Assignment 1 : Credit Based Scheduler using GTThreads

Name: Nihar Nikhil Mehta
GT Username: nmehta80

1. GT Threads package

The given GT Threads package consists of functionality to implement a priority based O(1) scheduler and a coscheduler. The scheduler is tested to perform matrix multiplication of two given matrices and store the output in the given third product matrix. The aim of the project is to implement a Credit based scheduler apart from existing schedulers.

The main function first initializes the application. This involves creating as many kernel level threads as there are number of CPU cores in the system. The kernel level threads are called kthreads, whereas the user level threads are called uthreads. Thus each kthread is associated with one virtual processor. Their signal handlers for the kthreads are installed by providing a time slice of 100ms. These kthreads have to perform the function of scheduling the best uthreads. Each kthread maintains three queues for this purpose: active, expires and zombie. After all the uthreads in the active queue have been processed for the provided time slice, the ones in the expire queue are scheduled and the active and expires queue are swapped. After a uthread has finished executing its function, it is pushed into the zombie queue.

Next, the matrices for the matrix multiplication are initialized. A total of 32 uthreads are created to perform matrix multiplication. These are divided among two CPUs, thus each kernel thread has to schedule 16 uthreads each. The rows of the matrix A are divided in 32 parts so that each uthread can multiply each part with the whole matrix B. A Uthread can have following states: UTHREAD_INIT, UTHREAD_RUNNABLE, UTHREAD_RUNNING, UTHREAD_CANCELLED and UTHREAD_DONE. During creation, the state is set to UTHREAD_INIT and it is pushed to the memory stack. Based on the group id provided, it is binded to an appropriate CPU via the corresponding kthread. Finally, it is pushed to the kthread's active runqueue.

When a kthread tries to schedule the best uthread, first it looks at the uthread which is running currently. If it is done or cancelled, it is pushed to the zombie queue. Otherwise it makes its state from RUNNING to RUNNABLE and pushes it to the expires queue. Next it tries to assign the first uthread in its active runqueue and makes its priority to RUNNING. If there is no uthread in the active runqueue, it swaps the active and expires queues. After all the uthreads are finished, the kthread debinds itself from the CPU and quits. The application quits after all kthreads have exited. Since each uthread is working on different rows, there can be no indeterminacy in the entries of the product matrix.

2. Summary of code

Following is the summary of the files provided:

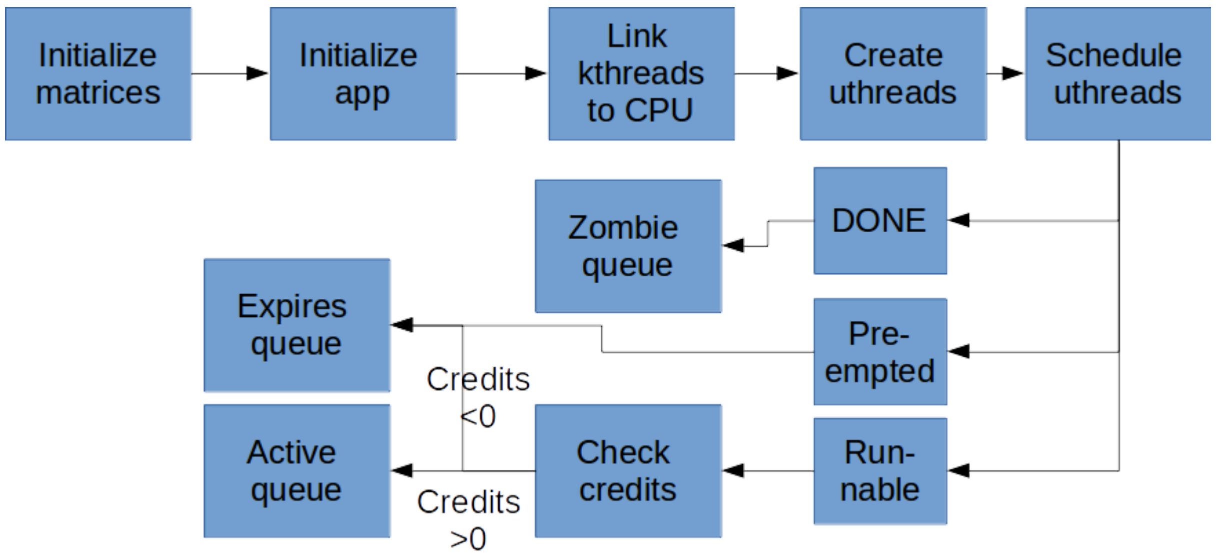
- gt_matrix.c : Contains the methods to generate, print and multiply matrices. The main function is called in this file. It initialized the matrices, initializes the kthreads and creates uthreads for matrix multiplication.
- gt_kthread.c : This file consists of functions to create and initialize kthreads, bind kthread to CPU, and exit kthreads.
- gt_uthread.c: Contains functions to initialize, create and schedule uthreads.
- gt_spinlock.c : Contains functions for spin lock and unlock
- gt_signal.c: Contains functions to initialize time slice and install signal handlers

