CS 452 Kernel Four

Louis A. Burke (laburke) and Taras Kolomatski (tkolomat)

Revised: June 14th, 2017

Overview

> *"In concert videos the guitarists face each other, instruments high and tight against their chests, reflecting one another on an invisible y-axis, shutting the audience out. The stances fit the songwriting: Foals are a band who offer their listeners little in the way of graspable emotion or explanation."*

Thus reads the 5.9 *Pitchfork* review for the debut *Foals* album, *Antidotes*. The band's sophomore release was much better received and marked a genre shift that remained in the subsequent two albums. To the delight of critics, *Total Life Forever* expressed clear and appealing emotional intent with fewer words and less jagged edges, musically speaking. However, the *Pitchfork* review missed the complexity and intent of *Antidotes*. One notable track is paced with halts and jumps and conveys some degree of anxiety in its fragmented lyrics, which describe physiological and quantified psychological responses to the feeling of love. As opposed to not communicating clearly its meaning, the song clearly communicates a lack of clear communication; a qualia which many find natural is interpreted as confusing and foreign. Much effort is placed in a failed attempt to describe state in parameters projecting orthogonally to the single dimension of meaning.

Similarly, a young Hal Incandenza in *Infinite Jest*, who can recite the OED from memory, is faced with psychotherapy sessions after the death of his father. As opposed to clearly communicating his state, he digests recreational, pedagogical, and professional literature in psychology to avoid *failing* the meetings. The opening chapter, occurring chronological last, reverses this situation. Hal has clarity of meaning and intent all while communicating zero bits of information to his audience. Effectiveness of communication is inversely correlated with the effort taken to produce it. Effective communication is perceived as natural and follows immediately from percieved social cues of one's conversational partner.

In our fourth kernel milestone we open the traditional communication channels instead of relying on busy waiting to communicate with both the user and the trains. While this incurs a level of difficulty, that is rewarded with the powerful userspace freedom of asynchronous task I/O.

Structure

Architecture

Our kernel is currently a micro kernel, any drivers or other modules will have to be implemented as separate tasks. It currently consists almost entirely of a single core loop of execution. This loop transfers control to an active task using a context switch. It also incorporates a scheduler to determine which task should be executed. Finally a software interrupt handler is installed in order to provide kernel functions to user tasks.

The tasks themselves are loaded into hard-coded memory locations. There are 5 sizes of tasks. There are 4 "giant" tasks which are provided with 2mb of stack each. There are 14 "big" tasks which are provided with 1mb of stack each. There are 24 "normal" tasks which are provided with 256kb of stack each. There are 28 "small" tasks which are provided with 64kb of stack each. Finally there are 16 "tiny" tasks which are provided with 16kb of stack each. This takes a full 30 megabytes of space, but the default load address

of 0x218000 eats away at over 2 of the 32 megabytes. In order to combat this, we instead load the kernel at 0x100000. We then allow all addresses up to 2mb to be used for kernel code. The tasks are given task space above this. To avoid stomping on RedBoot's stack we set the kernel's stack to the first giant task space and reset its stack address to that of some variable already known to be within the kernel's stack. This does mean that if the kernel were ever to enter the scheduler it would have the wrong stack, but since the kernel never leaves the "zombie" priority of -1, it will never happen.

Context Switch

To transfer control to and from the kernel, we execute handwritten assembly constituting the context switch. The state of a user task is stored between the user task's stack and its `TaskDescriptor` in kernel space. The TaskDescriptor contains the saved `sp`, the contents of the `spsr` at the last trap out of the task's user space, and the return value of the last system call made by the task. The top of the user's stack contain the saved values of `r4-r12, lr` followed by the saved `pc`. To trap out of the kernel we do the following:

1. *Pass the `sp`, `spsr`, and `rval` into the `asm_EnterTask` function (this places the arguments in registers),*
2. *Restore the task's `spsr`*
3. *Store `r4-r12, lr` on the kernel's stack,*
4. *Pop the saved `pc` from the user's stack to the lr_{svs} register (this increments `r0` where the argument is stored),*
5. *Mask `cpsr` to switch to system mode*
6. *Restore the user `sp`*
7. *Restore the user registers `r4-r12, lr`*
8. *Mask `cpsr` to switch to supervisor mode*
9. *Pass the return value into `r0`*
10. *`movs pc, lr`*

Notice that beginning at step 4, we have definitively switched from working in kernel memory to working in user memory. If we were working in a one line cache, trapping out of the kernel would induce at most one cache miss. This is made possible by storing everything that we need from the `TaskDescriptor` in registers via argument passing.

