CS 452 Kernel 2

Brian Forbes & Daniel McIntosh
20617899 & 20632796

May 29, 2018

1 Building, Running, and Usage

The repository for the code can be found at `https://git.uwaterloo.ca/baforbes/cs452-kernel/tree/k2`. To build the executable, run the following command from the root directory:

```
make
```

To build this document, run the following command from the root directory:

```
make docs
```

The executable will be built to `<repo_root>/bin/kernel.elf`

Once run, the program will initialize the kernel, and then create the first user task. This task will then create 3 other tasks: the Rock Paper Scissors (RPS) server, and 2 RPS clients. The clients will play 10 RPS games against each other, awaiting user input when they recieve replies from the RPS server so the game results are displayed.

2 Program Description

Note that each referenced file corresponds to a header file of the same name in the `include` folder (for example, `src/kernel.c` to `include/kernel.h`)

3 Program Structure

3.1 Message Passing

Each task struct now has a recieve queue (implemented via head and tail pointers). The handling for message passing syscalls is in `sys_handler.c`.

3.1.1 Send

When a task sends a message, it checks if the reciever is already send blocked. If so, it checks the length of the message, and compares it to the length the reciever expects. If these do not match, the return value will be set to an error. It then copies the message into the reciever struct, up to the minimum of the two lengths. If the reciever is not send blocked, the sender is added to the send queue of the receiver.

3.1.2 Receive

If the send queue of the task is not empty, the message is immediately copied into the reciever (including the length check as above), and the sender is moved to the reply blocked state. If the send queue is empty, the reciever moves into the send blocked state.

3.1.3 Reply

After checks to ensure that the task exists and is reply blocked, as well as the message length check, the message is copied into the reply message pointer, and the reply blocked task is moved into the ready state.

3.2 Nameserver

The nameserver implementation is in `name.c`. On startup, the nameserver stores its TID in a global int to ensure that all tasks know it. The nameserver stores names in a simple hashtable with linear probing (discussed in the Data Structures section). After initializing its storage struct, the nameserver enters a forever loop. On each iteration of the loop, it recieves a message, and executes the desired request. For WhoIs requests, the name is looked up in the hashtable. For RegisterAs, the name is added to the hash table.

3.3 RPS

The RPS implementation is in `rps.c`.

3.3.1 Server

The RPS server stores data in two structs (games, plays) and one int (unpaired). We chose to store only one unpaired task at a time, because there can never be two unpaired tasks at once (if there were, they ought to have been paired together already). On initialization, the data structures are initialized, and the server registers to the name server. It then enters a forever loop. On each iteration of the loop, first recieves a message. Thenm it edecutes the desired request. For signup requests: If there is no waiting task, it sets this task to the unpaired task. If there is a waiting task, it pairs the two tasks (by setting their values in the games array to each other) and replies to each of them.

For play requests: The opponent is determined via the value in the games array. If the opponent has already played their move (determined by the value in the plays array), it compares the two moves, and returns the victory (or quit) status to each task. If the opponent has not yet played, it sets the value in the plays array to the desired move.

Quit requests are implemented as play requests with a specific move value.

3.3.2 Client

After finding the RPS server TID and signing up for a game, clients play 10 moves. Each move is chosen via taking the low bits of the 40-bit debug clock modulo 3 - effectively randomly. After 10 games, they quit, and then exit.

3.4 Changes from Kernel 1

The architecture from Kernel 1 has not been significantly changed, but various optimizations were added. For example, unneccesary instructions were removed from `asm/activate.s`, the loop in `memcpy` was unrolled using duff's device, more information was added to aid GCC's optimization, and sequential stack pushes/pops of individual registers were merged into `stmdb` and `ldmia` instructions.

3.5 New Data Structures

The only major new data structure used is the hash table in the name server. We chose a hash table with linear probing because it was relatively small (only a factor 1.3 time larger than an array of the size of the maximum number of names we expected to store), easy to implement, and was a decent performance increase in the average case. We recognized that most stores and lookups only happened on kernel startup, but still saw benefit to these performance increases.

The hashtable was implemented as an array, with linear probing. To insert, the item is either put into the slot of the hash of the key, or the next empty slot below that (cyclically). Similarly, to look up a key, the slot for the hash of that key is queried. If it is not the correct item, the hashtable is probed linearly downwards (cyclically) until the key is found. In both operations, the operation fails if the hashtable is full.

