GDT Tutorial

From OSDev Wiki

Difficulty level
🟩⬜⬜⬜
Beginner

In the Intel Architecture, and more precisely in protected mode, most of the memory management and Interrupt Service Routines are controlled through tables of descriptors. Each descriptor stores information about a single object (e.g. a service routine, a task, a chunk of code or data, whatever) the CPU might need at some time. If you try, for instance, to load a new value into a segment register, the CPU needs to perform safety and access control checks to see whether you're actually entitled to access that specific memory area. Once the checks are performed, useful values (such as the lowest and highest addresses) are cached in invisible registers of the CPU.

Intel defined 3 types of tables: the Interrupt Descriptor Table (which supplants the IVT), the Global Descriptor Table (GDT) and the Local Descriptor Table. Each table is defined as a (size, linear address) to the CPU through the `LIDT`, `LGDT`, `LLDT` instructions respectively. In most cases, the OS simply tells where those tables are once at boot time, and then simply goes writing/reading the tables through a pointer.

Contents

Survival Glossary

Segment
: a logically contiguous chunk of memory with consistent properties (CPU's speaking)

Segment Register
: a register of your CPU that refers to a segment for a special use (e.g. `SS`, `CS`, `DS` ...)

Selector

a reference to a descriptor you can load into a segment register; the selector is an offset of a descriptor table entry. These entries are 8 bytes long, therefore selectors can have values 0x00, 0x08, 0x10, 0x18, ...

Descriptor
: a memory structure (part of a table) that tells the CPU the attributes of a given segment

What should i put in my GDT?

Basics

For sanity purpose, you should always have these items stored in your GDT:

- The null descriptor which is never referenced by the processor. Certain emulators, like Bochs, will complain about limit exceptions if you do not have one present. Some use this descriptor to store a pointer to the GDT itself (to use with the LGDT instruction). The null descriptor is 8 bytes wide and the pointer is 6 bytes wide so it might just be the perfect place for this.
- A code segment descriptor (for your kernel, it should have type=0x9A)
- A data segment descriptor (you can't write to a code segment, so add this with type=0x92)
- A TSS segment descriptor (trust me, keep a place for at least one)
- Room for more segments if you need them (e.g. user-level, LDTs, more TSS, whatever)

Sysenter/Sysexit

If you are using the Intel `SYSENTER`/`SYSEXIT` routines, the GDT must be structured like this:

- Any descriptors preceding (null descriptor, special kernel stuff, etc.)
- A DPL 0 code segment descriptor (the one that `SYSENTER` will use)
- A DPL 0 data segment descriptor (for the `SYSENTER` stack)
- A DPL 3 code segment (for the code after `SYSEXIT`)
- A DPL 3 data segment (for the user-mode stack after `SYSEXIT`)
- Any other descriptors

The segment of the DPL 0 code segment is loaded into an MSR. The others are calculated based on that value. See the Intel Instruction Reference for `SYSENTER` and `SYSEXIT` for more information.

The actual values you store there will depend on your system design.

Flat Setup

You want to have full 4 GiB addresses untranslated: just use

```
GDT[0] = {.base=0, .limit=0, .type=0};                     // Selector 0x00 cc
GDT[1] = {.base=0, .limit=0xffffffff, .type=0x9A};         // Selector 0x08 wi
GDT[2] = {.base=0, .limit=0xffffffff, .type=0x92};         // Selector 0x10 wi
GDT[3] = {.base=&myTss, .limit=sizeof(myTss), .type=0x89}; // You can use LTR(
```

Note that in this model, code is not actually protected against overwriting since code and data segment overlaps.

Small Kernel Setup

If you want (for specific reason) to have code and data clearly separated (let's say 4 MiB for both, starting at 4 MiB as well), just use:

```
GDT[0] = {.base=0, .limit=0, .type=0};                     // Selector 0x00 c
GDT[1] = {.base=0x04000000, .limit=0x03ffffff, .type=0x9A}; // Selector 0x08 w
GDT[2] = {.base=0x08000000, .limit=0x03ffffff, .type=0x92}; // Selector 0x10 w
GDT[3] = {.base=&myTss, .limit=sizeof(myTss), .type=0x89};  // You can use LTR
```

That means whatever you load at physical address 4 MiB will appear as code at `CS:0` and what you load at physical address 8 MiB will appear as data at `DS:0`. However it might not be the best design.

How can we do that?

Disable interrupts

If they're enabled, turn them off or you're in for trouble.

