Advanced OS Project1 Report

Written by: Joshua Sampson

This project required implementing three main functionalities: a credit scheduler, a yield function, and load balancing. To do so, a thorough basis of understanding with the GTThreads package was required. First and foremost, there is the existence of uthreads and kthreads. Kthreads are the actual threads that the kernel is aware of and schedules on each CPU core. As for the creation of kthreads in the application, gtthread_app_init (inside of gt_kthread.c) creates "the number of cores"-kthreads (i.e. if a device has 4 cores, upon execution, 4 kthreads would be created), and these threads are created in the same file in kthread_create. These kthreads then begin executing the gtthread_app_start and stops executing when there are no more threads in either run queue. Uthreads on the other hand are threads that can be assigned work to be completed. These uthreads are scheduled by each kthread which maintains its own run queues, one for active threads and one for expired threads. These are used in the priority scheduler provided and in the credit scheduler that was added on and will be touched on later in this report. Uthreads are created in uthread_create inside of gt_uthread.c. and are then added to their kthreads active run queue. After these threads are initialized, we are now in the execution phase of the application and these uthreads are constantly being scheduled and "de"-scheduled. The bulk of the remaining execution phase largely depends on uthread_schedule inside of gt_uthread.c. This function is executed given a few circumstances: the uthread has finished running entirely, the uthread has yielded (via the gt_yield), or a timer interrupt occurred (via SIGVTALRM). When executed, this function is responsible for scheduling uthreads on each kthread. In the original codebase, the scheduling decision was made via an O(1) priority scheduler.

Before jumping into the specific details as to how the O(1) priority scheduler is implemented, it's important to understand its overall behavior first. When a thread is preempted let's say, we need to select another thread to run. To do so, we want to look into the relevant kthread's active run queue and select the highest priority uthread in there. With this selected thread, assuming it is ready to run, we jump to its thread context. Now taking a closer look, first and foremost, the scheduler's O(1) time complexity relies solely on two masks: uthread_mask and group_mask. uthread_mask is used to tell the scheduler which priority levels have at least one uthread. The "lowest bit set" is equivalent to the lowest priority level, which in this implementation of the priority scheduler, is treated as the uthread with the highest priority (thus is the favored one to be scheduled next). After finding this lowest bit set (group of uthreads with the highest priority), we can use this value to index inside of prio_array which is an array defined as each index corresponding to a priority level 'i' (i.e. prio_array[2] contains all the uthreads that have a priority level of 2). Once we index into the prio_array, we can then use group_mask to tell us which groups have at least one thread. Using this data, we can index into the list of groups (where each index 'i' corresponds to uthread_group 'i') knowing we have a non-empty group and can select a thread from it that is runnable. All of this is how we select a thread to run. Now what if the current kthread has an empty active run queue, meaning that it has no more uthreads to run? For this scenario we acknowledge that kthread_best_sched_uthread will return 0 inside of uthread_schedule (when trying to select this "next thread" to run), which will let to the handler of a completed kthread which also returns back to our main function.

Now for the self-implemented scheduler, the credit scheduler. Again, starting with a bigger picture explanation (not specific to the experiment), references to the Xen credit scheduler will be made. Every time a new thread is created, it is assigned some credit value. Each time this thread is preempted, its credit count is deducted by some time measurement (i.e. CPU cycles/time on the CPU) and a new thread is selected. The way a thread is selected is based off the thread with the most number of credits. In "priority scheduler" terms, the more credits a thread has, the higher the priority of the thread, and the lesser the credits, the lower the priority of the thread. After deducting credits of the preempted thread, if its credit count is still "over" 0, the thread is considered "UNDER" with regards to its credit consumption, and if its credit count has now fallen to 0 or smaller, the thread is considered "OVER" (it has exceeding/is over its allotted credits). In the former case, we add the thread back to the run queue, and in the latter, we assume the existence of an expired run queue (which is necessary for a credit scheduler implementation) and add it to that. The credit scheduler also has a way of replenishing credits back to their original credit assignment when there are no more credits to run because all threads have exceeded their initial credit assignment, for example when all threads have exceeded their initial credits

