CS 452 Kernel 3

Felix Fung (f2fung)
Dusan Zelembaba (dzelemba)

June 11, 2013

1 How To Run

```
> load -b 0x00218000 -h 10.15.167.4 "ARM/f2fung/a3.elf"
> go
```

2 Submitted Files

Files listed here can be found under `________________`

2.1 md5sums

```
643b9c07859fa6a4ba33f112ceaf4efc  all_tests.h
296f7f84cfb2c088fad7b6a69268d00b  bitmask.c
35bd0114e8113fbd214e3a389b4844c3  bitmask.h
d8419bf98ae68cfec7cf8c5ad426df23  clockserver.c
4c86aadc2cca610cd67c7dd645371d33  clockserver.h
940c64a85e0ce0be2e92cdec7a6c3186  colorgcc
f14b1bd18405e4a13aa29074b4d3c265  context_switch.h
0b5b8505a24502e885a77bc4213b915b  context_switch.o
e6fe878a12d13ea01c7cdb38895772f8  context_switch.s
f96dbf43f2eb935a8d1fe48acdbbb730  debug.c
45c193986addf96a805a1200fec19e3e  debug.h
5a49cf03ce2afd47c862b6ab4ce8e3a6  events.h
90985d7c7002f9c2fd4a1116c83516ae  first_task.c
f5d5f01a856e51feeefb0c672539e687  first_task.h
9e71c7c723d7f16b3a5dfeb8c2a4a0e6  heap.c
e1bc08b27132977bdccf932a1432a917  heap.h
67976de6844564a9bd58822508f3a27c  icu.c
a046aadf54a41b680a89f97ed71d73e9  icu.h
afea4d1f6e49a4a945ddd357d0b9df6b  idle_task.c
574e859de18cdbe722764e3230f226ca  idle_task.h
2b7568f9b9e5dbbcb15437d2179ca3de  interrupt_handler.c
```

```
c5b1026567dfc98fa4be22dde2163d98  interrupt_handler.h
c89937a7b00988c424171430dc1a3235  kernel.c
3ac7a3b2def34929bbd3198847a84c50  kernel.h
385fa8297c63ad1c00283993d3c3ba7e  linked_array.c
deb1fdece9e49c8553e3ca0f0d8ba729  linked_array.h
f8775577c797a37a4d3f5e7874dff684  main.c
7e0e72519525a3ba322a9b4210f61dff  main.map
73b96d2cbe9a1397a87ccf24dd898ba1  Makefile
81330c063f3510d8b4a46fc3ee6bed6a  messenger.c
d9fcdc742e0118ccfc12d5f5f5f3c043  messenger.h
2c85c6b7919a92aec49ad1fcf10d074e  nameserver.c
b2047a8021138c14fd7eea61641669d5  nameserver.h
1dc118c000601dc8accbed9bf54a2076  orex.ld
d141a877fdf638c446d299532e5db186  priorities.c
ad8404d6ff9b3cd23ab35da21a70037f  priorities.h
d4112505ecd99a3c4745fde1735a0c35  queue.c
175206a0ac5e05e89f90855d013a58c9  queue.h
5e7448b250e18eb3bac69247367a2f59  rps_server.c
f42681e19e3dc50960b701be52a53546  rps_server.h
ee47b011a4aee5b1f9e3e8201bd3ff47  run_tests.c
0e390e153530b05118c87063a5dd3120  run_tests.h
2669fa2d59edd661eb5032365829c85f  scheduler.c
621140a6884a581a7e896c056efe717e  scheduler.h
a2a635a9a41459d5b1ecac434c898e3f  stdlib.c
bbfb9ed8184b862d2b73624989765847  stdlib.h
792ad97b51eece81a637452da6056674  strings.c
7a1b1692e08413e0632172e8bab8cd11  strings.h
1f9aacf2475e6d479db0006dddfbb754  syscall.c
ef7fb4831f561deccc67a1857ddf27af  syscall.h
289e1e90032ab3ea2124778d0f338923  tags
746b1aa0197b7982d0cbe332d4e061bb  task.c
8b9de320b012d017b7db5f384e7fd399  task.h
f6a07049571b3c0965a353a8e59acfbc  test_helpers.c
9570cd2291eb9823e5fb0c501aeea2b3  test_helpers.h
3cd62f7995e4d59b66f0bc66a750a743  timer.c
077b0816d3b8542fd64497f05d360c41  timer.h

6ce01266fda3b26d34fc191b81737a81  tests/assignment_1_test.c
493ae1fe8bf69ca728e92536230138bf  tests/assignment_3_test.c
9419f563ef21d56910b067f0d79d363c  tests/basic_test.c
01208b67347f69de75de78286f3efd74  tests/clockserver_test.c
d211d736533cb05c730a78b5150773bf  tests/hwi_test.c
55b24c0e448119af6615eab49c332fdf  tests/message_passing_test.c
9411afb1480ad3e38253a702d37d9f19  tests/multiple_priorities_test.c
6749a430b7547c5a3205f07722b1a48d  tests/nameserver_test.c
6d03ea974eb4d53e810726bbbe808486  tests/rps_server_test.c
```

```
1d6e4884db4db9241a636b02fc67c39e  tests/scheduler_speed_test.c
1719aad329fd9af560abd018d9f3592c  tests/srr_speed_test.c
825ce73ead9d85e46ad28f5d575d7966  tests/syscall_speed_test.c
1d4f94d256efbe01dad35ef9ffbc08fe  tests/task_creation_errors_test.c
```

