CS 452 Kernel 4

Felix Fung (f2fung)
Dusan Zelembaba (dzelemba)

June 25, 2013

1 How To Run

```
> load -b 0x00218000 -h 10.15.167.4 "ARM/dzelemba/a4_final.elf
> go
```

2 Submitted Files

Files listed here can be found under `/u1/dzelembaba/cs452/frozen_a4/`

2.1 md5sums

```
af6f803dcb02bdd4ebf053c187f85d0b  ./Makefile
f351ccf69b83adb1e2aa0ef559a11eb8  ./all_tests.h
296f7f84cfb2c088fad7b6a69268d00b  ./bitmask.c
35bd0114e8113fbd214e3a389b4844c3  ./bitmask.h
fdae4efbd96a15f367684687820e7394  ./clockserver.c
4c86aadc2cca610cd67c7dd645371d33  ./clockserver.h
940c64a85e0ce0be2e92cdec7a6c3186  ./colorgcc
f14b1bd18405e4a13aa29074b4d3c265  ./context_switch.h
e6fe878a12d13ea01c7cdb38895772f8  ./context_switch.s
7ee58c4791f3415af01dadf7bd9aaae6  ./debug.c
b0ccaa5d019223ba79f3b63b16a14131  ./debug.h
9b124aeb5f84630af58c3abc2843c1a6  ./events.h
ec41e8a9ef6942e983896c9925fcf83a  ./first_task.c
f5d5f01a856e51feeefb0c672539e687  ./first_task.h
62e7110b8fcd1b231e9ff6f9975ae1c8  ./heap.c
e1bc08b27132977bdccf932a1432a917  ./heap.h
3b096de077660480045b6f57a090e482  ./icu.c
60122cea48e43550bf96143b8310ab30  ./icu.h
82a36ae15796491b9bed143f575f1dcd  ./idle_task.c
574e859de18cdbe722764e3230f226ca  ./idle_task.h
bde87554c6753ab69679a0096a9f0eab  ./interrupt_handler.c
c5b1026567dfc98fa4be22dde2163d98  ./interrupt_handler.h
```

```
8a001ef254d4399629af4a7523807a10  ./ioserver.c
bbd1a58e9662dc9efab4ffbeb4655ab0  ./ioserver.h
54fa949d614cd5de9e5714c2e86372b7  ./kernel.c
3ac7a3b2def34929bbd3198847a84c50  ./kernel.h
385fa8297c63ad1c00283993d3c3ba7e  ./linked_array.c
deb1fdece9e49c8553e3ca0f0d8ba729  ./linked_array.h
16008763b80fd197f248791c6e6e193d  ./main.c
b6c68a3abdfad9bb041e3868045f9eef  ./main.map
57f30631f3dc41ff841ad52627d8a821  ./messenger.c
d9fcdc742e0118ccfc12d5f5f5f3c043  ./messenger.h
c29551c6747135dc2869e0447f716627  ./nameserver.c
b2047a8021138c14fd7eea61641669d5  ./nameserver.h
1dc118c000601dc8accbed9bf54a2076  ./orex.ld
d141a877fdf638c446d299532e5db186  ./priorities.c
ad8404d6ff9b3cd23ab35da21a70037f  ./priorities.h
b5c40d28eb3da5c6b60eb86c3db9bea4  ./queue.c
6560cd52ba8bd3316b4b0ee359709bb3  ./queue.h
4da63eed62783ca80daa07e1a0082490  ./rps_server.c
f42681e19e3dc50960b701be52a53546  ./rps_server.h
2ab912e5d9e4a896d740138ff032f908  ./run_tests.c
0e390e153530b05118c87063a5dd3120  ./run_tests.h
f78c21899d82bfcf0eedae208fbcdca5  ./scheduler.c
621140a6884a581a7e896c056efe717e  ./scheduler.h
