clock. Another advantage is just ns precision instead of μs. Finally for the results though, the runtimes are converted to ms for readability in the report and analysis.

The following table shows the aggregated mean and std deviation (in $mean \pm stddev$ format) for each thread and scheduler config:

uthread (size, credits)	$O(1)$ scheduler times (ms)		Credit scheduler times (ms)		Credit with yield times (ms)	
	Wall time	CPU time	Wall time	CPU time	Wall time	CPU time
$(32, 25)$	0.94 ± 0.40	0.43 ± 0.17	0.91 ± 0.42	0.42 ± 0.12	28.57 ± 93.90	0.41 ± 0.13
$(32, 50)$	1.86 ± 0.44	0.38 ± 0.08	1.85 ± 0.56	0.41 ± 0.12	29.64 ± 93.83	0.49 ± 0.35
$(32, 75)$	2.86 ± 0.57	0.42 ± 0.16	2.76 ± 0.77	0.38 ± 0.13	2.92 ± 0.85	0.41 ± 0.12
$(32, 100)$	3.77 ± 0.65	0.37 ± 0.07	3.57 ± 0.94	0.35 ± 0.04	4.24 ± 1.39	0.57 ± 0.84
$(64, 25)$	8.13 ± 1.67	2.70 ± 0.32	199.25 ± 195.88	2.67 ± 0.46	8.68 ± 2.42	2.76 ± 0.75
$(64, 50)$	167.08 ± 202.84	2.88 ± 0.93	12.74 ± 2.43	2.94 ± 0.96	29.50 ± 73.82	2.97 ± 0.83
$(64, 75)$	68.81 ± 134.65	2.88 ± 0.69	18.50 ± 2.41	2.75 ± 0.29	63.93 ± 115.90	3.13 ± 1.71
$(64, 100)$	24.62 ± 3.60	2.91 ± 0.82	24.16 ± 2.81	2.91 ± 1.00	40.17 ± 67.77	2.79 ± 0.67
$(128, 25)$	245.62 ± 194.23	22.93 ± 4.29	230.50 ± 177.91	22.10 ± 1.76	466.54 ± 137.58	21.60 ± 1.22
$(128, 50)$	262.64 ± 207.71	22.01 ± 1.85	298.59 ± 228.26	22.88 ± 3.00	508.98 ± 134.09	22.17 ± 3.08
$(128, 75)$	321.89 ± 225.31	22.87 ± 2.64	347.87 ± 246.84	21.90 ± 1.00	457.99 ± 108.45	22.23 ± 2.57
$(128, 100)$	386.28 ± 257.66	21.30 ± 0.81	339.10 ± 188.45	23.06 ± 3.23	547.51 ± 125.41	22.53 ± 4.00
$(256, 25)$	$1,713.68 \pm 141.57$	198.99 ± 10.57	$1,783.82 \pm 75.28$	204.63 ± 5.94	$1,701.37 \pm 126.51$	201.64 ± 9.84
$(256, 50)$	$1,723.91 \pm 93.30$	202.30 ± 12.11	$1,776.24 \pm 102.57$	205.53 ± 8.20	$1,703.62 \pm 138.07$	206.76 ± 12.21
$(256, 75)$	$1,736.52 \pm 98.30$	199.48 ± 10.83	$1,774.63 \pm 79.35$	203.13 ± 7.85	$1,704.66 \pm 113.84$	200.59 ± 10.34
$(256, 100)$	$1,765.90 \pm 88.57$	198.69 ± 9.61	$1,803.13 \pm 71.11$	205.09 ± 7.06	$1,674.07 \pm 126.05$	192.30 ± 9.92

Table 1: CPU and Wall time statistics for each uthread group for different scheduler configuration
All results from each of the three runs can be accessed in detail here

First in the comparison between O(1) and Credit scheduler, there doesn't seem to be much of an improvement in average wait times, ignoring system noise. I think that this is due to two reasons. Firstly, the credit assignment is not proportional to the amount of work done by the tasks, i.e. if we double matrix size, the runtime of that uthread increases 8-fold since we're using the basic $O(n^3)$ algorithm for multiplying matrices. So a better credit assignments would be one where the credits increase 8-folds. Secondly, the longer running tasks don't yield the CPU in this scenario, so they get penalized further. As noted in the Credit scheduler section, this is what the Credit scheduler is meant to do. I feel that increased wait times

in this case is actually a feature of the Credit scheduler and it works correctly. CPU time shouldn't change because of any changes in scheduler configurations and that is evident in Table 1. CPU time mainly depends on the computation performed by the tasks, which mostly independent of the interrupts. The only caveat would be that if too much time has passed and the cache for those array accesses is no longer hot, which doesn't seem to be the case here.

When comparing the second and third scheduler configs, we can see a drop in the Wall times of the longer running tasks when yielding is enabled. In my `gt_matrix` implementation, I yield every 64 rows of the outer loop of matrix multiplication, so yielding doesn't affect matrix sizes 32 and 64 anyways. As can be seen for the four thread groups with matrix sizes 256, the wall time reduces by nearly 7-10% when yielding is enabled.

Now let's look at how CPU time and wait times behave vs credits within a scheduler config. First for the normal Credit scheduler run, as can be seen in the bar charts below, the wait time increases for the uthreads as credits increase. This is because there is no yielding and the tasks with bigger credits get scheduled later in the main function by virtue of the uthread creation routine. So they end up being behind in the queue and have to wait slightly more.

Figure 2: Credit scheduler: Variation of average CPU and Wait times with credits for varying matrix sizes

With yield enabled (Figure 3 below), the first two cases (matrix sizes 32 and 64) aren't relevant because yield is invoked every 64 rows of computation. The spuriousness of wait times for these cases can be attributed to system noise. It can be verified in the "Credit with yield" sheet here that there are 2-3 instances of threads taking order of 100s of milliseconds which drives the average up. For the case of size 256 matrices, there is a drop in wait time as the number of credits increases. But the drop is not very significant compared to the wait time, so it is not that apparent in the figure. In hindsight, I think I should have added yield calls more frequently to demonstrate the difference in wait times better. Due to a time crunch with other course assignments, I wasn't able to do this because of the work needed to reorganize and analyze new data.

Figure 3: Credit scheduler with yield enabled: Variation of average CPU and Wait times with credits for varying matrix sizes

Although I couldn't get the program to finish executing in the load balancing scenario, I think load balancing wouldn't improve the wait times much in this scenario. This is because we are scheduling near-equal loads on all 4 CPUs in the experiments so in the average case, all CPUs would finish nearly around the same time and none would remain idle. However if we were to experiment with a larger number of uthreads with varying workloads, load balancing would certainly help in balancing the work done by each CPU and conserving time.

Note: My project built fine on the VM cluster advos-03 up to a point but stopped running due to memory issues or increased load towards the deadline of the project. The executable runs well most of the time on my personal machine but continues to segfault on the VM cluster for me most of the times, regardless of the scheduler configuration. I feel that it is due to the increase in `NUM_THREADS` *from 16 during earlier development phase to 128, and the VM isn't able to malloc that much memory well. I can bring my personal machine for the demo session and use the submitted project code zip from scratch, if that helps.*

References

[1] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, sep 2007.

[2] Xen. Credit scheduler. https://wiki.xenproject.org/wiki/Credit_Scheduler.

[3] Lingfang Zeng, Yang Wang, Wei Shi, and Dan Feng. An improved xen credit scheduler for i/o latency-sensitive applications on multicores. In *2013 International Conference on Cloud Computing and Big Data*, pages 267–274, 2013.