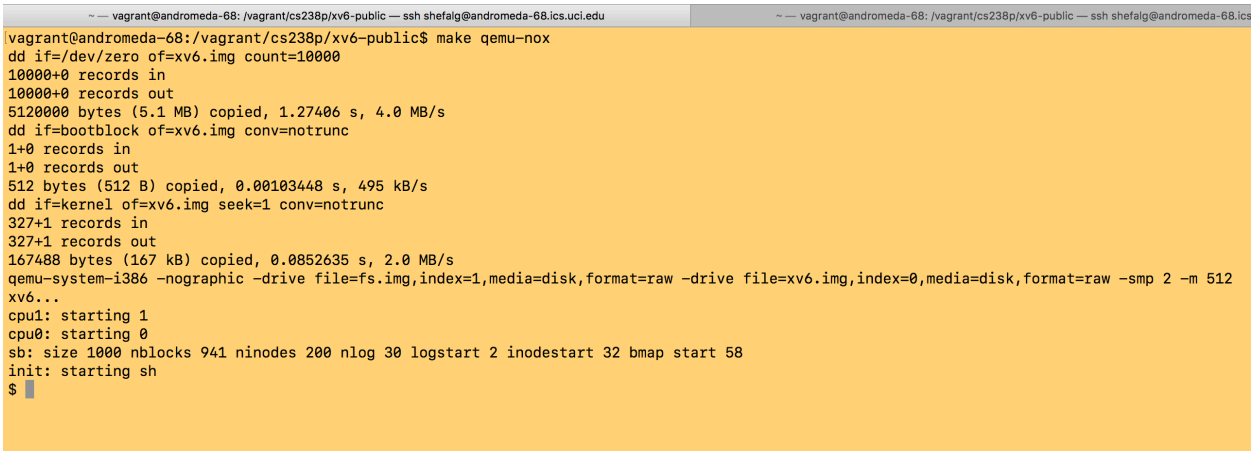
CS 238P | OPERATING SYSTEMS | Homework 2

[Name: Shefali Gupta Student ID: 57806943]

Exercise 1:

[GENERAL OBSERVATIONS]

To find start of kernel address, from one terminal run make qemu-nox



From another terminal, run: nm kernel | grep _start

```
vagrant@andromeda-68:/vagrant/cs238p/xv6-public$ nm kernel | grep _start
8010a48c D _binary_entryother_start
8010a460 D _binary_initcode_start
0010000c T _start
vagrant@andromeda-68:/vagrant/cs238p/xv6-public$
```

We found the address of kernel start as 0x10000c.

Inside vagrant, doing make qemu-nox-gdb.

```
~ — vagrant@andromeda-68: /vagrant/cs238p/xv6-public — ssh shefalg@andromeda-68.ics.uci.edu
~ — vagrant@andromeda-68: /vagrant/cs238p/xv6-public — ssh shefalg@andromeda-68.ics.uci.edu
vagrant@andromeda-68:/vagrant/cs238p/xv6-public$ make qemu-nox-gdb
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 1.19758 s, 4.3 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.001261 s, 406 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
327+1 records in
327+1 records out
167488 bytes (167 kB) copied, 0.0818023 s, 2.0 MB/s
*** Now run 'gdb'.
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512  -S -gdb tcp::260
01
```

Switching to another terminal and running gdb. We put a breakpoint on start address (0x10000c). Checking the registers.

```
Terminal  Shell  Edit  View  Window  Help
shefali — vagrant@andromeda-68: /vagrant/cs238p/xv6-publi
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>.
+ target remote localhost:26001
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb) br * 0010000c
Invalid number "0010000c".
(gdb) br * 0x10000c
Breakpoint 1 at 0x10000c
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x663    1635
ebx            0x0      0
esp            0x0      0x0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0xfff0   0xfff0
eflags         0x2      [ ]
cs             0xf000   61440
ss             0x0      0
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

We see that, only the edx, cd (code segment) and eip (instruction pointer) registers point to an address, to fetch and execute the very first instruction of xv6 booting, inside bootblock.asm

Also, the stack is initially empty.

