Kernel 3 Document

Ruoyi Wang 20373611

Zhenqian He 20383235

# 1. A description of how to access, make and operate your program.

## DOWNLOADING THE SOURCE CODE

Download the zip file

http://goo.gl/7U6aNM

Click File->Download

This will download the r59wang\_z23he\_k3.zip file

Unzip the zipfile will result in a k3\_submission folder

## REBUILD

Compile the program in k3\_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/z23he/kernel3.elf;go

## OPERATING THE PROGRAM

- The program will produce the outputs for the clock server tasks

- The user can press "q" to exit the program

# 2. A description of the structure of your kernel so far.

## CLOCK SERVER

### Data Structure

The clock server has a very similar implementation has the name server completed in kernel 2.

Because each task would be blocked when it calls delay or delayuntil, we would need a maximum of 64 element to hold the delays. Each element have the following structure

typedef struct {

int tid;

unsigned int delay;

} delay\_element;

the elements are stored in an array, and on each tick, the array is iterated, the delay is decremented and replies are sent if the delay is 0.

### Delay

This is a wrapper for send, much like "RegisterAs" in the nameserver, it sends a message to the clock server. The clock server will hold on to the message until the delayed time frame is met, then the message will be replied to.

### DelayUntil

This is the same as Delay, except the clock server will calculate the time frame based on the current time and time given.

### Time

This is another wrapper for send, the clock server, once received the call, will reply to the message immediately.

### clockServerNotifier

This is the notifier created by the clock server to notify itself about each tick. The notifier is another task that forever loops and awaits for the CLOCK\_EVENT, once the event occurs, a message will be sent to the clock server. The clockserver will then add a tick to its clock. The clock server will reply so the notifier can await again.

## SYSTEM IDLE TASK

The system idle task is just a task that loops forever at the lowest priority to make sure the system is always running, even if there's nothing happening.

## HARDWARE INTERRUPTS

The hardware interrupt kernel entry an exit are kept separated from the software interrupt kernel codes. Even though the code is kept separated, their functionality is quite similar.

### Entry code:

msr cpsr\_c, #0xdf /\* change to system mode \*/

stmdb sp!, {r0-r12, lr, ip} /\* store the task registers \*/

mov ip, sp

msr cpsr\_c, #0xd2 /\* change back to irq mode \*/

mov r3, lr /\* save the lr to r3 \*/

sub r3, lr, #4

msr cpsr\_c, #0xd3

ldmia sp!, {r1} /\* save all the argments \*/

ldmia sp!, {r0} /\* pop the kernal stack \*/

str r3, [r0, #0x4] /\* write to the lr of the TD \*/

mrs r2, spsr /\* save the spsr to r2 \*/

str r2, [r0, #0xc] /\* store spsr to TD \*/

str ip, [r0, #0x8] /\* store the sp to the TD \*/

mov r0, #20 /\* get the swi arguemnt and return it \*/

ldmia sp!, { r4-r12, lr} /\* pop the kernal stack \*/

mov pc, lr /\* this go back to kernel \*/

The first step done on entry is to go to system mode with interrupt turned off to prevent interrupt overloading. Then all the registers including the scratch registers are saved onto the user mode stack. Then switch to irq mode to get the lr to be saved in the task descriptor. Lastly, switch to supervisor mode and save the state to the task descriptor and pop the kernel stack before finally entering the kernel.

### Exit code:

stmdb sp!, { r4-r12, lr} /\* move registers to the kernal stack \*/

stmdb sp!, {r0} /\* move registers to the kernal stack \*/

stmdb sp!, {r1} /\* move registers to the kernal stack \*/

ldr r2, [r0, #0xc] /\* load the task spsr bak into cpsr \*/

msr spsr, r2 /\* TODO: need to investigate this line \*/

ldr lr, [r0, #4] /\* load the pc of the td \*/

msr cpsr\_c, #0xdf /\* change to system mode \*/

ldr sp, [r0, #8] /\* load the td sp \*/

ldmia sp!, {r0-r12, lr, ip} /\* reload r0 - r10 of the task \*/

msr cpsr\_c, #0xd3 /\* change to system mode \*/

movs pc, lr

First save the kernel registers to the kernel stack. Now load the spsr and the lr accordingly. Go to system mode and get the sp to unwind the stack. Lastly return to the user task with the movs command.

The key thing to note is that since there is a separate kernel entry and exit for software vs hardware interrupts, they must match each other correctly. A new state READY\_INT is created to make the differentiation. A task that was hardware interrupted will always return using the hardware kernel exit, and similarly for the software interrupts entry and exit will always match.

### Interrupt handling

Interrupt handling is done in the normal handle function. It has a jump entry for an interrupt request, which will read the interrupt status and handle it accordingly. Generally, it will clear the interrupt flag and unblock the task that was waiting on the particular even related to the interrupt.

### Event Block

A new state is created for tasks that are blocked on wait events. Since each type of event can only have at most one task waiting for it, a simple array of the blocked tasks is used to allow for fast constant time access of the blocked tasks given the event. So far the array size is 5, but can be increased if more types of events are introduced.

