OPERATING SYSTEMS

Intel's View of Memory Management

Jerry Breecher

Intel Memory Management

This set of slides is designed to explain the Memory Management Architecture used by Intel Pentium processors.

For these slides we will use the Intel document found at:

http://www.intel.com/design/processor/manuals/253668.pdf

Intel explains this document as a description of the hardware interface required by an Operating System in order to implement a Memory Management.

It's assumed that you are familiar with the normal picture of memory management as presented in Chapters 8 & 9 in this course.

How Do Operating Systems Use Memory Management

So I wrote a little program to probe the memory seen by a program. I ran that same program on Windows 2000, Windows XP and RedHat LINUX. I was looking at the addresses that were being used for various kinds of data/code in the program. I probed the addresses by asking for memory continually until something broke. For instance, did continual allocs until an error was returned. Here's a pseudo code of the program:

```
#define   ONE_MEG              1048576
#define  MEM_SIZE             3 * ONE_MEG
char     GlobalMemory[MEM_SIZE];                   // This is a global/static variable

int    main( int argc, char  *argv[] )
{
    int    FirstStackLocation;
    int    Mode, Temp, *TempPtr;
    int    Counter = 0;
    void   *MemPtr, *LastPtr;

    printf("Address of main(): %8X\n", (int)(&main) );
    while ( TRUE )                                        // Find highest memory until seg. fault
    {
        TempPtr = (int *)((int)main + (CODE_JUMP * Counter) );    // Address of location
        Temp = *TempPtr;
        printf( "Got address %X\n", (int)((int)main + (CODE_JUMP * Counter) ) );
        Counter++;
    }
```

Keeps touching memory until it takes a fault

How Do Operating Systems Use Memory Management

```
    printf("Address Start of Global:  %8X\n", (int)(&GlobalMemory) );
    printf("Address  End  of Global:  %8X\n", (int)(&GlobalMemory) + MEM_SIZE -1);
    MemPtr = malloc( ONE_MEG );
    printf("First location on heap:  %8X\n", (int)MemPtr );
    while( (MemPtr = malloc( ONE_MEG )) != NULL )
    {
        LastPtr = MemPtr;
        Counter++;
        if ( Counter %100 == 0 )
            printf("%5d alloc  on heap:%8X\n", Counter, (int)LastPtr +ONE_MEG - 1);
    }
    printf("Total bytes allocated:   %8X (Hex)\n", Counter * ONE_MEG );
    printf("Last  location on heap:  %8X\n", (int)LastPtr );
}
```

Iterates on allocs

```
#define    STACK_ALLOC    ONE_MEG
void  RecursiveRoutine( )
{
    char    Temp[ STACK_ALLOC ];

    printf("Begin/End of this allocation: %8X %8X\n",
             (int)&(Temp), (int)(&(Temp[STACK_ALLOC])) );
    RecursiveRoutine();
}
```

Iterates using lots of stack

How Do Operating Systems Use Memory Management

So I wrote a little program to probe the memory seen by a program. I ran that same program on Windows 2000, Windows XP and RedHat LINUX. I was looking at the addresses that were being used for various kinds of data/code in the program. I probed the addresses by asking for memory continually until something broke. For instance, did continual allocs until error was returned

Windows XP Memory Usage

Segment	First Address	Last Address	Size
Code	401000x	403000x	002000x ~ 8 Kbytes
Static (Global) Data	403000x	703000x	300000x ~ 3 megabytes
Heap	760000x	3A261000x	39800000x ~ 950 megabytes
Stack	22EF00x	16EF00x	1C0000x ~ 2 megabyte

The file MemoryDemo.exe is about 170Kbytes in size.

Declared a 3 Meg static array!.

Note these addresses grow down!

Note: 100000x == 1 Megabyte

How Do Operating Systems Use Memory Management

So I wrote a little program to probe the memory seen by a program. I ran that same program on Windows 2000, Windows XP and RedHat LINUX. I was looking at the addresses that were being used for various kinds of data/code in the program. I probed the addresses by asking for memory continually until something broke. For instance, did continual allocs until error was returned

LINUX Memory Usage

Segment	**First Address**	**Last Address**	**Size**
Code	8048400x	8049900x	001500x ~ 6 Kbytes
Static (Global) Data	8049A00x	8349A00	300000x ~ 3 megabytes
Heap	B7EE,B000x	01CE,4000x	B6000000x ~ 3 gigabytes
Stack	BFFB,7334x	29BA,91E0x	9640,0000x ~ 2.5 gigabyte

Declared a 3 Meg static array!.

Note these addresses grow down!

How can this sum to more than 4 gigs??

Note: 100000x == 1 Megabyte

How Do Operating Systems Use Memory Management

```
0x08048368 <main+0>:    55                     push   %ebp
0x08048369 <main+1>:    89 e5                  mov    %esp,%ebp
0x0804836b <main+3>:    83 ec 08               sub    $0x8,%esp
0x0804836e <main+6>:    83 e4 f0               and    $0xfffffff0,%esp
0x08048371 <main+9>:    b8 00 00 00 00         mov    $0x0,%eax
0x08048376 <main+14>:   83 c0 0f               add    $0xf,%eax
0x08048379 <main+17>:   83 c0 0f               add    $0xf,%eax
0x0804837c <main+20>:   c1 e8 04               shr    $0x4,%eax
0x0804837f <main+23>:   c1 e0 04               shl    $0x4,%eax
0x08048382 <main+26>:   29 c4                  sub    %eax,%esp
0x08048384 <main+28>:   83 ec 0c               sub    $0xc,%esp
0x08048387 <main+31>:   68 c0 84 04 08         push   $0x80484c0
0x0804838c <main+36>:   e8 1f ff ff ff         call   0x80482b0
0x08048391 <main+41>:   83 c4 10               add    $0x10,%esp
0x08048394 <main+44>:   e8 02 00 00 00         call   0x804839b <b>
0x08048399 <main+49>:   c9                     leave
0x0804839a <main+50>:   c3                     ret
```

```
void  b();
void  c();
int    main( )
{
    printf( "Hello from main\n");
    b();
}
// This routine reads the opcodes from memory and prints them out.
void   b()
 {
     char  *moving;

     for ( moving = (char *)(&main); moving < (char *)(&c); moving++ )
         printf( "Addr = 0x%x, Value = %2x\n", (int)(moving), 255 & (int)*moving );
 }
 void   c()
 {
 }
```

