Kernel 2
CS452 - Spring 2014
Real-Time Programming

Team
Max Chen - mqchen
mqchen@uwaterloo.ca

Ford Peprah - hkpeprah
ford.peprah@uwaterloo.ca

Bill Cowan
University of Waterloo
Due Date: Friday, 30^{th}, May, 2014

Table of Contents

1 Program Description

1.1 Getting the Program

To run the program, one must have read/write access to the source code, as well as the ability to make and run the program. Before attempting to run the pogram ensure that the following three conditions are met:

- You are currently logged in as one of `cs452`, `mqchen`, or `hkpeprah`.
- You have a directory in which to store the source code, e.g. `~/cs452_microkern_mqchen_hkpeprah`.
- You have a folder on the FTP srever with your username, e.g. `/u/cs452/tftp/ARM/cs452`.

First, you must get a copy of the code. To to this, log into one of the aforementioned accounts and change directories to the directory you created above (using `cd`), then run one of

```
git clone file:////u8/hkpeprah/cs452-microkern -b kernel2 .
                          or
git clone file:////u7/mqchen/cs452/cs452-microkern -b kernel2 .
```

You will now have a working instance of our kernel2 source code in your current directory. To make the application and upload it to the FTP server at the location listed above (`/u/cs452/tftp/ARM/YOUR_USERNAME`), run `make upload`.

1.2 Running the Program

To run the application, you need to load it into the RedBoot terminal. Ensure you've followed the steps listed above in the "Getting the Program" settings to ensure you have the correct directories and account set up. Navigate to the directory in which you cloned the source code and run `make upload`. The uploaded code should now be located at

`/u/cs452/tftp/ARM/YOUR_USERNAME/assn2.elf`

To run the application, go to the RedBoot terminal and run the command

```
load -b 0x00218000 -h 10.15.167.4 ‘‘ARM/YOUR_USERNAME/assn2.elf’’; go
```

The application should now begin by running through the game tasks before reaching a prompt. The generated files will be located in `DIR/build` where `DIR` is the directory you created in the earlier steps. To access and download an existing version of the code, those can be found at `/u/cs452/tftp/ARM/mqchen/assn2.elf` and `/u/cs452/tftp/ARM/hkpeprah/assn2.elf`.

1.3 Command Prompt

At the start, the user will run through 10 players who will each make a request to the server to signup and then play a game. The user can continue through these rounds by following the on-screen prompts. After all games have finished, the user will be presented with an on-screen prompt >, where the user can enter commands. The following commands are supported:

Command	**Description**
q / quit	Exit the prompt and shut down the kernel.
play	Play a game versus the computer.
playc	Watch two computers play a game.

2 Kernel Structure

2.1 Modifications

A new structure, `Envelope_t`, was defined to facilitate message passing between tasks. The struct contains the following information:

Field	**Description**
void *msg	Source address to copy message from
int msglen	Length of msg
void *reply	Destination address to copy reply to
int replylen	Length of reply
Task_t *sender	Pointer to TD of the Sender task
Envelope_t *next	Next envelope in the inbox - these are in a linked list

A pool of `Envelope_t` structures are allocated for message passing, and are retrieved/freed with a simple linked list of free structs. In addition, each Task Descriptor now has 3 additional fields of type `Envelope_t*`, inboxHead, inboxTail, and outbox, that are used to associate envelopes to tasks.
Three new states were added - `SEND_BL`, `RECV_BL` and `REPL_BL`, corresponding to the possible blocked states. In addition, the scheduler was modified such that the active task is only added back to the ready queue if it was not in one of the above states.

2.2 Send

On a Send, an `Envelope_t` is retrieved from the available pool, and if none are available, an error is returned (A more elegant solution could be implemented by blocking the task until one of structs is available). The msg, msglen, reply, and replylen parameters from the `Send` call are copied into the respective fields of the `Envelope_t`. In addition, the current task

pointer is added to the envelope as the sender, and the envelope is set as the outbox pointer of the sender.
The sender is moved to the `RECV_BL` state and the envelope is added to the tail of the receiver's inbound message queue, inboxTail.
If the receiver is in the `SEND_BL` state, it is added back to the ready queues.

