Segmentation

From OSDev Wiki

Contents

Real mode

In Real Mode you use a logical address in the form A:B to address memory. This is translated into a physical address using the equation:

```
Physical address = (A * 0x10) + B
```

The registers in pure real-mode are limited to 16 bits for addressing. 16 bits can represent any integer between 0 and 64k. This means that if we set A to be a fixed value and allow B to change we can address a 64k area of memory. This 64k area is called a segment.

```
A = A 64k segment B = Offset within the segment
```

The base address of a segment is the (A * 0x10) portion of the equation I showed. It should be obvious that segments can overlap.

Eg, the segment 0x1000 has a base address of 0x10000. This segment occupies the physical address range 0x10000 -> 0x1FFFF, However the segment 0x1010 has a base address of 0x10100. This segment occupies the physical address range 0x10100 -> 0x200FF

As you can see we could use either segment to reach physical addresses between 0x10100 and 0x1FFFF since the segments overlap.

The x86 line of computers have 6 segment registers (CS, DS, ES, FS, GS, SS). They are totally independent of one another.

CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Segment
FS	General Purpose Segments
GS	

DS,ES,FS,GS,SS are used to form addresses when you want to read/write to memory. They don't always have to be explicitly encoded, because some processor operations assume that certain segment registers will be used.

E.g.

MOV [SI], AX will write the word contained in ax to the address DS:SI

MOV ES:[DI], AX will write the word contained in ax to the address es:di

CMPSB will compare the byte at DS:SI to the byte at ES:DI, set the zero flag if they are equal and decrement/increment SI and DI according to the state of the direction flag.

As you can see, often the segment register being used is not contained in the instruction, but there is one being used. EVERY time you form an address on an x86 processor there will be a segment register involved.

Operations that affect segment registers

Beside CS, segment registers may be loaded with a general register (mov ds, ax) or with the top-of-stack (pop ds).

CS is the only Segment Register that cannot be directly altered. The only time (I'm sure I'm missing one) CS is altered is when the code switches execution into another segment. The only commands that can do this are:

Far Jump

Here the new value for CS is encoded in the jump instruction. Eg JMP 0x10:0x100 says to load CS with segment 0x10 and IP with 0x100. CS:IP is the logical address of the instruction to be executed.

Far Call

This is exactly the same as a far jump, but the current values of CS/IP are pushed onto the stack before executing at the new position.

INT

The processor reads the new value of CS/IP from the Interrupt Vector Table and then executes what is effectively a far call after pushing EFLAGS onto the stack.

Far Return

Here the processor pops the return segment/offset from the stack into CS/IP and switches execution to that address.

IRET

This is exactly the same as a far return apart from the processor popping EFLAGS off the stack in addition to CS/IP.

Apart from these cases no instruction alters the value of CS.

Protected Mode

> *Segmentation is considered obsolete memory protection technique in protected mode by both CPU manufacturers and most of programmers. It is no longer supported in long mode. The information here is required to get protected mode working; also 64 bit GDT is needed to enter long mode and segments are still used to jump from long mode to compatibility mode and the other way around. If you want to be serious about OS development, we strongly recommend using flat memory model and paging as memory management technique. For more information, consult x86-64.*
>
> *Read more about Global Descriptor Table*

In Protected mode you use a logical address in the form A:B to address memory. As in Real Mode, A is the segment part and B is the offset within that segment. The registers in protected mode are limited to 32 bits. 32 bits can represent any integer between 0 and 4 GiB.

Because B can be any value between 0 and 4GiB our segments now have a maximum size of 4 GiB (Same reasoning as in real-mode).

Now for the difference.

In protected mode A is not an absolute value for the segment. In protected mode A is a selector. A selector represents an offset into a system table called the Global Descriptor Table (GDT). The GDT contains a list of descriptors. Each of these descriptors contains information that describes the characteristics of a segment.

