Author: **Steve Gorman**

Sr. Software Engineer

Intel Corporation

Title: Overview of the Protected Mode Operation of the Intel Architecture

Abstract: As the Intel Architecture moves off the desktop into other computing applications, developers need to design their systems to take full advantage of Intel architecture's performance and extended addressing. This paper will provide for an introduction into the architecture's internal data structures that enable the extended address space, protection, multi-tasking, paging, and virtual-86 mode operation.

To take full advantage of the Intel386™ processor architecture's performance and extended addressing system developers need to understand the various features provided by the processor. This paper will provide for an introduction into the architecture's internal data structures that enable the extended address space, protection, multitasking, paging, and virtual-86 mode operation.

Addressing Memory

With the Intel Architecture (regardless of mode) all memory references are offsets from a base address. The base address, determined by the contents of a segment register, is called a selector.

The segment registers CS, SS, DS, ES, FS and GS are each used to reference a particular kind of segment characterized as "code," "data," or "stack." The CS register indicates which segment is the currently executing code. The instruction pointer (EIP) is the offset from the beginning of the code segment of where the next instruction fetched occurs. Intersegment control transfers (calls, jumps, returns, interrupts and exceptions) modify the contents of the CS register. All stack operations use the SS register to locate the stack segment. The DS, ES, FS, and GS registers are used to access up to four separate data areas (FS and GS were made available on the 80386 processor generation).

For real mode or an 8086/80186 processor the selector value (loaded into a segment register) represents the upper 16-bits of the 20-bit linear address. For protected mode the selector is an index into a descriptor table. The referenced descriptor, pointed to by the selector, holds the full 32-bit base portion of the memory address, as well as other information (Figure 1). Until the segment register is loaded with a new value, all future memory references using that segment add to the base a 32-bit offset to determine the linear address. If the page unit is enabled, the 32-bit linear address is translated into a 32-bit physical address. If the page unit is not enabled, the 32-bit linear address is the physical address. Segments in protected mode are not limited to only 64KB as in the 8086 processor, but can be up to 4GB in size.

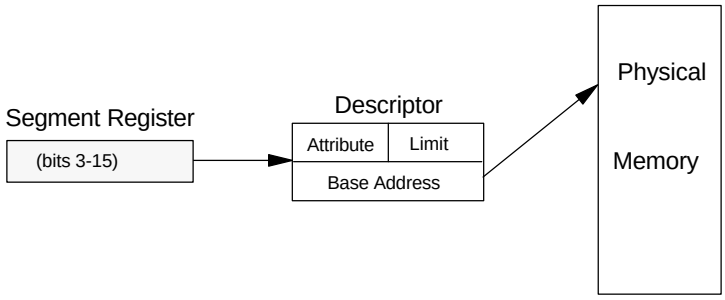


Figure 1. Protected Mode Addressing

Descriptors

There are two classes of descriptors: system and segment. For addressing memory we need only to concern ourselves with segment descriptors. Discussion of the system descriptors like gates, tasks and LDTs are handled where appropriate. All descriptors are 8-bytes each and reside in one of the different descriptor tables (see Descriptor Tables section). The segment descriptor describes the attributes of each segment. That is where the segment begins in memory or its base address, the size of the segment, the type of segment, and the access rights (Figure 2 & Table 1).

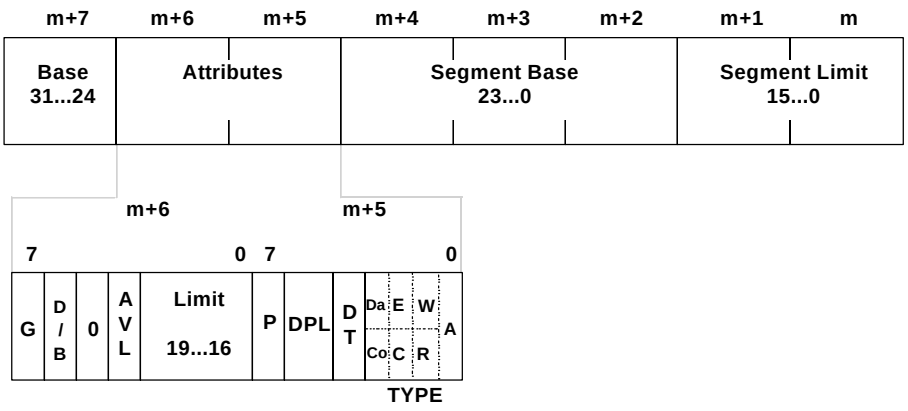


Figure 2. Segment Descriptor Layout

Bit Name	Bit Meaning
G	Granularity bit, used to determine if the limit is check on byte or 4KB page granularity.
D/B	Default/Big bit, for code segments represent the default operand size (16 or 32 bit). For expand down data segments it affects the operation of limit checking (this maintains compatibility with 286 protected mode expand down segments).
AVL	Available bit, this bit is available for use by the system designer.
P	Present bit, indicates that the specified segment is present in memory. Can be used to help with the implementation of virtual memory system.
DPL (2-bits)	Descriptor Privilege Level, Used by the protection mechanism.
DT	Descriptor Type, for systems descriptors DT=1 for segment descriptors DT=0.
Code/Data	Segment Type bit, indicates if the segment is a Code (=1) or Data (=0) segment. The setting of this bit effects the meaning of bits 1 and 2
E (Data)	If E=1 the data segment is an expand down data segment (typically used for stacks).
W (Data)	If W=1 the data segment is both read and write, else segment is read-only.
C (Code)	If C=1 then the code segment is a conforming code segment
R (Code)	If R=1 then the code segment is executable and readable, else it is execute-only.
A	Access bit, The processor automatically sets this bit whenever a descriptor is referenced. The bit is cleared by software.

