CS 452 Kernel 1

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

June 22, 2017

Overview

The higher cognitive functions of the human mind are conjectured to be enabled due to the structure that language provides. Society hence endows infants with both the ability to form organized thoughts and to communicate them. Conversely, the discourse of individuals constitutes a society; a strange loop is formed, enforcing the sporadic breakthroughs of eons past via positive feedback. Society is thus structure in the present, which encodes its history in structural invariants that provide pushback forces allowing a low entropy state to be maintained. Society is thus context and control.

Having established communication in assignment zero, we move on the the problems of context and control in our first kernel milestone. A program is a sequence of instructions that modify state. Many instructions are only meaningful in the context and assumptions of its past instructions. A single misplaced command or erased register value is all that is required to derail execution. A kernel virtualizes the CPU, allowing for the peaceful, indeed synergistic coexistence of multiple tasks.

Our realtime kernel dynamically manages tasks on a TS-7200 board with a Cirrus system-on-chip and an ARM core. It provides a scheduler and a system call interface to allow tasks to be easily written. Being a real-time kernel it must allow tasks with sufficiently high priority to be given the execution time they need to do what they have to.

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 $\texttt{lr}_{\texttt{svs}}$ 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 $\texttt{sp}_{\texttt{sys}}$)*
3. *Copy $\texttt{sp}_{\texttt{sys}}$ into `r1` as to save it's value before by the subsequent pushes,*
4. *Store `r4-r12,` $\texttt{lr}_{\texttt{sys}}$ 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.

Results

The output of our code is:

```
Created: 4
Created: 5
TID: 6  PID: 1
TID: 6  PID: 1
Created: 6
TID: 6  PID: 1
TID: 6  PID: 1
Created: 6
FirstUserTask: exiting
TID: 4  PID: 1
TID: 5  PID: 1
TID: 4  PID: 1
TID: 5  PID: 1
```

The explanation is as follows:

The user task (id 1) calls create on the two low priority tasks (here 4 and 5). Since these tasks are lower priority than it, they don't get rescheduled and the initial task runs again. Now it calls create to create a high priority task (here 6). Since this task is a higher priority than it, it immediately executes, and since the priority we set it was real-time, it runs to completion before the first task runs again. As such the entire task runs printing its id (6) and its parent id (1) to the screen. Now control returns to the first task and it calls create again. Since task 6 already completed, that id is available for use again, so the exact same id is re-used and the real-time task runs to completion identically. Now control returns to the first user task, which exits. This leaves only the low priority tasks to run. As they are the same priority, and it is low, they take turns making one system call at a time. Therefore their outputs are interweaved.

SHA of commit

The commit hash of the submission commit (which is on master) is:
`5d9b5b7e633d551df48fc81ebe0cf06ad2072b4c`
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)