Filling the table

You noticed that I didn't give a real structure for `GDT[]`, didn't you? That's on purpose. The actual structure of descriptors is a little messy for backwards compatibility with the 286's GDT. Base address are split on 3 different fields and you cannot encode any limit you want. Plus, here and there, you have flags that you need to set up properly if you want things to work.

```
/**
 * \param target A pointer to the 8-byte GDT entry
 * \param source An arbitrary structure describing the GDT entry
 */
void encodeGdtEntry(uint8_t *target, struct GDT source)
{
    // Check the limit to make sure that it can be encoded
    if ((source.limit > 65536) && (source.limit & 0xFFF) != 0xFFF)) {
        kerror("You can't do that!");
    }
    if (source.limit > 65536) {
        // Adjust granularity if required
        source.limit = source.limit >> 12;
        target[6] = 0xC0;
    } else {
        target[6] = 0x40;
    }

    // Encode the limit
    target[0] = source.limit & 0xFF;
    target[1] = (source.limit >> 8) & 0xFF;
    target[6] |= (source.limit >> 16) & 0xF;

    // Encode the base
    target[2] = source.base & 0xFF;
    target[3] = (source.base >> 8) & 0xFF;
    target[4] = (source.base >> 16) & 0xFF;
    target[7] = (source.base >> 24) & 0xFF;
    // And... Type
    target[5] = source.type;
}
```

Okay, that's rather ugly, but it's the most 'for dummies' I can come with ... hope you know about masking and shifting. You can hard-code that rather than convert it at runtime, of course. This code assumes that you only want 32-bit segments.

Telling the CPU where the table stands

Some assembly example is required here. While you could use inline assembly, the memory packing expected by `LGDT` and `LIDT` makes it much easier to write a small assembly routine instead. As said above, you'll use `LGDT` instruction to load the base address and the limit of the GDT. Since the base address should be a linear address, you'll need a bit of tweaking depending of your current MMU setup.

From real mode

The linear address should here be computed as `segment * 16 + offset`. I'll assume `GDT` and `GDT_end` are symbols in the current data segment.

```
gdtr DW 0 ; For limit storage
     DD 0 ; For base storage

setGdt:
   XOR   EAX, EAX
   MOV   AX, DS
   SHL   EAX, 4
   ADD   EAX, ''GDT''
   MOV   [gdtr + 2], eax
   MOV   EAX, ''GDT_end''
   SUB   EAX, ''GDT''
   MOV   [gdtr], AX
   LGDT  [gdtr]
   RET
```

From flat, protected mode

"Flat" meaning the base of your data segment is 0 (regardless of whether paging is on or off). This is the case if you're just been booted by GRUB, for instance. I'll assume you call `setGdt(GDT, sizeof(GDT))`.

```
gdtr DW 0 ; For limit storage
     DD 0 ; For base storage

setGdt:
   MOV   EAX, [esp + 4]
   MOV   [gdtr + 2], EAX
   MOV   AX, [ESP + 8]
   MOV   [gdtr], AX
   LGDT  [gdtr]
   RET
```

From non-flat protected mode

If your data segment has a non-zero base (e.g. you're using a Higher Half Kernel during the segmentation trick), you'll have to "`ADD EAX, base_of_your_data_segment_which_you_should_know`" between the "`MOV EAX, ...`" and the "`MOV ..., EAX`" instructions of the sequence above.

Reload segment registers

Whatever you do with the GDT has no effect on the CPU until you load selectors into segment registers. You can do this using:

```
reloadSegments:
   ; Reload CS register containing code selector:
   JMP   0x08:reload_CS ; 0x08 points at the new code selector
.reload_CS:
   ; Reload data segment registers:
   MOV   AX, 0x10 ; 0x10 points at the new data selector
   MOV   DS, AX
   MOV   ES, AX
   MOV   FS, AX
   MOV   GS, AX
   MOV   SS, AX
   RET
```

An explanation of the above code can be found here (http://stackoverflow.com/questions/23978486/far-jump-in-gdt-in-bootloader) .

What's so special about the LDT?

Much like the GDT (global descriptor table), the LDT (*local* descriptor table) contains descriptors for memory segments description, call gates, etc. The good thing with the LDT is that each task can have its own LDT and that the processor will automatically switch to the right LDT when you use hardware task switching.

Since its content may be different in each task, the LDT is not a suitable place to put system stuff such as TSS or other LDT descriptors: Those are the sole property of the GDT. Since it is meant to change often, the command used for loading an LDT is a bit different from the GDT and IDT loading. Rather than giving directly the LDT's base address and size, those parameters are stored in a descriptor of the GDT (with proper "LDT" type) and the selector of that entry is given.

```
               GDTR (base + limit)
              +-- GDT ------------+
              |                   |
SELECTOR ---> [LDT descriptor     ]----> LDTR (base + limit)
              |                   |     +-- LDT ------------+
              |                   |     |                   |
             ...                 ...   ...                 ...
              +-------------------+     +-------------------+
```

Note that with 386+ processors, the paging has made LDT almost obsolete, and there's no longer need for multiple LDT descriptors, so you can almost safely ignore the LDT for OS developing, unless you have by design many different segments to store.

What is the IDT and is it needed?

As said above, the IDT (Interrupt Descriptor Table) loads much the same way as the GDT and its structure is roughly the same except that it only contains gates and not segments. Each gate gives a full reference to a piece of code (code segment, privilege level and offset to the code in that segment) that is now bound to a number between 0 and 255 (the slot in the IDT).

The IDT will be one of the first things to be enabled in your kernel sequence, so that you can catch hardware exceptions, listen to external events, etc. See Interrupts for dummies for more information about the interrupts of X86 family.

