Chapter 5

Segmentation

Segmentation in x86 means accessing the memory through segments. Segments are portions of the address space, possibly overlapping, specified by a base address and a limit. To address a byte in segmented memory you use a 48-bit *logical address*: 16 bits that specifies the segment and 32-bits that specifies what offset within that segment you want. The offset is added to the base address of the segment, and the resulting linear address is checked against the segment's limit - see the figure below. If everything works out fine (including access-rights checks ignored for now) the result is a *linear address*. When paging is disabled, then the linear address space is mapped 1:1 onto the *physical address* space, and the physical memory can be accessed. (See the chapter "Paging" for how to enable paging.)

To enable segmentation you need to set up a table that describes each segment - a *segment descriptor table*. In x86, there are two types of descriptor tables: the *Global Descriptor Table* (GDT) and *Local Descriptor Tables* (LDT). An LDT is set up and managed by user-space processes, and all processes have their own LDT. LDTs can be used if a more complex segmentation model is desired - we won't use it. The GDT is shared by everyone - it's global.

As we discuss in the sections on virtual memory and paging, segmentation is rarely used more than in a minimal setup, similar to what we do below.

Accessing Memory

Most of the time when accessing memory there is no need to explicitly specify the segment to use. The processor has six 16-bit segment registers: `cs`, `ss`, `ds`, `es`, `gs` and `fs`. The register `cs` is the code segment register and specifies the segment to use when fetching instructions. The register `ss` is used whenever accessing the stack (through the stack pointer `esp`), and `ds` is used for other data accesses. The OS is free to use the registers `es`, `gs` and `fs` however it want.

Below is an example showing implicit use of the segment registers:

```
func:
    mov eax, [esp+4]
    mov ebx, [eax]
    add ebx, 8
    mov [eax], ebx
    ret
```

The above example can be compared with the following one that makes explicit use of the segment registers:

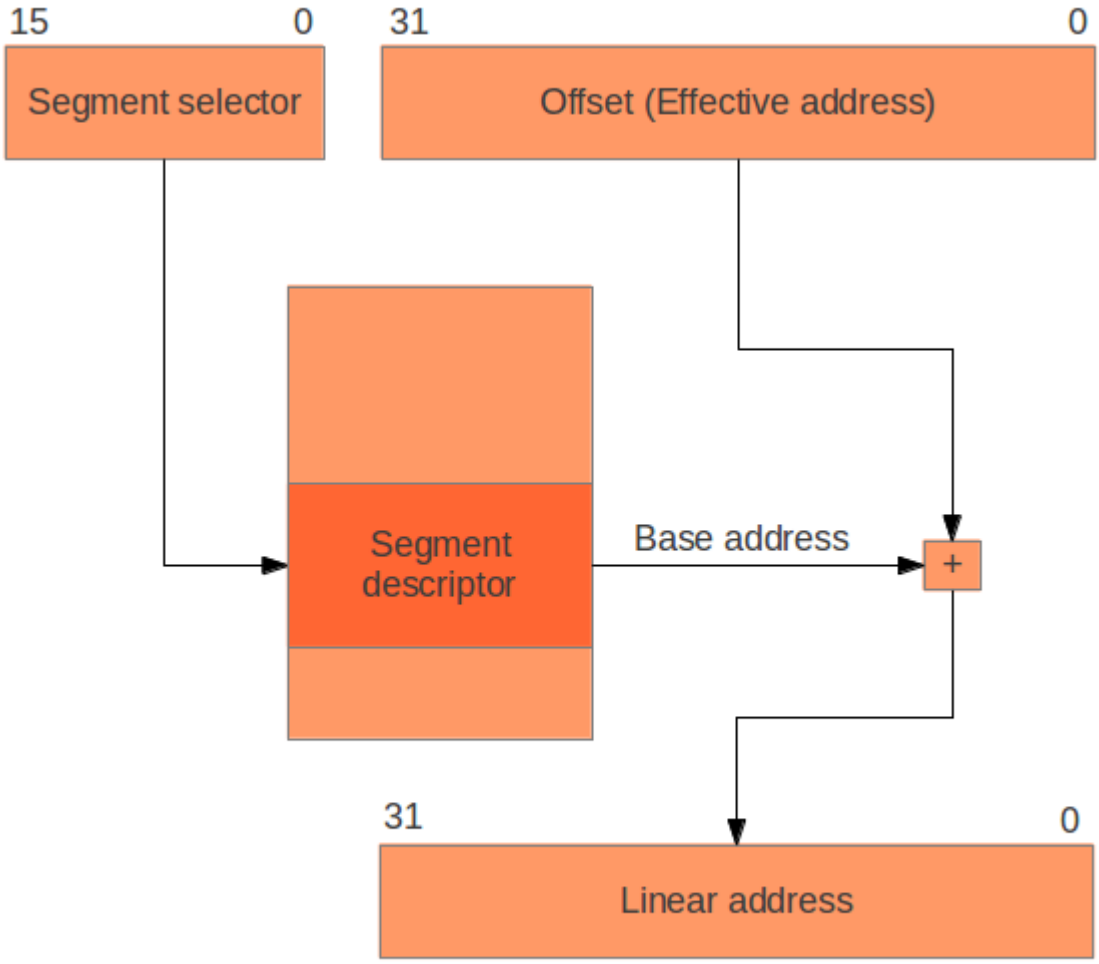

Figure 5.1: Translation of logical addresses to linear addresses.

```
func:
    mov eax, [ss:esp+4]
    mov ebx, [ds:eax]
    add ebx, 8
    mov [ds:eax], ebx
    ret
```

