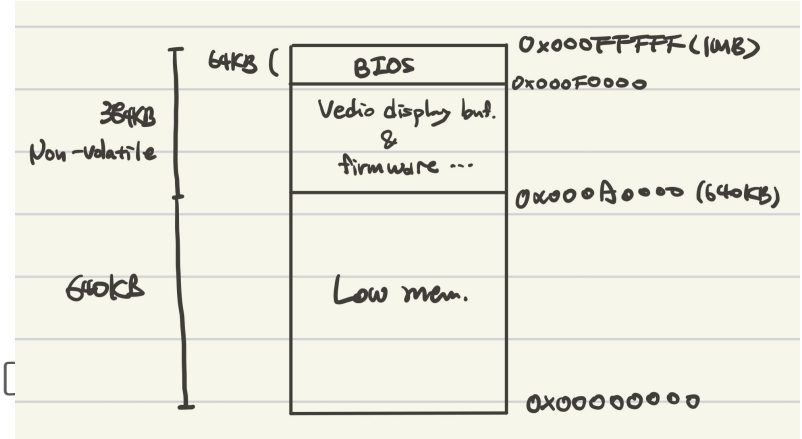
Lab0. Tutorial

1. Physical Address Space



The 1st PC

- 16-bit Intel 8088 processor
- 1MB의 physical memory만 사용할 수 있었다
- 640KB Low memory
  - 당시 PC가 쓸 수 있었던 RAM area (640KB)
  - 16KB, 32KB, 64KB RAM만 가능했다.
- 384KB : reserved by HW (Video display buffers, firmware 등을 위해..)
  - BIOS (Basic Input Output System)
    - jobs: 1. basic system initialization, 2.loading OS 3. pass control of the machine to OS
    - It is stored in

Early PC	Current PC
ROM(Read Only Memory)	updateable flash memory

1. Physical Address Space

After Intel “broke the one megabyte barrier”

hole : RAM 영역을 “Low memory”와 “Extended Memory”로 구분

- PC architects preserves the original layout for the low 1MB of physical address (640KB RAM area)
- in order to ensure backward compatibility with existing software (호환성 유지)

x86 processor가 4GB 이상의 RAM을 지원하기 시작

- 1MB 위로 확장 가능
- 2nd hole이 생기는 이유?

1. Physical Address Space

ROM BIOS

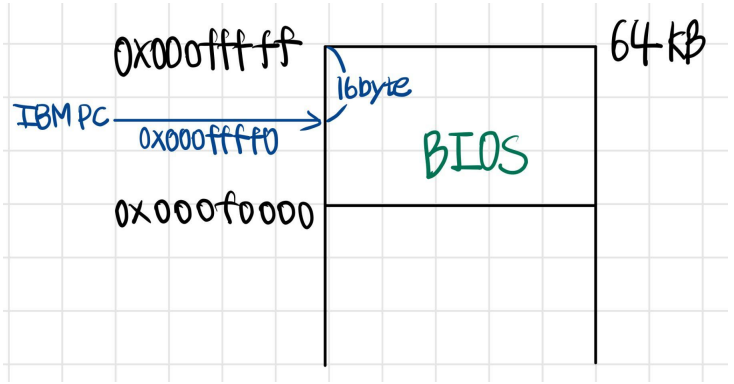
QEMU: virtual machine software

- To boot an OS, we need to hardware environment.
- QEMU allows us to get our own private hardware to boot an OS on.

1. Physical Address Space

ROM BIOS

- IBM PC는 physical addr. 0x000ffff0에서 executing을 시작한다.
  - 0x000ffff0: ROM BIOS를 위해 reserve된 맨 위의 64KB area에 위치.
- PC는 CS = 0xf000, IP = 0xfff0에서 execution을 시작한다.
- jmp instruction이 가장 처음 실행된다.
  - jump to CS=0xf000, IP=0xe05b -> **Q.**
- Physical address = 16 * segment + offset
  - segmented address CS:IP = 0xf000:fff0
  - 16 * 0xf000 + 0xfff0 = 0xf0000 + 0xfff0 = 0xffff0



1. Physical Address Space

Exercise 1.

