Global Descriptor Table

From Lowlevel

The Global Descriptor Table (also called GDT) is an i386 + -specific structure that contains information about memory sections (segments).

Table of Contents

segmentation

16-bit Real Mode

The 8086 is a 16-bit processor. This means that its registers are 16 bits wide and can therefore record values up to $2^{16} - 1 = 65535$. But now also pointers in this register must fit, and in this way only 64 KiB can be addressed, which did not represent a generous address space even with the introduction of the processor.

For this reason, a simple type of segmentation has been introduced in real mode. For each memory access, a base address is added, which is determined by the content of the corresponding segment register (*cs* for code, *ds* for data, *ss* for the stack). As a result, addresses are effectively 20 bits in size; In other words, 1 MiB of address space is accessible, with the segment register having to be changed when accessing more than 64 KiB.

The calculation of an effective address - in the example an access to *[ds: ax]* - runs as follows:

```
0x 1234 Address in ax
0x 4321 segment register ds
========
0x 44444 Actually accessed address
```

16-bit protected mode

Over time, the 1 MiB memory proved to be no longer sufficient, not least because of this almost half was reserved for the BIOS and other special tasks. In addition, the real-mode segmentation did not allow memory protection, because each program could still access all addresses, which made the distinction of different privilege levels impossible. Also a memory management became difficult, if one did not want to correct all segment addresses to be loaded each time. For this reason, Protected Mode introduced a new, more complex segmentation concept.

Since the 80286 had 24 address lines and could access 16 MiB of memory, and even memory protection information should be stored, the 16 bits of the segment register proved too tight to hold all this information. In addition, the data from the application programs should not be arbitrary, but should be changeable within the given framework. For this reason, the segment registers in protected mode only contain so-called selectors, i. H. the number of a segment and a few flags. The actual description of the segments is in the form of a table in memory, the *Global Descriptor Table*(GDT). The GDT consists of several entries (segment descriptors) of 64 bits each, which are directly behind each other. These entries could easily accommodate a 24-bit base address, a limit describing the size of the segment, privilege level, and more. Also for special entries was still room.

For individual programs, there is also another similarly structured table, the *Local Descriptor Table* (LDT). A program could thus access a number of segments specified by the system, which are described in both tables and were specified by the system according to the available resources and the needs of the program.

32-bit protected mode

Segments are still present in the 32-bit protected mode of the 386, but they play a slightly different role. They have at least partially lost their old task because the 386 is a 32-bit processor and could address the available 4 GiB of memory without segmentation.

On the one hand, however, the base address can continue to be useful - it allows code to be executed anywhere in memory, even if it starts from fixed addresses (this mechanism is rarely used). On the other hand, the segments have been extended so that they not only simply consist of a base address, but provide additional memory protection functionality, such as write protection or access only at a certain privilege level. In addition, a segment no longer has a fixed length of 64 KiB, but the length can be set individually and flexibly by the operating system for each segment. Protected mode segments can thus be positioned almost arbitrarily in the linear / virtual address space.

Despite all this, most operating systems mainly manage memory management in 32-bit mode through more flexible paging and do not take full advantage of GDT's capabilities. These operating systems use the "flat memory" model, which virtually bypasses segmentation: all memory is represented by a single, large segment.

Long fashion

Nevertheless, even in the "flat memory" model, the GDT continues to play a role in the administration of privileges, even though it has definitively lost its function as a storage manager in long mode, as this enforces the "flat memory" model. Last but not least, it has the crucial task of coordinating the switch between *legacy mode* for 32-bit programs and 64-bit mode. A thorough study of the GDT is therefore always useful.

selectors

As mentioned in the previous section, there are selectors in the segment registers in the protected mode which refer to descriptor tables. They have the following structure:

bits	value	importance
0 - 1	0x3	RPL (Requested Privilege Level). Specifies the privilege level (ring) to try to target the segment, if numerically smaller than the current one of the calling task, it will keep its current one
2	0x4	Specifies whether an entry in the GDT (bit not set) or in the LDT (bit set) is selected
3 - 15	0xfff8	Number of the entry in the GDT or LDT (counted from zero)

Thus, to load the segment described by the fifth GDT entry after *ds* , as the userspace segment (ring 3), one could use the following code: <asm> mov $ 0x23,% ax mov% ax,% ds </ asm>

structure

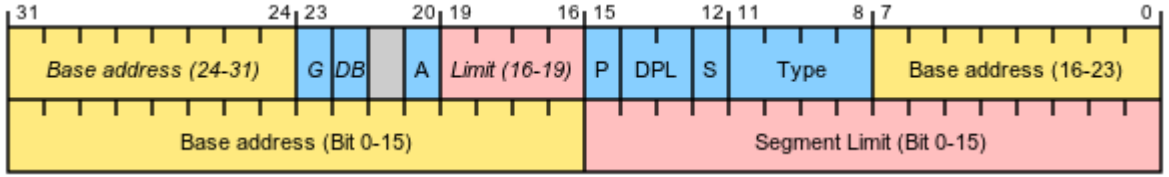
Each entry in the GDT consists of eight bytes.

When creating the GDT, note that the first entry is the so-called null descriptor. It is basically invalid and the entry can be left completely on 0. Only from the second entry do the descriptors actually used begin.

