Global Descriptor Table

From Wikipedia, the free encyclopedia

The **Global Descriptor Table** or *GDT* is a data structure used by Intel x86-family processors starting with the 80286 in order to define the characteristics of the various memory areas used during program execution, including the base address, the size and access privileges like executability and writability. These memory areas are called *segments* in Intel terminology.

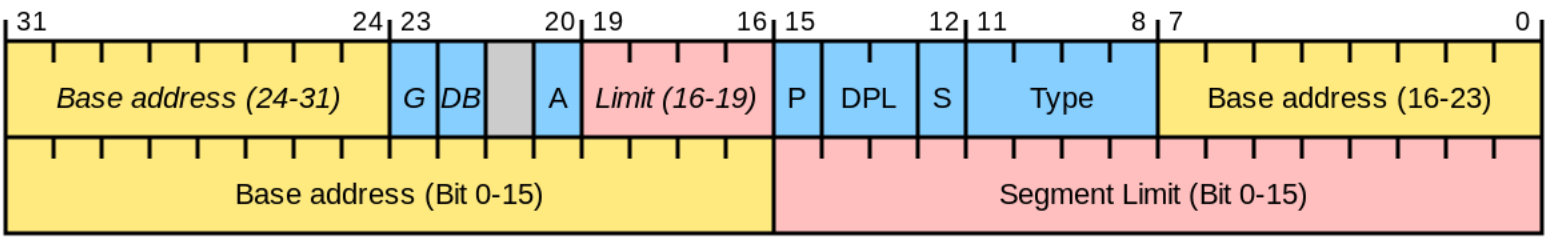
The *GDT* can hold things other than segment descriptors as well. Every 8-byte entry in the *GDT* is a descriptor, but these can be Task State Segment (or TSS) descriptors, Local Descriptor Table (LDT) descriptors, or Call Gate descriptors. The last one, Call Gates, are particularly important for transferring control between x86 privilege levels although this mechanism is not used on most modern operating systems.

There is also an LDT or *Local Descriptor Table*. The LDT is supposed to contain memory segments which are private to a specific program, while the GDT is supposed to contain global segments. The x86 processors contain facilities for automatically switching the current LDT on specific machine events, but no facilities for automatically switching the GDT.

Every memory access which a program can perform always goes through a segment. On the 386 processor and later, because of 32-bit segment offsets and limits, it is possible to make segments cover the entire addressable memory, which makes segment-relative addressing transparent to the user.

In order to reference a segment, a program must use its index inside the GDT or the LDT. Such an index is called a *segment selector* or *selector* in short. The selector must generally be loaded into a *segment register* to be used. Apart from the machine instructions which allow one to set/get the position of the GDT (and of the Interrupt Descriptor Table) in memory, every machine instruction referencing memory has an implicit Segment Register, occasionally two. Most of the time this Segment Register can be overridden by adding a Segment Prefix before the instruction.

Loading a selector into a segment register automatically reads the GDT or the LDT and stores the properties of the segment inside the processor itself. Subsequent modifications to the GDT or LDT will not be effective unless the segment register is reloaded.



Contents

GDT in 64-bit

The GDT is still present in 64-bit mode; a GDT must be defined, but is generally never changed or used for segmentation. The size of the register has been extended from 48 to 80 bits, and 64-bit selectors are always "flat" (thus, from 0x0000000000000000 to 0xFFFFFFFFFFFFFFFF). However, the base of FS and GS are not constrained to 0, and they continue to be used as pointers to the offset of items such as the process environment block and the thread information block.

If the System bit (4th bit of the Access field) is cleared, the size of the descriptor is 16 bytes instead of 8. This because, even though code/data segments are ignored, TSS are not, but the TSS pointer can be 64bit long and thus the descriptor needs more space to insert the higher dword of the TSS pointer.

64-bit versions of Windows forbid hooking of the GDT; attempting to do so will cause the machine to bug check.[1]

Local Descriptor Table

The **Local Descriptor Table** (LDT) is a memory table used in the x86 architecture in protected mode and containing memory segment descriptors: start in linear memory, size, executability, writability, access privilege, actual presence in memory, etc.