Memory Layout

```
0x0804839b <b+0>:     55                    push   %ebp
0x0804839c <b+1>:     89 e5                 mov    %esp,%ebp
0x0804839e <b+3>:     83 ec 08              sub    $0x8,%esp
0x080483a1 <b+6>:     c7 45 fc 68 83 04 08 movl   $0x8048368,0xfffffffc(%ebp)
0x080483a8 <b+13>:    81 7d fc d9 83 04 08 cmpl   $0x80483d9,0xfffffffc(%ebp)
0x080483af <b+20>:    73 26                 jae    0x80483d7 <b+60>
0x080483b1 <b+22>:    83 ec 04              sub    $0x4,%esp
0x080483b4 <b+25>:    8b 45 fc              mov    0xfffffffc(%ebp),%eax
0x080483b7 <b+28>:    0f be 00              movsbl (%eax),%eax
0x080483ba <b+31>:    25 ff 00 00 00        and    $0xff,%eax
0x080483bf <b+36>:    50                    push   %eax
0x080483c0 <b+37>:    ff 75 fc              pushl  0xfffffffc(%ebp)
0x080483c3 <b+40>:    68 d1 84 04 08        push   $0x80484d1
0x080483c8 <b+45>:    e8 e3 fe ff ff        call   0x80482b0
0x080483cd <b+50>:    83 c4 10              add    $0x10,%esp
0x080483d0 <b+53>:    8d 45 fc              lea    0xfffffffc(%ebp),%eax
0x080483d3 <b+56>:    ff 00                 incl   (%eax)
0x080483d5 <b+58>:    eb d1                 jmp    0x80483a8 <b+13>
0x080483d7 <b+60>:    c9                    leave
0x080483d8 <b+61>:    c3                    ret
```

```c
void  b();
void  c();
int    main( )
{
    printf( "Hello from main\n");
    b();
}
// This routine reads the opcodes from memory and prints them out.
void   b()
 {
     char  *moving;

     for ( moving = (char *)(&main); moving < (char *)(&c); moving++ )
         printf( "Addr = 0x%x, Value = %2x\n", (int)(moving), 255 & (int)*moving );
 }
 void   c()
 {
 }
```