Now taking a detailed explanation of the implemented credit scheduler, we start with looking at gt_uthread.h. Relevant fields that were added to assist in the implementation are "uthread_credits", "uthread_init_credits", "is_over", "start_time". First "uthread_credits" is used as a way to track live credits. "uthread_init_credits" is used when replenishing the credits once all uthreads within a kthread have exceeded their credit assignment. Then we have "is_over" which is used to mark uthreads that have either consumed all their credits or not to determine whether or not a uthread should be moved back to its kthread's active or expired run queue. Finally, "start_time" is used to mark once a uthread actually starts running and is used to assist in the calculation of the total elapsed time ran once the same uthread is preempted, to precisely determine how much to take away from its credits. Note that this field is set inside of uthread_schedule right before the selected thread begins executing again or for the first time. This field will change as the thread gets preempted and then reselected. Also note that in this implementation, when credits are assigned to each uthread, they are being scaled to the microsecond level, for example, given that these uthreads can either have 25, 50, 75, or 100 credits, these values have been multiplied by 1,000,000 because the GTThreads package works at the microsecond level. With these additional fields, we move to the scheduling logic which can be found in gt_uthread.c and gt_pq.c.

We start with uthread_schedule which behaves similarly to its description when discussing the priority scheduler but with a few modifications. Again, this function is called when the current thread has either yielded, been preempted, or completed executing. The first main implementation decision is where to deduct credits after the current running thread calls uthread_schedule. This is done inside of this function before marking the thread as "runnable". This is because we need to decide whether we want to add the thread back to the active run queue or empty run queue. To deduct credits, the "start_time" field is utilized. The current time stamp is captured and the difference between the two is then calculated, producing the total time it has ran since being removed from the run queue into a "running" state to being put back onto a run queue. After deducting credits, we then select the next best uthread to schedule with a call to sched_find_best_uthread_credit within gt_pq.c; this is where the credit scheduler is implemented.

With regards to how to store threads it was decided to reuse the priority hash table design provided. This is because the existing runqueue_t data structure is central to how threads are managed, and touching that, and alongside it, thread management, would have been a much more difficult task. Of course, it helps that it already serves as a priority queue. In addition, the priority hash table has O(1) access times meaning that if we were to keep a queue of credits, we would have to iterate over the entire queue to find the one with the highest credits. Now we could attempt to solve this by organizing our queue from most credits (at the head) to least credits (at the tail). However, this doesn't get rid of the issue as we now have to know where to place a new thread entering the run queue. To do this, in the worst case, we must go through the entire list only to find out we need to place it at the end of the queue. With this design decision established, let's further explore the sched_find_best_uthread_credit function. Like the priority scheduler, if our active run queue is empty, we switch it for its empty run queue. Assuming the empty run queue is filled up, we now have active threads to run. But recall that for a thread to be placed in the expired run queue, it must have ran out of credits. Thus, all threads now have <= 0 credits and must be replenished (to their initial credit amount). This is all handled accordingly. Once we move past this stage, we know we have valid threads to run. We then perform the same iteration across the priority hash tables but now with a difference goal: selecting the uthread with the most credits. Once we do so, we remove this selected uthread from its run queue and finally return from the function. We now resume execution back in uthread_schedule with our newly selected thread in hand. All we need to do now is jump to its context (alongside other specific details beforehand) and the uthread is now being run.