Table 1. Segment Descriptor Attributes

Descriptor Tables

There are three types of descriptor tables; the Global (GDT), Local (LDT) and Interrupt (IDT). The GDT, pointed to by GDTR, holds up to 8191 objects (GDT 0 is reserved) that are accessible across all tasks. The LDT, pointed to by LDTR, holds up to 8192 objects (LDT 0 is not reserved) that are local to each task. The IDT, pointed to by the (need I say) IDTR, holds up to 256 gates (more on gates later). Much like with the 8086, the IDT holds the destinations for the various interrupts. Each interrupt, software or hardware, has a vector number associated with it. This vector number serves as the index into the IDT. All three of these tables are created by software and reside in the memory space.

Each segment register has associated with it a descriptor that is internal to the chip. When loading a segment register with a selector value, the processor retrieves the descriptor from the descriptor table and loads it into a descriptor internal to the processor for that segment register.

Selectors

As mentioned before, a selector is the term used to describe either the contents of a segment register or a value to be loaded into a segment register. The selector value has three parts; the index, table indicator (TI) and requester's privilege level (RPL). The index tells us which descriptor to fetch from the descriptor table. The table indicator tells us which descriptor table to look into, the LDT (TI=1) or GDT (TI=0). The requester's privilege level is covered when we talk about protection. The following is a flow chart of what happens when loading a segment register with a selector value.

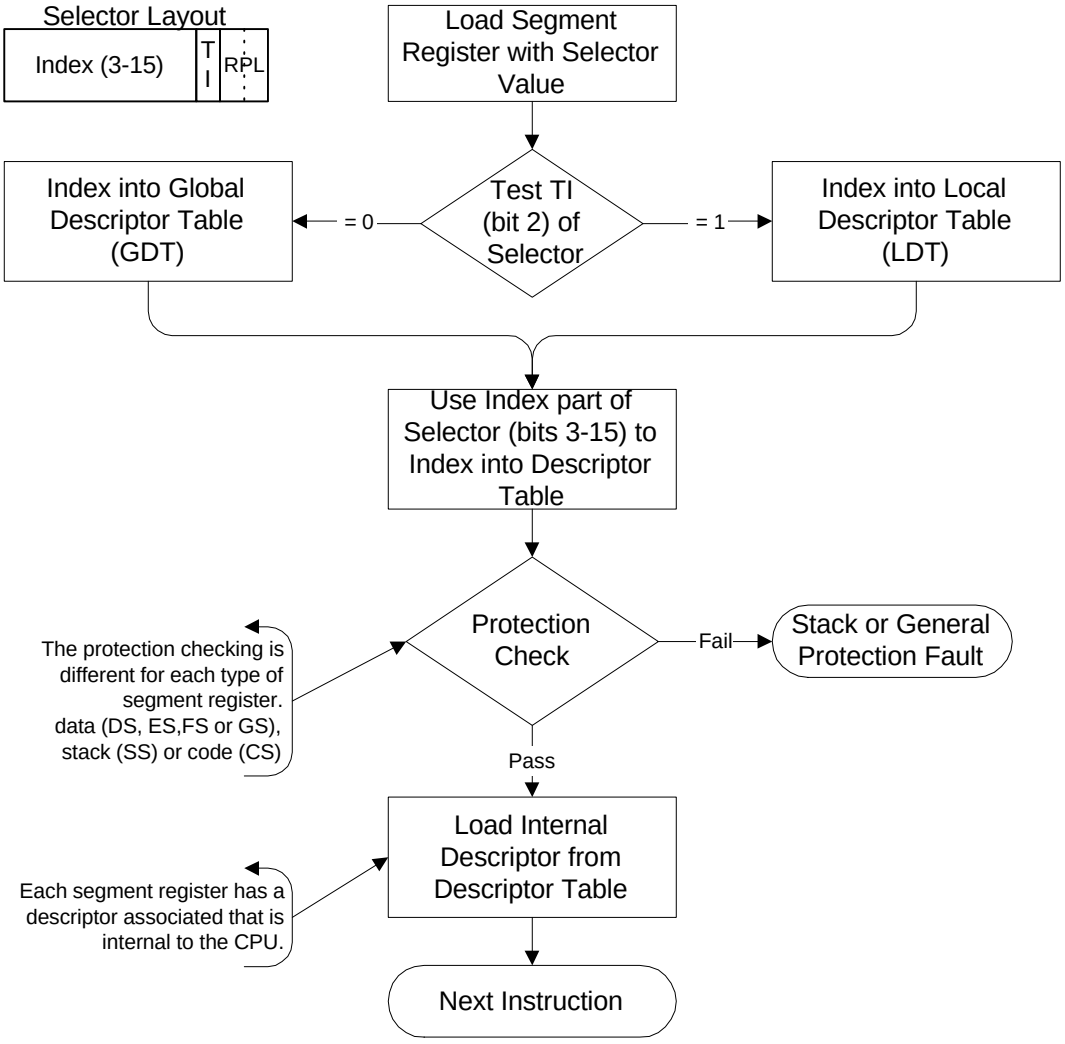


Figure 3. Loading a Segment Register Execution Flow

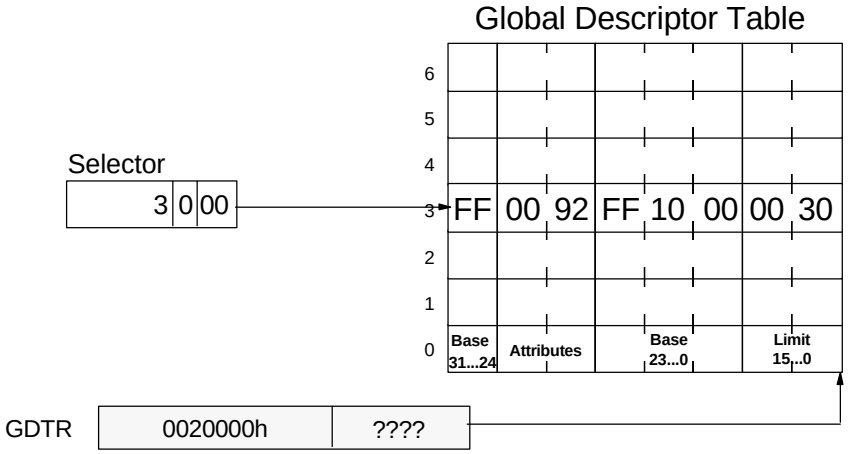
Back to Addressing

Now that we know about the three key pieces in determining an address, let's see how it's done (we will assume that we do not have a protection problem for now). A segment register is loaded with a selector value. The selector value is broken into its three parts (index, TI and RPL). A descriptor is fetched from the appropriate descriptor table using the TI and index (the RPL is used for protection

checking). The descriptor contents are then loaded into the internal descriptor set aside for that segment register. Now whenever making a memory access using the given segment register, add the offset to the base address in the internal descriptor.
Example:

```
Mov     AX,     18h
Mov     DS,     AX
Mov     ESI,    22h
Mov     EAX, [ESI]              : What Address is referenced?
```

