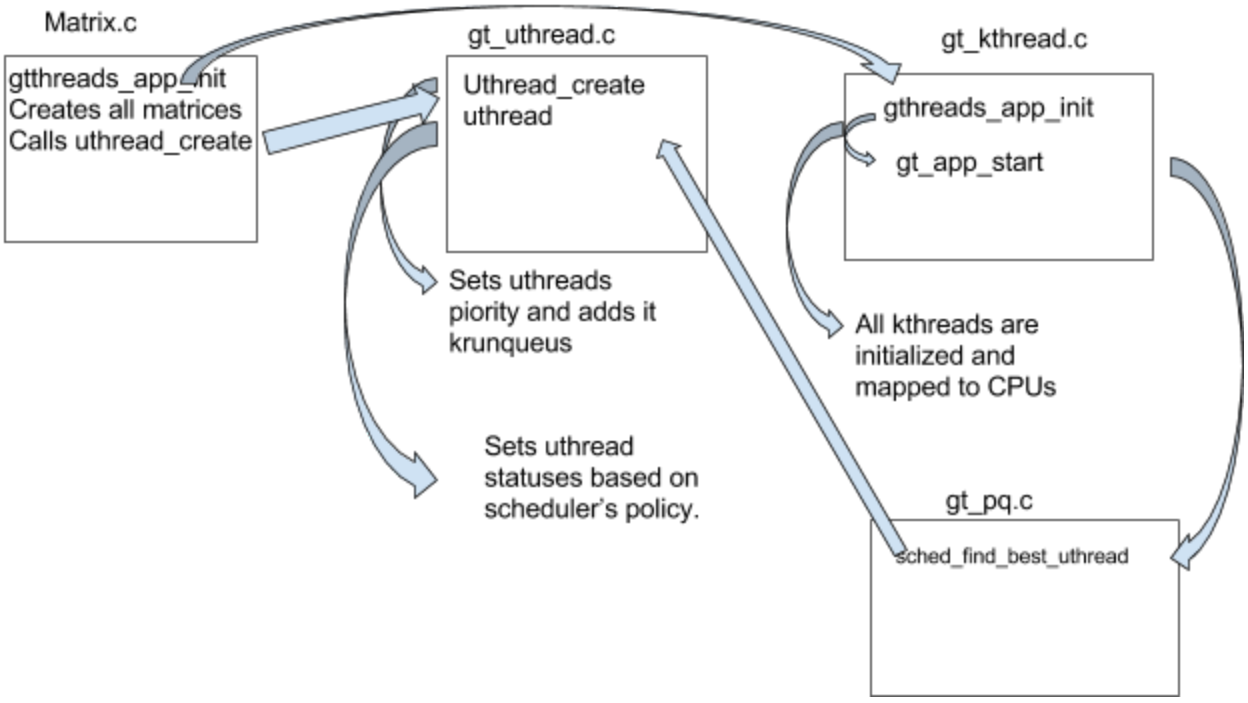
Project 1 Credit Scheduler

Write a report summarizing the implementation of the project

“GTthread is a user-level thread library” where the Kthreads, the kernel threads, manage CPU resources based on the chosen scheduler’s policy, and allows the uthreads, user threads, to make use of the CPU in a desired manner. Each Kthread has two runqueues each: an active runqueue and and an expired runqueue. Every uthread is associated to a kthread upon creation. When a timeslice for a given uthread reaches 0, the task is moved to the expired runqueue -- O(1) per scheduling operation. When the active runqueue runs dry, the two runqueues are swapped, again, a O(1) operation. Hence the name O(1) priority scheduler.

Show your understanding of the given GTThreads package (max. 2 pages) - try to jot down how the O(1) scheduler in the package is implemented with simple diagrams (e.g. function interactions or flow chart).