Some stuff to make your life easy

Tool for easily creating GDT entries.

```
// Used for creating GDT segment descriptors in 64-bit integer form.

#include <stdio.h>
#include <stdint.h>

// Each define here is for a specific flag in the descriptor.
// Refer to the intel documentation for a description of what each one does.
#define SEG_DESCTYPE(x)  ((x) << 0x04) // Descriptor type (0 for system, 1 for
#define SEG_PRES(x)      ((x) << 0x07) // Present
#define SEG_SAVL(x)      ((x) << 0x0C) // Available for system use
#define SEG_LONG(x)      ((x) << 0x0D) // Long mode
#define SEG_SIZE(x)      ((x) << 0x0E) // Size (0 for 16-bit, 1 for 32)
#define SEG_GRAN(x)      ((x) << 0x0F) // Granularity (0 for 1B - 1MB, 1 for 4
#define SEG_PRIV(x)     (((x) &  0x03) << 0x05)   // Set privilege level (0 -

#define SEG_DATA_RD        0x00 // Read-Only
#define SEG_DATA_RDA       0x01 // Read-Only, accessed
#define SEG_DATA_RDWR      0x02 // Read/Write
#define SEG_DATA_RDWRA     0x03 // Read/Write, accessed
#define SEG_DATA_RDEXPD    0x04 // Read-Only, expand-down
#define SEG_DATA_RDEXPDA   0x05 // Read-Only, expand-down, accessed
#define SEG_DATA_RDWREXPD  0x06 // Read/Write, expand-down
#define SEG_DATA_RDWREXPDA 0x07 // Read/Write, expand-down, accessed
#define SEG_CODE_EX        0x08 // Execute-Only
#define SEG_CODE_EXA       0x09 // Execute-Only, accessed
#define SEG_CODE_EXRD      0x0A // Execute/Read
#define SEG_CODE_EXRDA     0x0B // Execute/Read, accessed
#define SEG_CODE_EXC       0x0C // Execute-Only, conforming
#define SEG_CODE_EXCA      0x0D // Execute-Only, conforming, accessed
#define SEG_CODE_EXRDC     0x0E // Execute/Read, conforming
#define SEG_CODE_EXRDCA    0x0F // Execute/Read, conforming, accessed

#define GDT_CODE_PL0 SEG_DESCTYPE(1) | SEG_PRES(1) | SEG_SAVL(0) | \
                     SEG_LONG(0)     | SEG_SIZE(1) | SEG_GRAN(1) | \
                     SEG_PRIV(0)     | SEG_CODE_EXRD

#define GDT_DATA_PL0 SEG_DESCTYPE(1) | SEG_PRES(1) | SEG_SAVL(0) | \
                     SEG_LONG(0)     | SEG_SIZE(1) | SEG_GRAN(1) | \
                     SEG_PRIV(0)     | SEG_DATA_RDWR
```

```
#define GDT_CODE_PL3 SEG_DESCTYPE(1) | SEG_PRES(1) | SEG_SAVL(0) | \
                     SEG_LONG(0)     | SEG_SIZE(1) | SEG_GRAN(1) | \
                     SEG_PRIV(3)     | SEG_CODE_EXRD

#define GDT_DATA_PL3 SEG_DESCTYPE(1) | SEG_PRES(1) | SEG_SAVL(0) | \
                     SEG_LONG(0)     | SEG_SIZE(1) | SEG_GRAN(1) | \
                     SEG_PRIV(3)     | SEG_DATA_RDWR

void
create_descriptor(uint32_t base, uint32_t limit, uint16_t flag)
{
    uint64_t descriptor;

    // Create the high 32 bit segment
    descriptor  =  limit       & 0x000F0000;         // set limit bits 19:16
    descriptor |= (flag <<  8) & 0x00F0FF00;         // set type, p, dpl, s, g
    descriptor |= (base >> 16) & 0x000000FF;         // set base bits 23:16
    descriptor |=  base        & 0xFF000000;         // set base bits 31:24

    // Shift by 32 to allow for low part of segment
    descriptor <<= 32;

    // Create the low 32 bit segment
    descriptor |= base  << 16;                       // set base bits 15:0
    descriptor |= limit  & 0x0000FFFF;               // set limit bits 15:0

    printf("0x%.16llX\n", descriptor);
}

int
main(void)
{
    create_descriptor(0, 0, 0);
    create_descriptor(0, 0x000FFFFF, (GDT_CODE_PL0));
    create_descriptor(0, 0x000FFFFF, (GDT_DATA_PL0));
    create_descriptor(0, 0x000FFFFF, (GDT_CODE_PL3));
    create_descriptor(0, 0x000FFFFF, (GDT_DATA_PL3));

    return 0;
}
```

See Also

Articles

- Global Descriptor Table

Threads

External Links

Retrieved from "http://wiki.osdev.org/index.php?title=GDT_Tutorial&oldid=18813"

Categories: Level 1 Tutorials | Tutorials | X86 CPU