CS 452 K4

Names: Robert Elder, Christopher Foo
ID #: 20335246, 20309244
Userids: relder, chfoo

Date due: June 24, 2013

Running

The executable is located at `/u/cs452/tftp/ARM/relder-chfoo/k4-submit/kern.elf`.

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

```
load -b %{FREEMEMLO} -h 10.15.167.4 ARM/relder-chfoo/k4-
submit/kern.elf
go -c
```

Commands

tr TRAIN SPEED Set the train speed.

rv TRAIN Slows, stops, and reverses train. The final speed is hard coded to 5.

sw SWITCH DIRECTION Changes the turnout direction. DIRECTION is either S or C.

q Quits the program.

Description

Kernel

- Caching improves the performance of the program and will be mandatory for this deliverable. Due to some issues with timing that we will be addressing soon, some of the important user tasks will deadlock if you do not run the program with caches. The root cause of this problem is related to improperly attempting to send bytes to the train when the FIFO buffer is empty instead of waiting for CTS to be driven low, then high.

System Calls

- System calls support up to 5 arguments.

`Create` Returns the new task id, `ERR_K_INVALID_PRIORITY -1`, or `ERR_K_OUT_OF_TD -2`

`MyTid` Returns the current task id

`MyParentTid` Returns the parent task id. The parent task id is always returned regardless of the parent's state.

`Pass` (Rescheduling happens as normal in the background.)

Exit Task is marked as `ZOMBIE` (and rescheduling happens as normal in the background).

Send Sends a message to the given task ID. `-3` code is not implemented.

Receive Blocks until a message is received. Returns the size of the message which will be typically `MESSAGE_SIZE 16`

Reply Replies a message to the task. On errors `-3 -4`, an assert will fire before returning to aid in debugging.

RegisterAs Prepares a `NameServerMessage` structure with a message type of `REGISTER_AS` and sends the message to the Name Server. `0` is always returned because the Task ID is hard-coded and the call should never send to the wrong task.

WhoIs Prepares a `WHO_IS` message type and sends it to the Name Server. As noted in `RegisterAs`, we either return a Task ID or 0 if the task has not been created. However, the task ID returned may be in a zombie state.

AwaitEvent Marks the task as `EVENT_BLOCKED`. The task will be unblocked by the Scheduler. This call always returns 0 and the user task will be responsible for obtaining the data themselves. `AwaitEvent` supports only 1 task per event type.

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.

Getc Sends a message to either Keyboard Input Server or Train Input Server. It will block until the servers have a character to return.

Putc Sends a message to either Screen Output Server or Train Output Server. The servers will place the character into the server's Char Buffer.

PutString Formats the string and calls `Putc` for every character.

PutcAtomic Like `Putc`, but accepts multiple characters and guarantees the characters are placed into the queue sequentially. This call is useful to ensure that two byte commands are not separated by a single byte command.

SendTrainCommand Sends a message type `TRAIN_COMMAND` to the Train Command Server. The call is for convenience.

Memory model

The memory model looks like this:

```
+----------------+ 0x0200_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
```

Entry

The entry point is located in `kern.c`.

The kernel follows the following:

1. Sets the location of our SWI and interrupt routines.
2. Sets the location of our stacks.
3. Initialize the kernel (File `private_kernel_interface.c`:`k_InitKernel()`).
    1. Save the SP and LR values so the kernel can exit back to RedBoot.
    2. Initialize the pseudo Task Descriptor.
    3. Initialize the queues.
    4. Set the SP and LR value of the pseudo Task Descriptor to the Kernel State
    5. Call the `asm_KernelExit` routine to push the values to the register.
4. Jump to KernelTask_Start (File `tasks.c`)
5. Start our first user task that starts the 4 other generic tasks.

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

Messages are `structs` that are casted into `char*`. This casting allows us to manipulate messages more easily with type safety rather than dealing with raw `char`. Note we use GCC attribute syntax to word align the character array as the GCC compiler does not realize we are type punning.

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 message size is fixed to 16 bytes. Using a fixed value allows for consistency. As well, this low value is meant to reduce the time spent on message copying.

Task Descriptor (TD)

File: `task_descriptor.c`

The TD, a `struct`, holds important information such as the task id, state, and return values.

Queues

File: `queue.c`

The queue, a `struct`, is implemented as a ring buffer. A start and end index is used to point to the start and end of the array. Each item is a `void*`. The ring buffer allows adding and removing an item from the queue in constant time. A null pointer is returned if the queue is empty.

Priority Queue

The Priority Queue uses 32 levels of priority by using 32 Queues.

