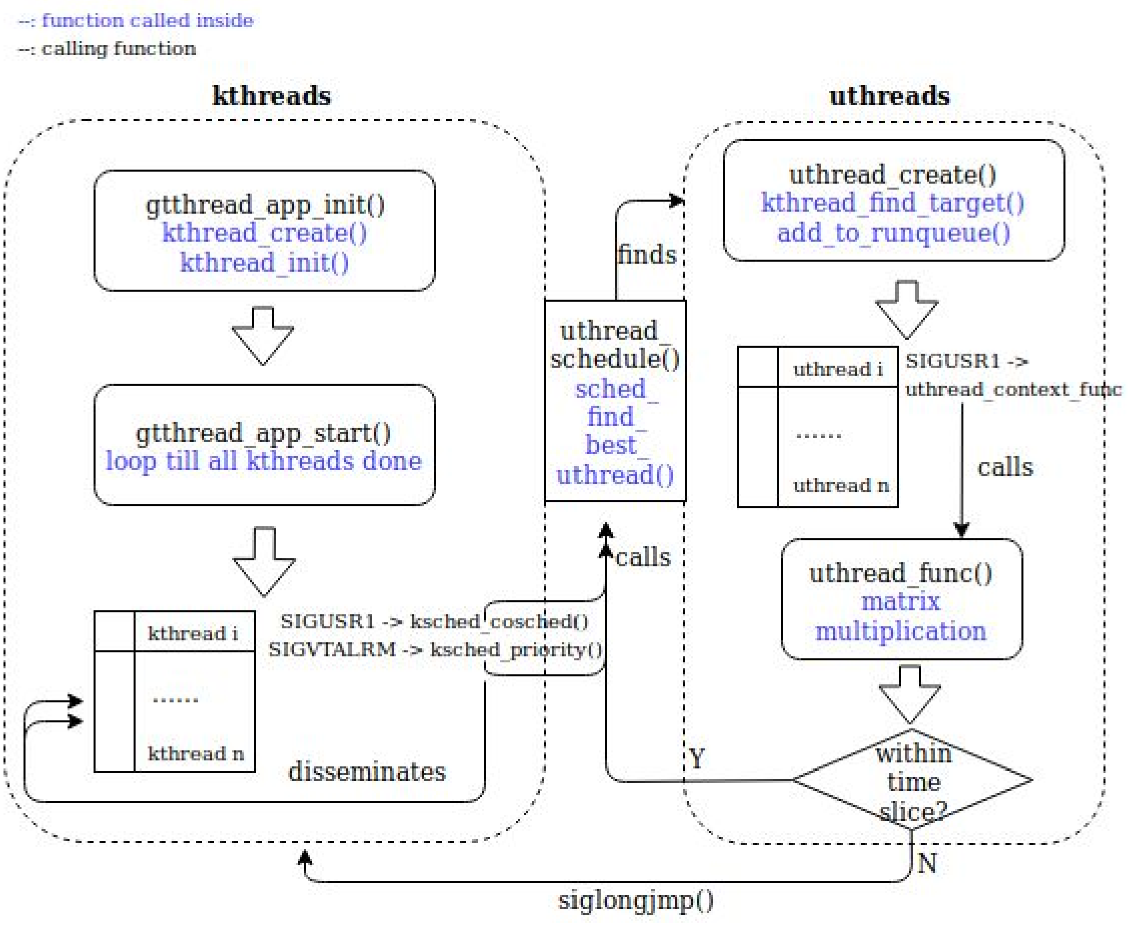
CS6210 Project 1 Report

Kailun Li
GTID: 903417169

1. Mechanism of gtthreads Package



(1) kthread

Creation: In the given package, a *kthread_context_t* structure is a process generated on physical CPU. At first an initial process is created, and then other processes are created by duplicating this process using *clone()* system function.

Signal handling: Each kthread is installed with two signal handlers a signal handlers (installed in *kthread_init()*): *kthread_sched_timer* and *ksched_sched_relay*. *kthread_sched_timer* catches VTALRM signal and make schedulings at the end of every time slice. Since every time the VTALRM signal can be caught only once by one kthread, this kthread is responsible to relay the scheduling signal to other kthreads by disseminating the SIGUSR1 signal. The *kthread_sched_relay* of every other kthread catches the SIGUSR1 signal and make a scheduling.