Finally, for the last part of this report, let's touch on how the gt_yield function and load balancing is implemented. First off, gt_yield is called halfway through the matrix multiplication completion inside of uthread_mulmat. Here, we have a very simple function. If we are running the priority scheduler, we immediately make a call to uthread_schedule to schedule the next best thread and if we are running the credit scheduler, we also do the exact same thing. Note that we pass different arguments for both as different functions handle the different schedulers. The decision to incorporate this simple yield function was solely based on the idea of centralizing all credit deduction and scheduling details in uthread_schedule and only in uthread_schedule. As for load balancing, this is handled in three key functions, kthread_load_balance, kthread_lb_priority and kthread_credit_priority. In kthread_load_balance, the decision was made to select a kthread to steal from with the most uthreads in its active run queue. This is because it seemed to be the most helpful way for load balancing to occur, relieving work that a kthread has to do (from all its uthreads) even if by just a miniscule margin. After selecting this kthread, assuming we are runningwith the credit scheduler, we call kthread_lb_credit. Here we perform the same logic as in sched_find_best_uthread_credit to select the uthread with the most amount of credits. This is to maintain the integrity of the credit scheduler "prioritizing" uthreads with the most credits. After selecting this uthread from the relevant kthread's active run queue we perform a switch where we bring in the selected uthread from the selected kthread, to the kthread that has no more work to do. Below we find results from two experiments ran with the credit scheduler with and without load balancing and further elaboration on the results.