첫 instruction (jump to 0xf000:205b)이 끝난 후

```
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d416
0x0000e062 in ?? ()
(gdb) si
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb) si
[f000:e068]    0xfe068: mov    %edx,%ss
0x0000e068 in ?? ()
(gdb) si
[f000:e06a]    0xfe06a: mov    $0x7000,%sp
0x0000e06a in ?? ()
(gdb) si
[f000:e070]    0xfe070: mov    $0x2d4e,%dx
0x0000e070 in ?? ()
(gdb) si
[f000:e076]    0xfe076: jmp    0x5575ff02
0x0000e076 in ?? ()
```

BIOS가 run할 때,

1. setup interrupt descriptor table
2. initialize the PCI bus and some important devices
3. then search for bootable device (floppy, hard drive, CD-ROM)
4. >> read boot roader and transfers control to it

2. The Boot Loader

sector: disk's minimum transfer unit, 512 bytes

- 읽기, 쓰기 연산이 하나 이상의 sector 크기로 이루어진다

boot sector: bootable disk의 첫 번째 sector

- boot loader의 코드가 들어있음

BIOS가 bootable disk를 찾으면,

1. 첫 번째 sector인 boot sector를 메모리에 load한다. (0x7c00 ~ 0x7dff)
2. jmp to CS:IP=0000:7c00
3. control of the machine을 boot loader에게 넘겨준다

2. The Boot Loader

PC의 발전으로 인해 CD-ROM으로부터 boot할 수 있게 됐다.

boot from	Floppy or hard disk	CD-ROM
sector size	512 bytes	2048 bytes

>> 하나의 sector만 가져오는 것이 아니라, 더 큰 boot image를 memory로 load할 수 있게 되었다. (control을 넘겨주기 전에 boot image를 가져올 수 있게 됨)

2. The Boot Loader

Our Boot Loader

- conventional hard drive boot mechanism
- bootasm.S (assembly source file) & bootmain.c (C source file)

2 Main Functions of Boot Loader

1. 프로세서를 real mode 에서 32-bit protected mode로 전환
    a. real mode: physical addr.를 사용하는 모드. PC가 boot할 때, 몇 개의 instruction을 real mode에서 실행한다.
    b. 32-bit protected mode
        i. 이 모드에서만 SW가 (physical) 1MB 위의 메모리에 접근할 수 있다.
        ii. semented address를 physical address로 변환하는 과정이 다르다.
        iii. offset이 16-bit 대신 32-bit이다.
2. x86의 특정 I/O instruction을 통해 IDE disk device register에 직접 접근해서 hard disk로부터 커널을 읽어온다.

Exercise 2

- At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
- What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?
- *Where* is the first instruction of the kernel?
- 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?

Exercise 2

Q1. the switch point from 16- to 32-bit mode

```
//PAGEBREAK!
  # Complete the transition to 32-bit protected mode by using a long jmp
  # to reload %cs and %eip.  The segment descriptors are set up with no
  # translation, so that the mapping is still the identity mapping.
  ljmp    $(SEG_KCODE<<3), $start32
```

Exercise 2

Q2 ~ Q3 The last instruction of boot loader : call entry()

```
(gdb) x/40i 0x7d3b
   0x7d3b:      push   %ebp
   0x7d3c:      mov    %esp,%ebp
   0x7d3e:      push   %edi
   0x7d3f:      push   %esi
   0x7d40:      push   %ebx
   0x7d41:      sub    $0xc,%esp
   0x7d44:      push   $0x0
   0x7d46:      push   $0x1000
   0x7d4b:      push   $0x10000
   0x7d50:      call   0x7cf8
   0x7d55:      add    $0xc,%esp
   0x7d58:      cmpl   $0x464c457f,0x10000
   0x7d62:      je     0x7d6c
   0x7d64:      lea    -0xc(%ebp),%esp
   0x7d67:      pop    %ebx
   0x7d68:      pop    %esi
   0x7d69:      pop    %edi
   0x7d6a:      pop    %ebp
   0x7d6b:      ret
```

```
0x7d6c:      mov    0x1001c,%eax
0x7d71:      lea    0x10000(%eax),%ebx
0x7d77:      movzwl 0x1002c,%esi
0x7d7e:      shl    $0x5,%esi
0x7d81:      add    %ebx,%esi
0x7d83:      cmp    %esi,%ebx
0x7d85:      jb     0x7d96
0x7d87:      call   *0x10018
0x7d8d:      jmp    0x7d64
0x7d8f:      add    $0x20,%ebx
0x7d92:      cmp    %ebx,%esi
0x7d94:      jbe    0x7d87
0x7d96:      mov    0xc(%ebx),%edi
0x7d99:      pushl  0x4(%ebx)
0x7d9c:      pushl  0x10(%ebx)
0x7d9f:      push   %edi
0x7da0:      call   0x7cf8
0x7da5:      mov    0x14(%ebx),%ecx
0x7da8:      mov    0x10(%ebx),%eax
0x7dab:      add    $0xc,%esp
0x7dae:      cmp    %eax,%ecx
```

readseg() in for()

call entry()