Intel Memory Management

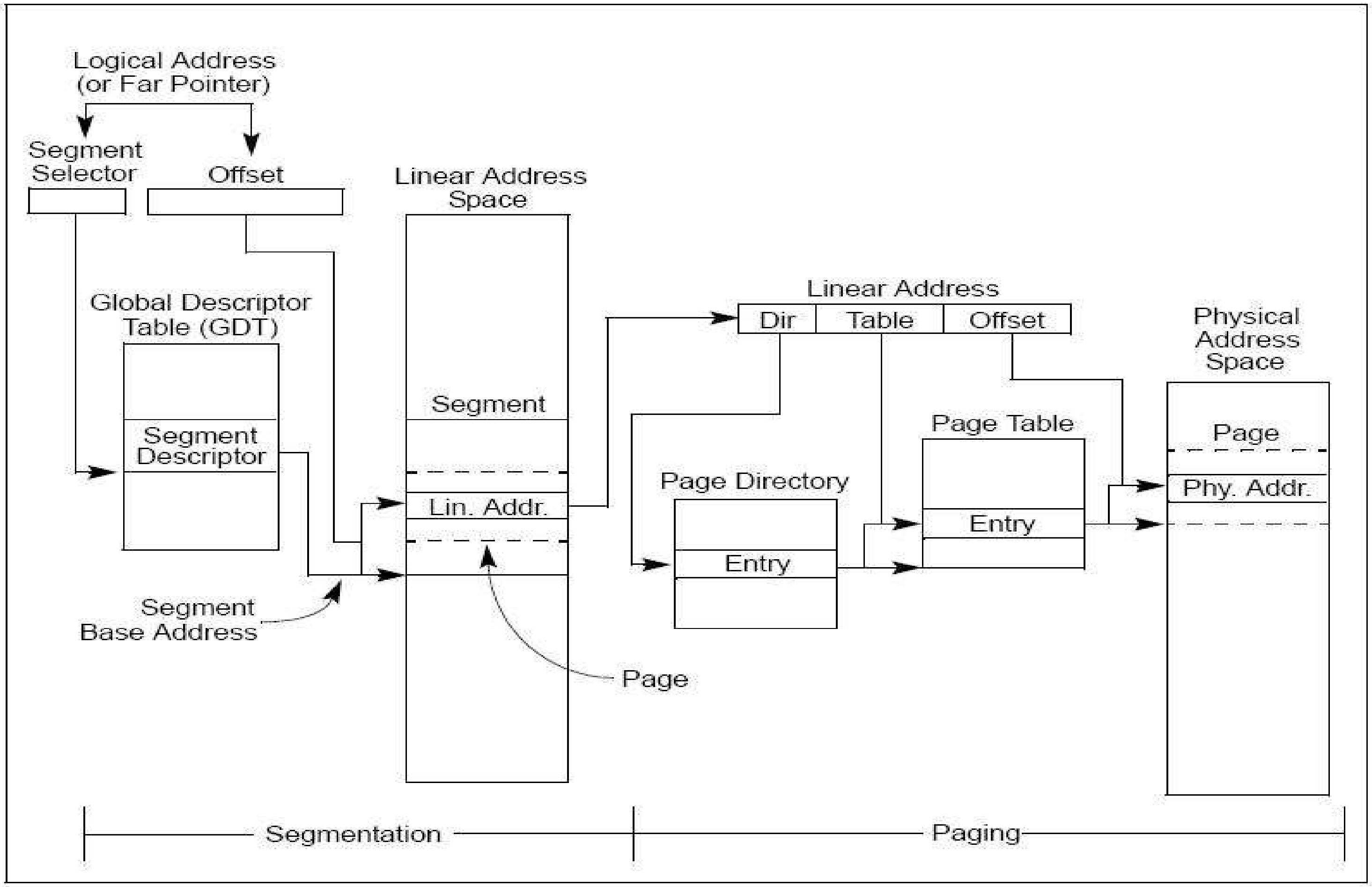


Figure 3-1. Segmentation and Paging

This is an overview of the hardware pieces provided by Intel. It's what we have to work with if we're designing an O.S.

Intel Memory Management

The memory management facilities of the **IA-32** architecture are divided into two parts:

Segmentation

Segmentation provides a mechanism of isolating individual code, data, and stack modules so that multiple programs (or tasks) can run on the same processor without interfering with one another.

When operating in protected mode, some form of segmentation must be used.

Paging.

Paging provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program’s execution environment are mapped into physical memory as needed. Paging can also be used to provide isolation between multiple tasks.

These two mechanisms (segmentation and paging) can be configured to support simple single program (or single-task) systems, multitasking systems, or multiple-processor systems that used shared memory.

Intel Memory Management

See Figure 3-1.

Segmentation gives a mechanism for dividing the processor's addressable memory space (called the **linear address space**) into smaller protected address spaces called **segments**.

Segments are used to hold code, data, and stack for a program andr to hold system data structures (such as a TSS or LDT).

Each program running on a processor, is assigned its own set of segments.

The processor enforces the boundaries between segments and insures that one program doesn't interfere with the execution of another .

The segmentation mechanism allows typing of segments to restrict operations that can be performed.

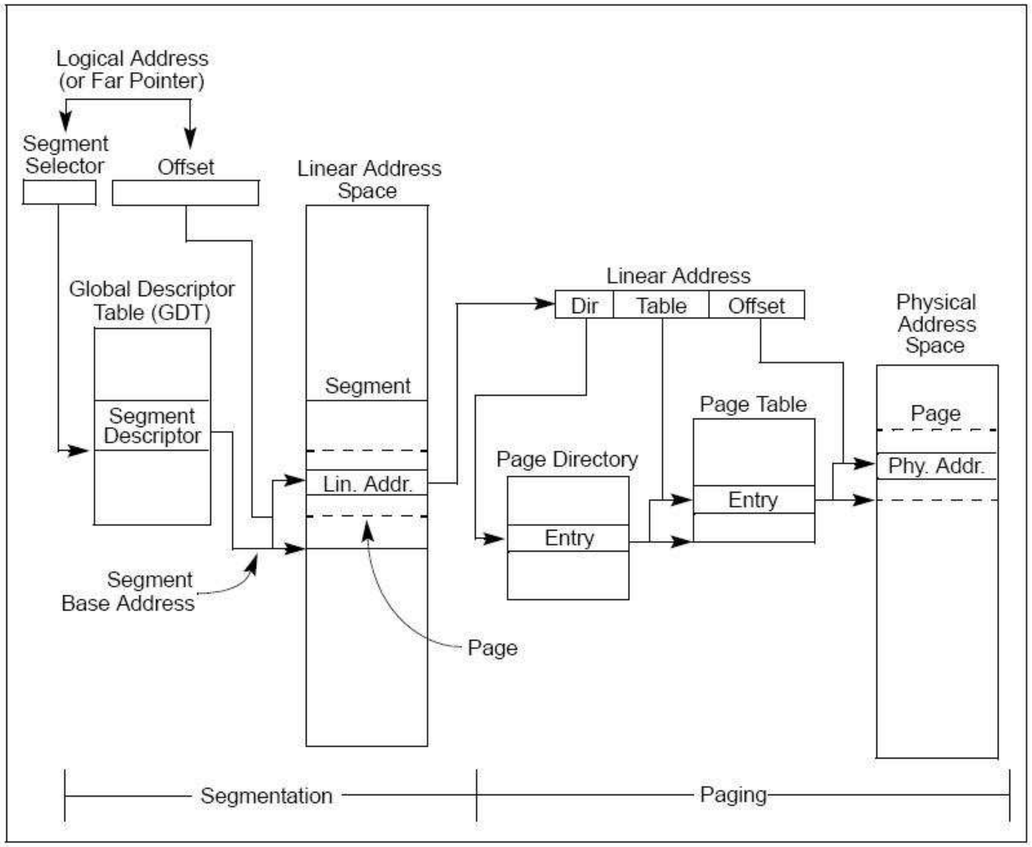


Figure 3-1. Segmentation and Paging

Intel Memory Management

See Figure 3-1.