Each segment descriptor contains the following information:

- The base address of the segment
- The default operation size in the segment (16-bit/32-bit)
- The privilege level of the descriptor (Ring 0 -> Ring 3)
- The granularity (Segment limit is in byte/4kb units)
- The segment limit (The maximum legal offset within the segment)
- The segment presence (Is it present or not)
- The descriptor type (0 = system; 1 = code/data)
- The segment type (Code/Data/Read/Write/Accessed/Conforming/Non-Conforming/Expand-Up/Expand-Down)

For the purposes of this explanation I'm only interested in 3 things. The base address, the limit and the descriptor type.

If the descriptor type is clear (System type) then the descriptor isn't actually describing a segment, it's describing either one of the special gate mechanisms, where to find an LDT, or a TSS. These have nothing to do with general addressing, so I'll assume a descriptor type of 1 (code/data type) and leave you to read the Intel manuals for the rest.

The segment is described by its base address and limit. Remember in real-mode where the segment was a 64k area in memory? The only difference here is that the size of the segment isn't fixed. The base address supplied by the descriptor is the start of the segment, the limit is the maximum offset the processor will allow before producing an exception.

So the range of physical addresses in our protected mode segment is:

```
Segment Base -> Segment Base + Segment Limit
```

Given a logical address A:B (Remember that A is a selector) we can determine the physical address it translates to using:

```
Physical address = Segment Base (Found from the descriptor GDT[A]) + B
```

All the other rules from real-mode still apply.

Notes

- Segments can overlap
- CS,DS,ES,FS,GS,SS are independent of each other
- CS cannot be changed directly

In protected mode CS can also be changed via the TSS or a gate.

Notes Regarding C

- Most C compilers assume a flat-memory model.
- In this model all the segments cover the full address space (Usually 0->4Gb on x86). In essence this means that we completely ignore the A part of our A:B logical address. The reason for this is that most processors don't actually have segmentation (Plus it's a hell of a lot easier for the compiler to optimize).
- This leaves you with 2 descriptors per privilege level (Ring 0 and Ring 3 normally), one for code and one for data, which both describe precisely the same segment. The only difference being that the code descriptor is loaded into CS, and the data descriptor is used by all the other segment registers. The reason you need both a code and data descriptor is that the processor will not allow you to load CS with a data descriptor (This is to help with security when using a segmented memory model, and although useless in the flat-memory model it is still required because you can't turn off segmentation).
- In general if you want to use the segmentation mechanism, by having the different segment registers represent segments with different base addresses, you won't be able to use a modern C compiler, and may very well be restricted to just Assembly.
- So, if you're going to use C, do what the rest of the C world does, which is set up a flat-memory model, use paging, and ignore the fact that segmentation even exists.

Notes Regarding Pascal[FPC]

Above may apply in theory to FreePascal, however, in reality is ignored, if the compiler at all pays any attention to same. The twin segments for CODE and DATA are used, and as specified above,therefore, required.Size limits, however, are respected.(does NOT have to be 4GB in length)

"*In general if you want to use the segmentation mechanism, by having the different segment registers represent segments with different base addresses, you won't be able to use a modern C compiler, and may very well be restricted to just Assembly."

This is simply NOT true for Freepascal.

The 'A in A:B' is what allows 48 and 64 bit pointer references, not only with Pascal's NewFrontier unit, but FreePascal as well(Word:Longint Pointer reference).

- Assumption of CODE and DATA occupying the same space, (at least with PAE NX bits and Paging units not used) allows ROGUE/virus like code in the first place to take advantage of the machine. INTEL Specs even say this. CODE and DATA must be KEPT separate. Microsoft still is plagued with this problem,despite having NX bits enabled even in the latest OSes.

See Also

Articles

Threads

External Links

- Removing the Mystery from SEGMENT : OFFSET Addressing (http://mirror.href.com/thestarman/asm/debug/Segments.html)
- Aug 2008: Memory Translation and Segmentation (http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation) by Gustavo Duarte

Category: Memory management