3. Credit based scheduler

Credit based scheduler is a proportional fair share scheduler that schedules job as per the weights given to them. The weight indicates the CPU time a job gets in proportion to other jobs. For example, job A with weight double of job B will get twice as much time as the job B. Also, there is a cap which limits the maximum amount of CPU time a job can get. Even if the CPU is idle, the job will not be scheduled if it exceeds the cap time limit. To ensure thay each job gets a minimum time to run, there is a parameter context switch rate limiting which ensures that a job cannot be preempted before running for a specific time. Each job (VCPU) gets some initial credit which is equal to its weight. When the job is running, credits proportional to the running time of the job are used. There are two levels of priorities that are defined for the credit based scheduler: OVER and UNDER. UNDER priority means the credits haven't been used up yet and OVER priority means that the credits have become negative. Initially all the jobs have UNDER priority and have positive credits. If at all, the credit for a specific job becomes negative, its priority must be changed to OVER and the credits must be replenished again. The jobs with priority UNDER are always scheduled first and then only the jobs with priority OVER are scheduled. Credit scheduler uses a time slice of 30ms. A CPU has to schedule best VCPU whenever a VCPU completes its time slice or yields the CPU. Two queues are maintained for the priorities. The active queue consists of all the jobs with UNDER priority whereas the expires queue contains the jobs with OVER priority. The best VCPU is the head of the active runqueue. However, if there are no jobs with UNDER priority in the current runqueue, the CPU should look for other CPUs having such jobs. This is called load balancing. This ensures that each VM gets a fair share of CPU resources. Thus whenever there is any runnable job with positive credits, it will always be scheduled before those whose credits have been expired. Hence it is called a fair share scheduler because of its work conserving nature.

4. Objective of the project

We need to change the implementation of GT Threads library to accommodate credit based scheduling in addition to currently existing priority scheduling and co-scheduling. For this purpose, four weights (25, 50, 75, 100) and four matrix_sizes are provided (32, 64, 128, 256). Thus 16 matrix multiplications have to be performed. The number of kthreads must be equal to the number of CPUs in the system. 128 uthreads need to be created. Thus for each of the 16 combinations, 8 uthreads will be working on the matrices. The aim is to output the mean and standard deviations of the running time and the total time for each process. Thus all 8 uthreads will perform same multiplication and the times will be averaged out over all of them. The credits are the same as the weights. The cap for each thread is set to be 1. The time slice needs to be 30ms. If a thread is running for a specific period of time, its credits need to be subtracted by the proportion of the time slice used multiplied by a specific factor (10 in my implementation). As soon as the credits become negative, the uthread is pushed in the expires queue and credits are replenished equal to the weight. Moreover, load balancing needs to be implemented wherein if a CPU has no active jobs, it should look for other CPUs with active jobs. If no CPU has active jobs, then schedule the jobs in the expires queue.



5. Design and implementation

1. Declared global variables total_thread_time and running_thread_time for saving the total and execution times for all the 128 uthreads. Declared number of cpus as global variable. Declared a global flag kthreads_initialized to ensure performing load balancing on uthreads only after all kthreads have been initialized. These changes are reflected in gt_include.c
2. Maximum number of rows have been set to 256 and number of threads to 128. Each of the threads will have a weight, cap, time_slice, start row and end row. We need to perform the same matrix multiplication on 8 threads for all the 16 combinations of weights and matrix sizes so that we can estimate average running and total times.
3. A custom function has been defined for matrix multiplication with the credit scheduler. The uthread will compute products of square matrices from start row to end row for both matrices.
4. The main function defined in gt_matrix.c requires an argument: 0 for priority scheduler, 1 for credit based scheduler. The number of cpus are initialized here by making a system configuration function call. If the argument is 1 (corresponding to Credit based scheduler), the app is initialized, matrices are initialized. For all 16 combinations of matrix multiplications, 8 threads are created which perform the same task. Their group id is the CPU that they shall be assigned to. Thus, in a system with 8 CPUs, each thread is scheduled on a different CPU. The uthreads are initially allotted credits equal to their weights. Finally the app is exited and the running times and total times saved in the global variables are used to calculate mean and standard deviation of the execution and overall times for each process. All these changes have been made in gt_matrix.c
5. App initialization: It takes a single parameter SchedulerChoice which checks if it is a priority based or credit based scheduler. This scheduler type is saved in a shared variable ksched_info->sched_type. Spinlocks are used when changing this shared variable. When all the kthreads (equal to num_cpus) have been initialized, set the kthreads_initialized flag. These changes have been reflected in gt_kthread.c
6. The matrices A and B both with a size of 256 x 256 have been initialized by 1 and the matrix C of 256 x 256 has been initialized with 0. Next, 128 uthreads are created. Each uthread needs to compute various times, so timeval objects created_time, finished_time, curr_start_time, curr_end_time, curr_run_time, total_run_time are declared for each uthread. A flag is_prempted is also declared which is set whenever a thread is preempted. A function gt_yield is declared for this purpose. These changes are incorporated in gt_uthread.h
7. When uthread is created, store the created_time and set is_preempted to 0. Whenever uthread is scheduled, first get the current RUNNING uthread. If its is the credit scheduler, compute the run time of the RUNNING uthread and add it to the total run time. Subtract a proportional amount of credits from available credits. If thread is DONE/CANCELLED, compute the finish time and store the total time from spawning to finish in the global variable total_thread_time. If the thread has yet to perform the function, make its state RUNNABLE. Next, check if remaining credits are positive. If yes, check if thread was preempted. If it was preempted, push it to the

