CS 452 Assignment 2

Felix Fung (f2fung)
Dusan Zelembaba (dzelemba)

May 31, 2013

1 How To Run

```
> load -b 0x00218000 -h 10.15.167.4 "ARM/cs452_10/dzelemba_f2funq_a2.elf"
> go
```

2 Submitted Files

Files listed here can be found under `/u1/dzelembaba/cs452/frozen_a2`

2.1 md5sums

```
6e054a9754c1a726af237bd4e60a8781  ./Makefile
9ba6ec651d0fdbfe434e3664b69623c3  ./all_tests.h
b8f6809658948d4086826b1e3c0241c3  ./context_switch.h
cef010d3ef08df27f78a2298e86e9a81  ./context_switch.s
d4a0109b4748bb3ec5e3b19b346c23da  ./debug.c
40e13142b4df44ef2164c04dc072130a  ./debug.h
e7134b2fe5ce58f46704dd38879a1439  ./kernel.c
664366bb7fe0a090ebdfe1b3009a14ba  ./kernel.h
f8775577c797a37a4d3f5e7874dff684  ./main.c
81330c063f3510d8b4a46fc3ee6bed6a  ./messenger.c
d9fcdc742e0118ccfc12d5f5f5f3c043  ./messenger.h
4fc206eaaddecc78bcc8d8a16273bfb2  ./nameserver.c
b2047a8021138c14fd7eea61641669d5  ./nameserver.h
1dc118c000601dc8accbed9bf54a2076  ./orex.ld
d4112505ecd99a3c4745fde1735a0c35  ./queue.c
175206a0ac5e05e89f90855d013a58c9  ./queue.h
bf7a472ac76d62c0c40525aaf3ad57a6  ./rps_server.c
f42681e19e3dc50960b701be52a53546  ./rps_server.h
71530a536c54208f0ddf5d1ef5c85d90  ./run_tests.c
0e390e153530b05118c87063a5dd3120  ./run_tests.h
b25b2920df8162967f50304d43393db1  ./scheduler.c
e4da5a09534a9882c0dadda80816cf59  ./scheduler.h
```

```
26189fb402a0f5afa5edc24627503f3a  ./stdlib.c
d438ff4163f9bc5c25ccb1a9aa8f2e48  ./stdlib.h
792ad97b51eece81a637452da6056674  ./strings.c
7a1b1692e08413e0632172e8bab8cd11  ./strings.h
97e3c872d2f52044dda678c85548fa7b  ./syscall.c
3889853c5bab245cb2f5bfdc3948791a  ./syscall.h
398da41ad3ed1ef20cf2980182b4fc68  ./task.c
60b875a81187f49bc949b8d7044502e1  ./task.h
f6a07049571b3c0965a353a8e59acfbc  ./test_helpers.c
9570cd2291eb9823e5fb0c501aeea2b3  ./test_helpers.h
db8bae8ff95a9ccc357a4e26ad12dbab  ./tests/assignment_1_test.c
cbf5541b0a2af623339ea90891c9525e  ./tests/basic_test.c
d492a73d943d7da5b66671829e8197d7  ./tests/message_passing_test.c
2769176d92c2eb972f29957500c870c8  ./tests/multiple_priorities_test.c
4096c48f657c198ab7f5c0c569a8e1f7  ./tests/nameserver_test.c
14a754cca9c10a5716cb52265301b14f  ./tests/rps_server_test.c
3ae955c9788cce1d42200c3c645cde07  ./tests/srr_speed_test.c
b82e6c2633130a4dd53eb1eb32a39ac6  ./tests/syscall_speed_test.c
6ec9a04bcbec1d7a04f714f258043e61  ./tests/task_creation_errors_test.c
5a8c8a8d874df99d7bbe29d48de86801  ./timer.c
6e6b166bbbda7d6b491c1ee0582c806e  ./timer.h
7d5ccf5c18cc3cbfb89907ddd4512531  ./main.map
```

2.2 Header Files

`context_switch.h` - Function definitions for compiler to use our assembly functions.
`debug.h` - Macros wrapping debugging functions that can compile away.
`messenger.h` - Function definitions for messenger.
`nameserver.h` - Function definitions for starting the nameserver.
`queue.h` - Function definitions for our queue implementation.
`rps_server.h` - Functions definitions for starting and using the rock, paper, scissors server.
`scheduler.h` - Function definitions for scheduler.
`string.h` - Functions definitions for string library.
`stdlib.h` - Function definitions for commonly used functions.
`syscall.h` - Function definitions for syscalls.
`task.h` - Contains Task structure and function definitions.
`timer.h` - Function definitions for timer.