All the segments in a system are contained in the processor's **linear address space**.

To locate a byte in a particular segment, **a logical address** (also called a far pointer) must be provided.

A logical address has :

1. **The segment selector – a** unique identifier for a segment - provides an offset into a descriptor table (such as the global descriptor table, GDT) to a data structure called a segment descriptor.

   This segment descriptor specifies the size of the segment, the access rights and privilege level for the segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment).

   See 3.4.2 Segment Selectors" for more details.

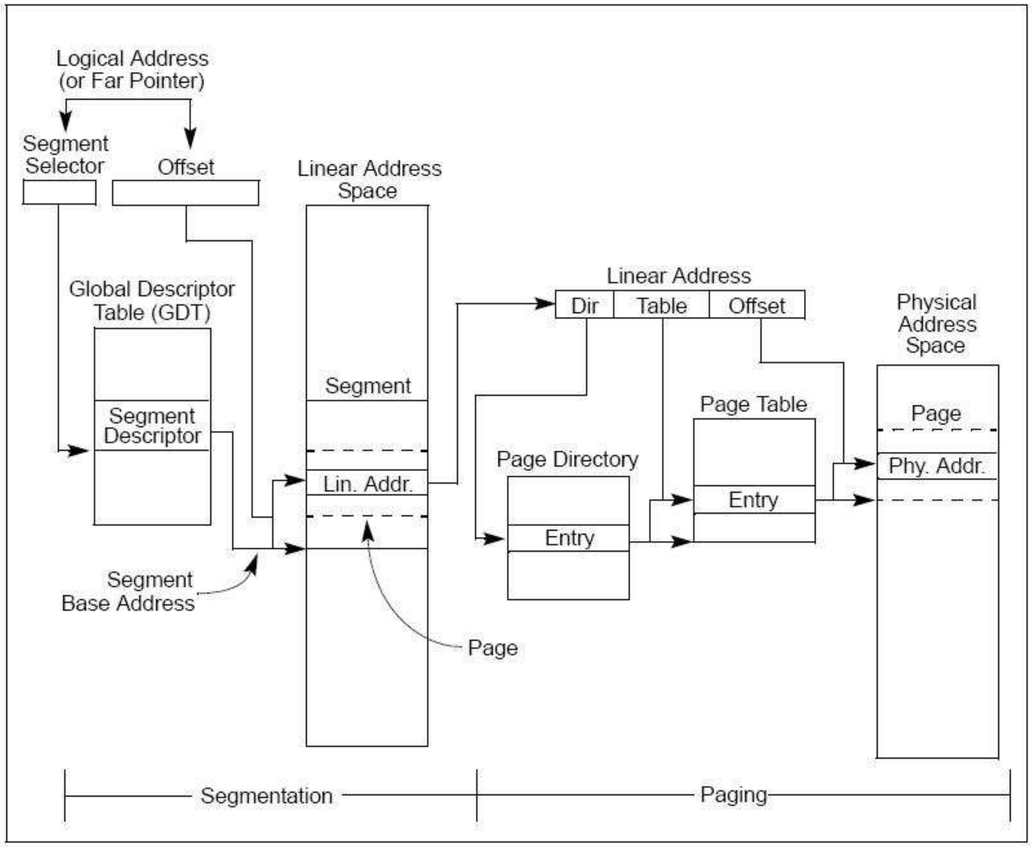


Figure 3-1. Segmentation and Paging

2. **The offset** part of the logical address -added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a **linear address** in the processor's linear address space.

Intel Memory Management

3.2.1 Basic Flat Model

The simplest memory model for a system is the basic “flat model,”

the operating system and application programs have access to a continuous, unsegmented address space.

.

To implement a basic flat memory model with the IA-32 architecture, at least two segment descriptors must be created:

- one for referencing a code segment and
- one for referencing a data segment (see Figure 3-2).
- both segments, however, are mapped to the entire linear address space: that is, both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes.

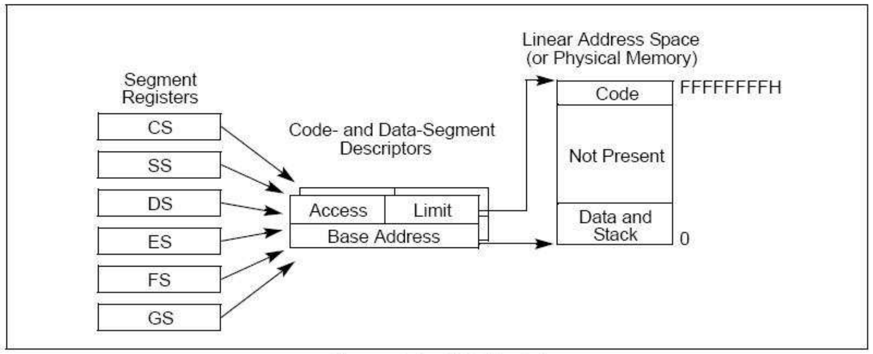


Figure 3-2. Flat Model

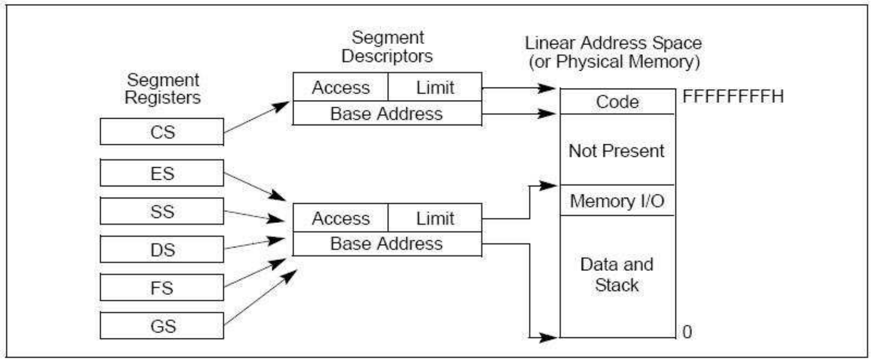


Figure 3-3. Protected Flat Model

Intel Memory Management

3.2.2 Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists (see Figure 3-3).

