

Prerequisites

Jassie Tsai 2002 NCTU



Outline

- ✍ 80386 Protected Mode Review
 - Registers
 - Memory Management
 - Protection Mechanism
 - Interrupts and Exceptions
- ✍ AT&T x86 Assembly Syntax
- ✍ Some Important Features of Linux
- ✍ Related Resources

80386 Protected Mode Review

Registers

80386 Protected Mode Review

General-Purpose Registers

31	16	15	8	7	0	16-bit	32-bit
	AH		AL			AX	EAX
	BH		BL			BX	EBX
	CH		CL			CX	ECX
	DH		DL			DX	EDX
	BP						EBP
	SI						ESI
	DI						EDI
	SP						ESP

- ✍ General-Purpose Register
- ✍ Segment Register
- ✍ Flag Register
- ✍ Memory-Management Registers
- ✍ Control Registers

80386 Protected Mode Review

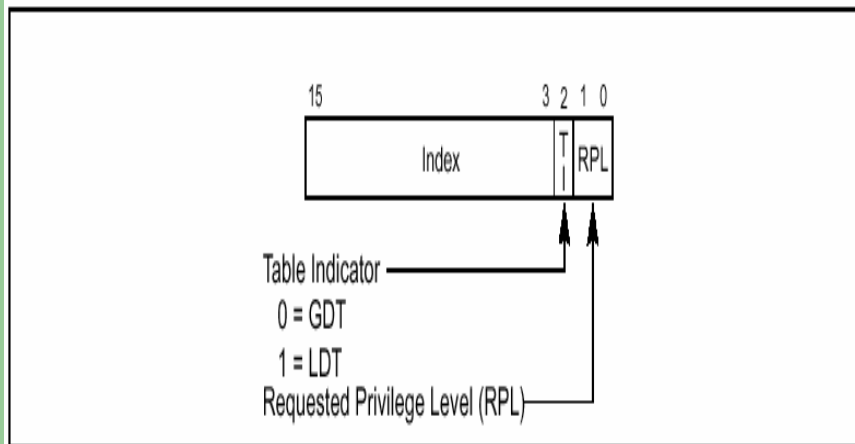


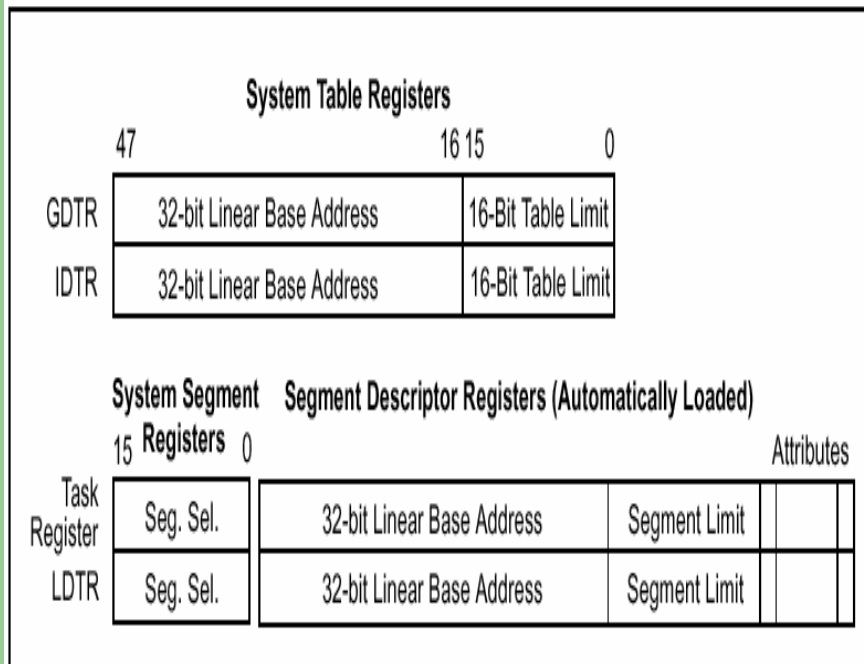
Figure 3-6. Segment Selector

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

Figure 3-7. Segment Registers

- ✍ General-Purpose Register
- ✍ Segment Register
- ✍ Flag Register
- ✍ Memory-Management Registers
- ✍ Control Registers

80386 Protected Mode Review



-  General-Purpose Register
-  Segment Register
-  Flag Register
-  Memory-Management Registers
-  Control Registers

