CS 452 Kernel 1

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

May 22, 2015

1 Program Operation

```
> load -b 0x00200000 -h 129.97.167.12 "ARM/sun-chang-team/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. Save user task `sp` into `r0`, variable register called `sp_`
2. Save user task `spsr` into `r1`, variable register called `spsr_`
3. Stack all kernel registers and return address {`r2-r12`, `lr`} on kernel stack.
4. Pop task `pc` as first word off `r0` (task `sp`) into kernel `lr` (restoring state for `movs`).
5. Restore `spsr` from `spsr_`.
6. Switch to system mode:
   (a) Restore user task `sp` from `sp_`.

(b) Unroll trap frame from task `sp` for registers {`r0-r12`, `lr`}. Note this is the task's `lr` and not the kernel's.

(c) Switch to supervisor mode `msr cpsr_c, #0xd3`.

7. Jump to userspace `movs pc, lr`.

From user space to kernel: Via `swi n` jumping to `KernelEnter` label

1. Switch to system mode:

   (a) Store user task registers {`r0-r12`, `lr`} on user stack. `pc`.

   (b) Move `sp` into `r0`. Note this is `sp_`.

2. Switch to supervisor mode.

3. Push `lr` (the task's `pc`) onto stack pointed by `r0` (task `sp`) and increment `r0`.

4. Move `spsr` into `r1`.

5. Pop kernel registers from kernel stack {`r2-r12`, `lr`}.

6. Restore `r0` (task `sp`) and

7. Restore `r1` (task `spsr`) into task descriptor.

2.1.1 Description in armlish

The piece of code responsible for context switch is:

```
register unsigned int *sp_  asm("r2") = active->sp;   // r2 <- sp
register unsigned int spsr_ asm("r3") = active->spsr; // r3 <- spsr
*(sp_ + 1) = first.ret;       // save ret on stack
unsigned int arg0, arg1;
asm volatile(

    "stmfd sp!, {r4-r12, lr}\n\t"  // save kregs on kstack
    "ldmfd %0!, {lr}\n\t"          // sp_ <- lr (the stored pc)
    "msr spsr, %1\n\t"              // spsr <- spsr_
    "msr cpsr_c, #0xdf\n\t"         // switch to system mode
    "mov sp, %0\n\t"                // sp <- sp_
    "ldmfd sp!, {r0-r12, lr}\n\t"  // pop task's registers
    "msr cpsr_c, #0xd3\n\t"         // switch to supervisor mode
    "movs pc, lr\n\t"              // jump to userspace
```

```
        "KernelEnter:\n\t"              // label (swi jumps here)
        "msr cpsr_c, #0xdf\n\t"         // switch to system mode
        "stmfd sp!, {r0-r12, lr}\n\t"  // store task registers
        "mov %0, sp\n\t"                // sp_ <- sp save task's sp
        "msr cpsr_c, #0xd3\n\t"         // switch to supervisor mode
        "stmfd %0!, {lr}\n\t"          // sp + 0 <- lr save task's pc to stack
        "mrs %1, spsr\n\t"              // spsr_ <- spsr save activity's spsr
        "ldmfd sp!, {r4-r12, r14}\n\t"  // unroll kregs from kstack

        "mov %2, r0\n\t"                // copy arg0
        "mov %3, r1\n\t"                // copy arg1


        : "+r"(sp_), "+r"(spsr_), "=r"(arg0), "=r"(arg1)  // output
        : // input
        : "r0", "r1"  // force asmblr not use any of these registers
    );
    active->sp = sp_;
    active->spsr = spsr_;
```

2.1.2 Trap Frame

When the user does a syscall, a trap frame is set up and calls the kernel. The trap frame is pushed on the calling task's stack, storing its state (registers). The layout of registers stored is:

```
[     ] <-- SP after storing trap frame
[ PC  ]
[ LR  ]
[ r12 ]
[ r11 ]
[ ... ]
[ r0  ] <-- SP at SWI instruction
[ ... ]
```

Initializing the trap frame and returning from a call is written in assembly code, and can be found in context_switch.s, or inlined with `asm` operand. On return, the result of the syscall is stored in register`r0` and execution resumes at the point where the syscall occurred.

2.2 Syscalls

Syscalls defined in C functions, in `syscall.{c,h}` files. They execute `swi n` where `n` is the syscall defined in the header.

2.3 Tasks

2.4 Scheduling

3 Source Code Location

Code is located under/u1/j53sun/cs452team/.

Compiling is as simple as runningmake, which will also copy the localkernel.elf to/u/cs452/tftp/ARM/cs452_05/kernel_k0.elf.

File md5sums

```
j53sun@ubuntu1204-002:~/cs452team$ md5sum *
99b14b7cdf8c75a76aa95a84baa99b7a  bwio.d
e4b6fb3bba23dbc7894365d089569084  bwio.s
7325710433d4c3f2ca2ef7d024b4090c  context_switch.h
1380c061df6d3fcf5ca731ca0870f12a  context_switch.s
e9ecc0c507565cc766ec637a9aec3ab6  cpsr.h
208557261c197d03bdc6b12d51bb17a6  doxygen.config 17928a24ca4436c15190a8e77b2608eb  kern
e87799ad275ab3fd1199dba2ea334e5c  linker.ld
bb9d37dae24e4e08a43483e8ca8e110f  Makefile
53fdea2ffa00ca6f9bbbea8f47d7b5ea  readme.md
7e3c4c63adb0b9be9ae4fa8f3677957c  scheduler.c
f80b164c2c64c33e7f0c3c4f053806b9  scheduler.h
620368217046ce3bd0f60b135af8f7c8  sftp-config.json d6bdf5714a8d499da29f1064973580ae  st
c4514d457ae97d5926b8267f8702aa8c  syscall.c
f9a2fd02c38d6487ecb03eedac299098  syscall.h
d3575bed4f5cacfc59d13e7ee2b3931d  task.h
e4a2610f5d6aba2f5d7b46a720e91668  ts7200.h
```

4 Program Output

```
RedBoot> go
Program completed with status 0
```

Explaination