2.3 Source Files

`context_switch.s` - ARM assembly used to switch in and out of tasks and kernel mode.
`debug.c` - Debugging facilities.

`kernel.c` - The infinite kernel loop. Interrupts are processed here and tasks and scheduled.
`main.c` - Beginning of execution, described below.
`messenger.c` - Coordinates message passing.
`Makefile` - Our makefile.
`nameserver.c` - Implementation of the nameserver, and WhoIs/RegisterAs syscalls.
`queue.c` - A simple queue implementation.
`rps_server.c` - Implementation of the rock, paper, scissors server and wrappers for Send that clients can use to talk to the rps server..
`scheduler.c` - Our scheduler.
`stdlib.c` - Common functions. Not actually a standard library.
`strings.c` - String library.
`syscall.c` - System calls.
`task.c` - Creating and managing tasks.
`timer.c` - Timing functions.

2.4 Test Files

`tests/basic_test.c` - This is a basic test. (This is only run for debugging)
`tests/message_passing_test.c` - Tests all aspects of Send, Receive, Relpy.
`tests/multiple_priorities_test.c` - Tests the correct scheduling of multiple tasks with different priorities.
`tests/nameserver_test.c` - Tests WhoIs & RegisterAs.
`tests/rps_server_test.c` - Implements a set of clients for the nameserver.
`tests/srr_speed_test.c` - The test that is run to analyze our performance.
`tests/syscall_speed_test.c` - Test the time required to execute various sys calls.
`tests/task_creation_errors_test.c` - Tests error return values for task creation.

`run_tests.h` - Header for run_tests.c.
`run_tests.c` - Calls all our tests.
`test_helpers.h` - Header for test_helpers.c. Different types of asserts and such.
`test_helpers.c` - Test helpers. Asserts and other things.

3 Program Description

3.1 Message Passing

To make better use of the cache, we've made a space-saving statically allocated data structure for message passing. We create a fixed-size array of INBOXes, MAX_TASKS in number. Each INBOX contains fields we reuse to contain EITHER a sent message and reply address (params of Send()), or the receiving address (params of Receive()). There are no collisions because Send() and

Receive() cannot be concurrent for a single task. Each INBOX contains a NEXT pointer, which may point to another INBOX or NULL allowing INBOXes to be chained together in a linked-list fashion. Then for MAX_TASKS, we also store 'head' and 'tail' pointers to INBOXes to represent the receive queue for each task in constant space relative to the length of the receive queue.

In summary, when blocking, tasks will write their addresses to their own INBOX, and it'll be chained into the receiving task's linked-list. By reusing INBOXes for future messages, we never allocate more memory.

3.2 Nameserver

The nameserver is a user space task that is started by the kernel. It uses a simple queue of strings, int pairs to hold each registration it is given. This results in an $O(n)$ (in number of registrations) runtime for both RegisterAs and WhoIs. This was considered acceptable because n will usually be small and these syscalls tend to be called during initialization, where performane isn't as critical.

The nameserver's task id is stored in a static variable that is visible only to the nameserver. The nameserver exposes a function, `get_nameserver_tid()` that provides access to the nameserver's tid. This way we can ensure that the nameserver only writes to this variable once during startup to avoid race conditions.

If a non-existent registration is passed to WhoIs, it will return -1. This was the simplest implementation for now and if a blocking version of WhoIs is needed in the future a seperate syscall, WhoIs_Blck, will be writted to provide synchronization methods for tasks needing to wait on other tasks to start.

3.3 Rock, Paper, Scissors Server

The Rock Paper Scissors (RPS) server uses 3 data structures to maintain the state of all the games. First it uses a single variable, `waiting_to_play` to indicate the task id of a task that has called signup, but hasn't been matched yet. Then, it uses two arrays, `players` and `moves`, of size `MAX_TASKS` to control game state. Index i of `players` represents who task i is playing against and index i of `moves` represents task i's move. Both of these ararys use -1 to represent no game in progress or no move has been played.

