CS 6210 Project 1 Report

Sam Britt

February 7, 2012

Implementation

The overall structure of the design was to factor out all the scheduler-specific code into modules that adhere to a generic "scheduler" interface, and hooks put in place throughout the `kthread` and `uthread` modules to initiate scheduler-specific functionality. In this way, the user can provide the scheduler she wishes to use as an option to `gtthread_app_init()`, and the scheduling policies can be loaded at runtime. Both the priority scheduler provided with the package and the new completely fair scheduler were designed to meet this interface. The interface includes the functions:

- `scheduler_init()` This function is called at the start of the application, and allows the scheduler to initialize any data structures it needs. The scheduler will have access to its data though a global variable.
- `kthread_init()` Called after the initialization of every `kthread`, to allow the scheduler to do any bookkeeping it may need to do for each `kthread`
- `uthread_init()` Called after the creation of every `uthread`. The scheduler must return an appropriate `kthread` for it to be scheduled on.
- `preempt_current_uthread()` Called after the timer interrupt to preempt the currently running `uthread`. It is here that the scheduler can insert the uthread into the appropriate ready queue, if needed.
- `pick_next_uthread()` Called to choose the next runnable `uthread`.
- `resume_uthread()` Called just before returning control back to the user's application. Its main function is to set a timer for the appropriate timeslice amount. The library will be woken at this point to schedule the next thread.

All these functions are set up through function pointers maintained in a global `scheduler` data structure. The pointers are defined at application startup, so there should be little or no performance lost during the scheduling of `uthreads`.

For the completely fair scheduler, the data structures are as follows: The scheduler maintains an array of `cfs_kthread` structures, one for each virtual processor being used. Each `cfs_kthread` maintains a pointer to the `kthread` structure used by the rest of the system, as well as CFS-specific data. For example, each `cfs_kthread` has a pointer to the red-black tree holding all the schedulable entities for that processor, a pointer to the currently running `uthread`, the current latency or "epoch length" for the system, the current load on the system, and the current value of the minimum virtual runtime (`vruntime`) of all the `uthread`s in its runqueue. Similarly, the `cfs_uthread` structure maintains a pointer to the corresponding `uthread` used by the rest of the system, in addition to the current values of it's virtual runtime, its priority, and a pointer to its node in the red-black tree.

At each point of the above interface, these data structures are updated appropriately. For example, during `uthread_init()`, a node is created with the current minimum `vruntime` and inserted into the appropriate `cfs_kthread`'s red-black tree. In addition, the `cfs_kthread`'s load is increased, and this potentially increases the latency past a threshold. Similarly, when a `uthread` has completed execution, it is removed from the red-black tree and the `kthread` load is reduced.

When a `uthread` is preempted, the time it spent in execution is first calculated. Its `vruntime` is then updated as

$$\texttt{vrutime} = \texttt{cputime} \times \texttt{priority} \tag{1}$$

In this manner, higher priority (lower value) tasks have their `vruntime` increase more slowly, keeping them more "to the left" in the red-black tree. Lower priority tasks have their `vruntime` increase quickly and will be further "to the right," and thus will wait longer to be scheduled. After the new `vruntime` is calculated, it is inserted back into the red-black tree keyed on `vruntime` minus the minimum `vruntime`. The subtraction handles possible overflow in the `vruntime` value.

Picking the next `uthread` to schedule is a simple operation: simply pick the node in the tree with the minimum `vruntime` value. The `min_vruntime` value for the `cfs_kthread` is also updated with this value.

Before the `uthread` resumes execution, a timer is set. The value of this timeslice is determined at every context switch, and it depends on the current load and `uthread` priority as

$$\texttt{timeslice} = \texttt{latency} \times \frac{\texttt{priority}}{\texttt{load}} \tag{2}$$

where `load` is simply the sum of the priorities of all the threads waiting to be scheduled. In this way, high priority, interactive tasks get smaller timeslices (and their `vruntime` stays small).

The compiled in default is a latency of 20 ms. Virtual processors are chosen in a round-robin fashion; there is no co-scheduling or grouping of `uthreads`. Times are to microsecond resolution. The red-black tree implementation used can be found at `http://www.mit.edu/~emin/source_code/red_black_tree/index.html`. It was modified to support caching of the left-most node for quick retrieval.

Results

The scheduler was tested by having 128 `uthread`s multiply square matrices of different sizes in parallel. The results are shown in Table .

Due to issues in the code (see below), I could not get the application to run reliably on small matrices; that is, matrices with less than about 64 elements per edge. Therefore, I started with matrices of size 128 and went to 512. All threads were running in parallel on their own matrix; there were 32 threads per matrix size.

"CPU Time" is the time the `uthread` spent actually on the processor; that is, excluding the time waiting in the ready queue. "Elapsed time" is the amount of time between `uthread` creation and the completion of its task; it includes all the overhead of scheduling and waiting for other tasks.

The results are mostly as expected: the threads with larger matrices showed higher execution times and variabilities. The increase in CPU time is initially linear; for example; for example, doubling the matrix size from 128 to 256 caused an 8-fold increase in CPU time (which is linear because the larger matrix has 4 times as many elements, and twice as many calculations to perform per element in the "inner loop."). The elapsed time, however, grows much more rapidly.

Table 1: CPU time and total execution time when using 128 threads to multiply matrices of various sizes. The total application time was 11.117 s.

Matrix Size	CPU Time (μs)		Elapsed Time (μs)	
	mean	(std dev)	mean	(std dev)
128	4030	(1752)	4028	(1752)
200	14 459	(3415)	464 505	(409 631)
256	32 736	(8049)	936 797	(214 329)
512	579 843	(59 293)	9 215 725	(966 977)

Implementation Issues

Besides taking far more time than I had planned, the major issue is some race condition that occurs most readily when the `uthread`s are given short tasks to perform. Often, if the tasks can complete within their initial timeslice, a segmentation fault or a deadlock will occur. I have not been able to track this bug down. If the `uthread`s are given longer tasks, so that they use their timeslices fully, the issue seems to be rarer.