Report

Tianfa Li, 903847230

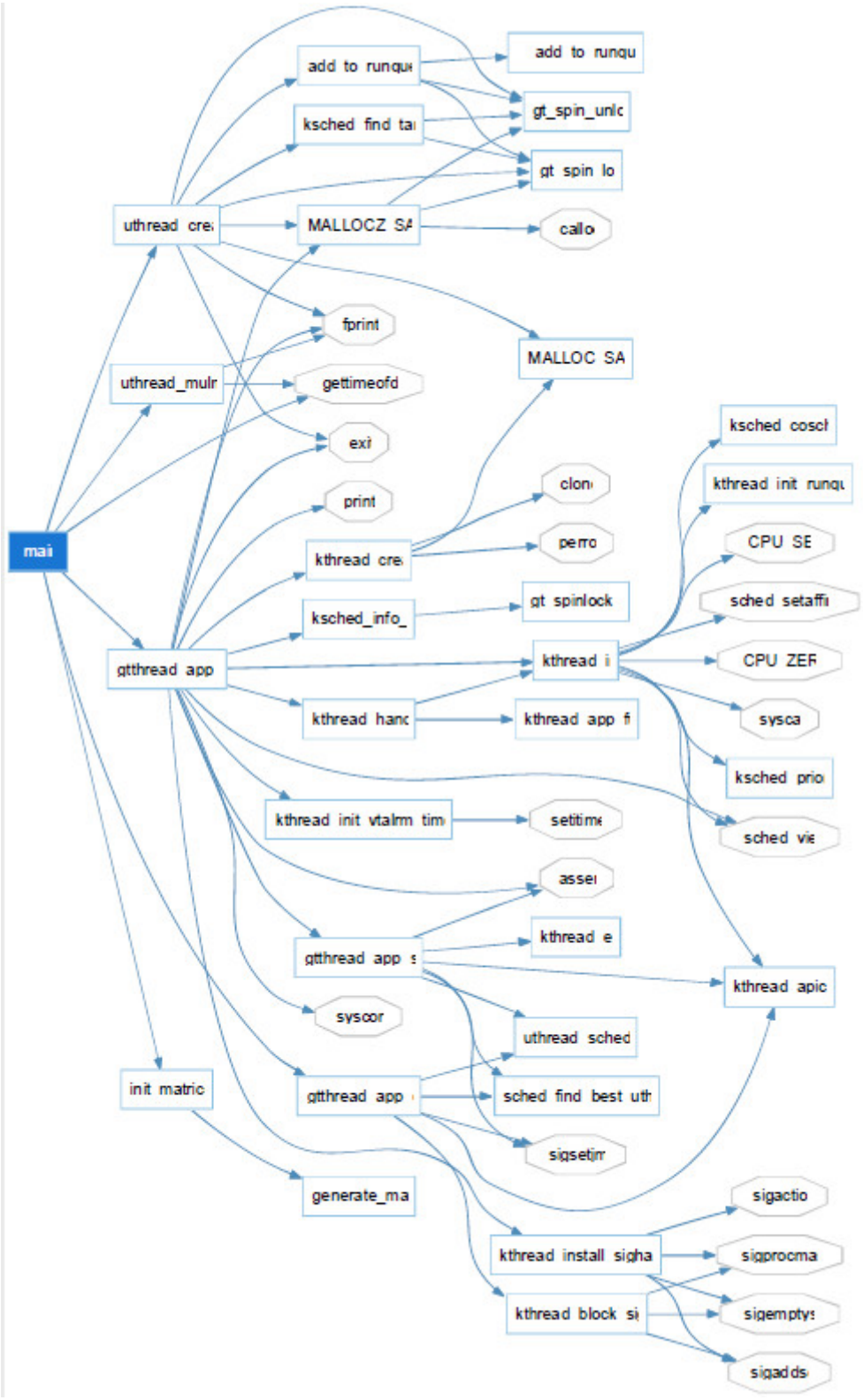
GTThreads package:

The GT Thread package contains an O(1) scheduler for the operations of adding, deleting, and selecting the next user thread for scheduling from its queue. Each kernel thread, serving as a pseudo-CPU, is associated with two run queues: one active and the other expired. Initially, all tasks are placed in the active run queue. When a task is preempted, it is moved to the expired run queue. The active and expired run queues are swapped once the active queue is empty, a process achieved by exchanging two pointers, which is executable in O(1) time.

