Protected Mode

By Chris Giese

What is Protected Mode?

The 8088 CPU used in the original IBM PC was not very scalable. In particular, there was no easy way to access more than 1 megabyte of physical memory. To get around this while allowing backward compatability, Intel designed the 80286 CPU with two modes of operation: *real mode*, in which the '286 acts like a fast 8088, and *protected mode* (now called 16-bit protected mode). Protected mode allows programs to access more than 1 megabyte of physical memory, and protects against misuse of memory (i.e. programs can't execute a data segment, or write into a code segment). An improved version, 32-bit protected mode, first appeared on the '386 CPU.

How do Real Mode and Protected Mode Differ?

Table 1: differences between real- and protected modes.

	Real Mode	**16-bit Protected Mode**	**32-bit Protected Mode**
Segment base address	20-bit (1M byte range) = 16 * segment register	24-bit (16M byte range), from *descriptor*	32-bit (4G byte range), from *descriptor*
Segment size (limit)	16-bit, 64K bytes (fixed)	16-bit, 1-64K bytes	20-bit, 1-1M bytes or 4K-4G bytes
Segment protection	no	yes	yes
Segment register	segment base adr / 16	*selector*	*selector*

I thought protected mode didn't use segmented memory...

The segments are still there, but in 32-bit protected mode, you can set the segment limit to 4G bytes. This is the maximum amount of physical memory addressable by a CPU with a 32-bit address bus. Limit-wise, the segment then "disappears" (though other protection mechanisms remain in effect). This reason alone makes 32-bit protected mode popular.

What's a descriptor?

In real mode, there is little to know about the segments. Each is 64K bytes in size, and you can do with the segment what you wish: store data in it, put your stack there, or execute code stored in the segment. The base address of the segment is simply 16 times the value in one of the segment registers.

In protected mode, besides the segment base address, we also need the segment size (limit) and some flags indicating what the segment is used for. This information goes into an 8-byte data structure called a *descriptor*:

Table 2: code/data segment descriptor.

Lowest byte	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Highest byte
Limit 7:0	Limit 15:8	Base 7:0	Base 15:8	Base 23:16	Access	Flags, Limit 19:16	Base 31:24

This is a 32-bit ('386) descriptor. 16-bit ('286) descriptors have to top two bytes (Limit 19:16, Flags, and Base 31:24) set to zero. The Access byte indicates segment usage (data segment, stack segment, code segment, etc.):

Table 3: access byte of code/data segment descriptor.

Highest bit	Bits 6, 5	Bit 4	Bits 3	Bit 2	Bit 1	Lowest bit
Present	Privilege	**1**	Executable	Expansion direction/ conforming	Writable/ readable	Accessed

- Present bit. Must be set to one to permit segment access.
- Privilege. Zero is the highest level of privilege (*Ring 0*), three is the lowest (*Ring 3*).
- Executable bit. If one, this is a code segment, otherwise it's astack/data segment.
- Expansion direction (stack/data segment). If one, segment grows downward, and offsets within the segment must be **greater than** the limit.
- Conforming (code segment). Privilege-related.
- Writable (stack/data segment). If one, segment can be written to.
- Readable (code segment). If one, segment can be read from. (Code segments are not writable.)
- Accessed. This bit is set whenever the segment is read from or written to.

The 4-bit Flags value is non-zero only for 32-bit segments:

Table 4: flags nybble.

Highest bit	Bit 6	Bit 5	Bit 4
Granularity	Default Size	**0**	**0**

The Granularity bit indicates if the segment limit is in units of 4K byte pages (G=1) or if the limit is in units of bytes (G=0).

