CS 452: Kernel 1

Jason Sun (#20387090) and Shuo Chang (#20378235)

May 22, 2015

1 Program Operation

```
> load -b 0x00200000 -h 129.97.167.12 "ARM/j53sun/k1.elf"
> go
```

All system calls required by assignment are supported:

`int Create(int priority, void (*code)())` Schedule a task with specified `priority` and function pointer `code`.

`int MyTid()` Return the task id for the calling task.

`int MyParentTid()` Return the task id of the parent of the calling task.

`void Pass()` No-op for entering the kernel.

`void Exit()` Exits the calling task and never schedule it again.

2 Kernel Details

2.1 Context Switch

From kernel space to user space:

1. `TaskDescriptor *td`, `Syscall **request` are passed to `KernelExit()`
2. Store all kernel registers onto kernel stack
3. Change to system mode
4. Put `td->sp` to user's `sp`
5. Put `td->ret` to `r0` which returns result of system calls to user task
6. Load all user registers: `r1` contains `pc` of user mode, whereas `r2` contains the saved `cpsr` of user mode
7. Change back to supervisor mode

8. Put saved user mode `cpsr` into `spsr` of supervisor mode
9. `movs pc, r1` to jump to user code while simultaneously change `cpsr`

From user space to kernel space:

1. Put `lr` of supervisor mode in `r1`
2. Put `spsr` of supervisor mode, which is the saved `cpsr` of user mode, into `r2`
3. Change to system mode
4. Store user registers `r1-r12` and `lr`, to user stack
5. Move `sp`, `r0` to `r2`, `r3`
6. Change back to supervisor mode
7. Load multiple from stack into `r0`, and `r1`: `r0` contains pointer to task descriptor, `r1` contains pointer to pointer to request
8. Store `r2` to `td->sp`, `r3` to `*request`
9. Load the rest of the kernel's registers (`r2-r12`) from stack

2.1.1 Description in ARM

The piece of code responsible for context switch is:

```
KernelExit:
    stmfd   sp!, {r0-r12, lr}
    msr cpsr_c, #0xdf
    ldr sp, [r0, #12]
    ldr r0, [r0, #8]
    ldmfd   sp!, {r1-r12, lr}
    msr cpsr_c, #0xd3
    msr spsr, r2
    movs pc, r1
KernelEnter:
    mov r1, lr
    mrs r2, spsr
    msr cpsr_c, #0xdf
    stmfd   sp!, {r1-r12, lr}
    mov r2, sp
    mov r3, r0
    msr cpsr_c, #0xd3
    ldmfd sp!, {r0, r1}
    str r2, [r0, #12]
    str r3, [r1]
    ldmfd sp!, {r2-r12, pc}
```

2.1.2 Trap Frame

When the user does a syscall, a trap frame is set up on the top of the user stack to store user's current registers. The layout of registers stored is:

```
[ R1 (PC)   ] <-- SP after storing trap frame
[ R2 (CSPR) ]
[   ...     ]
[ R12       ]
[ LR        ]
[   ...     ] <-- SP at SWI instruction
```

Initializing the trap frame for the first time is done in `taskCreate()`, and for later context context switches, the trap frame is handled in `context_switch.s`, written in assembly code. On return, the result of the syscall is stored in `r0`, and execution resumes at function `swi()`, where the syscall occurred.

2.2 Syscalls

Syscalls defined in C functions, in `syscall.{c,h}` files. There is a `Syscall` structure that contains the syscall type, and args 1 and args 2.

2.3 Tasks

A task can be created off a function pointer and represents a chunk of code to execute.

2.3.1 Task Descriptor

A TaskDescriptor struct holds:

- task id, contains an index into a global table of task descriptors preallocated.
- parent id, whoever called `Create()`.
- return value,
- stack pointer, `sp` and `spsr` are manipulated by the context switch.
- saved program status register
- and a pointer to the next task descriptor for singly linked list.

Currently only can create 128 tasks before failing to create more tasks.

2.3.2 Scheduling

Tasks each has a priority level. The scheduler tracks this tasks' priority via 32 ring buffer queues.

A bitmask keeps track of which of the 32 queues contains tasks. Using this bitmask, we efficiently computing the number of right leading zeroes in the bitmask with De Bruijn table lookup.

The kernel calls`taskSchedule()` on each loop, and the queue with the highest priority is returned. The head of that queue is rotated to be the tail and the pointer is returned as the next task to be scheduled.

2.3.3 Task Creation

A task is created by specifying a priority, a function pointer, and parent task id. The `Create()` syscall is implemented by this function.

A task descriptor is filled in to the task table. Then a stack is allocated, a size of 4096 words. There is syscall to change a task's stack size. It also initializes a trap frame by setting`pc` to the value of the function pointer and saved stored program register.

Finally the kernel adds the task descriptor to priority queue.

2.3.4 Task Exit & Deletion

Once a task is removed from the priority queues, the task will not be scheduled again. No effort is made to reclaim task descriptors.

3 Source Code Location

Code is located under `/u1/j53sun/cs452k1`/.

Compiling by running `make`, which also copies the local `kernel.elf` to `/u/cs452/tftp/ARM/j53sun/k1.elf`.

```
File md5sums

    8da586d949d31e239dfbe0c8356588f6  bwio.c
    db0bab80ffcef52c8ce1a968c65587a9  bwio.h
    c99bf6f10dd0ec08f47124c8968a7a54  context_switch.h
    1ba8cd1b57c22116b57e96d22022cec5  context_switch.s
    e9ecc0c507565cc766ec637a9aec3ab6  cpsr.h
    7742f42b8758e1c75de72f01a94a4ce0  kernel.c
    e87799ad275ab3fd1199dba2ea334e5c  linker.ld
    7c1b255735fd098a6ecd2b8a8903a0d9  Makefile
    867bebe51a877a07650ec35b39a8808a  scheduler.c
    d053b19cbdee9ddb872ad8c69841c478  scheduler.h
    d6bdf5714a8d499da29f1064973580ae  stdbool.h
    9a67eb2e94c96d24e676c2c87553cca5  syscall.c
```

```
efdd19b65fbc4089056d0dd3fbc1f2c5  syscall.h
9c9e816e473306b143c273e6fa48a300  task.c
d5beb6f6b87d1d257d48b271aa8ce34d  task.h
2c5fc627ac5386f1f96a65d2f8dc9d67  ts7200.h
5b63a07500bc5f2ccb08a47ccc7fadaa  user_task.c
962dc8dab4c71088e86d7c7c6bb9adc8  user_task.h
```

4 Program Output

```
RedBoot> go
Created: 2
Created: 3
Task 4, Parent: -1
Task 4, Parent: -1
Created: 4
Task: 5, Parent: -1
Task: 5, Parent: -1
Created: 5
First: exiting
Task: 2, Parent: -1
Task: 3, Parent: -1
Task: 2, Parent: -1
Task: 3, Parent: -1
No task scheduled; exiting...
Program completed with status 0
```

Explaination

1. Task 2 and 3 has lower priority than the creating task, so the creating task is still going to be scheduled to run.
2. When task 4 and 5 are created, they have a higher priority than the creating task. Thus they each run to completion before scheduler returns to the creating task (and thus delaying creation of second task).
3. The creating task then exits, after the second create call returns.
4. Only then does task 2 and 3 get to run. They alternate running every time they call pass because they have the same priority in a circular queue.