According to the GTThread documentation, the function `sched_find_best_thread()` executes the following tasks:

1. It attempts to identify the highest priority RUNNABLE user thread in the active run queue by setting the lowest bit in the `uthread_mask` to obtain the highest priority index.
2. With this priority index, it accesses the priority queue and, through the `group_mask`, identifies the user thread group.
3. It then adds the relevant user thread to the end of the queue and removes it from the active run queue.

Overall Flow:



Credit Scheduler:

When initialized, the system allocates a default amount of credit to each user thread (uthread). This credit determines how long, in milliseconds, a uthread can run. The duration is directly proportional to the amount of credit. When a uthread is executed, it expends a portion of its credit, reflecting the time it spends on the CPU. As long as a uthread maintains a positive credit balance, it stays in the active run queue. However, once its credit is fully depleted, the uthread is transferred to the expired run queue and is recredited with the default amount to be used in future cycles.

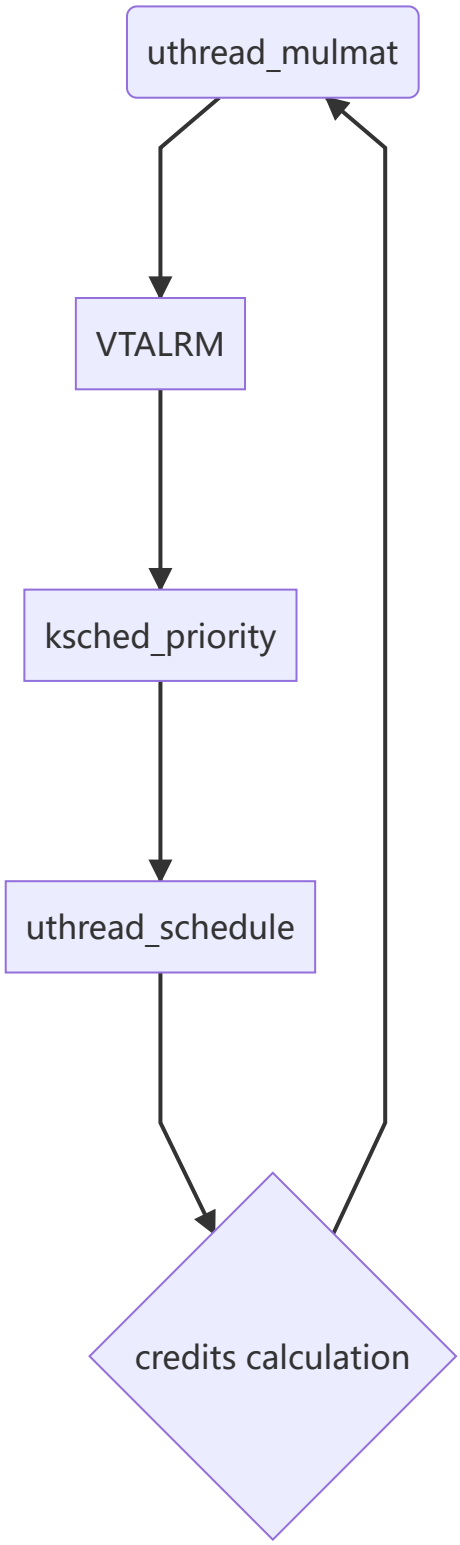
Implementation details:

Set up a new structure called credit_t and within the structure there is starting default credits for each uthread and other type of data like started-time and microseconds that consumed. In the creation of a new uthread, default and remaining credits are set using the `uthread_create()` function, with both values initialized to the specified credits. The new uthread is then added to the active runqueue with the `add_to_runqueue` function, utilizing the active runqueue and lock from the `kthread_runq`.

When a `VTALRM` signal is received by any of the kthreads, the `ksched_priority()` handler function is triggered to select a new uthread for scheduling through the `uthread_schedule()` function. This function is key in implementing the credit scheduler algorithm. It includes a process of credit caculation, which updates the timestamp and deducts credits based on the elapsed time since the last update, measured in milliseconds.