When in user space, the active task yields control to the kernel by making a system call. System calls have a number, the code, that identifies the call type, and zero or more arguments. The only syscall with arguments in K1 is `Create`. The system call functions are wrappers for a mixed C/arm function that pushes the code and arguments on the user stack and issue an `swi`. At kernel startup, we write the address of a assembly function, `asm_EnterKernel`, to the trap table entry associated with software interrupts. This function performs the following operations:

1. *Copy into `r3`, the value of the current link register, which is the saved stack pointer at the time of the `swi`,*

2. *Mask* `cpsr` *to switch to system mode (this gives us access to* sp_{sys}*)*
3. *Copy* sp_{sys} *into* `r1` *as to save it's value before by the subsequent pushes,*
4. *Store* `r4-r12,` lr_{sys} *on the user space stack,*
5. *Store* `r3`*, and hence the saved* `pc`*, on the user's stack,*
6. *At this point,* `r4-r7` *are scratch registers; we use them to store the syscall arguments from the user stack, starting at the address currently in* `r1`*,*
7. *Copy the saved* `sp` *into* `r0` *to return,*
8. *Mask* `cpsr` *to switch to supervisor mode,*
9. *Save the syscall arguments in a kernel data structure (the address is written in code space at startup),*
10. *Copy the* `spsr` *into* `r1` *to return it*
11. *Restore* `r4-r12, lr` *from the kernel's stack, but place* `lr` *in the* `pc` *to return from* `asm_EnterTask`*.*

We switch from working in user memory to kernel memory at step 10. Once again, there is a single dividing point. Upon returning to the function that called `asm_EnterTask`, we will use the values in `r0,r1` to save the `sp` and saved program status register in the `TaskDescriptor`. These two functions exemplify the $ABB^{-1}A^{-1}$ design pattern.

The Ersatz Trap Frame

THE context switch allows us to reenter a task after having previously exited it via a software interrupt. Indeed, there must be a user state that is to be restored on the user space stack. Yet, we are able to reuse this infrastructure to enter a newly created task. We do this by writing a *ersatz trap frame* to the user stack during creation. From the kernel's perspective, we enter a new task *in medias ras* as though returning from an interrupt.

Scheduler

OUR scheduler is a modified implementation of the O(1) scheduler found in older versions of the linux kernel.

This scheduler currently consists of one stack for each provided priority (0-31). Stacks are used instead of queues to simplify execution and reduce memory footprint. These stacks do **not** correspond to the distinct priority levels directly. A task of any priority may eventually appear in any of the stacks.

When asked to schedule the next task, the scheduler merely returns the top of the 0th stack. Meanwhile it moves the task it returns to the top of the stack corresponding to it's priority. Thus if it has priority 10, it will be moved to the 10th stack. Importantly this means that if a task has priority 0 it will be placed directly onto the active stack to be popped immediately thereafter. As such priority 0 is reserved for tasks that must operate in real-time. These tasks will be scheduled immediately and no other tasks will be scheduled until they complete. Additionally this rescheduling happens extremely quickly, requiring only a handful of instructions. The effective C-code for a real-time task's rescheduling is:

```
int irrelevant = 1;
while (irrelevant) {
struct TaskDescriptor *ret = state->exhausted[0];
if (ret) {
state->exhausted[0] = ret->next;

if (ret->priority > 0) {
ret->next = *state->exhausted;
*state->exhausted = ret;
return ret;
}
}
}
```

Meanwhile if the 0th stack does not have an active task more work is done. It is here that this method performs slower than a round robin, but since by definition there must not be any real-time priority tasks left, taking a few extra cycles is not a problem. At this point each stack is moved one index lower, while the current 0th "active" stack is moved to the last stack. This cycles the tasks up one level of priority. In this manner a "highest" priority task (not real-time, but highest "fair" priority) will be rescheduled to the very next stack to be loaded. However a lowest priority task will have to wait for the 0th stack to empty a lot of times in order to get a single chance to run, after which it must wait again.