3195ae9768831b925bbe2045b8cd4c99  ./simple_sm.c
2a521c524bcdc6a4e301e9a923dc136d  ./simple_sm.h
d8e47dc8188b5b96afde359fbdc306e2  ./stdio.c
2ac1166e171b2ed1dea1e1667313c1e1  ./stdio.h
137c17fe3caba47c40e94bd7bbbec34f  ./stdlib.c
c1ff97f3805e8d7dbbf1cb33b015e34b  ./stdlib.h
0077ca5ad3b96616b9d58756ba4bb89a  ./string.c
8873c1a40c39ee31376a9b9fc12c70b9  ./string.h
792ad97b51eece81a637452da6056674  ./strings.c
7a1b1692e08413e0632172e8bab8cd11  ./strings.h
e8f18c3bf7a41896193f4b88ff5b7776  ./syscall.c
92dd1bcc7759c1cabeb6c3857d512d2a  ./syscall.h
45dbbbd7dc17c05a54fc2004028fcf77  ./task.c
8b9de320b012d017b7db5f384e7fd399  ./task.h
cb2aca3d1a4b7eb834acb7a3095702ac  ./test_helpers.c
cd42684f05820c5322c424f711652b5f  ./test_helpers.h
db212207fd1f822e6eac0e12504a74aa  ./tests/assignment_1_test.c
411e9b47d5dc508dff42812ff9f5a4fc  ./tests/assignment_3_test.c
81f4f7f81935b93a59c81d89d469b0fa  ./tests/assignment_4_test.c
36f8e02155a2ec078a6b9f39c7fa84c6  ./tests/basic_test.c
55d83782fdbdbd446d40150475e99a64  ./tests/clockserver_test.c
70ba721afbb029c5f5fd274610cd04e5  ./tests/hwi_test.c
bcb10795dece42854f1afdfeba58abc3  ./tests/message_passing_test.c
```

```
fb8849e3daee133a6dbd8d46aae51841  ./tests/multiple_priorities_test.c
e84e447e74452e7a0e121584432b0101  ./tests/nameserver_test.c
5b8c9bea1aa4bc89e901d3e804716727  ./tests/rps_server_test.c
a034d2695ffcb6bab511bc351d2c4787  ./tests/scheduler_speed_test.c
2c82682f782d8288a790d0873ee3fd99  ./tests/srr_speed_test.c
202c88f7b9170f8ca2eff1fa057737e8  ./tests/syscall_speed_test.c
0a009e1c2cf45b3a2e4fe54aa323bf7a  ./tests/task_creation_errors_test.c
d190d2b8da8936e79eb1b4d0cdd81903  ./tests/train_test.c
c83a8c9eb4d8bfd8ce93024e85e90615  ./tests/uart1_intr_test.c
3cd62f7995e4d59b66f0bc66a750a743  ./timer.c
077b0816d3b8542fd64497f05d360c41  ./timer.h
7553044ad445f2d963e97ccd87e8f965  ./train.c
9762ef219ec6f5854ef07a56cca470fb  ./train.h
473d468b5c307bb6eeaeca7d35acec54  ./ts7200.h
e9d1033301a81ab2d09fe477d7e69008  ./uart.c
04e5789e07fb586a1b6b101acf73a96b  ./uart.h
5626c0ebb1f29d24414bbd9cf9bb18ab  ./unittests/all_tests.c
a29718fd71b276fa65c0d946c5a5cd02  ./unittests/bitmask_tests.c
6fe66e343f638b7b2910da75028b113d  ./unittests/heap_tests.c
fe0a9d5b709dcd146c0c93875bf4a248  ./unittests/linked_array_tests.c
7c943438d2b499acbc3de149af0c2a6f  ./unittests/strings_tests.c
f068e72416d05f20660390bb1e50a4ef  ./unittests/test_helpers.c
24bc70d54528b73d8d7eb23f573921bc  ./unittests/test_helpers.h
```

