The world of Protected mode

by Gergor Brunmar

Memory models

In the first tutorial, we just printed a character onto the screen using the BIOS interrupts. With this tutorial, I'll try to explain how to enter the *32-bit Protected mode*.

We start as before, by telling the compiler that we want 16-bit instructions and a memory base address of 0x7C00.

```
[BITS 16]

[ORG 0x7C00]
```

Now we want to enter the Protected mode. Before we actually do this, I'll need to explain a little about memory access. In Real mode, you access the memory linear. This is not the case in Protected mode. As described in Chapter 3 in Intel's Architectural Manual(1.2MB PDF), the x86 can handle memory access in two ways: *Segmented* or *Paged*.

With the *Segmented* memory model, instead of linear memory, you divide the memory into small or large segments, like the name tells us =). This is the model we'll be working with in this tutorial. Say we want to write something on to memory address 0xB8000 - the color video memory. Then we could define a segment, that starts at address 0xB8000. Say this is our 08h segment. To access the linear address of *0xB8002*, we write *08h:0002h* instead. Simple as that. The limit is that we can only have 8192 different segments, but that should be enough =). A good thing is that we can choose if the segment should be read-only or read/write and they can overlap each other.

The other memory model called *Paging* is used in favor for virtual memory. If a demand for a non-existing page is requested, the CPU generates an exception and temporary stops the program. Then it's up to the OS (us), to load that page from disk to memory and then the application continues, without even knowing what we've done. Pages are normally in the range of 4 kB.

Practical applications

That was the theoretic part of my tutorial. Our goal for this tutorial is to enter Protected mode. To do this, we must choose a memory model. I chose Segmented memory, because it's the easiest one. That will do (for now at least). *How do we tell the CPU what model to use?* The default model in protected mode is segmented memory. To change to Paging, set bit 31 in the CR0 register. This only works in protected mode, not in real mode.

What we must do, is set up our segments. This is done by creating and loading a GDT, Global Description Table. We should have at least two segments; one code segment and one data segment, overlapping in our case, for simplicity. The structure of the GDT is specified in Intel's manual (see above for link) to:

1st Double word:

Bits	Function	Description
0-15	Limit 0:15	First 16 bits in the segment limiter

16-31	Base 0:15	First 16 bits in the base address

2nd Double word:

Bits	Function	Description
0-7	Base 16:23	Bits 16-23 in the base address
8-12	Type	Segment type and attributes
13-14	Privilege Level	0 = Highest privilege (OS), 3 = Lowest privilege (User applications)
15	Present flag	Set to 1 if segment is present
16-19	Limit 16:19	Bits 16-19 in the segment limiter
20-22	Attributes	Different attributes, depending on the segment type
23	Granularity	Used together with the limiter, to determine the size of the segment
24-31	Base 24:31	The last 24-31 bits in the base address

Now, we want to fill in our GDT. The first segment is always set to 0 and is called the *Null Segment*. This is reserved by Intel. If we try to load the Null Segment, a *General Protection Exception* will occur. We specify the Null Segment, by writing a 64-bits containing 0:

```
gdt:

gdt_null:
   dq 0
```

Now for our code segment. The first 16 bits sets the limit (see table above, 1st Double word). We aim at a limit of 4 GB (0FFFFFh limit total). Next, we set the base address to 0 (start of memory).

```
gdt_code:
   dw 0FFFFh
   dw 0
```

Double word 2 of the segment descriptor is a little more complicated. The first 8 bits continues on the base address, so will still set that to 0.

```
db 0
```

Then there's the 4 Type bits. Bit 8 is an access flag and is set on the first access by the CPU. We don't have any use for this, so leave it 0. The next bit sets if the segment should be readable. Set this bit to avoid any complications later. Bit number 10 is a thing called conforming. If this bit is set, then less privileged code segments is allowed to jump to or call this segment. In an OS, we don't want that, so we clear this bit to 0. The last bit of the segment type specifies if it is a code or data segment. Set this bit, because we're designing the data segment later. For the 4 type bits we add together and get: 1010b (binary). Readable code segment, nonconforming.