In more detail, there are three important main components in the project - the matrix, the user level threads, and the kernel level threads. The matrices are tasks that need to be carried out by the uthreads, and the uthreads are allowed to use CPU resources

once they are linked to a kthread via the kthread's runqueue. The way a uthread uses a CPU resource is based on the type of scheduler chosen at the time of program initialization. The GTthread library provides a variety of functionalities that allow one to implement these schedulers, but also provide the O(1) priority scheduler as the default scheduler for the program. The diagram above describes, in a rather poor manner, the flow of project 1, from beginning to the main cycle. Gt_matrix.c is the entry point of the program, this files has functionalities for creating the matrices, calling the app initializer and passing the matrices to the gt_uthread_create function, that sits in gt_uthread.c. The gthread_app_init initializes all kthreads, and maps them to the machine's CPUs. Each kthread are equipped with two runqueus - active and expires, and when a uthread is created, it is added to the active runqueue, and from there on, the program makes use of the uthread_schule and sched_find_best_uthread to add the uthread to the appropriate runqueue, and to set its statuses relative to the scheduler's policy. On the standard priority scheduler, for example, each uthread starts with the same priority level. Each uthread is allowed to run on the CPU for the entirety of the timeslice, and after that, if the uthread still has work to do, it gets added to the expired runqueue. Once a kthread runs out of uthreads in its active runqueue, the scheduler swaps the two runqueues, making the expired runqueue the active runqueue thereof. The process repeats until all uthreads are done with their task, and thus, all runqueues are empty.

Present briefly how the credit-based scheduler works (max. 2 pages) - try to explain basic rules (or the algorithm) for scheduling.

A credit scheduler allows vCPUS to make use of the of the CPU resources based on the vCPU attributed weight, and cap. The weight attribute determines how much a vCPU can use the CPU relative to all the other vCPU weights on the same CPU. For example, a vCPU with a weight 100 gets to use the CPU four times more than a vCPU with a weight of 25. The cap means whether or not the vCPU will run in work-conserving or non-conserving mode. In this project, however we skipped the cap attribute and forced all kthreads to run on work-conserving mode. An important concept of the credit scheduler is the timeslice. A vCPU's weight does not determine which vCPU will run first. The vCPU will run for the entirety of the timeslice (unless it gets preempted based on implementation, like the yield functionality) and afterwards credits are deducted from the vCPU's balance. There are two states for the vCPUs regarding the credits: under and over. Negative credit puts the vCPU in priority OVER, positive, UNDER.

There are a variety of ways to implement a proportional fair share virtual CPU scheduler. The way it was designed in this project is by mapping the possible weights to timeslices. For example, a weight of 25 corresponds to a whole timeslice, whatever the

timeslice is. Therefore, a vCPU with a weight of 50 gets to run for two timeslices before it gets moved to the expired runqueue. That's two times more compared to the vCPU with the weight of 25. Another implementation feature worth mentioning is that each vCPU, at the time of its creation, is assigned the same amount of credit as its weight e.g a vCPU with weight 25 starts with 25 credits. Also a vCPU runs only for the entirety of one timeslice before it gets put back on the runqueue relative to its priority state, OVER or UNDER. That prevents a vCPU with a higher credit from monopolizing the CPU for, say, four timeslices before the vCPU with a weight of 25 can start running.

Sketch your design (max. 2 pages) - try to show your implementation plan of the credit-based scheduler involved in the given GTThreads package, that is, how to modify the given package for introducing credit-scheduler into it

There are key functions and attributes that can be modified on the GTthread library to make the implementation of the credit scheduler possible. The main changes made for this project were:

Gt_matrix.c

- Modified matrix_mulmat() so it would make one uthread multiply an entire matrix.
- Modified the main() so it would generate all 16 combinations of weight-matrix size needed, and well as invocating the uthread_create 8 times per combination. It's also where you get to set the scheduler type, which is passed as an argument to the gtthread_app_init, and stored in the kthread_shared_info struct.
- Added statistics functionalities as per the project requirement.

Gt_uthread.c

- Introduce new instances to the uthread_t struct to accommodate the demands for the credit scheduler, such as weight, credit etc.
- Uthread_create → record the time in which the uthread is created, which will be used to calculate the total time. Set the credit = weight.
- Uthread_schedule 1 → set the time in which the uthread starts running, and the time in which it gets preempted, the difference between these two gives you the the CPU usage on the current run. Credits should be deducted based on this difference. For example credit = credit - 25(preempted_time - scheduled_time)/timeslice.
- Uthread_scheule 2 → In case a uthread runs out of credit, replenish its credit based on its weight and add it to the expired runqueue, otherwise, add it back to

the active runqueue. Also a uthread is set to yield, it should be removed from the active runqueue and added to the expired runqueue.