You don't need to use `ss` for storing the stack segment selector, or `ds` for the data segment selector. You could store the stack segment selector in `ds` and vice versa. However, in order to use the implicit style shown above, you must store the segment selectors in their indented registers.

Segment descriptors and their fields are described in figure 3-8 in the Intel manual [33].

The Global Descriptor Table (GDT)

A GDT/LDT is an array of 8-byte segment descriptors. The first descriptor in the GDT is always a null descriptor and can never be used to access memory. At least two segment descriptors (plus the null descriptor) are needed for the GDT, because the descriptor contains more information than just the base and limit fields. The two most relevant fields for us are the *Type* field and the *Descriptor Privilege Level* (DPL) field.

Table 3-1 in chapter 3 of the Intel manual [33] specifies the values for the Type field. The table shows that the Type field can't be both writable *and* executable at the same time. Therefore, two segments are needed: one segment for executing code to put in `cs` (Type is Execute-only or Execute-Read) and one segment for reading and writing data (Type is Read/Write) to put in the other segment registers.

The DPL specifies the *privilege levels* required to use the segment. x86 allows for four privilege levels (PL), 0 to 3, where PL0 is the most privileged. In most operating systems (eg. Linux and Windows), only PL0 and PL3 are used. However, some operating system, such as MINIX, make use of all levels. The kernel should be able to do anything, therefore it uses segments with DPL set to 0 (also called kernel mode). The current privilege level (CPL) is determined by the segment selector in `cs`.

The segments needed are described in the table below.

Index	Offset	Name	Address range	Type	DPL
0	`0x00`	null descriptor			
1	`0x08`	kernel code segment	`0x00000000 - 0xFFFFFFFF`	RX	PL0
2	`0x10`	kernel data segment	`0x00000000 - 0xFFFFFFFF`	RW	PL0

Table 5.1: The segment descriptors needed.

Note that the segments overlap - they both encompass the entire linear address space. In our minimal setup we'll only use segmentation to get privilege levels. See the Intel manual [33], chapter 3, for details on the other descriptor fields.

Loading the GDT

Loading the GDT into the processor is done with the `lgdt` assembly code instruction, which takes the address of a struct that specifies the start and size of the GDT. It is easiest to encode this information using a "packed struct" as shown in the following example:

```
struct gdt {
    unsigned int address;
    unsigned short size;
} __attribute__((packed));
```

If the content of the `eax` register is the address to such a struct, then the GDT can be loaded with the assembly code shown below:

```
lgdt [eax]
```

It might be easier if you make this instruction available from C, the same way as was done with the assembly code instructions `in` and `out`.

After the GDT has been loaded the segment registers needs to be loaded with their corresponding segment selectors. The content of a segment selector is described in the figure and table below:

```
Bit:     | 15                              3 | 2  | 1 0 |
Content: | offset (index)                     | ti | rpl |
```

Name	Description
rpl	Requested Privilege Level - we want to execute in PL0 for now.
ti	Table Indicator. 0 means that this specifies a GDT segment, 1 means an LDT Segment.
offset (index)	Offset within descriptor table.

Table 5.2: The layout of segment selectors.

The offset of the segment selector is added to the start of the GDT to get the address of the segment descriptor: `0x08` for the first descriptor and `0x10` for the second, since each descriptor is 8 bytes. The Requested Privilege Level (RPL) should be `0` since the kernel of the OS should execute in privilege level 0.

Loading the segment selector registers is easy for the data registers - just copy the correct offsets to the registers:

```
mov ds, 0x10
mov ss, 0x10
mov es, 0x10
.
.
.
```

To load `cs` we have to do a "far jump":

```
; code here uses the previous cs
jmp 0x08:flush_cs   ; specify cs when jumping to flush_cs

flush_cs:
    ; now we've changed cs to 0x08
```

A far jump is a jump where we explicitly specify the full 48-bit logical address: the segment selector to use and the absolute address to jump to. It will first set `cs` to `0x08` and then jump to `flush_cs` using its absolute address.

Further Reading

- Chapter 3 of the Intel manual [33] is filled with low-level and technical details about segmentation.
- The OSDev wiki has a page about segmentation: http://wiki.osdev.org/Segmentation
- The Wikipedia page on x86 segmentation might be worth looking into: http://en.wikipedia.org/wiki/X86_memory_segmentation