Of course this makes the scheduler horrendously inefficient in the case that a single low priority task is the only one running. However, since low priority tasks by definition don't need the scheduler to run quickly, this is not a problem.

Main Loop and System Calls

The main loop starts by initializing all of the task objects on its giant stack. It then sets up the first user task with priority 1 and adds it to the scheduler. Finally, the main loop is run. As long as the scheduler has a task ready to run we enter said task and wait for it to call a software interrupt. When it does we analyze the stack to determine what system call was sent and what its arguments were. Then we enact whatever is requested.

Message Passing

The message passing functionality is merely added to the existing scheduler functionality. Negative priorities are now used to indicate blocked tasks, so that the scheduler knowns not to reschedule them when it ignores them on the top of the active queue. In order to know if a blocked task has to be rescheduled or not, the scheduler marks each task with whether it is currently in the scheduling system or not.

As for message passing itself, each task descriptor now contains a queue of tasks waiting to send to it. When it calls receive it will accept one of those tasks and advance the queue. If the queue is empty it will block until it is not.

As for sending, tasks merely contain two pointers to buffers that can be written to by other tasks. Replying is pretty similar to send, except it also has to unblock the target.

The way we setup our kernel interface, kernel commands can only take 3 arguments. This allows all arguments to be passed in registers, but is not enough arguments for the message passing calls. To solve

this we had two options. The first was to arbitrarily wrap all arguments in a struct and pass a pointer to it, but that seemed clunky. Instead we created a buffer utility type. By replacing buffer/size pairs with a single buffer pointer we reduce even send to just three arguments. This also allows us to write the optimized and advanced (truncation checks) buffer copier in a separate module, with minimal overhead.

Name Server

Our nameserver is just a user task like any other. It is created early and takes up only a very small task space, but runs at high priority. The name server is fairly rudimentary, as it doesn't need to deal with a large dynamic load.

It consists of a buffer of stored names and their associated task ids. This buffer is filled from the top down with names as they are reported to the task. Instead of searching for old names to replace in the buffer, newly registered tasks merely continue to write names lower in the buffer. Meanwhile when searching for a name, the buffer is traversed from low to high, in reverse order. In this manner the newer names will always register before the older ones.

While this leaves us with a fixed total number of "RegisterAs" calls, the number we are allowed is easily modified at compile-time. Since we can know in advance exactly how many named services we will need, we can precompute this value reliably.

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.

Drivers

Our kernel continues to maintain its identity as a microkernel by keeping as much execution as possible in user tasks. To accomplish this the hardware interrupt is a tiny separate thread of execution running on a small fixed stack within the kernel's main stack. The handler does nothing but dispatch incoming events to live user tasks which then handle them.

All user handlers must disable whatever interrupts they wish to use lest they immediately be rescheduled after asking to wait for an event. This separates the concerns of the kernel and the drivers allowing the kernel to function independently of the devices attached to the various ports.

Since the notifiers will be reactivated every time their interrupts occur even if their task is trivial, it is of vital importance that they spend as little time as possible in send blocked states. To accomplish this, each notifier has its own courier to send its data back to the server that services it. In this way no matter how busy the server gets, the notifier will not have to be send blocked on the server's response (as the courier can respond immediately before delivering the message to the server).

Services and Servers

The servers, as well as other services are implemented using a templated system allowing for a large amount of code reuse for common design patterns in task implementation.

For instance, the generic server has a user implement an initializer, an internal data structure, a message data structure and a handler. It fills in the typical receive loop that is common to all services.

This system is highly extensible and in future we wish to implement a number of patterned tasks. Some services that would benefit greatly from templating are the three main tasks involved in any sparse resource: the notifier, courier, and proprietor.