# 3. The location of all source code you created for the assignment and a set of MD5 hashes of each file. The code must remain unmodified after submission until the assignments are returned.

feb4494a4ccb42685db62f65bd9c8465 include/ts7200.h

5fc9980bb9bbcbeedc9e226f8364961c kernel/bwio.c

b9b7470560031946dab7f959b1bace6d kernel/int\_ker\_ent\_exit.asm

5f801609c2ee3af70a7bc962f4a74494 kernel/ker\_ent\_exit.asm

b8abe1b687ea0cbd1e86a22beda0de38 kernel/kernel.c

657c7d099d8321824466f66d48bb48a2 kernel/kernel.h

fe178ee05cba742a7b79153dbb1e4edf kernel/nameserver.c

4c7903df0ada55219827ec269eba5185 kernel/nameserver.h

57d7ecc932e069e6f3c8f5044ea66a6f kernel/queue.c

605a1438ac7738c0c486430e52b165e5 kernel/queue.h

b6c5125ddef0fd675ce94cdec50f05be lib/a.out

25757ddee6d82a16be926075e04b4972 lib/bwio.c

d32dda3f6cd59b210c03d1ed8332c581 lib/bwio.h

6ceb1a0c85fdab0f2119876ee03917cd lib/libbwio.a

1c5df5933e36876c27d5dfbe3583abb8 lib/libutil.a

86b1a62b68ad1e4341f89be2e5448079 lib/util.c

01322e434a9ce0580ded7f1ff1b8a0e2 lib/util.h

90002ed2bb0f1a3a5cb64bd2a3b6a7b5 makefile

9d0946ab946bf682994dedaa174192d3 orex.ld

5a5fac36e60d3baee135dd3721ce4467 tasks/Tasks.c

acfd9a58c040faf90fbc782c937f4883 tasks/Tasks.h

642b4579317103575b591413580282d9 tasks/clockserver.c

f0da0dd5b3ebffb8597c305dead61152 tasks/clockserver.h

7795519dc4161c26c8ce2bfdc6e97aa8 tasks/rps.c

4ada5de8073ec8c6fab66719f12c8f66 tasks/rps.h

# 4. Output produced by your client tasks and an explanation of why it occurs in the order it does

Task: 5 | ticks delayed: 10 | 11 | delays completed: 0

Task: 5 | ticks delayed: 10 | 21 | delays completed: 1

Task: 6 | ticks delayed: 23 | 24 | delays completed: 0

Task: 5 | ticks delayed: 10 | 31 | delays completed: 2

Task: 7 | ticks delayed: 33 | 34 | delays completed: 0

Task: 5 | ticks delayed: 10 | 41 | delays completed: 3

Task: 6 | ticks delayed: 23 | 47 | delays completed: 1

Task: 5 | ticks delayed: 10 | 51 | delays completed: 4

Task: 5 | ticks delayed: 10 | 61 | delays completed: 5

Task: 7 | ticks delayed: 33 | 67 | delays completed: 1

Task: 6 | ticks delayed: 23 | 70 | delays completed: 2

Task: 5 | ticks delayed: 10 | 71 | delays completed: 6

Task: 8 | ticks delayed: 71 | 72 | delays completed: 0

Task: 5 | ticks delayed: 10 | 81 | delays completed: 7

Task: 5 | ticks delayed: 10 | 91 | delays completed: 8

Task: 6 | ticks delayed: 23 | 93 | delays completed: 3

Task: 7 | ticks delayed: 33 | 100 | delays completed: 2

Task: 5 | ticks delayed: 10 | 101 | delays completed: 9

Task: 5 | ticks delayed: 10 | 111 | delays completed: 10

Task: 6 | ticks delayed: 23 | 116 | delays completed: 4

Task: 5 | ticks delayed: 10 | 121 | delays completed: 11

Task: 5 | ticks delayed: 10 | 131 | delays completed: 12

Task: 7 | ticks delayed: 33 | 133 | delays completed: 3

Task: 6 | ticks delayed: 23 | 139 | delays completed: 5

Task: 5 | ticks delayed: 10 | 141 | delays completed: 13

Task: 8 | ticks delayed: 71 | 143 | delays completed: 1

Task: 5 | ticks delayed: 10 | 151 | delays completed: 14

Task: 5 | ticks delayed: 10 | 161 | delays completed: 15

Task: 6 | ticks delayed: 23 | 162 | delays completed: 6

Task: 7 | ticks delayed: 33 | 166 | delays completed: 4

Task: 5 | ticks delayed: 10 | 171 | delays completed: 16

Task: 5 | ticks delayed: 10 | 181 | delays completed: 17

Task: 6 | ticks delayed: 23 | 185 | delays completed: 7

Task: 5 | ticks delayed: 10 | 191 | delays completed: 18

Task: 7 | ticks delayed: 33 | 199 | delays completed: 5

Task: 5 | ticks delayed: 10 | 201 | delays completed: 19

Task: 6 | ticks delayed: 23 | 208 | delays completed: 8

Task: 8 | ticks delayed: 71 | 214 | delays completed: 2

Note: The "delays completed” is zero-based.

Note: The third column is current clock tick.

The output is in the order of "delay completed" times the "ticks delayed". Which is the expected outcome.