CS 452 Kernel 4

Brian Forbes & Daniel McIntosh
20617899 & 20632796

June 18, 2018

1 Building, Running, and Usage

The repository for the code can be found at `https://git.uwaterloo.ca/baforbes/cs452-kernel/tree/k4`. 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 the other tasks: the name server, clock server, the four uart servers (send and recieve, for both uarts), the command and terminal servers, and the idle task.

The program will then create the terminal interface, accept various commands, output sensor data, and various other information. Trains and switches can be controlled with the following commands:

1. `tr <train_number> <train_speed>`
   Sets any train in motion at the desired speed, where speed is between 0 and 14. To stop, set the speed to 0. To turn on lights, add 16 to any speed command.

2. `rv <train_number>`
   Reverses any train.

3. `sw <switch_number> <switch_direction>`
   Throws the given switch to straight (S) or curved (C).

4. `q`
   Quits the program, and returns to RedBoot.

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 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.

2.2 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.

2.3 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. The 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`

2.3.1 IRQ Handling

A second kernel entry point was 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, in addition to the IRQ SPSR. If it is not an IRQ, the SWI Lr and service mode SPSR are saved instead, as previously. 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. Event values have been added for all required interrupts (the clock interrupt, and the uart interrupts). 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.

2.4 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.

2.5 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.

Quit: The kernel exits.

EnterCriticalSection: Interrupts are turned off. This is only used for panic, currently.

ExitCriticalSection: Interrupts are turned on. This is currently unused.

Destroy: Destroys the calling task.

CreateWithArgument: Creates a task, passing the given argument into its $r0$ register.

StoreValue: Stores a value in the kernel, with the given integer tag. Tags are defined in the enum `StorableValue`.

GetValue: Retrieves a values stored by StoreValue for the given tag. The GetValue and StoreValue system are used for statistic data, such as storing the idle percentage without sending from the idle task.

For each of these (except Exit), after the action, the task is added back to the ready queue. Actions not described above (Send, Recieve, Reply, AwaitEvent) are discussed in detail in the next section.

2.6 Message Passing

Each task struct now has a recieve queue (implemented via head and tail pointers). The handling for message passing syscalls is in `sys_handler.c`.

2.6.1 Send

When a task sends a message, it checks if the reciever is already send blocked. If so, it checks the length of the message, and compares it to the length the reciever expects. If these do not match, the return value will be set to an error. It then copies the message into the reciever struct, up to the minimum of the two lengths. If the reciever is not send blocked, the sender is added to the send queue of the receiver.

2.6.2 Receive

If the send queue of the task is not empty, the message is immediately copied into the reciever (including the length check as above), and the sender is moved to the reply blocked state. If the send queue is empty, the reciever moves into the send blocked state.

2.6.3 Reply

After checks to ensure that the task exists and is reply blocked, as well as the message length check, the message is copied into the reply message pointer, and the reply blocked task is moved into the ready state.

2.7 AwaitEvent

When `AwaitEvent` is called, if the event is the UART transmit event, the interrupt is enabled in the uart. This had to be done in the kernel, because if the interrupt is enabled before AwaitEvent was called, it would immediately fire, and then the waiting task would never recieve the interrupt.

To turn off an interrupt, the kernel checks in the VIC to decide which interrupt it is. For the clock interrupt, it is cleared by writing to the clock clear register. For the uart events, the kernel checks the uart IntIdIntClr register. For the recieve interupt, the data register is read. For the transmit interrupt, it disables the interrupt in the uart. To clear the modem register, the kernel writes to the IntIdIntClr register.

2.8 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
- calculates (theoretical_idle_time)/total_time as percentage.

The time it takes for the inner loop to complete when not interrupted, 15728 , 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.

The idle percentage to be output is then calcualted as an exponential moving average $(\alpha(pct_{new}) - (1 - \alpha)(avg_{old}))$, with a value for α of 0.6. After it is calculated, the idle time is stored using `StoreValue` to be printed by the Clock Printer, which updates every tenth of a second.

2.9 Nameserver

The nameserver implementation is in `name.c`. On startup, the nameserver stores its TID in a global int to ensure that all tasks know it. The nameserver stores names in a simple hashtable with linear probing (discussed in the Data Structures section). After initializing its storage struct, the nameserver enters a forever loop. On each iteration of the loop, it recieves a message, and executes the desired request. For WhoIs requests, the name is looked up in the hashtable. For RegisterAs, the name is added to the hash table.

2.10 Clock Server

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

2.10.1 Implementation

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.

2.10.2 Clock Printer

There is also an auxiliary clock task that prints the time every tenth of a second. It stores the current time on startup, sends to the terminal server, and then uses delayuntil to ensure that it wakes up at the time expected for the next tick to happen. DelayUntil is used instead of Delay since the send to the terminal server takes time.

In addition to printing the clock, the clock printer also sends the idle time to the terminal server to be printed.

2.11 UART Servers

UART servers are implemented in `src/uart.c`. Each uart has two servers: a send server, and a recieve server. They also each have a notifier for the send and recieve interrupt, and UART1 has a notifier for the modem interrupt.

2.11.1 Receive Server

The recieve server contains a buffer of tasks waiting to recieve data (which is small, as very few tasks (most likely, one task)) should be waiting to recieve at a time. It also contains a much larger recieve buffer, into which data which has not yet been retrieved by another task is stored. The recieve server recives one of two messages:

NOTIFY_RCV Upon reception, if there is a task waiting to recieve, it replies with the data directly to that task. If not, it adds it to the recieve buffer.

