CS 452: Kernel 2

Jason Sun (#20387090) and Shuo Chang (#20378235)

May 28, 2015

This document builds upon Kernel 1 documentation, and only describes the changes.

1 Program Operation

Compiling by running `make`, which also copies the local `kernel.elf` to `/u/cs452/tftp/ARM/j53sun/k2.elf`.

```
> load -b 0x00200000 -h 129.97.167.12 "ARM/j53sun/k2.elf"
> go
```

See also Source Code location section.

2 Kernel Structure

Algorithms and data structures.

2.1 Task Descriptors

Added the following:

- `MessageStatus status`: Task statuses for message passing: `{none, send_block, receive_block, reply_block}`.
- `int *send_id`: the pointer to sender's task id from the `Receive()` call.
- `void *send_buf`: the pointer to sender's message from the `Send()` call.
- `void *recv_buf`: the pointer to receiver's buffer, set when `Receive()` is called first.
- `unsigned int send_len, recv_len`: the length of `send_buf` and `recv_buf`.

2.2 Syscalls

New syscalls are introduced for task communication.

- `int Send( int tid, void *msg, int msglen, void *reply, int replylen )`
- `int Receive( int *tid, void *msg, int msglen )`
- `int Reply( int tid, void *reply, int replylen )`

Added `arg3, arg4, arg5` to Syscall data structure to support a maximum of 5 system call arguments.

2.3 Task Communication

- `handleSend()`, `handleReceive()`, and `handleReply()` are the kernel's system call handlers
- `handleSend()` checks whether the intended receiver is currently receive blocked. If receiver's status is "receive_block", copy the buffer from sender to receiver directly, and set it's status to "reply_block"; else, enqueue to the receiver's send queue then update it's status to "send_block". It also copies sender's reply buffer and reply buffer length into recv_buf, recv_len in sender's task descriptor for later use in `handleReply()`.
- `handleReceive()` checks whether there are senders in receiver's send queue. If there is one, or more than one senders, dequeue the sender, copy the message found in sender's task descriptor, update sender's status to reply_block, and add receiver to the ready queue. If there are no queued senders, set receiver's status to "receive_block" and store the pointer to the buffer and length to the receiver's task descriptor.
- `handleReply()` copies the message from the receiver to the sender. The sender's reply buffer is stored in recv_buffer in it's task descriptor, by `handleSend()`.
- All three functions checks for error conditions specified in the Kernel Description, and sets the return values for `Send()`, `Receive()`, and `Reply()` system calls.

2.4 Memcpy

Not a very optimized implementation for `memcpy()`.

2.5 Nameserver

The nameserver is created during kernel initialization by the first user task. Therefore its task id is entirely deterministic, and is be shared as a constant between tasks.

The nameserver keeps track of tasks in a `{name, task_id entry}`, in table. The table has a static size of 256. Insertion, and lookup, scans the entire table.

Linear scan is used instead of something with better performance because the nameserver is anticipated not to be called after a task initializes, therefore its performance is not important.

2.6 Rock/Paper/Scissor (RPS)

The `rpsUserTask()` is called by the kernel init code to start the nameserver, rps server, and makes rps players. The number of players and the maximum simultaneous games are hard-coded.

- The rps server keeps an array of players, which just consist of the player's tid, what their request was (if any), and who their opponent tid is (if any). Insertion, keeping track of games, and removing players all does a linear scan.
- When a player calls in, the server tries to match another player. If another player had called in before and was waiting, the server then matches them together.
- Server communicates with other tasks if a player had left during their game session.
- The rps player each plays a hard-coded number of games, and uses a PRNG to generate what they choose to play.

The game pauses after each round. Pressing a key would continue.

3 Source Code Location

- Repository location is `gitlab@git.uwaterloo.ca:j53sun/cs452team.git`
- SHA1 hash of commit for submission: `36d4f01ed738c92da9f7858288d255450f2b6a57`

4 Listing of All Files Submitted

All the files within repository.

5 Game Task Priority

- Nameserver runs at the a highest priority.
- RPS server runs with a priority below nameserver.
- RPS client (the player) runs with priority below RPS server.

Being send blocked is better than recieve because it is more responsive. The priority is arranged this way so that players do not need to wait on server, and server do not need to wait on nameserver.

6 Measurements

#	Bytes	Cache	Send First	Group name	O2 flag	Time μs
1	4	off	yes	changsun	off	350.7
2	64	off	yes	changsun	off	849.3
3	4	on	yes	changsun	off	25.3
4	64	on	yes	changsun	off	59.1
5	4	off	no	changsun	off	340.9
6	64	off	no	changsun	off	842.6
7	4	on	no	changsun	off	24.7
8	64	on	no	changsun	off	57.7
9	4	off	yes	changsun	on	171.3
10	64	off	yes	changsun	on	327.7
11	4	on	yes	changsun	on	11.3
12	64	on	yes	changsun	on	19.7
13	4	off	no	changsun	on	167.2
14	64	off	no	changsun	on	323.4
15	4	on	no	changsun	on	11.0
16	64	on	no	changsun	on	19.4

- For 64 byte messages, the majority of time are being spent in `memcpy()`. Since `memcpy()` copies byte-for-byte, it is not very efficient when not optimized because it needs to load and store index and also two pointers, 64 times each. This adds a lot of overheads to message passing.
- For 4 byte messages, the time is split between `memcpy()` and scheduling (enqueue and dequeues). Scheduler can be improved by switching implementation to a circular list (using linear at the moment).

7 Game Task Program Output

```
RedBoot> go
Program completed with status 0
```

Explaination