Exercise 2

Q3 ~ Q4

```
(gdb) x/40i *0x10018
   0x0: push   %ebx
   0x1: incl   (%eax)
   0x3: lock push %ebx
   0x5: incl   (%eax)
   0x7: lock ret
   0x9: loop   0xb
   0xb: lock push %ebx
   0xd: incl   (%eax)
   0xf: lock push %ebx
   0x11:        incl   (%eax)
   0x13:        lock push %esp
   0x15:        incl   (%eax)
   0x17:        lock push %ebx
   0x19:        incl   (%eax)
   0x1b:        lock push %ebx
   0x1d:        incl   (%eax)
   0x1f:        lock movsl %ds:(%esi),%es:(%edi)
   0x21:        incb   (%eax)
   0x23:        lock xchg %ebp,%ecx
   0x26:        add    %dh,%al
   0x28:        push   %ds
   0x29:        xlat   %ds:(%ebx)
```

1st instruction of kernel

```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
for(; ph < eph; ph++){
  pa = (uchar*)ph->paddr;
  readseg(pa, ph->filesz, ph->off);
  if(ph->memsz > ph->filesz)
    stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
}
```

^ Q.

phnum: the number of entries in the program header table.
phoff: pointer of the start of the program header table.

2. The Boot Loader

Loading The Kernel

ELF format : Executable Linkable Format

- C source code들을 컴파일러가 object file로 컴파일하면, 링커가 그 컴파일된 object file들을 하나의 binary image로 만드는데, 이 때 ELF format의 binary로 만든다.
- loading information을 담은 header로도 이해할 수 있다.
- ELF header의 뒤에는 메모리의 특정 주소로 load될 코드나 데이터인 program section이 따라온다.
- Boot loader는 load 되는 데이터나 코드를 수정 없이 메모리에 로드하고 실행을 시작한다.

https://refspecs.linuxfoundation.org/elf/elf.pdf

2. The Boot Loader

Loading The Kernel

ELF binary :

- fixed length ELF header
- variable length program header
  - load될 program section의 list를 담고 있다.
- program section:
  - .text: 프로그램의 executable instructions
  - .rodata : Read-only data (ex. C 컴파일러에 의해 만들어지는 ASCII string constants)
  - .data : initialized data (ex. 초기화된 global variable: int x = 5)
  - .bss: uninitialized global variable
    - linker가 프로그램의 memory layout을 구성할 때 uninitialized global var.을 위한 공간을 .bss에 reserve한다. 이때 .bss section의 주소와 크기를 기록한다 (contents를 저장할 필요는 X)
    - loader나 프로그램 자체가 해당 global var.들에 0을 할당한다.
    - .data가 바로 뒤에 따라온다. (in memory)

2. The Boot Loader

objdump -h kernel

Load address : 그 섹션이 load되어야 하는 메모리 주소

Link address : 그 섹션이 실행될 메모리 주소

not loaded into memory by the program loader

Q. LMA of .text
: 0010000
>> Right after the end of BIOS?

```
tjdp99@fuzzer-master:~/xv6$ objdump -h kernel

kernel:     file format elf32-i386

Sections:                                  link addr.   load addr.
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00006ea2  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000009ec  80106ec0  00106ec0  00007ec0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000af88  8010a520  0010a520  0000b516  2**5
                  ALLOC
  4 .debug_line   000025e8  00000000  00000000  0000b516  2**0
                  CONTENTS, READONLY, DEBUGGING
  5 .debug_info   0001051b  00000000  00000000  0000dafe  2**0
                  CONTENTS, READONLY, DEBUGGING
  6 .debug_abbrev 00003946  00000000  00000000  0001e019  2**0
                  CONTENTS, READONLY, DEBUGGING
  7 .debug_aranges 000003a8  00000000  00000000  00021960  2**3
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_str    00000e5e  00000000  00000000  00021d08  2**0
                  CONTENTS, READONLY, DEBUGGING
  9 .debug_loc    00005281  00000000  00000000  00022b66  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_ranges 00000700  00000000  00000000  00027de7  2**0
                  CONTENTS, READONLY, DEBUGGING
 11 .comment      00000029  00000000  00000000  000284e7  2**0
                  CONTENTS, READONLY
```

