Assignment 1 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.

Download the zip file

Go to http://goo.gl/XZVl8c

Click File->Download

This will download the r59wang\_z23he\_k1.zip file

Unzip the zip file will result in a k1\_submission folder

Navigate to k1\_submission/kernel, the makefile resides there

Compile the program in kernel 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

The kernel will need no user interaction and will create tasks, print, and exit themselves accordingly

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

## DATA STRUCTURES

### The kernel stack holds the following data:

active - a pointer to the active task

tds - an array of 64 task descriptors

td\_pq - an array of 16 priority queues

args - an array of 5 arguments that are passed by the request

free\_list\_lo and free\_list\_hi - 64 bits that indicate the free task descriptors

We choose to have a simple array to pass the arguments because we feel like the structure of the array better describe an argument list semantically compared to structs. We used 2 32-bit unsigned int as our free list, where each bit describes which of the td is free to use. We feel that this method requires the least maintenance, and therefore less likely to have bugs. The priority queue simple has a pointer to the front and end of the queue, for constant push and pop.

### The task descriptor contains the following fields:

unsigned int tid;

unsigned int pc;

unsigned int sp;

unsigned int spsr;

unsigned int ret; //return

unsigned int priority;

unsigned int parent\_tid;

unsigned int state;

struct td\_t \* next;

The purpose of the fields should be self-explanatory from the names. We make sure to never change the order of the fields as they are accessed based on offset from assembly.

### Priority Queue

The priority queue is constructed with as an array of queues. The array has a size of 16 to represent each of the 16 priorities, where 0 is highest priority and 15 is the lowest.

Each element in the queue is represented by 2 pointers, the "front" pointer points to the td at the front of the queue, where the "back" pointer points to the last td the end of the queue. Below is the declaration of the element

typedef struct td\_queue\_t {

struct td\_t \* front;

struct td\_t \* back;

} td\_queue;

This structure is used as it only takes a total of 16 \* 2 variables in total. Each queue also takes a constant time of to pop and push.

### Free List

We are using 2 unsigned integers "free\_list\_lo", and "free\_list\_hi" to keep track which TDs are available.

Each unsigned int is 32 bits, 2 unsigned integers will track the 64 TDs.

Each bit of "free\_list\_lo", and "free\_list\_hi" will represent the availability of the TD, where 0 is free, and 1 is used.

Note "free\_list\_lo" will represent the TDs 31-0 and "free\_list\_hi" will represent the TDs 63-32.

We chose to use two unsigned integers to keep track of which TD is free, this implementation will minimize the amount of memory usage. While the run time of the algorithm of finding the first free TD is n, we only have a total of 64 TDs, which means the worst case is actually 64.

## KERNEL ENTRY AND EXIT

### Kernel exit executes the following steps:

- save the registers to kernel stack including the 2 arguments passed into it

- go to system mode

- get the sp and the return value from the td

- go back to supervisor mode

- load the cpsr from the td spsr

- load the pc of the task

### Kernel entry executes the following steps:

- get the location of the argument array by popping it off the kernel stack

- now load 5 arguments into the array regardless if how many arguments there are ( make the assumption there are less than 5 fore now, might need adjustments in the future )

- move the lr to a register (r3)

- go to system mode

- store the user registers to the stack

- save the sp in the ip

- go to supervisor mode

- get the active task descriptor from the kernel stack

- save the lr (now in r3) as pc, sp (now in ip), spsr to the task descriptor

- move the swi argument to r0 as the return value

- unwind the rest of the kernel stack

There can be a lot of optimization done to our current context switch, but for the current tasks it operates well. The assembly is done through the NOP pattern to ensure that the state does not have unexpected changes.

## SCHEDULING

The scheduling algorithm uses the priority queue discussed in the previous section

### Schedule (getting the task that should run)

The scheduling algorithm will loop through all of the 16 priorities, starting from 0. Look for the highest priority and pop it from the queue.

The loop process, once again has a linear run time, but since there is only 16 priorities, the loop will always execute less than or equal to 16 times.

### Push task back (putting a task on to the priority queue)

After the request has been cleared, if the process is still alive, it's td will be again pushed on to the priority queue

Since we know the priority at this point, the insert time will be constant.

## KERNEL CALLS AND HANDLE:

Kernel functions such as Create, Pass, MyTid all simply have one 1 line:

swi n

Where n, identifies the type of kernel call. then kernel entry will return the the n value to be passed to the handle function. There the handle function will have a switch statement that fullfills the request accordingly.

# 3. Source code and MD5s

d32dda3f6cd59b210c03d1ed8332c581 k1\_submission/include/bwio.h

6e91255abc9047f2c83a198c756b358c k1\_submission/include/interface.h

9af226f127c1fd759530cd45236c37b8 k1\_submission/include/ts7200.h

ba868ea1845b6aa4af4cb1feee528228 k1\_submission/lib/libbwio.a

306b737ea7dedb289c07eb3928c97424 k1\_submission/kernel/kernel.s

ea4b5fefedd6ca87106ec92294429de3 k1\_submission/kernel/kernel.c

defb1a74b525d158825a442581e8aa01 k1\_submission/kernel/kernel.h

9bcd562566ba01c869c8c2759ff90e64 k1\_submission/kernel/orex.ld

5fc9980bb9bbcbeedc9e226f8364961c k1\_submission/kernel/bwio.c

6311086f6d5f3fb82a2d7ebb78dd31c6 k1\_submission/kernel/makefile

d16460574fb912f398956879052e1140 k1\_submission/kernel/ker\_ent\_exit.s

ea1c60c36282fbf53aaccd864f3cecce k1\_submission/kernel/kernel.o

28801b0cad024f24826d94082c5fd72e k1\_submission/kernel/kernel.map

fda42ae90bafb1f58202f93437b4e16e k1\_submission/kernel/kernel.elf

# 4. Output explanation

Output produced:

Created: 1

Created: 2

Spawned task tid: 3

Spawned parent task tid: 0

Spawned task tid: 3

Spawned parent task tid: 0

Created 3

Spawned task tid: 4

Spawned parent task tid: 0

Spawned task tid: 4

Spawned parent task tid: 0

Created 4

First: Exiting

Spawned task tid: 1

Spawned parent task tid: 0

Spawned task tid: 2

Spawned parent task tid: 0

Spawned task tid: 1

Spawned parent task tid: 0

Spawned task tid: 2

Spawned parent task tid: 0

This behaviour is expected. Since the first task creates 2 tasks that are lower priority than itself it will print created 1 and 2 before those task will run.

The first task than creates a task with a higher priority than itself which runs immediately, it completes before giving up the cpu, since it has the highest priority.

Then the first task runs again and prints the tid of the task that just finished and creates the last task. The behaviour is repeated from the previous task.

The first task now completes, allowing the first two created tasks to run. Since the remaining tasks have the same priority, they will toggle control of the cpu each time pass is called. This continues until they complete and program exits.