(2) uthread

Interaction with kthread: the *uthread_struct_t* structures are threads running under the existing processes (kthread). At the end of every time silce, the *uthread_schedule* function is called within the signal handler *kthread_sched_timer*. It will choose the next uthread held by current kthread, and jump to the context of the chosen uthread.
Signal handling: A uthread is installed with a *uthread_context_func* to response to SIGUSR2 signal. On initialization of each uthread, it make a system call to generate SIGUSR2 signal so that *uthread_context_func* , which executes the actual task, gets executed. If the execution finished within a time slice, it will call the *uthread_schedule* to find the next uthread to run.
(3) Structures in Other Files
The structures in other files can be viewed as auxiliary structures to implement the kthread/uthread mechanism.

2. O(1) Scheduler

In the package, the O(1) scheduling algorithm is implemented in *sched_find_best_uthread()* function, and is passed in as the input argument of *uthread_schedule* function. Before calling the actual O(1) scheduling algorithm, the *uthread_schedule()* function first deals with the last executed thread; then, it calls the scheduling algorithm to find the next uthread, and jump to the context of the next uthread.
The uthreads in a kthread are held in *kthread_runqueue_t* structure. Within this structure, there are 3 mutually exclusive run queues: *active_runq* holds the uthreads to be scheduled in current round; *expires_runq* holds uthreads that have already been scheduled in current round; *zombie queue* holds the uthreads that have completed their tasks and thus should no longer be scheduled.
The work flow of O(1) scheduler is as following:
<a> Lock the current *kthread_runqueue_t*.
<b> When the *active_runq* is not empty, the uthread with highest priority within the run queue will be found by using the wrapped *bsfl* operation (which is of constant time) on the priority mask. The chosen uthread is pulled out of the queue, then executed, and put into *expires_runq* after execution.
<c> In another case, when the *active_runq* is empty, if the *expires_runq* is also empty, then return NULL and the current kthread is done. Otherwise, the scheduler switches the *active_runq* and the *expires_runq*, finds the and pulls out uthread with highest priority using the same method in <b>.
<d> Release the lock on current *kthread_runqueue_t*.

3. Credit-Based Scheduler

In this scheduling algorithm, every virtual CPU (in our case, the role is played by uthread) has a <weight, cap> pair. The *weight* (initial credits) specifies the relative CPU time that this vCPU should be given, and the *cap* specifies the upper bound of CPU time allotted to it. Each CPU (in our case, kthread) has two run queues: UNDER queue and OVER queue.
The work flow of this scheduler is as following:
<a> As uthread runs, it consumes its credit in proportion to the CPU time it has taken.
<b> At the end of a time slice, if current uthread has negative credits, it is put in the tail of an OVER queue which means it will not be scheduled in this round; if credits is positive, it is inserted at the tail of UNDER queue which holds uthreads that will be scheduled later in this round.
<c> When the UNDER queue is empty, it switches the OVER queue and UNDER queue, and bumps the negative credits in current UNDER queue to positive, and starts the next round.
<d> If a kthread find both of its queues are empty, it will take the uthread from other uthreads from the UNDER queue of other kthreads and balance the work load.

4. Design of Credit-Based Scheduler

(1) Credit Scheduler without Load Balancing

Since the major interaction between the kthreads and uthreads is *uthread_schedule()* function, I decided to add another choice for the function pointer that is passed to this function (i.e. add another scheduling function that is at the same level as *ksched_find_best_uthread()*). The major changes needed are as following:
<a> Change the KTHREAD_VTALRM_USEC to 30000 (time slice is 30 ms)
<b> Add 4 fields to the *uthread_struct_t* structure: *initial_credit* (which is the initial weight of this uthread), *credit* (current credits left for this uthread), *last_start_time* (the last time this uthread gets scheduled and began execution), and *tot_cpu_time* (the actual cpu time this uthread consumed). The *cap* in described in Xen Wiki is not used in my implementation.
<c> Add another function *ksched_get_head()* as a second scheduling choice in addition to *ksched_find_best_uthread()*. Compared to O(1) scheduling algorithm, it uses the *active_runq* as the UNDER queue, and uses the *expires_runq* as the OVER queue. On a queue switch, I choose to shift the negative credit values in OVER queue by the uthread’s initial credit (if result after the shift is still negative, assign it to value 1). I use this shifting approach because at the start of a single time slice, a uthread might have a positive credit value that is less than what will be consumed in this time slice. But we still let it execute for the whole time slice, which means this uthread has taken longer CPU time than it is supposed to. So in the next round, the