1) Break down the selector value: 18h = 0000000000011 0 00
   Index = 3
   Table Indicator (TI) indicates the descriptor is in the GDT.
   Requester's privilege level (RPL), not used in calculating the address.
2) Find the indexed descriptor in the appropriate descriptor table, TI = 0 so look in the GDT



3) Extract the base address information from the indicated descriptor: 0FFFF1000h.
4) Add the offset to the base: ESI = 22h, 0FFFF1000h + 22h = 0FFFF1022h is the referenced address..

These system data structures (selectors, descriptors, and descriptor tables) are key to the operation of the processor in protected mode.

Protection

The protection mechanism provided by the architecture probably causes the most fear, but should be the most praised feature. The purpose of protection features is to help detect and identify bugs as well as to develop secure, reliable operating systems. It is important to remember that not all the provided protection features need to be utilized in order to take advantage of the 32-bit architecture.

If we look back at the segment descriptor you will see information in the descriptor that relates to more than just its base address in memory (Figure 2 & Table 1). The additional information provided is primarily for the implementation of a protected system:

- How programs can access different types of segments,
- ensuring accesses within the limits of the segment (limit checking),
- maintaining privilege levels or who has access to a segment,
- and controlling access to privileged instructions.

Type Checking

Type checking assures that a segment is being used as intended. In a code segment for execution, writes do not occur to read-only segments. Therefore, some simple and obvious rules apply based on the type of descriptor that is being referenced (the following is from the *i486 Microprocessor Programmer's Reference Manual*).

1. When a selector for a descriptor is loaded into a segment register. Certain segment registers can contain only certain descriptor types; for example:
    - The CS register only can be loaded with a selector for an executable segment.
    - Selectors of executable [code] segments which are not readable cannot be loaded into data-segment registers [DS, ES, FS, GS].
    - Only selectors of a writable data segment can be loaded into the SS register.
2. Certain segments can be used by instructions only in certain predefined ways; for example:
    - No instruction may write into an executable [code] segment.
    - No instruction may write into a data segment if the writable bit is set.
    - No instruction may read an executable [code] segment unless the readable bit is not set.

These rules provide the capability for programmers to put into place an easy way to ensure that segments are being used as intended. This checking is performed whenever an attempt is made to load a segment register with a new value. If any of these rules are violated by an application, a general protection fault (interrupt 13) will occur.

Let's give an example: One of our tasks (T1) gathers data from a sensor and maintains that data in a table. Another task (T2) needs access to the data for processing, but we want T1 to be the only one that can modify the data. With an 8086 or non-protected/segmented architecture, T1 would most likely have to either provide T2 a copy of the table upon request or provide T2 a pointer to the table. The problem with a copy is that the copy is out of date. The pointer solution, would provide T2 the ability to modify the data, even if by accident. Neither are good solutions, but using one of the protection features of the Intel architecture, T1 could create two separate descriptors for the table. One descriptor for itself (T1) that has both read and write access and one for T2 that is read-only. Segment aliasing occurs when the same memory is referenced by two or more descriptors. What if T1 only wanted to provide T2 with access to only part of the table (check out limit checking)?

Here is a small piece of example code that takes advantage of segment aliasing. The routine loads the IDT table with an interrupt routine using an alias for the IDT.

```
typedef unsigned short   WORD;        /* 16-bit value */
typedef unsigned long    DWORD;       /* 32-bit value */
typedef _Packed struct {
    WORD  OffsetLow;
    WORD  Selector;
    WORD  Attrib;
    WORD  OffsetHigh;
} GATE_DESCRIPTOR;

typedef union {
    GATE_DESCRIPTOR far *lpGateDescrpt;
    struct {
        DWORD Offset;
        WORD Selector;
    }x;
```

```
}MK_DESCRPT_PTR;


/*    SetInterruptVector
      Description:
            Loads the interrupt vector table with the address of the
            interrupt routine.  The vector table entry number is
            determined by the vector number.

      Parameters:
            InterProc   Address of interrupt function, will be loaded
                        into the interrupt table.
            Vector      Which IDT vector to initialize.
            ISR_Type    Specifies if the gate descriptor should be of
                        type trap  or gate.
      Returns:    None

      Assumptions:
            Compiler supports far, interrupt and __segment keywords
            Code executes at privilege level 0
            GDT has a Alias for the IDT defined by IDT_ALIAS
*/
void SetInterruptVector(void (far interrupt *IntrProc)(void), int
Vector, int IntrType)
{
    MK_DESCRPT_PTR idt;

      /*** Create pointer to IDT[vector] using IDT alias ***/
    idt.x.Selector = IDT_ALIAS << 3;
      /* Offset from beginning of IDT to write descriptor */
    idt.x.Offset   = Vector*8;

      /* Get selector value  for gate descriptor */
    idt.lpGateDescrpt->Selector = (__segment)IntrProc;
      /* Set field type */
    if(IntrType == INTERRUPT_ISR)
        idt.lpGateDescrpt->Attrib = INTR_TYPE;
    else
        idt.lpGateDescrpt->Attrib = TRAP_TYPE;

      /* Set gate descriptor offset to point to ISR */
    idt.lpGateDescrpt->OffsetLow = (WORD)((DWORD)IntrProc);
    idt.lpGateDescrpt->OffsetHigh = (WORD) ((DWORD)IntrProc >> 16);
}
```

