Hint for OS HW9

> Write a kernel module and a user-mode program to show how the logical address of a variable in the user-mode program is converted to the physical address in x86-64 environment.

All Info from here -> Intel Software Developer Manual

Step 0 How to build a Kernel module ?

FYI. EN / CN

Step 1 Virtual Memory Address to Linear Address

In Manual P83:

3.7.3.1 Memory Operands in 64-Bit Mode

In 64-bit mode, a memory operand can be referenced by a segment selector and an offset. The offset can be 16 bits, 32 bits or 64 bits (see Figure 3-10).

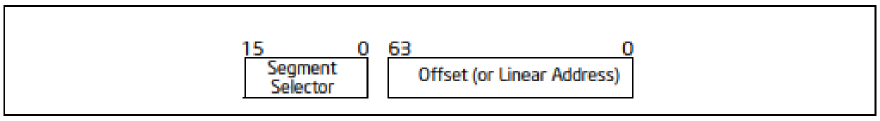
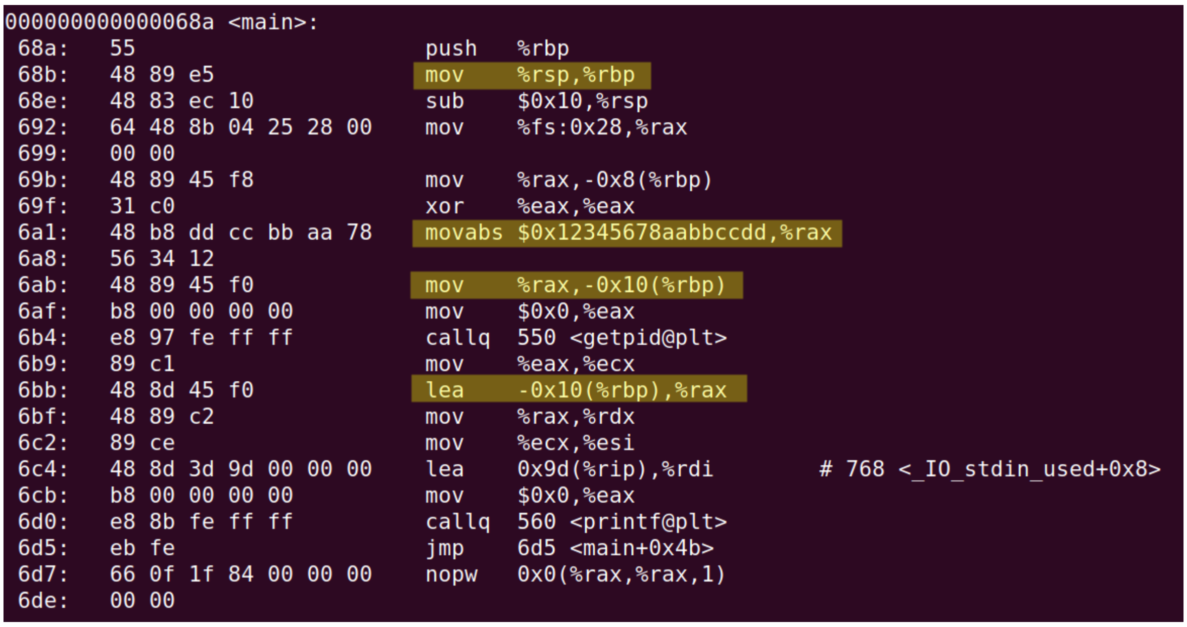


Figure 3-10. Memory Operand Address in 64-Bit Mode

And since the value is saved on the stack, it seems we need to get the segment selector:

> Manual P161
>
> Data manipulation instructions that specify the EBP register as a base register automatically address locations within the stack segment instead of the data segment.



```
000000000000068a <main>:
 68a:   55                      push   %rbp
 68b:   48 89 e5                mov    %rsp,%rbp
 68e:   48 83 ec 10             sub    $0x10,%rsp
 692:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
 699:   00 00
 69b:   48 89 45 f8             mov    %rax,-0x8(%rbp)
 69f:   31 c0                   xor    %eax,%eax
 6a1:   48 b8 dd cc bb aa 78    movabs $0x12345678aabbccdd,%rax
 6a8:   56 34 12
 6ab:   48 89 45 f0             mov    %rax,-0x10(%rbp)
 6af:   b8 00 00 00 00          mov    $0x0,%eax
 6b4:   e8 97 fe ff ff          callq  550 <getpid@plt>
 6b9:   89 c1                   mov    %eax,%ecx
 6bb:   48 8d 45 f0             lea    -0x10(%rbp),%rax
 6bf:   48 89 c2                mov    %rax,%rdx
 6c2:   89 ce                   mov    %ecx,%esi
 6c4:   48 8d 3d 9d 00 00 00    lea    0x9d(%rip),%rdi        # 768 <_IO_stdin_used+0x8>
 6cb:   b8 00 00 00 00          mov    $0x0,%eax
 6d0:   e8 8b fe ff ff          callq  560 <printf@plt>
 6d5:   eb fe                   jmp    6d5 <main+0x4b>
 6d7:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
 6de:   00 00
```