Figure 2-4. Memory Management Registers

80386 Protected Mode Review

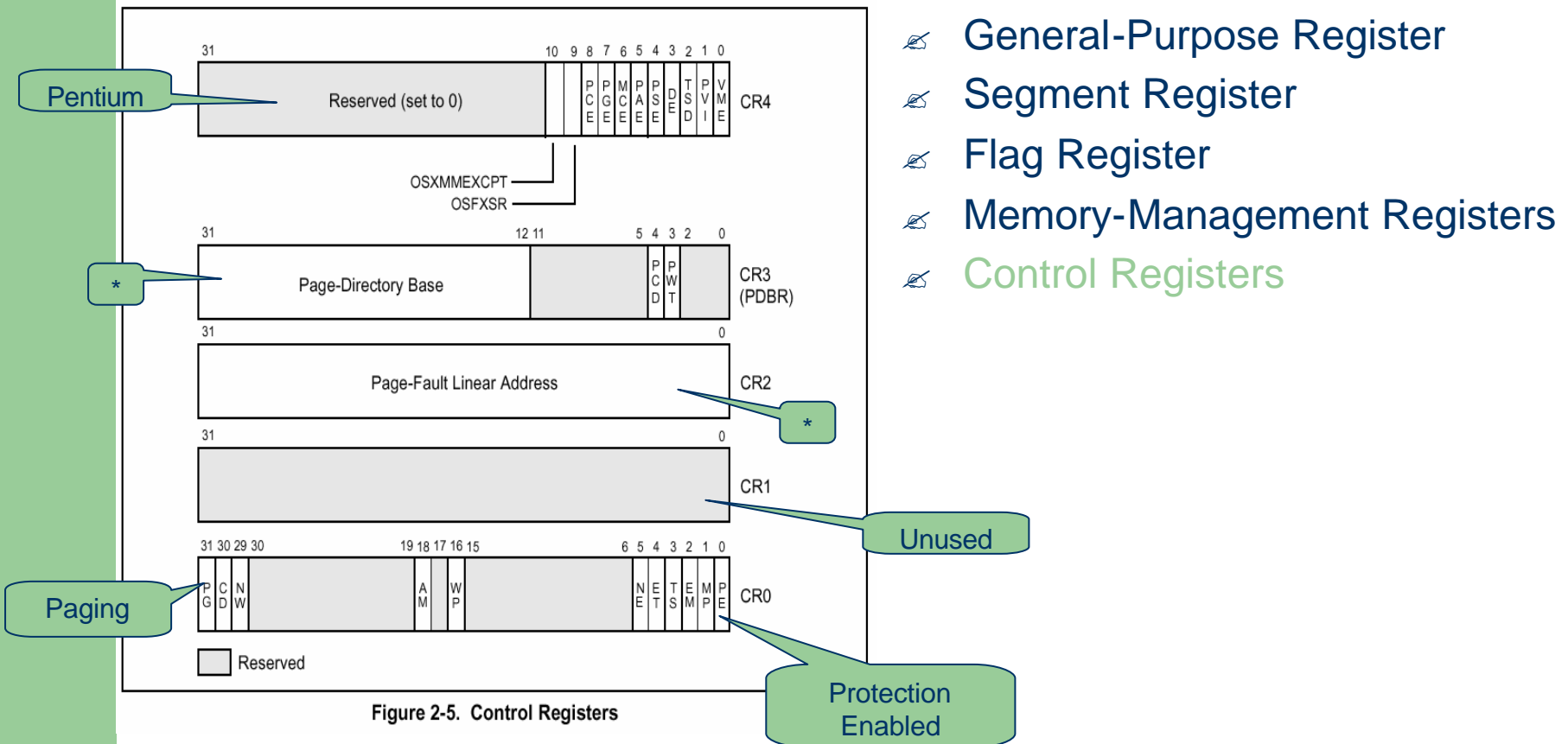


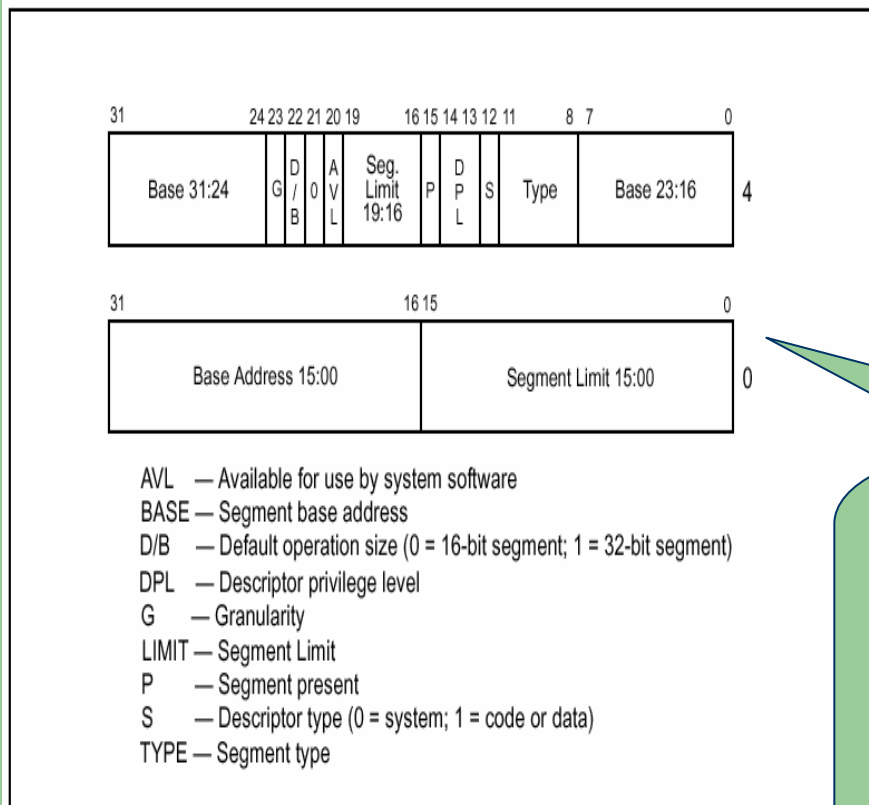
Figure 2-5. Control Registers

80386 Protected Mode Review

Memory Management

80386 Protected Mode

A segment is a region of memory. Segment Descriptor is a data structure to describe the attributes of a segment.



✍ Segment Descriptor

✍ Segmentation

✍ Paging

✍ Page Directory Entry

✍ Page Table Entry

✍ The Whole Mapping

The most important attributes of a segment are:

Base Address -> The starting address of this segment

Limit -> The size of this segment

Type -> code or data or something else?

G -> The granularity of Limit field. (1B or 4KB)

W -> Write-enable?

DPL -> Privilege level needed to access this segment

P -> Present.

Figure 3-8. Segment Descriptor

80386 Protected Mode Review

If P is 0 (Not present in memory), most of other fields could be used freely by OS.

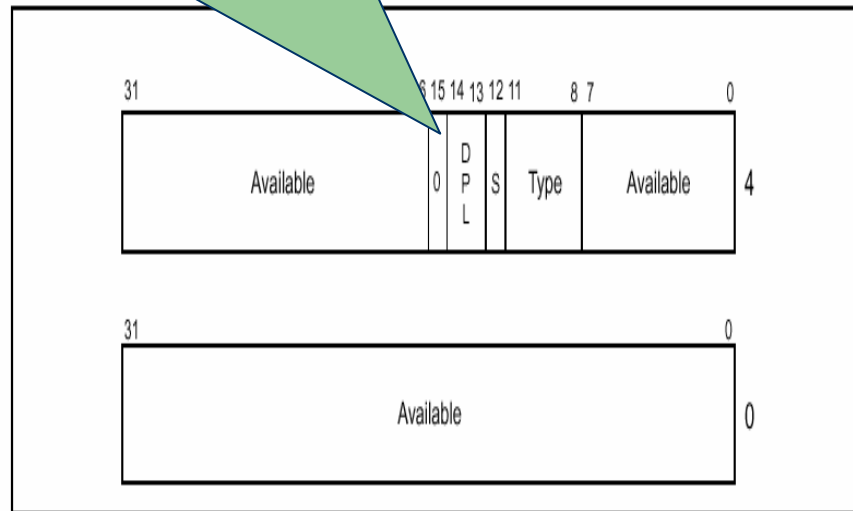
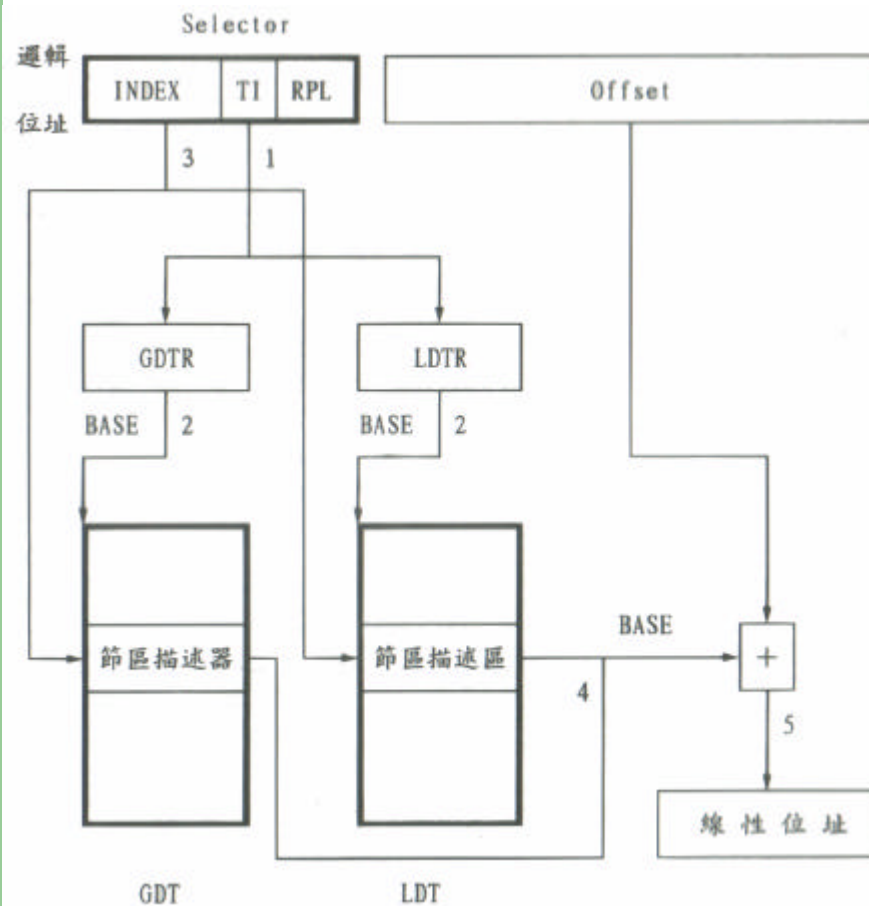


Figure 3-9. Segment Descriptor When Segment-Present Flag Is Clear

- ✍ Segment Descriptor
- ✍ Segmentation
- ✍ Paging
- ✍ Page Directory Entry
- ✍ Page Table Entry
- ✍ The Whole Mapping

80386 Protected Mode Review



- ✍ Segment Descriptor
- ✍ Segmentation
- ✍ Paging
- ✍ Page Directory Entry
- ✍ Page Table Entry
- ✍ The Whole Mapping

Segmentation is necessary in x86 architecture. Any **logical address** (The addresses CPU sees) consists of a 16 bits segment register and a 32 bits offset. This logical address will be translate to **linear address** by x86 segmentation mechanism.

80386 Protected Mode Review

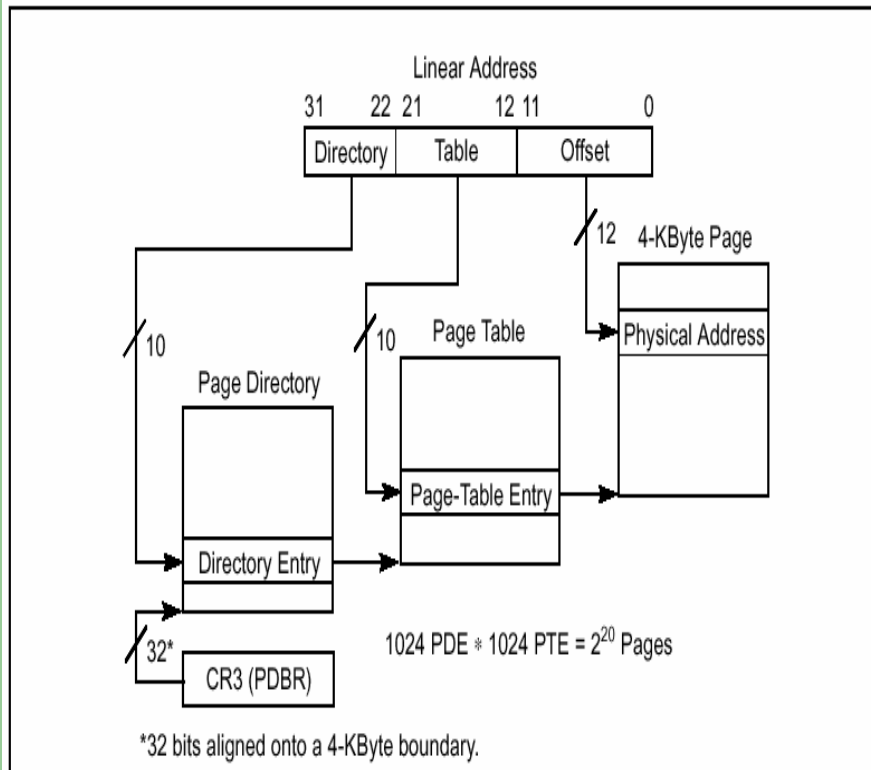


Figure 3-12. Linear Address Translation (4-KByte Pages)

- ✍ Segment Descriptor
- ✍ Segmentation
- ✍ Paging
- ✍ Page Directory Entry
- ✍ Page Table Entry
- ✍ The Whole Mapping

Paging is optional. 80x86 adopt a two level page table layout.