2. The Boot Loader

- Link address와 Load address는 보통 같다.
- The boot loader는 ELF program header 를 사용하여 section을 어떻게 load할지 결정한다
- program header는 ELF object 의 어느 부분을 어느 주소에 load 할지 특정한다

ph->paddr : 읽어온 program section들이 저장될 physical addr.들을 담고 있다.

```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
for(; ph < eph; ph++){
  pa = (uchar*)ph->paddr;
  readseg(pa, ph->filesz, ph->off);
  if(ph->memsz > ph->filesz)
    stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
}
```

2. The Boot Loader

objdump -x kernel

```
kernel:     file format elf32-i386
kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

Program Header:
    LOAD off    0x00001000 vaddr 0x80100000 paddr 0x00100000 align 2**12
         filesz 0x000078ac memsz 0x000078ac flags r-x
    LOAD off    0x00009000 vaddr 0x80108000 paddr 0x00108000 align 2**12
         filesz 0x00002516 memsz 0x0000d4a8 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rwx
```

memory에 load 되어야 하는 object의 영역

Exercise 3

Boot sector의 load addr.와 link addr. = 0x7c00

0x7c00

```
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/40i 0x7c00
=> 0x7c00:      cli
   0x7c01:      xor    %eax,%eax
   0x7c03:      mov    %eax,%ds
   0x7c05:      mov    %eax,%es
   0x7c07:      mov    %eax,%ss
   0x7c09:      in     $0x64,%al
   0x7c0b:      test   $0x2,%al
   0x7c0d:      jne    0x7c09
   0x7c0f:      mov    $0xd1,%al
   0x7c11:      out    %al,$0x64
   0x7c13:      in     $0x64,%al
   0x7c15:      test   $0x2,%al
   0x7c17:      jne    0x7c13
   0x7c19:      mov    $0xdf,%al
   0x7c1b:      out    %al,$0x60
```

0x7c01

Q

```
=> 0x7c00:      xchg   %ax,%ax
   0x7c02:      nop
   0x7c03:      cli
   0x7c04:      xor    %eax,%eax
   0x7c06:      mov    %eax,%ds
   0x7c08:      mov    %eax,%es
   0x7c0a:      mov    %eax,%ss
   0x7c0c:      in     $0x64,%al
   0x7c0e:      test   $0x2,%al
   0x7c10:      jne    0x7c0c
   0x7c12:      mov    $0xd1,%al
   0x7c14:      out    %al,$0x64
   0x7c16:      in     $0x64,%al
   0x7c18:      test   $0x2,%al
   0x7c1a:      jne    0x7c16
   0x7c1c:      mov    $0xdf,%al
   0x7c1e:      out    %al,$0x60
   0x7c20:      lgdtl  (%esi)
```

Exercise 3

0x7c00

```
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/40i 0x7c00
=> 0x7c00:      cli
   0x7c01:      xor    %eax,%eax
   0x7c03:      mov    %eax,%ds
   0x7c05:      mov    %eax,%es
   0x7c07:      mov    %eax,%ss
   0x7c09:      in     $0x64,%al
   0x7c0b:      test   $0x2,%al
   0x7c0d:      jne    0x7c09
   0x7c0f:      mov    $0xd1,%al
   0x7c11:      out    %al,$0x64
   0x7c13:      in     $0x64,%al
   0x7c15:      test   $0x2,%al
   0x7c17:      jne    0x7c13
   0x7c19:      mov    $0xdf,%al
   0x7c1b:      out    %al,$0x60
```

0x7cf0

Q

```
(gdb) c
Continuing.
```

```
tjdp99@fuzzer-master:~/xv6$ objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001c0  00007cf0  00007cf0  00000074  2**2
```

2. The Boot Loader

- Boot loader와 달리, kernel의 load addr.와 link addr.는 다르다
  - kernel은 low addr.에 load되고, high addr.에서 실행된다 (link addr.)

