CS1217

Neeraj Pandey (neeraj.pandey_ug21@ashoka.edu.in),
Raunak Basu (raunak.basu_ug21@ashoka.edu.in)

Exercise 3

(a) In the file boot.S, line with “orl $CR0_PE_ON, %eax” is making the processor start executing 32-bit code from 16-bit real mode.
The Instruction with “ ljmp $PROT_MODE_CSEG, $protcseg” pointer points to the 32-bit code.

```
# Switch from real to protected mode, using a bootstrap GDT
# and segment translation that makes virtual addresses
# identical to their physical addresses, so that the
# effective memory map does not change during the switch.
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0

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

(b) The last instruction in bootloader is (in /boot/main.c): `((void (*)(void)) (ELFHDR->e_entry))();`

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

The first instruction is in the /kern/entry.S file, movl $(RELOC(entry_pgdir)), %eax.

```
# Load the physical address of entry_pgdir into cr3.  entry_pgdir
# is defined in entrypgdir.c.
movl    $(RELOC(entry_pgdir)), %eax
```

(c) The first instruction of the Kernel is at 0x0010000c.

(d) Inside the main.c file in /boot/main.c, the bootloader loads the first page of the disk using an elf file (ELF header files) and using conditional statements it checks for if the file loaded is of elf file type and then it loops through every program which loads each part of the kernel into the memory.

Solution (4):
Code file edited

Solution (5):
Code file edited with address from 0x7c00 to 0x7c02 in makefrag.

Solution (6):
The memory 0x100000 is null as the program has not got to this address location yet and before the bootloader tries to do anything, the memory block 0x100000 is empty. Meanwhile the bootloader is loading with kernels entry functions, before the memory is filled with [02 b0 ad 1b 00 00 00 00]. The .text code segment will be loaded to the address space.

```
/* AT(...) gives the load address of this section, which tells
   the boot loader where to load the kernel in physical memory */
.text : AT(0x100000) {
    *(.text .stub .text.* .gnu.linkonce.t.*)
}

PROVIDE(etext = .); /* Define the 'etext' symbol to this value */
```

Solution 7:

Once we have stepped over the instructions, the address 0x00100000 and 0xf0100000 is pointing to the same location inside of the kernel code. This can be figured out by using the command x/20i 0xf0100000 which shows that they represent instructions from after entry into

kernel, and the former address was pointing to the start of the kernel and the latter was blank. Going through the kernel.asm file, we can say that the first insturction to fail could be “movl $0x0,%ebp”. This is because the comments shows that the changes made by enabling the paging have not taken effect from right after the paging is activated.

```
    # Clear the frame pointer register (EBP)
    # so that once we get into debugging C code,
    # stack backtraces will be terminated properly.
    movl    $0x0,%ebp            # nuke frame pointer
f010002f:   bd 00 00 00 00          mov    $0x0,%ebp
```

Solution 8:

(a) The interface between printf.c and console.c is cputprint(). When we look at the console.c, the file exports the line cputchar(int c). Printf.c file uses the cputchar parameter to call vprintfmt in the printfmt.c file. So, we can say that printf is responsible to output strings which needs to be printed and the file console.c is responsible for printing stuff on the hardware end and exports cputchar.

(b) This chunk of code informs that the console is of a 2D array and pushes the information to the next line, and whenever the crt pos is greater than the crt size, the console is full and we move everything to the next line in order to display the information.

(c) The fmt points to the address of (x %d, y %x, z %d) and ap points to the element at the starting index of the list.
With gdb, we can do the following:
- cprintf()
- vcprintf() call to the void vprintfmt
- vprintmt call to the pitch(int)

(d) The following code will give the output:
He110 World
Here, 57616 means (in hexadecimal) 0x110[Hell0], the value 0x00646c72 is in the form of a little endian which is a string. 0x72 is is character “r”, character “x” as 0x6c, character “d” as 0x64, and 0x00 as the end of the string.

(e) x=3, y=-267321412

(f) We will have to cprintf() will have to list the variables in the reverse order to get the correct output.

For instance, cprintf("%d%d",a, b,) will be cprintf(b, a, "%d%d").

Solution 9:

```
  # Clear the frame pointer register (EBP)
  # so that once we get into debugging C code,
  # stack backtraces will be terminated properly.
  movl    $0x0,%ebp           # nuke frame pointer

  # Set the stack pointer
  movl    $(bootstacktop),%esp
```

```
    # Clear the frame pointer register (EBP)
    # so that once we get into debugging C code,
    # stack backtraces will be terminated properly.
    movl    $0x0,%ebp           # nuke frame pointer
f010002f:   bd 00 00 00 00          mov    $0x0,%ebp

    # Set the stack pointer
    movl    $(bootstacktop),%esp
f0100034:   bc 00 00 11 f0          mov    $0xf0110000,%esp
```

So, we can conclude that the kernel stack starts at 0xf0116000 and as the stack builds up, stack pointer will point to bootstacktop.

Solution 10:
In kernel.asm, a bunch of backtrace commands exists, and we can use gdb to set breakpoints at addresses and identify the differences between the consecutive addresses. The difference is 0x20.

Solution 11:
Code file edited