Limit Checking

The concept of limit checking is very simple. Limit checking prevents a program from accesses outside the bounds of a segment. Limit checking is automatically performed by the processor every time a memory reference is made to either a code or data segment. If a segment is defined to be 22-bytes long, an application should not be allowed to use that segment to access data above base+21h, precisely what limit checking prevents. If you noticed the limit part of the descriptor is only 20-bits (1 MB worth) and segments can be up to 4GB in size. This is where the descriptors granularity (G) bit comes into play. If the G-bit = 0 the limit is checked to the byte, making the maximum size of a segment 1MB. If the G-bit = 1, limit checking is performed with a granularity of 4KB (4KB*1MB = 4GB), thus allowing segments up to 4GB in size (Figure 4).

- G-bit = 0, Segment upper bound = limit, MAX segment size = 1MB
- G-bit = 1, Segment upper bound = SHL(limit,12,1), MAX segment size = 4GB.

For expand-down data segments (typically stacks) the limit is interpreted differently. Valid accesses are when the referenced offset is greater than the limit (a limit of 0 is the maximum expand-down segment size).

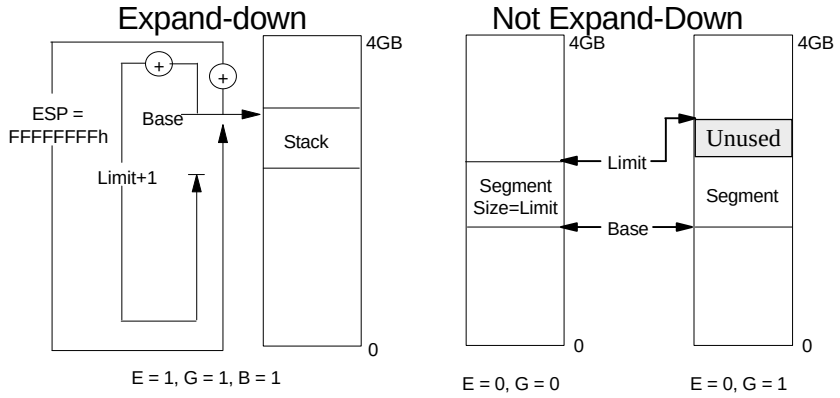


Figure 4. Limit Checking

Since pushing values onto the stack decrement the ESP, once ESP reaches Limit a stack fault (interrupt 12) will occur. For all other limit checking, violations result in a general protection fault. Limit checking is essentially disabled by setting the G-bit and the limit field to its maximum value.

Privilege Levels

The Intel Architecture provides a way of controlling what types of applications that have access to; system objects, segments, and memory or I/O mapped hardware. This is done in the protection mechanism by having up to four different privilege levels, numbered 0 to 3. A lower number implies greater privileges. There is no explicit way to turn off privilege levels, but it can be done by assigning all objects and segments a privilege level of 0. In this paper, I'll tell you how it works and leave it up to your imagination to figure out how you want to use it.

Assignments of privilege levels is a key element in improving the reliability of operating systems (OS), specially operating systems that support user-created applications. For example, giving the OS the highest privilege level (0) and the applications the lowest (3), prevents applications from accessing OS objects. For those systems that can trust all the task or applications the OS could choose to run, everything can be set to a privilege level of 0, essentially eliminating privilege levels.

What Effects Privilege Level Checking?

Three data structures contain the privilege level information:

- CPL, Current Privilege Level is the least two bits of the CS register. This is the privilege level of the currently executing code.
- DPL, Descriptor Privilege Level is the two bit field in a descriptor.
- RPL, Requestor's Privilege Level is the least two bits of a selector value. The RPL should represent the privilege level of code that created the selector.

These three data structures are checked whenever an attempt is made to load a segment register with a selector valuse. The privilege level checking is done differently for data, stack, and code segments.

Privilege Levels - Restricting Access to Data

The currently executing code has access to any data segment that is of a lesser privilege level (numerically greater). It cannot access data of a greater privilege level. This prevents less privilege (or trusted) code to access data that belongs to more trusted code, while allowing more trusted/privileged code to access lower privileged data (if DPL >= CPL && DPL >= RPL then allow the access). Loading a data segment register with code segment selector is also valid as long as the code segment is marked readable. The exception is the stack, where each privilege has its own stack. When loading the stack segment register (SS), the selector value must be of the same privilege level (RPL=CPL=DPL).

Privilege Levels - Control of Execution Transfers (Code Segments)

Transferring control of execution to another segment is done via the CALL, JMP, RET, INT, or IRET instructions. When making a control transfer the target selector must point to a descriptor that is:

- another code segment,
- a gate (call, interrupt or trap),
- or a task state segment (a new task).

Privilege level checking is performed differently depending on the type of destination and the instruction used. In this section we'll address transferring control to another segment via the CALL, JMP and RET instructions and transferring control through a call gate using the CALL and JMP instructions. The remaining types will be mentioned in the sections where appropriate (see Multitasking & Interrupts sections).

Conforming versus Non-Conforming

Transferring control directly to another code segment using the CALL or JMP instructions has two rules since there are two type of code segments, conforming and non-conforming. For a non-conforming (or "normal") code segment the source and destination of the call must be have the same privilege level (CPL must equal DPL): otherwise a GP fault is issued.

Conforming segments are used when you have code that you want access to by applications of multiple privilege levels. When the control transfer is made to the conforming segment, the conforming segment takes on the CPL of the segment that referenced it. This means that control is not transferred to code of a higher privilege level when a call or jump to a conforming segment is made. The DPL of a conforming segment must be equal to or less than CPL (DPL <= CPL). This means that the DPL of a conforming segment is used to indicate the highest privilege code that is allowed to access it (the level of trust given to the conforming segment code).

