CS 452 K1

Names: Robert Elder, Christopher Foo
ID #: 20335246, 20309244
Userids: relder, chfoo
Date due: May 27, 2013

Running

The executable is located at `/u/cs452/tftp/ARM/relder-chfoo/k1-submit/kern.elf`.
It is executed using the regular commands:

```
load -b 0x00218000 -h 10.15.167.4 ARM/relder-chfoo/k1-
submit/kern.elf
go
```

Description

Kernel

The entry point is located in `kern.c`.
The kernel follows the following:

1. Sets the location of our SWI routine.
2. Sets the stack pointer to accommodate our Kernel State.
3. Initialize the kernel (File `private_kernel_interface.c`:`k_InitKernel()`).

    1. Save the SP and LR values so the kernel can exit back to RedBoot.
    2. Initialize the pseudo Task Descriptor.
    3. Initialize the queues.
    4. Set the SP and LR value of the pseudo Task Descriptor to the Kernel State
    5. Call the `asm_KernelExit` routine to push the values to the register.

4. Jump to KernelTask_Start (File `tasks.c`)
5. Start our first user task that starts the 4 other generic tasks.

Scheduling

Scheduling occurs for all kernel calls. Before the current task descriptor is swapped out,

1. The user task's SP, LR, and SPSR values are saved into the current task descriptor.
2. Any related values are also saved into the TD.
3. The next task is selected (`schedule_next_task()`).
    1. The current task is set to `READY`.
    2. A task is removed from the Priority Queue.
        - Any tasks in the `ZOMBIE` state are not re-queued. Go back to step 1.
    3. Use the pointer to the task as the next task to be run.
    4. Set the selected task as `ACTIVE`.
    5. Reschedule the selected task by adding it back to the Priority Queue.
4. If there no more tasks in the Priority Queue, the kernel exits back to RedBoot using the values we saved on the Kernel State.
5. Otherwise, the SP, LR, and return values are saved into the Kernel State.
6. Assembly routine `asm_KernelExit` pushes these values to the registers.

User tasks

The tasks are defined in `tasks.c`.

`KernelTask` is a pseudo task with a pseudo task descriptor. It is simply used to start the first user task and does not get rescheduled. It has a task id of 0.

The first user task is called `FirstTask`. It has a task id of 1. The other tasks are called `GenericTask`.

System Calls

The system calls should be complete in respect to the specifications.

`Create` Returns the new task id, `ERR_K_INVALID_PRIORITY -1`, or `ERR_K_OUT_OF_TD -2`

`MyTid` Returns the current task id

`MyParentTid` Returns the parent task id. The parent task id is always returned regardless of the parent's state.

`Pass` (Rescheduling happens as normal in the background.)

`Exit` Task is marked as `ZOMBIE` (and rescheduling happens as normal in the background).

Algorithms and Data structures

Queue

File: `queue.c`

The queue, a `struct`, is implemented as a ring buffer. A start and end index is used to point to the start and end of the array. Each item is a `void*`. The ring buffer allows adding and removing an item from the queue in constant time. A null pointer is returned if the queue is empty.

Priority Queue

The priority queue consists of 5 queues for 5 levels of priority: highest, high, normal, low, and lowest. `NORMAL` is the default priority. Adding and removing an item is constant time.

Priority	**Int**
HIGHEST	1
HIGH	2
NORMAL	3
LOW	4
LOWEST	5

Performance can be improved for removing an item in the priority queue. It currently checks all queues. An additional variable that tracks the highest, non-empty queue could be used.

Task Descriptor (TD)

File: `task_descriptor.c`

The TD, a `struct`, holds important information such as the task id, state, and return values.

Kernel State

File: `kernel_state.h`

The Kernel State is a `struct` stored at `0x01500000 - sizeof(KernelState)`. It contains values such as the SP, LR, and return values that are set and retrieved in C code. Once these values are set, a routine is run in assembly code that pushes these values to the appropriate registers. The same information is also written to the struct directly when entering a kernel function. This method makes it convenient for writing in C.

The Kernel State also contains information about the Task Descriptors.

Source Code

The source code is located at `/u4/chfoo/cs452/group/k1-submit/io/kernel-1-submission`. It can be compiled by running `make`.

Source code MD5 hashes:

```
chfoo@nettop37:~/cs452/group/k1-submit/io/kernel-1-
submission$ md5sum *
6f52c9e07c8e16288b0f6e70ac1bbd52  Makefile
bb97a5a42f82d99c9766caa1277ee231  buffer.c
5be428c52822585e9e397ff12f9af96f  buffer.h
e270fd64ae08a0317d37fadedd24cabb  kern.c
634a19ff734f7bb6c8b33f110e66696b  kern.elf
d41d8cd98f00b204e9800998ecf8427e  kern.h
b00a171e052d7c818750f58a3bdcf27c  kernel_control_flow.pdf
52dd3c8bac8b93e7bc9024ca3e56b00a  kernel_stack.pdf
98f7a503cb32985bcd45b4f75b1844d8  kernel_state.h
4aa618b9753c5292e5d9e5c95d297f10  orex.ld
ee534990a4714e0699c3e38aae6ec9d1  private_kernel_-
interface.c
bae820d4171cdc89818dbff01d5ac374  private_kernel_-
interface.h
48aaad68699d272e84cc0794d9149d7a  public_kernel_-
interface.c
90621ac9a036d7786da4b8afd2df482e  public_kernel_-
interface.h
9cb336d84ff0e62c35f9c6ba24b5ab05  queue.c
dd0449e95a89088411b71aac6825b6cf  queue.h
50c0e650f22e669776f99c5b9fe41d84  readme.rst
9070188c20a1d659520f46c95e8c60be  robio.c
e9f9061a7e008eb95988b478164a75df  robio.h
5b5afc928a7807d129e319ad4cd7c557  swi_kernel_interface.s
89c7c55442b259b16bc5336bbc567fe2  task_descriptor.c
fa673eaf431d48587330386fa421a961  task_descriptor.h
a0b2c347ea4836aaf330c43bd55fdd9a  tasks.c
e37b5f09bcd33f5c1665fe85fad38f6e  tasks.h
```

Elf MD5 hash:

```
chfoo@nettop37:/u/cs452/tftp/ARM/relder-chfoo/k1-
submit$ md5sum kern.elf
634a19ff734f7bb6c8b33f110e66696b  kern.elf
```

Git sha1 hash: `cacb8815c528e5b1533254233b04c3ba4eb96c74`

Output

The executable prints the following:

1. `FirstTask`, with ID 1, prints the message about creating two tasks 2 and 3
2. Task 4 executes.
    - Task 4 executes because it is created with `HIGH` priority. The `FirstTask` has only `NORMAL` priority.
3. `FirstTask` prints that it created task 4.
4. Task 5 executes.
    - Task 5 has `HIGH` priority
5. `FirstTask` prints that it created task 5.
6. `FirstTask` exits.
7. Task 2 runs.
    - Task 2 has `LOW` priority so it runs only now.
8. Task 3 runs.
9. Task 2 runs.
    - Task 2 and 3 have equal priority so they are queued right after each other.
10. Task 3 runs.