2.2 Header Files

`bitmask.h` - Function definitions for bitmask.
`clockserver.h` - Kernel API for initializing clock server.
`context_switch.h` - Function definitions for compiler to use our assembly functions.
`debug.h` - Macros wrapping debugging functions that can compile away.
`events.h` - Constants for interacting with AwaitEvent.
`heap.h` - Function definitions for heap.
`icu.h` - All methods for interacting with the interrupt control unit.
`idle_task.h` - Kernel API for initializing idle task.
`interrupt_handler.h` - Processing of interrupts.
`ioserver.h` - Header for kernel talking to the ioserver.
`kernel.h` - Methods to initialize and run the kernel.
`linked_array.h` - An array with double pointers between inserted elements.
`messenger.h` - Function definitions for messenger.
`nameserver.h` - Function definitions for starting the nameserver.
`priorities.h` - Scheduling priorities.
`queue.h` - Function definitions for our queue implementation.
`rps_server.h` - Functions definitions for starting and using the rock, paper, scissors server.
`scheduler.h` - Function definitions for scheduler.

`simple_sm.h` - Simple state machines that tracks whether certain events have occured.
`stdio.h` - All methods to print to terminal, includes regular printf as well as bwprintf.
`stdlib.h` - Function definitions for commonly used functions.
`string.h` - Functions definitions for string structure.
`strings.h` - Functions definitions for string library.
`syscall.h` - Function definitions for syscalls.
`task.h` - Contains Task structure and function definitions.
`timer.h` - Function definitions for timer.
`train.h` - Function definitions for sending commands to the train set.
`ts7200.h` - Macros for important memory locations.
`uart.h` - Function definitions for methods talking to the uarts.

2.3 Source Files

`bitmask.c` - Generic bitmask functions. Used in faster scheduler.
`clockserver.c` - Clock server.
`context_switch.s` - ARM assembly used to switch in and out of tasks and kernel mode.
`debug.c` - Debugging facilities.
`first_task.c` - Spawns the name server and the clock server.
`heap.c` - A heap.
`icu.c` - Abstractions over the ICU.
`idle_task.c` - Idle task.
`interrupt_handler.c` - Processes hardware-trigger interrupts.
`ioserver.c` - ioserver implementation along with syscalls that talk to the ioserver.
`kernel.c` - The infinite kernel loop. Interrupts are processed here and tasks and scheduled.
`linked_array.c` - Implementation of our linked array. Was once used in the scheduler.
`main.c` - Beginning of execution, described below.
`messenger.c` - Coordinates message passing.
`Makefile` - Our makefile.
`nameserver.c` - Implementation of the nameserver, and WhoIs/RegisterAs syscalls.
`priorities.c` - A hack. A function that allows us to segment kernel-created tasks from user-tasks. Used to determine when the kernel can exit.
`queue.c` - A simple queue implementation.
`rps_server.c` - Implementation of the rock, paper, scissors server and wrappers for Send that clients can use to talk to the rps server..
`scheduler.c` - Our scheduler.
`stdio.c` - Output to the terminal.
`stdlib.c` - Common functions. Not actually a standard library.
`string.c` - String structure implementation.
`strings.c` - String library.

`syscall.c` - System calls.
`task.c` - Creating and managing tasks.
`timer.c` - Timing functions.
`train.c` - Sending commands to the train set.
`uart.c` - Methods talking to the uarts.

2.4 Test Files

`tests/basic_test.c` - Creates tasks of different priorities.
`tests/message_passing_test.c` - Tests all aspects of Send, Receive, Reply.
`tests/multiple_priorities_test.c` - Tests the correct scheduling of multiple tasks with different priorities.
`tests/nameserver_test.c` - Tests WhoIs & RegisterAs.
`tests/rps_server_test.c` - Implements a set of clients for the nameserver.
`tests/srr_speed_test.c` - The test that is run to analyze our performance.
`tests/syscall_speed_test.c` - Test the time required to execute various sys calls.
`tests/task_creation_errors_test.c` - Tests error return values for task creation.

