CS-6210: Advanced Operating Systems
Project 1: GTThreads Library – CFS Scheduler
Final Report

Vinay Bharadwaj (902825919)

1. Design & Implementation

To integrate the CFS scheduler and the related RB tree data structure into the existing library, I modified two main structures - the existing kthread_runqueue_t structure and uthread_struct_t structure.

a. The active and expired runqueues and the priority arrays were gotten rid of and a CFS RQ RB tree was put in place.

b. The uthread_t thread structure was modified to include CFS related variables such as vruntime, weight, on_rq. A flag yielded keeps track of whether the thread yielded.

CFS_RQ Structure and CFS variables:

The CFS_RQ structure is a RB tree. Each CPU has its own CFS_RQ structure instead of the active and expired runqueues. For the RB tree library, I used an open source RB tree library from literateprograms (cited in source code).

I then modified the rbtree structure to include nr_running and load_weight variable. nr_running keeps track of number of active threads in the CFS runqueue and load_weight keeps track of the load of the runqueue. The rbtree structure also has the root node of the tree from where all the other nodes in the tree can be reached. Each rb_node points to a node in the rbtree and the rb_node structure has the following fields – key, value, left, right, parent and color.

The tree is sorted according to the **key** and **value** points to the value of the node, which in our case is a pointer to a uthread_t structure and thus binds the uthread to that particular node.

The rbtree structure was then integrated into the kthread_runqueue_t structure, replacing the active and expired runqueues.

The following variables were used as is from the original CFS scheduler implementation:

1. sched_latency – Set to 20ms default. This is the scheduling epoch.
2. sched_min_granularity – Set to 4ms default. This is the minimum thread preemption latency.
3. sched_nr_running – Set to 5 by default.

The scheduling epoch is set based on the number of running threads. The relation is :
sched_latency = sched_min_granularity x nr_running.

CFS tree & timer related functions:

Apart from the rbtree_create(), rbtree_insert(), rbtree_delete(), rbtree_lookup() functions provided by the library, I had to write the following functions to implement the scheduler:

1. find_leftmost() - This function takes the rbtree as an argument and returns the leftmost node in the rbtree.
2. updata_curr() - This function is used to update the vruntime of the currently running thread.
3. update_min_vruntime() - Updates the minimum vruntime of the CFS runqueue by setting it to the vruntime of the leftmost element in the tree.
4. sched_period() - Takes the number of running threads as an argument and calculates the scheduling epoch based on it.
5. set_sched_timer() - Gets the scheduling epoch by calling sched_period() based on the nr_running variable. It then sets the sched_latency to this value.
6. sched_timer() - Returns the value of sched_latency. This is used when setting the timer for SIGVTALRM. It makes sure that the new timer value is always the up-to-date value of sched_latency.
7. preempt_timer() - Returns the value of min_granularity. This is used to set the timer for SIGALRM which has the preempt_uthread() function associated with it. The running thread is preempted every min_granularity time and checked whether it is still the leftmost. Else it is rescheduled.

CFS and thread scheduling related functions:

The following functions were newly introduced into the uthreads package:

1. gt_yield() - This function yields the processor when called by a running thread. It can be used to efficiently use idle CPU when the thread is waiting for some user input or I/O bound task.
2. preempt_uthread() - Checks at sched_min_granularity intervals whether the current thread is still the leftmost. If not, it is rescheduled.

The following functions were modified to implement CFS in the package:

1. uthread_create() - Here, instead of adding to the runqueue, we add the thread to the CFS tree structure. We initialize the vruntime, weight, etc., and add to CFS runqueue.
2. uthread_schedule() - Schedules the leftmost thread in the CFS tree for execution. It also updates the vruntimes of the current executing thread and puts it back in the tree.
3. sched_find_best_uthread() - Returns the leftmost thread in the tree which is thren used by the scheduler to schedule. Also updates the min_vruntime of the CFS runqueue.

CFS runqueue related functions:

1. add_to_cfs_runqueue() - Inserts the thread into the RB tree and updates the nr_running variable.
2. rem_from_cfs_runqueue() - Deletes thread from the RB tree and decrements the nr_running variable.

2. **Matrix Multiplication:**

The existing code for matrix multiplication split the matrices between all threads and each thread would multiply a part of the matrix. To make each thread multiply its own copy of matrices, I have modified the program as follows:

1. Generate matrices within the mulmat function so that each thread gets its own copy of matrices.

2. Split the threads into 4 groups of 32 threads each, namely – group0 (multiplying matrices of size 32), group1 (multiplying matrices of size 64), group 2 (multiplying matrices of size 128) and group 3 (multiplying matrices of size 256).
3. Collected statistics on vruntime and calculated mean vruntime, mean total execution time, standard deviation of vruntime and standard deviation of total execution time for each thread group.

Results:

The mean vruntimes and total execution times and Standard deviation of vruntime and total execution time for each group are as given below:

GROUP	Mean vruntime (us)	Mean total execution time(s)	SD of vruntime	SD of total execution time
Group 0 (Size 32)	3089.937500	1.375000	0.018167	2.002928
Group 1 (Size 64)	5769.000000	1.5	0.021076	2.165064
Group 2 (Size 128)	30210.093750	1.562500	0.059622	2.181913
Group 3 (Size 256)	1073722442.093750	1.625000	0.16455	1.996090

Observations:

1. The mean vruntimes for each group goes on increasing. This is because the groups that multiply matrices of bigger sizes take more time to complete. As a consequence, Group 0 threads have lesser mean vruntime because they take lesser processor time than Group 1 threads to complete execution and so on.

2. The mean total execution times for each group of thread goes on increasing for groups multiplying larger sized matrix. This is expected behavior because threads in higher group numbers take more time to finish execution. Further, this total execution time may also include waiting times of threads that are waiting to be scheduled.
3. The most interesting observation is the standard deviation of vruntime. As we can see, irrespective of the group and the size of matrices the standard deviation of vruntimes across all the groups remains very small. This indicates that each group of threads is getting fair amount of CPU time corresponding to its workload.
4. In the standard deviation of total execution times though, we can see a bigger standard deviation than vruntimes. This is because the total execution time also depends on various factors such as wait time, when the thread is waiting to be scheduled. As a result, there is a larger difference between the total execution times of threads and hence the standard deviation for total execution time is greater than the standard deviation of vruntimes.

Comments:

1. The statistics have been commented out in the matrix program to show the clear termination of the program without the clutter of statistics. These can be uncommented out if needed.
2. My future work on this project is to improve the vruntimes and implement finer granularity. Also, the code needs to be cleaned and unnecessary variables / functions need to be removed.
3. Overall, an amazing experience implementing a new scheduler. I have implemented a posix compliant gtthreads library with round robin scheduler previously when auditing the same course.