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/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 modemsr cpsr_c, #0xd3.

7. Jump to userspace movs pc, lr.

From user space to kernel: Via swi n jumping toKernelEnter 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 issp_.

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. Restorer0 (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. 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/cs452team/`.

Compiling by running `make`, which also copies the local `kernel.elf` to`/u/cs452/tftp/ARM/j53sun/cs452t`

`File md5sums`

```
8da586d949d31e239dfbe0c8356588f6  bwio.c
db0bab80ffcef52c8ce1a968c65587a9  bwio.h
d02b490ecfa9f94ca03ccb1004f23efe  context_switch.h
1ba8cd1b57c22116b57e96d22022cec5  context_switch.s
e9ecc0c507565cc766ec637a9aec3ab6  cpsr.h
6bf72fad920c3d9e326401a101e31ac0  k1.lyx
524bf9b87a0c352c3f62a46c15618ca3  k1.pdf
8fa76c583dac06f80d4028cdee20d7c3  kernel.c
e87799ad275ab3fd1199dba2ea334e5c  linker.ld
7c1b255735fd098a6ecd2b8a8903a0d9  Makefile
53fdea2ffa00ca6f9bbbea8f47d7b5ea  readme.md
d8e9046f472dc425d1cb3f884e0c939e  scheduler.c
0954bdc95abf5a2dfe291e04b8dedb80  scheduler.h
d6bdf5714a8d499da29f1064973580ae  stdbool.h
d8382c43b39efd0f066522396ddb41b5  syscall.c
702846c6401a9fddeb232c1244e03511  syscall.h
8e4e2530209d10a7f2a7b9c450cafd08  task.c
11d63ce61f395dde30e120fd11ed183a  task.h
2c5fc627ac5386f1f96a65d2f8dc9d67  ts7200.h
1e7dca0aa66b495b8d4f1e89490b88f1  user_task.c
962dc8dab4c71088e86d7c7c6bb9adc8  user_task.h
```

4 Program Output

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

Explaination