Project 1: Process-wide-CFS using GTThreads – Final Report

Name: Raghavendra V. Belapure

Part I: Project Implementation

- Implementation option chosen : using multiple k-threads (one k-thread bound to each CPU)

The project is implemented as follows:

1. **Initialization:**
   Initially, the thread arguments are initialized and two source matrices are created for every thread.

2. **Kthread create:**
   Then kthreads are created. Number of kthreads equals number of cores in the system. Every kthread is then bound to each CPU. These kthreads now wait for uthreads to get scheduled.

3. **Scheduler set up:**
   Each kthread has its own scheduler. This scheduler routine is now installed as a signal handler for SIGSCHED.

4. **Thread groups:**
   The application creates two different thread groups. Group 0 contains optimal number of threads, that is, number of threads in group 0 equals number of cores. Group 1 contains more number of threads than group 0, and hence, it is called as greedy group. These thread groups simulate different processes.

5. **Uthread creation and timeslices**
   Now, uthreads are created and are included into thread groups. The uthread is mapped with kthread. As soon as the uthread is created, it sends SIGSCHED to the kthread to which it is assigned. The kthread chooses the newly created task for scheduling and calculates a timeslice. The timeslice depends upon the latency and load of kthread along with the priority of uthread. It also depends upon the number of threads within the thread group.
   The timesclice calculation is done as follows. The process scheduler should assign equal time to all the processes in the system. This time is then divided to all the threads within that process. The time-slice for thread-wide-cfs is given by

$$Timeslice_t = \frac{kthread\ latency\ \times\ uthread\ priority}{kthread\ load}$$

   But, the time-slice calculated for each thread by the process-wide-cfs should be

$$Timeslice_p = \frac{Timeslice_t}{process\ count \times \#\ of\ threads\ within\ process} = \frac{kthread\ latency\ \times\ uthread\ priority}{kthread\ load \times process\ count \times \#\ of\ threads\ within\ process}$$

6. **vruntime:**

   The scheduler now sets an interval timer for the uthread and starts running the uthread. When the timer expires, SIGVTALRM (which is same as SIGSCHED according to the macro definition) is sent to the kthread. This invokes the scheduler. Scheduler now preempts the current running uthread and calculates the vruntime for the same. According to timeslice calculation, the individual threads in the greedy group will run for less amount of time than the threads in optimal group. As a result, the red-black tree will select the greedy threads for execution. As a result, we need to apply a weight to the vruntime. We calculate vruntime for the threads as –