expires queue with a priority of OVER and replenish the credits equal to the weight of the uthread. If it hasn't been preempted, then push it back to the active queue. However, if the credits are used up, then push to the expires queue with a priority of OVER. Find next best uthread and make its state as RUNNING and schedule it.

8. For implementing preemption of threads, all uthreads with thread ID multiple of 10 are preempted twice during their execution. The function gt_yield is called for this purpose. This function just halts the execution, sets is_preempted flag to be 1 and finally calls uthread_schedule again so that this uthread goes to the expires queue. All the changes have been made in gt_uthread.c
9. Finally in order to maintain the work conserving nature of the scheduler, load balancing needs to be performed. For this purpose, variables cur_k_ctx, cpu_krunq and cpu_runq have been defined. If there are any active uthreads, they must be scheduled first. But whenever the scheduler looks for the best uthread and there is no uthread in the active queue with UNDER priority, it looks for other CPUs. cpu_krunq and cpu_runq and the runqueues corresponding to the other CPU. If this CPU also does not have any active uthread, then continue the loop. Otherwise, return the best uthread in cpu_runq for scheduling. If no uthreads have been returned, it means that all CPUs have uthreads in OVER priority. So then look at the expires queue of current CPU and schedule the best uthread. This is the correct implementation of load balancing.
10. This completes the design and implementation of the credit based scheduler. The next section highlights the results of the computed averages and standard deviations of the times for all 16 combinations of weights and matrix_sizes.

6. Implementation issues

The code works perfectly for O(1) priority scheduler (./bin/matrix 0) without any errors/faults. For the credit based scheduler, it works great for most of the executions, but in few cases, deadlocks and segmentation faults are encountered. The deadlock maybe due to unavailability of CPU resources and the code can be optimized to avoid such adverse conditions. The segmentation faults are mainly due to memory leaks and can be debugged using gdb and in few cases, they are due to an assertion error in the load balancing code which checks for other CPUs when there are no uthreads in the active runqueue of the current CPU. The code currently works for cap equal to 1 which can be extended to other values too. Similarly, context-switch rate limiting also hasn't been implemented to ensure that a uthread gets a minimum time to run before being preempted. These can be good additions to the code.

7. Summary of Results

Process ID	Weight	matrix _size:	Mean_run_ Time (us)	Mean_total_ Time (us)	Std_run_time (us)	Std_total_time (us)
0	25	32	1313.5	240390.125	2393.236328	621134.6875
1	25	64	8085.25	166234.375	5043.556152	245957.90625
2	25	128	65412	749837.25	12248.708008	783502.8125
3	25	256	478799.875	2490601	83286.40625	499066.875
4	50	32	316.25	6308.125	23.647146	1230.087769
5	50	64	6476.875	332045.875	3129.709961	744594.625
6	50	128	54542.625	761821.25	20129.621094	960963.4375
7	50	256	460687.75	1578005	64300.605469	321747.90625
8	75	32	446.75	236622.125	178.178802	619884.25
9	75	64	6854.125	69984.125	4171.416504	28817.025391
10	75	128	70478.5	817238.75	15018.060547	698672.375
11	75	256	508017.125	2316787	83863.742188	522990.40625
12	100	32	682.5	340595.125	960.272461	885600
13	100	64	6565.75	329491.625	2912.57959	750095.5
14	100	128	54583	418873.25	9472.337891	231495.03125
15	100	256	493189.125	1875737.125	63031.390625	667084.75

Analysis: As expected, smaller sized matrices are multiplied very fast whereas the larger matrices take considerably more time. On an average, 32 x 32 matrix multiplication is performed in 1ms whereas 256 x 256 matrix multiplication requires almost 5s of execution time. Another thing to notice is that the total times are not as less as the running times for small matrices. This is because they may have not been scheduled early. This is the essence of the credit based scheduler that everyone gets a fair share of CPU time. Finally, the standard deviations for total times are quite high again because the order in which uthreads were scheduled makes a lot of difference.