But in Linux, all processes running in User Mode use the same pair of segments to address instructions and data. In Linux 4.15:

```
// In arch/x86/kernel/process_64.h
```

```
void
start_thread(struct pt_regs *regs, unsigned long new_ip, unsigned long new_sp)
{
    start_thread_common(regs, new_ip, new_sp,
                __USER_CS, __USER_DS, 0);
}
static void
start_thread_common(struct pt_regs *regs, unsigned long new_ip,
            unsigned long new_sp,
            unsigned int _cs, unsigned int _ss, unsigned int _ds)
{
    ...

    regs->ip        = new_ip;
    regs->sp        = new_sp;
    regs->cs        = _cs;
    regs->ss        = _ss;
    regs->flags     = X86_EFLAGS_IF;
    force_iret();
}

// In arch/x86/include/asm/segment.h
#define __KERNEL_CS         (GDT_ENTRY_KERNEL_CS*8)
#define __KERNEL_DS         (GDT_ENTRY_KERNEL_DS*8)
#define __USER_DS           (GDT_ENTRY_DEFAULT_USER_DS*8 + 3)
#define __USER_CS           (GDT_ENTRY_DEFAULT_USER_CS*8 + 3)
```

So, we don't need to pass value of ss in user mode, we can simply use macro __USER_DS to get the segment selector. The format of a segment register looks like following: (From Chapter 3, "Protected-Mode Memory Management," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.)

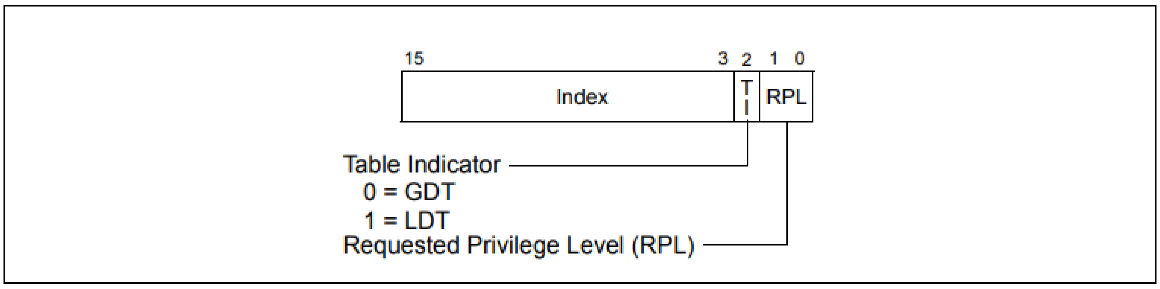


Figure 3-6. Segment Selector

OK, now you get the value of the segment register and its format, the next step you need to do, is get the address of the Global/Local Descriptor Table, to get the descriptor and linear address.

The format of the descriptor.