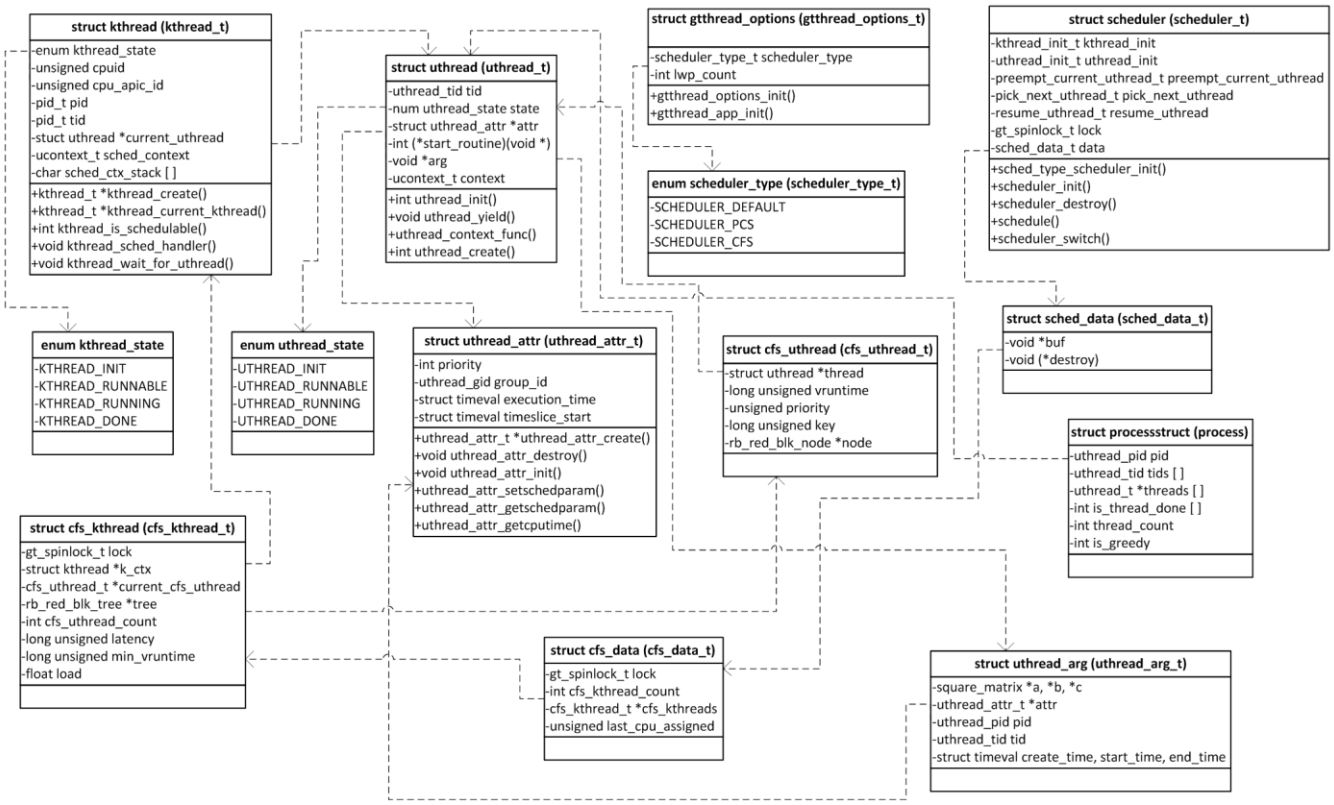
   $$vruntime = cpu\ time\ \times thread\ priority\ \times \#\ of\ threads\ within\ the\ process$$

   This will cause the threads in greedy group to have more weight in the red-black tree, so that the threads in the optimal thread group will receive fair amount of CPU time.

7. **Ending condition:**

   The process structure keeps track of which of the member threads have completed. When an uthread gets completed, the scheduler checks whether all the threads in any of the processes is complete. If all threads in a process are completed, the scheduler marks a “stop flag”. Before resuming next process, scheduler checks the value of this flag. If the flag is set, the scheduler will initiate actions to exit the kthread and will eventually call kthread_exit() instead of scheduling next uthread. In this way, the other threads will be terminated.

