Lab 1: PC Bootstrap and GCC Calling Conventions

Sriram V - CS11B058

Exercise 1

- Read and understood the AT&T syntax from Brennan's Guide to Inline Assembly.
- Extended inline assembly code takes the form
  `asm ( "statements" : output_registers : input_registers : clobbered_registers);`
- We use the following register loading codes in this function:

```
a        eax
b        ebx
c        ecx
d        edx
S        esi
D        edi
I        constant value (0 to 31)
q,r      dynamically allocated register (see below)
g        eax, ebx, ecx, edx or variable in memory
A        eax and edx combined into a 64-bit integer (use long longs)
```

Exercise 2

- When the BIOS runs, it sets up an interrupt descriptor table and initializes various devices such as the VGA display.
- This is where the `Starting SeaBIOS` message in the QEMU window comes from.
- After initializing the PCI bus and all the important devices the BIOS knows about, it searches for a bootable device such as a floppy, hard drive, or CD-ROM.
- Eventually, when it finds a bootable disk, the BIOS reads the boot loader from the disk and transfers control to it.

Exercise 3

- We set a breakpoint at `0x7c00` (`b *0x7c00`). The boot sector will be loaded here.
- First the code is assembled for 16-bit mode (line `14` in `boot/boot.S - .code16`)
- Interrupts are cleared with `cli`.
- `%ds`, `%es` and `%ss` are all set to 0.
- A20 is then enabled.
- GDT is loaded at `line 48` in `boot/boot.S`
- `CR0` is set to enable Protected mode.
- We now jump to the next instruction, but in the 32-bit code segment. doing `si` in `gdb` tells us that `ljmp $0xb866,$0x87c32` does this. CS is set here (`$PROT_MODE_CSEG`).
- Lines `60` to `65` set the other registers in the GDT.
- Now the stack pointer is set up, and the call to `bootmain()` is made.
- `readseg()` is called to read the first page off the disk. A check is done if it is a valid ELF.
- ph is the program header segment. The for loop reads the segments one the total number of segments present (`e_phnum`)

- In `readseg()` the segments and offsets are translated into sector numbers and necessary round downs are made.
- Direct mapping to memory is done since paging isn't enabled.
- `readsect()` reads in data from the Hard Disk by using the sectors and the offsets.
- Once the for loop in `bootmain()` is done, it uses the function pointer (to the entry point of the kernel that has just been loaded) to step into the beginning of the kernel code.

Questions

1. **At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?**

- Before `line 55` in `boot/boot.S`, where a long jump to 32-bit code happens. The switch is due to setting `CR0`'s first bit to 1.

2. **What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?**

- Last instruction boot loader executes is the call to jump to the kernel's entry point (`0x100000c`). The instruction is located at `0x7d65` and we see that the jump is to the address stored at `0x10018`.
- First kernel instrucion is `movw $0x1234,0x472` (warm boot)

3. **Where is the first instruction in the kernel?**

- At `0x100000c` - the `.globl entry` in `kern/entry.S`.

4. **How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?**

- The ELF header specifies the offset for the program headers, and how many are there. The program headers contain the address of the program segment loaded into the ram, its size and the offset from the beginning of the file at which the first byte of the segment is present.

Exercise 4

- Read relevant portions of THe C Programming Language by Kernighan and Richie.
- An important concept here is that of pointer arithmetic. If `int *p = (int*)100`, then `(int)p + 1` would evaluate as 101, but `(int)(p + 1)` would be equal to 104.

Exercise 5

- The parameter `-Ttext` in `boot/Makefrag` is modified from `0x7C00` to `0x7F00`. The code is recompiled and `gdb` is restarted.
- The error occurs at `line 55` in `boot/boot.S` – The long jump into the 32-bit code segment.
- This happens because the `Ttext` parameter tells the assembler to link to the `0x7F00` address (ie., the code will execute from there), but the BIOS loads the code at `0x7C000` and the GDT data at `0x7c64`. But since the link address is `0x7f00`, at the `ljmp`, when the GDT is accessed, we end up looking for it at `0x7f64` instead.

Exercise 6

- When the BIOS enters the bootloader, using `x/8x 0x00100000` gives:

```
0x100000:   0x00000000  0x00000000  0x00000000  0x00000000
0x100010:   0x00000000  0x00000000  0x00000000  0x00000000
```

- When the bootloader enters the kernel, the same command gives:

```
0x100000:   0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0x100010:   0x34000004  0x0000b812  0x220f0011  0xc0200fd8
```

- They are different because when the BIOS enters the bootloader, nothing has been loaded here.
- However, the bootloader loads the kernel at `0x00100000` (The load address "LMA", when we do `objdump -h obj/kern/kernel`)
- Hence, we see values loaded into these addresses at the second breakpoint.