`tests/assignment_3_test.c` - As described in assignment 3, a test spawning multiple clients to the clock server with various delay intervals.

`run_tests.h` - Header for run_tests.c.
`run_tests.c` - Calls all our tests.
`test_helpers.h` - Header for test_helpers.c. Different types of asserts and such.
`test_helpers.c` - Test helpers. Asserts and other things.

3 Program Description

3.1 Initialization

To initialize the kernel we do many things:

- Enable caches
- Set software interrupt handler address to our function k_enter
- Set hardware interrupt handler address to out function hwi_enter;
- Initialize timers, uarts, interrupts and kernel data structures.
- Create a first user task that creates all our servers.

3.2 Context Switch

We save all the registers on the top of the user's stack. During task create, we fill the user's stack with initial register values so that the first context switch into the user task behaves exactly like other context switches.

To support hardware interrupts we had to enhance out context switch. Our old context switch stored registers r4 - r13, the pc of the user, and the CPSR of the user on the user's stack. For hardware interrupts, we also had to store registers r0 - r3. To do this, we created a wrapper around `k_enter` and `k_exit`: `hwi_enter`.

The wrapper saves registers r0 - r3, the user's CPSR and PC on the user's stack. Then it sets the `Request` object to null (to signal an interrupt request) and sets up the SPSR_pc and LR_pc so that `k_exit` will jump back into the wrapper. It then jumps to `k_enter`.

Once the kernel finishes, we get back into the wrapper and restores the user's scratch registers, CPSR and PC.

3.3 Task Id Provisioning

We use a queue which is initialized with the universe of available task ids (0-127). These task ids point into an array of Task structures. We don't recycle task ids because we don't intend on every creating more than 127 ad infinitum.

3.4 Task Descriptors

Task descriptors are structs. They contain their tid, parent's tid, priority, return value for their last system call, stack pointer and a pointer to their stack space. This is suboptimal, but has been thus far unnecessary to optimize.

3.5 Scheduler

Our scheduler uses an array of queues to implement the priority queues. Our implementation uses a bitmask.

Our bitmask has one bit for every priority. If a bit is set, that means there are tasks's ready to be run for that priority. So, when a queue empties we set its bit to 0 and when a queue gets a first element we set its bit to 1. Then `scheduler_get_next_task` becomes a find lowest bit operation (0 is highest priority). To do this we used the linux kernel's implementation of `fls`.

3.6 Message Passing

To make better use of the cache, we've made a space-saving statically allocated data structure for message passing. We create a fixed-size array of INBOXes, MAX_TASKS in number. Each INBOX contains fields we reuse to contain EITHER a sent message and reply address (params of Send()), or the receiving address (params of Receive()). There are no collisions because Send() and Receive() cannot be concurrent for a single task. Each INBOX contains a NEXT

pointer, which may point to another INBOX or NULL allowing INBOXes to be chained together in a linked-list fashion. Then for MAX_TASKS, we also store 'head' and 'tail' pointers to INBOXes to represent the receive queue for each task in constant space relative to the length of the receive queue.

In summary, when blocking, tasks will write their addresses to their own INBOX, and it'll be chained into the receiving task's linked-list. By reusing INBOXes for future messages, we never allocate more memory.

3.7 Nameserver

The nameserver is a user space task that is started by the kernel. It uses a simple queue of strings, int pairs to hold each registration it is given. This results in an $O(n)$ (in number of registrations) runtime for both RegisterAs and WhoIs. This was considered acceptable because n will usually be small and these syscalls tend to be called during initialization, where performance isn't as critical.

The nameserver's task id is stored in a static variable that is visible only to the nameserver. The nameserver exposes a function, `get_nameserver_tid()` that provides access to the nameserver's tid. This way we can ensure that the nameserver only writes to this variable once during startup to avoid race conditions.

If a non-existent registration is passed to WhoIs, it will return -1. This was the simplest implementation for now and if a blocking version of WhoIs is needed in the future a separate syscall, WhoIs_Blck, will be written to provide synchronization methods for tasks needing to wait on other tasks to start.

