CS 452 K3

Names:	Robert Elder, Christopher Foo
ID #:	20335246, 20309244
Userids:	relder, chfoo
Date due:	June 12, 2013

Running

The executable is located at `/u/cs452/tftp/ARM/relder-chfoo/k3-submit/kern.elf`.

The entry point is located at **"0x00045000"** or `%{FREEMEMLO}` It can be executed with caching enabled:

```
load -b %{FREEMEMLO} -h 10.15.167.4 ARM/relder-chfoo/k3-
submit/kern.elf
go -c
```

Description

Kernel

- The SWI vector entry code has been fixed by setting it to the correct location.
- Caching improves the performance of the program and Clock Slow Warnings should only appear on task creation and shutdown. We recognize that without caching more warnings may be printed. See Performance.
- When a user task is interrupted by the timer IRQ handler, the user state is pushed onto the IRQ handler stack. We recognize that this is not what we're supposed to do and we plan to fix this in the next deliverable. For now we have focused on meeting the timing requirements, and making sure that the amount of time before we unblock event blocked tasks is less than 10ms. The next step will be to push state onto the user stack, so that we can context switch directly to the Clock Notifier task, instead of just unblocking it.

System Calls

`AwaitEvent` Marks the task as `EVENT_BLOCKED`. The task will be unblocked by the Scheduler via the timer interrupt. This call currently does not return anything useful. The next deliverable will decide on how data is communicated back to the task.

`Time` Wraps a `Send` to the Clock Server. It first queries the Name Server for the Clock Server and then sends a `TIME_REQUEST` message. It expects back a `TIME_REPLY` message and returns the time.

`Delay` Similar to `Time`, it sends a `DELAY_REQUEST` message and expects back a `DELAY_REPLY` message.

`DelayUntil` Similar to `Time`, it sends a `DELAY_UNTIL_REQUEST` message and expects back a `DELAY_REPLY` message.

`TimeSeconds`, `DelaySeconds`, `DelayUntilSeconds` Same as above but in seconds. It simply converts the ticks into seconds before calling the system calls. These calls are simply for convenience.

Memory model

The memory model is now changed to look like this:

```
+-----------------+ 0x0020_0000
| RedBoot Stack   |
+-----------------+ 0x01fd_cfdc Starting value of redboot stack
| Redboot Buffer*|              after box reset
+-----------------+ 0x01FD_B09C
| Kernel Stack    |
+-----------------+ 0x01FD_B09C - sizeof(KernelState) - 400kb
| IRQ Stack       |              = KernelEnd
+-----------------+ KernelEnd - 500kb
| User Stacks     |
|                 |
+-----------------+ 0x0005_2804 (_EndOfProgram specified in orex.ld)
| Kernel          |
+-----------------+ 0x0004_5000 (%{FREEMEMLO} RedBoot alias)
| RedBoot         |
+-----------------+ 0x0000_0000
```

Redboot Buffer

After investigating some problems related to observing program crashes on the second and third execution of the 'go' command, it was discovered that redboot does not properly clean up its stack each time you run a program. Each time someone runs a program on a board, redboot pushes 80 bytes onto its stack and never removes it, unless you reset the board. This means that if no one ever reset the board, eventually the redboot stack will crawl through all of memory, and overwrite the user's kernel. It looks like no one else ever encountered this because they don't any data near the redboot stack like we do.

To prove that this is the case, you can create a simple program as follows:

```
int main(){
    asm (
        "LDR r1, [PC, #0]\n" // Load r1 with a memory ad-
dress we can save the sp into
        "ADD PC, PC, #0\n" // Jump over the address
        ".4byte 0x01000000\n" // SP gets saved here ev-
ery time the program executes
        "STR SP, [r1, #0]\n" // Save the stack pointer, then do dump -
b 0x01000000 -l 4, values increases by 0x50 each time until reset.
    );

    return 0;
}
```

Each time you run this program, you will observe that the saved stack value decreases by 0x50. I attempted to account for this on the exit of my main method, by creating a modified exit routine in assembly that pops the extra information off the stack, but this does not seem to matter.

`${FREEMEMLO}`

After consulting the RedBoot documentation, the entry point was moved to `0x00045000` to free up more memory for user stacks. We believe that this new memory location marks the start of safe memory that is not used as a guarantee

from redboot and we have not found any reason we cannot move the entry point to this location. This values comes from the a redboot alias %{FREEMEMLO} that can be used when loading the program instead of the literal address.

As well, we are able to have assert checks on stack boundaries. Using the `_EndOfProgram` linker symbol, we can check if a user stack pointer overwrites the kernel. There are checks for each user stack as well.

Stack values and sizes are configurable, and will generally give appropriate assertions if the memory model has conflicts that can cause corruption.

