Protected-Mode Memory Management

by Yariv Kaplan

When the processor is running in protected-mode, two mechanisms are involved in the memory translation process: **Segmentation** and **Paging**. Although working in tandem, these two mechanisms are completely independent of each other. In fact, the paging unit can be disabled by clearing a single bit in an internal processor register. In this case, the linear addresses which are generated by the segmentation unit pass transparently through the paging unit and straight to the processor address bus.

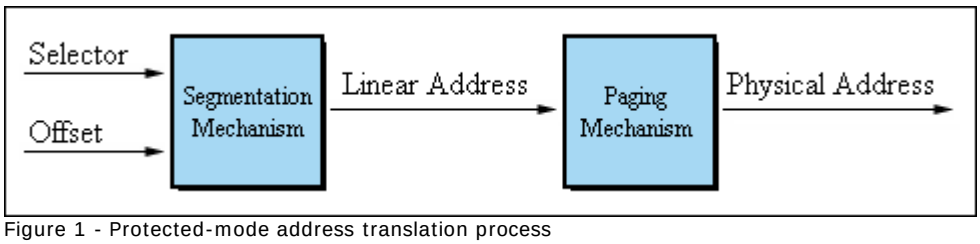

Figure 1 - Protected-mode address translation process

Segmentation

The role of the segmentation unit is the same as on the 8086 processor. It allows the operating system to divide programs into logical blocks and place each block in a different memory region. This makes it possible to regulate access to critical sections of the application and help identify bugs during the development process. The implementation of the segmentation unit on the 80386 (and above) is simply an extension of the old 8086 unit. It includes several new features such as the ability to define the exact location and size of each segment in memory and set a specific privilege level to a segment which protects its content from unauthorized access.

Not only real-mode applications use segment registers for accessing memory. The same process takes place under protected-mode. However, there are several differences which should be considered. First, there is a slight change in terminology. Under protected-mode, segment registers receive the name **Selectors** which reflects their new role in the memory translation process. Although still 16-bit in size, their interpretation by the processor is inherently different. Figure 2 presents the structure of a selector along with the various bit-fields which comprise it.

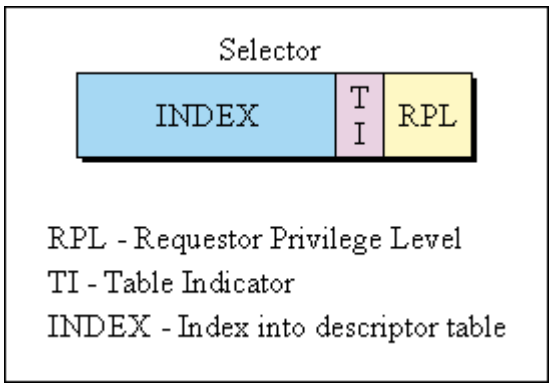

Figure 2 - Internal composition of a selector

Instead of shifting segment registers (selectors) four bits to the left and adding an offset (like in real-mode), the processor treats each selector as an index to a **Descriptor Table**.

Descriptor Tables

Descriptor tables reside in system memory and are used by the processor to perform address translation. Each entry in a descriptor table is 8 bytes long and represents a single segment in memory. A descriptor entry contains both a pointer to the first byte in the associated segment and a 20-bit value which represents the size of the segment in memory. Several other fields contain special attributes such as a privilege level and the segment's type. Figure 3 presents the exact structure of a descriptor entry along with a description of each of its internal fields.