Call Gates

Call gate is the method used for transferring control from lower privilege code to higher privileged code. A gate descriptor has four fields; a destination selector value, a 32-bit destination offset, a word parameter copy count, and a DPL (see Figure 5). The destination selector in the call gate descriptor is a selector for a code segment that will be the destination for the CALL or JMP. The 32-bit offset is the offset into the destination segment (what to load into EIP). The word parameter count is the amount of information to copy from the lower privileged to the higher privileged stack. The gate DPL, like a conforming segment DPL, is used to determine what privilege levels have access to the gate. For example, a gate with a DPL of 2 can be accessed by code running at privilege levels 0, 1, or 2, but if accessed from privilege level 3 a general protection fault will occur.

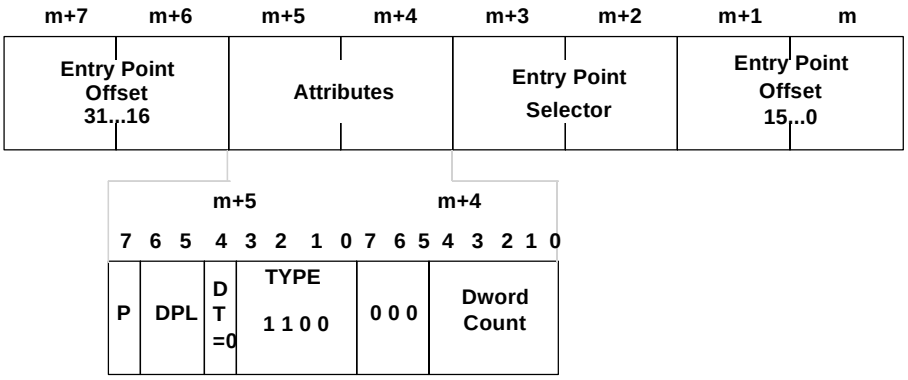


Figure 5. Call Gate Descriptor Format

When loading the CS with a selector for a call gate descriptor (a system descriptor type, not segment), the gate DPL must be equal to or greater than the maximum of CPL and RPL. The CPL must be equal to or greater than the destination DPL (Figure 6).

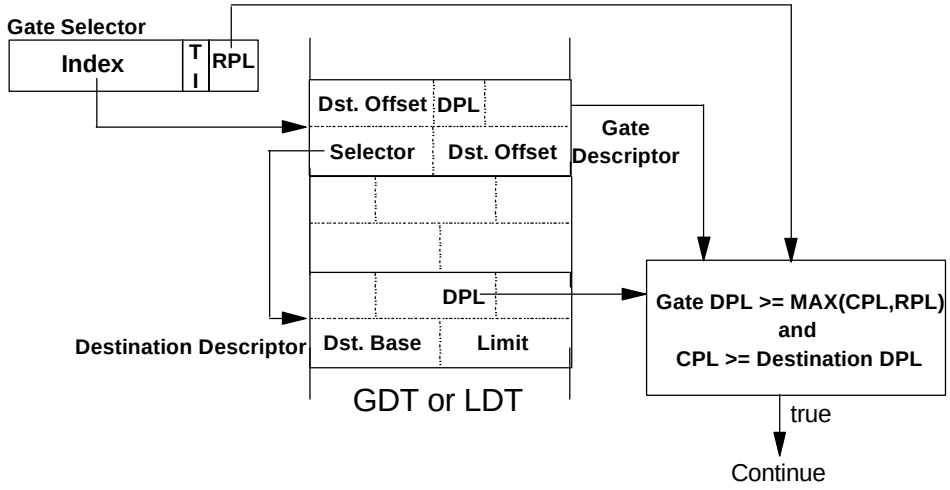


Figure 6. Call Gate Privilege Checking

To put it simply, if you want to call or jump to code at a higher privilege level you have to go through a gate.

RET Instruction
The RET instruction should return without a problem, but because of the possibility that the called function modified the return selector, the processor must perform privilege checking. The RET instruction is allowed to return control to code of a lesser privilege level. In doing so, the RET instruction expects the stack to look like a call, though a call gate was made to get to the higher privilege. The RET instruction looks at the saved CS value on the stack. If its RPL is less privileged than the CPL, then a return across privilege levels occurs. Otherwise the RPL should match the CPL, and a normal return is performed. If neither of these are true, than a general protection fault will occur.

Protected Instructions

The protection mechanism prevents the execution of system sensitive instructions from privilege levels other than those trusted by the operating system. Some instructions are limited to execution from privilege level 0 tasks only, while others are sensitive to the I/O privilege level (IOPL). For a list of

privilege level 0 only and I/O privilege level sensitive instructions, reference the *i486 Microprocessors Programmer's Reference Manual,* Chapter 6.

I/O Privilege Level Sensitive Instructions

A two bit field in the EFLAGS register defines the I/O privilege level (IOPL). I/O instructions are instructions that typically deal with interfacing to external hardware. These instructions are: IN, INS, OUT, OUTS, CLI and STI (port I/O, disable and enable interrupts). When the processor is running in Virtual-86 mode, the following instructions are added to the list: PUSHF, POPF, INT n, IRET (these are added to allow for better emulation of an 8086 environment under protected mode). The IOPL defines what privilege level is required to access I/O sensitive instructions. The IOPL can only be modified by code running at privilege level 0. If code at a lesser privilege level attempts to change the IOPL, no exception is signaled and the IOPL remains unchanged.

Since each task uses its own copy of the EFLAGS register, each task can be given different levels of access to the I/O sensitive instructions. In addition, each task has an I/O permission bitmap associated with it. This I/O permission bitmap can be used to grant access to port I/O instructions on a port by port bases.