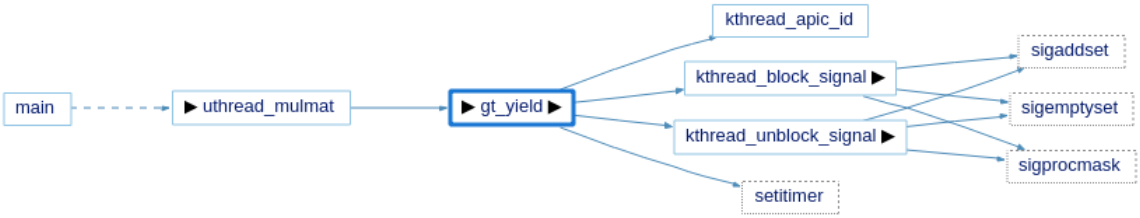
Should the uthread's credits drop below zero, it is moved to the expired runqueue; otherwise, it remains in the active runqueue. Before executing `setlongjmp()`, the function sets up a timer. Since the `gt_yield()` function may potentially change the timer, the `kthread_init_vtalrm_timeslice()` is used to reconfigure the timer.

Sketch:



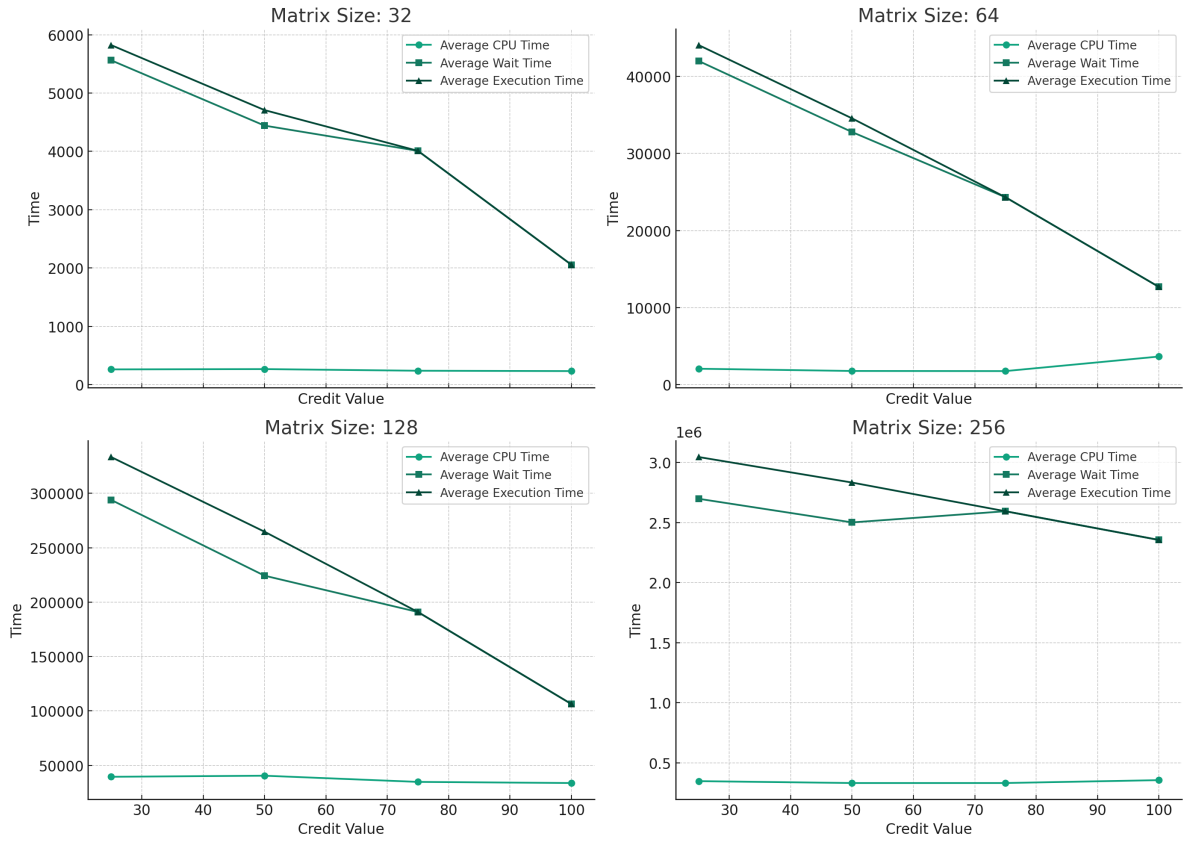
The algorithm also includes a simple load balancing step: When current kthread has no best-fit uthread and the total number of currently runnable uthreads is not zero then this kthread is marked as idle and it will iteratively find runnable uthreads in other kthread's runqueue and insert this new found runnable uthread to its own runqueue.