Message Passing

Kernel Messages, messages that are copied into the kernel, are now stored into an array, using Dynamic Memory Allocation (see below), instead of using a combination of ring buffers and queues. Refactoring to a simpler solution allows us to reduce the load on our brain while debugging the kernel. See Dynamic Memory Allocation for more information.

The maximum message size is now 16 bytes. This was done to reduce the time spent on message copying.

Clock Server

File: `clock.c`

The Clock Server runs in a loop receiving messages from the Clock Notifier or user tasks via the Public Kernel Interface wrappers. Whenever it receives a Event Notification from the Clock Notifier, it increments its tick counter. The tick size is defined to be 10ms.

Clock Notifier

File: `notifier.c`

The Clock Notifier runs in a loop:

1. Call `AwaitEvent`
2. Send a `NOTIFIER` message with `CLOCK_TICK_EVENT` id to the Clock Server.
3. Go to 1.

Data Structures

The Clock Server maintains a array mapping of TIDs to clock ticks in absolute time. Accesses to this mapping are constant time.

In order to address a bug in managing message queue data, we implemented a heap—the memory management kind—that is used only by kernel when queueing messages. The algorithm that performs the memory allocation is linear time, however this is ok because in practice this is bounded by the number of tasks, which is known to be less than 50. We have stress tested our kernel with several hundred tasks, and the empirical measurements of timings still keeps us under our goal of 10ms for being able to respond to events. We plan to further improve the run time of this function in the future.

Delay Requests

Whenever the Clock Server receives a delay request message, it checks whether the time is past in time. If so, it immediately replies back. Otherwise, it stores the requested time into the array mapping of TIDs to ticks.

Unblocking

After handling each received message, the Clock Server will check the array mapping of TID to delay time for ticks that are in the past. If so, it will reply back. This search is linear. See Performance.

Clock Slow Warning

Timer4 was enabled to use for debugging the performance of the kernel. The Clock Server uses this debug timer to time how long it takes for it to receive a notification from the Clock Notifier. It will print out a red warning message if the time is longer than the tick time (10ms) by 1ms.

Interrupt Handler

File: `kernel_irq.c`

Timer3 is enabled and counts down from 5080 to give 10ms interrupt intervals. The kernel also sets the CPSR to allow interrupts.

The interrupt handler will call the scheduler to unblock tasks and it also acknowledge Timer3.

The interrupt handler currently assumes that it is the Timer3 interrupt since no other interrupts are enabled. The next deliverable will check for the correct interrupt source.

Scheduler

File: `scheduler.c`

Changes:

- Scheduler code is now in its own file.
- Number of tasks in each event states are now tracked for debugging purposes.
- 32 levels of priority has been implemented.
- Blocked tasks are not requeued in the ready queue until it is actually ready.

The Scheduler has an array mapping of `EventID` to boolean. This array tracks whether at least one task is waiting on an event.

Event Unblocking

When the Scheduler is asked to unblock events on a particular `EventID`, it firsts checks the `EventID` array mapping. If it is true, then it continues.

The Scheduler will use linear search to find tasks that are `EVENT_BLOCKED` and change its state to `READY`. See Performance.

Priority Levels

Named priority levels have been maintained for backwards compatibility.

Priority	Int
HIGHEST	0
HIGH	8
NORMAL	16
LOW	24
LOWEST	31

Queue

File: `queue.c`

The `PriorityQueue` now uses an integer to track which priority level has items. When a bit is 1, it means there

is at least one item in the queue. For example, `00110000...` means there is at least one item in priority 2 and 3 queues. The count leading zero instruction is used so that we no longer need check all 32 queues when getting an item.

Memory

File: `memory.c`

`m_strcpy` has optimization improvements. It now can copy strings at 1, 8, or 32 octets at a time using block load and store instructions.

Dynamic Memory Allocation

A simple, but linear time, Dynamic Memory Allocation or heap was implemented. It is currently used for storing Kernel Messages.

It uses an array of booleans to track which blocks of memory have been allocated. The blocks of memory are implemented as a `char` array.

To allocate memory, it searches the array of booleans for a free spot and returns a pointer. Freeing memory simply requires calculating the index of array of boolean and setting it to 0.

See Performance.

RPS

The `RPSServer` has been refactored to fix synchronization problems. It is used for stress testing the OS. At least 480 tasks should run without problems.

Nameserver

Maximum name length has been arbitrary reduced to 8 bytes (including the null terminator) to fit within the reduced size Kernel Message.

IdleTask and AdministratorTask

The Administrator Task is responsible for helping us exiting to RedBoot.

The Idle Task runs when all tasks are blocked. The Administrator Task keeps track the number of tasks running. The Clock Clients will tell the Administrator Task when it has shutdown. After all tasks have exited, the Administrator Task will tell the Idle Task to exit.