The LDT is the sibling of the Global Descriptor Table (GDT) and defines up to 8192 memory segments accessible to programs - note that unlike the GDT, the zeroeth entry is a valid entry, and can be used like any other LDT entry. Also note that unlike the GDT, the LDT cannot be used to store certain system entries: TSSs or LDTs. Call Gates and Task Gates are fine, however.

History

On x86 processors not having paging features, like the Intel 80286, the LDT is essential to implementing separate address spaces for multiple processes. There will be generally one LDT per user process, describing privately held memory, while shared memory and kernel memory will be described by the GDT. The operating system will switch the current LDT when scheduling a new process, using the LLDT machine instruction or when using a TSS. On the contrary, the GDT is generally not switched (although this may happen if virtual machine monitors like VMware are running on the computer).

The lack of symmetry between both tables is underlined by the fact that the current LDT can be automatically switched on certain events, notably if TSS-based multitasking is used, while this is not possible for the GDT. The LDT also cannot store certain privileged types of memory segments (e.g. TSSes). Finally, the LDT is actually defined by a descriptor inside the GDT, while the GDT is directly defined by a linear address.

Creating shared memory through the GDT has some drawbacks. Notably such memory is visible to every process and with equal rights. In order to restrict visibility and to differentiate the protection of shared memory, for example to only allow read-only access for some processes, one can use separate LDT entries, pointed at the same physical memory areas and only created in the LDTs of processes which have requested access to a given shared memory area.

LDT (and GDT) entries which point to identical memory areas are called *aliases*. Aliases are also typically created in order to get write access to code segments: an executable selector cannot be used for writing. (Protected mode programs constructed in the so-called *tiny* memory model, where everything is located in the same memory segment, must use separate selectors for code and data/stack, making both selectors technically "aliases" as well.) In the case of the GDT, aliases are also created in order to get access to system segments like the TSSes.

Segments have a "Present" flag in their descriptors, allowing them to be removed from memory if the need arises. For example, code segments or unmodified data segments can be thrown away, and modified data segments can be swapped out to disk. However, because entire segments need to be operated on as a unit, it is necessary to limit their size in order to ensure that swapping can happen in a timely fashion. However, using smaller, more easily swappable segments means that segment registers must be reloaded more frequently which is itself a time-consuming operation.

Modern usage

The Intel 80386 microprocessor introduced *paging* - allocating separate physical memory pages (themselves very small units of memory) at the same virtual addresses, with the advantage that disk paging is far faster and more efficient than segment swapping. Therefore, modern 32-bit x86 operating systems use the LDT very little, primarily to run legacy 16-bit code.

Should 16-bit code need to run in a 32-bit environment while sharing memory (this happens e.g. when running OS/2 1.x programs on OS/2 2.0 and later), the LDT must be written in such a way that every *flat* (paged) address has also a selector in the LDT (typically this results in the LDT being filled with 64 KiB entries). This technique is sometimes called *LDT tiling*. The limited size of the LDT means the virtual flat address space has to be limited to 512 megabytes (8191 times 64 KiB) - this is what happens on OS/2, although this limitation was fixed in version 4.5. It is also necessary to make sure that objects allocated in the 32-bit environment do not cross 64 KiB boundaries; this generates some address space waste.

If 32-bit code does not have to pass arbitrary memory objects to 16-bit code, e.g. presumably in the OS/2 1.x emulation present in Windows NT or in the Windows 3.1 emulation layer, it is not necessary to artificially limit the size of the 32-bit address space.

References

1. "Patching Policy for x64-Based Systems" (http://www.microsoft.com/whdc/driver/kernel/64bitPatching.mspx). "If the operating system detects one of these modifications or any other unauthorized patch, it will generate a bug check and shut down the system."

External links

- Intel Architecture Software Developer's Manual Volume 3: System Programming (http://download.intel.com/design/PentiumII/manuals/24319202.pdf)
- GDT Tutorial (http://www.osdever.net/bkerndev/Docs/gdt.htm)
- GDT Tutorial (http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html) by James Molloy, slightly more practically oriented.
- GDT Table (http://wiki.osdev.org/Global_Descriptor_Table) at OSDev.org

- GDT Tutorial (http://wiki.osdev.org/GDT_Tutorial) at OSDev.org

Retrieved from "http://en.wikipedia.org/w/index.php?title=Global_Descriptor_Table&oldid=623971141"

Categories: X86 architecture | Memory management
