CS 452 Kernel 3

Louis A. Burke (laburke) and Taras Kolomatski (tkolomat)
June 25, 2017

Overview

The beginning of the industrial revolution was marked by the development of steam power, enabling manufacturing and locomotion. Whereas an old Dutch sailor was compelled into an aesthetic contemplation he neither understood nor desired, face to face for the last time in history with something commensurate to his capacity for wonder, the inertial march of progress proved psychologically overwhelming. Post the revelations of Darwin, Freud, and Einstein, how could one maintain a static view of the context of humanity, the control one could exert on one's environment, or the continuity of mind and time. Literature reflected this, deforming mind and time in postmodern work, from the dreamscape of *Finnegans Wake* to the dissociation of Slothrop in *Gravity's Rainbow*. Indeed, one could not walk to a room in which the women come and go, speaking of Michelangelo, without the worries of eating a peach and disturbing the universe interrupting one's narrative.

In our third kernel milestone we forsake static deterministic execution, opening ourselves the overwhelming interruptions of the exterior world. Additionally our kernel now allows us to time things, delaying critical execution paths until the time is right for them to reactivate. To deal with the large amount of blocking that is now present, an idle task has also been created.

Structure

Notifiers

One of the major changes to the code is the delegation of some tasks as notifiers. These tasks spend most of their time event blocked and are directly linked to the hardware interrupt handler which reactivates them. This introduces a level of separation between hardware interrupts and the tasks that depend on them. Notifiers are by definition tasks which await events by calling the `AwaitEvent` kernel primitive.

A new system call was created in order to facilitate these notifiers and still allow the kernel to determine when it is safe to return.

Alongside this system call, a new set of system calls similar to those of `Send`/`Receive`/`Reply` were created. These new system calls `Share`/`Obtain`/`Respond` merely share a raw pointer with another task rather than copying data. This is very beneficial in a few cases where read-only data is passed between tasks. While this still incurs a cache miss, it does not have to copy the memory which significantly improves performance.

Clock

Our newest notifier is the clock notifier. It consists solely of a call to find its parent's id and an infinite loop that waits for the timer interrupt then sends to the clock server to communicate that a tick has occurred.

The clock server is responsible for creating the clock notifier and maintaining the tick counter. It is also responsible for responding to requests from other tasks. To do this, it registers itself with the name server.

Once registered the clock server enters an infinite fetch/decode loop for requests. When a time request is made it responds with the ticks counter. When a delay or delay until call is made the end-time is calculated and the caller's id is inserted into a priority queue. Whenever a tick is reported by the notifier the first element in the queue is checked, if it is due then it is responded to and the process repeats itself.

Hardware Interrupts

The major design decision that we made regarding handling hardware interrupts was to not pass through the kernel, thus not allowing for rescheduling. Our HWI handler is a four line C function, exploiting all the magic of `gcc` attributes. This function takes two lines to disable the interrupt (the constant is folded in the compiled assembly), and two more to check that the notifier is event blocked, and if so, to call an inlined function that unblocks the notifier. We used the `__attribute__((interrupt("IRQ")))`. As our internal function call is inlined, `gcc` will know exactly which registers were clobbered in the HWI handler, and will thus save only those registers to the stack while being careful to avoid erasing the scratch and argument registers. This attribute further adds a `^` to the final `ldfm` command as to set the `cpsr` when restoring the `pc`. The assembly for the HWI handler is 31 lines in `O2`, but that is longer than any execution path through it (although it's the potentiality for cache misses that makes or breaks performance). This handful of instructions is all that happens between when a HWI is raised while in user mode, and the return to the user task.

Idle Task and Timing

The idle task is not a real task like other user tasks. It is never present in the scheduler and will never be scheduled by normal execution. However, when the scheduler returns that it cannot find any active tasks the kernel explicitly activates a hard-coded idle task. This idle task does nothing but call the system call `Pass` to return control to the kernel.

In order to time execution, the kernel activates the 40-bit debugging clock as soon as it starts. It then calculates how long the kernel took to initialize and begins timing a number of different values.

When the scheduler returns a task, the kernel saves the elapsed time as kernel time, then activates that task. When that task interrupts back to the kernel, it records the elapsed time as user time as well as recording it in the task. Then the kernel handles the interrupt and records the time it takes as handler time. This repeats until the scheduler does not return a task.

When the scheduler has no tasks the elapsed kernel time is instead recorded as idle time. In this way the repeated failed scheduling attempts make up the bulk of the idle work of the system alongside the recording of the timer values.

When full optimization options are enabled (with the "make prod" compilation) this kernel benchmarking system is not enabled. The reason these recordings are disabled is because they are fairly costly, especially since they involve operations on 64-bit integers. In order to benchmark the kernel even with optimizations enabled there is another compilation option ("make prodebug") which compiles with optimization but still enables the "`DEBUG_MODE`" flag.

Scheduling

Another optimization performed in this version of the kernel was an update to the scheduler we use. One of the problems with the previous algorithm was that its runtime grew quadratically with the highest priority (lowest priority number) task. Since the idle task effectively has a priority of one higher than the maximum priority, it would cause the scheduler to spend a lot of time just rescheduling the idle task.

The main cause of this was the time taken to "shift" each scheduling queue up one level. This required a loop over each queue. To prevent this, the scheduler structure was expanded from a small list of pointers to a large array of pointers and queues. While it now takes $O(n^2)$ space and $O(n^2)$ time on initialization, it only takes $O(1)$ time to reschedule a task (where n is the number of priorities, it has always been $O(1)$ in number of tasks and thus $O(1)$ during runtime).

Additionally we found that the compiler refused to inline the rescheduling function. Unfortunately that function is one of the most called functions in the entire code-base and the extranneous stack manipulation is a significant hindrance to performance. Upon closer inspection it was discovered that the reason the compiler refused to inline the function was because it was written recursively.

By manually unrolling the recursion into a loop we were finally able to get the compiler to fully inline the rescheduling process. This significantly improved the performance of our scheduler even beyond the performance it had before. Additionally this actually reduced the size of the code as it turns out to have marginally simplified the generated assembly.

Results

The output of the tasks is rather long, consisting of over 50 lines. As such we will not be printing it in its entirety here since we would be likely to make a mistake during the copying process. Instead the program can be run and the output inspected manually.

The order of the output is predominantly determined by the delays of the tasks. Since the priority 3 task's delay is much shorter, it prints more often. As for when two tasks would unblock at the same time it would depend on their priorities which executed first. This could be tested by removing a single tick from the lowest priority task.

Finally the code prints the number of ticks on the 40-bit timer that elapse in various parts of execution. You may notice that the total time is less than the sum of all of the time parts, the reason for this is because it does not account for the time it takes the kernel to initialize. This is additional useful debugging information, but does not provide insight into the "steady-state" of the kernel. As such only time since the first task is scheduled is counted towards total time.

Finally at the end we divide the time in the idle task by the total time to come up with a percentage of time spent idle. While this is not fully deterministic, we have found that with optimizations turned on we reliably get a 96% idle time.

SHA of commit

The commit hash of the submission commit (which is on master) is:

`1f5869acc49dcd5b81a1ed9d5b59a08dac0a65b5`

The repository can be cloned from:

`gitlab@git.uwaterloo.ca:laburke/cs452_kernel.git` (SSH)

or

`https://git.uwaterloo.ca/laburke/cs452_kernel.git` (HTTPS)

There is a README in the root directory of the repository which outlines the loading command. To run the program once it is compiled just restart the machine and run `load -b 0x100000 -h 10.15.167.5 "ARM/path/to/kernel.elf"` to load it. Note that the loaded address is different from the default value.