Performance

For this deliverable, we have found the performance of the kernel to be acceptable after all tasks have been created. Acceptable is defined when the Clock Server does not lose more than 1ms from the Clock Notifier. We have kept linear solutions for now, because we believe that lost ticks during start up and shutdown is not important as the system is not doing anything useful during that time. However, we are still working on improving the overall context switching of the kernel.

Source Code

The source code is located at `/u4/chfoo/cs452/group/k3-submit/io/kernel3`. It can be compiled by running `make`.

Source code MD5 hashes:

```
chfoo@nettop40:~/cs452/group/k3-submit/io/kernel3$ md5sum */*.* *.*
50ef0e1e3c71ab1e795fc3d39f75ef9d  include/bwio.h
9af226f127c1fd759530cd45236c37b8  include/ts7200.h
da5c58f5a70790d853646f4a76f4c540  buffer.c
1f9a730c5017ddd24e18523d27dc471e  buffer.h
7f0e23ca0b7a2d818ca0d89f44a9becc  clock.c
12a8e72b6edd3ce9d39eec8f40face92  clock.h
1eaabf4c531773b21a4476aa9fbc3e06  kern.c
84c480712ffdc5fc8c854eeddba7ee75  kern.elf
d41d8cd98f00b204e9800998ecf8427e  kern.h
61a363555055c09fa50cacbcf133fc3d  kernel_irq.c
7dd2e35c54b6e20fd30ccdc3f8cc8c78  kernel_irq.h
0a6099b9d838bf192589c5d18a73d6a9  kernel_state.h
eeea82060a8efac1f1846b8e49cfc699  memory.c
c69b2cd31667898de90b5ea6968b34d5  memory.h
adcff2244ac92050360eacd7ab4f5dd9  message.c
4a69b1710f2b62b62dc12034c5a061ef  message.h
586eb93d3bdbf0b0895d278286a42982  nameserver.c
83a806d2e93bb4fbc2316ba853e3ff6c  nameserver.h
b5dc849ede8d0e14e1b8c93b364c2c2f  notifier.c
e1068badfd5a00f1fc498907eaece5fc  notifier.h
8d46598b0da4113f701c348f64657a84  orex.ld
ebaa2b3e71275031c2a1ce6feabb5113  private_kernel_interface.c
0bb2f28edaa36009df8693eb8e70248c  private_kernel_interface.h
05dc90397d0064c2a0183fb5a904424b  public_kernel_interface.c
f7fa9aae27bde825d09f995b237bedbf  public_kernel_interface.h
8878081d654354ea6357008d0b757342  queue.c
edad985ef0a0e1364ff31f81fdce035b  queue.h
91fbdbffeb090806d35dc54cb2e0627a  random.c
7b31c57ff692317d816c839156382596  random.h
58251ec1b8c900d4627f03baaf8a793a  readme.rst
eb5a60f060d101d2536e96298aab4112  readme.tex
7e9cbadd0b0bfbb4cb42477bcd1d4cc7  robio.c
d85c51626cee0d148dc9506211b5b2b2  robio.h
a2f7a0f7b52cf98176cb215f8232497e  rps.c
616ea2c1d0d273b41c55cbba5096a145  rps.h
4825d154b846a1c8f566502f157c9fed  scheduler.c
f07b9c5a26befff2ca7ae5faef6f113b  scheduler.h
4778a48d9ab01c1ca35b914275a56641  swi_kernel_interface.s
3470592bb0bfcd96ff5c597d5692e644  task_descriptor.c
5dd67fbba64e041c0acaba983aca92e6  task_descriptor.h
eeb70ad77d28002eb76c8d02425e7db0  tasks.c
c7ac97c4750ffa3af955d3d329a9e42d  tasks.h
```

Elf MD5 hash:

```
chfoo@nettop40:~$ md5sum '/u/cs452/tftp/ARM/relder-chfoo/k3-
submit/kern.elf'
84c480712ffdc5fc8c854eeddba7ee75  /u/cs452/tftp/ARM/relder-chfoo/k3-submit/kern.elf
```

Git sha1 hash: `c20bb3a31e2fb6f507e9e6aace28e99c10d9f454`

Output

Based on the values described, the tasks should output in chronological order:

```
| 3, 4, 5, 6
=============
  10 .  .  .
  20 .  .  .
  .  23 .  .
  30 .  .  .
  .  .  33 .
  40 .  .  .
  .  46 .  .
  50 .  .  .
  60 .  .  .
  .  .  66 .
  .  69 .  .
  70 .  .  .
  .  .  .  71
  80 .  .  .
  90 .  .  .
  .  92 .  .
  .  .  99 .
  100.  .  .
  110.  .  .
  .  115.  .
  120.  .  .
  130.  .  .
  .  .  132.
  .  138.  .
  140.  .  .
  .  .  .  142
  150.  .  .
  160.  .  .
  .  161.  .
  .  .  165.
  170.  .  .
  180.  .  .
  .  184.  .
  190.  .  .
  .  .  198.
  200.  .  .
  .  207.  .
  .  .  .  213
```