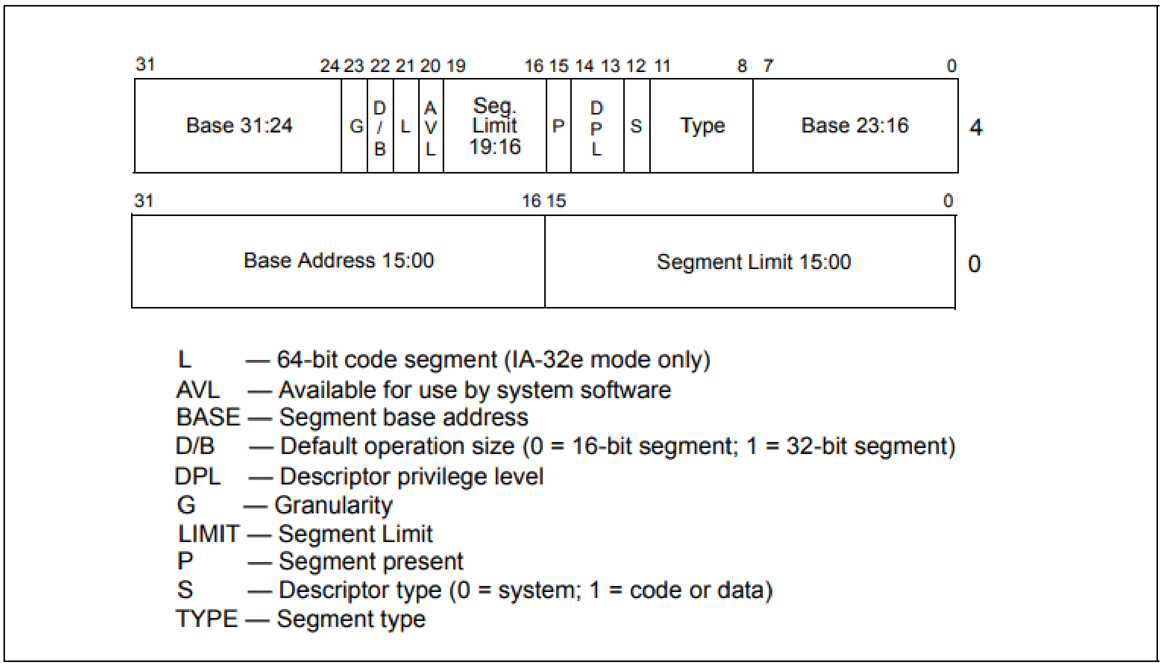

31
24 23 22 21 20 19
16 15 14 13 12 11
8 7
0
Base 31:24
G
D / B
L
A V L
Seg. Limit 19:16
P
D P L
S
Type
Base 23:16
4
31
16 15
0
Base Address 15:00
Segment Limit 15:00
0
L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 3-8. Segment Descriptor

Hit: You can use inline assemble to get [G|L]DT, and use desc_ptr struct can make things easier.

```
struct desc_ptr xdtr;
asm("s[g|l]dt %w0":"=m"(xdtr));
```

Step 2 Linear Address to Physical Address

Since we have a logical address, if we want to convert it to physical address, we need the information of its pgd. But since we have in Kernel, the value in CR3 is not same as our user program, luckily, we can get it from task struct using pid. But, using this method we get a virtual memory.

Hit: using `virt_to_phys()` to convert.

```
// In inlcude/linux/sched.h
struct task_struct {
    ...
    struct mm_struct        *mm;
    ...
}

// In include/linux/mm_types.h
struct mm_struct {
    ...
    pgd_t * pgd;
    ...
}
```

Now, you get the physical address of pgd, but you can not directly access this address.

Hit: Using `kmap()` to map a frame, don't forget to use `kunmap` to free the space.

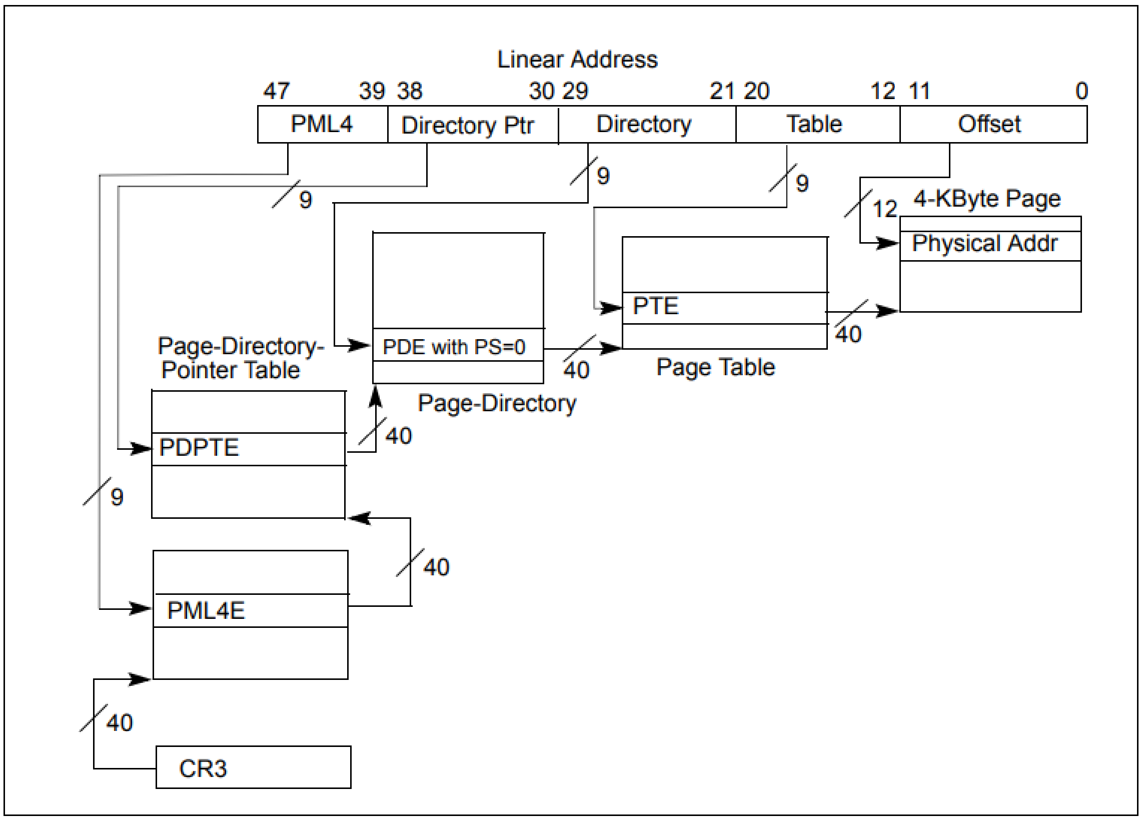


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Next, use the offsets to find the value we want.

Any Problems @ me in QQ groups :)