80386 Protected Mode Review

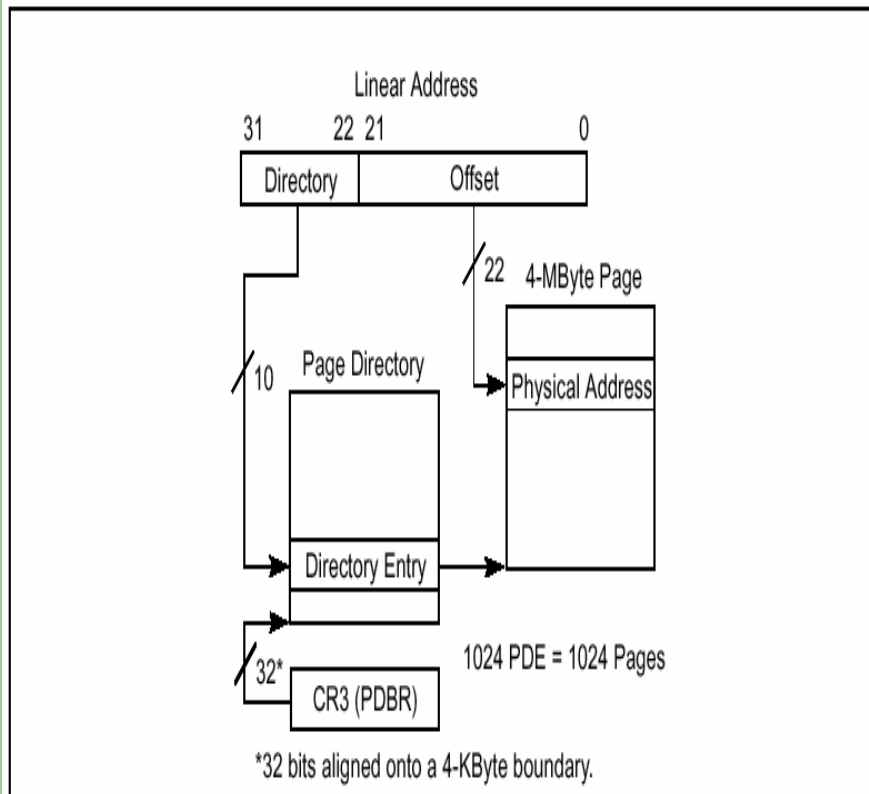


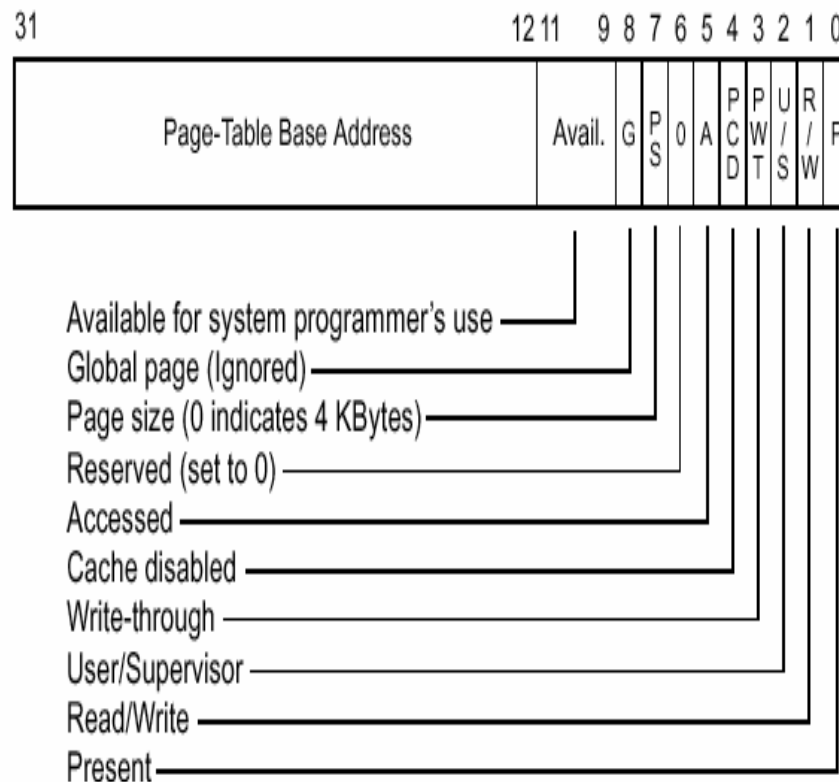
Figure 3-13. Linear Address Translation (4-MByte Pages)

- ✍ Segment Descriptor
- ✍ Segmentation
- ✍ **Paging**
- ✍ Page Directory Entry
- ✍ Page Table Entry
- ✍ The Whole Mapping

One level paging is also supported in Pentium processor.

80386 Protected Mode Review

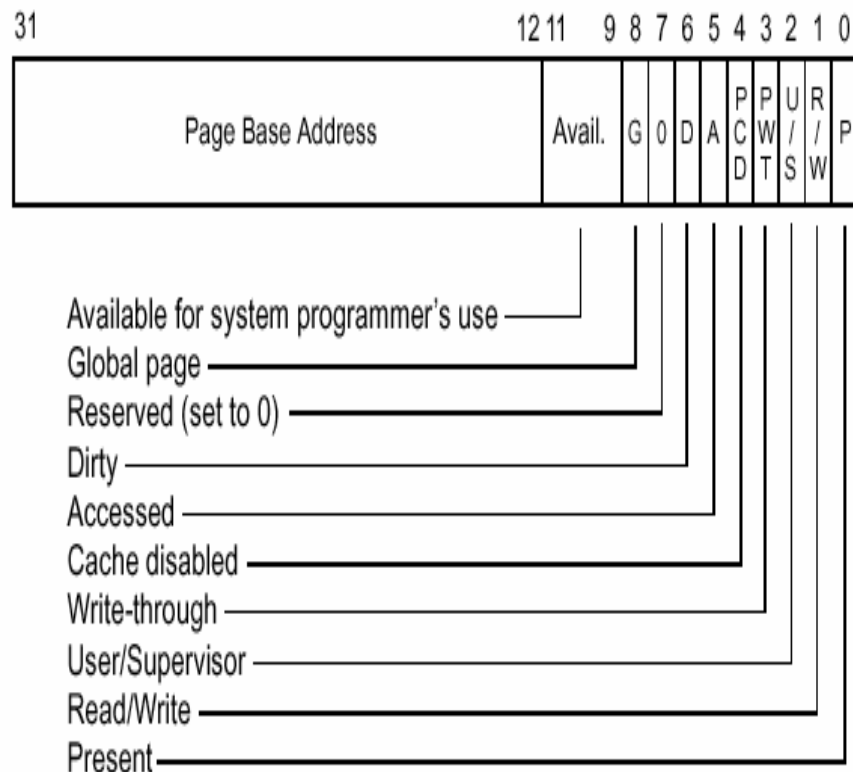
Page-Directory Entry (4-KByte Page Table)



- ✍ Segment Descriptor
- ✍ Segmentation
- ✍ Paging
- ✍ Page Directory Entry
- ✍ Page Table Entry
- ✍ The Whole Mapping

80386 Protected Mode Review

Page-Table Entry (4-KByte Page)



- ✍ Segment Descriptor
- ✍ Segmentation
- ✍ Paging
- ✍ Page Directory Entry
- ✍ Page Table Entry
- ✍ The Whole Mapping

