ChCore Lab 1: Booting a machine

517021910851-于亚杰

Part A: Bootstrap

Exercise 1

> Read or at least carefully scan the Part A1 and A3 of the Arm® Instruction Set Reference Guide to be familiar with ARM ISA. Part D is a useful reference for ARM instructions. You can scan the section 1 and 2 in this guide to be familiar with gcc AArch64 grammar quickly. Please write down some differences compared with x86-64 you have learned before.

Some differences I've observed are:

- ARM uses a RISC architecture while X86 uses a CISC architecture
- load and store instructions are incorporated in X86
- X86 has smaller code sizes
- instruction length of X86 is variable while it's fixed in ARM
- ARM heavy use of ram
- X86-64 has fewer registers
- A “word” on x86 is 16-bits and a “doubleword” is 32-bits. A “word” for ARM is 32-bits and a “doubleword” is 64-bits.
- no branch prediction(ARM) vs. branch prediction(X86)

Exercise 2

> Use GDB's where command to trace the entry (first function) and its address of the boot loader. Where is this function defined? You can also use readelf -S build/kernel.img to read the symbol table.

```
(gdb) where
#0  0x0000000000080000 in _start ()
--------------------------------
> readelf -S build/kernel.img
 [ 1] init              PROGBITS         0000000000080000  00010000
       0000000000049680  0000000000000008 WAX       0     0     4096
```

So the function is defined in the init section of the ELF and its starting address is 0x80000.

PartB: The Boot Loader

Exercise 3

> Before we introduce multi-processors, we focus on a single processor. In Raspi3, all processors boot simultaneously. Please read the boot code in boot/start.S and explain how to hang other processors according to the control flow.

```
mrs     x8, mpidr_el1
```

The code first get the processor id, and store it in register x8.

```
mov     x9, #0xc1000000
bic     x8, x8, x9
cbz     x8, primary
```

Then use conditional branch to differentiate one processor from secondary processors.
If x8 AND not 0xc1000000 == 0, the code will switch to Primary part to continue booting which happens to only one processor. And other processors will go on to secondary_hang part and just hang there forever.

Exercise 4

> You have learned basic pointer operations in the ICS course. Then download the code for pointer-warmup.c, compile by gcc pointer-warmup.c, run ./a.out, and make sure you understand all of the outputs. How all the values in printed lines 2 to 4 get there, and why the values printed in line 5 are seemingly corrupted.

line 1: print address of the pointers and each variable has been allocated 8 bytes space
line 2: a[i] is assigned long value
line 3: *(c + 2) refers to c[2] and 3[c] refers to c[3]
line 4: c = c + 1; means the address of c is increased by 8 bytes
line 5: c = (long *) ((char *) c + 1); means the address of c is increased by 1 bytes and *c still is a long value occuping 8 bytes. And change the value of *c will change the first byte of a[2]. By the way big endian storage should also be mentioned here.
line 6: b = (long *) a + 1; address of b = address of a + 8 bytes; c = (long *) ((char *) a + 1); address of c = address of a + 1 byte;

Exercise 5

> Read the objdump information of build/kernel.img, the code of boot loader is stored in .init section, and codes of the kernel are stored in .text, explain the differences of their LMA and VMA. And try to explain why these differences happened.

`init` section has the same LMA and VMA while `.text` section has different ones. Since the boot_loader is started first, there's no need to extinguish the two addresses and it's faster to directly boot from the load address. And more importantly the kernel is run later so in order to save the RAM the code of the kernel can stay in ROM for a while and copied to the RAM when it's needed in run time. Another reason may be that before the kernel space is setup the code has to loaded in the user space, and then copied to kernel space.

Part C: The Kernel

Exercise 6

> tfp_printf and printk are the interface function, and they support format print similar to standard printf. While we omit the code to print numbers in different bases (e.g., 2, 8, 10, 16), please fill the functions write_num inboot/print/printf.c and printk_write_num in kernel/common/printk.c to implement this function. They have the same logic.

divide `n` by `base` until it comes to 0
store the remainder from the end of the array to the beginning, if neg is not zero, print '-' first
then call `write_string` with argument `str` referring to the position of the most significant digit of `n`

