CS-1217 Operating Systems

Spring 2024

Lab 1

March 4, 2024

Exercise 1

1. The

```
# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
```

in *boot.S* causes the switch from 16-bit to 32-bit mode.

2. For the last instruction of the boot loader executed,

   - in *main.c*:

```
// call the entry point from the ELF header
// note: does not return!
((void (*)(void)) (ELFHDR->e_entry))();
```

   - in *boot.asm*:

```
((void (*)(void)) (ELFHDR->e_entry))();
    7d6b: ff 15 18 00 01 00      call   *0x10018
```

   The first instruction to be loaded is:

```
f010000c:   66 c7 05 72 04 00 00    movw   $0x1234,0x472
```

3. Since the last instruction the boot loader executes is call *0x10018, the first instruction of the kernel should be at this address. When we look at this address using gdb, we get:

```
(gdb) x/1x 0x10018
0x10018:        0x0010000c
```

   Hence, the first instruction is at 0x0010000c

4. The boot loader initializes pointers ph and eph to point to the program headers in the ELF header. ph is set to point to the first program header by calculating its address relative to the start of the ELF header while eph is set to point to the end of the program headers by adding the number of program headers (e_phnum) to ph.

```
// load each program segment (ignores ph flags)
ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;
for (; ph < eph; ph++)
    // p_pa is the load address of this segment (as well
    // as the physical address)
    readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

Exercise 2

1. *console.c* exports cputchar, getchar, and iscons, while cputchar is used as a parameter when *printf.c* calls vprintfmt from *printfmt.c*

2. In the *console.c* file, it verifies if the cursor has reached the buffer's end, i.e. the screen is full. If it has, the buffer is scrolled up by one row, the last row is cleared with spaces, and the cursor is positioned at the start of the last row to make space for newer information.

3.
    - In the call to cprintf(), *fmt* points to the format string of its arguments while *ap* points to the variable arguments after *fmt*.
    - For this part, I modified the monitor.c and added the snippet to it, then ran gdb and got:

```
cprintf (fmt=0xf0101ad2 "x %d, y %x, z %d\n")
vcprintf (fmt=0xf0101ad2 "x %d, y %x, z %d\n", ap=0xf0115f64 "\001")
cons_putc (c=120)
cons_putc (c=32)
va_arg(*ap, int)
Hardware watchpoint 4: ap
Old value = 0xf0115f64 "\001"
New value = 0xf0115f68 "\003"
cons_putc (c=49)
cons_putc (c=44)
cons_putc (c=32)
cons_putc (c=121)
cons_putc (c=32)
va_arg(*ap, int)
Hardware watchpoint 4: ap
Old value = 0xf0115f68 "\003"
New value = 0xf0115f6c "\004"
cons_putc (c=51)
cons_putc (c=44)
cons_putc (c=32)
cons_putc (c=122)
cons_putc (c=32)
va_arg(*ap, int)
Hardware watchpoint 4: ap
Old value = 0xf0115f6c "\004"
```

```
New value = 0xf0115f70 "T\034\020?\214_\021??\027\020??_\021??\027\020?_\021?_\021?"
cons_putc (c=52)
cons_putc (c=10)
```

4. The output is He110 World, because 57616=0xe110, so the first half of output is He110 because 57616 is read in hexadecimal. i=0x00646c72 is treated as a string, so it will be printed as ‘r’=(char)0x72, ‘l’=(char)0x6c, ‘d’=(char)0x64, and 0x00 is treated as a mark of end of string.

   We will see He110, Wo in a big-endian machine. 57616 will still be read as e110 because only its numeric value matters when being printed. However, when i = 0x00646c72 is treated as the string, the 0x00 terminates the string at Wo

5. After ‘y=’, the decimal value of 4 bytes right above where 3 is placed in the stack will be printed.

6. GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. We can just store the number of arguments as an integer. To adapt <u>cprintf</u> for GCC's new calling convention, its interface would need to include the number of arguments being passed by modifying <u>cprintf</u> to accept a count of arguments along with a format string and a variable argument list. The caller would need to provide the count of the arguments while invoking the function.
   For example, the interface of <u>cprintf</u> could be updated to something like <u>cprintf(const char *fmt, int num_args, ...)</u>.

Exercise 3

Using *kernel.asm*, we can notice that the starting address for the test_backtrace is 0xf0100040. Using gdb, we get:

```
+ symbol-file kernel
(gdb) b *0xf0100040
Breakpoint 1 at 0xf0100040: file kern/init.c, line 13.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 1, test_backtrace (x=5) at kern/init.c:13
13      {
(gdb) i r
eax            0x0      0
ecx            0x3d4    980
edx            0x3d5    981
ebx            0xf0111308       -267316472
esp            0xf010ffdc       0xf010ffdc
ebp            0xf010fff8       0xf010fff8
esi            0x10094  65684
edi            0x0      0
eip            0xf0100040       0xf0100040 <test_backtrace>
eflags         0x46     [ PF ZF ]
cs             0x8      8
---Type <return> to continue, or q <return> to quit---
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x10     16
gs             0x10     16
(gdb) c
Continuing.
=> 0xf0100040 <test_backtrace>: push   %ebp
```

```
(gdb) c
Continuing.
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 1, test_backtrace (x=4) at kern/init.c:13
13      {
(gdb) i r
eax            0x4      4
ecx            0x3d4    980
edx            0x3d5    981
ebx            0xf0111308       -267316472
esp            0xf010ffbc       0xf010ffbc
ebp            0xf010ffd8       0xf010ffd8
esi            0x5      5
edi            0x0      0
eip            0xf0100040       0xf0100040 <test_backtrace>
eflags         0x92     [ AF SF ]
cs             0x8      8
---Type <return> to continue, or q <return> to quit---
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x10     16
gs             0x10     16
(gdb)
```

The difference of ebp between the two breakpoints is 0x20, so every time it pushes 8 4-byte words as follows:

return address
saved ebp
saved ebx
abandoned
abandoned
abandoned
abandoned
var x for calling next test_backtrace

The return instruction pointer typically points to the instruction after the call instruction (why?) → The return instruction pointer points to the instruction after the call instruction because it ensures sequential execution, allowing the CPU to resume executing code sequentially after a function call, and it aligns with function call conventions, where the return address is pushed onto the stack before jumping to the called function, simplifying the management of function calls and returns for efficient control flow.

Why can't the backtrace code detect how many arguments there actually are? How could this limitation be fixed? → The backtrace code cannot detect the number of arguments passed to a function because C does not provide built-in mechanisms to check the number of arguments at runtime. The number of arguments passed to a function is fixed and determined at compile time, making it impossible for the backtrace code to dynamically determine the number of arguments. We could use a different calling convention or function signature that includes information about the number of arguments. You could define a struct to hold both the number of arguments and the arguments themselves, passing this struct to the backtrace function, giving it the number of arguments and their values.