80386 Protected Mode Review

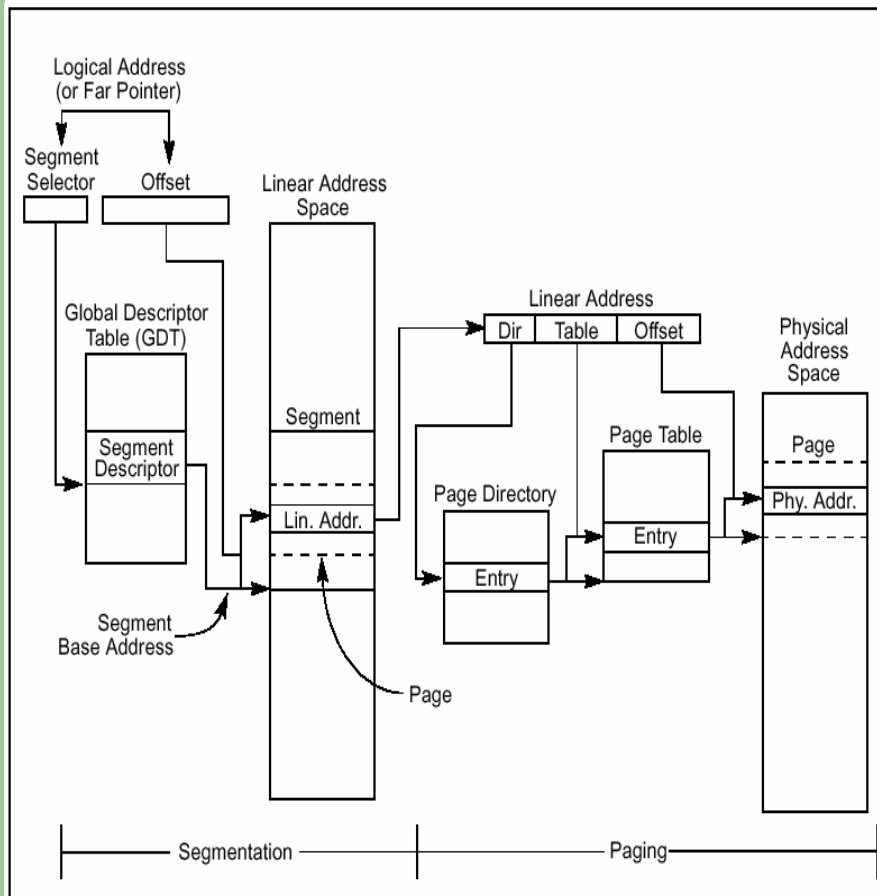


Figure 3-1. Segmentation and Paging

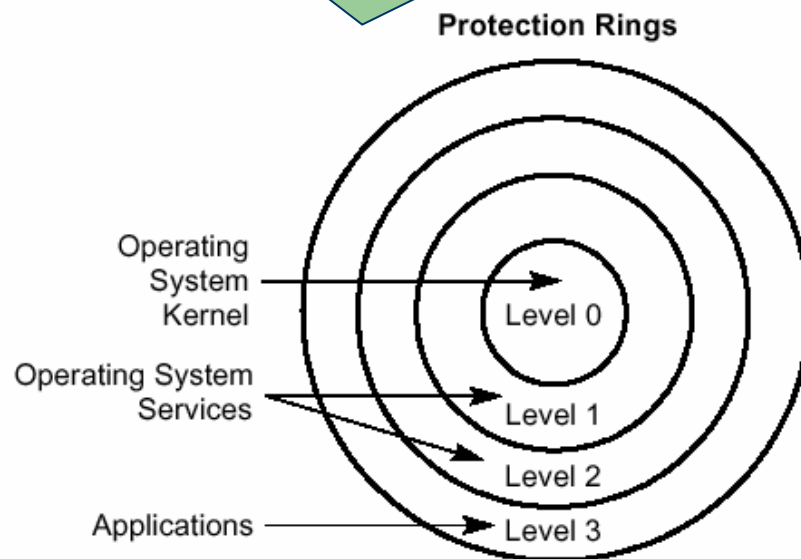
- ✍ Segment Descriptor
- ✍ Segmentation
- ✍ Paging
- ✍ Page Directory Entry
- ✍ Page Table Entry
- ✍ The Whole Mapping

80386 Protected Mode Review

Protection Mechanism

80386 Protected Mode Review

80386 provides 4 privilege level, Linux uses only level 0 (Kernel Mode) and level 3 (User Mode).



- ✍ Privilege Level
- ✍ CPL: The Privilege Level of Running Program.
- ✍ DPL: The Lowest Privilege Level Needed to Reference This Segment.
- ✍ U/S, R/W of Page Directory / Table Entry.
- ✍ TSS (Task State Segment)
- ✍ TSSD (Task State Segment Descriptor)
- ✍ Stack Switching

80386 Protected Mode Review

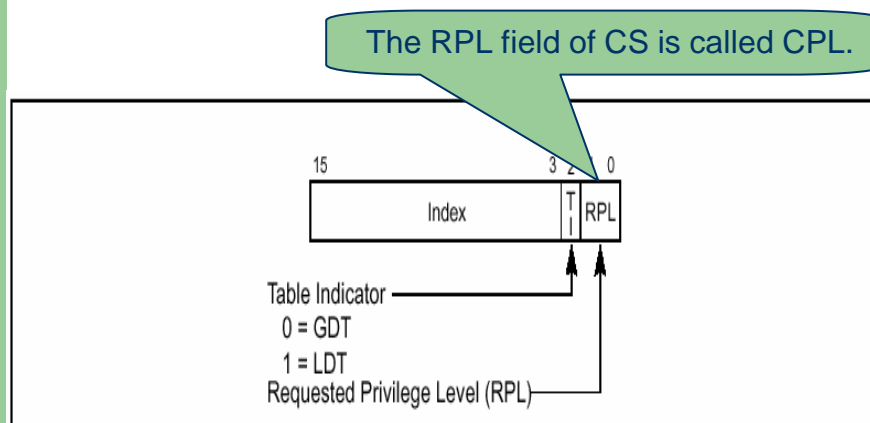


Figure 3-6. Segment Selector

- ✍ Privilege Level
- ✍ CPL: The Privilege Level of Running Program.
- ✍ DPL: The Lowest Privilege Level Needed to Reference This Segment.
- ✍ U/S, R/W of Page Directory / Table Entry.
- ✍ TSS (Task State Segment)
- ✍ TSSD (Task State Segment Descriptor)
- ✍ Stack Switching

80386 Protected Mode Review

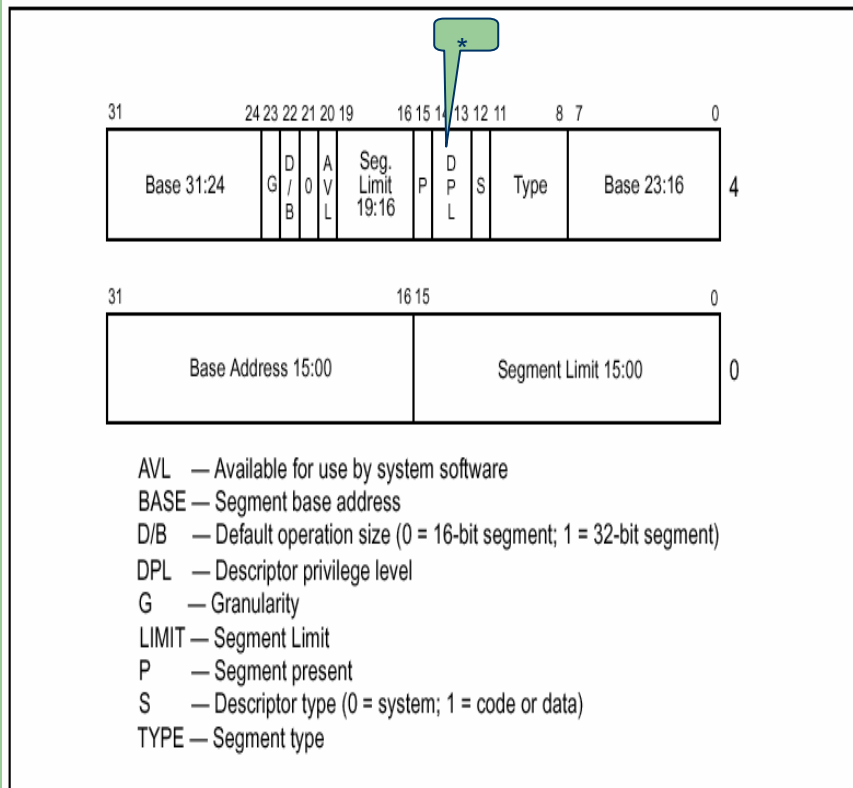
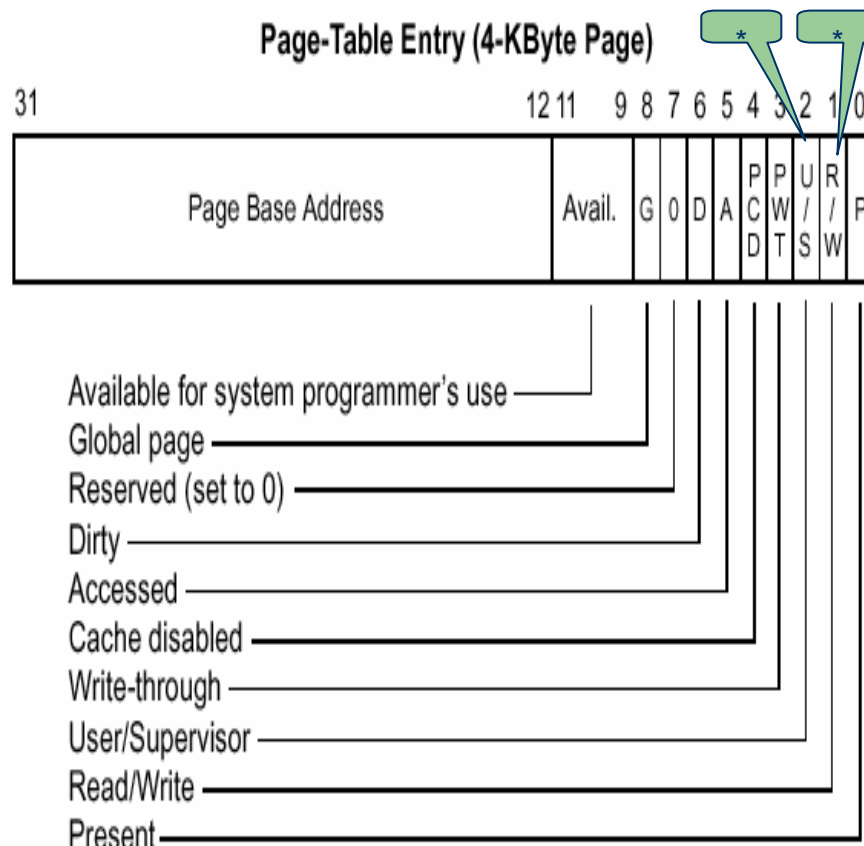


