Assignment 2 Document

Ruoyi Wang 20373611

Zhenqian He 20383235

# 1. A description of how to operate your program in a readme file, including the full pathname of your executable file which we will download for testing.

## DOWNLOADING THE SOURCE CODE

Download the zip file

go to http://goo.gl/uDY3AW

click File->Download

this will download the r59wang\_z23he\_k2.zip file

Unzip the zipfile will result in a k2\_submission folder

## REBUILD

compile the program in k2\_submission folder

make clean && make

copy the compiled elf file onto tftp

cp kernel.elf /u/cs452/tftp/ARM/ur\_user\_name/

change the permission of the copied file

chmod 744 /u/cs452/tftp/ARM/ur\_user\_name/kernel.elf

load the program onto the board

load -b 0x00218000 -h 10.15.167.4 "ARM/ur\_user\_name/kernel.elf

type go

go

## LOADING AND RUNNING THE PRE-COMPILED VERSION

load -b 0x00218000 -h 10.15.167.4 "ARM/r59wang/k2.elf;go

OPERATING THE PROGRAM

- enter 'q' to quit program

- press any other key to view next match results

# 2. Kernel description, its structure, data structure and explanations

## DATA STRUCTURES

### Message

This is the struct used by the client to pass messages. The message struct holds only two things, the message and the type. The message contains the character array that is too be passed, and the Type describes the type of the message:

typedef struct message\_t

{

char \*value;

int type;

} message;

we wanted to keep the message simple so that client code won't be complex either.

### Mail

Mail is a structure that is internal to the kernel. When the user calls send or receive the kernel function will wrap the message structs with meta data into the mail. Even though it is only used by the kernel it is kept on the kernel stack. Here is what the struct looks like:

typedef struct mailbox\_t {

int \*sender\_tid;

message \*msg;

int msg\_len;

message \*rpl;

int rpl\_len;

struct mailbox\_t \* next;

} mailbox;

On top of the messages, the mail also contains the sender it reference which is kept as a reference for Send to manipulate. It also has the reply address so the kernel can keep track of where to reply to when reply is called. Lastly, the mail contains a next pointer since it is the node for the SendQ structure

### SendQ

The SendQ is a is a queue maintained by the kernel to keep track of the messages that are sent to the task. If there are multiple messages sent to a particular task they will be queued up in the SendQ of the task, before the task replies to them. This allows messages to support multiple send natively. Since the nodes of the queue is kept on the user stacks it does not require any kernel memory. The head of the sendQ lives inside the task descriptors.

### Task Descriptors

Two more fields are added to the task descriptors since the last lab. They are SendQ and arguments:

typedef struct td\_t {

unsigned int tid;

unsigned int pc;

unsigned int sp;

unsigned int spsr;

unsigned int ret;

unsigned int priority;

unsigned int parent\_tid;

unsigned int state;

unsigned int args[5]; // newly added

mailbox \*sendQ; // newly added

struct td\_t \* next;

} td;

The SendQ is explained in the previews section to support data queuing. The decision to include the arguments in the task descriptor arisen from the need support output arguments. Since each task needs to track its output arguments, it is necessary to have that stored for each task.

## SEND, RECEIVE, REPLY

### Send

Send follows 2 code paths, the following is psudo code for how it operates:

if receiving task is send blocked

fill in the receiving tasks request

unblock receiving task

current task (sending task) becomes reply blocked

else

push the message to the receivers sendQ

become receive blocked

### Receive

Receive also follows 2 code paths, the following is psudo code for how it operates:

if the sendQ is not empty

fill in the output arguments with the sendQ content

pop the front of the SendQ

set sender to be reply blocked

else

become send blocked

add a empty node to the sendQ for the sender to fulfill

### Reply

The following pseudo code describes how Reply operates:

If the sender is reply blocked

fill in the output argument of the sender with the reply

unblock the sender

## NAMESERVER

### Data Structure

The nameserver keeps the registration data in an array of structs. The struct is as follows

typedef struct {

char task\_name[32];

int tid;

} name\_server\_element;

The struct keeps track of a task name to its corresponding task id. The array acts as a list and is not ordered.

### RegisterAs

Overwrite is allowed, that is, if a task is killed and re-created, when it re-registers, the nameserver is able to update the array accordingly. This is, however, under the assumption that there will be no malicious user program that will try to take over a name. Because only we are writing the user code, such malicious user programs should not exist.

Because overwrite is allowed, and the array is unordered, the registration will be have a linear run time

### WhoIs

Because is array is unordered, WhoIs has to linearly look through all the elements in the array to find the right one. Therefore WhoIs also has a linear run time

Justification

This data structure, as well as the RegisterAs and WhoIs implementation was chosen because its ease of implementation. Because the code is so small, the room for error is also very small, which makes the NameServer very robust.

## RPS SERVER

The server will FOREVER loop receive. On each receive the server will determine what type of message it is and process it accordingly.

SIGNUP will be queued until two players signed up, which then they will be put into a playerMatchUp array and the queue will be flushed, the server will also reply to the two players that they are play now.