Additionally, to handle voluntary preemptions, indicated by calls to the `gt_yield()` API, a `yield_flag` variable is introduced. **When a uthread has finished its calculation of matrix, it will call `gt_yield()` to set a timer that triggers `VTALRM` signal and then invokes the `ksched_priority()` handler function to do scheduling.**



Workload Without Load Balancing:

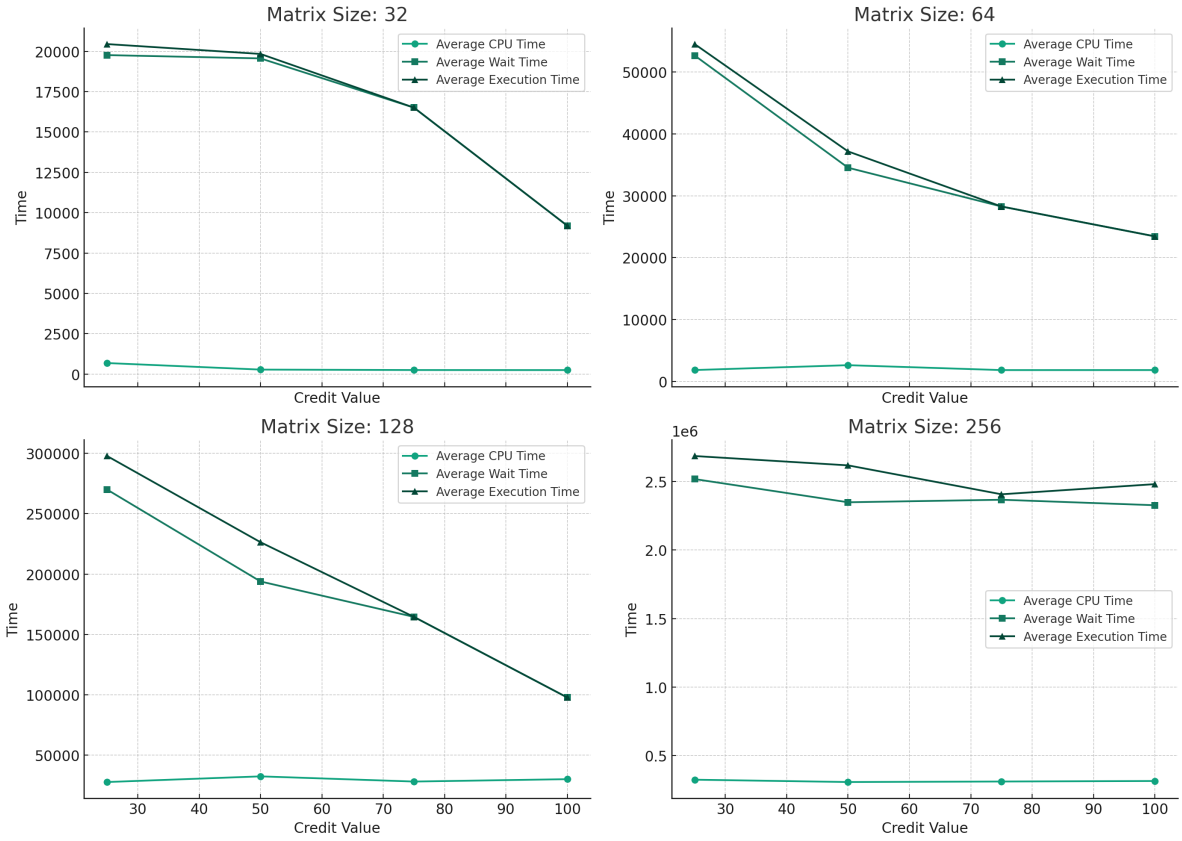
Matrix Size	Credit Value	Average CPU Time	Average Wait Time	Average Execution Time
32	100	231	2055	2055
32	75	237	4008	4008
32	50	265	4441	4707
32	25	260	5561	5822
64	100	3646	12684	12684
64	75	1760	24324	24324
64	50	1775	32782	34557
64	25	2065	42003	44068
128	100	33684	106322	106322
128	75	34725	190901	190901
128	50	40418	224293	264712
128	25	39422	293948	333370
256	100	355984	2355908	2355908
256	75	332015	2594739	2594739
256	50	331944	2501730	2833675
256	25	347645	2698528	3046173