```
(gdb) x/24x $esp
0x0:    0x00000000      0x00000000      0x00000000      0x00000000
0x10:   0x00000000      0x00000000      0x00000000      0x00000000
0x20:   0x00000000      0x00000000      0x00000000      0x00000000
0x30:   0x00000000      0x00000000      0x00000000      0x00000000
0x40:   0x00000000      0x00000000      0x00000000      0x00000000
0x50:   0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
```

Right before call to bootmain, the stack pointer register (esp) is initialized.

movl $start, %esp ; %esp= 0x7c00

Now putting a breakpoint on 0x7c00 (where call to bootmain() occurs) , and inspecting the state of registers for every instruction executed until we hit the breakpoint.

```
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb) br * 0x7c00
Breakpoint 1 at 0x7c00
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x663    1635
ebx            0x0      0
esp            0x0      0x0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0xfff0   0xfff0
eflags         0x2      [ ]
cs             0xf000   61440
ss             0x0      0
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) si
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x6ac8
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xfd2e1
0x0000e062 in ?? ()
(gdb) si
[f000:e066]    0xfe066: xor    %dx,%dx
0x0000e066 in ?? ()
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x663    1635
ebx            0x0      0
esp            0x0      0x0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0xe066   0xe066
eflags         0x46     [ PF ZF ]
cs             0xf000   61440
ss             0x0      0
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

```
gs             0x0      0
(gdb) si
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x6ac8
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xfd2e1
0x0000e062 in ?? ()
(gdb) si
[f000:e066]    0xfe066: xor    %dx,%dx
0x0000e066 in ?? ()
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x663    1635
ebx            0x0      0
esp            0x0      0x0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0xe066   0xe066
eflags         0x46     [ PF ZF ]
cs             0xf000   61440
ss             0x0      0
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) si
[f000:e068]    0xfe068: mov    %dx,%ss
0x0000e068 in ?? ()
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0x0      0x0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0xe068   0xe068
eflags         0x46     [ PF ZF ]
cs             0xf000   61440
ss             0x0      0
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

Before calling mov $0x7000, %esp, state of registers is as:

```
(gdb) si
[f000:e06a]    0xfe06a: mov    $0x7000,%esp
0x0000e06a in ?? ()
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0x0      0x0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0xe06a   0xe06a
eflags         0x46     [ PF ZF ]
cs             0xf000   61440
ss             0x0      0
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

After the instruction is executed, the Stack register (esp) points to an address location of 0x7000.

```
(gdb) info reg
eax            0x0        0
ecx            0x0        0
edx            0x0        0
ebx            0x0        0
esp            0x7000     0x7000
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0xe070     0xe070
eflags         0x46       [ PF ZF ]
cs             0xf000     61440
ss             0x0        0
ds             0x0        0
es             0x0        0
fs             0x0        0
gs             0x0        0
(gdb) $
```

Similary, before call to 0xf0574, state of registers and stack is as follows:

```
=> 0xf09ab:     call   0xf0574
0x000f09ab in ?? ()
(gdb) info reg
eax            0xf5b58  1006424
ecx            0x6fd4   28628
edx            0xf447b  1000571
ebx            0x0      0
esp            0x6fcc   0x6fcc
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0xf09ab  0xf09ab
eflags         0x2      [ ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x10     16
gs             0x10     16
(gdb) x/24x $esp
0x6fcc: 0x000f34da      0x000f447b      0x000f5b5c      0x00000000
0x6fdc: 0x00000000      0x00000000      0x00000000      0x00000000
0x6fec: 0x00000000      0x00000000      0x00000000      0x00000000
0x6ffc: 0x00000000      0x00000000      0x00000000      0x00000000
0x700c: 0x00000000      0x00000000      0x00000000      0x00000000
0x701c: 0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
```

