CS 452 Kernel 1

Brian Forbes & Daniel McIntosh
20617899 & 20632796

June 5, 2018

1 Building, Running, and Usage

The repository for the code can be found at `https://git.uwaterloo.ca/baforbes/cs452-kernel`. To build the executable, run the following command from the root directory:

```
make
```

To build this document, run the following command from the root directory:

```
make docs
```

Once run, the program will initialize the kernel, and then create the first user task. This task will then create 4 other tasks: 2 of maximum priority, and 2 of minimuim priority. Each of these tasks will print a line containing their task id and their parents task id, call Pass, print the same line again, and call Exit.

2 Program Description

Note that each referenced file corresponds to a header file of the same name in the `include` folder (for example, `src/kernel.c` to `include/kernel.h`)

3 Program Structure

3.1 Kernel Structure

The kernel contains a set of priority queues for task scheduling. While these queues are non-empty, it schedules a task from these queues, activates that task, and then handles the results. This process happens in `kernel.c`.

The kernel initialization process loads the address of the kernel entry (defined in `asm/activate.s` as the label `KERNEL_ENTRY_POINT`) into 0x28.

3.2 Scheduling

To schedule tasks, the kernel uses an array of ready queues (one per priority level). To schedule a task, the head of each ready queue is checked. The first task in the highest-priority non-empty ready queue is chosen as the next task to run. To add a task to a ready queue, it is added to the tail of that queue.

3.3 Activation

The activation code is in `asm/activate.s`. To activate a task, the following steps (in order) are taken:

1. Kernel registers $4 - 12, lr$ are saved onto the stack. Then, the CPSR is saved onto the kernel stack, as well as $r0$ (the argument containing the task struct).
2. CSPR_usr is loaded from the task struct, and SPSR_svc is set to it (so that movs will return it to user mode).
3. Tthe LR from the task struct is loaded into lr_svc, and the return value from the task struct is loaded into $r0$.
4. The kernel switches to System mode, reloads the user stack pointer, and then reloads user registers $r4 - r12, lr$.
5. `movs pc, lr` is called, to complete the context switch into user code.

Then, once another system call occurs, the kernel performs the following:

1. The kernel switches to System mode, stores the user registers $r4 - r12, lr$, and then stores the stack pointer.
2. The task descriptor is loaded from the kernel stack
3. The stack pointer, link register, SPSR, and arguments are stored in the task structure.
4. The syscall number is retrieved from the SWI call.
5. The kernel CPSR is reloaded from the kernel stack
6. The kernel registers $r4 - r12, lr$ are loaded from the kernel stack.
7. `mov pc, lr` is called to return from `activate`

3.4 Returning to the Kernel

Syscalls (to return to the kernel) are implemented in `src/syscall.c`. All syscalls work the same way: the link register, and then registers $r0 - r3$ are pushed onto the user stack, and then `swi` is called with the syscall ID as an argument. Once the syscall is completed, the registers are restored from the user stack, and the function returns.

3.5 Handling Results

Handle is implemented in `src/kernel.c`. It checks the id of the syscall, and then performs the action for that ID:

Create: A new task is created and added to the ready queues.

MyTid : The return value is set to the TID of the task.

ParentTid: The return value is set to the TID of the parent of the task.

Pass: Nothing happens.

Exit: The task is not added back to a ready queue.

For each of these (except Exit), after the action, the task is added back to the ready queue.

3.6 Data Structures

The only non-trivial data structure used is the ready queues. Each queue is a linked list, implemented by adding a pointer to the next task to the task struct (in `include/task.h`). The kernel stores the head and tail of each task queue (one per priority level).

All task data structure are allocated as part of kernel initialization. On startup, the kernel allocates an array of Task structures (in `task_pool`). Because tasks are never destroyed at this point, to allocate a task, it simply takes the next task from this array (using a counter variable) .

4 Explanation of Program Output

The output of the program should be:

```
Created: 1
Created: 2
MyTid: 3, MyParentTid: 0
MyTid: 3, MyParentTid: 0
Created: 3
MyTid: 4, MyParentTid: 0
MyTid: 4, MyParentTid: 0
Created: 4
FirstUserTask: exiting
MyTid: 1, MyParentTid: 0
MyTid: 2, MyParentTid: 0
MyTid: 1, MyParentTid: 0
MyTid: 2, MyParentTid: 0
```

First, the two low priority tasks are created. These are lower priority than the kernel, so they are not scheduled immediately. Then, task 3 is created, but it's creation is not printed by the first user task yet (because task 3 is higher priority, and therefore scheduled first. So, that task prints both of its outputs, and exits. Then, the first user task is the highest priority task again. It completes the printing for task 3 and creates task 4, not yet printing for the same reason. After task 4 prints its output, the first user task is the highest priority, allowing it to print the creation of task 4, and exit. Now, the two remaining tasks share the same low priority. So, they are each scheduled evenly, and the two lines are printed in turn by each of them, before they both exit.

5 SHA1 Hash of the Repository at the Time of Last Commit (commit) not including this PDF