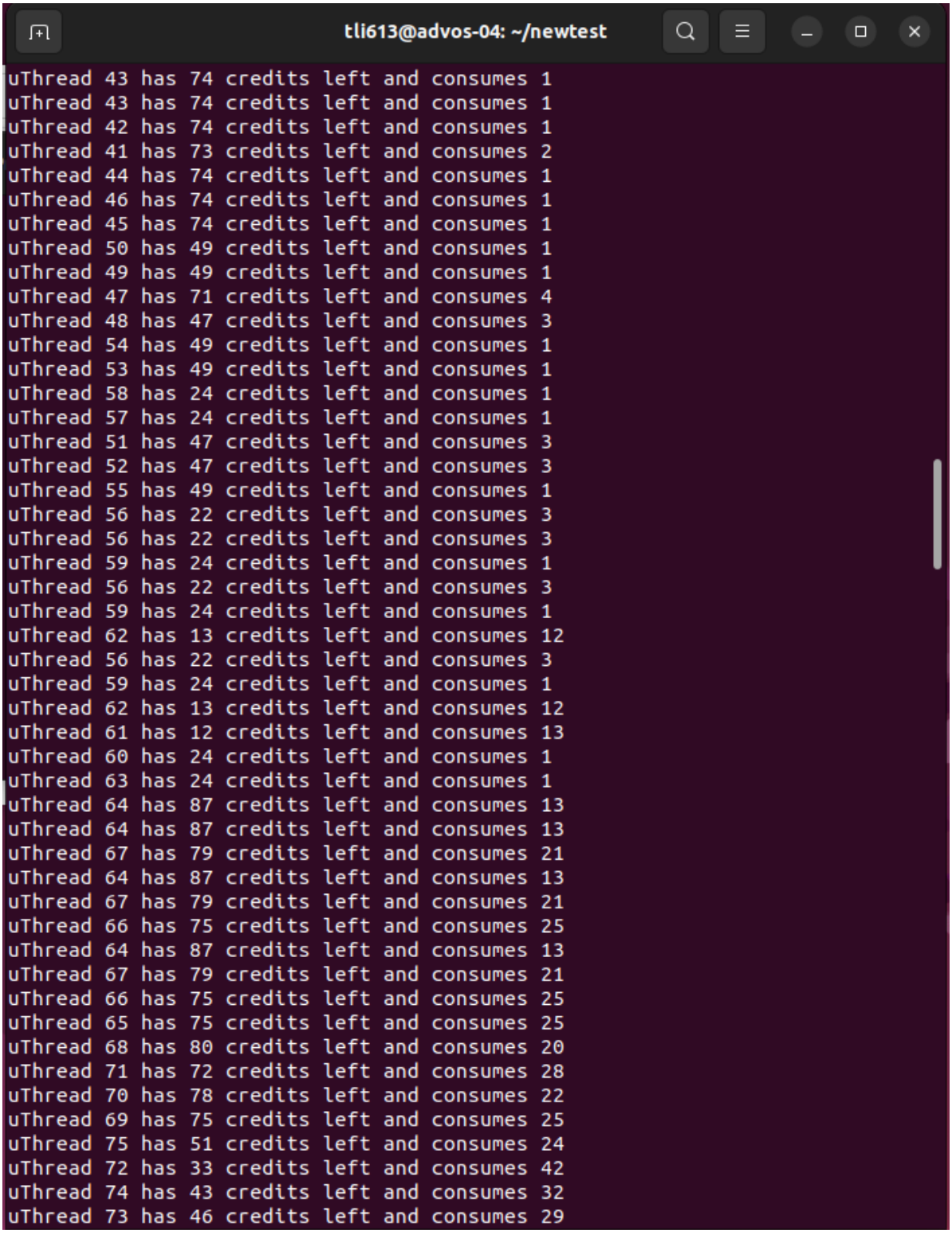
Workload With Load Balancing:

Matrix Size	Credit Value	Average CPU Time	Average Wait Time	Average Execution Time
32	100	246	9190	9190
32	75	249	16509	16509
32	50	276	19559	19836
32	25	682	19764	20447
64	100	1842	23438	23438
64	75	1838	28261	28261
64	50	2631	34525	37157
64	25	1846	52648	54495
128	100	30175	97800	97800
128	75	28199	164579	164579
128	50	32493	193922	226415
128	25	27802	270010	297813
256	100	315629	2326160	2479948
256	75	311857	2366219	2405898

Matrix Size	Credit Value	Average CPU Time	Average Wait Time	Average Execution Time
256	50	308103	2347690	2616809
256	25	325157	2516865	2684896



More credits result in less wait time while have the same CPU time. With load balance the total execution time decreases. ***In conclusion, the credit scheduler work as expect and the load balancing mechanism improve the performance. The credit scheduler perform better than O(1) priority scheduler.***



Implementation issues:

Sometimes I run into segmentation faults but trying a few more times will get the result.