As a hash function, we chose the djb2 function (from `http://www.cse.yorku.ca/~oz/hash.html`). This is because it is extremely simple, fast, and has good distribution of hashes.

4 Explanation of Program Output

The output of the program should be:

```
Created RPS Server: 2
Created RPS Client 1: 3
```

```
Created RPS Client 2: 4
```

After this, the output will be 20 lines of the form:
¨$(id): I played $(play), Result: $(result)¸ detailing the results of 10 games. Each line represents the output from one of the RPS tasks. It contains the ID of the task, the move that task chose, and the result of that task.

Instead of having the RPS server pause after every game, we chose to have the RPS clients pause. This is to ensure that not only were moves being sent correctly and results were being computed correctly, but the clients were also recieving the correct reply message.

5 Timing Data

This data has also been emailed in the requested format, but it is added here for posterity.

Message length	Caches	Send before Reply	Optimization	Time
4	off	yes	off	308
64	off	yes	off	641
4	on	yes	off	22
64	on	yes	off	44
4	off	no	off	298
64	off	no	off	635
4	on	no	off	22
64	on	no	off	44
4	off	yes	on	138
64	off	yes	on	242
4	on	yes	on	10
64	on	yes	on	16
4	off	no	on	137
64	off	no	on	241
4	on	no	on	10
64	on	no	on	16

6 File Hashes and Commit hash

Output of `sha1sum src/* include/* asm/*`

Hash	File
97231c9fde9f0d4155cc6733c81473fe43145871	src/circlebuffer.c
873aa16244be9feb8696a41bd6860e380c738773	src/clock.c
8010e18b2f848f65ca49fd70ad0250822f01d686	src/comio.c
0796c2a8d3aa58db79c116ee2fe0f7c23269396a	src/hashtable.c
ede430e48ae75191c3981e59702be57ac6e365d2	src/kernel.c
240681f6e6a4ff7e6c21913f475c010f8fa94836	src/minheap.c
93c61448622b997a5316f3630371f96eef74637c	src/msg_metrics.c
818c1cb7dfba9b64774d250d6f43599ff823ea48	src/name.c
6bbf2f4b05ea5200dfde4529268618ff0dc4ad10	src/rps.c
9ca83aed60b535be101ea973aa8e321dfa260577	src/sys_handler.c
8ab24aed7959069820c1b78ae5127d3296cf7230	src/syscall.c
448dbc529557a03523524015b8771d7f91717743	src/tasks.c
a4ba75a0a842651ec25dde2fb3ada423a9394930	src/track_data.c
c27eea7678367b8e67648851035b7f216581d592	src/util.c
304c87bfa20cc259217a5b89218c22ace9782fd7	include/bwio.h
fbd69d26075f67eaf1ae17d09be5d26b202a4382	include/circlebuffer.h
1f42302fdde007ec9d73a0cc37d80140c4531fe6	include/clock.h
d871edc6c39e07102bb5fbfe10cbc657bf64c4e1	include/comio.h
92d08a1c7b26241036cda9d946bf78ad278282b8	include/debug.h
e400d1667b2aa52683dc8980709635beed86b455	include/elem.h
c41c6a180ea06c5e61f7e9eb3bbe858b98506309	include/err.h
cbb7faeff032fb3703540cac1a152a310a08565a	include/hashtable.h
e8eb6c8c7327bc68be8373102e7f4b19913a26a7	include/kernel.h
ac8eaf2964566e3bcda903e3dcf27b3abe49dfb5	include/message.h
b75bead1e2e343f8b80db372925d27583c30005b	include/minheap.h
723d801a3c99cd243bc9e12b15c319d8ae16da6a	include/msg_metrics.h
e52482b9f0f9c45c6d0e7b6644c543c65238e5b3	include/name.h
82a1cc6e0e1f232f0e56eec87dc6e2173ca1fb76	include/rps.h
b988d142755c58c9321f98c4605887b0caeab2c6	include/sys_handler.h
371c2704f5ba742ec5d1fcc5eb876036a1c10a1e	include/syscall.h
dd749b1bc62069f06791465613ceece826824869	include/tasks.h
464a4766793704c28e49960fb3b78283124fecd6	include/track_data.h
f5c6374f35eda215237b9692e2f5c602cb405d08	include/track_node.h
d701602b5ceeff844853f2360d5a7fa3d35ade33	include/ts7200.h
78eb6344868e03101c348ffb0debd079530286e8	include/util.h
020477a7be0fdc05a32a2f918a46f4a9062750f4	asm/activate.s