Gt_pq.c

- Uthread_find_best_sched → This functionality is important. It takes a knruqueue as an argument, and returns the the best uthread to the scheduled. In the case of the credit scheduler, if a kthread's active runqueue happens to be empty e.g if(!(runq->uthread_mask)), we search for other kthreads with uthreads in their active runqueue, and add it the current kthread that ran dry. Only after all the other kthreads run out of uthreads in their active runqueue we start looking into the expired runqueues. When looking into other kthreads runqueue one need to be careful with race conditions. Therefore, the spinlock library should be used accordingly, but the priority scheduler offers enough example that can be used in this case. In short, a kthread locks its krunqueue before it starts doing the above operations, only after the kthread is done looking into, removing or adding to its runqueue should it release the lock. The above implementation can be thought of the load balancer.

Results without yielding.

W	Size	m_t_t	std_t_t	m_runtime	std_runtime
25	32	193	468045.99788	171	40.00000
25	64	1403	466836.00506	1381	1170.00000
25	128	33247	434992.00677	33060	32848.99950
25	256	3884108	3415868.46443	272560	272349.00527
50	32	33489	434749.99793	199	12.00000
50	64	138003	330236.00618	1733	1522.00000
50	128	62695	405543.99803	29226	29014.99971
50	256	3918012	3449772.56533	239766	239555.00078
75	32	76090	392148.98981	172	39.00000
75	64	241278	226961.00106	1382	1171.00000
75	128	252918	215320.99736	67506	67294.99952
75	256	817345	349105.99715	263918	263706.99668
100	32	141450	326789.00114	175	36.00000
100	64	353203	115035.99882	1399	1188.00000
100	128	253927	214311.99627	110234	110022.99920
100	256	885307	417068.00013	274730	274519.00113

Results with Yielding.

Thread with weight = 100 and id = 112 is YIELDING
Thread with weight = 100 and id = 124 is YIELDING
Thread with weight = 100 and id = 124 is YIELDING
Thread with weight = 100 and id = 120 is YIELDING
Thread with weight = 100 and id = 125 is YIELDING
Thread with weight = 100 and id = 121 is YIELDING
Thread with weight = 100 and id = 126 is YIELDING
Thread with weight = 100 and id = 127 is YIELDING
Thread with weight = 100 and id = 122 is YIELDING
Thread with weight = 100 and id = 124 is YIELDING
Thread with weight = 100 and id = 125 is YIELDING
Thread with weight = 100 and id = 126 is YIELDING
Thread with weight = 100 and id = 127 is YIELDING
Thread with weight = 100 and id = 123 is YIELDING

W	Size	m_t_t	std_t_t	m_runtime	std_runtime
25	32	223	268761.99350	184	16.00000
25	64	1455	267529.99790	1422	1254.00000
25	128	20858	248127.00129	20675	20507.00017
25	256	2736697	2467711.99668	237735	237567.00000
50	32	21052	247933.00153	210	42.00000
50	64	93111	175874.00290	1787	1619.00000
50	128	51153	217831.99868	30132	29963.99973
50	256	2758978	2489993.02440	216410	216242.00251
75	32	51307	217678.00014	178	10.00000
75	64	192549	76436.00073	1383	1215.00000
75	128	71766	197218.99852	20483	20315.00017
75	256	584646	315661.00217	192106	191937.99532
100	32	71920	197065.00012	174	6.00000
100	64	292478	23492.99947	1388	1220.00000
100	128	268248	737.00000	29974	29805.99993
100	256	2680523	2411538.04494	256170	256001.99999

Comments:
It can be noted that the the matrix size 256 provided enough work to make the credit scheduler effect noticeable. That is to say, the combination weight-25-matrix_size-256 took roughly four times longer from the time of its creation to finish compared to the combination weight-100-matrix_size-256, which was expected. However, when yielding is in place (the yield function was set to make the combination

weight-25-matrix_size-256 to yield once) the combination weight-25-matrix_size-256 ends up taking almost the same amount of time to finish as the combination weight-25-matrix_size-256.

Problems with the code

There are ways to make the code more efficient, specially on the load balancer side. Also, the code is filled with hard-coded numbers because of my lack of experience C and pointers prevented me from making variables available on other parts of the code. With the experience I gained on the project, and thanks to the TAs I could make my code a lot more efficient now but time is up.

Directions on the Make file.
Make
Make matrix
./bin/matrix 1 (1 for credit screduler)