CS 452 K2

Names: Robert Elder, Christopher Foo
ID #: 20335246, 20309244
Userids: relder, chfoo
Date due: May 31, 2013

Running

The executable is located at `/u/cs452/tftp/ARM/relder-chfoo/k2-submit/kern.elf`.

It is executed using the regular commands:

```
load -b 0x00218000 -h 10.15.167.4 ARM/relder-chfoo/k2-
submit/kern.elf
go
```

Description

Kernel

Scheduling

- The blocked states are now used. They corresponds to those explained in the lecture:
  - `RECEIVE_BLOCKED`
  - `SEND_BLOCKED`
  - `REPLY_BLOCKED`
- Tasks that blocked are simply rescheduled by re-queueing it.

System Calls

- Added support for 5th argument.

The system calls are not entirely complete according specifications and are noted below.

`Send` The function behaves as expected except the error `-3` code is not implemented.

`Receive` The function behaves as expected. The size of the message sent would be typically 100 bytes since it is the convention used in the kernel.

`Reply` The function behaves as expected except error codes `-3` and `-4` are not implemented.

`RegisterAs` The function behaves as expected. It uses the `NameServerMessage` structure and sends the message. `0` is always returned because the Task ID is hard-coded and the call should never send to the wrong task.

`WhoIs` The function behaves as expected. As noted in `RegisterAs`, we either return a Task ID or 0.

Name Server

File: `nameserver.c`

The name server uses a 2D `char` array. The maximum name is limited arbitrary to 100 letters including the null terminator. 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.

- 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`.

Rock Paper Scissors

File: `rps.h`

There are two server-client structures: `RPSServer` and `RPSClient`. `RPSMessage` is the message structure.

The kernel starts the `RPSServer` and 3 `RPSClient`. All of them are executed at `NORMAL` priority as to allow us to catch synchronization bugs.

RPSServer

1. Registers with the name server
2. `Receive` and processes message with sign up, quit, and play types.
3. Outputs if there is only 1 or less items in the sign up queue.

Sign Up

1. Add the task ID to the sign up queue.
2. Mark it as signed up on the sign up array.
3. `Reply` with a `SIGN_UP_OK` type message.

Quit:

1. Mark it as quit (0) on the sign up array.
2. `Reply` with a `GOODBYE` type message.

Player Selection Players are selected by going through the sign up queue. Players are not requeued if they have quit.

Play

1. Get the player's choice by matching the Task IDs
2. Once we have two choices, we can `Reply` with the outcome.
3. `Reply` a `NEGATIVE_ACK` message type if the client cannot play yet.

RPSClient

Each client wants to play 5 times.

1. Look up the task ID of the `RPSServer`.
2. Sign up
3. Decide their choice and `Send` it.
4. If a `NEGATIVE_ACK` is received, try again. This is a simple polling method that is robust.
   - This could be improved/avoided by having the `RPSServer` notify or `Send` to the client to let it know it is ready.
5. Read the message and print out results.
6. Waits for keyboard input

Algorithms and Data structures

Queue

File: `queue.c`
A change was made so that it returns the current item count.

Task Descriptor (TD)

File: `task_descriptor.c`
The TD was modified so it contains message pointer such as the address of TID for `Receive`

Kernel State

File: `kernel_state.h`
An array of `KernelMessage` and its related variables was added to the `KernelState`

Memory Operations

File: `memory.c`
Functions were added so that strings (potentially binary data) could be copied. They are simple and copy one `char` at a time.
Possible improvements: Instructions could be written in assembly that make use of the block copying mode.

Kernel Message

File: `message.h`
`KernelMessage` is a `struct` that contains the message string. It contains origin and destination IDs and pointers.

Messages

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`. The message size is fixed arbitrary 100 for consistency.

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.

Source Code

The source code is located at `/u4/chfoo/cs452/group/k2-submit/io/kernel2`. It can be compiled by running `make`.