Now, proceeding so on, right before bootmain is callled, at 0x7c00 (our breakpoint), state of registers and stack is:

```
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) info reg
eax            0xaa55   43605
ecx            0x0      0
edx            0x80     128
ebx            0x0      0
esp            0x6f20   0x6f20
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x7c00   0x7c00
eflags         0x202    [ IF ]
cs             0x0      0
ss             0x0      0
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) x/24x $esp
0x6f20: 0xf000d239      0x00000000      0x00006f62      0x00000000
0x6f30: 0x00008d0d      0x00008cd4      0x00000000      0x00000000
0x6f40: 0x00006ae5      0x00007c00      0x00007c00      0x00000080
0x6f50: 0x000f1c5d      0x000f3c9a      0x00000000      0x00007c00
0x6f60: 0x00000000      0x00000000      0x00000000      0x00000000
0x6f70: 0x00000000      0x00800000      0x00000000      0xaa550000
(gdb)
```

EIP register holds value of next instruction to be executed, which is 0x7c00. ESP points to 0x6f20.
Before calling bootmain at address 0x7d34, ESP points to 0x7c00

```
=> 0x7c48:      call   0x7d34
0x00007c48 in ?? ()
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x80     128
ebx            0x0      0
esp            0x7c00   0x7c00
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x7c48   0x7c48
eflags         0x6      [ PF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x0      0
gs             0x0      0
(gdb)
```

[ANSWERS TO QUESTIONS]

Question: Where in bootasm.S is the stack pointer initialized?

Answer:
In bootasm.S, right before call to bootmain, the stack pointer register (esp) is initialized.

movl $start, %esp ; // %esp= 0x7c00

```
# Set up the stack pointer and call into C.
movl    $start, %esp
call    bootmain
```

Question: Single step through the call to bootmain; what is on the stack now?
Answer: When the IP points to the bootmain subroutine, the content in the stack is now **0x7c4d,** which is the next instruction address.

(State of registers and top of stack before call to bootmain at 0x7d34)

```
=> 0x7c48:      call   0x7d34
0x00007c48 in ?? ()
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x80     128
ebx            0x0      0
esp            0x7c00   0x7c00
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x7c48   0x7c48
eflags         0x6      [ PF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x0      0
gs             0x0      0
(gdb) x/24x $esp
0x7c00: 0x8ec031fa      0x8ec08ed8      0xa864e4d0      0xb0fa7502
0x7c10: 0xe464e6d1      0x7502a864      0xe6dfb0fa      0x16010f60
0x7c20: 0x200f7c78      0xc88366c0      0xc0220f01      0x087c31ea
0x7c30: 0x10b86600      0x8ed88e00      0x66d08ec0      0x8e0000b8
0x7c40: 0xbce88ee0      0x00007c00      0x0000e7e8      0x00b86600
0x7c50: 0xc289668a      0xb866ef66      0xef668ae0      0x9066feeb
(gdb)
```

(State of registers and top of stack, right after call to bootmain)

```
(gdb) si
=> 0x7d34:      push   %ebp
0x00007d34 in ?? ()
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x80     128
ebx            0x0      0
esp            0x7bfc   0x7bfc
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x7d34   0x7d34
eflags         0x6      [ PF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x0      0
gs             0x0      0
(gdb) x/24x $esp
0x7bfc: 0x00007c4d      0x8ec031fa      0x8ec08ed8      0xa864e4d0
0x7c0c: 0xb0fa7502      0xe464e6d1      0x7502a864      0xe6dfb0fa
0x7c1c: 0x16010f60      0x200f7c78      0xc88366c0      0xc0220f01
0x7c2c: 0x087c31ea      0x10b86600      0x8ed88e00      0x66d08ec0
0x7c3c: 0x8e0000b8      0xbce88ee0      0x00007c00      0x0000e7e8
0x7c4c: 0x00b86600      0xc289668a      0xb866ef66      0xef668ae0
(gdb)
```

Question: What do first the assembly instructions of bootmain do to the stack?