   Thus, the updated data structure will be as follows –

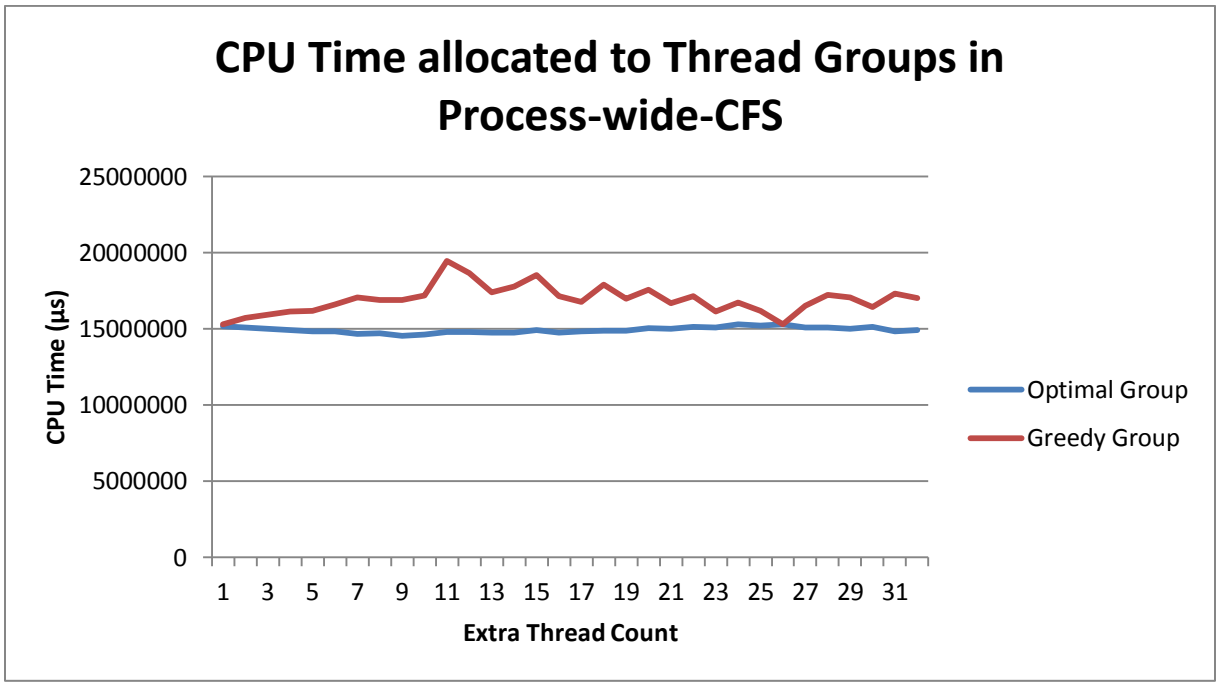


Part II: Results

The results are obtained by running the program on Killerbee cluster. Matrix size is set to 512. Extra thread count is varied from 1 to 32 so that, the number of threads in greedy group vary from 25 to 48. The test was carried out for both Process-wide CFS and Thread-wide CFS. CPU time allocated to individual threads and for processes was observed. Graphs of CPU Time allocated to the thread group vs. number of extra threads in greedy group are plotted for both Process wide and Thread wide CFS schedulers.

The results conclude that, the process wide scheduler succeeds in allocating equal CPU power for both the threads. The scheduler reduces the time-slice allocated for the individual threads in greedy thread group in order to facilitate fair treatment to the threads in optimal thread group.

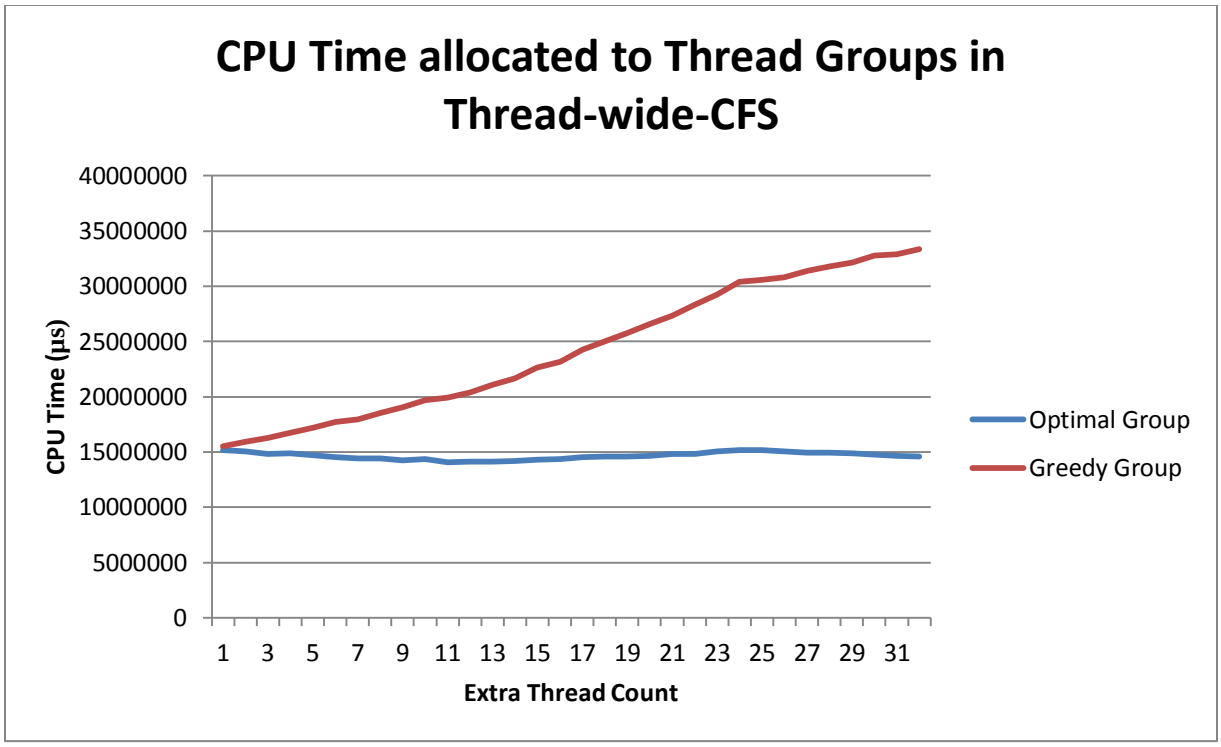
The graphs are shown below.