When an IOPL sensitive instruction is executed, the following checks are performed before the instruction is executed:

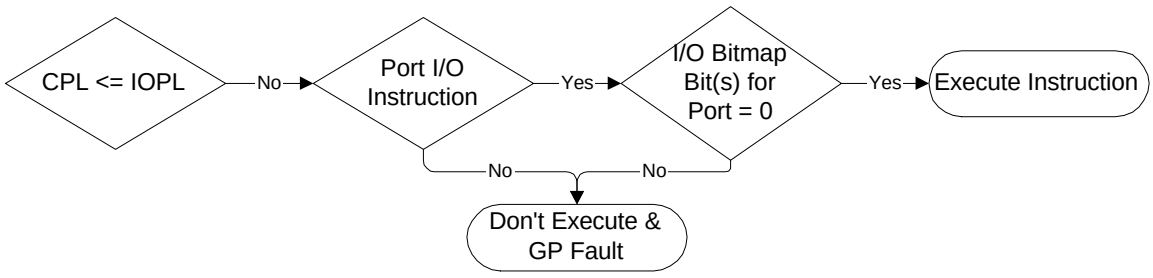


Figure 7. Execution of IOPL Sensitive Instructions

Having IOPL sensitive instructions allows the system designer to create an operating system that has complete control over access to the external hardware. If all applications running under a given OS run at a privilege level of 3 and the IOPL is set to 0, the application is forced to use the OS facilities to access the hardware. The I/O permission bitmap can be used to grant access to individual hardware devices on a task-by-task bases. For memory-mapped hardware devices the same applies. The OS can allocate descriptors for the hardware device and make the selector available as needed.

Paging Unit

The addition of the page unit to the Intel architecture enhanced its ability to support virtual memory systems and to support multiple Virtual-86 mode tasks. The page unit is yet another feature of the Intel architecture that does not have to be used or implemented by the system designer in order to use the architecture.

How the Page Unit Works

The page unit is turned on by setting the PG bit on the CR0 register. If the PG bit is set, the linear address generated by the segmentation unit is passed to the page unit for translation. This translation

will yield the final physical address. If the PG bit is cleared, the result of the segmentation unit is the final physical address (See Figure 8)

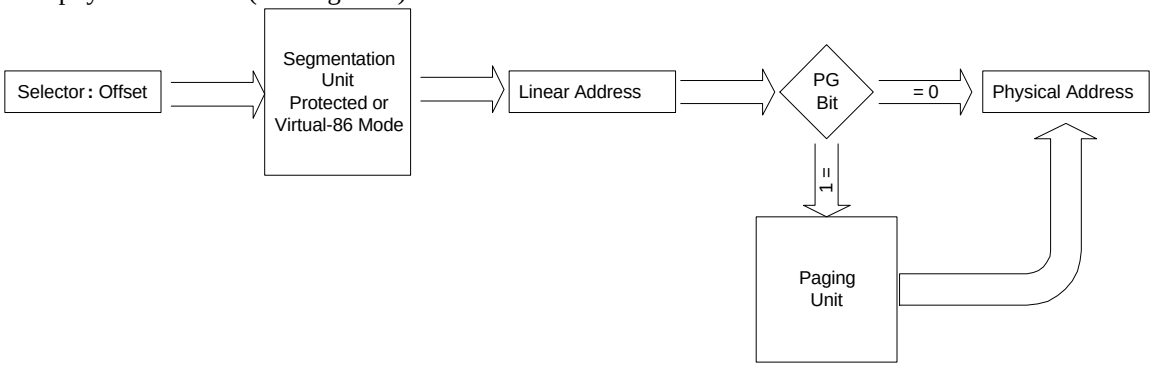


Figure 8. Address Translation

The page unit breaks memory into 4KB fixed-size chunks called pages. To figure out which page to access for a given memory reference, the page unit breaks down the 32-bit linear address into three pieces, a 10-bit directory table index, a 10-bit page table index and a 12-bit offset within a page. The directory and page table indexes work much the same way a selector works with the descriptor tables. First the directory table index is used to find the appropriate directory table entry within the current directory table, pointed to by the CR3 register. The directory table entry has the address of the appropriate page table in its most significant 20-bits (all tables and pages must be on a 4KB boundary). The page index is then used to find the appropriate page table entry within the specified page table. The page table entry has the most significant 20-bits of the physical address, and this plus the 12-bit offset results in the final physical address (see Figure 9).

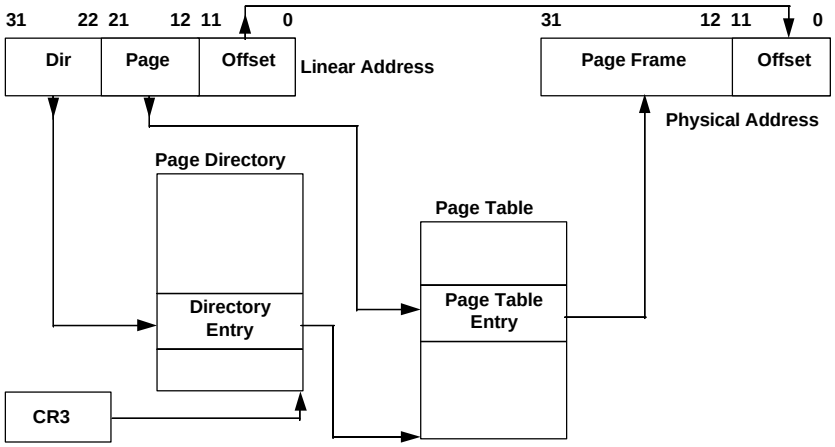


Figure 9. Page Unit Linear to Physical Address Translation

.

You can see by the information in the directory table and page table entries that the page unit is good for more than just translating addresses (see Figure 10). The additional information allows the system designer to easily implement a virtual memory design as well as provide another way of controlling access to memory regions.