Figure 3-8. Segment Descriptor

- ✍ Privilege Level
- ✍ CPL: The Privilege Level of Running Program.
- ✍ DPL: The Lowest Privilege Level Needed to Access This Segment.
- ✍ U/S, R/W of Page Directory / Table Entry.
- ✍ TSS (Task State Segment)
- ✍ TSSD (Task State Segment Descriptor)
- ✍ Stack Switching

80386 Protected Mode Review



- ✍ Privilege Level
- ✍ CPL: The Privilege Level of Running Program.
- ✍ DPL: The Lowest Privilege Level Needed to Reference This Segment.
- ✍ U/S, R/W of Page Directory / Table Entry.
- ✍ TSS (Task State Segment)
- ✍ TSSD (Task State Segment Descriptor)
- ✍ Stack Switching

80386 Protected Mode Review

A bit map to tell CPU which I/O port can be accessed by this process.

31	I/O Map Base Address	T	100
	LDT Segment Selector		96
	GS		92
	FS		88
	DS		84
	SS		80
	CS		76
	ES		72
	EDI		68
	ESI		64
	EBP		60
	ESP		56
	EBX		52
	EDX		48
	ECX		44
	EAX		40
	EFLAGS		36
	EIP		32
	CR3 (PDBR)		28
	SS2		24
	ESP2		20
	SS1		16
	ESP1		12
	SS0		8
	ESP0		4
	Previous Task Link		0

Reserved bits. Set to 0.

Privilege Level

TSS is a block of memory to describe the context of a process. 80386 provides facilities to simplify context switching. Linux does not use this facilities to do context switching. But TSS is still needed for each process because it contain some important information which will be checked when executing certain instructions.

Table Entry.

TSS (Task State Segment)

TSSD (Task State Segment Descriptor)

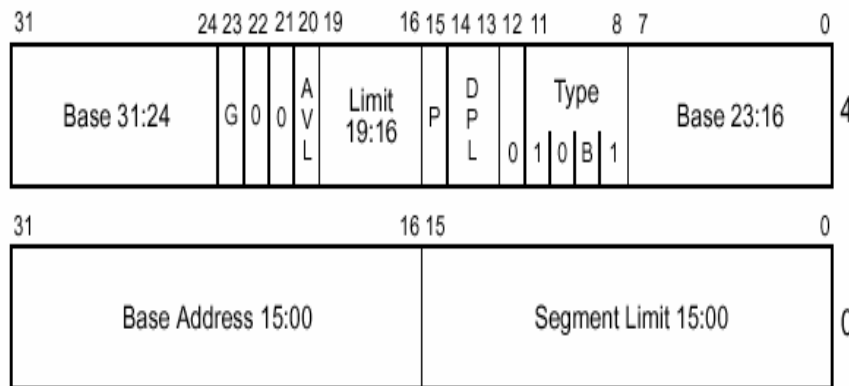
Stack Switching

Three stack address provided for level 0, 1 and 2. Used in Stack switching.

80386 Protected Mode Review

TSSD is a descriptor used to describe a TSS. TSSD is usually put in GDT.

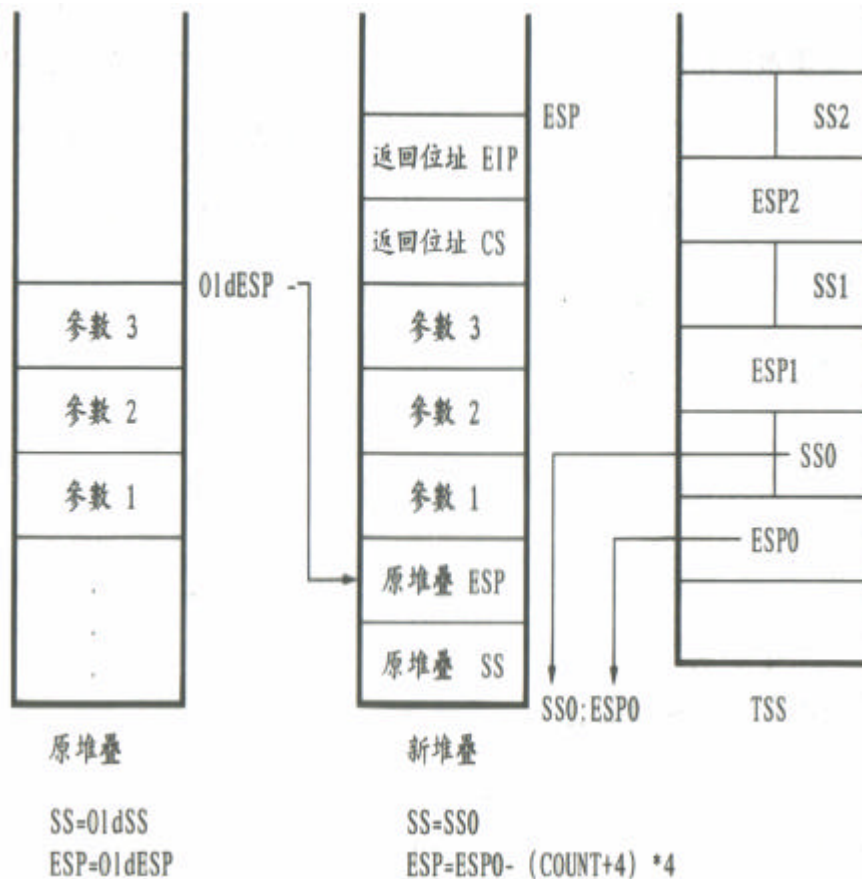
TSS Descriptor



AVL Available for use by system software
 B Busy flag
 BASE Segment Base Address
 DPL Descriptor Privilege Level
 G Granularity
 LIMIT Segment Limit
 P Segment Present
 TYPE Segment Type

- ✍ Privilege Level
- ✍ CPL: The Privilege Level of Running Program.
- ✍ DPL: The Lowest Privilege Level Needed to Reference This Segment.
- ✍ U/S, R/W of Page Directory / Table Entry.
- ✍ TSS (Task State Segment)
- ✍ TSSD (Task State Segment Descriptor)
- ✍ Stack Switching