The graph shows that both optimal and greedy thread groups get equal amount of CPU power allocated to them. As a result, greedy process does not dominate the CPU by having very large amount of threads.

The graph for Thread wide CFS shows that the scheduler gives fair treatment to all the threads instead of processes. As a result, the amount of CPU time allocated to the process increases linearly with the number of threads in the process. As a result, the process with more number of threads will dominate the CPU resulting in starvation of the process with lower number of threads.

The graph for Thread-wide CFS is shown below –



The data used for the graphs is given below. More detailed results, which show CPU time and elapsed time for each individual thread are present (results_process_wide_cfs.txt , results_thread_wide_cfs.txt).

Process Wide CFS

Extra Thread Count	Optimal Group	Greedy Group
1	15166890	15323591
2	15084342	15736680
3	14994854	15931757
4	14934798	16129804
5	14842268	16211636
6	14859149	16626654
7	14675571	17086157
8	14727949	16918564
9	14549504	16913645
10	14637442	17220856
11	14797626	19449906
12	14788697	18657171
13	14772708	17395482
14	14774187	17772358
15	14924973	18560337
16	14779683	17140887
17	14847387	16763161
18	14878685	17934556
19	14880492	17006387
20	15054412	17576064
21	15010772	16698407
22	15143521	17139061
23	15109706	16147247
24	15324909	16747806
25	15205387	16208474
26	15324409	15324409
27	15098650	16514466
28	15094797	17245973
29	15014724	17063563
30	15122620	16463194
31	14868312	17336027
32	14947010	17033761

Thread Wide CFS

Extra Thread Count	Optimal Group	Greedy Group
1	15155958	15518115
2	15069961	15941949
3	14851990	16246837
4	14910981	16738125
5	14684800	17197455
6	14565262	17721402
7	14423905	17943240
8	14435575	18553547
9	14265605	19029098
10	14352939	19682645
11	14091612	19918617
12	14158906	20395457
13	14154777	21075928
14	14204579	21659044
15	14322281	22644756
16	14344457	23162735
17	14518993	24281348
18	14589466	25027431
19	14597641	25771511
20	14664789	26608703
21	14799570	27348695
22	14826158	28299886
23	15042969	29226774
24	15179791	30383527
25	15172937	30577049
26	15058444	30795498
27	14950469	31420775
28	14930674	31805528
29	14861193	32160701
30	14778204	32799426
31	14655376	32871598
32	14590501	33372214

Part III: Implementation Issues

The process-wide CFS scheduler must end when one of the processes is complete. Currently, when one process has completed its work, it sets a global flag called stop_all_threads. Each kthread checks this flag before allocating CPU to the next selected uthread. As a result, the kthread will exit only after the current execution schedule for the present uthread is complete. The currently executing uthreads will run for little extra time till they complete the current timeslice. The optimal thread group is going to complete first in case of our program. As the greedy group is allowed to complete the current execution schedule, we can see in the graph that overall allocated CPU time is more for greedy thread group than optimal thread group.

This could be avoided by using signals. When a process group completes, it should send a signal such as SIGUSR2 to the parent process. The parent process will send a signal to all its children. The children (i.e. the kthreads) will catch that signal by installing a handler and the actions to stop the kthread will be written in handler. The parent again waits for all the kthreads to exit. As kthreads are child processes, SIGCHLD will be sent to parent once they terminate. Now, the parent should proceed to calculate the results. This would eliminate the excess allocation of CPU to greedy threads.