Named priority levels are available. Note the highest priority is 0 and the lowest priority is 31.

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

When retrieving an item, the Priority Queue 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 avoid checking all 32 queues when getting an item. `0` is returned when there is no item.

Char Buffer

The Char Buffer is a queue of characters. Like Queue, it is implemented as a ring buffer. However, the Char Buffer requires checking whether the Char Buffer is empty before getting an item. It is necessary because a return value of 0 indicates a byte value of 0.

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.

Name Server

File: `nameserver.c`

The name server uses a 2D `char` array. The maximum name is limited arbitrary to 8 letters including the null terminator. The small value reduces message copying and string comparison time. The array index corresponds to the Task ID for simplicity and constant time operations.

It does the following:

1. `Receive` a message casted to `NameServerMessage`
2. Determine the request type.
3. Look up or set the value in the array.
4. If it receives a `NAME_SERVER_SHUTDOWN` message type, it will `Exit()`

- Requests to set the name multiple times overwrites the previous value.
- 0 is returned when an invalid Task ID is provided.
- The Task ID is hard-coded to `2`.
- Look ups are linear time but bounded to the maximum name size of 8.
- Setting names are constant time.

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.

After reviewing feedback from the previous deliverable, we have refactored the memory heap to work in constant time for both requesting and releasing memory. The use of the heap is further justfied since it is only used when queuing messages inside the kernel. Constant time allocation and de-allocation is accomplished by use of a stack. The stack is initialized to contain pointers to all memory blocks that are free. A request for memory pops a pointer from the top of the stack, and de-allocation pushes the released pointer onto the stack. This allows constant time random-access de-allocation, while maintaining constant time allocation.

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

Unblocking tasks on events has been improved to work in constant time. 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`

Vectored interrupts are used.

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.

UART1RXINTR1, UART1TXINTR1, UART2RXINTR2, UART2TXINTR2, are enabled when there is a Task waiting for it. The IRQ handlers will disable the respective interrupt after it has fired. UART Clocking problems are avoided as our context switch is greater than 50 NOPs.

Scheduler

File: `scheduler.c`

- Has 32 levels of priority.
- Number of tasks in each event states are tracked for debugging purposes.

- Blocked tasks are not requeued in the ready queue until it is actually ready.
- Preemptive scheduling is supported

The following describes the process of context switching:

1. a) If the context switch occurs because of an interrupt, the task's state is pushed onto the IRQ stack then popped back onto the user's stack.
   b) If the context switch occurs because of a kernel call, the user's state is saved before entering the SWI call. We are aware that this design is discouraged, but because everything is working right now, we have put refactoring this on the back burner.

In both cases we remember what method was used to enter the kernel so we can invoke a symmetric exit routine when the task is re-scheduled. The design decision to have two methods of entering and exiting the kernel was done to allow future optimizations related to the fact that some context switch operations only need to be done for one method and not the other. This was also done to preclude the possibility of errors resulting from re-scheduling a process via the wrong method. For example, attempting to set r0 to set a non-existent return value for a task that was preempted.

2. The user task's SP, LR, and SPSR values are saved into the current task descriptor.
3. Any related values are also saved into the TD.
4. The next task is selected (`schedule_next_task()`).
   1. A task is removed from the Priority Queue.
      - Any tasks in the `ZOMBIE` or blocked state indicates a logic error.
   2. Use the pointer to the task as the next task to be run.
   3. Set the selected task as `ACTIVE`.
   4. Reschedule the selected task by adding it back to the Priority Queue.
5. If there no more tasks in the Priority Queue, the kernel exits back to RedBoot using the values we saved on the Kernel State.
6. Otherwise, the SP, LR, and return values are saved into the Kernel State.
7. Assembly routine `asm_KernelExit` pushes these values to the registers.

Events

When a task calls `AwaitEvent`, the Task ID is placed into an array mapping of Event ID to Task TD.

If the Event is related to Serial IO, the appropriate interrupt is enabled for it.

When the Scheduler is asked to unblock events on a particular `EventID`, it will check the array mapping and unblock this task by adding it to the Ready Queue.

Event IDs

`CLOCK_TICK_EVENT` A Timer3 clock tick interrupt has fired

`UART1_RX_EVENT` A UART1 receive holding register empty interrupt has fired

`UART1_TX_EVENT` A UART1 transmit holding register empty interrupt has fired

`UART2_RX_EVENT` A UART2 receive holding register empty interrupt has fired

`UART2_TX_EVENT` A UART2 transmit holding register empty interrupt has fired

Memory

File: `memory.c`

- `m_strcpy` copies strings at 1, 8, or 32 octets at a time using block load and store instructions.
- `m_strcmp` compares two strings, 1 character at a time.