A protection exception is generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

More complexity can be added to this protected flat model to provide more protection.

Example: For the paging mechanism to provide isolation between user and supervisor code and data, four segments need to be defined:

- code and data segments at privilege level 3 for the user,
- and code and data segments at privilege level 0 for the supervisor.

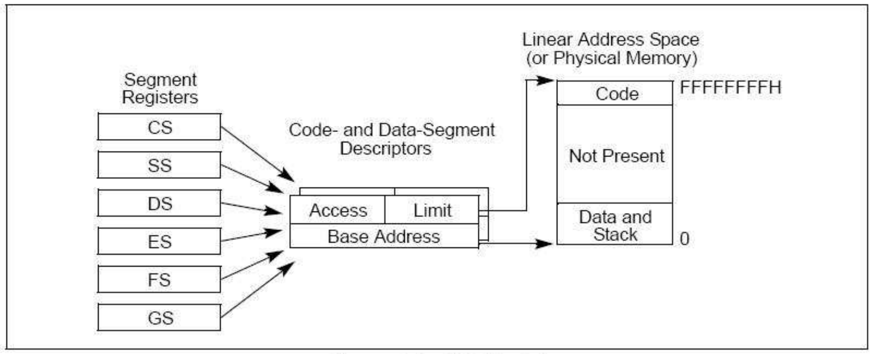


Figure 3-2. Flat Model

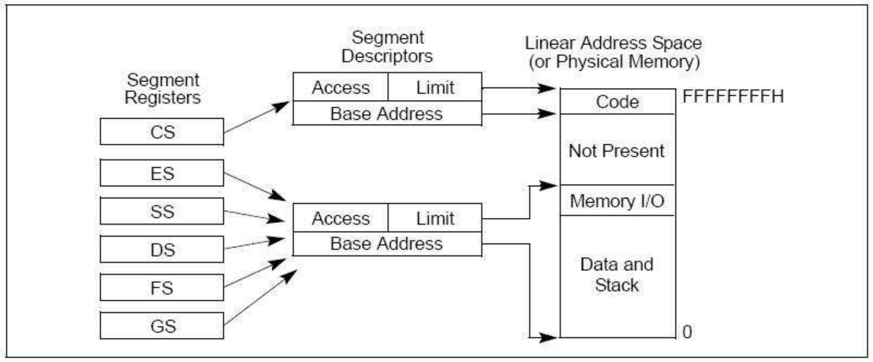


Figure 3-3. Protected Flat Model

Intel Memory Management

3.2.3 Multi-Segment Model

A multi-segment model (shown here) uses the full capabilities of segmentation to provide hardware enforced protection of code, data structures, and programs and tasks.

- each program (or task) has its own table of segment descriptors and its own segments.
- segments can be completely private to their programs or shared among programs.
- Access to segments and to program environments is controlled by hardware.

Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments.

- The access rights information created for segments can also be used to set up protection rings or levels.
- Protection levels can be used to protect operating system procedures from unauthorized access by application programs.

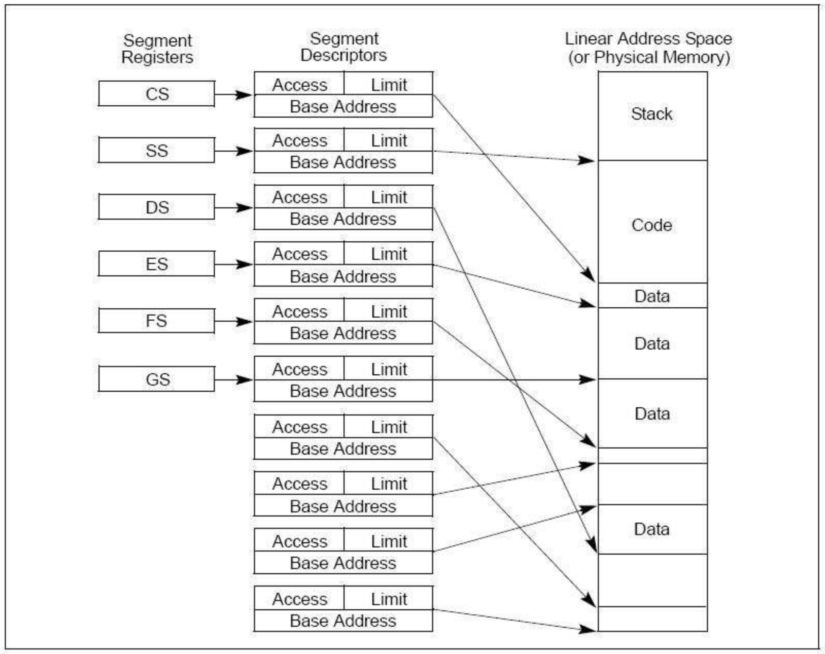


Figure 3-4. Multi-Segment Model

Intel Memory Management

3.3 PHYSICAL ADDRESS SPACE

In protected mode, the IA-32 architecture provides a normal physical address space of 4 Gbytes (2^{32} bytes).