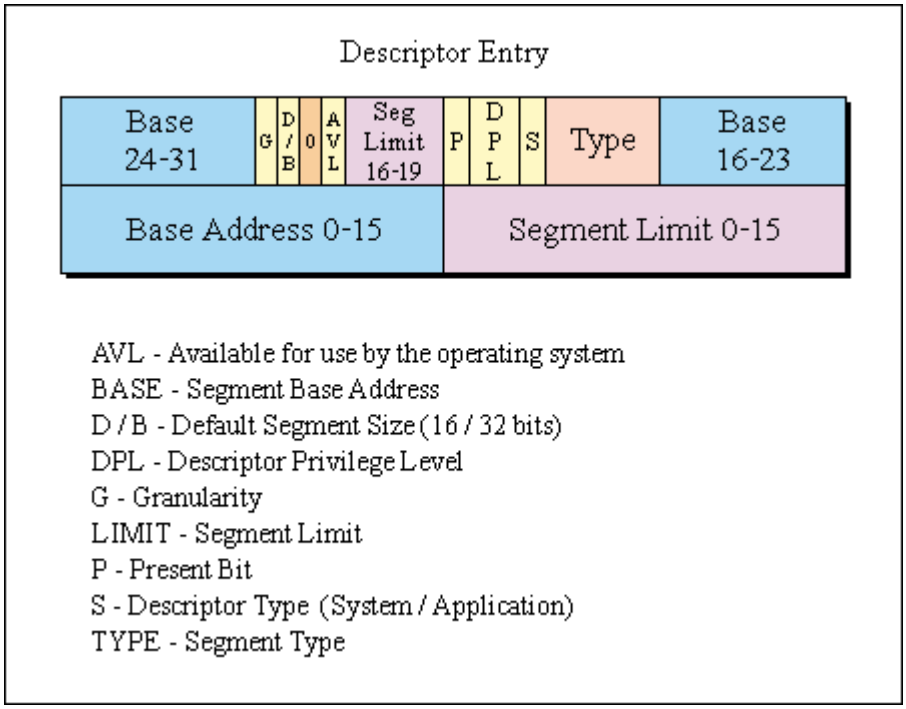

Figure 3 - Structure of a descriptor entry

Table 1 contains a complete list of all descriptor fields and their functionality.

Field	*Designated Role*
BASE	**Segment Base Address (32-bits)** This field points to the segment's starting location in the 4GB linear address space.
D/B	**Segment Size Bit** When the descriptor entry describes a code segment, this bit is used to specify the default length of operands and addresses. When the bit is set, the processor assumes a 32-bit segment. When the bit is clear, a 16-bit segment is assumed. When the descriptor entry describes a data segment, this bit is used to control the operation of the stack. When this bit is set, stack operations use the ESP register. When this bit is clear, stack operations use the SP register.
DPL	**Descriptor Privilege Level (2-bits)** This field defines the segment privilege level. It is used by the protection mechanism built into the processor to restrict access to the segment.
G	**Granularity Bit** This bit controls the resolution of the segment limit field. When this bit is clear, the resolution is set to one byte. When this bit is set, the resolution is set to 4KB.
LIMIT	**Segment Limit (20-bits)** This field determines the size of the segment in units of one byte (When the granularity bit is clear) or in units of 4KB (When the granularity bit is set).
P	**Segment Present Bit** This bit specifies whether the segment is present in memory. When this bit is clear, a segment-not-present exception is generated whenever a selector for the descriptor is loaded into one of the segment registers. This is used to notify the operating system of any attempt to access a segment which has been swapped to disk (virtual memory support) or which was not previously allocated (a protection violation).
S	**Descriptor Type Bit** This bit determines whether this is a normal segment or a system segment. When this bit is set, this is either a code or a data segment. When this bit is clear, this is a system segment.
Type	**Segment Type (4-bits)** When the descriptor entry describes a code segment, this field determines the type of the segment: execute-only or execute-read, conforming or non-conforming. When the descriptor entry describes a data segment, this field determines the type of the segment: read-only or read-write, expand-down or expand-up. Accessing an expand-up segment with an offset which exceeds the segment limit value, causes an exception. The limit field in expand-down data segments is treated differently by the processor. Offsets which cause an exception to occur in expand-up segments are valid in expand-down segments. An access into an expand-down segment must be done with an offset larger than the segment limit

or else an exception is generated.
Decreasing the segment limit value in an expand-down segment causes memory to be allocated at the bottom of the segment. This is very useful for stacks since they tend to grow toward lower memory addresses.

Table 1 - Description of bit-fields in a descriptor entry

Since each selector points to a specific descriptor entry, there is a one to one relationship between selectors and segments in memory. This concept is demonstrated in the following figure.