2.3 Receive

On a Receive, the inbox of the current task is checked. If there is no inbound messages, the task is moved to the `SEND_BL` state, and nothing else is done. In this case, the task will be unblocked by Send when a message is actually available, and the user task will have to make another call to Receive to get the message. This process is transparent to the user and is done through the user-mode Receive function.
If there is a message available, then the corresponding values in the envelope are copied into the provided pointers, and the sender task is moved to the `REPL_BL` state. The envelope is then removed from the head of the queue so the next message can be received.

2.4 Reply

For Reply, the intended sender task's outbox parameter is used to find the envelope, and the provided reply message is copied into the provided pointer. The sender is then added back to the ready queue and the envelope is released back into the envelope pool.

2.5 Design Decisions

The first significant design decision was how to block tasks on Send/Receive, and how to reschedule them when they become unblocked. Since tasks that are being blocked are always the active task, removing them from the ready queues can be achieved simply by changing their states from `Active` to one of the blocked states, and only add `Active` tasks back to the ready queue. Since the tasks aren't being blocked waiting to request some resource, there is no need to track them in any other way besides removing them from the ready queue.
Rescheduling tasks as they become unblocked can be done as long as a reference to their task descriptor is available. In the case of both Send and Reply, the tid is known, so the TD of the intended task is known as well (in fact, this is required to enqueue the envelope for Send).
The second design decision was how to Receive block. Inside Receive, up to 2 calls to the kernel may be made. The first one is made optimistically, and will return if a message is retrieved successfully from a queue. However, if the queue is empty, then the task is blocked, and Receive will make a second kernel call once it becomes unblocked to retrieve the message.
This approach results in an extra call to the kernel in the Receive-Before-Send case, but makes the code in the primitive simpler as it does not need to handle the special case of storing the Receive parameters. In addition, Send can simply unblock the receiver and

does not need to make the decision between directly copy the message instead of appending to the receiver's queue. This is an optimization that can be made later if the necessity arises.

3 Nameserver

The following function calls have been implemented for the nameserver:

Call	Prototype	Description
RegisterAs	`int RegisterAs(char *name)`	Register the current task with the name `name` in the NameServer. Returns 0 on success, -1 or -2 on error. A single task may register under several different names, but each name is assigned to a single task.
WhoIs	`int WhoIs(char *name)`	Lookup the task with the given name in the NameServer and returns its tid if found. Returns the tid on success, otherwise -1 or -2. Does not block waiting for registration.
UnRegister	`int UnRegister(char *name)`	Unregister the current task iff its name and tid match what is in the NameServer. Returns 0 on success, -1 or -2 on error. Does not remove the found task if the tid found at the hash location does not match the tid of the caller.

NameServer uses a hash map as the data structure to store the tids in; using the names passed by the registering tasks as the keys to hash with. It acts as a global lookup table for tasks to find other tasks. The tid of NameServer is determined at run time by it assigning a value to a global variable within the file and externed through the header files.

4 Data Structures

4.1 HashMap/HashTable

Our HashTable implemented has three attributes:

1. `size` - The size of the array in the HashTable.

2. `data` - An array of stored integers that have been hashed to.

3. `assigned` - An array indicating whether an index in the array is occupied or not (1 for assigned, 0 unassigned).

And the following functions:

1. `init_ht(HashTable*)` - Initialize an hashtable by zero'ing out the assigned bits.

2. `insert_ht(HashTable*, char*, int)` - Insert the integer at the hashed index of the char*.

3. `exists_ht(HashTable*, char*)` - Returns 0 if the hashed index is unassigned, otherwise 1.

4. `lookup_ht(HashTable*, char*)` - Returns the value stored at the hashed index. Will return 0 if the hashed index is unassigned, thus a user should check `exists_ht` before calling for a lookup.

5. `delete_ht(HashTable*, char*)` - Delete what is pointed to by the hashed index. Does nothing if it is unassigned.

5 Algorithms

5.1 Randomization

To implement randomization, we used the Mersenne Twister algorithm for our psuedorandom number generator, as it is a common generator for random numbers, and will guarantee us a deterministic output provided we know the current index in the twister array. Random numbers can be generated by calling either `random()` or `random_range(lower_bound, upper_bound)` from either the user or kernel structure.

1. Populates array with initial values.

2. Randomizes all items in the array.

3. Generates and returns random value at index.

4. Increments to next index.

5.2 NameServer Lookup

To implement lookup to allow the NameServer to register and identify tasks by name, a HashMap (mentioned in the section above) was used. To add items to the hash map we needed to convert the string names into integers. To do this, we used the `djb2` hashing algorithm. We ignore collisions as this is a closed space in which we can ensure there will never be a collision.