This is the address space that the processor can address on its address bus. This address space is flat (unsegmented), with addresses ranging continuously from 0 to FFFF,FFFFH. This physical address space can be mapped to read-write memory, read-only memory, and memory mapped I/O. The memory mapping facilities described in this chapter can be used to divide this physical memory up into segments and/or pages.

The IA-32 architecture also supports an extension of the physical address space to 2^{36} bytes (64 GBytes); with a maximum physical address of F,FFFF,FFFFH. This extension is invoked

• Using the physical address extension (PAE) flag, located in bit 5 of control register CR4.

-- Talked about later.

Intel Memory Management

3.4 LOGICAL AND LINEAR ADDRESSES

The processor uses two stages of address translation to arrive at a physical address: logical-address (via segments) translation and linear address space (via paging) translation.

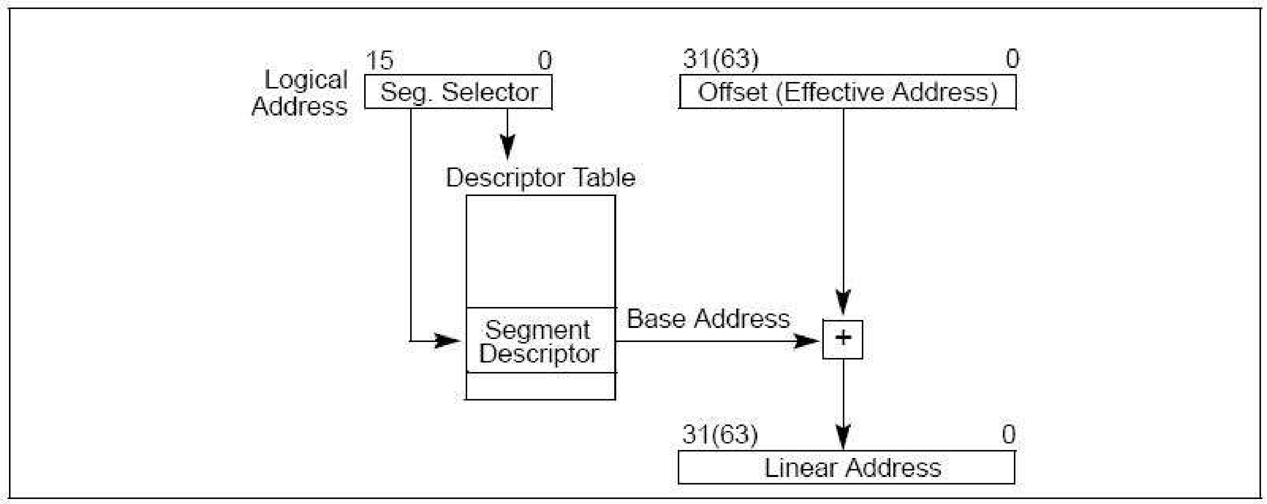


Figure 3-5. Logical Address to Linear Address Translation

Intel Memory Management

3.4 LOGICAL AND LINEAR ADDRESSES

Every byte in the processor's address space is accessed with a logical address. A **logical** address consists of a 16-bit segment selector and a 32-bit offset (see Figure 3-5).

.A **linear** address is a 32-bit address in the processor's linear address space. The linear address space is a flat (unsegmented), 2^{32}-byte address space, with addresses ranging from 0 to FFFF,FFFFH.

The linear address space contains all the segments and system tables defined for a system.

To translate a logical address into a linear address, the processor does the following:

1. Uses the offset in the segment selector to find the descriptor for the segment in the GDT or LDT and reads it into the processor, or uses the appropriate segment register.
2. Examines the segment descriptor to check the access rights and range of the segment – makes sure the segment is accessible and has legal offset.
3. Adds the base address of the segment to the offset to form a linear address.

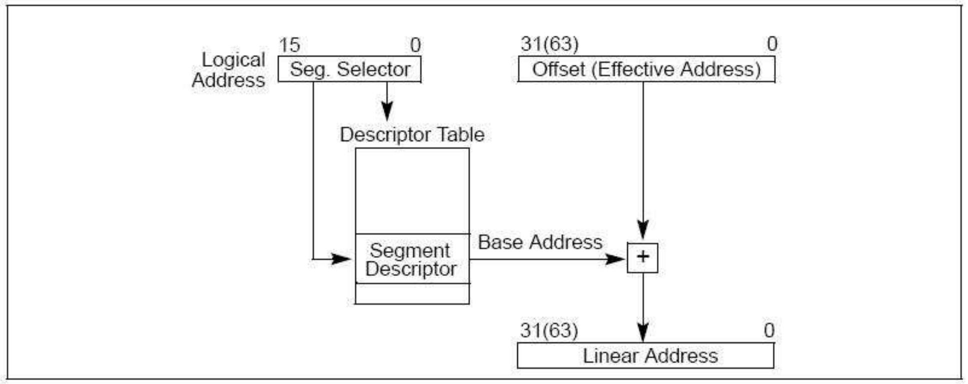


Figure 3-5. Logical Address to Linear Address Translation

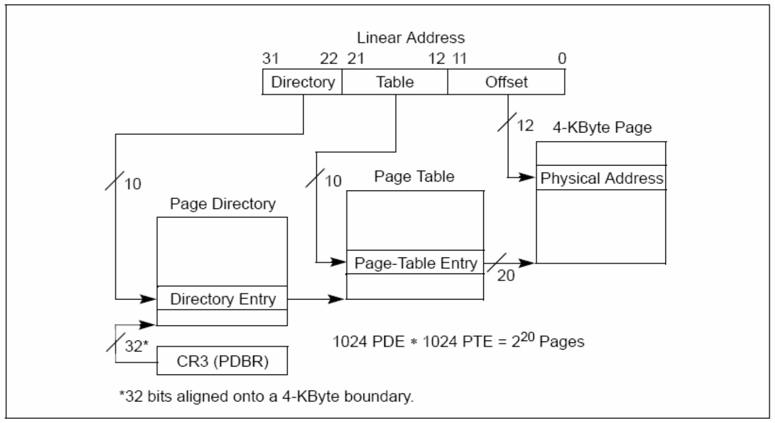


Figure 3-12. Linear Address Translation (4-KByte Pages)