This ordering gives and expected printing sequence of
3-3-4-3-5-3-4-3-3-5-4-3-6-3-3-4-5-3-3-4-3-3-5-4-3-6-3-3-4-5-3-3-4-3-5-3-4-6
which is identical to the ordering that our program produces:

```
[...Output trimmed...]
FirstTask Start tid=1
ClockServer TID=3: start
FirstTask begin receive
```

```
RegisterAs for ClckSvr returned OK. tid=3
ClockNotifier TID=9: start
RegisterAs for Admin returned OK. tid=4
ClockClient TID=5: start
ClockClient TID=6: start
ClockClient TID=7: start
ClockClient TID=8: start
FirstTask Exit
ClockClient TID=5: Got delay_time=10, num_delays=20
ClockClient TID=6: Got delay_time=23, num_delays=9
ClockClient TID=7: Got delay_time=33, num_delays=6
ClockClient TID=8: Got delay_time=71, num_delays=3
SLOW! 13144us
RegisterAs for Idle returned OK. tid=10
ClockClient TID=5: I just delayed delay_time=10, i=0
ClockClient TID=5: I just delayed delay_time=10, i=1
ClockClient TID=6: I just delayed delay_time=23, i=0
ClockClient TID=5: I just delayed delay_time=10, i=2
ClockClient TID=7: I just delayed delay_time=33, i=0
ClockClient TID=5: I just delayed delay_time=10, i=3
ClockClient TID=6: I just delayed delay_time=23, i=1
ClockClient TID=5: I just delayed delay_time=10, i=4
ClockClient TID=5: I just delayed delay_time=10, i=5
ClockClient TID=7: I just delayed delay_time=33, i=1
ClockClient TID=6: I just delayed delay_time=23, i=2
ClockClient TID=5: I just delayed delay_time=10, i=6
ClockClient TID=8: I just delayed delay_time=71, i=0
ClockClient TID=5: I just delayed delay_time=10, i=7
ClockClient TID=5: I just delayed delay_time=10, i=8
ClockClient TID=6: I just delayed delay_time=23, i=3
ClockClient TID=7: I just delayed delay_time=33, i=2
ClockClient TID=5: I just delayed delay_time=10, i=9
ClockClient TID=5: I just delayed delay_time=10, i=10
ClockClient TID=6: I just delayed delay_time=23, i=4
ClockClient TID=5: I just delayed delay_time=10, i=11
ClockClient TID=5: I just delayed delay_time=10, i=12
ClockClient TID=7: I just delayed delay_time=33, i=3
ClockClient TID=6: I just delayed delay_time=23, i=5
ClockClient TID=5: I just delayed delay_time=10, i=13
ClockClient TID=8: I just delayed delay_time=71, i=1
ClockClient TID=5: I just delayed delay_time=10, i=14
ClockClient TID=5: I just delayed delay_time=10, i=15
ClockClient TID=6: I just delayed delay_time=23, i=6
ClockClient TID=7: I just delayed delay_time=33, i=4
ClockClient TID=5: I just delayed delay_time=10, i=16
ClockClient TID=5: I just delayed delay_time=10, i=17
ClockClient TID=6: I just delayed delay_time=23, i=7
ClockClient TID=5: I just delayed delay_time=10, i=18
ClockClient TID=7: I just delayed delay_time=33, i=5
ClockClient TID=7: Exit
ClockClient TID=5: I just delayed delay_time=10, i=19
```

```
ClockClient TID=5: Exit
ClockClient TID=6: I just delayed delay_time=23, i=8
ClockClient TID=6: Exit
ClockClient TID=8: I just delayed delay_time=71, i=2
ClockClient TID=8: Exit
AdministratorTask_Start: Got 4 shutdowns needed 4, shutdown send 1
SLOW! 12815us
NameServer_PrintTable: Tid=3 Name=ClckSvr
NameServer_PrintTable: Tid=4 Name=Admin
NameServer_PrintTable: Tid=10 Name=Idle
SLOW! 12885us
ClockServer TID=3: end
ClockNotifier TID=9: exit
AdministratorTask Exit
No tasks in queue!
[...Output trimmed...]
```

The 'SLOW' statements occur when it would have taken more than 10ms to unblock a task that was blocked on `AwaitEvent`. For now, these situations only occur during startup and shutdown, and we plan to address this before the next part of the kernel. Note that this does not occur during the required testing.