CS 6210: Advanced Operating Systems **January 14, 2012**

Project 1: Thread Scheduling with `GTThreads`

Instructor: Professor Karsten Schwan(schwan@cc) TA: Minsung Jang (minsung@gatech)

0. Outline

The goal of this project is to understand the fair scheduling algorithm and implement the *completely fair scheduler* (CFS), which is the default task scheduler in up-to-date Linux, as an example of fair schedulers.

You may discuss ideas with colleagues in the class, but the project must be done individually. Copying others is NEVER allowed for any reason. Please refer to the Georgia Tech honor code available at http://www.honor.gatech.edu/.

Turn in your final deliverables for the project on **February 4, 2012 by 11:59pm on T-square** website only. Emails or other methods for submission will not be accepted. In addition, you must deliver an interim report for the project on **January 23, 2012 by 11:59pm** (as explained in more detail below.)

Since the project requires you to implement codes using the C language, perform performance tests, and document your work, **you should start it as soon as possible**.

1. Goal

The goal of this project is to understand the fair scheduling algorithm and implement the completely fair scheduler (CFS), which is the default task scheduler in up-to-date Linux, as an example of fair schedulers. To do so you must modify the given `GTThreads` library. The library implements an `O(1)` priority scheduler and a priority co-scheduler for reference.

2. Details

The project minimally requires the following to receive full credit:

- Implement the completely fair scheduler (CFS) for the SMP `GTThreads` library provided.
- Implement a function for matrix multiplication, using the provided code
- Implement a library function for voluntary preemption (`gt_yield()`). When a user-level thread executes this function, it should yield the CPU to the scheduler, which then schedules the next thread (per its scheduling scheme).
- Write a `Makefile` with the basic rules for compilation and clean up. Also, include a `README` on how to run your project and other parameters if required.
- Write **an interim and a final report** (in PDF format) summarizing your implementation and detailing the results (as explained below).

Each item is explained in the sections below. You may also be asked to deliver a demo and presentation to the TA showing the results.

Directions: Please upload your project on the T-square website. No emails for submission will be accepted. Use the `GTThreads` package provided to you as a starting point. The first thing you have to do is to take a look at the provided package very carefully. If you have any questions, you are encouraged to bring them to

the TA during his office hours (to be posted soon) and you can also use the forums and wiki on the T-square website for posting queries, and to exchange information with other students.

3. Instructions for the given GTTThreads Library

`GTThreads` is a user-level thread library. Some of the features of the library are as follows:

- Multi-Processor support: The user-level threads (`uthreads`) are run on all the processors available on the system.
- Local runqueues: Each CPU has its own `runqueue`. The user-level threads (`uthreads`) are assigned to one of these runqueues at the time of creation. Part of the work in the project might involve using some metrics before assigning these `uthreads to` the processor and/or to perform run-time `runqueue` balancing
- `O(1)` priority scheduler and co-scheduler: The library implements these two scheduling algorithms. The code can be used for reference.

The code runs on the `Killerbee` cluster machines. It should also run on other CoC machines. `Killerbee` is operated under Redhat Enterprise Linux 5.x. However, it is recommended that you do your initial development and testing on your local machines (the class assumes that most of you are able to access at least dual core machines). Once your work is sufficiently stable, you can test it on the one of `Killerbee` machines. We might also ask you to demo it on the `killerbee` machines. Please make sure you don't leave any zombie/running processes behind on `Killerbee` when you log off.

4. The Completely Fair Scheduler

Introduction: The `O(1)` scheduler has been replaced with its successor, the completely fair scheduler (CFS) since kernel 2.6.23 . The design of CFS tries to achieve the following goals [1]:

- Provide good interactive performance while maximizing overall CPU utilization
- Ensure fair distribution of CPU time to each entity
- Implement the modular scheduler framework by introducing Scheduling Classes

Note: The functionality and performance of this scheduler heavily depend on a specific data structure, **Red-Black Tree** (RBTree). Moreover, the scheduler on current Linux provides a lot of additional features to support various kinds of demands from kernel developers. However, the project does not aim to implement the tree from the ground up and mimic all features of the scheduler existing in Linux. Therefore, to make the project feasible, some simplifications are given, as follows:

- Ignore the scheduling classes/modular scheduler. Support "`SCHED_NORMAL (SCHED_OTHER)`" only
- Ignore the "group scheduling" feature as well as "power saving." Focus on how to fairly allocate CPU resources to each task, which is related to "`vruntime`."
- Use the RBTree libraries available on various open source projects (or even in the Linux Kernel.) You are still fine if you want to implement your own library for RBTree, but note that this may be quite time-consuming.

