CS 452 Kernel 3

Brian Forbes & Daniel McIntosh
20617899 & 20632796

June 7, 2018

1 Building, Running, and Usage

The repository for the code can be found at `https://git.uwaterloo.ca/baforbes/cs452-kernel/tree/k3`. 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
```

The executable will be built to `<repo_root>/bin/kernel.elf`

Once run, the program will initialize the kernel, and then create the first user task. This task will then create 7 other tasks: the Name server, the Idle task, the clock server, and 4 clock clients.

Each client will send a message to the first user task, waiting for the delay and iteration parameters from the first user task. Upon reception of those parameters, each client loops for the recieved number of iterations, waiting the recieved delay and printing the delay and the current iteration number each loop.

The parameters provided to the clients are, in order, delay times of $[10, 23, 33, 71]$ and iteration numbers of $[20, 9, 6, 3]$.

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`)

2.1 Context Switch

To facilitate interrupts, the context switch was updated to now push all user registers onto the stack (except PC and SP, which are saved separately). As a

part of this, syscall arguments are now saved on the user stack as well (they are pushed as registers r0-r3, with the fifth argument pushed onto the stack before the syscall. Previously, these were being saved into the task struct.

A second kernel entry point has been added to `asm/activate.s` for interrupt requests: `IRQ_ENTRY_POINT`. It saves the value 1 in the IRQ Stack Pointer. Later in the context switch, this value is checked. If the syscall is an IRQ, the IRQ LR is saved before returning to the kernel. If it is not an IRQ, the SWI Lr is saved instead, as previously.

2.2 IRQ Handling

In our `TaskQueue` struct, we added another array of task descriptor pointers to store which task, if any, is waiting on an event. We use a constant array to map the event values passed to AwaitEvent to an interrupt type/value. The only event value we currently have is `EVENT_CLK_3` = 0, for the clock 3 timer interrupt (ie. interrupt 51). We will add more as we need them. Event values will always be consecutive integer values, starting at 0. Doing it this way allows us to control which interrupts we handle first, and allows us the aforementioned array as small as possible, thereby increasing cache locality.

Curently, we don't need to return anything from AwaitEvent, so we dont. If we did, that would be managed by setting the return value in the task descriptor after the interrupt has been handled.

2.3 Clock Server

The clock server is implemented in `src/clock.c`

It stores the current time (in 10ms ticks), and a minheap of the TIDs of tasks that have called delay. After initializing this data, it spawns a Notifier process. The notifier awaits the clock interrupt event, and then sends a notification to the clock server.

Once the clock server has finished both of these initialization steps, it begins an infinite loop of receiving, and then handling messages. There are four different message types:

NOTIFIER Upon reception, the clock server replies to the notifier task, and adds a tick to the current time. It then notifies all tasks which have delay deadlines which end before the current time, by replying to them.

TIME The clock server replies to the requesting task with the current time.

DELAY The clock server adds the task and the deadline (calculated as current time + delay time) to the minheap of waiting tasks.

DELAYUNTIL The clock server adds the task and the deadline (provided by the requester) to the minheap of waiting tasks.

2.4 Idle Time

To measure idle time, we create a task with the lowest possible priority early in the first user task. The task runs an infinite loop in which it:

- reads the current time from the 40-bit debug timer
- runs an 8 instruction cycle loop (5 Data Ops + 1 branch), 500 000 times
- reads the current time from the 40-bit debug timer again
- compares the two times to get the time it took
- outputs 39321/total_time as a percentage at the top of the terminal

The time it takes for the inner loop to complete when not interrupted, 39321, was calculated experimentally and theoretically. First, we calculated that, for a total of 4 000 000 cpu cycles, running at a cpu clock speed of 100 MHz, it would take 0.04 seconds, or 39321.6 ticks of the 983.04 kHz debug timer. Then, to verify our theoretical calculations, we ran the kernel, with first user task only starting the idle task, all interupts off, and the idle task just outputing the difference in times instead of doing the division. The output from this confirmed our calculations and demonstrated that the time, as expected, the CPU is not clocked to run faster than it can load a cached instruction. While this approach does not account for any extra time potentially taken to re-load the 6 instructions into cache after the idle task is interrupted, we decided that the increased accuracy we would gain by locking those 6 instructions into the cache was not worth the effort, nor was the idle task important enough to sacrifice performance in the rest of the kernel.

2.5 Changes from Kernel 2

The only major change to the Kernel 2 architecture was the updates to `asm/activate.s`, as described in the Context Switch section.

2.6 New Data Structures

A minheap was used in the clock server to effeciently store the waiting tasks. It was implemented on top of a constant-size array. It has three operations: `add`, `remove_min`, `peek_min`.

add The element is added to the end of the array, and then bubbled upwards. This is logarithmic in the size of the array, which is a constant - so this takes constant time.

peek_min The element at the 0^{th} index is returned. This takes constant time.

remove_min The element at the 0^{th} index is swapped with the last element, and then removed. The new 0^{th} element is bubbled down. The removed element is then returned. Similarly to `add`, this is logatirhmic in the size of the array, which is still constant.

We chose a minheap for this purpose because it was more efficient compared to a sorted linked list, and more simple to implement without dynamic memory allocation. We felt as though implementing a pool of linked list nodes was more complex and error-prone than our minheap implementaion.

3 Explanation of Program Output

In the top-left corner, the idle time is displayed. The rest of the program outputs lines of the form:
`$(TID): $(DELAY). $(I)/$(N)`
Where `$(I)` is the current number of delays this task has completed, and `$(N)` is the total number of delays for this task.

4 Known Limitations

At the moment, the program cannot be run twice in a row without resetting the ARM box.

5 File Hashes and Commit hash

Output of `sha1sum src/* include/* asm/*`

Hash	File