For stack segments, the Default Size bit is also known as the B (Big) bit, and controls whether 16- or 32-bit values are pushed and popped. For code segments, the D bit indicates whether instructions will operate on 16-bit (D=0) or 32-bit (D=1) quantities by default. To expand upon this: when the D bit is set, the segment is *USE32*, named after the assembler directive of the same name. The following sequence of hex bytes:

```
B8 90 90 90 90
```

will be treated by the CPU as a 32-bit instruction, and will disassemble as

```
mov eax, 90909090h
```

In a 16-bit (*USE16*) code segment, the same sequence of bytes would be equivalent to

```
mov ax,9090h
nop
nop
```

Two special opcode bytes called the *Operand Size Prefix* and the *Address Length Prefix* reverse the sense of the D bit for the instruction destination and source, respectively. These prefixes affect only the instruction that immediately follows them.

Bit 4 of the Access byte is set to one for code or data/stack segments. If this bit is zero, you have a *system segment*. These come in several varieties:

- *Task State Segment (TSS)*. These are used to simplify multitasking. The '386 or higher CPU has four sub-types of TSS.
- *Local Descriptor Table (LDT)*. Tasks can store their own private descriptors here, instead of the GDT.
- *Gates*. These control CPU transitions from one level of privilege to another. Gate descriptors have a different format than other descriptors:

Table 5: gate descriptor.

Lowest byte	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Highest byte
Offset 7:0	Offset 15:8	Selector 7:0	Selector 15:8	Word Count 4:0	Access	Offset 23:16	Offset 31:24

Note the Selector field. Gates work through indirection, and require a separate code or TSS descriptor to function.

Table 6: access byte of system segment descriptor.

Highest bit	Bits 6, 5	Bit 4	Bits 3, 2, 1, 0
Present	Privilege	**0**	Type

Table 7: System segment types.

Type	Segment function	Type	Segment function
0	(invalid)	8	(invalid)
1	Available '286 TSS	9	Available '386 TSS
2	LDT	10	(undefined, reserved)
3	Busy '286 TSS	11	Busy '386 TSS
4	'286 Call Gate	12	'386 Call Gate
5	Task Gate	13	(undefined, reserved)
6	'286 Interrupt Gate	14	'386 Interrupt Gate
7	'286 Trap Gate	15	'386 Trap Gate

Whew! For now, just remember that TSSes, LDTs, and gates are the three main types of system segment.

Where are the descriptors?

They are stored in a table in memory: the *Global Descriptor Table (GDT)*, *Interrupt Descriptor Table (IDT)*, or one of the Local Descriptor Tables. The CPU contains three registers: GDTR, which must point to the GDT, IDTR, which must point to the IDT (if interrupts are used), and LDTR, which must point to the LDT (if the LDT is used). Each of these tables can hold up to 8192 descriptors.

What's a selector?

In protected mode, the segment registers contain *selectors*, which index into one of the descriptor tables. Only the top 13 bits of the selector are used for this index. The next lower bit choses between the GDT and LDT. The lowest two bits of the selector set a privilege value.

How do I enter protected mode?

Entering protected mode is actually rather simple, and is is described in many other tutorials. You must:

- create a valid Global Descriptor Table (GDT),
- (optional) create a valid Interrupt Descriptor Table (IDT),
- disable interrupts,
- point GDTR to your GDT,
- (optional) point IDTR to your IDT,
- set the PE bit in the MSW register,
- do a far jump (load both CS and IP/EIP) to enter protected mode (load CS with the code segment selector),
- load the DS and SS registers with the data/stack segment selector,
- set up a pmode stack,
- (optional) enable interrupts.

How do I get back to Real Mode?

On the '386:

- disable interrupts,
- do a far jump to a 16-bit code segment (i.e. switch briefly to 16-bit pmode),
- load SS with a selector to a 16-bit data/stack segment,
- clear the PE bit,
- do a far jump to a real-mode address,
- load the DS, ES, FS, GS, and SS registers with real-mode values,
- (optional)set IDTR to real-mode values (base 0, limit 0xFFFF),
- re-enable interrupts.

Before returning to real mode, CS and SS must contain selectors pointing to descriptors that are "appropriate to real mode". These have a limit of 64K bytes, are byte-granular (Flags nybble=0), expand-up, writable (data/stack segment only), and present (Access byte=1xx1001x).