Exercise 7

> Where do codes of kernel stack initialization lay? Which function does the kernel initialize its stack? Where is its stack located in the memory? How does the kernel reserve space for its stack?

```
(gdb) x/g $sp
0xffffff00000d20f0 <kernel_stack+8192>: 0x0000000000000000
(gdb) p &kernel_stack
$1 = (char (*)[4][8192]) 0xffffff00000d00f0 <kernel_stack>
```

The codes of kernel stack initialization lay in `kernel/head.S` . function `start_kernel` will initialize its stack. The stack size is initialized to 8KB(KERNEL_STACK_SIZE), and the stack bottom address is `0xffffff00000d20f0` while the stack top address is `0xffffff00000d00f0` .

```
ldr     x2, =kernel_stack
add     x2, x2, KERNEL_STACK_SIZE
mov     sp, x2
```

kernel_stack+KERNEL_STACK_SIZE is assigned to the stack pointer and thus the space is reserved.

Exercise 8

> To become familiar with the C calling conventions on the AArch64, find the address of the test_backtrace function in kernel/main.c by gdb, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 64-bit words does each recursive nesting level of test_backtrace push on the stack, and what are those words?

```
(gdb) p &test_backtrace
$1 = (void (*)(long)) 0xffffff00000cc020 <test_backtrace>
```

The function address of `test_backtrace` is `0xffffff00000cc020`.

```
(gdb) x/10g $sp
0xffffff00000d2060 <kernel_stack+8048>: 0xffffff00000d2080    0xffffff00000cc070
0xffffff00000d2070 <kernel_stack+8064>: 0x0000000000000003    0xffffffffffffffff
```

Every time `test_backtrace` calls itself the stack is decreased by 32Bytes. So that is four 64-bit words pushed on the stack. And the four words in sequence are frame pointer(last function), return address(the address to return to), argument `x` and `0xffffffffffffffff` (this value is changed by function `printk_format` and are not used in function `test_backtrace`).

Exercise 9

> In AArch64, return address (x30), frame pointer (x29), and arguments are transmitted by registers. However, when a caller function calls a callee function, these registers should be stored in the stack, where do these values store?

At the beginning of the callee function, a `stp` instruction is executed to push the frame pointer(x29) and return address(x30) onto the stack. As is shown in the following code:

```
# x0 is the argument passed here
# push x30 x29
0xffffff00000cc020 <test_backtrace>     stp    x29, x30, [sp, #-32]!
# store a callee-saved register
0xffffff00000cc024 <test_backtrace+4>   mov    x29, sp
0xffffff00000cc028 <test_backtrace+8>   str    x19, [sp, #16]
```

To be more informative, x30 is pushed prior to x29. And the insturction above is not the common case since sp grows by 32 bytes while two registers combined are 16 bytes. So the insturction is

practically equivalent to: (1) sp = sp - 16 (2) push x30 (3) push x29.
The excessive bytes are left for function arguments.

```
# change the function to accept 5 arguments and they are shown to be pushed in the following way
(gdb) x/10xg $sp
0xffffff00000d1fe0 <kernel_stack+7920>: 0xffffff00000d2020      0xffffff00000cc0a0
0xffffff00000d1ff0 <kernel_stack+7936>: 0x0000000000000003      0x0000000000000011
0xffffff00000d2000 <kernel_stack+7952>: 0x0000000000000012      0x0000000000000013
0xffffff00000d2010 <kernel_stack+7968>: 0x0000000000000014      0xffffffffffffffff
```

The function arguments are pushed onto the stack(in reversed order) before the callee function begins. By the way, sp is 16 bytes-aligned so sometimes there could be a 8-bytes space waste.

Exercise 10

> Implement the backtrace function mon_monitor in kernel/monitor.c as specified above. Use the same format as in the example; otherwise, the grading script will be confused. Because of compiler optimization, you only need to consider the case of test_backtrace.

As is shown in the last exercise, the stack from bottom to top is like:
args(if #args is odd, add 0xffffffffffffffff for alignment intention), fp0, lr0, args, fp1, lr1.....
So by the help of `read_fp()`, it's easy to locate all the frames because *fp would refers to the position of the previous frame and the five arguments.
The last problem is about when to stop tracing. From `boot/start.S`, I found that the stack pointer is set to 0x1000+boot_cpu_stack, which is 0x88530, before calling `init_c`. And also the frame address of the outer-most function is actually 0x0. So I guess both ways are reasonable to stop tracing when *fp equals 0 or fp + 16 equals 0x88530.