All methods for talking to the nameserver are implemented in `rps_server.c`, so a client need not know any of the details of how the RPS server expects Send messages to be formatted. `rps_server_test.c` provides an implementation of a client. The RPS server pauses at the end of each round and outputs the result of that round and asks to enter a charcter to proceed to the next round.

The game's were all run at the same priority level so that they would run concurrently and better test the RPS server. The output from the game is

shown in RPS Game Output.

3.4 Timing

All timing code is being run in user-mode. When testing the send-receive-reply speeds, we only measure the full round-trip. In the case of send before receive, we begin timing before being send blocked to when the task is active again. (Implying that receive and reply have been called from the other task). We ensure there are only two tasks and so our sender task goes directly from ready to active without further delay. In the case of receive before send, we begin timing before being receive blocked and after we return from reply. Based on our implementation, these times all contain the time required to copy messages across buffers.

Our memcpy implementation is naive and I believe a lot of our time is being spent here. Increasing message size adds a significant performance hit as seen in our performance results.

4 RPS Game Output

```
Rock Paper Scissor Server Starting...

Player 4 Starting...
Player 5 Starting...
Player 6 Starting...
Player 7 Starting...
Player 4 Acquired RPS Tid...
Player 5 Acquired RPS Tid...
Player 6 Acquired RPS Tid...
Player 7 Acquired RPS Tid...
Player 5 Signed Up...
Player 4 Signed Up...

Round Complete
Player 4 played: ROCK, Result: TIE
Player 5 played: ROCK, Result: TIE
Enter character to continue to next round...

Player 7 Signed Up...
Player 6 Signed Up...

Round Complete
Player 6 played: PAPER, Result: LOST
Player 7 played: SCISSORS, Result: WIN
Enter character to continue to next round...
```

```
Round Complete
Player 5 played: SCISSORS, Result: WIN
Player 4 played: PAPER, Result: LOST
Enter character to continue to next round...


Round Complete
Player 7 played: SCISSORS, Result: TIE
Player 6 played: SCISSORS, Result: TIE
Enter character to continue to next round...


Round Complete
Player 4 played: ROCK, Result: WIN
Player 5 played: SCISSORS, Result: LOST
Enter character to continue to next round...


Round Complete
Player 6 played: ROCK, Result: LOST
Player 7 played: PAPER, Result: WIN
Enter character to continue to next round...


Round Complete
Player 5 played: PAPER, Result: LOST
Player 4 played: SCISSORS, Result: WIN
Enter character to continue to next round...


Round Complete
Player 7 played: SCISSORS, Result: WIN
Player 6 played: PAPER, Result: LOST
Enter character to continue to next round...


Round Complete
Player 4 played: PAPER, Result: WIN
Player 5 played: ROCK, Result: LOST
Enter character to continue to next round...


Round Complete
Player 6 played: SCISSORS, Result: LOST
Player 7 played: ROCK, Result: WIN
```

```
Enter character to continue to next round...

Player 4 Done...
Player 5 Done...
Player 6 Done...
Player 7 Done...
Player 4 Starting...
Player 6 Starting...
Player 4 Acquired RPS Tid...
Player 6 Acquired RPS Tid...
Player 6 Signed Up...
Player 4 Signed Up...

Round Complete
Player 4 played: ROCK, Result: LOST
Player 6 played: PAPER, Result: WIN
Enter character to continue to next round...


Round Complete
Player 6 played: SCISSORS, Result: WIN
Player 4 played: PAPER, Result: LOST
Enter character to continue to next round...


Round Complete
Player 4 played: ROCK, Result: TIE
Player 6 played: ROCK, Result: TIE
Enter character to continue to next round...


Round Complete
Player 6 played: PAPER, Result: LOST
Player 4 played: SCISSORS, Result: WIN
Enter character to continue to next round...


Round Complete
Player 4 played: PAPER, Result: LOST
Player 6 played: SCISSORS, Result: WIN
Enter character to continue to next round...

Player 4 Done...
Player 6 Done...
Main Exiting...
```

```
Program completed with status 0
```

This game involves 4 clients each playing a predetermined set of moves. The player number is their task id. Player's 4 and 6 signup for another game after they finish to test that quit properly resets state. The game's are all run at the same priority level, which is why the RPS server alternates playing a round of each game. The game between player 4 and 5 starts first, because they were the first tasks created.