80386 Protected Mode Review



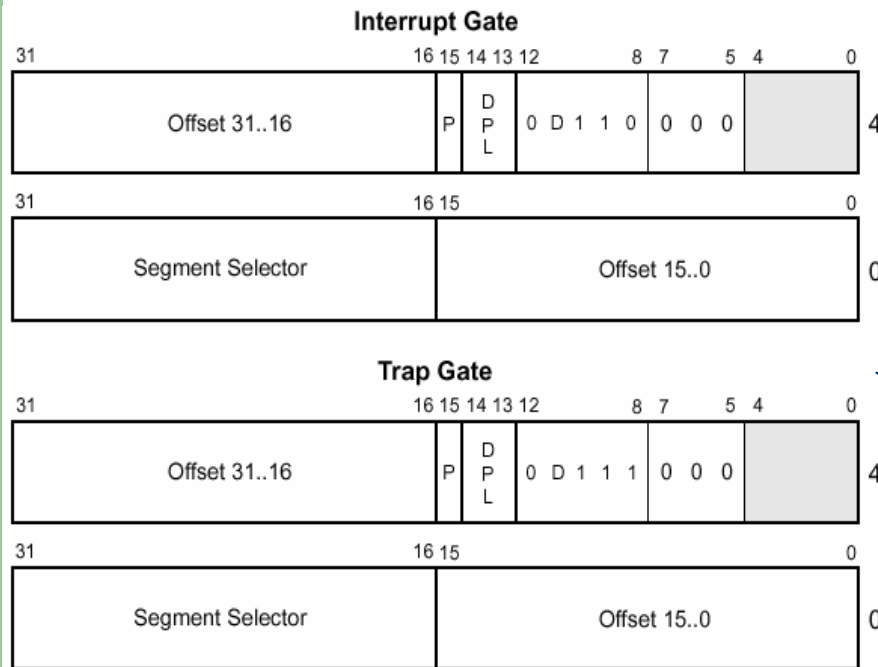
- ✍ Privilege Level
- ✍ CPL: The Privilege Level of Running Program.
- ✍ DPL: The Lowest Privilege Level Needed to Reference This Segment.
- ✍ U/S, R/W of Page Directory / Table Entry.
- ✍ TSS (Task State Segment)
- ✍ TSSD (Task State Segment Descriptor)
- ✍ Stack Switching

80386 Protected Mode Review

Interrupts and Exceptions

80386 Protected M

The address and attributes of a ISR are described with a descriptor called Interrupt Descriptor.



Interrupt Descriptor

- Interrupt Gate
- Trap Gate

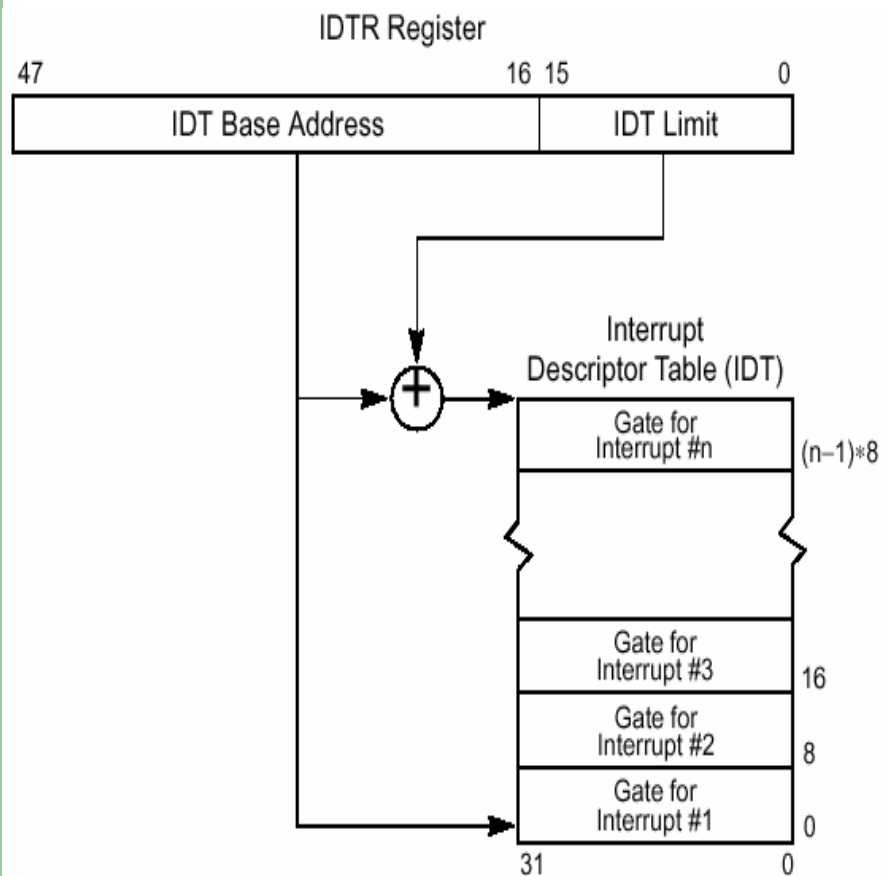
IDT (Interrupt Descriptor Table) & IDTR (Interrupt Descriptor Table Register)

There are two such descriptor – Interrupt Gate and Trap Gate. The different between the two descriptor is that when interrupt is up, ISRs described by Interrupt Gate will disable interrupt before execute ISR. ISRs described by Trap Gate will not change the value of IF.

DPL Descriptor Privilege Level
 Offset Offset to procedure entry point
 P Segment Present flag
 Selector Segment Selector for destination code segment
 D Size of gate: 1 = 32 bits; 0 = 16 bits

Reserved

80386 Protected Mode Review

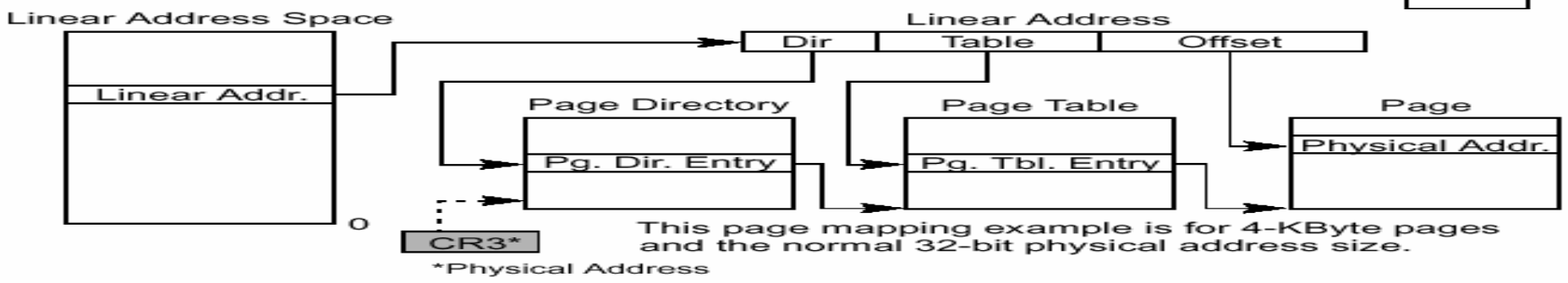
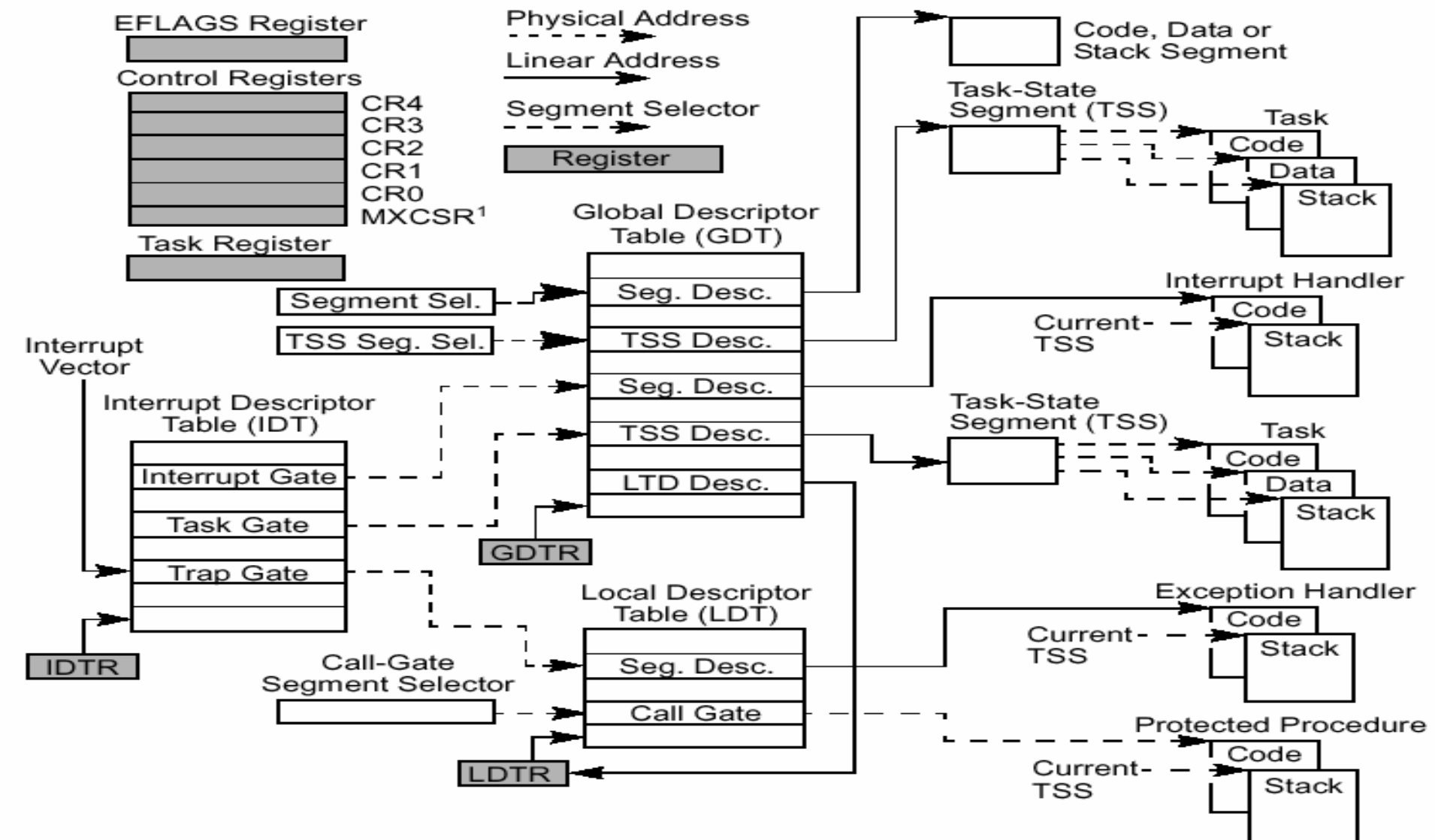