GETCH If there is data in the recieve buffer, it sends it directly to the task. If not, it adds the task to the recieve queue.

2.11.2 Send Server

The send server contains a send buffer (which is very large, as it is conceivable that a lot of data needs to be transmitted in large spurts, such as the outputting of the initial terminal), a CTS state, and a notifier state. In general, it behaves as a three-condition state machine: If the CTS has been negated and re-asserted, there is data to transmit, and the transmit notifier is ready, it sends a byte to the transmit notifier. For UART2, the modem condition is ignored.

It recieves three kinds of messages:

NOTIFY_SEND This updates the state of the transmit notifier to ready. If this results in all three conditions being statisfied, it sends the byte to the notifier to be sent to the uart.

NOTIFY_MODEM The CTS state is updated. Before a send happens, CTS must be negated, and then re-asserted. As such, there are 3 states: `CTS_ASSERTED`, `CTS_NEGATED`, and `SEND_COMPLETE` for the state where CTS is asserted but has not yet been negated. Similarly, if all conditions are now satisfied, the byte is sent to the notifier.

PUTCH The byte is either immediately sent (if the other two conditions are satisfied), or added to the transmit buffer, to be sent once the conditions are satisfied by the notifiers.

2.11.3 Notifiers

The transmit notifier waits for the transmit event (which results in the interrupt being both enabled, and later diabled by AwaitEvent), and then notifies the server. It then writes the byte sent back by the server. The send notifier waits

for the recieve event, and notifies the server, as does the modem notifier with the modem event.

2.12 Terminal Server

To prevent race conditions between print statements, all COM2 output is printed by the Terminal Server (`src/terminal.c`). The Terminal server recieves various forms of imput from other tasks (for example: echo, backspace, and newline from the command parser, switch statusses from the command server, sensor statusses from the sensor server) and outputs them using a courier. There is a buffer into which output is placed, and whenever the courier notifies the task that it is ready to send another character, it is replied to with that character.

It recieves input from COM2 via the command parsing server, which outputs in the form of `ECHO` requests. When a command is parsed, it is sent to the command server.

2.13 Command Server

Similar to the terminal server, the command server prevents race conditions between sending commands over COM1. It acts in a similar, way, recieving commands from the terminal server (once they are parsed).

Instead of using a courier for the command server, we decided to use Putc directly, since we thought the transmit buffer was large enough to never cause a significant command delay, for this assignment. In the future, it is likely that a send buffer and a courier will be added, similar to the terminal server.

The commands work in the following way:

tr The bytes to set the train to the given speed are sent to the server. The speed is also updated in in the command server, to be used for reversing.

rv The bytes to reduce the speed to 0 is sent to the server. Then, a short-lived task is spawned which waits the calculated stopping time ($75 + 350(15 - s)$, where s is the current speed). This stopping time is a simple linear approximation. After the delay, a notifier to sent the reverse command is sent to the server. Then, after another 100ms delay, a notifiaction to re-accelerate the train is sent.

sw A Solenoid Notifier task exists, which loops forever, delaying for 170 (150 plus a small buffer) milliseconds before notifying the server. When a sw command is sent, it is added to the switch buffer. Every time the server recieves a solenoid notification, it checks if there is a switch in the buffer. If so, it sends the switch command, and replies to the notifier. If not, it sends the solenoid off command, and replies, noting that the solenoid is now off. When there are no switches and the solenoid is off, the notifier is not responded to until the next switch command is recieved. This way, one switch can be flipped every 170ms, and the solenoid will be turned off on time after.

q The kernel exits the handler loop.

3 Data Structures

3.1 Ready Queues

TEach 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).

3.2 Task Pool

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) .

3.3 Hash Table

A hash table is used in the name server. We chose a hash table with linear probing because it was relatively small (only a factor 1.3 time larger than an array of the size of the maximum number of names we expected to store), easy to implement, and was a decent performance increase in the average case. We recognized that most stores and lookups only happened on kernel startup, but still saw benefit to these performance increases.

The hashtable was implemented as an array, with linear probing. To insert, the item is either put into the slot of the hash of the key, or the next empty slot below that (cyclically). Similarly, to look up a key, the slot for the hash of that key is queried. If it is not the correct item, the hashtable is probed linearly downwards (cyclically) until the key is found. In both operations, the operation fails if the hashtable is full.

As a hash function, we chose the djb2 function (from `http://www.cse.yorku.ca/~oz/hash.html`). This is because it is extremely simple, fast, and has good distribution of hashes.

3.4 Minheap

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.

3.5 Circluar Buffer

Circular buffers are used in a lot of user code, for our re-implementation of A0. In every area where servers need to store multiple pieces of data in order, a circular buffer is used (for example: `terminal.c` uses a circular buffer to store output that has not yet been sent to the uart server, the uart servers store data that has not yet been sent to the uarts or from the servers, and many others).

We chose to use circular buffers because they allow data to be continuously stored effectively infinitely (up to a cap at any given time), so data could be freely written and read in constant time. It was implemented as an array, with a read and write index. There is also an empty flag, to differentiate between the full and empty cases.

4 Known Limitations

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

Also, reversing the same train again (before the previous reverse has completed) results in undefined behavior.