Intel Memory Management

3.4.2 Segment Selectors

A segment selector is a 16-bit identifier for a segment (see Figure 3-6). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment. A segment selector contains the following items:

Index — Selects one of 8192 descriptors in the GDT or LDT.

TI (table indicator) flag — Specifies the descriptor table to use: GDT or LDT

Requested Privilege Level (RPL) — Specifies the privilege level of the selector. The privilege level can range from 0 to 3, with 0 being the most privileged level.

Intel Memory Management

3.4.3 Segment Registers

To reduce address translation time and coding complexity, the processor provides registers for holding up to 6 segment selectors (see Figure 3-7).

Each of these segment registers support a specific kind of memory reference (code, stack, or data).

At least the code-segment, data-segment, and stack-segment registers must be loaded for a program to run..

The processor provides three additional data-segment registers (ES, FS, and GS), which can be used to make other data segments available to the currently executing program (or task).

To access a segment, a program must get to it via a segment register.

Although a system can define thousands of segments, only 6 can be available for immediate use.

There are instructions available so the OS can set up segment registers.

Note how the address translation actually goes through the segment register rather than through the Descriptor Table.

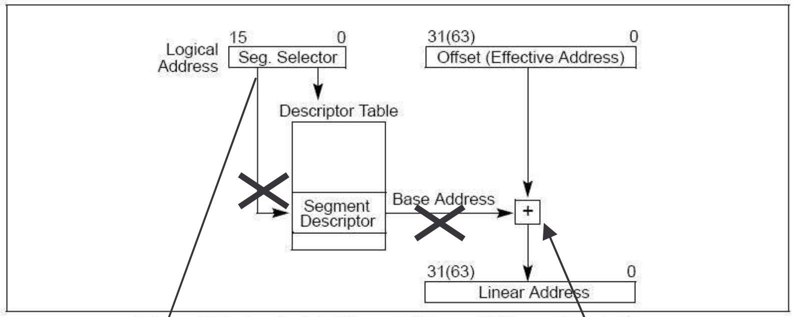


Figure 3-5. Logical Address to Linear Address Translation

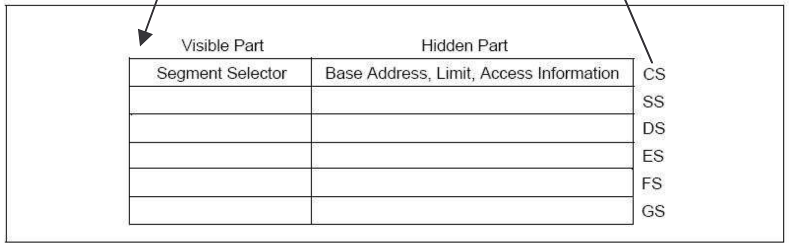


Figure 3-7. Segment Registers

Intel Memory Management

Every segment register has a “visible” part and a “hidden” part.

When a segment selector is loaded, the processor also loads the hidden part of the segment register with the base address, segment limit, and access control information from the descriptor pointed to by the segment selector.

This allows the processor to translate addresses without taking extra bus cycles to read the base address and limit from the segment descriptor.

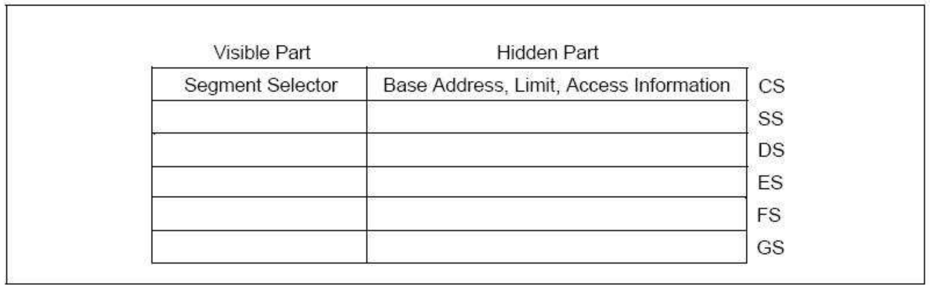
In systems in which multiple processors have access to the same descriptor tables, it is the responsibility of software to reload the segment registers when the descriptor tables are modified.

If this is not done, an old segment descriptor cached in a segment register might be used after its memory-resident version has been modified.

Two kinds of instructions are provided for loading the segment registers:

1. Direct load instructions such as the MOV, LES, LGS, and LFS instructions explicitly reference the segment registers.

2. Implied load instructions such as the far pointer versions of the CALL, JMP, and RET instructions, the SYSENTER and SYSEXIT instructions, and the IRET, INT*n*, INTO and INT3 instructions. These instructions change the contents of the CS register as an incidental part of their operation.