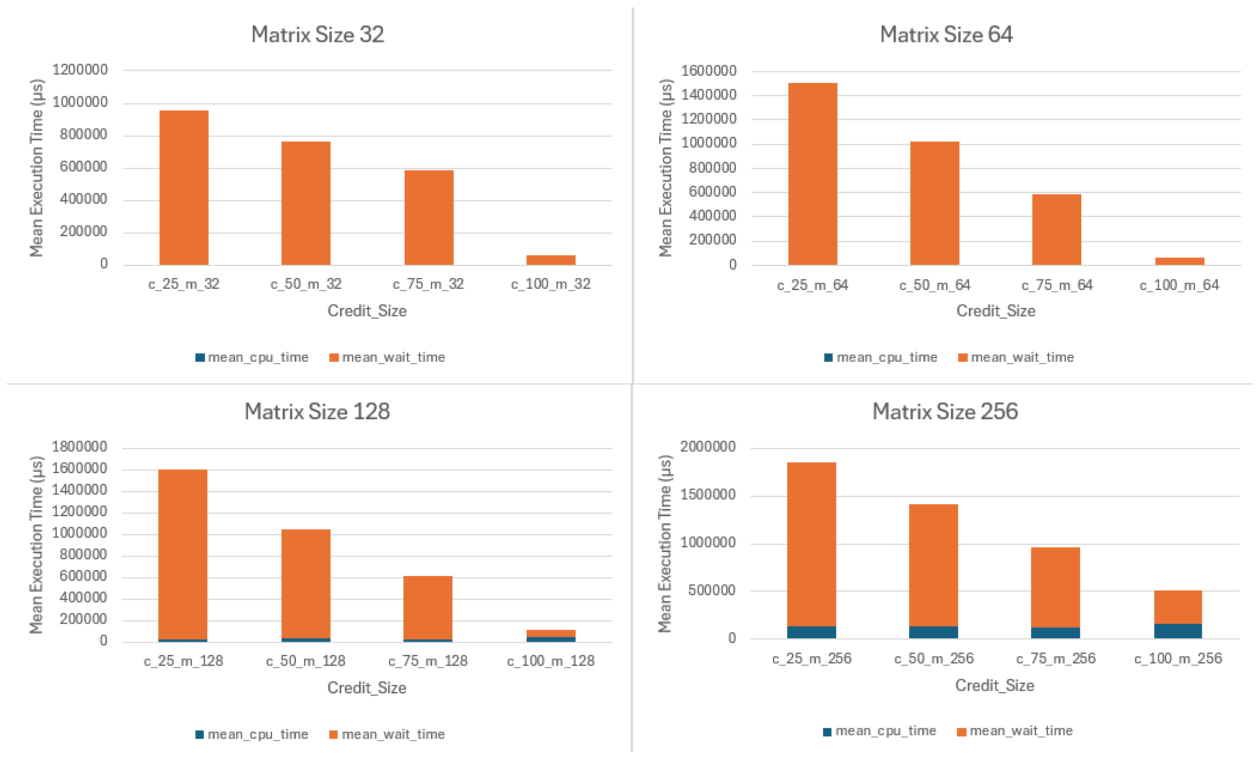


Figure 1. Credit Scheduler Results <u>without</u> Load Balancing

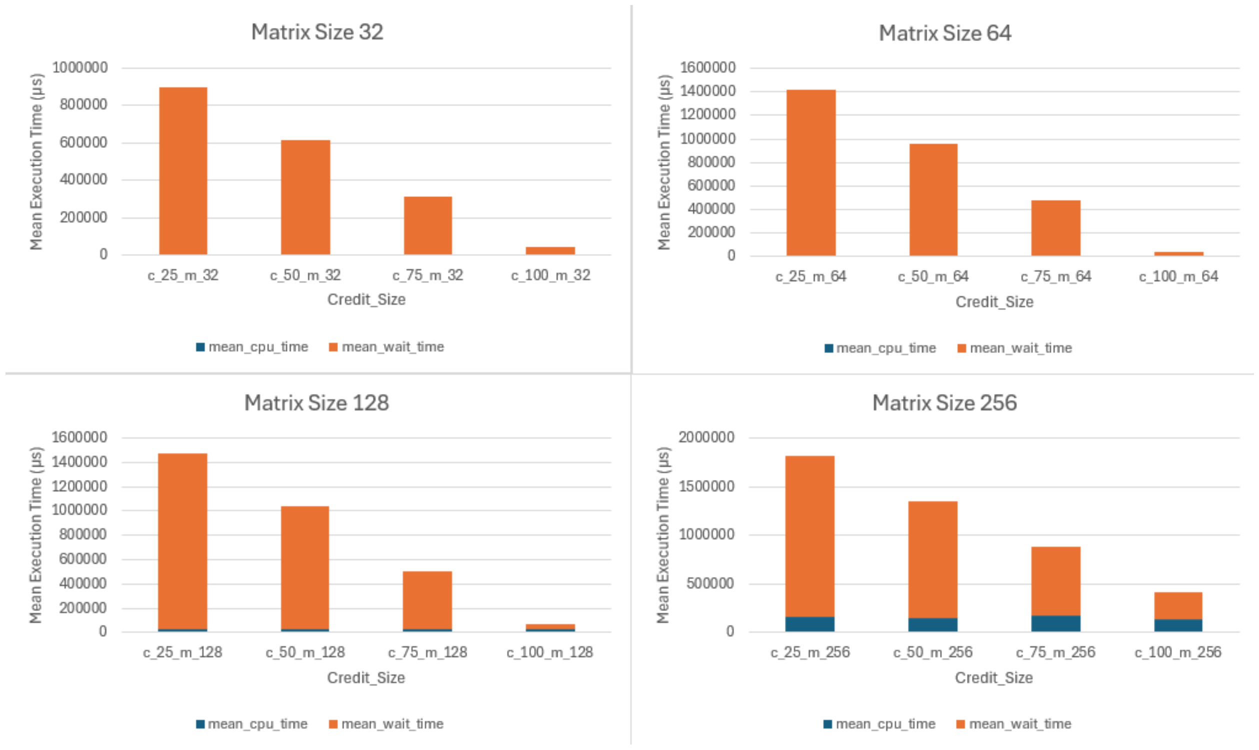


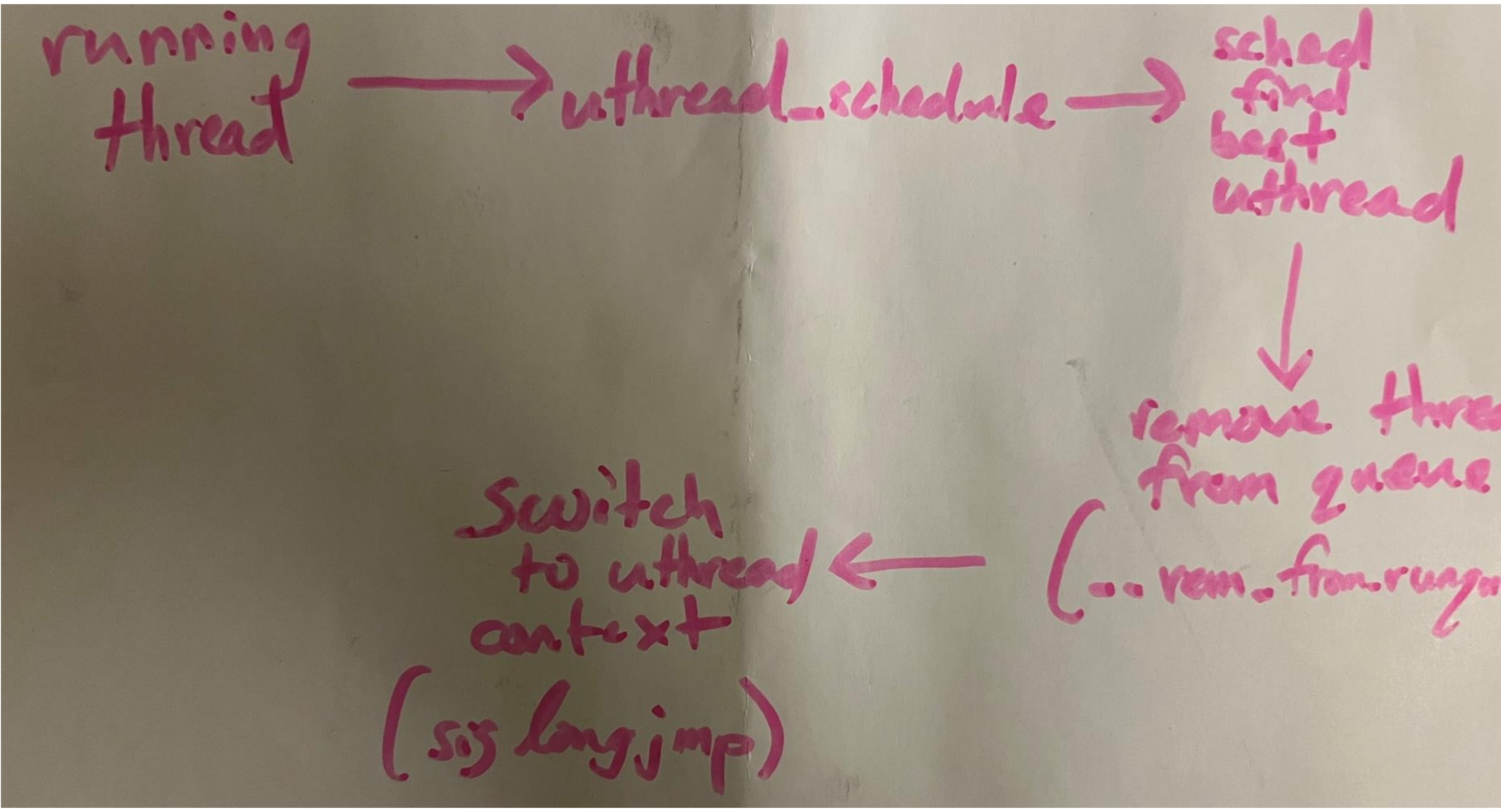
Figure 2. Credit Scheduler Results <u>with</u> Load Balancing

Based on the graphs generated of the mean cpu, wait, and execution times, the scheduler is working as expected. First let's note that while in Figure 1 and 2 for graphs "Matrix Size 32" and "Matrix Size 64" there is barely any "mean_cpu_time" visible, there is still a mean execution time present, but compared to the wait time, it is almost negligible. This is not an error of the scheduler but rather a reflection as to the effectiveness of the scheduler, meaning that it is preempting threads consistently, causing threads to wait. Further along the mean wait times, we notice that they decrease as our credits increase, for every graph. This is exactly what we should expect since we prioritize higher credits, meaning in theory we want to run all 100 credits first, then 75 credits, etc. This is exactly why the decrease in wait times occurs consistently across all graphs. We also note that the execution times where visible is as expected in the sense that they are relatively the same across all "Credit_Size" bars. This is as we expect because every thread is doing the same amount of work. As for the differences between the run with load balancing versus without it, we expect the mean cpu times to be roughly the same but the wait times to decrease as we get smaller credit groups. For example, with load balancing, once a kthread runs out of credits, it takes the best thread from another kthread to schedule on itself meaning that we schedule and finish executing the <u>total</u> higher credit threads even faster than before because of "outside" assistance. This means that we run through 100 credit threads faster, which means we get to the 75 credits faster, where we then get through those faster, getting us to 50 credit threads even faster than before, and so on and so forth. Given this expectation, the graphs match accordingly. Across every single graph, compared to its counterpart, the wait times in each bar in the "with load balancing" graphs is less even if by just a small amount than those in the "without load balancing" graphs. These findings all confirm the effectiveness and efficiency of the credit scheduler especially with the addition of load balancing. Thus with these findings, load balancing is a mechanism which improves performance.