PLAY will be stored each player's play (r/p/s) into a playerPlay array. If both players from the playerMatchUp has played then the results will be calculated and replied. If upon a player play, the opponent has sent quit before, a "opponent has quit" will be replied.

QUIT will be stored in the playerPlay array. a reply will be sent right away. If the player's opponent has already played, a "opponent has quit" will be replied to that player.

## RPS Client

The client will generate a random number between 5 and 10 as the number of times it wants to play the game.

It generates a random r/p/s. then signs up and plays the the throw. The process will repeat.

After the desired amount has been played, the player will call quit.

# 3 & 4. Source code and MD5s

f47beb24b24de182157c5937a227bdac k2\_submission/Perf

d612addd5e16ee06b870c07ad6ebf2eb k2\_submission/README

feb4494a4ccb42685db62f65bd9c8465 k2\_submission/include/ts7200.h

646f59ed3027a40ebb5a224ca4a48475 k2\_submission/kernel/ker\_ent\_exit.asm

2346b2f758d7adbf5dfc2c38cfbd1ac2 k2\_submission/kernel/kernel.c

bdc8977e06856a148e403a5493beede9 k2\_submission/kernel/kernel.h

605a1438ac7738c0c486430e52b165e5 k2\_submission/kernel/queue.h

5fc9980bb9bbcbeedc9e226f8364961c k2\_submission/kernel/bwio.c

57d7ecc932e069e6f3c8f5044ea66a6f k2\_submission/kernel/queue.c

fe178ee05cba742a7b79153dbb1e4edf k2\_submission/kernel/nameserver.c

1a69f84e7cda9f0c4a76bb93b8c436b8 k2\_submission/kernel/README

4c7903df0ada55219827ec269eba5185 k2\_submission/kernel/nameserver.h

acf5cbd58430eb0908e0ab796e6c735f k2\_submission/kernel.elf

6ceb1a0c85fdab0f2119876ee03917cd k2\_submission/lib/libbwio.a

25757ddee6d82a16be926075e04b4972 k2\_submission/lib/bwio.c

d32dda3f6cd59b210c03d1ed8332c581 k2\_submission/lib/bwio.h

bec86307038bfec7aa389e3562a367d0 k2\_submission/lib/libutil.a

c6538628a9719d785aa380e531606c8b k2\_submission/lib/util.c

01322e434a9ce0580ded7f1ff1b8a0e2 k2\_submission/lib/util.h

03e9787757653cf473db65e21d3ce200 k2\_submission/makefile

9d0946ab946bf682994dedaa174192d3 k2\_submission/orex.ld

ec06466d88a33331e86ce93e8df102c7 k2\_submission/tasks/Tasks.c

b475056a53d4a9420025cbf8c9da4ba0 k2\_submission/tasks/Tasks.h

# 5. Game task priority

All the game tasks all have the same priority.

The game tasks both server and client were given the same priority because it does not matter which tasks are being executed first or second, as long as they all get a turn. The namespace will not be an issue as well since the clients will loop on the Whois call for the rps server until they get a valid response.

# 6. Measurements, where time is being spent

## MEASUREMENTS

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MSG LENGTH | CHACHING | SEND BEFORE RECEIVE | O2 OPTIMIZATION | GROUP NAME | TIME (in us) |
| 4 bytes | off | yes | off | r59wang\_z23he | 394 |
| 64 bytes | off | yes | off | r59wang\_z23he | 881 |
| 4 bytes | on | yes | off | r59wang\_z23he | 31 |
| 64 bytes | on | yes | off | r59wang\_z23he | 65 |
| 4 bytes | off | no | off | r59wang\_z23he | 398 |
| 64 bytes | off | no | off | r59wang\_z23he | 889 |
| 4 bytes | on | no | off | r59wang\_z23he | 29 |
| 64 bytes | on | no | off | r59wang\_z23he | 62 |
| 4 bytes | off | yes | on | r59wang\_z23he | broken |
| 64 bytes | off | yes | on | r59wang\_z23he | broken |
| 4 bytes | on | yes | on | r59wang\_z23he | broken |
| 64 bytes | on | yes | on | r59wang\_z23he | broken |
| 4 bytes | off | no | on | r59wang\_z23he | broken |
| 64 bytes | off | no | on | r59wang\_z23he | broken |
| 4 bytes | on | no | on | r59wang\_z23he | broken |
| 64 bytes | on | no | on | r59wang\_z23he | broken |

## TIME SPENT

Much of the time is spent on strcpy, as sending a larger message is much slower than sending a small message. The rest of the time is spent on context switching and the actual execution of logic of the message passing logic.

# 7. Output

The output will be appearing in pairs, which is one round of RPS.

Each one of the paired output will contain the following info

player's tid

player's play: either (r)ock, (p)aper, or (s)cissors

player's result: which can be "WIN", "LOSE", or "TIE"

The output prints are on each player's task, right after they received a reply from "PLAY".

The pair output was able to work because the "getc" for user input is on the server task, right before the server replies to both players in the current round. The two replies will be sent in sequence, each reply will trigger the print from the player's task, which results in the paired prints.

Each player's play of r/p/s is randomly generated, each player will sign up and play a random (5-10) amount of times, so there is no definitive output results without running the program.