Source code MD5 hashes:

```
chfoo@linux032:~/cs452/group/k2-
submit/io/kernel2$ md5sum *.* */*.*
da5c58f5a70790d853646f4a76f4c540  buffer.c
1f9a730c5017ddd24e18523d27dc471e  buffer.h
e270fd64ae08a0317d37fadedd24cabb  kern.c
b00a171e052d7c818750f58a3bdcf27c  kernel_control_flow.pdf
707a17591d47c33efb8f3c4dc5dc7b72  kern.elf
52dd3c8bac8b93e7bc9024ca3e56b00a  kernel_stack.pdf
84a5537b040cccb5d8ef47b4915018e8  kernel_state.h
d41d8cd98f00b204e9800998ecf8427e  kern.h
7a7803ae8e9733a3bb9e27f07a2e0855  memory.c
c782fb4d47461c3c448d646ff43271c3  memory.h
adcff2244ac92050360eacd7ab4f5dd9  message.c
f358c0a299df10810a2247bedb04571a  message.h
19abaa01e22b14b4e7aa51001c042eb6  nameserver.c
53f58016672e3a2a02c3a5aee480ec50  nameserver.h
4aa618b9753c5292e5d9e5c95d297f10  orex.ld
759f729becf9837439f4acbf14fd029c  private_kernel_-
interface.c
bef673553ff2738e5355c9c0d8c9fb77  private_kernel_-
interface.h
0656c0cea9a29f56d2db883a50ef0884  public_kernel_-
interface.c
c5f6e83ffce706a065b591a86c5e0abb  public_kernel_-
interface.h
8bb4ea6e2e00ae9c9bad30f682dbe9af  queue.c
8c282e71bf30800f9d749685dba46de5  queue.h
91fbdbffeb090806d35dc54cb2e0627a  random.c
7b31c57ff692317d816c839156382596  random.h
eb1310d4951124acd3fd563cc989e687  readme.rst
3bf0193cced01283304b36167df3594a  robio.c
6fe98c156b7860cd10e5c8e7c7ef39ef  robio.h
```

```
0e7b69960f90c7e25b85f4cf79fb1edd  rps.c
f7e4d15f1a3bf56aa3f5b59c4453ef1d  rps.h
d55a63fb8522de9736cc3833ff0a9025  swi_kernel_interface.s
d22a28c9457c285a63ea0ff7091b5f6b  task_descriptor.c
b4bea5f2aa97403c3ad8c67de0ffb76c  task_descriptor.h
1d21eae6d91007bfd0d6dd6e35266aa9  tasks.c
e1bc57af359db93c3982f8c0af896fcc  tasks.h
50ef0e1e3c71ab1e795fc3d39f75ef9d  include/bwio.h
9af226f127c1fd759530cd45236c37b8  include/ts7200.h
```

Elf MD5 hash:

```
chfoo@linux032:/u/cs452/tftp/ARM/relder-chfoo/k2-
submit$ md5sum kern.elf
707a17591d47c33efb8f3c4dc5dc7b72  kern.elf
```

Git sha1 hash: `2b864d846dce67bae5bf4ad86ac0650588f5f85f`

Output

The executable prints:

- Task creation
- Who wants to sign up
- What clients choose
- The chosen player task IDs
- The result of the game
- The client's reasoning for winning/losing
- When a client quits

Note that a keyboard response is needed when the *client* gets the result. This allows us to see what the *server* decided before continuing execution. So two presses of the keyboard are needed for each round.

Unimplemented Code

Because the user tasks are not finished, the program will end up at an assertion failure. The last task remaining will not be able to play because it will continue to poll and wait.

If more time was available, we would have implemented:

- Detect when 1 client is remaining and `Send` a `SHUTDOWN` message type to the client.
- The client will stop and exit.
- Stop the `PRSServer` and the name server
- Exit cleanly.

Performance Measurement

TODO