Interrupt Descriptor

- Interrupt Gate
- Trap Gate

IDT (Interrupt Descriptor Table) & IDTR (Interrupt Descriptor Table Register)

Interrupt Descriptors are collected into Interrupt Descriptor Table (IDT). A register called IDTR point to the starting physical address of IDT.



*Physical Address

AT&T x86 Assembly Syntax

- ✍ In order to maintain compatibility with the output of gcc, as support AT&T System V/386 assembler syntax. Notable differences between the two syntaxes are:
 - AT&T immediate operands are preceded by '\$'; Intel immediate operands are undelimited. (Intel 'push 4' is AT&T 'pushl \$4')
 - AT&T register operands are preceded by '%'; Intel register operands are undelimited. (Intel EAX is AT&T %EAX)
 - AT&T absolute (as opposed to PC relative) jump/call operands are prefixed by '*'; they are undelimited in Intel syntax.
 - AT&T and Intel syntax use the opposite order for source and destination operands. Intel 'add eax, 4' is 'addl \$4, %eax'

AT&T x86 Assembly Syntax

- ✍ In AT&T syntax the size of memory operands is determined from the last character of the operand name. Opcode suffixes of **'b'**, **'w'**, and **'l'** specify the byte, word (16 bits), and long (32 bits) memory reference. Intel **'mov al, byte ptr foo'** is **'movb foo, %al'**.
- ✍ Immediate long jumps and calls are **'lcall/ljmp \$section, \$offset'** in AT&T syntax; the Intel syntax is **'call/jmp far section:offset'**.
- ✍ The far return instruction is **'ret \$stack-adjust'** in AT&T syntax; Intel syntax is **'ret far stack-adjust'**

AT&T x86 Assembly Syntax

- ✍ Almost all opcodes have the same names in AT&T and Intel format. But there are a few exceptions:
 - Intel **'cbw'**, **'cwde'**, **'cwtl'**, **'cdq'** are called **'cwtl'**, **'cwtl'**, **'cwtd'**, and **'cltd'** in AT&T naming.
 - Far **call/jump** are **'lcall'** and **'ljmp'** in AT&T syntax.
 - The **sign/zero extend instruction** are modified to be **movsxx** and **movzxx**. Possible **xx** are **'bl'** (from byte to long), **'bw'** (from byte to word), and **'wl'** (from word to long)

AT&T x86 Assembly Syntax

- ✍ Opcode prefixes are used to modify the following opcode. Here is a list of opcode prefixes:
- Section override prefixes **'cs'**, **'ds'**, **'ss'**, **'es'**, **'fs'**, **'gs'**.
 - The bus lock prefix **'lock'**.
 - The wait for coprocessor prefix **'wait'**.
 - The **'rep'**, **'repe'**, and **'repne'** prefixes are added to string instructions to make them repeat `%ecx` times.

AT&T x86 Assembly Syntax

- ✍ An Intel syntax indirect memory reference of the form
 - section: [base + index * scale + disp]
is translated into the AT&T syntax
section: disp(base, index, scale)
 - For example:
 - ✍ AT&T: -4(%ebp), Intel: [ebp - 4]
 - ✍ AT&T: foo(,%eax,4), Intel: [foo+ eax*4]
 - ✍ AT&T: foo(,1), Intel: [foo] => This is a syntactic exception
 - ✍ AT&T: %gs:foo, Intel: gs:foo



Some Important Features of Linux



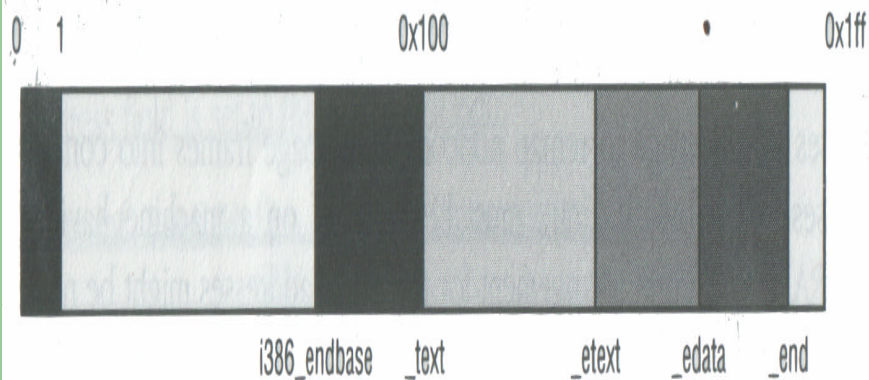
Segmentation In Linux

- ✍ UNIXs (including Linux) do not use segmentation mechanism because it is unpopular and complicated. The Linux use segmentation only when required by the Intel 80x86 architecture.
- ✍ In particular, all processes use the same logical address, so the total number of segments to be defined is quite limited and it is possible to store all Segment Descriptor in the Global Descriptor Table (GDT). Local Descriptor Tables are not used by kernel.
- ✍ The following are the segments used by Linux:
 - `__KERNEL_CS`: A kernel code segment, shared by all kernel threads. Starting from 0 to $2^{32} - 1$, highest privilege level.
 - `__KERNEL_DS`: A kernel data segment, shared by all kernel threads. Starting from 0 to $2^{32} - 1$, highest privilege level.

Segmentation In Linux

- `__USER_CS`: A user code segment, shared by all user processes. Starting from 0 to $2^{32} - 1$, lowest privilege level.
- `__USER_DS`: A user data segment, shared by all user processes. Starting from 0 to $2^{32} - 1$, lowest privilege level.
- A Task State Segment (TSS) for each processor. The descriptors of these segments are stored in the GDT.

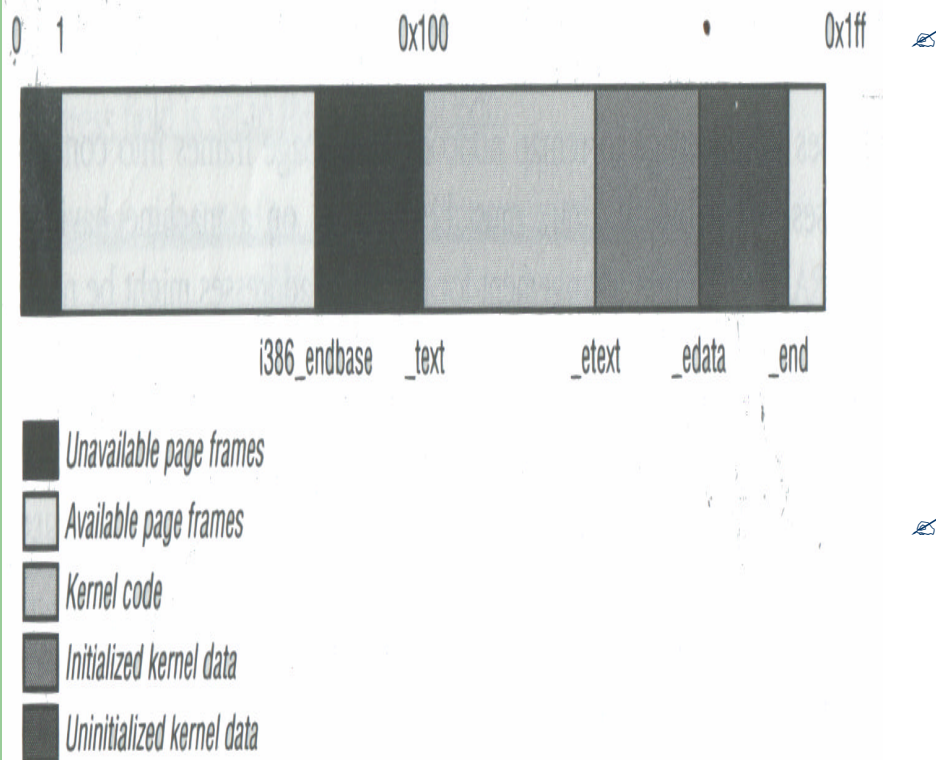
Reserved Page Frames



- Unavailable page frames
- Available page frames
- Kernel code
- Initialized kernel data
- Uninitialized kernel data

- The kernel's code and data are stored in a group of reserved page frames. A page contained in one of these page frames can never be dynamically assigned or swapped to disk.
- As a general rule, the Linux kernel is installed in RAM starting from physical address 0x10000, that is, from the second megabyte.