1. String is hashed, which is bounded by $O(n)$ where n is the length of the string.
2. Lookup in the hash table is $O(1)$ at this point, by accessing the element in the array at the index generated by the hash.
3. Puts are $O(1)$ by the same methodology.

6 Game Tasks

6.1 Priorities

Task Name	Task ID	Priority
FirstTask	0	15
NameServer	1	15
Server	2	11
Client (NXCLZ)	3	7
Client (HIIJS)	4	8
Client (JUWKR)	5	5
Client (PDYEO)	6	1
Client (NLZEM)	7	7
Client (OGXKF)	8	3
Client (YFLPE)	9	1
Client (CXWTY)	10	6
Client (ABJFT)	11	8
Client (TXRHX)	12	6

The priority for the FirstTask (the task that bootstraps the NameServer, Server, and Clients/-Players) was chosen to be 15 because 15 is the highest possible priority supported by our kernel, thus allowing the task to proceed without yielding to another task, self the Name-Server which blocks on `Receive`, to allow it to create all the tasks as soon as possible before exiting. The priority of the NameServer was chosen to be 15 to ensure that the NameServer was created and did any of the necessary setup work before any other task that would need

it was to be run. This ensure that any task calling `RegisterAs` or `WhoIs` would succeed, as the NameServer `RCV_BLK`s (Receive Blocks) as the last step in its setup. So before any task needs the NameServer, it is already setup and waiting to be sent messages. The priorities of the Server and the Clients are unimportant with respect to one another, as any client with a higher priority than the server would just block waiting for the server to respond to it; the priorities were essentially random to allow for diversity in the result. The only important factor was to ensure that the priorities were less than the NameServer to ensure that the NameServer was ready before they would need it. There were two different approaches to do this; have the first task send a message to the nameserver, blocking and allowing the nameserver to setup and respond before creating the remaining tasks, or have the other tasks lower priority so that the NameServer was the next task scheduled after the first task exited; we chose the latter approach. Since the tasks `SEND_BLK` when they message the server, the server's priority does not matter, as they will be put on its queue when its execute to avoid a task with higher priority looping until it is given a response.

6.2 Game Task Output

The output from the GameTask is as follows:

```
Player HIIJS(Task 4) throwing PAPER
Player ABJFT(Task 11) throwing ROCK
HIIJS won the round
Press any key to continue:

Player NXCLZ(Task 3) throwing PAPER
Player NLZEM(Task 7) throwing SCISSORS
NLZEM won the round
Press any key to continue:

Player CXWTY(Task 10) throwing ROCK
Player TXRHX(Task 12) throwing SCISSORS
CXWTY won the round
Press any key to continue:

Player JUWKR(Task 5) throwing PAPER
Player OGXKF(Task 8) throwing SCISSORS
OGXKF won the round
Press any key to continue:

Player PDYEO(Task 6) throwing SCISSORS
Player YFLPE(Task 9) throwing SCISSORS
```

```
Round was a TIE
Press any key to continue:

Player PDYEO(Task 6) throwing SCISSORS
Player YFLPE(Task 9) throwing ROCK
YFLPE won the round
Press any key to continue:
```

The implementation of `random` uses a set seed, so the results from the game are deterministic, which allows us to argue that the results will always be the same as above. First, the explanation of how Rock-Papers-Scissors works. To begin a game of Rock-Paper-Scissors, two parties must agree to play, at which point, each party throws one of {Rock, Paper, Scissors} simulataenously. Rock beats Scissors, Scissors beats Paper, and for some god awful reason, Paper beats Rock. If both parties throw the same hand, the round ends in a tie, and neither party is victorious. Task 4 and 11 have the highest priority of the client/player tasks, so they are the two first to run. They both signup with the server by sending a SIGNUP request, and give their hands by sending a PLAY request. Since task 4 threw PAPER and task 11 threw ROCK, task 4 wins the round, and they both send a QUIT request if the result is not a tie. The result of the game is sent back as a reply to the two players indicating if they won, if they lost, or if it was a tie. Any tasks that make a request to PLAY while a game is in session, is queued by the server and replied to when there is a free slot for that task to join. The next two tasks then run, and so on, in order of decreasing priority. On a tie, such as in the first case of task 6 and task 9, they play again by sending another PLAY request. So, the order of the tasks by priority is {4, 11, 3, 7, 10, 12, 5, 8, 6, 9}, which takes into account the order the tasks were created. So 4 plays against 11, 3 against 7, 10 against 12, 5 against 8, and 6 against 9 as this is the order in which the tasks run and signup, and the first two tasks to signup are the first two to play; the server sends a reply indicating they can now play and to send it their hands. After each round, the server prints out the tasks in order of when they signed up, what they threw, and the result of the result, then prompts the actual user for input to continue execution.