Answer: push %ebp → saves the content of %ebp (0) to the top of the stack.

```
=> 0x7d34:      push   %ebp
0x00007d34 in ?? ()
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x80     128
ebx            0x0      0
esp            0x7bfc   0x7bfc
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x7d34   0x7d34
eflags         0x6      [ PF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x0      0
gs             0x0      0
(gdb) x/24x $esp
0x7bfc: 0x00007c4d      0x8ec031fa      0x8ec08ed8      0xa864e4d0
0x7c0c: 0xb0fa7502      0xe464e6d1      0x7502a864      0xe6dfb0fa
0x7c1c: 0x16010f60      0x200f7c78      0xc88366c0      0xc0220f01
0x7c2c: 0x087c31ea      0x10b86600      0x8ed88e00      0x66d08ec0
0x7c3c: 0x8e0000b8      0xbce88ee0      0x00007c00      0x0000e7e8
0x7c4c: 0x00b86600      0xc289668a      0xb866ef66      0xef668ae0
(gdb) si
=> 0x7d35:      mov    %esp,%ebp
0x00007d35 in ?? ()
(gdb) x/24x $esp
0x7bf8: 0x00000000      0x00007c4d      0x8ec031fa      0x8ec08ed8
0x7c08: 0xa864e4d0      0xb0fa7502      0xe464e6d1      0x7502a864
0x7c18: 0xe6dfb0fa      0x16010f60      0x200f7c78      0xc88366c0
0x7c28: 0xc0220f01      0x087c31ea      0x10b86600      0x8ed88e00
0x7c38: 0x66d08ec0      0x8e0000b8      0xbce88ee0      0x00007c00
0x7c48: 0x0000e7e8      0x00b86600      0xc289668a      0xb866ef66
```

Question: Look in bootmain in bootblock.asm for the call that changes eip to 0x10000c. What does the call do to the stack?

Answer: From bootblock.asm, we find that kernel entry() function is called **at 0x7dbc**. The compiled assembly instructions show that entry address of kernel was saved at the **address 0x10018**. We put a breakpoint at 0x7dbc and execute all instructions, until we hit this breakpoint.State of registers and stack before the call is:

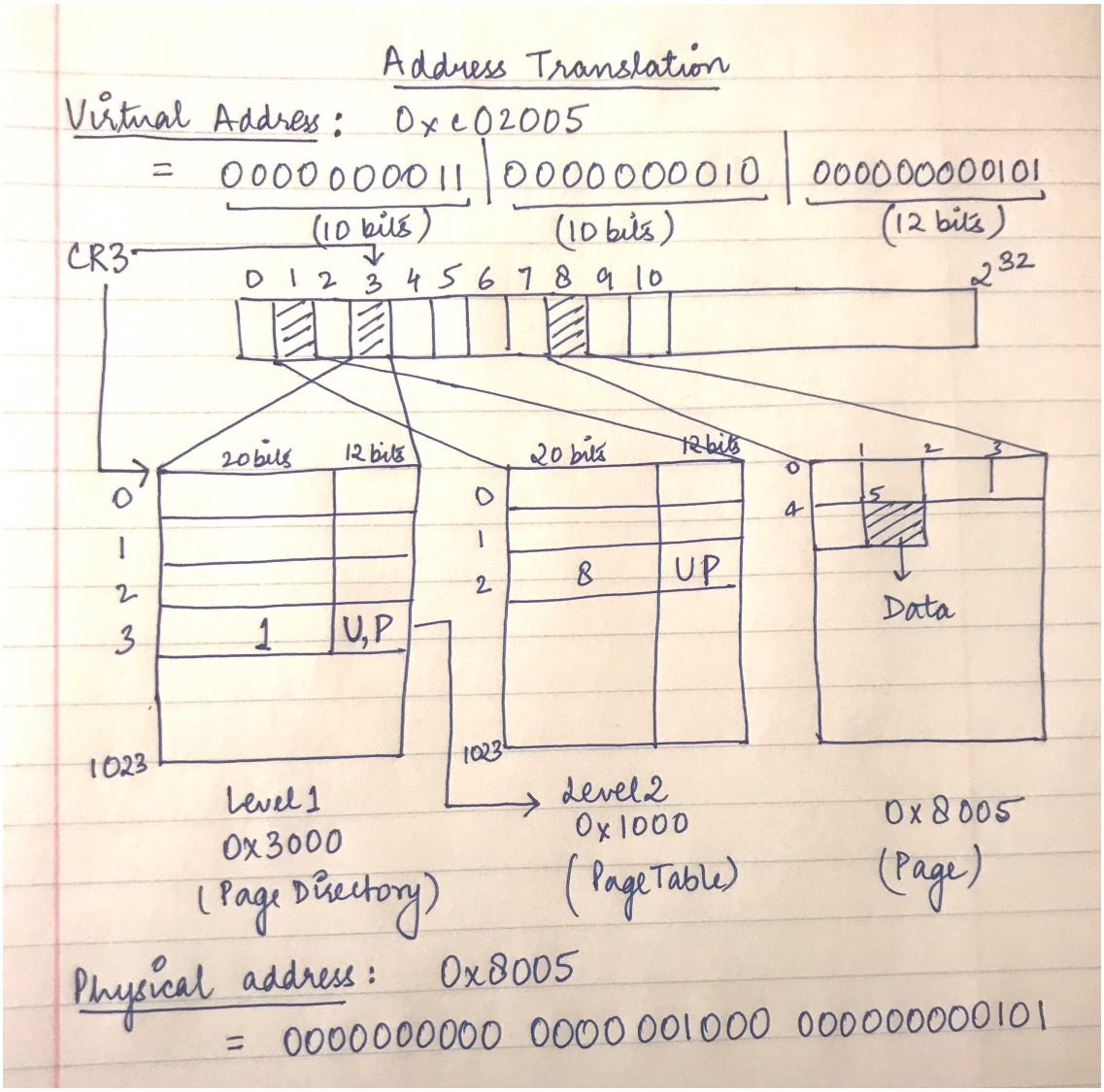
```
(gdb) br * 0x7dbc
Breakpoint 2 at 0x7dbc
(gdb) c
Continuing.
=> 0x7dbc:      call   *0x10018

Breakpoint 2, 0x00007dbc in ?? ()
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x1f0    496
ebx            0x10074  65652
esp            0x7bc0   0x7bc0
ebp            0x7bf8   0x7bf8
esi            0x0      0
edi            0x1154a8 1135784
eip            0x7dbc   0x7dbc
eflags         0x46     [ PF ZF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x0      0
gs             0x0      0
(gdb) x/24x $esp
0x7bc0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7bd0: 0x00000000      0x00000000      0x00000000      0x00010074
0x7be0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7bf0: 0x00000000      0x00000000      0x00000000      0x00007c4d
0x7c00: 0x8ec031fa      0x8ec08ed8      0xa864e4d0      0xb0fa7502
0x7c10: 0xe464e6d1      0x7502a864      0xe6dfb0fa      0x16010f60
(gdb)
```

After call,

```
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) info reg
eax            0x0      0
ecx            0x0      0
edx            0x1f0    496
ebx            0x10074  65652
esp            0x7bbc   0x7bbc
ebp            0x7bf8   0x7bf8
esi            0x0      0
edi            0x1154a8 1135784
eip            0x10000c 0x10000c
eflags         0x46     [ PF ZF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x0      0
gs             0x0      0
(gdb) x/24x $esp
0x7bbc: 0x00007dc2      0x00000000      0x00000000      0x00000000
0x7bcc: 0x00000000      0x00000000      0x00000000      0x00000000
0x7bdc: 0x00010074      0x00000000      0x00000000      0x00000000
0x7bec: 0x00000000      0x00000000      0x00000000      0x00000000
0x7bfc: 0x00007c4d      0x8ec031fa      0x8ec08ed8      0xa864e4d0
0x7c0c: 0xb0fa7502      0xe464e6d1      0x7502a864      0xe6dfb0fa
(gdb)
```

Here, we see that the call to 0x10018 jumps to kernel code address 0x10000c. The return address of C function call entry() is also saved on stack (0x00007dc2).