Exercise 7

- `movl %eax, %cr0` is the line that enables Paging.
- `x/8x 0x00100000` gives us:

```
0x100000:   0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0x100010:   0x34000004  0x0000b812  0x220f0011  0xc0200fd8
```

- `x/8x 0xf0100000` gives:

```
0xf0100000 <_start+4026531828>: 0xffffffff  0xffffffff  0xffffffff  0xffffffff
0xf0100010 <entry+4>:   0xffffffff  0xffffffff  0xffffffff  0xffffffff
```

- After single-stepping over the `movl` command, we see that the 8 words at `0x00100000` are the same, but now the addresses at `0xf0100000` are mapped to `0x00100000`. Hence doing `x/8x 0xf0100000` gives us:

```
0xf0100000 <_start+4026531828>: 0x1badb002  0x00000000  0xe4524ffe  0x7205c766
0xf0100010 <entry+4>:   0x34000004  0x0000b812  0x220f0011  0xc0200fd8
```

- If the `movl` line is commented out (ie., Paging is disabled) and the code is recompiled, then execution fails at `line 68` in `kern/entry.S` (`jmp *%eax`). This is because `relocated` is linked at the address `0xf010002f` ("VMA" is `0xf0100000` for the kernel), and this addreass is loaded into `%eax`.
- But since paging wasn't enabled, the mapping failed.

Exercise 8

- The following code is added at lines 209 - 211 to enable printing of octal numbers.

```
num = getuint(&ap, lflag);
base = 8;
goto number;
```

Questions

1. **Explain the interface between `printf.c` and `console.c`. Specifically, what function does `console.c` export? How is this function used by `printf.c`?**

- `printf.c` is able to access the `console.c` functions through their function declarations in `inc/stdio.h`.
- printf.c defines the high-level printing functions used by the kernel, while console.c defines low-level functions used to perform some kinds of console I/O, like keyboard input support, serial I/O, parellel port and display output.
- Specifically, `console.c` exports `cputchar()` that `printf.c` uses in `putch()`.

2. **Explain the following code from `console.c`:**

```
if (crt_pos >= CRT_SIZE) {
        int i;
        memcpy(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
        for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
                crt_buf[i] = 0x0700 | ' ';
        crt_pos -= CRT_COLS;
}
```

- When the console buffer is full, it means that all the lines on the screen are filled. Hence, for new lines to be printed, the screen must scroll down. This code copies the content after the first `CRT_COLS` onwards onto the display, thus removing the topmost row. It then blanks out the bottom-most row (the for loop goes to each column element in the last row). Then it sets the position of the next output to the first column element of the last, blanked-out row. This has resulted in a scrolling down of one line.

3. **Trace the code execution step-by-step**

   i. **In the call to cprintf(), to what does `fmt` point? To what does `ap` point?**

   - `fmt` points the address (`0xf0101a69`) of the string to be parsed:
     `"x %d, y %x, z %d\n"`
   - `ap` points to the addresss (`0xf010ffe4`) of the value of x - `0x00000001`

   ii. **List (in order of execution) each call to `cons_putc`, `va_arg`, and `vcprintf`. For `cons_putc`, list its argument as well. For `va_arg`, list what `ap` points to before and after the call. For `vcprintf` list the values of its two arguments.**

   - Order of calling: `vcprintf` > `va_arg` > `cons_putc`
   - The argument for cons_putc is the variable to be printed on the screen.
   - `ap` points to the address of the first argument in the argument list before the call, and to the next argument in the list after the call.
   - `fmt` points the address (`0xf0101a69`) of the string to be parsed: `"x %d, y %x, z %d\n"`
   - `ap` points to the addresss (`0xf010ffe4`) of the value of x - `0x00000001`

4. **Run the following code.**

```
unsigned int i = 0x00646c72;
cprintf("H%x Wo%s", 57616, &i);
```

What is the output? Explain how this output is arrived.

- The output is "He110 World"
- This is because 57616 in Decimal is e110 in Hex.