- e_entry : 프로그램의 entry point의 link addr.
  - 프로그램이 실행되어야 하는 text section의 주소 : 0x0010000c **Q. 0x00100000?**

```
tjdp99@fuzzer-master:~/xv6$ objdump -f kernel

kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

Exercise 4

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00 >> The point the BIOS enters the boot loader
(gdb) b *0x0010000c
Breakpoint 2 at 0x10000c >> Boot loader enters the kernel
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/20x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
0x100020:       0x00000000      0x00000000      0x00000000      0x00000000
0x100030:       0x00000000      0x00000000      0x00000000      0x00000000
0x100040:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    mov    %cr4,%eax
```

Boot loader가 load한 kernel image

```
Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/20x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
0x100020:       0x0100000d      0xc0220f80      0x10b5c0bc      0x2ea0b880
0x100030:       0xe0ff8010      0x90669066      0x90669066      0x90669066
0x100040:       0x53e58955      0x10b5f4bb      0x0cec8380      0x106ec068
```

```
tjdp99@fuzzer-master:~/xv

kernel:     file format e
architecture: i386, flags
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

3. The Kernel

kernel.ld

```
/* Simple linker script for the JOS kernel.
   See the GNU ld 'info' manual ("info ld") to learn the syntax. */

OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)

SECTIONS
{
        /* Link the kernel at this address: "." means the current address */
        /* Must be equal to KERNLINK */
        . = 0x80100000;

        .text : AT(0x100000) {
                *(.text .stub .text.* .gnu.linkonce.t.*)
        }

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

3. The Kernel

Using virtual memory to work around position dependence

OS kernel을 high virtual addr. (ex. 0xf0100000, 0x80100000)에 link하는 이유는, 유저 프로그램이 사용할 프로세서의 virtual addr.를 남겨두기 위해서이다.

많은 machine들은 addr. 0xf0100000에 대해 physical memory를 가지고 있지 않다.

>> 프로세서의 memory management HW를 사용해, physical addr. 0x00100000을 mapping하는 virtual addr. 0xf0100000을 사용한다.

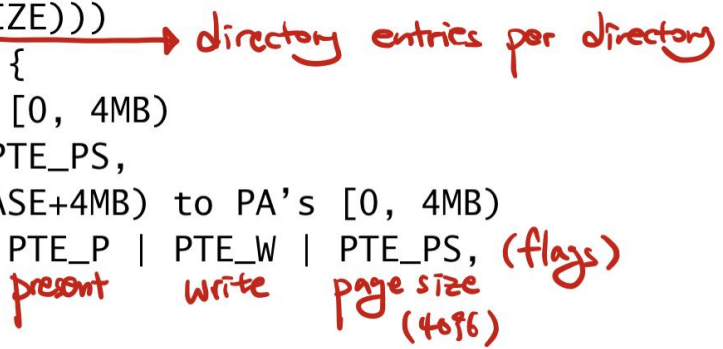
- 0xf0100000 : link addr. (커널 코드가 실행될 곳)
- 0x00100000 : load addr. (boot loader가 실제로 커널을 올리는 physical mem. (RAM의 1MB 지점 = ROM BIOS의 바로 위)
  - 0x0 이 아니라 0x100000인 이유 : 0xa0000:0x100000이 I/O device를 포함하기 때문

>> 따라서 PC가 적어도 1메가 이상의 physical mem.를 가지고 있어야 한다.

3. The Kernel

- 나중에는 physical memory의 256MB(0x00000000~0x0fffffff)를 mapping할 것이지만, 지금은 우선 4MB를 mapping할 것이다.
  - main.c의 1411~1417 라인에 있는 entrypgdir을 사용

```
1411 __attribute__((__aligned__(PGSIZE)))
1412 pde_t entrypgdir[NPDENTRIES] = {
1413   // Map VA's [0, 4MB) to PA's [0, 4MB)
1414   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1415   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1416   [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1417 };
1418
```



3. The Kernel

entry.S가 CR0_PG flag를 설정할 때 까지 memory reference는 linear addr.를 사용한다. (physical addr.와 같다. - boot.S가 linear addr.에서 physical addr.까지의 identity mapping을 해 주기 때문)

CR0_PG가 설정되면, VM HW에 의해 physical addr.로 변환되는 virtual addr.를 사용한다.

entry_pgdir이 virtual addr.를 physical addr.로 변환해준다

- 0xf0000000 ~ 0xf0400000 >> 0x00000000 ~ 0x00400000
- 0x00000000 ~ 0x00400000 >> 0x00000000 ~ 0x00400000

이 범위에 없는 virtual addr.들은 HW exception을 일으킨다

- interrupt handling이 아직 설정되지 않았기 때문에, QEMU가 machine state를 dump하고 끝낸다.