Exercise 2:



During the translation of virtual address to physical address, the virtual address of 32 bits is split into 10,10 and 12 bits. The first 10 bits indicate the index in level 1 page table (Page directory), which in this example is 3 in the physical memory address 0x3000. Since, level 2 page table table address is given as (0x1000), this means entry at 3 in PDE is 1. The second chunk of 10 bits in the virtual address, give the index in the level 2 page table, where address of final page table is present. In this case, it is 2. Now, since we are given final page in physical memory as having address 0x8005, this means entry at index 2 in level 2 page table, points to page at index 8 in the physical memory. The last 12 bits of the virtual address(5 in this case), indicate the index in the physical page, where the data is present.

Exercise 3:

Before kvmalloc() is called:

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[00000-003ff]  PDE[000]     --S-A---WP 00000-003ff
[80000-803ff]  PDE[200]     --S-A---WP 00000-003ff
(qemu)
```

We set a breakpoint in main and hit next, until kvmalloc() is called.

```
(gdb) b main
Breakpoint 1 at 0x80102e70: file main.c, line 19.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x80102e70 <main>:   push   %ebp

Breakpoint 1, main () at main.c:19
19      {
(gdb) n
=> 0x80102e7a <main+10>:        movl   $0x80400000,0x4(%esp)
20        kinit1(end, P2V(4*1024*1024)); // phys page allocator
(gdb) n
=> 0x80102e8e <main+30>:        call   0x80106b30 <kvmalloc>
21        kvmalloc();      // kernel page table
(gdb) n
=> 0x80102e93 <main+35>:        call   0x80103000 <mpinit>
22        mpinit();        // detect other processors
(gdb)
```

Snapshot of Page table after kvmalloc() is called:

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[80000-803ff]  PDE[200]     ----A--UWP
  [80000-800ff]  PTE[000-0ff] --------WP 00000-000ff
  [80100-80101]  PTE[100-101] ---------P 00100-00101
  [80102-80102]  PTE[102]     ----A----P 00102
  [80103-80105]  PTE[103-105] ---------P 00103-00105
  [80106-80106]  PTE[106]     ----A----P 00106
  [80107-80107]  PTE[107]     ---------P 00107
  [80108-8010a]  PTE[108-10a] --------WP 00108-0010a
  [8010b-8010b]  PTE[10b]     ----A---WP 0010b
  [8010c-803ff]  PTE[10c-3ff] --------WP 0010c-003ff
[80400-8dfff]  PDE[201-237] -------UWP
  [80400-8dfff]  PTE[000-3ff] --------WP 00400-0dfff
[fe000-fffff]  PDE[3f8-3ff] -------UWP
  [fe000-fffff]  PTE[000-3ff] --------WP fe000-fffff
(qemu)
```

main calls kvmalloc() to create and switch to a page table with the mappings above KERNBASE required for the kernel to run. After paging is enabled, kernel will start executing from virtual address space.

- 3 PDEs are present in Virtual address range 80000-803ff, 80400-8dffff and fe000-fffff.
- All PDEs are User, Writable and Present (as suggested by flags).
- First PDE is accessed. Other two PDEs are not accessed.
- 80000-803ff is level 1 PDE, which contains 1024 PTEs.
- 80400-8dfff contains PDEs in range 201-237 which map physical memory uptil PHYSTOP.
- fe000-fffff contains PDEs that map page tables for I/O devices.