HW2

- Exercise 1: Finding and breaking at an addr.

Question 1: What is on the stack?

For xv6 with the address of _start 0x10000c (obtained by setting a break point). "info reg" at _start point in gdb command get:

Registers	Hex Value	Value	Explanation
eax	0x0	0	Last function's return value
ecx	0x0	0	Last function's return value
edx	0x1f0	496	Last used data
ebx	0x10074	65652	Last used value for source basic address
esp	0x7bcc	0x7bcc	Stack Pointer
ebp	0x7bf8	0x7bf8	Stack Frame Bottom Pointer
esi	0x10074	65652	Last used value for source change address
edi	0x0	0	Last function's return value is 0 and is stored in %edi
eip	0x10000c	0x10000c	Instruction pointer of next execution
eflags	0x46	[PF ZF]	Flags during calculation, PF means there're even number of 1s in the (binary) result, ZF means the calculation results in 0
cs	0x8	8	Code segment, set to 8 while booting
ss	0x10	16	stack segment

ds	0x10	16	data segment
es	0x10	16	extra segment
fs	0x0	0	flag segment
gs	0x0	0	global segment

“x/24x $esp” at _start point in gdb command get:

```
(gdb) x/24x $esp
0x7bcc: 0x00007db7      0x00000000      0x00000000      0x00000000
0x7bdc: 0x00000000      0x00000000      0x00000000      0x00000000
0x7bec: 0x00000000      0x00000000      0x00000000      0x00000000
0x7bfc: 0x00007c4d      0x8ec031fa      0x8ec08ed8      0xa864e4d0
0x7c0c: 0xb0fa7502      0xe464e6d1      0x7502a864      0xe6dfb0fa
0x7c1c: 0x16010f60      0x200f7c78      0xc88366c0      0xc0220f01
```

and explained as below:

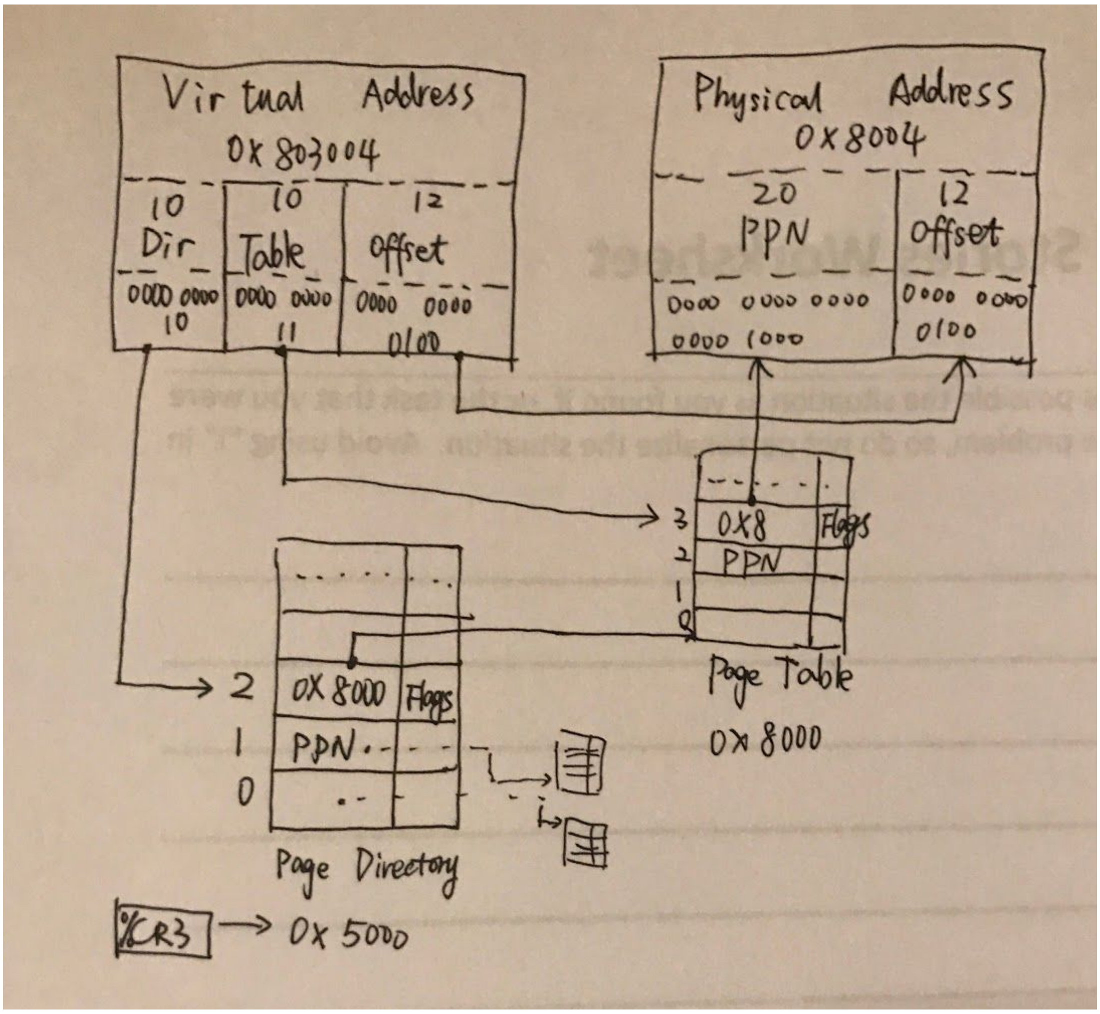
<table>
<tr><th>By info reg</th><th>Addr. (omit top 0x0000)</th><th>Value</th><th>Explanation</th></tr>
<tr><td>esp</td><td>7bcc</td><td>0x00007db7</td><td>Since the last instruction executed was "call *0x10018", according to calling convention, the return address of last control flow will be pushed into the stack. This return address is exactly 0x00007bd7</td></tr>
<tr><td></td><td>7bd0</td><td>0x00000000</td><td rowspan="7">After entering bootmain(in bootblock.asm):
7d35: 83 ec 1c sub $0x1c,%esp
make space on stack for local variable(elf, ph, eph, entry, pa etc.)</td></tr>
<tr><td></td><td>7bd4</td><td>0x00000000</td></tr>
<tr><td></td><td>7bd8</td><td>0x00000000</td></tr>
<tr><td></td><td>7bdc</td><td>0x00000000</td></tr>
<tr><td></td><td>7be0</td><td>0x00000000</td></tr>
<tr><td></td><td>7be4</td><td>0x00000000</td></tr>
<tr><td></td><td>7be8</td><td>0x00000000</td></tr>
</table>

	7bec	0x00000000	old %ebx value which is 0 at that time
	7bf0	0x00000000	old %esi value which is 0 at that time
	7bf4	0x00000000	old %edi value which is 0 at that time
ebp	7bf8	0x00000000	old %ebp value which is 0 at that time
	7bfc	0x00007c4d	Return address from bootasm.S

- Exercise 2: Understanding page tables

Question 1: Explain how virtual to physical address translation works

Illustrate organization of the x86, 4K, 32bit page table through a simple example. Assume that the hardware translates the virtual address '0x803004' into the physical address '0x8004'. The physical addresses of the page table directory (Level 1) and the page table (Level 2) involved in the translation of this virtual address are respectively 0x5000 and 0x8000. <u>The entry 1 of the Global Descriptor Table (GDT) contains the base of 0x1000000 and the limit of 2GBs. The DS register contains the value 0x8.</u>



Virtual Address
0x 803004
10
Dir
10
Table
12
offset
0000 0000 10
0000 0000 11
0000 0000 0100
Physical Address
0x8004
20
PPN
12
offset
0000 0000 0000 0000 1000
0000 0000 0100
0X8
Flags
PPN
Page Table
0x 8000
0X 8000
Flags
PPN
Page Directory
%CR3
0x 5000

Question 2: What is the state of page tables after xv6 is done initializing the first 4K page table?