Debugging

Now that the terminal's output is reserved for full-time use, we have had to turn to other methods of debugging. Given that there is a large amount of unused space between the bottom of our kernel and the top of RedBoot, we have elected to place our debugging output there. We then use a custom log reader to extract the data we store beneath our kernel and return it to a readable running commentary on the state of our kernel.

This allows us to continue to use our `debugio` library as well as its printf counterpart to print critical debugging information in real time. The only difference is that now that information is logged to data starting at address `0x50000` instead of being busy-waited to the terminal.

Assignment Zero Prime: Model-View-Controller

We followed the model-view-controller paradigm in our re-design of assignment zero. There are three controllers: for user input on the terminal, for input from the Maŕklin controller, and for reading the timer at set delays. The clock controller and view were short-circuited as it's use of interrupts is fundamentally different from those generated by I/O: the I/O controllers read multiple bytes before formatting a request to the model, whereas as soon as a timer delay completes we update the screen.

The terminal input controller sends an echo request to the model for every character typed by the user, but records the command in progress. When a carriage return is sent, the controller puts the command through a DFA generated by `RAGEL`. If the command is legal, its arguments are wrapped in a `struct` and are sent to the model.

The Maŕklin input controller runs in a loop which places a request for sensor states, reads 10 bytes, wraps them in a `struct`, and sends them to the model.

The model receives these requests, changes its representation of state, and sends to a view if and only if the request is required to generate output on the UARTs. The views that correspond to graphic presentation on the terminal are: the echo of the user's command, the display of switch states, the table of sensor activations, and the displayed time. The last view, as stated above, does not pass through the model. The views that correspond to the train are a general processor of commands, and two tiny views that coordinate delays for reversing trains and flipping switches. The tiny views are created for each event that requires coordination, and their functionality demonstrates that out kernel is able to recycle memory.

Once the program has launched, the following five commands are available:

Assignment Zero Prime: Commands

1. `tr <train-number> <speed>`
Where `<train-number>` $\in [1, 80]$, and `<speed>` $\in [0, 14]$.

This command sets the speed of a train. It does not affect whether or not the function key is toggled.

2. `l <train-number>`
Where `<train-number>` $\in [1, 80]$.

This command toggles the headlights of a train. It does not affect the train's speed.

3. `rv <train-number>`
Where `<train-number>` $\in [1, 80]$.

This command first stops a train, giving it three seconds to do so, then reverses it's direction and sets the speed of the train to it's original value. This does not affect the status of the headlights.

4. `sw <switch-number> <state>`
Where `<switch-number>` $\in [1, 18] \cup [153, 156]$, and `<state>` $\in \{\texttt{S}, \texttt{C}\}$.

This command sets the state of a specified switch to either straight or curved, corresponding to arguments `` `S' `` and `` `C' ``, respectively. The four switches constituting the two three way branches must be referred to by their decimal values.

5. `q`

This command terminates the program. Upon termination, the screen will clear, display an exit message, and state the maximum number of ticks $\left(\frac{1}{2} msec/tick\right)$ *that any iteration of the pooling loop took to complete. This number is often* 6 *if all commands are tested.*

Results

While our submission technically meets the requirements of the assignment, it is much less robust than either of our original A0 submissions. The reason for this is that unlike most other groups we do not intend to continue to use a text-based interface for long. Instead we intend to implement a true graphical user interface as a separate executable on the linux machine. This will allow us to provide more visual feedback more quickly and finer grained control over input (e.g. clicking on an image of the track to stop a train there).

It is because of this that our A0 remake is not as full featured as it otherwise might be. The important thing is that we have reliable interrupt-driven communication working on both com ports and a robust interface for intertask communication and scheduling.

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/user/kernel.elf"
```

to load it. Note that the loaded address is different from the default value.

Timing statistics, including percent idle: Are printed as debugging messages, specifically to low memory locations. To access debug output, run:

```
load -b 0x50000 -h 10.15.167.5 "ARM/user/log.elf"
```