Exercise 5

```
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb) x/8x 0x80100000
0x80100000 <multiboot_header>:  0x00000000      0x00000000      0x00000000  0x00000000
0x80100010 <entry+4>:   0x00000000      0x00000000      0x00000000      0x00000000
(gdb) si
=> 0x100028:    mov    $0x8010b5c0,%esp
0x00100028 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb) x/8x 0x80100000
0x80100000 <multiboot_header>:  0x1badb002      0x00000000      0xe4524ffe  0x83e0200f
0x80100010 <entry+4>:   0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
```

Exercise 5

After comment out `movl %eax, %cr0`

```
(gdb) si
=> 0x100025:    mov    $0x8010b5c0,%esp
0x00100025 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb) x/8x 0x80100000
0x80100000 <multiboot_header>:  0x00000000      0x00000000      0x00000000      0x000000
00
0x80100010 <entry+4>:   0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
```

3. The Kernel

Formatted Printing to the Console

```c
#include "types.h"
#include "stat.h"
#include "user.h"

static void
putc(int fd, char c)
{
  write(fd, &c, 1);
}

static void
printint(int fd, int xx, int base, int sgn)
{
  static char digits[] = "0123456789ABCDEF";
  char buf[16];
  int i, neg;
  uint x;

  neg = 0;
  if(sgn && xx < 0){
    neg = 1;
    x = -xx;
  } else {
    x = xx;
  }

  i = 0;
  do{
    buf[i++] = digits[x % base];
  }while((x /= base) != 0);
  if(neg)
    buf[i++] = '-';

  while(--i >= 0)
    putc(fd, buf[i]);
}
```

```c
// Print to the given fd. Only understands %d, %x, %p, %s.
void
printf(int fd, const char *fmt, ...)
{
  char *s;
  int c, i, state;
  uint *ap;

  state = 0;
  ap = (uint*)(void*)&fmt + 1;
  for(i = 0; fmt[i]; i++){
    c = fmt[i] & 0xff;
    if(state == 0){
      if(c == '%'){
        state = '%';
      } else {
        putc(fd, c);
      }
    } else if(state == '%'){
      if(c == 'd'){
        printint(fd, *ap, 10, 1);
        ap++;
      } else if(c == 'x' || c == 'p'){
        printint(fd, *ap, 16, 0);
        ap++;
      } else if(c == 's'){
        s = (char*)*ap;
        ap++;
        if(s == 0)
          s = "(null)";
        while(*s != 0){
          putc(fd, *s);
          s++;
        }
      } else if(c == 'c'){
        putc(fd, *ap);
        ap++;
      } else if(c == '%'){
        putc(fd, c);
      } else {
        // Unknown % sequence.  Print it to draw attention.
        putc(fd, '%');
        putc(fd, c);
      }
      state = 0;
    }
  }
}
```

3. The Kernel

```
7918 static int panicked = 0;
7919
7920 static struct {
7921   struct spinlock lock;
7922   int locking;
7923 } cons;
7924
7925 static void
7926 printint(int xx, int base, int sign)
7927 {
7928   static char digits[] = "0123456789abcdef";
7929   char buf[16];
7930   int i;
7931   uint x;
7932
7933   if(sign && (sign = xx < 0))
7934     x = -xx;
7935   else
7936     x = xx;
7937
7938   i = 0;
7939   do{
7940     buf[i++] = digits[x % base];
7941   }while((x /= base) != 0);
7942
7943   if(sign)
7944     buf[i++] = '-';
7945
7946   while(--i >= 0)
7947     consputc(buf[i]);
7948 }
7949
```

ex. (300, 10, 1)

buf [0 0 3 -]