Reserved Page Frames

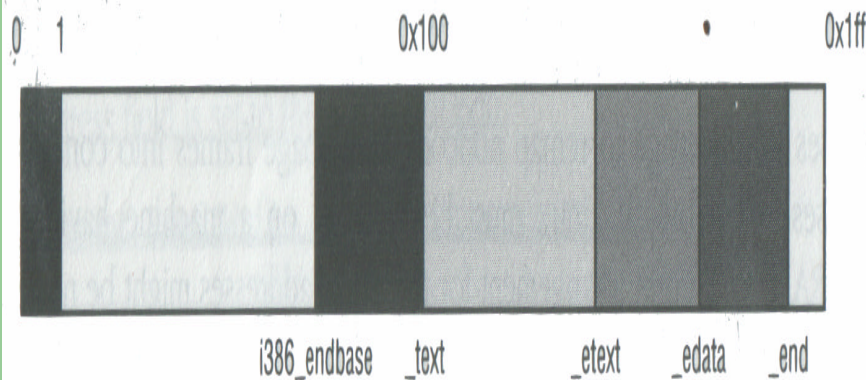


The reason is that PC architecture has several peculiarities:

- Page frame 0 is used by BIOS to store the system hardware configuration detected during Power-On Self-Test.
- Physical addresses ranging from 0xa0000 to 0xffff are reserved to BIOS routines and to map the internal memory of ISA graphics cards.

In order to avoid loading the kernel into noncontiguous page frames, Linux prefers to skip the first megabyte of RAM.

Reserved Page Frames



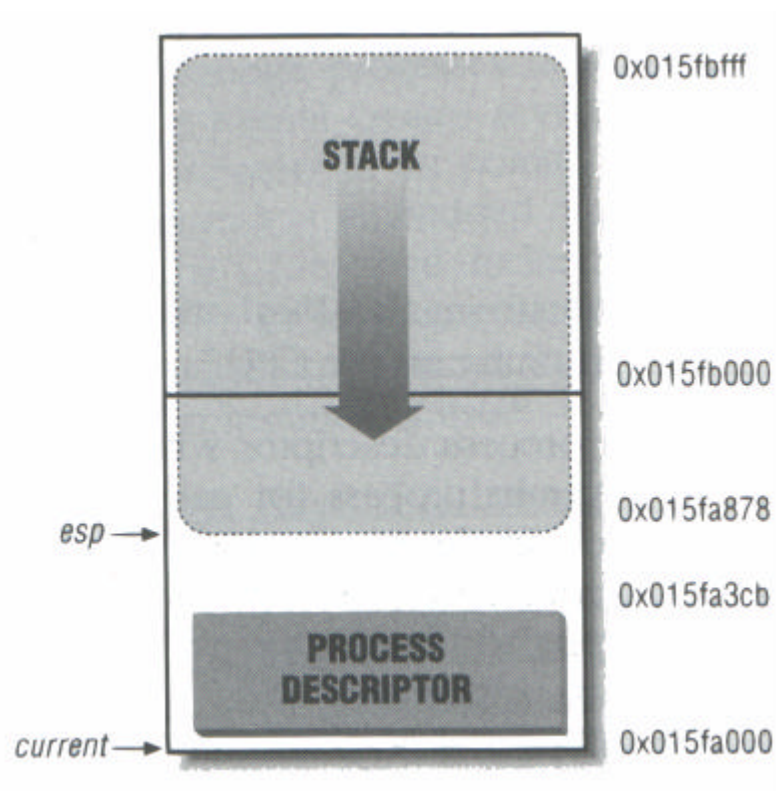
- Unavailable page frames
- Available page frames
- Kernel code
- Initialized kernel data
- Uninitialized kernel data

- The symbol `_text`, which corresponds to physical address `0x100000`, denotes the linear address of the first byte of kernel code.
- The end of the kernel code is similarly identified by the symbol `_etext`.
- Kernel data is divided into two groups: initialized (DATA) and uninitialized (BSS).
- The initialized data starts right after `_etext` and ends at `_edata`.
- The uninitialized data follows and ends up at `_end`.
- The linear address corresponding to the first physical address reserved to the BIOS or to the hardware device is stored in the `i386_endbase`.

Process Page Table

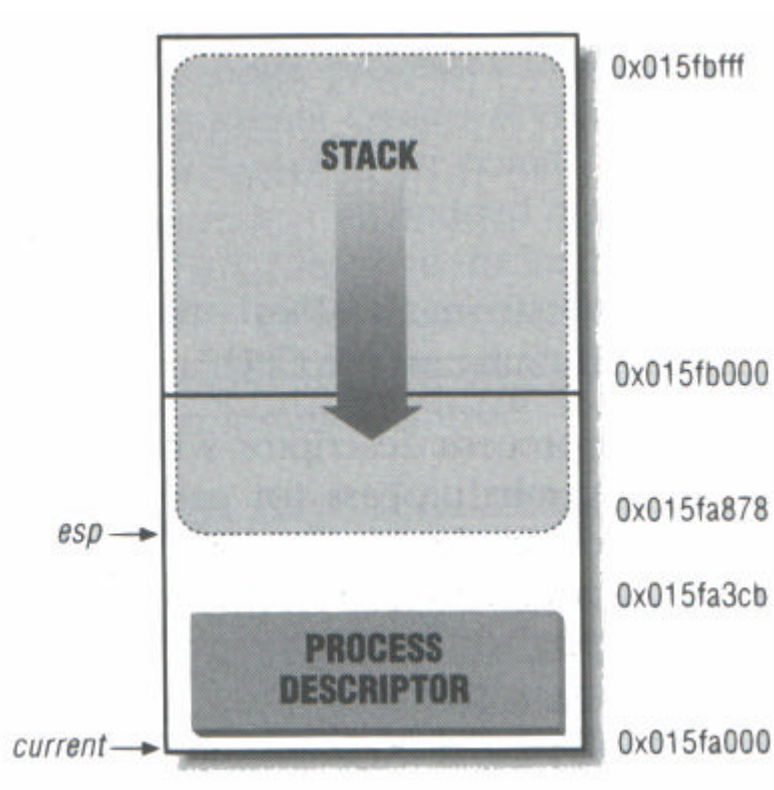
- ✍ The linear address space of a process is divided into two parts:
 - Linear addresses from 0 to `PAGE_OFFSET - 1` (usually, `3G - 1`) can be addressed when the process is in either User or Kernel Mode.
 - Linear address from `PAGE_OFFSET` to `0xffffffff` can be addressed only when the process is in Kernel Mode.
- ✍ The content of the entries of the Page Table that map linear addresses lower than `PAGE_OFFSET` depends on the specific process. Conversely, the remaining entries are the same for all process; they are kernel data and code.

Process Descriptor & Kernel Mode Process Stack



- ✍ Every user process or kernel thread have a process descriptor (PCB, or `task_struct`).
- ✍ Every user process must have two stack; one for User Mode and one for Kernel Mode.
- ✍ The kernel stack for a user process is stored in current TSS and stack switching takes place at interrupt time.
- ✍ Linux stores two different data structures for each process (or kernel thread) in a single 8KB memory area: the process descriptor and the Kernel Mode process stack.

Process Descriptor & Kernel Mode Process Stack



- ✍ The C language allows such a hybrid structure to be conveniently represented by means of the following union construct:

```
union task_union {  
    struct task_struct task;  
    unsigned long stack[2048];  
};
```

- ✍ Hence, the kernel can easily obtain the process descriptor pointer of the process currently running on the CPU from the value of the esp register as follow:

```
movl $0xffffe000, %ecx  
andl %esp, %ecx  
movl %ecx, p
```

Related Resources

- ✍ “Intel Architecture Software Developer’s Manual”, Volume 1,2,3. Those books can be download from Intel’s official web site.
- ✍ “80486 PENTIUM 保護模式原理與實務”, 旗標.
- ✍ The official manuals of gcc, make, ld, as, binutils can be download from <http://www.gnu.org/manual/> .