

CS 452 K2

Names: Robert Elder, Christopher Foo
ID #: 20335246, 20309244
Userids: relder, chfoo
Date due: May 31, 2013
Extension date: Jun 3, 2013

Running

The executable is located at `/u/cs452/tftp/ARM/relder-chfoo/k2a-submit/kern.elf`.
It is executed using the regular commands:

```
load -b 0x00218000 -h 10.15.167.4 ARM/relder-chfoo/k2a-  
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 are blocked are simply rescheduled by re-queueing it. Efficiency could be improved by having a separate blocked queue.

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, however, errors `-3 -4` include an assert that will aid in debugging.

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.
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.

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.
3. The server is marked for shutdown.

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 WAIT message type if the client cannot play yet.
4. Reply a SHUTDOWN message if any one signed up has quit.
 - This design decision was made so that the shutdown case is well defined.

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_WAIT 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/k2a-submit/io/kernel2. It can be compiled by running make.

Source code MD5 hashes:

```
chfoo@nettop38:~/cs452/group/k2a-submit/io/kernel2$ md5sum */**.* **.*
50ef0e1e3c71ab1e795fc3d39f75ef9d  include/bwio.h
9af226f127c1fd759530cd45236c37b8  include/ts7200.h
da5c58f5a70790d853646f4a76f4c540  buffer.c
1f9a730c5017ddd24e18523d27dc471e  buffer.h
e270fd64ae08a0317d37fadedd24cabb  kern.c
147be2be54ef2d9f4cad65a5794e7db7  kern.elf
d41d8cd98f00b204e9800998ecf8427e  kern.h
b00a171e052d7c818750f58a3bdcf27c  kernel_control_flow.pdf
52dd3c8bac8b93e7bc9024ca3e56b00a  kernel_stack.pdf
84a5537b040cccb5d8ef47b4915018e8  kernel_state.h
7a7803ae8e9733a3bb9e27f07a2e0855  memory.c
c782fb4d47461c3c448d646ff43271c3  memory.h
adcff2244ac92050360eacd7ab4f5dd9  message.c
04eded8ccd08bc5f73ff2eaf0274f076  message.h
e80ae585145cca0cc59cc19c9dcf9b7b  nameserver.c
53f58016672e3a2a02c3a5aee480ec50  nameserver.h
4aa618b9753c5292e5d9e5c95d297f10  orex.ld
e208b867b9bac6cf5c81c6d73f9b3f8d  private_kernel_interface.c
bef673553ff2738e5355c9c0d8c9fb77  private_kernel_interface.h
0656c0cea9a29f56d2db883a50ef0884  public_kernel_interface.c
774ba0cc7254fa6203bc796b85d716cb  public_kernel_interface.h
8bb4ea6e2e00ae9c9bad30f682dbe9af  queue.c
8c282e71bf30800f9d749685dba46de5  queue.h
91fbdbffeb090806d35dc54cb2e0627a  random.c
```

7b31c57ff692317d816c839156382596	random.h
5346b105cb8defb34e3aa6c4a853ff5c	readme.pdf
e18b4677ae9866d43b7711a219042fff	readme.rst
a03dcb23183838a3968c9e4cf1fa1183	readme.tex
3bf0193cced01283304b36167df3594a	robio.c
6fe98c156b7860cd10e5c8e7c7ef39ef	robio.h
f207d682e54e737639f45f3a17078b6b	rps.c
89e7636353a31fd35b73a82ee49a20e0	rps.h
d55a63fb8522de9736cc3833ff0a9025	swi_kernel_interface.s
d22a28c9457c285a63ea0ff7091b5f6b	task_descriptor.c
db54248f4493240552b83a78d7656822	task_descriptor.h
1d21eae6d91007bfd0d6dd6e35266aa9	tasks.c
e1bc57af359db93c3982f8c0af896fcc	tasks.h
952d806f62f1ef2d28b508485a7545c2	timing.ods

Elf MD5 hash:

```
chfoo@nettop38:/u/cs452/tftp/ARM/relder-chfoo/k2a-
submit$ md5sum kern.elf
147be2be54ef2d9f4cad65a5794e7db7  kern.elf
```

Git sha1 hash: 7f890df902b0c688f32fb0488ffe15032964f1c1

Output

The executable prints:

- Task creation
- Who wants to play
- Who wants to sign up
- What clients choose
- The pair the server has decided to play
- The result of the game
- The client's reasoning for winning/losing
- When a client quits

In Summary:

1. Server and Clients start up
2. Clients request to play
3. Games are played
4. One client quits and the system tries to shutdown
5. Clean exit

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.

Performance Measurement

Message length	Caches	Send before Reply	Optimization	Group	Time (us)
4	off	yes	off	2	549
64	off	yes	off	2	2088
4	on	yes	off	2	broken
64	on	yes	off	2	broken
4	off	no	off	2	547
64	off	no	off	2	2090
4	on	no	off	2	broken
64	on	no	off	2	broken
4	off	yes	on	2	broken
64	off	yes	on	2	broken
4	on	yes	on	2	broken
64	on	yes	on	2	broken
4	off	no	on	2	broken
64	off	no	on	2	broken
4	on	no	on	2	broken
64	on	no	on	2	broken

The time spent in the code is likely in the string copy functions. Since they are unoptimized and only copy one `char` at a time, it is very expensive and operates in linear time.