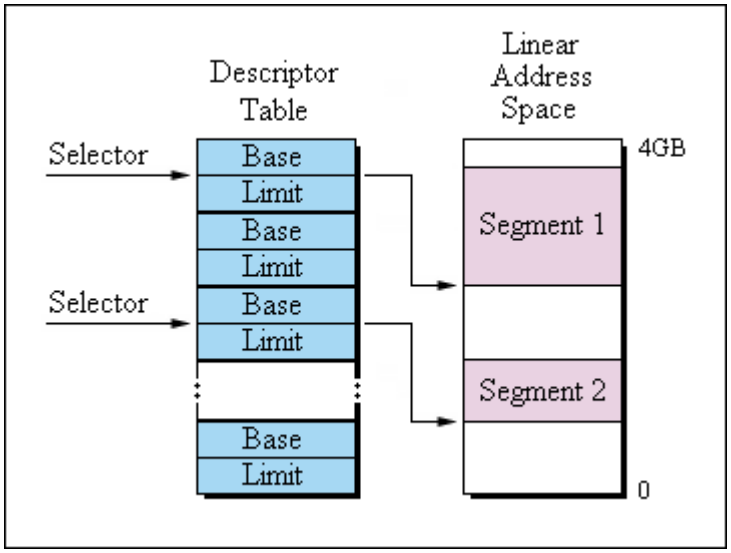

Figure 4 - Relationship between selectors and segments

As figure 5 shows, linear address calculation (which can be the physical address if paging is disabled) is done by using the selector as an index to the descriptor table, getting the base address of the segment, and adding the offset.

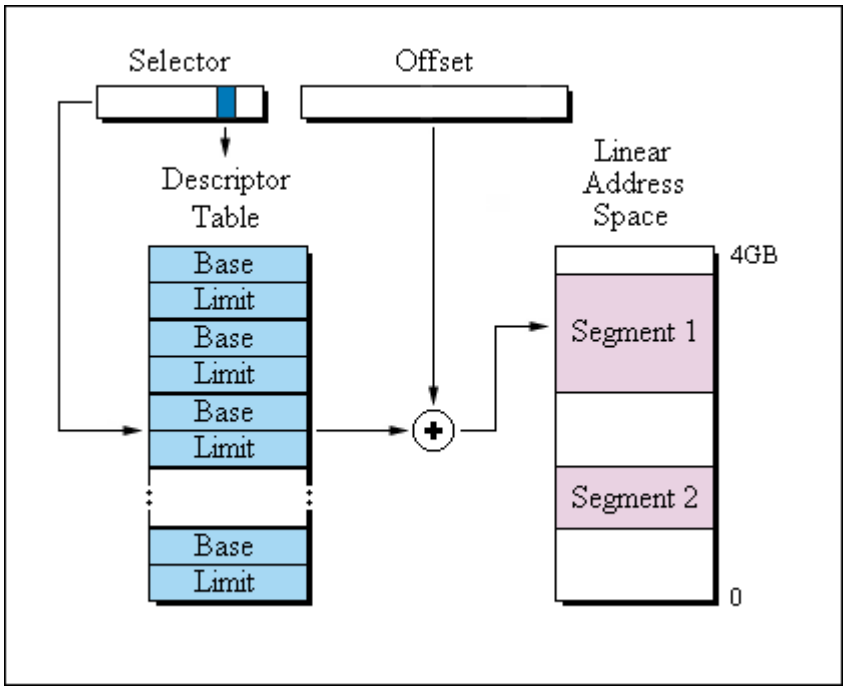

Figure 5 - Virtual to linear address translation

Two types of descriptor tables are used by the processor when working in protected-mode. The first is known as the **GDT** (Global Descriptor Table) and is used mainly for holding descriptor entries of operating system segments. The second type is known as the **LDT** (Local Descriptor Table) and contains entries of normal application segments (although not necessarily). During initialization, the kernel creates a single GDT which is kept in memory until either the operating system terminates or until the processor is switched back to real-mode.

Whenever the user starts an application, the operating system creates a new LDT to hold the descriptor entries which represent the segments used by the new task. This makes it possible for the operating system to isolate each task's address space by enabling a different LDT whenever a task switch occurs. Bugs and other errors in the application cannot affect other running processes and are limited in scope to the currently mapped memory segments.

Note that not all operating systems behave exactly as described above (for instance, all Windows applications share a single LDT). However, this is the recommended programming practice as offered by Intel.

When looking for a specific descriptor entry, the addressing unit in the processor uses the TI bit (which is part of the selector) to decide which descriptor table should be used (the GDT or the currently active LDT). Figure 6 shows this process in clarity.