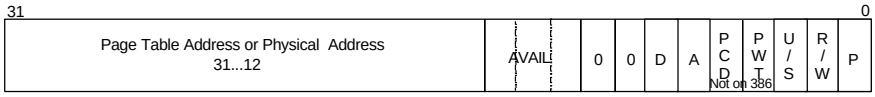


Figure 10. Page Table & Directory Table Entry Format

- AVAIL Available for OS specific usage.
- D Dirty Bit, automatically set whenever the page is written to. Used for virtual memory systems.
- A Access bit, automatically set whenever the page is read or written to. Used for virtual memory systems.
- PCD Page Cache Disable, indicates whether or not a page is cached in the internal cache
- PWT Page Write-Through bit controls the write policy for the second-level caches used with the write-back enchanced processors.
- U/S User/Supervisory access bit, page table level protection. If the bit is set, the page is accessible from all privilege levels. If cleared, the page is accessible from all levels except three.
- R/W page Read/Write protection bit. If the bit is 0, the page is a read-only page otherwise the is both read and writable.
- P Present bit indicates if the page is present in memory or not. Used for virtual memory systems.

Using the Page Unit for Virtual Memory Systems

In each page and directory table entry there are bits provided specifically for the support of virtual memory systems (D, A, P bits). The Present bit is set and cleared by the operating system to indicate that a given page has been swapped out of memory. If an access to that page then occurs by an application or the OS itself, a page fault is signaled. If all of the physical memory is being utilized, the page fault handler can use the information in the Dirty and Accessed bits to help determine which page in memory to swap out next. With the page unit, virtually anything can be swapped, but there are a number of issues when you swap system objects. The best advice is to avoid swapping system objects.

Paging With Virtual-86 Mode

Another key application for the page unit is in the ability for an operating system to support multiple Virtual-86 tasks (see Virtual-86 Mode section, for more information on Virtual-86 mode). Each Virtual-86 mode task can only produce a 20-bit address, which would restrict its access to only the first 1MB of memory, if paging were not used. But with the use of the paging unit, the 20-bit linear address put out by the segmentation unit can be translated to any address within the 4GB address space. This is done by assigning to each Virtual-86 task its own page directory and page table. Since the page directory table is pointed to by the CR3 register, a task switch between two or more Virtual-86 tasks need only change the CR3 register to point to the page directory table for that task (see Figure 11).

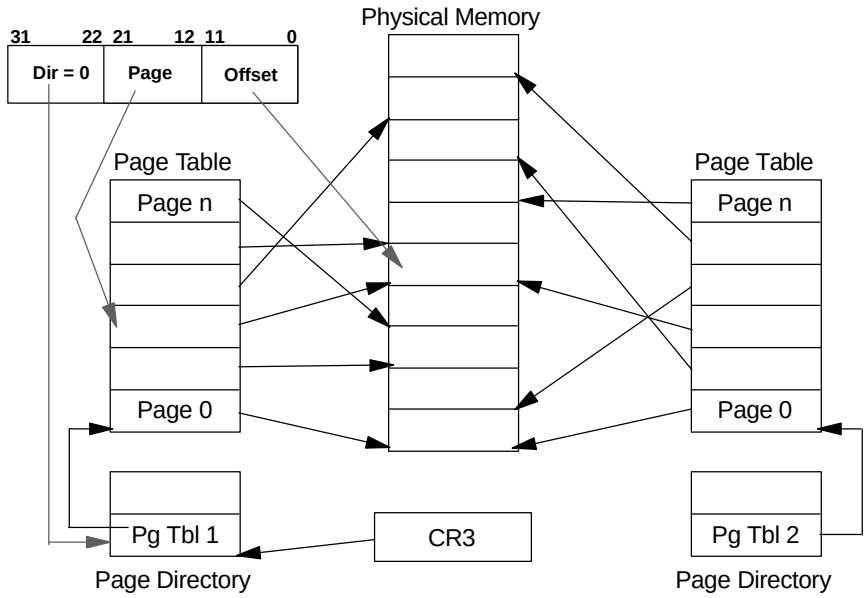


Figure 11. Paging with Virtual-86 Mode

Multitasking

The multitasking support is a capability provided by the Intel architecture that the system designer can choose not to use. Many of the available multitasking operating systems that run on the Intel architecture do not use the hardware multitasking support provided. One of the key reasons is that the functionality of the hardware task switch can be completely supported using software task switching. Since many of the multitasking operating systems also require additional processing when switching tasks, performing the additional work is of little concern. In fact, the task switching may actually be faster because of the amount of processor information that must be preserved between task switches. For example, if an operating system elects not to use the local descriptor table, page unit or run with multiple privilege levels, the amount of information that must be preserved when switching tasks is less than what would be saved by using the hardware task-switching.

The operation of the hardware multitasking depends on these additional system data structures:

- Task State Segment (TSS); segment for saving and restoring the processor state.
- Task state segment's descriptor; it points to a TSS and must be in the GDT.
- Task register (TR); holds the selector for the current task's TSS.
- Task gate descriptor.

The task state segment (TSS), like all segments, has a descriptor associated with it, called a task state segment descriptor. The processor distinguishes a TSS descriptor from other segments by looking at the type field (a value of 010x1, where x represents the busy state of the task). The TSS descriptors looks the same as a code type descriptor except for the D-bit and the contents of the type field.

A TSS is used to save and to load the processor state whenever a task switch is initiated. The TSS will hold the complete processor state plus the I/O bitmap. A TSS can be used to hold information other than the processor state.

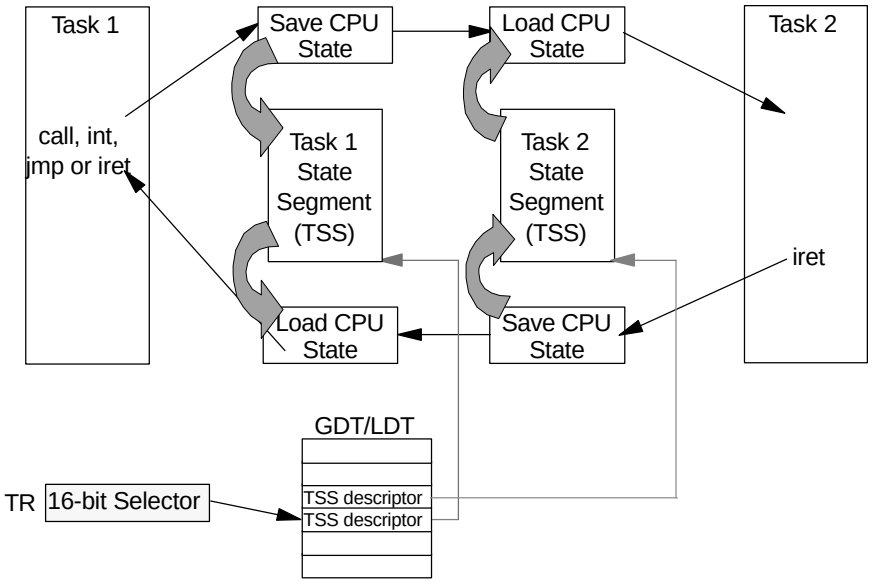


Figure 12. Task Switch Steps

A hardware task switch occurs in the following cases:

- The current task executes a JMP or CALL instruction that refers to either a TSS or task gate descriptor;
- An interrupt (software or hardware) or exception vectors to task gate in the IDT;
- The current task executes an IRET instruction when the Nested Task (NT) flag is set.

The task gate descriptor (descriptor type field of: 00101) looks just like a interrupt or trap gate descriptor, with the exception of the type field, and also works the same way.

Interrupts

The interrupt process works much the same as in the 8086, except that instead of having a table of new CS:IP values, the table of task, interrupt or trap gates is known as the interrupt descriptor table (IDT). The IDT is pointed to by the IDTR register, which holds the IDT's base address and limit. Since interrupts may occur at random, gates are used to ensure that a control transfer can be made regardless of the CPL. Interrupt and trap gates work the same as call gates (mentioned earlier) without the transfer of parameters from one privilege level stack to another. Interrupt and trap gates are identical in operation except for the setting of the interrupt enable flag (IF) bit in the EFLAGS register. Interrupt gates clear the IF flag (disabling interrupts), while trap gates do not modify the IF flag. Since EFLAGS is pushed onto the stack as part of the interrupt process, the state of the IF flag will be restored when an IRET is executed. Interrupts that reference a task gate will cause a task switch to occur (see the Multitasking section).

Sources for interrupts are external events, exceptions, and instructions (INT0, INT3, INT n, and BOUND). The information placed on the stack for either external event or instruction interrupt sources is the same, while additional information (an error code) may be placed on the stack for processor exceptions.

Let's look at the process, the processor executes an INT 2 instruction (see Figure 13, Interrupt Vector Process):

- Calculate address of where to fetch the interrupt descriptor from. Take Vector Number * Size of Descriptor (2*8) + IDTR Base Address.

- Read the descriptor to determine what segment and offset within the segment to vector to. Our descriptor shows an 18h for the selector, which represents the GDT table entry number 3 and 00003B2Ch for the offset. The 9Fh represents the attributes, the F tells us we have a trap gate.
- Read the GDT[3] descriptor to determine the destination base address and add the offset from the IDT[2] trap gate to get the linear address in memory to vector to (0FF1000h + 3B2Ch = FF4B2Ch).

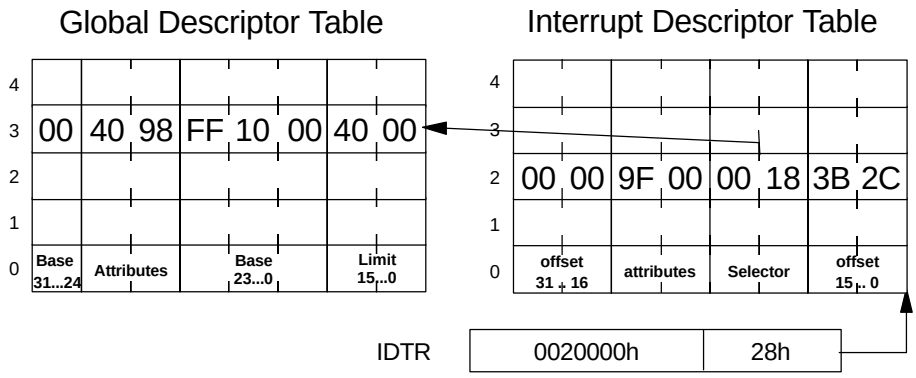


Figure 13, Interrupt Vector Process

Virtual-86 Mode

Virtual-86 (V86) mode provides support for executing one or more 8086/80186 programs from within the protected environment. Anyone who has run one or more DOS shells from within Windows has seen V86 mode in action. While running in V86 mode, the segmentation unit calculates addresses the same way an 8086/80186 processor does. That is, a segment register is loaded with a 16-bit base address that is multiplied by 16. The offset is then added to the 20-bit base to result in a 20-bit linear address. The difference is that the application is running under the protected environment and is assigned a privilege level of 3. Therefore it must run within the constraints of that privilege level (IOPL and privilege instructions), unlike running the processor in real mode. When a processor exception or interrupt occurs, the processor switches back to a protected mode handler to deal with the exception or interrupt. The switch back to protected mode, when an exception or interrupt occurs, allows the protected mode operating system to:

- Be protected from unwanted access to processor registers, hardware, and global structures.
- Easier coordination of shared resources among several V86 and other tasks.
- Functions of an 8086/80186 operating system to be easily emulated by the protected mode OS.

Using the page unit with V86 mode is not required if only one V86 task is running, but in order to run more that one V86 task the page unit must be used. The page unit is used to translate the 20-bit linear address for each task to a new physical address. To do this, create a directory table and a page table for each V86 task. When switching between V86 tasks simply change the contents of the CR3 register to point to the directory table for that task (see Paging Unit section).

Summary

As you can see, there are a number of enabling capabilities existing in the 80386 processor's protected mode environment that allow the system developer design a very robust and secure operating system. Yet the architecture is flexible enough to allow the system designer to create the type of system required. This paper is just an overview of these capabilities and does not go into the detail needed to design a system. Since the 80386 architecture is one of the most popular, many reference books exist.

The reference books that I use are the basic *I486 Microprocessors Programmer's Reference Manual* and *80386 Systems Software Writer's Guide* by Intel.