In the case of this experiment, a credit scheduler is better as a priority scheduler can lead to starvation of lower priority tasks and if the higher priority tasks happened to be those of matrix size 256, for example, then it would take almost forever before getting to the other threads/workloads. Also, in this case, the experiment revolved around computations (matrix multiplication across small and much larger sized matrices) which is the environment a credit scheduler tends to thrive in prioritizing CPU utilization and equal distribution of resources as opposed to prioritizing only one thread.

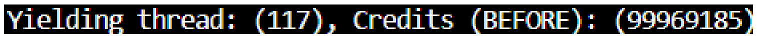
This concludes the lab report. The upcoming pages include Appendix A, B, and C.

Appendix A


running thread
uthread_schedule
sched find best uthread
remove thread from queue
(..rem_from_runqu
switch to uthread context
(sys longjmp)

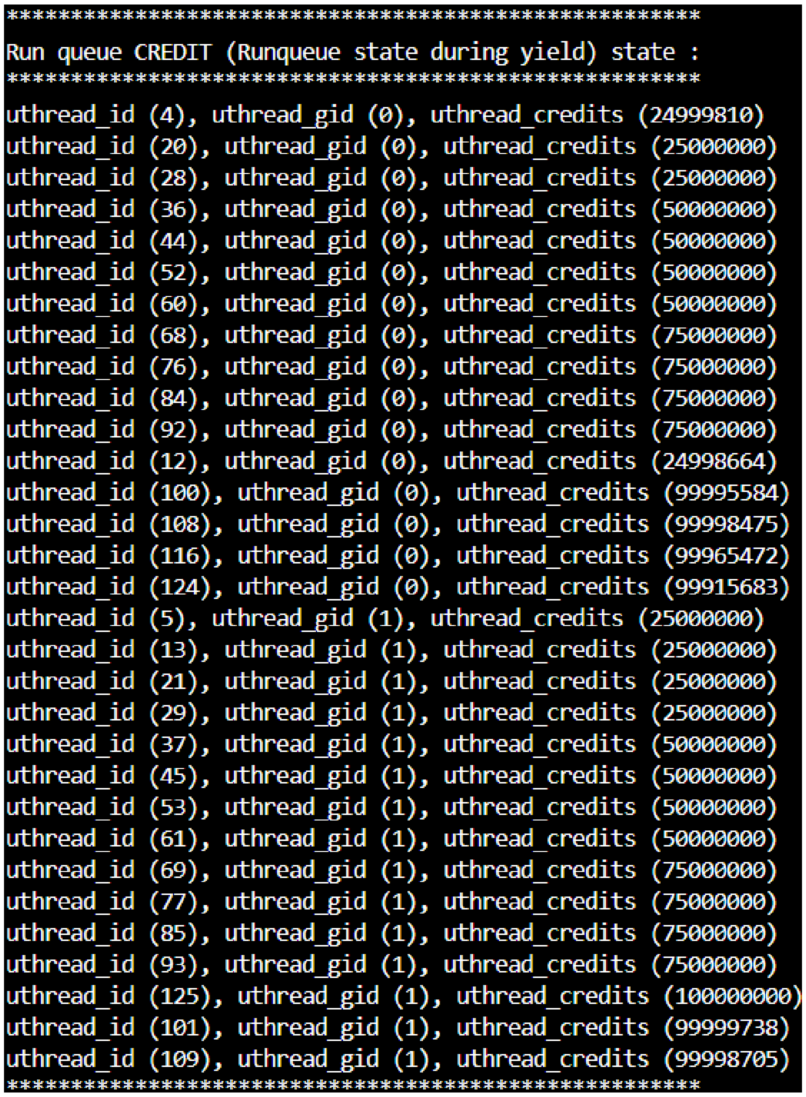
Appendix B

Printed credit <u>before</u> yield:



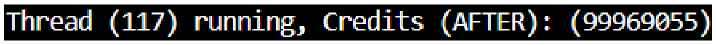
```
Yielding thread: (117), Credits (BEFORE): (99969185)
```

Run queue of uthread credits:



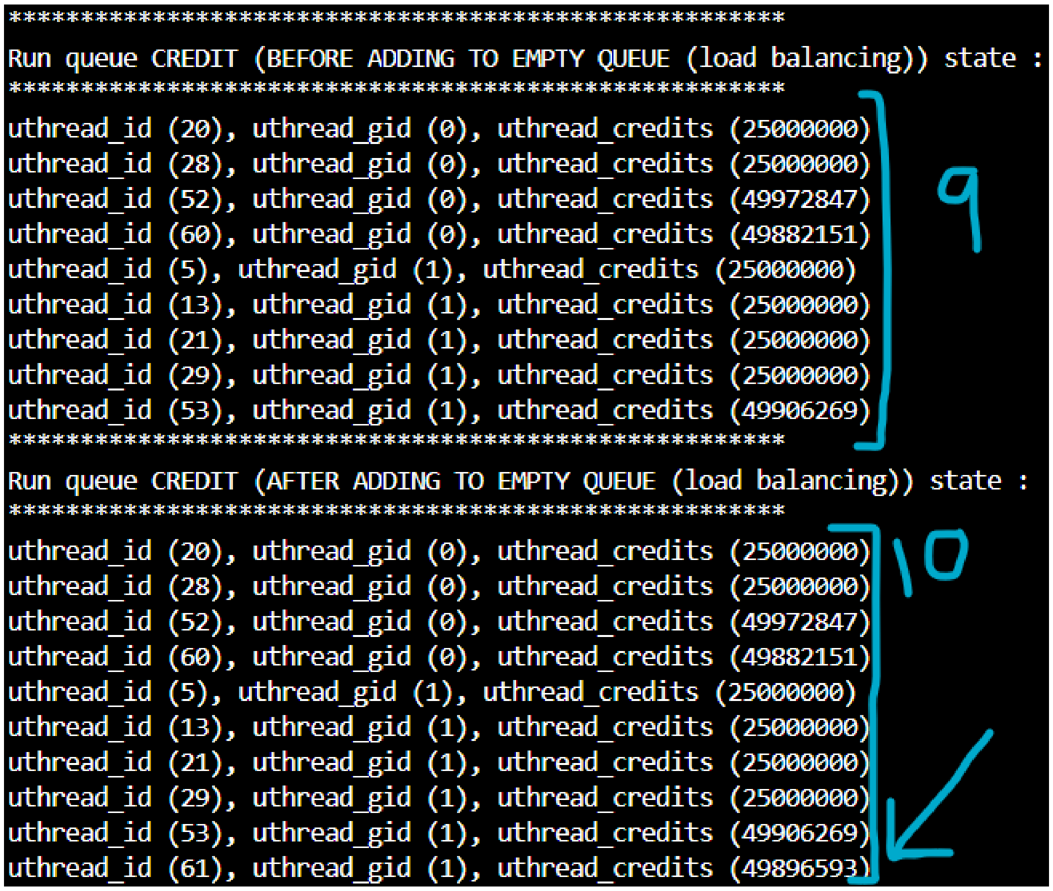
```
*******************************************************
Run queue CREDIT (Runqueue state during yield) state :
*******************************************************
uthread_id (4), uthread_gid (0), uthread_credits (24999810)
uthread_id (20), uthread_gid (0), uthread_credits (25000000)
uthread_id (28), uthread_gid (0), uthread_credits (25000000)
uthread_id (36), uthread_gid (0), uthread_credits (50000000)
uthread_id (44), uthread_gid (0), uthread_credits (50000000)
uthread_id (52), uthread_gid (0), uthread_credits (50000000)
uthread_id (60), uthread_gid (0), uthread_credits (50000000)
uthread_id (68), uthread_gid (0), uthread_credits (75000000)
uthread_id (76), uthread_gid (0), uthread_credits (75000000)
uthread_id (84), uthread_gid (0), uthread_credits (75000000)
uthread_id (92), uthread_gid (0), uthread_credits (75000000)
uthread_id (12), uthread_gid (0), uthread_credits (24998664)
uthread_id (100), uthread_gid (0), uthread_credits (99995584)
uthread_id (108), uthread_gid (0), uthread_credits (99998475)
uthread_id (116), uthread_gid (0), uthread_credits (99965472)
uthread_id (124), uthread_gid (0), uthread_credits (99915683)
uthread_id (5), uthread_gid (1), uthread_credits (25000000)
uthread_id (13), uthread_gid (1), uthread_credits (25000000)
uthread_id (21), uthread_gid (1), uthread_credits (25000000)
uthread_id (29), uthread_gid (1), uthread_credits (25000000)
uthread_id (37), uthread_gid (1), uthread_credits (50000000)
uthread_id (45), uthread_gid (1), uthread_credits (50000000)
uthread_id (53), uthread_gid (1), uthread_credits (50000000)
uthread_id (61), uthread_gid (1), uthread_credits (50000000)
uthread_id (69), uthread_gid (1), uthread_credits (75000000)
uthread_id (77), uthread_gid (1), uthread_credits (75000000)
uthread_id (85), uthread_gid (1), uthread_credits (75000000)
uthread_id (93), uthread_gid (1), uthread_credits (75000000)
uthread_id (125), uthread_gid (1), uthread_credits (100000000)
uthread_id (101), uthread_gid (1), uthread_credits (99999738)
uthread_id (109), uthread_gid (1), uthread_credits (99998705)
*******************************************************
```