→ -300

```
7950 // Print to the console. only understands %d, %x, %p, %s.
7951 void
7952 cprintf(char *fmt, ...)
7953 {
7954   int i, c, locking;
7955   uint *argp;
7956   char *s;
7957
7958   locking = cons.locking;
7959   if(locking)
7960     acquire(&cons.lock);
7961
7962   if (fmt == 0)
7963     panic("null fmt");
7964
7965   argp = (uint*)(void*)(&fmt + 1);
7966   for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
7967     if(c != '%'){
7968       consputc(c);
7969       continue;
7970     }
7971     c = fmt[++i] & 0xff;
7972     if(c == 0)
7973       break;
7974     switch(c){
7975     case 'd':
7976       printint(*argp++, 10, 1);
7977       break;
7978     case 'x':
7979     case 'p':
7980       printint(*argp++, 16, 0);
7981       break;
7982     case 's':
7983       if((s = (char*)*argp++) == 0)
7984         s = "(null)";
7985       for(; *s; s++)
7986         consputc(*s);
7987       break;
7988     case '%':
7989       consputc('%');
7990       break;
7991     default:
7992       // Print unknown % sequence to draw attention.
7993       consputc('%');
7994       consputc(c);
7995       break;
7996     }
7997   }
```

1byte씩

% 다음 문자.

(NULL char.)

10진수로 print

16진수

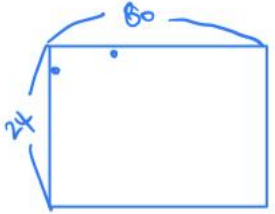
string 출력

% 출력.

error.

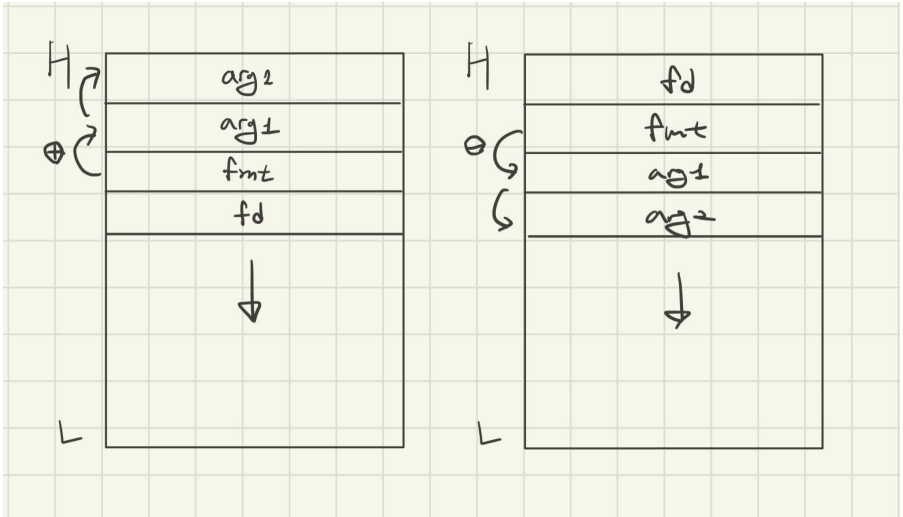
```
8004 void
8005 panic(char *s)
8006 {
8007   int i;
8008   uint pcs[10];
8009
8010   cli();
8011   cons.locking = 0;
8012   cprintf("cpu with apicid %d: panic: ", cpu->apicid);
8013   cprintf(s);
8014   cprintf("\n");
8015   getcallerpcs(&s, pcs);
8016   for(i=0; i<10; i++)
8017     cprintf(" %p", pcs[i]);
8018   panicked = 1; // freeze other CPU
8019   for(;;)
8020     ;
8021 }
```

```
8050 #define BACKSPACE 0x100
8051 #define CRTPORT 0x3d4
8052 static ushort *crt = (ushort*)P2V(0xb8000);  // CGA memory
8053
8054 static void
8055 cgaputc(int c)
8056 {
8057   int pos;
8058
8059   // Cursor position: col + 80*row.
8060   outb(CRTPORT, 14);
8061   pos = inb(CRTPORT+1) << 8;
8062   outb(CRTPORT, 15);
8063   pos |= inb(CRTPORT+1);
8064
8065   if(c == '\n')
8066     pos += 80 - pos%80;
8067   else if(c == BACKSPACE){
8068     if(pos > 0) --pos;
8069   } else
8070     crt[pos++] = (c&0xff) | 0x0700;  // black on white
8071
8072   if(pos < 0 || pos > 25*80)
8073     panic("pos under/overflow");
8074
8075   if((pos/80) >= 24){  // Scroll up.
8076     memmove(crt, crt+80, sizeof(crt[0])*23*80);
8077     pos -= 80;
8078     memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
8079   }
8080
8081   outb(CRTPORT, 14);
8082   outb(CRTPORT+1, pos>>8);
8083   outb(CRTPORT, 15);
8084   outb(CRTPORT+1, pos);
8085   crt[pos] = ' ' | 0x0700;
8086 }
```



```
8100 void
8101 consputc(int c)
8102 {
8103   if(panicked){
8104     cli();
8105     for(;;)
8106       ;
8107   }
8108
8109   if(c == BACKSPACE){
8110     uartputc('\b'); uartputc(' '); uartputc('\b');
8111   } else
8112     uartputc(c);
8113   cgaputc(c);
8114 }
```