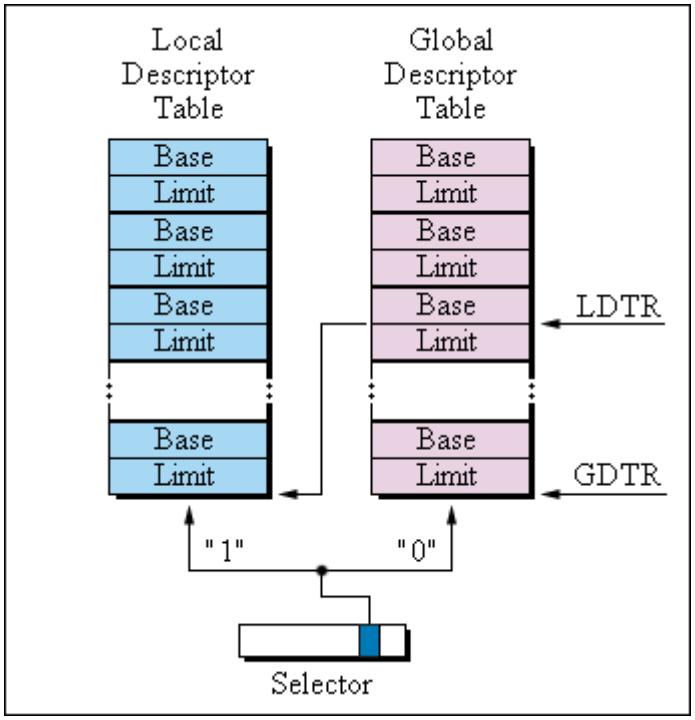

Figure 6 - The Table Indicator bit

The linear address and size of the GDT itself are stored in a special processor register called **GDTR**. During bootstrap, the operating system initializes this register to point to the newly created GDT and does not modify its value during the entire session. In much the same manner, the **LDTR** register contains the size and position of the currently active LDT in memory (In fact, it serves as a selector to the GDT, pointing to a descriptor entry which contains all the relevant information). The fact that the LDTR register serves as a selector to the GDT rather than contain specific values, makes it possible for the operating system to switch easily between different LDTs and prevent inter-task memory corruptions.

In order to clarify the operation of the segmentation mechanism in protected-mode, I'll use a small C code snippet:

```
char far *pChar;

pChar = GlobalAlloc(GMEM_FIXED, 100);
pChar[50] = 'A';
```

This piece of code asks Windows to allocate a 100 bytes buffer and store its address in a far pointer called pChar. If you examine the code inside GlobalAlloc, you will see that Windows allocates a selector in its LDT whenever you call this function in your application. So pChar actually contains both a selector and an offset. To modify the allocated buffer, the processor uses the LDTR register to index its GDT and find the currently active LDT descriptor. Then, it accesses the LDT by using the selector part of pChar as an index to retrieve the segment base address. The offset (50) is added to the retrieved segment base address and the letter 'A' is written to the calculated memory location.

Note that the information presented above applies only to the 16-bit implementation of Windows. The behavior of GlobalAlloc on the Win32 platforms is inherently different.

Paging

Paging is a mechanism which helps the operating system to create unique virtual (faked) address spaces while it also has a major role in memory simulation using disk space - A process commonly known as **Virtual Memory** support.

The 32-bit linear address generated by the segmentation unit can be optionally fed into the paging unit to undergo a second address manipulation process. There is no mathematical correlation between a linear address and its associated physical counterpart but instead, special tables in memory known as the **Page Tables** assist the paging unit in transforming the input linear address into a physical address which is sent to the processor bus.

Applications live inside a 4GB linear address space and have no indication of how physical memory is organized. This has numerous benefits since no application can see or modify other applications data structures - One of the features required by most multitasking operating systems.

The paging unit treats the linear and physical address spaces as a collection of consecutive 4KB pages (The Pentium and Pentium Pro can also handle 4MB pages). A linear page can be mapped to any of the physical pages or it can be marked as non-present to make it sensitive to memory accesses. Trying to read or modify a non-present page causes the processor to generate a **Page Fault Exception** (exception 0xEh) which is usually handled by the operating system internal code.