Dynamic Memory Allocation

A simple, Dynamic Memory Allocation or heap was implemented. For this deliverable it has been refactored to use constant time allocation and deallocation. It is currently used for storing Kernel Messages.

See Data Structures for implementation details.

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.

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.

Random Number Generator

File: `random.h`

A LCG is used as the random number generator. It uses the GCC values as noted on Wikipedia. The seed is multiplied by an arbitrary number to get the generator going.

Assert

The assert statement has been enhanced to show Thomas The Tank Engine. Please do not be alarmed when you see it.

The assert function has also been modified to make sure that interrupts are disabled when an assertion is fired so that a user task that assert fails does not simply get ignored when its time quantum expires. This was necessary because of preemption.

Serial IO

File: `uart.c`

- FIFOs were not used for this deliverable.

The following Serial IO notifiers call `AwaitEvent`

Task	Event ID	Reports to
Keyboard Input Notifier	UART2_RX_-EVENT	Keyboard Input Server
Screen Output Notifier	UART2_TX_-EVENT	Screen Output Server

... continued on next page

Task	Event ID	Reports to
Train Input Notifier	UART1_RX_-EVENT	Train Input Server
Train Output Notifier	UART1_TX_-EVENT	Train Output Server

UART Bootstrap Task

The UART Bootstrap Task is responsible for setting up the UART clock speeds and settings. It also starts up the servers.

Keyboard Input Server, Train Input Server

The Input Servers receive keyboard and train inputs. They have a Char Buffer and receive byte data as notified. `Getc` callers will have their task IDs queued. Once Char Buffer contains data, the `Getc` callers will be replied with the character.

Screen Output Server, Train Output Server

The Output Servers send screen and train outputs. They have a Char Buffer and send bytes as notified. `Putc` callers will send the character to the server and the character is queued onto the Char Buffer. Once it is OK to transmit by checking the CTS flag, the character is popped from the Char Buffer and transmitted.

Train Servers

File: `train.c`

Train Server

The Train Server is responsible for handling sensor data from the Train Sensor Reader and queries from the UI Server. It also starts the Train Sensor Reader and Train Command Server

Data Structures The Train Server stores its sensor data into bit flags. The least significant bit represents the first sensor. This scheme allows easier masking:

- `flag & 1<<0` is the first sensor
- `flag & 1<<1` is the second sensor
- `flag & 1<<15` is the 16th sensor

As well, the Train Server stores the last Time the sensor was triggered.

Train Sensor Reader

The Train Sensor Reader task is responsible for sending track sensor commands and reading them from the train controller. It calls the Train Command Server for the data and manipulates the bytes into a easier to handle form. It then sends the values to the Train Server.

Train Command Server

The Train Command Server is responsible for receiving Train Command messages such as `SPEED` and `READ_SENSOR`. It calls `Putc` and `Getc` as required. Passing all train commands through this server is a form of mutual exclusion. It ensures that commands are fully sent to the trains and commands are not mangled by different tasks.

UI Servers

File: `ui.c`, `ansi.c`

UI Server

The UI Server is responsible for drawing the textual user interface. It draws a header, the time since start up, the command prompt, and a table of sensors readings.

The command prompt supports up to 80 characters. Once this limit is reached, no input will be accepted and displayed. It supports backspace. Pressing the Enter key will execute the command and a response will be displayed under the command prompt.

When a sensor is triggered, the UI Server will display an X on the table. Since the UI Server does not update quickly, a quickly activated and deactivated sensor may not display.

UI Timer

The UI Timer is responsible for sending a message to the UI Server. The timer tells the UI to update the clock on the screen.

UI Keyboard Input Task

The UI Keyboard Input task is responsible for calling `Getc` and sending the character to the UI Server.

Performance

In this deliverable we have made several changes that significantly improve the performance of our kernel:

1) Time slicing
2) Constant time memory allocation
3) Constant time unblocking of tasks on events.
4) Works with all gcc optimization levels.

Time slicing now occurs each time a timer interrupt fires. This allows a slow running user task to be preempted on a timer interrupt, and we can then schedule the notifier immediately so that any high priority user tasks can be unblocked quicker. This is especially important since it means we no longer have to worry about worst case execution time of low priority processes that may mistakenly avoid calling the Pass function.

Constant time memory allocation is now used instead of the linear time memory allocation that was used before. The implementation details of this are described in the data structures section.