7 Performance Measurements

7.1 Profiling Technique

Our Makefile was modified to support building in profile mode, with the option of adding 3 define switches (message size, order, cache) as well as the -O2 option that modified our code to operate under the required settings. This is made easy by a script at bin/profiler.sh which generates the .elf executables for each configuration and uploads it to the tftp server. The receiver task must register itself with the nameserver so it can be found by the sender.

This is done by creating the receiver task first, then having the genesis task send a message to it to force a context switch. After the register is complete, the sender task is created and the genesis task exits so the two can run.
The profiling is done using the 40-bit debug timer, and two functions (startProfile, endProfile) that takes the difference between the value of the 40-bit timer as they are called, and cumulates that value into a running average. The send-¿receive-¿reply chain is done 20,000 times for accuracy, and the average is printed once both tasks finish their expected number of sends and receives.

7.2 Results

Message Length	Caches	Send Before Receive*	Optimization	Microseconds
4 bytes	off	yes	off	343.8453713
64 bytes	off	yes	off	462.8687691
4 bytes	on	yes	off	24.41505595
64 bytes	on	yes	off	31.53611394
4 bytes	off	no	off	378.4333672
64 bytes	off	no	off	496.439471
4 bytes	on	no	off	27.46693795
64 bytes	on	no	off	35.60528993
4 bytes	off	yes	on	192.2685656
64 bytes	off	yes	on	231.9430315
4 bytes	on	yes	on	12.20752798
64 bytes	on	yes	on	15.25940997
4 bytes	off	no	on	215.6663276
64 bytes	off	no	on	255.3407935
4 bytes	on	no	on	14.24211597
64 bytes	on	no	on	16.27670397

***** - Assignment says "Send Before Reply", however, replies are non-blocking and don't depend on a send to occur.

7.3 Explanation

Caching increases the performance drastically, and optimization also increases the performance. The larger the message, the more time will be spent during message passing due to memcpy, and Receive-Before-Send takes a performance hit due to the extra call required by Receive.

