Lab 1 Report

Team: Anonymous
CS14B023, Rahul Kejriwal
CS14B039, Bikash Gogoi

Exercise 1:

We read through provided materials.

Exercise 2:

We stepped through the first few instructions.
The BIOS initially starts execution at 0xffff0 and then jumps to 0xfe05b. It initializes ss to 0x0 and esp to 0x7000 thereby setting up a functional stack. It goes on to disable maskable interrupts, writes on NMI, enables A20 and reads real time clock. It then loads the IDT and GDT from 0xf68f8 and 0xf68f8 respectively. It then enables protected mode by setting the first bit of CR0 and doing a ljmpl to start the protected mode addressing. It then sets up the ds, es, ss to 0x10, i.e., the third descriptor in GDT. This segment descriptor has base 0x0.

Exercise 3:

The beginning of the for loop is

```
7d47:    39 f3                       cmp    %esi,%ebx
```

The end of the for loop is

```
7d5f:    eb e6                       jmp    7d47
```

Code that will run after for loop completes is

```
((void (*)(void)) (ELFHDR->e_entry))();
7d61:    ff 15 18 00 01 00      call   *0x10018
```

- The instruction

  ```
  ljmp $PROT_MODE_CSEG, $protcseg
  ```

  causes the switch from 16-bit to 32-bit mode. After this instruction processor starts executing 32-bit code, i.e., instruction

  ```
  movw   $PROT_MODE_DSEG, %ax
  ```

  onwards processor executes in 32-bit mode.
- The last instruction the bootloader executed is

  ```
  ((void (*)(void)) (ELFHDR->e_entry))();
  ```

The first instruction of the kernel loaded is

```
movw    $0x1234, 0x472
```

- Last instruction of the bootloader is a call:

```
call   *0x10018
```

and [0x10018] = 0x0010000c, so the first instruction of the kernel is at 0x0010000c.

- Bootloader finds the number of sector to load from the elf header and loads them by the following codes

```
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 4:

1. The first print statement outputs 3 different addresses that were allotted to a, b and c, let them be addr1, addr2, addr3.

2. The second print statement outputs **200**, **101**, **102** and **103** as the values of a[0], a[1], a[2], a[3]. c points to the first element of array a and hence, c[0] = *(c+0) = *c = *a = *(a+0) = a[0] can be used to modify to the first element of the array.

3. The third print statement outputs **200**, **300**, **301**, **302** as the values of a[0], a[1], a[2], a[3]. Here, c[1] = a[1] as shown earlier. Also, *(c+2) = *(a+2) = a[2] and 3[c] = *(3+c) = *(3+a) = a[3].

4. The fourth print statement outputs **200**, **400**, **301**, **302** as the values of a[0], a[1], a[2], a[3]. After `c = c+1`, c points to a[1].

5. The fifth print statement outputs **200**, **128144**, **256**, **302** as the values of a[0], a[1], a[2], a[3]**.**
   Here, `c = (int *) ((char *) c + 1)` makes c point to the next byte rather than the next int variable. Now, c points to the second byte of a[1]. 500 = 0x000001f4 and 400 = 0x00000190. After doing *c = 500, a[1] becomes 0x<u>0001f4</u>90 = 128144. And 301 = 0x0000012d and so a[2] becomes 0x00000<u>100</u> = 256 (the underlined portion is the value written by *c = 500).

6. The sixth print statement outputs addr1, addr1 + 0x4, addr1 + 0x1 as the values of a,b,c.
   Here, `b = (int *) a + 1` increments the address contained in b by 0x4 (pointer arithmetic done wrt size of base type, i.e., 0x4 bytes for int).
   Here, `c = (int *) ((char *) a + 1);` increments the address contained in b by 0x1 (pointer arithmetic done wrt size of base type, i.e., 0x1 bytes for char).

Exercise 5:

```
ljmp  $PROT_MODE_CSEG, $protcseg
```

is the instruction that will break if the link address is wrong.

Right code:

```
ljmp    $0xb866,$0x87c32
```

Wrong code:

```
ljmp    $0x8,$0x7032
```

(depends on the wrong value of link address)

On modifying the link address, the above instruction jumps to an incorrect address the execution of which causes triple fault (as the instruction could not be found) and the execution remains at the same eip address forever.

Exercise 6:

At the point BIOS enters bootloader:

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

At the point bootloader enters kernel:

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

They are different because nothing is loaded at those locations initially, but in the second case kernel is loaded from the location 0x100000 onwards (since 0xf0100000 onwards maps to 0x100000 onwards uptill 4 MB).

Exercise 7:

```
(gdb) x/x 0x100000
0x100000:    0x1badb002
(gdb) x/x 0xf0100000
0xf0100000 <_start+4026531828>:    0x00000000
(gdb) si
=> 0x100028:    mov      $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/x 0x100000
0x100000:    0x1badb002
(gdb) x/x 0xf0100000
0xf0100000 <_start+4026531828>:    0x1badb002
```

Initially paging is not enabled so the higher address 0xf0100000 contains 0x0 as there is nothing loaded in that location while 0x100000 contains some value as kernel is loaded there. But after

one step paging is enabled and 0xf0100000 is mapped to 0x100000, so both address contains the same value.

```
jmp    *%eax
```

is the first instruction that will not work the way it should be if the mapping weren't in place. This will jump to some different location.

Exercise 8:

The codes required to print octal numbers is:

```
case 'o':
      num = getuint(&ap, lflag);
      base = 8;
      goto number;
```

1. printf.c uses the function cputchar() defined in console.c, which is a high level function for console I/O. console.c exports cputchar(). This function is used by printf.c which makes a wrapper around this function to allow counting no of characters printed to the console.

2. This code checks if the current line on the console is full depending on the size of screen. If it is full, it removes the already printed portion from the buffer, pads the buffer if it becomes less than line size and finally resets the location where next char is to be printed to 0.

3. fmt points to the character array "x %d, y %x, z %d\n".
   ap points to the variable argument list va_list containing x, y, z.

   Order of Execution:

```
cprintf( "x %d, y %x, z %d\n", x, y, z )
        vcprintf(fmt, ap)              where  fmt = "x %d, y %x, z %d\n",    ap = x, y, z
                cons_putc('x')
                cons_putc(' ')
                va_arg()               before call ap points to x, after call points to y
                cons_putc('1')
                cons_putc(',')
                cons_putc(' ')
                cons_putc('y')
                cons_putc(' ')
                va_arg()               before call ap points to y, after call points to z
                cons_putc('3')
                cons_putc(',')
                cons_putc(' ')
```

```
cons_putc('z')
cons_putc(' ')
va_arg()                before call ap points to z,  points to junk afterwards
cons_putc('4')
cons_putc('\')
cons_putc('n')
```

4. The output is He110 World. The hexadecimal conversion of 57616 is e110, so first part of the output is He110. For second part the value of i (0x00646c72) is treated as string and as the hardware is little endian so lower significant bytes are stored at lower memory address, so it will print byte by byte from lower address to higher address. Since 0x72 = r, 0x6c = l, 0x64 = d, and 0x0 = null in ascii, so it will print rld and then stop as the null character is encountered. Hence final output is He110 World. In big endian the output will be He110 Wo as the first byte of 0x00646c72 will be 0x00, which is null. To get the same result the value of i have to be 0x726c6400 and we don't need to change 57616.

5. It will print a decimal value of the 4 bytes right above (pushed earlier) where 3 is placed in the stack. This happens because while calling a function, arguments are pushed to stack, and the cprintf does not know the count of parameters actually passed. So it instead reads additional parameter from stack whenever it finds a format specifier in the format string.

6. We push an integer at the end of arguments specifying the number of arguments. Then, in cprintf, we can pop the specified no of arguments, reverse them and push them back to the stack. Thus, without modifying the vprintfmt(), we would have achieved the goal.

Exercise 9:

The ebp value is set to 0x0 while esp value is set to 0xf0110000 (virtual address) in the entry.s file. The stack pointer, esp is initialized to the top(highest address 0xf0110000) of the space and the stack expands down. Note that at this point ss refers to third entry in GDT (setup by the bootloader) which has base 0x00000000 (physical address) and limit 0xffffffff (granularity was set to 1).

The stack space is reserved by placing it within the data section of the entry.s file:

```
.data
###################################################################
# boot stack
###################################################################
       .p2align     PGSHIFT             # force page alignment
       .globl       bootstack
bootstack:
       .space       KSTKSIZE
```

```
        .globl      bootstacktop
bootstacktop:
```

Exercise 10:

The test_backtrace function starts at 0xf0100040 (virtual address). Each recursive level of nesting pushes 8 32-bit words onto the stack. The first two words pushed are the value of the ebp for the calling function and the ebx value from the calling function. The next 4 words are left blank (by decreasing the value of esp). Then, the argument (= x-1) is pushed followed lastly by the return address for eip. A sample set of the 8 words are:

0xf010ff9c: 0xf0100068 0x00000003 0x00000004 0x00000000
0xf010ffac: 0x00000000 0x00000000 0x00000005 0xf010ffd8

Here, 0xf010ff9c is the current esp, 0xf0100068 is the return address for eip, 0x00000003 is the argument for recursive call of test_backtrace, 0x00000005 is the value of ebx pushed to stack, and 0xf010ffd8 is the ebp value from the calling function.

Exercise 11:

We have implemented the backtrace function code in the kern/monitor.c file.

```
int mon_backtrace(int argc, char **argv, struct Trapframe *tf){
      // Your code here.
      uint32_t ebp, esp;

      ebp = read_ebp();
      while(ebp != 0){
            int *p = (int*)ebp+1;
            cprintf("ebp %08x eip %08x args %08x %08x %08x %08x
%08x\n", ebp, *p, *(p+1), *(p+2), *(p+3), *(p+4), *(p+5));
            ebp = *(p-1);
      }
      return 0;
}
```

Exercise 12:

We modified the mon_backtrace function:

```
int mon_backtrace(int argc, char **argv, struct Trapframe *tf){
      // Your code here.
      uint32_t ebp, esp;

      ebp = read_ebp();
```

```
        while(ebp != 0){
                int *p = (int*)ebp+1;
                cprintf("ebp %08x eip %08x args %08x %08x %08x %08x %08x\n",
ebp, *p, *(p+1), *(p+2), *(p+3), *(p+4), *(p+5));
                struct Eipdebuginfo inf;
                int ret = debuginfo_eip((uintptr_t)*p, &inf);
                if(ret != -1){
                        cprintf("\t%s:%d: ", inf.eip_file, inf.eip_line);
                        int k;
                        for(k=0;k<inf.eip_fn_namelen;k++)
                                cprintf("%c", inf.eip_fn_name[k]);
                        cprintf("+%d\n", *p-(int)inf.eip_fn_addr);
                }
                ebp = *(p-1);
        }
        return 0;
}
```

and completed the implementation of debuginfo_eip() by adding lines:

```
int tl = lline, tr = rline;
stab_binsearch(stabs, &tl, &tr, N_SLINE, addr);
if(tr < tl)
        return -1;
info->eip_line = stabs[(tl + tr)/2].n_value;
```

We also added the mon_backtrace function to the array commands[] to make it available on the command line:

```
static struct Command commands[] = {
        { "help", "Display this list of commands", mon_help },
        { "kerninfo","Display information about the kernel",mon_kerninfo},
        { "mon_backtrace", "Display stack backtrace", mon_backtrace },
        { "backtrace", "Display stack backtrace", mon_backtrace },
};
```