This is exactly the essence of virtual memory. When programs consume all available memory, the operating system attempts to free memory by swapping least recently used (LRU) pages of memory to disk. The swapped pages are marked as non-present so when they are later accessed by their owning application, they would be automatically reloaded from disk by the operating system (The code handling the page exception is responsible for loading these pages and mapping them back to physical memory). You can think of this process

as if the operating system "steals" memory from background tasks in order to give it to the currently active application.

The real impressive characteristic of the virtual memory mechanism is that it is completely transparent to application code. After loading a page from disk and remapping it to physical memory, the operating system reexecutes the instruction which caused the page fault exception to occur, so the running application is not even aware of the fact that part of its memory was stored temporarily on disk.

Before enabling the paging unit, the operating system must construct a table in memory known as the page directory table. The 1024 DWORD entries of the page directory table hold physical addresses of another set of paging tables called page tables. Page tables participate in the last stage of the address translation process since they actually contain the physical addresses of the 4KB chunks of memory.

A close examination of the paging process reveals that the processor breaks the linear address into three components before turning it into a physical one. The top 10 bits of the linear address are used by the processor to index the page directory table. The processor retrieves the entry from the table and uses it to find the physical address of a page table. The next 10 bits of the linear address serve as an index into the corresponding page table. By adding the page table entry value (which represents the physical address of a physical page in memory) and the 12 lower bits of the linear address (the offset into the page), the processor can locate the requested physical address and send it to its address bus. Figure 7 shows the internal structure of a linear address and its interpretation by the processor.

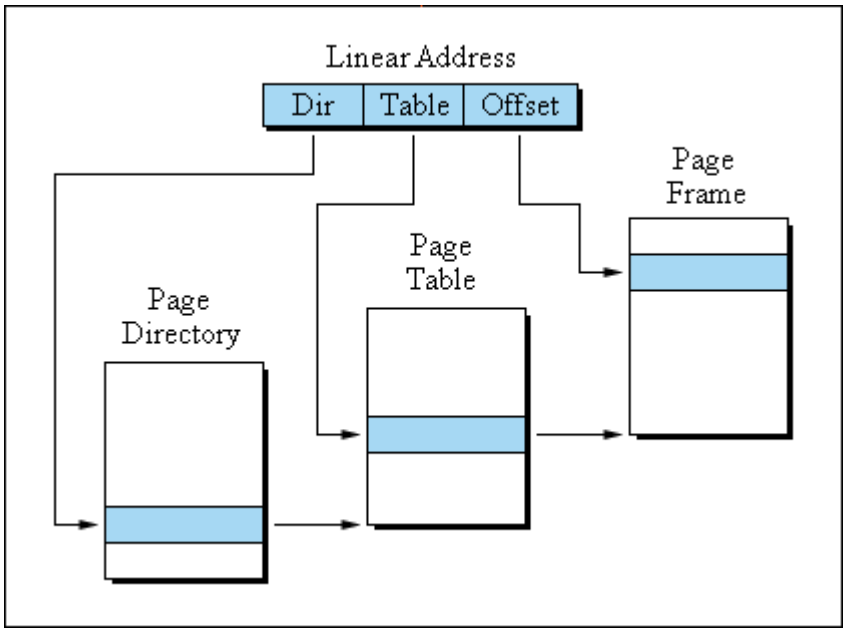

Figure 7 - Components of a linear address

The two-level address indirection mechanism (page directory and page table) was chosen to solve the problem related to the memory occupied by the page tables themselves. Since each page table (including the page directory table) occupies 4KB of physical memory (1024 entries * 4 bytes each), 1024 page tables (one page table for each entry in the page directory) would require 4MB of memory - An awful waste of valuable memory bytes. The use of the page directory table solves this problem since each entry (which represents a 4KB page table) can be

marked as non-present. Non-present page tables eliminate the need to allocate a chunk of 4KB physical memory to hold their content.

Under Windows 95, each running 32-bit process is mapped to the 4MB-2GB range of linear address space. Although 32-bit applications share the same linear address space, Windows loads each application to a different physical memory location (if enough memory is available). Whenever a task switch occurs, Windows modifies its page tables to reflect the new linear to physical mapping scheme and swaps least recently used pages of memory to disk.

This concludes our exploration of the memory management unit on the 80x86 processors. The next article will cover the complex issue of exceptions and interrupts in protected-mode.