Constant time unblocking of tasks has been added by adhering to the convention that only one task can be blocked on a particular event at a time. This removes the necessity to iterate through all tasks checking for their state, and unblocking them if they are blocked on the event being triggered. This update saved as much as 540us on interrupts that involved unblocking tasks.

Finally, our kernel has been updated to work in all compilation levels. Our O3 version runs about twice as fast as the O0 version.

Source Code

The source code is located at `/u4/chfoo/cs452/group/k4-submit/io/kernel4`. It can be compiled by running `make`.

Source code MD5 hashes:

```
linux032:~/cs452/group/k4-submit/io/kernel4> md5sum */*.* *.*
50ef0e1e3c71ab1e795fc3d39f75ef9d  include/bwio.h
9af226f127c1fd759530cd45236c37b8  include/ts7200.h
79cb95e061329765d7dd25c84bf70456  ansi.c
f1288936a928a94bd2170d746e1326a4  ansi.h
b8c8b5fafcd1fd43beaeee7da1e5550f  buffer.c
04c39523dd006155ba353fb3ba1dddfb  buffer.h
b5fcae4a1fde32dcd00b1895c3605961  clock.c
e458870bb98bf9740555b34b15550715  clock.h
fb923f79a79d5e46207578e84e5e695e  kern.c
34599bdbc904f30c66674b59d3f4e2b4  kern.elf
4e3dfc92cbd22b1989e190a26517d536  kernel_irq.c
71f92a7be8c22d1cb5d9cc15be002aa3  kernel_irq.h
5313c05e6242631f379b5141ebca4f5f  kernel_state.h
d41d8cd98f00b204e9800998ecf8427e  kern.h
5439df921ac46fd07959e43125fefa91  memory.c
b16265e8b0bfe3a510b3a25e05b8674a  memory.h
adcff2244ac92050360eacd7ab4f5dd9  message.c
85549de410590cc5033d8bf7399cf504  message.h
a90ee4bc4ce7f46d54b6363d8c8e4ba6  nameserver.c
f9335969b8c71be878a915c26e7a606c  nameserver.h
ca6745d94663494b718c48da853f30a9  notifier.c
f5af74c5dd1325eccdd70e4903640da2  notifier.h
78a32a3a80cad8a4cc40de1ce18fbe29  orex.ld
269c595b95aef41bc8c5931ce1e3c1cf  private_kernel_interface.c
2771aa5221a2b30980916a3ef55b9446  private_kernel_interface.h
080185c0fc743fa77108f3c5ee8e3baa  public_kernel_interface.c
681b789f0146b2589daae803f968ed2c  public_kernel_interface.h
cbdfb94443bcf8e298edc0cf3101b5d1  queue.c
c1157ac2859cc277d1df431dc85502d3  queue.h
cd7239008bf3fc8474819d9183b0cc0f  random.c
7b31c57ff692317d816c839156382596  random.h
9a43cb84e02f05d5b8ffca6e500d98b0  readme.rst
eb5a60f060d101d2536e96298aab4112  readme.tex
9806864a4bf5766bb12b1bd9c1467121  robio.c
0db55f361cec927bba27cdec6b5c2e55  robio.h
545aa195d4f3e0e844edac05715e094d  rps.c
222f8edffbfde11ee553b7561f4c10a8  rps.h
42432905121b8fe5715dcf1ec5367196  scheduler.c
606f31e642dd29602786c86d563b4470  scheduler.h
ecb99ee78c86924b6d54321792e68006  swi_kernel_interface.s
bd8b74d610f4d36af2ac42141d19e9a1  task_descriptor.c
7a03d71b4ace688192d79fcd44fd7420  task_descriptor.h
d09cc801a5f44195d473495a9dd0d78b  tasks.c
0d3699b1a8224eb6995bb042834f66b5  tasks.h
4bf41d22949ce27f1f2d7e5f32a0570f  train.c
6b8ab8c41f1c384a8dc0556e87b8651b  train.h
19f2f37c12c3cec419176823b30c1304  uart.c
205a1d49a746f5d83f3a6e0d5c981492  uart.h
214a8b02f9d8b6c9212685d493867b38  ui.c
b298d8ac96488af2ef904fbeac9d2abf  ui.h
```

```
5b609bdd0235c3858e16c053b8e53bfd  va_list_def.h
```

Elf MD5 hash:

```
linux032:~> md5sum /u/cs452/tftp/ARM/relder-chfoo/k4-submit/kern.elf
34599bdbc904f30c66674b59d3f4e2b4  /u/cs452/tftp/ARM/relder-chfoo/k4-submit/kern.elf
```

Git sha1 hash: `6cd3138e4c4808f1fbfe1790264f038a8cc67110`