8 MD5 Hashes

Source files can be accessed at either `/u7/mqchen/cs452/cs452-microkern` or `/u8/hkpeprah/cs452-microkern`. The listing of all the fields being submitted alongisde their MD5 hashes and locations are as follows:

```
8afa04fd4ff12bc483271286d52dfa00  /u8/hkpeprah/cs452-microkern/bin/cs452-upload.sh
de6700ffc18bb2c8f15a491fc2929d13  /u8/hkpeprah/cs452-microkern/bin/md5.sh
7d3d938f3360ca46d07b07d6fed3711c  /u8/hkpeprah/cs452-microkern/bin/profiler.sh
40e6f5862869392d9733ea2d6defbb68  /u8/hkpeprah/cs452-microkern/include/bwio.h
045d5b074001cbda021d83b6a76a0a84  /u8/hkpeprah/cs452-microkern/include/mem.h
19fbec18bdcd2bf2169e51577153fbcf  /u8/hkpeprah/cs452-microkern/include/string.h
8283bfb7e4399ebcbb8179b961658deb  /u8/hkpeprah/cs452-microkern/include/syscall.h
df52cac60678102063e1a09775f4ad10  /u8/hkpeprah/cs452-microkern/include/task.h
742e69b52b3f4cda1898363ef1246fb0  /u8/hkpeprah/cs452-microkern/include/ts7200.h
c72fe0f489f488c99fd6f358e6ed51dc  /u8/hkpeprah/cs452-microkern/include/types.h
d898edf77661ac9a98e2a0c6b9d9b9a6  /u8/hkpeprah/cs452-microkern/include/vargs.h
c1dbfc5172f74422600146c22ca13003  /u8/hkpeprah/cs452-microkern/include/utasks.h
32737a52d46f4d1d70418be469426f9b  /u8/hkpeprah/cs452-microkern/include/k_syscall.h
0d4f70ea95299ed99ce1169d4d19fdda  /u8/hkpeprah/cs452-microkern/include/stdio.h
c1b25aa7cdcd789673ce0166ed95bd92  /u8/hkpeprah/cs452-microkern/include/stdlib.h
2f5a88265c61a41eadccb8b3aa6aade4  /u8/hkpeprah/cs452-microkern/include/term.h
02177f311ecf9a9e993573e780b60a73  /u8/hkpeprah/cs452-microkern/include/syscall_types.h
02da5c5ed64ddf5a5ff08090e1d407cf  /u8/hkpeprah/cs452-microkern/include/hash.h
87c6cd6bf280cb0a927e6c15a2ab8db6  /u8/hkpeprah/cs452-microkern/include/kernel.h
65b51124e5ee634ebdbba3664aa22a63  /u8/hkpeprah/cs452-microkern/include/random.h
ee96caa753939377392c69bf9502f552  /u8/hkpeprah/cs452-microkern/include/time.h
717b31cadb3b90f022dc0690530d1aab  /u8/hkpeprah/cs452-microkern/include/perf_test.h
0c3ff8cf4fed935399a5dadc2b9dc14d  /u8/hkpeprah/cs452-microkern/include/server.h
3c5f1c3403d70d5bc149a7976f3e9fb7  /u8/hkpeprah/cs452-microkern/include/shell.h
138fe549546812cf198e6ae19ea61e8e  /u8/hkpeprah/cs452-microkern/include/util.h
8402c682ff31b15549689baa00ea45b6  /u8/hkpeprah/cs452-microkern/lib/libbwio.a
24cfc902bf98e46713ee4daf1dc7c01d  /u8/hkpeprah/cs452-microkern/Makefile
0f60a59fdd1e9a6cb4ec4aa01acc54ad  /u8/hkpeprah/cs452-microkern/src/mem.c
1efb2e496393dc5465abf0480940928d  /u8/hkpeprah/cs452-microkern/src/orex.ld
4a4a162d7de498e89ab976f72601e7ae  /u8/hkpeprah/cs452-microkern/src/string.c
2c0f3aba5a53aac5c8ff78886315c555  /u8/hkpeprah/cs452-microkern/src/syscall.c
2944b5f88933dcb18949666a2166ad39  /u8/hkpeprah/cs452-microkern/src/task.c
434efff8a8426f1d98c2dd05e4c1456e  /u8/hkpeprah/cs452-microkern/src/main.c
fb02e0ba097afaabae2927e17634270b  /u8/hkpeprah/cs452-microkern/src/hash.c
b59c60300d76a7443e5e4260f70a8232  /u8/hkpeprah/cs452-microkern/src/stdio.c
5e4ef8a626c50e440f460ac0e76d9289  /u8/hkpeprah/cs452-microkern/src/stdlib.c
7a96137012e7594136ada1d672ba87bf  /u8/hkpeprah/cs452-microkern/src/k_syscall.c
8cc626e0fbd9b3e5b82c94595339a514  /u8/hkpeprah/cs452-microkern/src/term.c
c7407dce56f57a09adefa8d9da090974  /u8/hkpeprah/cs452-microkern/src/server.c
e9facfa26e0292dd9ec45587eac01321  /u8/hkpeprah/cs452-microkern/src/kernel.c
6719f648f05c9756ea0fe244a3f95e7e  /u8/hkpeprah/cs452-microkern/src/random.c
52014a086ab65f52ed72471b5cbe42b7  /u8/hkpeprah/cs452-microkern/src/time.c
ebe3801abd7ec411fe0ce35330559696  /u8/hkpeprah/cs452-microkern/src/tasks/utasks.c
b3a9b7222c0e4890e792f8e08f58a5aa  /u8/hkpeprah/cs452-microkern/src/tasks/perf_test.c
5a35fdb34c03b6fa42ef36105622c168  /u8/hkpeprah/cs452-microkern/src/tasks/shell.c
ee681b5aebaeb4ead2fde72443af08b1  /u8/hkpeprah/cs452-microkern/tests/tasks2.c
befff66a349448e4e08aab2f7abbaa51  /u8/hkpeprah/cs452-microkern/tests/tasks1.c
```