A: set breakpoint on main function, run continue (c),and run (n) executing functions inside main until you exit kvmalloc(). Then use "info pg " to get information of page table:

```
(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
```

With execution of "kvmalloc()", info pg changed to the above from the following, by allocate one page table for the machine for the kernel address space for scheduler processes.

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

The new page table state shows range of virtual address. For the first line, the virtual address of the first 20 bit starts at 80000, ends at 803ff, which entry page dir PDE is 200. And it's FLAGs are ----A--UWP. The last line tells the mapping physical address (range) of this given virtual address (range).

And the first line's indenting following lines are the detailed page table PTE V2P mapping info. The same as the following 2 PDE lines.

Combined with the "info mem" results:

```
(qemu) info mem
0000000080000000-0000000080100000 0000000000100000 -rw
0000000080100000-0000000080108000 0000000000008000 -r-
0000000080108000-000000008e000000 000000000def8000 -rw
00000000fe000000-0000000100000000 0000000002000000 -rw
```

and its related src for kernel's mapping：

```
// This table defines the kernel's mappings, which are present in
// every process's page table.
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};
```

we know these memories are for I/O space、 kern text and read-only data (as shown in the following from the xv6 book, setting up two ranges of virtual addresses that map to the same physical memory range is a common use of page table) ，kernel read/write data and memory ，other I/O devices.

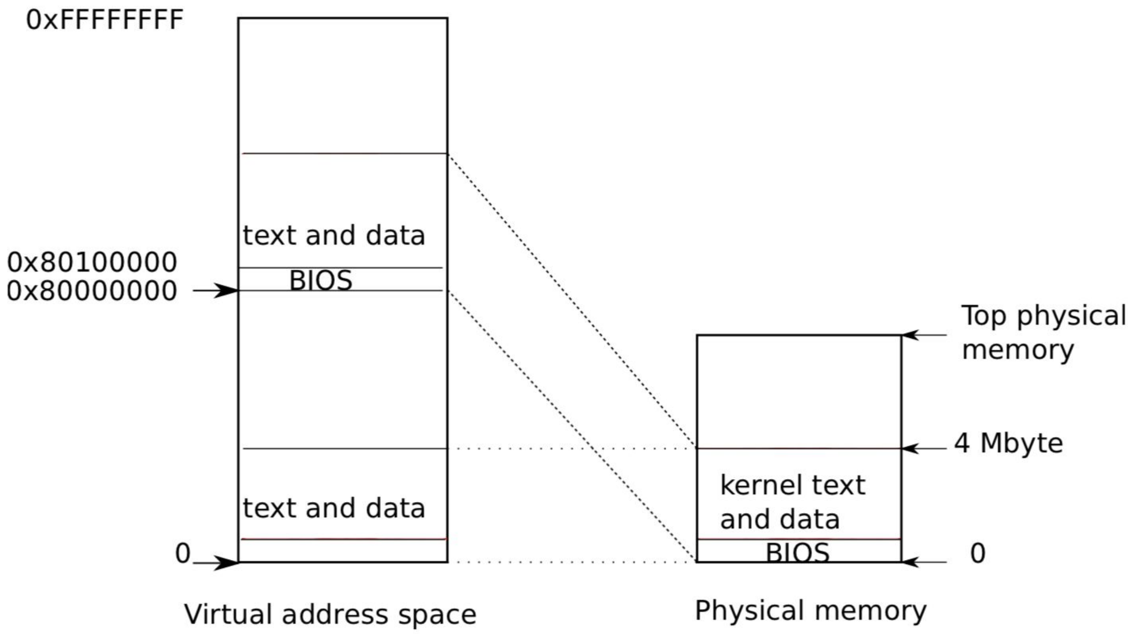


Figure 1-3. Layout of a virtual address space