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.

After a successful switch into kernel mode, the`n` is extracted by looking at the word before the task's`pc` (conveniently located at the top of the trap frame) and masking out the`swi` portion.

Argument passing is handled by explicitly placing parameters into`r0-r5`. For example, from the implementation of`Create()`:

```
int Create(int priority, void (*code)()) {
    // some error checking
    register unsigned int priority_in_ret_out asm("r0") = priority;
    register void (*code_in)() asm("r1") = code;
    // call swi n and return whats in r0 (priority_in_ret_out)
}
```

In this way, we explicitly tell gcc to place the parameters in registers we specify. On the kernel side, these parameters are extracted from the trap frame for`r0-r5` and handled from there.

Return values from syscalls are handled by explicitly manipulating the trap frame for the calling task. Replacing the value of`r0` in the trap frame with the return value puts it in the right place when the task is next scheduled.

2.3 Tasks

A task represents a unit of code that can be scheduled independent of other tasks in the system. In our kernel, a task is spawned based off a function pointer.

2.3.1 Task Descriptor

All tasks are tracked by a descriptor, which holds onto its task id,`sp`,`spsr`, and parent task id (the one who called`Create()`). The`sp` and`spsr` are manipulated by the context switch. There is a global table of tasks where descriptors are allocated from and a task id is merely an index into this table.

In its current form, the kernel only allocates enough memory to track 100 tasks.`Create()` calls after 100 tasks have been spawned fail with a return value of`-2`.

2.3.2 Prioritization and Scheduling

At create time, all tasks have a specified priority. This is tracked by queues, which in the current form, are 32 ringbuffers with a maximum buffer size of 16.

There is also a bitmask which tracks the queue population. If there is a runnable task at priority`p`, the`1 << p` bit of the bitmask will be set.

When a task needs to be scheduled, the queue with the highest priority and active tasks is found by efficiently computing the number of right leading zeroes in the bitmask[**?**]. The head of the queue is then rotated to be the tail and returned as the next task to be scheduled.

2.3.3 Task Creation

A task is created by specifying a priority, a function pointer, the number of words to reserve for it's stack, and the parent task id. It then creates an artificial trap frame and sets`pc` to the value of the function pointer (after compensating for a relocation of`0x200000`, the parameter to RedBoot's`load -b`) and the frame pointer (`r12`) to be equivalent to the`sp`. It then allocates a task descriptor and stores this information there. Finally, it adds the descriptor to the right task queue and sets the appropriate bit in the bitmask.

The`Create()` syscall is implemented by this function. Due the API constraints, the`Create()` syscall currently specifies a stack size of 1024 words, which will break horribly should the task need more. Currently, there exists no user callable syscall that allows specification of the stack size.

2.4 Task Deletion

`Exit()` is implemented by finding the populated queue with the highest priority and removing the task at the tail of the queue (since the calling task must exist at that position since we just scheduled it). Once removed from the priority queues, we guarantee that the task will never be scheduled again. However, no effort is made to reclaim task descriptors.

3 Source Code Location

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

Compiling is as simple as running`make`, which will also copy the local`kernel.elf` to`/u/cs452/tftp/ARM/cs452_05/kernel_k0.elf`.

`File md5sums`

```
324234234234234  bwio.c
```

4 Program Output

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

Explaination