Visible Part	Hidden Part	
Segment Selector	Base Address, Limit, Access Information	CS
		SS
		DS
		ES
		FS
		GS

Figure 3-7. Segment Registers

Intel Memory Management

3.5 SYSTEM DESCRIPTOR TYPES

When the S (descriptor type) flag in a segment descriptor is clear, the descriptor type is a system descriptor. The processor recognizes the following types of system descriptors:

- Local descriptor-table (LDT) segment descriptor.
- Task-state segment (TSS) descriptor.
- Call-gate descriptor.
- Interrupt-gate descriptor.
- Trap-gate descriptor.
- Task-gate descriptor.

These descriptor types fall into two categories: system-segment descriptors and gate descriptors.

System-segment descriptors point to system segments (LDT and TSS segments). Gate descriptors are in themselves “gates,” which hold pointers to procedure entry points in code segments (call, interrupt, and trap gates) or which hold segment selectors for TSS’s (task gates).

Table 3-2 shows the encoding of the type field for system-segment descriptors and gate descriptors.

Intel Memory Management

3.5.1 Segment Descriptor Tables

A segment descriptor table is an array of segment descriptors (see Figure 3-10). A descriptor table is variable in length and can contain up to 8192 (213) 8-byte descriptors.

There are two kinds of descriptor tables:

• The global descriptor table (GDT)

• The local descriptor tables (LDT)

Each system must have one GDT defined, which may be used for all programs and tasks in the system.

Optionally, one or more LDTs can be defined. For example, an LDT might be defined for each separate task being run.

The GDT is not a segment itself; instead, it is a data structure in linear address space. The base linear address and limit of the GDT must be loaded into the GDTR register.

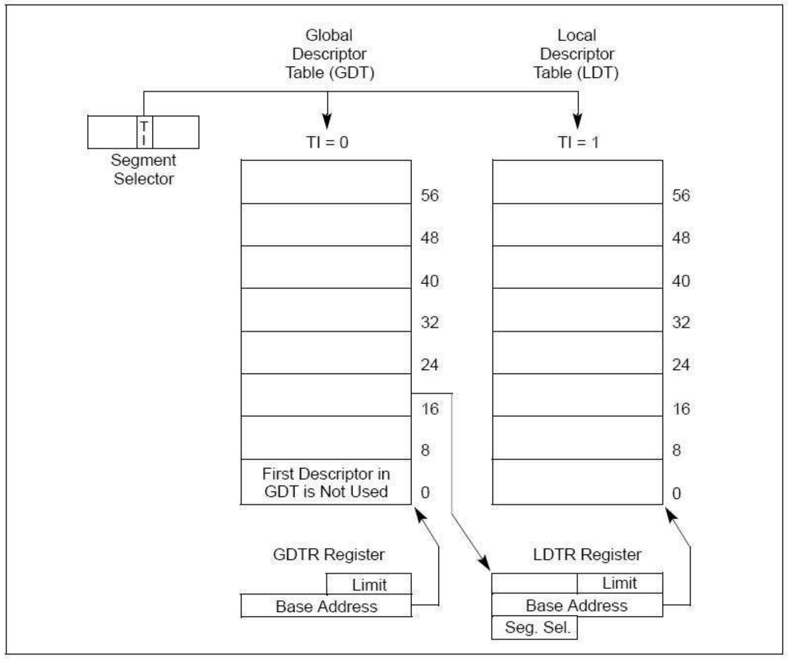


Figure 3-10. Global and Local Descriptor Tables

Intel Memory Management

3.5.1 Segment Descriptor Tables

The LDT is located in a system segment of the LDT type.

The GDT must contain a segment descriptor for the LDT segment. If the system supports multiple LDTs, each must have a separate segment selector and segment descriptor in the GDT.

An LDT is accessed with its segment selector. To eliminate address translations when accessing the LDT, the segment selector, base linear address, limit, and access rights of the LDT are stored in the LDTR register.

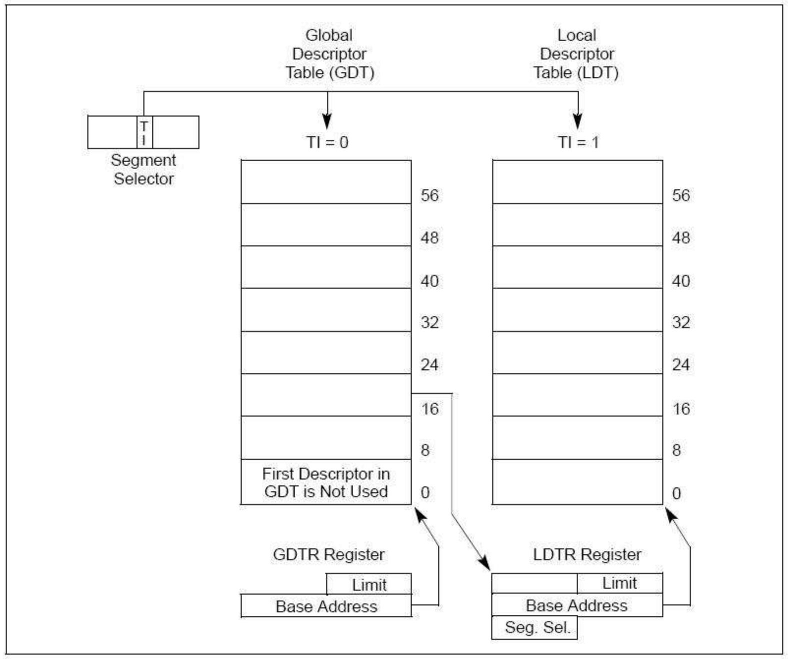


Figure 3-10. Global and Local Descriptor Tables

Coming up!! How does the Intel processor do paging?