Preemptive Thread Library

CS 415

Programming Project 3 - Code Phase

Due: 25 March 2008

Keeping in mind the design details from from the previous assignment description, this will go over the details needed to implement that design.

Getting Started

Do not worry about the project-3-for-design code. There are some small changes in many of the header files, so simply discard those files.

First back up the files from your last project. Then copy in the new files, taking care to merge your old minithread implementation into the new minithread.c file. Other files can overwrite the older versions of the same file, as there should be no other duplicate files you have implemented.

It is suggested to first get your priority queue and multilevel queues implemented, to then write the test cases for them, and ensure that they work correctly before moving on.

Then add support for clock interrupts and preemption to your minithreads and attempt to run app_buffer and app_elevator. Both of these should run fine, though if you have synchronization problems in your elevator, you will need to fix those.

Once you are sure that your threads are working with preemption, implement the alarm module. After you get it working, you will be able to run the app_sieve program. It is fairly demanding, so it is likely to uncover synchronization problems that were not exposed previously.

Only after you get alarms correctly working should you move on to the multilevel scheduler. Depending on how modularized your minithread design is, it should be a fairly small change to your code, but as with adding the preemption, there are some tiny details that are very important.

Once you have correctly implemented multilevel feedback scheduling, you should no longer see the greedy thread in the middle of the execution of app_sieve. However, this could potentially starve the greedy thread and others like it, so last you should add aging to your scheduler. Each thread is assigned an age based on when it is admitted to the ready queue. Threads which are not in the highest priority level of the queue are moved up to the next higher level (just a normal append on that level) after their age grows past a certain limit. This ensures that if some threads are making progress, then all threads are making progress.

At the top of minithreads.c are many defines which are used to control the tuning aspects of the scheduler and by extension of alarms. These are set so that app_sieve should work well on the computers in the CSUG lab, but once you are confident that your implementation is working correctly, play around with these settings (also including the period, which is defined in interrupts.h) and see the impact on performance. We can tune the difference between our long and short threads, in addition to tuning the overall responsive of our threads. At what point does trying to make the system more responsive actually backfire because of

increasing overhead? Is this value different depending on how often semaphores and locks are used? Is this value different when threads frequently block?

Finally, see if you can find a way to measure the various overheads in our system. It is allowable to include interrupts.h in an app for this purpose, so that you will have access to the *ticks* variable, but is there any way to do it without this variable? (Remember not to modify the included apps. Start in a fresh file, though you can copy over included code. However, please do not use code from the internet or from past versions of this class.)

Interesting things to measure would be the average time for a forced context switch, the average time for a voluntary context switch, the average time for a semaphore acquire under varying levels of contention, and the average latency introduced by the alarm system, particularly that caused by the scheduler.

Rules of Thumb

Figuring out when to enable and disable interrupts will be tricky. Not enough interrupt disabling, and race conditions ensue. Too much, and your system responsiveness drops dramatically. (The provided applications should all execute fairly quickly).

With alarm firing and ready queue aging you will be tempted to update values in every member of a queue on every interrupt. Don't! Figure out a way to work it so that a single external value changes and all values of items can be static. Remember to exploit ordering in FIFO and priority queues - it will let you avoid unnecessary work.

In general do very little in interrupt handler. There is no need to check something here if you know you won't be able to act on it until later (again, alarms and aging will be tempting to always keep updated, but often times there will be no way to act on updated information until several periods later.

Items of Note

Among the changes to the APIs are a more defined private interface to alarms which means that alarms and minithreads need not know about the internal structure of the other.

Also, multilevel_queues and priority_queues now have a peak operation which works just like dequeue with the exception that the queue is not modified.

There is a global variable provided by interrupts.h called *ticks*. It tells the number of periods that have elapsed since you turned on clock interrupts, and as such it will let you deal more intelligently with dropped interrupts when implementing preemptive scheduling and alarms.