Printed credit <u>after</u> yield:



```
Thread (117) running, Credits (AFTER): (99969055)
```

Appendix C



```
***********************************************************
Run queue CREDIT (BEFORE ADDING TO EMPTY QUEUE (load balancing)) state :
***********************************************************
uthread_id (20), uthread_gid (0), uthread_credits (25000000)
uthread_id (28), uthread_gid (0), uthread_credits (25000000)
uthread_id (52), uthread_gid (0), uthread_credits (49972847)
uthread_id (60), uthread_gid (0), uthread_credits (49882151)
uthread_id (5), uthread_gid (1), uthread_credits (25000000)
uthread_id (13), uthread_gid (1), uthread_credits (25000000)
uthread_id (21), uthread_gid (1), uthread_credits (25000000)
uthread_id (29), uthread_gid (1), uthread_credits (25000000)
uthread_id (53), uthread_gid (1), uthread_credits (49906269)
***********************************************************
Run queue CREDIT (AFTER ADDING TO EMPTY QUEUE (load balancing)) state :
***********************************************************
uthread_id (20), uthread_gid (0), uthread_credits (25000000)
uthread_id (28), uthread_gid (0), uthread_credits (25000000)
uthread_id (52), uthread_gid (0), uthread_credits (49972847)
uthread_id (60), uthread_gid (0), uthread_credits (49882151)
uthread_id (5), uthread_gid (1), uthread_credits (25000000)
uthread_id (13), uthread_gid (1), uthread_credits (25000000)
uthread_id (21), uthread_gid (1), uthread_credits (25000000)
uthread_id (29), uthread_gid (1), uthread_credits (25000000)
uthread_id (53), uthread_gid (1), uthread_credits (49906269)
uthread_id (61), uthread_gid (1), uthread_credits (49896593)
```