On the '286, you can't simply clear the PE bit to leave protected mode. The only way out is to reset the CPU. This can be done by telling the keyboard controller to pulse the reset line of the CPU, or it can be done by *triple-faulting* the CPU (see Robert Collins' web site: www.x86.org).

What pitfalls have you encountered?

- You must pay extreme attention to detail here. One wrong bit will make things fail. Protected mode errors often triple-fault the CPU, making it reset itself. Be prepared to see this happen again and again.
- Most library routines probably won't work. `printf()`, for example, won't work because it evenutally calls either a DOS or BIOS service to put text on the screen. Unless you have a *DOS extender*, these

services are unavailable in protected mode. I had good luck using `sprintf()` to put formatted text in a buffer, which I then wrote to the screen with my own protected-mode routine.

- Before clearing the PE bit, the segment registers **must** point to descriptors that are appropriate to real mode. This means a limit of exactly 0xFFFF (see other restrictions above). One of my demo programs had ES pointing to a text-video segment. With a limit of 0xFFFF, things worked swimmingly. With a limit of 3999 (80 * 25 * 2 - 1), the system froze up after returning to real mode and trying to use the ES register.

  > Actually, for DS, ES, FS and GS, the segment limit must be 0xFFFF *or greater*. If you give the segment a limit of 0xFFFFF and make it page-granular, you can access up to 4G of memory from real mode. This has been dubbed *unreal mode*. However, limits other than 0xFFFF (or page-granularity) for CS or SS cause big problems in real mode.

- You can **not** use the '286 LMSW instruction to clear the PE bit. Use MOV CR0, nnn.
- Load **all** segment registers with valid selectors after entering protected mode. I forgot to do this with ES. A protected-mode routine pushed ES, loaded it with a valid selector, and used it. When it tried to pop the old, invalid (real-mode) selector back into ES, it crashed.
- The IDTR **must** also be reset to a value that is appropriate to real-mode before re-enabling interrupts (see above).
- Not all instructions are legal in real mode. If you attempt to use task state segments for multitasking, note that executing the LTR instruction in real-mode will cause an illegal instruction interrupt.
- Descriptor tables in ROM? Section 10.4.3 of 386INTEL.TXT states

  > The GDT (as well as LDTs) should reside in RAM, because the processor modifies the accessed bit of descriptors.

  However, one of my sources (thanks Vinay) states that later CPUs will not attempt to set the Accessed bit in a descriptor if that bit is already set. Check the docs for the CPU you are using.
- The naive code described here will crash if the PC is in Virtual 8086 (V86) mode. This is a fourth mode of operation found on the 386 CPU, with addressing similar to real mode but some of the protection mechanisms of protected mode. You may know that a Windows (or OS/2, or Linux) DOS box runs in V86 mode, but you may not realize that memory managers such as EMM386 also put the CPU in V86 mode.

If you want to start simple, try these tips:

- Don't worry about returning to real mode. Use the reset button :)
- Leave interrupts disabled.
- Don't use an LDT.
- Put only four descriptors in the GDT: null, code, stack/data, and text video.
- Set the segment bases to real-mode values i.e. 16 * real-mode segment register value. This lets you address variables in the same way in both real and protected modes.
- Set all segment limits to their maximum (0xFFFF for 16-bit protected mode).
- Leave all privilege values set to 0 (Ring 0, highest privilege).

- Install some crude exception handlers that simply scribble a message into video memory then halt:

  ```
  void unhand(void)
  {       static const char Msg[]="U n h a n d l e d   I n t e r r u p t ";

          disable();
          movedata(SYS_DATA_SEL, (unsigned)Msg,
                  LINEAR_SEL, 0xB8000,
                  sizeof(Msg));
          while(1); }
  ```

  The alternating spaces in the message are treated as attribute bytes by the PC video hardware, making the text an eye-catching black on green. Put an interrupt gate in the appropriate (all?) descriptor of the IDT, with a selector to your code segment in the trap gate's selector field, and the address of this routine in its offset field.

This tutorial is mirrored here with Chris Giese's permission.