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.
`idle_task.h` - Kernel API for initializing idle task.
`messenger.h` - Function definitions for messenger.
`nameserver.h` - Function definitions for starting the nameserver.
`queue.h` - Function definitions for our queue implementation.
`priorities.h` - Scheduling priorities.
`rps_server.h` - Functions definitions for starting and using the rock, paper, scissors server.
`scheduler.h` - Function definitions for scheduler.
`string.h` - Functions definitions for string library.
`stdlib.h` - Function definitions for commonly used functions.
`syscall.h` - Function definitions for syscalls.
`task.h` - Contains Task structure and function definitions.
`timer.h` - Function definitions for timer.

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.
`kernel.c` - The infinite kernel loop. Interrupts are processed here and tasks and scheduled.
`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.
`stdlib.c` - Common functions. Not actually a standard library.
`strings.c` - String library.
`syscall.c` - System calls.
`task.c` - Creating and managing tasks.
`timer.c` - Timing functions.

2.4 Test Files

`tests/basic_test.c` - This is a basic test. (This is only run for debugging)
`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 Scheduler

Our old scheduler used an array of queues to implement the priority queues. It was mentioned that this approach would be too slow once we added more priorities, so we upgraded our implementation to use 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`.

3.2 Context Switch

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 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 all hardware interrupts our kernel is interested in on boot. When an interrupt occurs, the kernel checks each active interrupt bit (sorted by priority) and handles the highest one. This unblocks the user task. Our AwaitEvent does not return values.

3.4 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.

4 Program Output

```
RedBoot> load -b 0x00218000 -h 10.15.167.4 "ARM/f2fung/debug/main.elf"
Using default protocol (TFTP)
Entry point: 0x0031d3f4, address range: 0x00218000-0x003231c0
RedBoot> go
tid: 7, delay interval: 10, # delay completed: 1
tid: 7, delay interval: 10, # delay completed: 2
tid: 8, delay interval: 23, # delay completed: 1
tid: 7, delay interval: 10, # delay completed: 3
tid: 9, delay interval: 33, # delay completed: 1
tid: 7, delay interval: 10, # delay completed: 4
tid: 8, delay interval: 23, # delay completed: 2
tid: 7, delay interval: 10, # delay completed: 5
tid: 7, delay interval: 10, # delay completed: 6
tid: 9, delay interval: 33, # delay completed: 2
tid: 8, delay interval: 23, # delay completed: 3
tid: 7, delay interval: 10, # delay completed: 7
tid: 10, delay interval: 71, # delay completed: 1
tid: 7, delay interval: 10, # delay completed: 8
tid: 7, delay interval: 10, # delay completed: 9
tid: 8, delay interval: 23, # delay completed: 4
tid: 9, delay interval: 33, # delay completed: 3
tid: 7, delay interval: 10, # delay completed: 10
tid: 7, delay interval: 10, # delay completed: 11
tid: 8, delay interval: 23, # delay completed: 5
tid: 7, delay interval: 10, # delay completed: 12
tid: 7, delay interval: 10, # delay completed: 13
tid: 9, delay interval: 33, # delay completed: 4
tid: 8, delay interval: 23, # delay completed: 6
tid: 7, delay interval: 10, # delay completed: 14
tid: 10, delay interval: 71, # delay completed: 2
tid: 7, delay interval: 10, # delay completed: 15
tid: 7, delay interval: 10, # delay completed: 16
tid: 8, delay interval: 23, # delay completed: 7
tid: 9, delay interval: 33, # delay completed: 5
tid: 7, delay interval: 10, # delay completed: 17
tid: 7, delay interval: 10, # delay completed: 18
tid: 8, delay interval: 23, # delay completed: 8
tid: 7, delay interval: 10, # delay completed: 19
```

```
tid: 9, delay interval: 33, # delay completed: 6
tid: 7, delay interval: 10, # delay completed: 20
tid: 8, delay interval: 23, # delay completed: 9
tid: 10, delay interval: 71, # delay completed: 3
Assignment 3 Passed!
Main Exiting...

Program completed with status 0
```

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.