- Since x86 uses the little-endian system, the bytes are accessed from right to left. so `0x72`=‘r’ is printed first, followed by the next bytes `0x6c`=‘l’, `0x64`=‘d’ and `0x00`=‘\0’

5. **In the following code, what is going to be printed after y=? (note: the answer is not a specific value.) Why does this happen?**

```
cprintf("x=%d y=%d", 3);
```

- Since cprintf uses va_arg, ap will initially point to the address of the memory location holding 3, and for `y` ap points to the value on the stack, after 3, which will be some junk value.

6. **Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?**

- In this case we simply need to reverse the order. We start with the end of the list, and we keep decreasing the address instead of increasing it.

Exercise 9

- The kernel initializes its stack at `line 77` in `kern/entry.S` (`movl $(bootstacktop),%esp`).
- By checking `obj/kern/kernel.asm`, we see that `bootstacktop` takes the value `0xf0110000`. Hence, the stack is located at this location.
- The kernel reserves space for the stack at `line 93` in `kern/entry.S` (`.space KSTKSIZE`). `KSTKSIZE` is defined in `inc/memlayout.h` as `(8*PGSIZE)` with `PGSIZE` defined in `inc/mmu.h` as equal to `4096` bytes. Hence, we get 8*4096 bytes = 32768 bytes = 8192 stack frames, each having a size of 4 bytes.
- Since the stack grows downwards in an x86 system, the stack pointer is initialised to the top of of the stack(`bootstacktop`), and as values are pushed, the value of the stack pointer is reduced by 4.

Exercise 10

- `%ebp` gets pushed onto the stack.
- `%ebx` gets pushed onto the stack.
- `mov %ebx,0x4(%esp)` pushes `%ebx` onto the stack as a parameter (x) to `cprintf()`.
- `movl $0xf0101b60,(%esp)` pushes the hex value onto the stack, which is the string parameter to `cprintf()`.
- The `call` to `cprintf()` pushes the return address `0xf010005a` onto the stack.
- `mov %eax,(%esp)` pushes `%eax`, the value of `(x-1)` onto the stack.
- The `call` to `test_backtrace()` pushes the return address `0xf0100069` onto the stack.
- Hence for the `test_backtrace()` function, `%ebx`, the arguments, and `%eip` of the return address are pushed before the function is called, followed by the pushing in of `%ebp` when the function starts.

Exercise 11

- The following code is added at `line 63` in `kern/monitor.c`:

```
cprintf("Stack backtrace:\n");

int* ebp = (int*)read_ebp();
int eip, i;

while (ebp != 0) {
    eip = *(ebp+1);
    cprintf("ebp %08x  eip %08x  args ", ebp, eip);
```

```
        for(i = 1; i <= 5; i ++) {
            cprintf("%08x ", *(ebp+i+1));
        }
        cprintf("\n");
        ebp = (int*)(*ebp);
    }
```

- `%ebp` is the current value of `ebp`.
- The value at the 32-bit location above `%ebp` is the value of `eip`. Hence, pointer arithmetic is used to add 4 bytes to `%ebp` and `\*` is used to access the value at the location.
- Since the arguments are the values above `eip`, pointer arithmetic is once again used to add 8-24 bytes to acess the first 5 arguments.

Exercise 12

- The above code at `line 63` in `kern/monitor.c` is modified into:

```
struct Eipdebuginfo info;
cprintf("Stack backtrace:\n");

int* ebp = (int*)read_ebp();
int eip, i;

while (ebp != 0) {
    eip = *(ebp+1);
    cprintf("ebp %08x  eip %08x  args ", ebp, eip);
    for(i = 1; i <= 5; i ++) {
        cprintf("%08x ", *(ebp+i+1));
    }
    cprintf("\n");

    //debug info
    debuginfo_eip(eip, &info);
    cprintf("%s:%d: %.*s+%d\n", info.eip_file, info.eip_line,
            info.eip_fn_namelen, info.eip_fn_name, eip-info.eip_fn_addr);

    ebp = (int*)(*ebp);
}
```

- Here we have added code to display additional information using the `debuginfo_eip()` method.
- In `debuginfo_eip()` in `kern/kdebug.c` the following lines of code are added at `line 183`:

```
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
if(lline <= rline) {
    info->eip_line = stabs[lline].n_desc;
} else {
    return -1;
}
```

- This code enables us to get the line number of the line of code to which `eip` points.
- The stab structure and constants are defined in `inc/stab.h`, which is from where `debuginfo_eip()` gets the `__STAB_*` structure details from.
- The bootloader loads the stab data at `0x001022fc`, and links to it at `0xf01022fc` (The value of `stabs` in `debuginfo_eip()`), which is the address at which all the accesses to the symbol table are made.