Calling convention



첫 번 째 argument	`ap = (uint*)(void*)&fmt + 1;`	ap = (uint*)(void*)&fmt - 1;
다음 argument	`ap++;`	ap -- ;

Exercise 6

Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which “end” of this reserved area is the stack pointer initialized to point to?

After jump to entry point from entry() in bootmain.c,

Through setting cr0 which is control register, CR0_PG means paging is enabled

```
#define CR0_WP          0x00010000
#define CR0_PG          0x80000000
```

```
#define NPROC        64  // maximum number
#define KSTACKSIZE 4096  // size of per-pro
```

```
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

  # Set up the stack pointer.
  movl $(stack + KSTACKSIZE), %esp

  # Jump to main(), and switch to executing at
  # high addresses. The indirect call is needed because
  # the assembler produces a PC-relative instruction
  # for a direct jump.
  mov $main, %eax
  jmp *%eax

.comm stack, KSTACKSIZE
```

```
(gdb) x/20i 0x0010000c
=> 0x10000c:    mov    %cr4,%eax
   0x10000f:    or     $0x10,%eax
   0x100012:    mov    %eax,%cr4
   0x100015:    mov    $0x109000,%eax
   0x10001a:    mov    %eax,%cr3
   0x10001d:    mov    %cr0,%eax
   0x100020:    or     $0x80010000,%eax
   0x100025:    mov    %eax,%cr0
   0x100028:    mov    $0x8010b5c0,%esp
   0x10002d:    mov    $0x80102ea0,%eax
   0x100032:    jmp    *%eax
   0x100034:    xchg   %ax,%ax
   0x100036:    xchg   %ax,%ax
   0x100038:    xchg   %ax,%ax
   0x10003a:    xchg   %ax,%ax
   0x10003c:    xchg   %ax,%ax
   0x10003e:    xchg   %ax,%ax
   0x100040:    push   %ebp
   0x100041:    mov    %esp,%ebp
   0x100043:    push   %ebx
```

- CR0_PG를 CR0에 넣으면 Paging HW를 enable한다
  - 그러나 여전히 low addr.에서 실행하고 있는 상태이다.
- 커널을 High mem.에 mapping하기 위해
  - esp (stack pointer)를 initialize한다 (movl $(stack + KSTACKSIZE), %esp
  - high addr.의 커널 스택이 valid
- main으로 jump한다.
  - Q. p.23 “indirect jump”가 필요하다

+) Stack

- esp : stack pointer, 현재 사용되는 stack의 가장 낮은 지점을 point
- ebp : base pointer
  - SW convention을 따르는 register.
  - C 함수의 entry에서, 함수의 prologue code가 직전 함수의 base pointer(ebp)를 스택에 저장(push)
  - 현재의 esp를 ebp에 copy
  - >> stack에 저장된 ebp를 사용해 nested call을 backtrace할 수 있다!

Assembly trap handlers (p.42)

int instruction : switch the processor from user mode to kernel mode by generating trap

xv6 는 x86 hardware가 int instruction을 만났을 때 handling할 수 있게 set up 한다

tvinit(3317): x86 hardware가 256 interrupts를 가지고 있어서, 256개의 entry를 가진 IDT(interrupt descriptor table)를 set up한다