shifting approach will make it “compensate for” the extra CPU time it has taken. If we set the negative credit value directly to its initial credit, it might end up taking extra CPU time for nothing. It is found in my experiment that this shifting approach can help the initial credit make more difference when two uthread have same workload.
<d> To make the user able to decide the scheduling algorithm using command-line argument, I add a new *uthread_sched_alg* to the *ksched_share_info* structure which is shared by the program, and pass the command-line argument to it. Every time uthread_schedule is called, choose the scheduling algorithm to pass to it according to the value of this field.
<e> In *uthread_schedule()*, after the next uthread to run is found by the scheduling algorithm, update the *last_start_time* field of this uthread to start counting the actual CPU time it will take. And at the beginning of *uthread_schedule()*, calculate the difference between the current time and *last_start_time* of the current uthread. Add add this time interval to the *tot_cpu_time* field of this uthread. Subtract the credits of current uthread proportional to this interval (in my case, it’s 1 credit for 2 microseconds), and insert the uthread to the tail of UNDER or OVER queue depending on its credit left.
<f> add an array in *kshed_shared_info* to store the *tot_cpu_time* of each uthread. When all kthreads are done, return a copy of this array from *gtthread_app_exit()* function and use this array in main function to print out the CPU time consumed by each set of uthreads.

(2) Load Balancing
In *uthread_schedule()*, when we find that both UNDER and OVER queues of current kthread are empty, we iterate through other kthreads that are not done yet and UNDER queues are not yet empty, lock the kthread_runqueue of that kthread, and pulls the tail of its UNDER queue as the current thread of the empty kthread.

(3) gt_yield()
Call the *uthread_schedule()* function to yield to the next uthread to be scheduled.

5. Implementation Results

* Raw time stands for time from uthread creation to finishing, cpu time stands for cpu time actually taken
* The combinations of credit and matrix size are presented in a nested for loop, in which outer loop iterates through matrix sizes {32, 64, 128, 256} and inner loop iterates through initial credits {25, 50, 75, 100}
* Machine used was advos-05. Some of the results in the same set vary a lot, but the outcome from my local machine (which runs ubuntu 18.04) have more uniform values within the same uthread set (the trend are similar to that on advos-5 -- not apparent in two smaller matrices, but observable in two larger matrices).

raw time (from uthread creation to completion):

uthread set#	mean (us)	standard deviation (us)
0	323010	321563
1	325883	319235
2	326337	319228
3	326769	319208
4	333878	324271
5	338555	322412
6	354712	322461
7	357370	322389
8	1789848	1617546
9	1291584	1688005
10	1018436	421827
11	640717	439480
12	4417419	1866279
13	4160425	2079901
14	3603304	2157008
15	4558036	2272419

CPU time (time actually spent running)

uthread set#	mean (us)	standard deviation (us)
0	4453	9135
1	2430	5105
2	451	142
3	437	125
4	7125	7816
5	4686	5248
6	16162	11246

7	2662	261
8	80255	39909
9	72643	37024
10	72174	34683
11	66913	31624
12	652141	267629
13	629142	298990
14	615542	294699
15	634549	268524

6. Implementation Issues

(1) Deadlocks when running towards the very end.

Possible reason: the kthreads might kick few uthreads around before finishing all executing all uthreads.

(2) The decremented credit of a certain uthread might be larger than what could be consumed in an entire time slice.

Possible reason: the cpu time accounting mechanism in my code might be incorrect for some cases. Also, I noticed that in *ksched_priority()*, the *uthread_schedule()* function is called for current kthread after it has relayed the scheduling signal to all other kthreads. Since I wrote the CPU time calculation in *uthread_schedule()*, this might make this kthread take more CPU time before calling *uthread_schedule()*. Putting *uthread_schedule()* before relaying the signal causes stack smashing, which I didn't make to solve.

(3) The mean and standard deviation of CPU time is calculated using the copy of array in *kshced_shared_info* structure returned by *gtthread_app_exit()*.

Possible improvement: calculate and print out the value during execution instead of after exit the app.