Resources:

[1] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, and Fun Wey. 2008. Towards achieving fairness in the Linux scheduler. *SIGOPS Oper. Syst. Rev. 42, 5 (July 2008)*, 34-43

[2] Jacek Kobus and Rafał Szklarski, Completely Fair Scheduler and its tuning, at http://www.fizyka.umk.pl/~jkob/prace-mag/cfs-tuning.pdf

[3] sched-design-CFS.txt located at $KERNEL_SRC[1]/Documentation/scheduler
[4] sched_fair.c located at $KERNEL_SRC/kernel
[5] Inside the Linux 2.6 Completely Fair Scheduler, at http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/
[6] Multiprocessing with the Completely Fair Scheduler, at http://www.ibm.com/developerworks/linux/library/l-cfs/index.html
[7] Completely Fair Scheduler, at http://www.linuxjournal.com/magazine/completely-fair-scheduler?page=0,0

5. Results and Report

Threading the Matrix Multiplication

A very rudimentary code for multiplying matrices is provided in the matrix directory. The code generates matrices (with each entry as 1) and then creates a number of threads to multiply the matrices. The output of the multiplication corresponds to calculating `C = A×B`. Each thread calculates a fraction of the rows in `C` by calculating the same stripe of rows in `A` times the entire matrix `B`. Use this code to write a function that multiplies its own matrix. This is to mean that each `uthread` is working on its own individual matrix.

Test Cases

To get full credit for results, you must execute the following test cases:

- Run 128 `uthreads`
- Each `uthread` will work on a matrix of its own
- Matrix sizes are ranging in {32, 64, 128, 256}

Since there will be 128 `uthreads`, there will be 32 `uthreads` for each matrix size. (It isn't always interesting to parallelize matrix multiplication. Here multiple sets of threads need to be running over your scheduler with different workloads.)

Collect the time taken (to the accuracy of micro-seconds) by each `uthread`, from the time it was created, to the time it completed its task. Also measure the CPU time that each `uthread` spent running (i.e., excluding the time that it spent waiting to be scheduled) every time it was scheduled.

The final output should be as follows: For each set of 32 `uthreads` (based on a given matrix size), print the mean and the standard deviation of both, the individual thread run times, and the total execution times. It will be useful if some output is printed while the process is running. (e.g., you may want to print messages when an `uthread` is put back in the queue, when it is picked from the queue, etc.) You can print the output to a `per-kthread` file instead, if you want.

Reporting and summarizing results

For an interim report, you must include the following:

- Show your understanding of the given `GTThreads` package (max. 2 pages) - try to jot down how an `O(1)` scheduler in the package is implemented with simple diagrams (e.g. function interactions or flow chart).
- Present how CFS works in Linux briefly (max. 2 pages) - try to explain basic rules (or the algorithm) for scheduling in CFS.
- Sketch your design (max. 2 pages) - try to show your implementation plan of CFS involved in the given `GTThreads` package, that is, how to modify the given package for introducing CFS into it

[1] $KERNEL_SRC > 2.6.23

To get full credit for the write up, you must include the following for the final submission:

- Write a report summarizing the implementation of the project.
- Summarize results for all the test cases above.
- Mention clearly the implementation issues in your final submission. For example, you might want to point out some inefficiency in your code, etc. This doesn't mean you don't need to fix blatant errors, if there are any minor issues, like performance, don't spend too much time trying to fix it, but make sure you list ways in which you might improve it.

6. Submission

Include all reports, all source codes including Makefile, etc., in the top (and same) directory, compress them into a ZIP file, and then upload that on T-Square.