3.8 AwaitEvent

Our AwaitEvent implementation takes in an abstract event id whose value has no correspondence with interrupt addresses or offset numbers. A user task passes in an event id, defined in events.h and the kernel will queue up the task in the appropriate event queue. This will put the user task in a new state, EVENT_BLOCK. We only expect an event queue to need to contain 1 Task so our structure of all queues is a simple array `Task* event_queues[NUM_EVENTS]`.

We enable the hardware interrupts corresponding to a specific event the first time a user task calls AwaitEvent on that event. This way interrupts (such as UART_TX) that get asserted right away won't get missed. It also has the added benefit that if a user program doesn't use a specific event, we won't needlessly track it.

When an interrupt occurs, the kernel checks each active interrupt bit (sorted by priority) and handles the highest one. This unblocks the user task. If the specific event being waited on requires a return value, we put it in the `retval` field in the `Task` structure, so that the user task will see it as a return value.

3.9 Clock Server

Our clock server is primarily a while loop which spends most of its time RECV_BLOCKed. It will receive messages either from other user tasks calling Delay or DelayUntil or the notifier. Delay or DelayUntil for times in the past will call Pass() instead of contacting the clock server. When a delaying task contacts the clock server, it is inserted in a priority queue (implemented as a heap) ordered by the absolute time of when it will be awakened. A heap was chosen for its `O(log(n))` inserts and deletes. In the worst case, all tasks will be delaying and the heap will be max size. Inserting then deleting all tasks from this heap will be `O(nlog(n))` in contrast to a linked-list with `O(n)` inserts making it `O(n^2)` total.

The notifier is a special task spawned by the clock server that waits on timer interrupts, and when it receives one, notifies the clock server and goes back to waiting for timer interrupts. When the clock server is contacted by the notifier, it increments a tick count. We then peek at the top of the heap to see if the current tick count is greater or equal to the upcoming wake time, it will awaken that task and repeat the check for any other tasks that qualify.

After each delay we print the `tid`, delay interval, and number of delays completed. What we see in the output, should be and is a list of delays such that they are sorted by the `no. delays completed * delay completed`. This makes sense because context switches should take relatively no time so the `nth` statement for a particular `tid` will be printed after its `n` accumulated delays.

3.10 IO Server

To handle all input/output to and from the terminal and and the train controller, we use a single server design who spawns 4 independent notifiers. 2 notifiers for checking when we are able to transmit to either the train controller or the terminal. 2 notifiers for checking if we've received data from either the train controller or the terminal. Since these events are independent, this is the minimum number of notifiers that made sense. The number of context switches are roughly the same while the single server design allowed us to save on `tid`s and reduced the amount of boilerplate code to write similar underlying server tasks.

3.10.1 When are we ready to transmit to the train?

Our notifiers are kept simple and listen to only a single abstract event each such as CAN_TRANSMIT_TO_TRAIN. In order to surface such an event, we must wait for both the CTS and TIS interrupts before we can guarantee this. Thus we've placed a simple state machine in the interrupt handler. Each interrupt sets a bit and we check that both bits are set before firing the event.

3.11 Terminal I/O

Now that we have an ioserver, we rewrote the bwio library to send requests to the ioserver instead. This lead to 2 complications:

1. Kernel code calls printf and the kernel can't send requests to the ioserver.
2. bwio relied on sending chars one at a time, but now that tasks can get interrupted, messages can get intermixed with one another.

To solve 1), we created a function `in_userspace` that checks the CPSR to see if the current code is running in user mode. If it is, it calls the ioserver, otherwise it calls bwputc. This isn't optimal as the kernel printing might crash the ioserver. However, this is ok for now as we just need the kernel to print when a fatal error occurs.

To solve 2), we added a new syscall, `Putstr`, that allows user tasks to send an entire string to the ioserver. Then, we modified our stdio to build up a string instead of sending one character at a time.