Structure of an entry

byte	Surname	bits

0	limit	0-7
1	limit	8-15
2	base	0-7
3	base	8-15
4	base	16-23
5	Accessbyte	0-7 (complete)
6	limit	16-19
6	flags	0-3 (complete)
7	base	24-31



importance

Surname	importance
limit	Size of the segment - 1 (either in bytes or in 4KiB units - see flags)
base	The address where the segment starts
Accessbyte	Access information (ring, executable, etc.)
flags	Defines the segment size unit and 16/32 bit.

The access byte

bit	value	Surname	importance
7	0x80	Present bit	Must be 1 for an active entry
6	0x60	Privilege	Ring - from ring 0 (kernel mode) to ring 3 (user mode)

and 5			
4	0x10	Segment bit	Is 1 for code / data segments; 0 for gates and TSS . Bits 0 through 3 in this table apply only to code / data segments, otherwise they contain the exact segment type (for example, 0x9 for 386-TSS)
3	0x08	Executable bit	At 1 the memory segment is a code segment, at 0 a data segment
2	0x04	Direction Bit / Conforming Bit	Meaning depends on the executable bit
1	0x02	Readable Bit / Writable Bit	If a code segment is allowed to read, a write is allowed on a data segment
0	0x01	Accessed bit	If 0 is set then the processor will set it on access.

The flags

bit	value	Surname	importance
3	0x8	Granularity bit	If 0, the limit unit is 1 byte, 1 is 4 KiB instead
2	0x4	Size bit	0 means 16 bit protected mode, with 1 32 bit protected mode.
1	0x2	Long Mode Bit	0 means Protected Mode, 1 means Long Mode. If the long mode bit is set , the size bit **must be** 0! If the long mode bit is 0, the size bit indicates the bit size of the protected mode segment. *This bit should only be set for one code segment, for other segment types it should be set to 0.*
0	0x1	Available bit	This bit is at the disposal of the programmer.

Set up the GDT

Before we consider which segments we want to define in our GDT, we first need a function to create GDT entries in a reasonably readable form. Assuming the GDT is represented as an array of 64-bit integers, it could look like this. The code does nothing else but move the parameters to the correct position according to the tables above:

<c> static void set_entry (int i, unsigned int base, unsigned int limit, int flags) {

```
  gdt [i] = limit & 0xffffLL;
  gdt [i] | = (base & 0xffffffLL) << 16;
  gdt [i] | = (flags & 0xffLL) << 40;
  gdt [i] | = ((limit >> 16) & 0xfLL) << 48;
  gdt [i] | = ((flags >> 8) & 0xffLL) << 52;
  gdt [i] | = ((base >> 24) & 0xffLL) << 56;
```

} </ C>

Of course, what segments you want to create depends on the architecture of your kernel. My suggestion is not the absolute and only truth, but it is a good start if you are unsure. The basis is the GDT as týndur they used and how they can be found in most other operating systems. Memory protection is done via paging , so that only a minimal set of segments remains:

- A zero descriptor (gdt [0] = 0)
- A code segment for the kernel (Present, Ring 0, Executable, 32 bits, Base 0, Limit 4 GiB)
- One data segment for the kernel (Present, Ring 0, Non-Executable, 32 Bit, Base 0, Limit 4 GiB)
- A code segment for the userspace (Present, Ring 3, Executable, 32 Bit, Base 0, Limit 4 GiB)
- One data segment for the userspace (Present, Ring 3, Non-Executable, 32 Bit, Base 0, Limit 4 GiB)
- A task state segment for multitasking (Present, Ring 3, 386-TSS)
- A task state segment for treating double faults (Present, Ring 3, 386-TSS)

Absolutely necessary are only zero descriptor and code / data segment for the kernel. As soon as multitasking is to come into play, the TSS is due and with user-space processes it then needs the code / data segments for ring 3.

The second TSS for Double Faults is completely optional. Double faults are often caused by the fact that z. For example, the paging trees have been overridden and the handler for a Page Fault itself no longer works. A TSS offers the opportunity here to jump into a clean processor state and at least issue a meaningful error message.

Loading the GDT

After the kernel has a GDT in memory that it wants to use for its segments, it must change the GDTR register. This register contains the address and the so-called limit of GDT & # 150; this corresponds to its length minus one byte & # 150; and is loaded via the **lgdt** command . Since the register is 6 bytes wide, this instruction can not take the new value directly as an operand, but expects a pointer to a memory location containing these 6 bytes:

<c> static uint64_t gdt [GDT_ENTRIES];

// ...

struct {struct {

```
  uint16_t limit;
  void * pointer;
```

} __attribute __ ((packed)) gdtp = {

```
  .limit = GDT_ENTRIES * 8-1,
  .pointer = gdt,
```

}; asm volatile ("lgdt% 0":: "m" (gdtp)); </ c>

However, this will only change the reference to the table, but the change will not actually affect it. The table is accessed only when a segment register is reloaded. Therefore all segment registers have to be reloaded now:

<asm> mov $ 0x10,% ax mov% ax,% ds mov% ax,% it mov% ax,% fs mov% ax,% gs mov% ax,% ss ljmp $ 0x8, $ .1 .1: < / asm>

Web Links

Protected-Fashion-Tutorial of FH Zwickau (http://www.fh-zwickau.de/doc/prmo/pmtutor/text/)

Retrieved from " http://www.lowlevel.eu/w/index.php?title=Global_Descriptor_Table&oldid=11158 "