Bit 12 in the 2nd Double Word is set if the segment is either a data or a code segment. This is the case, so set this bit. Next up is the privilege level. The two bits can contain a value in the range of 0-3, with 0 meaning the most privileged and 3 the least privileged. Because this segment is a part of our OS, this should be set to 0. After that, there's the present flag. Set this bit, add up and we get: 1001b. Combine this with the above and we get:

```
db 10011010b
```

Last 16 bits left. Bits 0-3 here (16-19 in the 2nd Double Word) is the last bits in the segment limit. Set this to

0Fh. Unfortunatly, the compiler doesn't allow us to specify lesser than 8 bits, so we have to combine this value with the next four bits.

Bit 4 represents a flag of 'Available to System Programmers' and is ignored by the CPU. It means that you can use this bit to a purpose of your choice. I'll just ignore it, because I haven't found a way to use it (yet?). Intel has reserved bit 5 and it should always be 0. Then there's the Size bit and should be set in our case (this tells the CPU we have 32-bit code and not 16-bit). Bit 7 - Granularity... If this bit is set, the limiter multiplies the segment limit by 4 kB. In our case, this is what we want. We wanted a limit of 4 GB (maximum), and the limit we set was 0FFFFFh. Now, if we multiply this by 01000h and add 0FFFh, what do we get? 4 GB - Yeah! We still have 0Fh from the 16-19 bits. The value there was 0Fh, which is 1111b. Put it all together and we get:

```
db 11001111b
```

The only thing left now is the 8 bits remaining on the base address, and we still write 0 here:

```
db 0
```

Puh! That wasn't very nice... Let's hope it works =). Now for the data segment. 1st Double word exactly as the code segment:

```
gdt_data:
   dw 0FFFFh
   dw 0
```

Same for the first 8 bits of the 2nd Double Word

```
db 0
```

The first Type bit (Accessed) is the same as before. Bit 9 is different from the code segment. Instead of enabling read access, we enable write access. I recommend setting this bit, because otherwise, you're not allowed to write to your variables or any memory address. The 10th bit handles the expand direction. We want to expand down, so this bit should be cleard. Bit 11 is same as the code segment, but now we want a data segment, so this should *not* be set. All the bits 12-15 are the same as the code segment, so now we can sum up:

```
db 10010010b
```

The last 16 bits are almost the same as the last 16 bits for a code segment. 'Big' is the name for bit 6. This is related to the segment limit and should be set to allow 4 GB.

```
db 11001111b
db 0
```

There, now we have a Null Segment, a code segment and a data segment. All we need now is to let the CPU find them. This is where the LGDT instruction is used. The instruction takes an address to a GDT descriptor. This tells the CPU where to find the GDT and how big it is. The GDT Descriptor is 48-bits long:

The GDT Descriptor

Bits	Function	Description
0-15	Limit	Size of GDT in bytes
16-47	Address	GDT's memory address

In order to know the size of the gdt, we want it to be calculated at compile time. This can be done in Nasm,

but we need to add a line, just after the data segment code:

```
gdt_end
```

Then all we need to do is to write:

```
gtd_desc:
   db gdt_end - gdt
   dw gdt
```

The actual GDT and the GDR desctiptor, should be placed in between the code and the boot sector identifyer. Look at the source, if you don't understand what I mean. Now, on the the instructions. Before we do anything, we have to make sure that we're the only one executing at the moment. The only thing that could disturb us, is the interrupts. So we disable them (this should be placed right after the ORG command).

```
cli
```

Before we execute the LGDT instruction which loads our new segments' attributes, we need to load the DS-register. The address to our GDT-descriptor is DS:gdt_desc and we don't know what DS is. We can't set the register directly, so we must go through AX.

```
xor ax, ax
mov ds, ax
```

Now, we're all set to execute the LGDT instruction.

```
lgdt [gdt_desc]
```

There... Now we have our GDT and are ready to enter the world of Protected mode. As you can read in Section 2.5 in Intel's 3:rd architectural manual, this is done by setting bit 0 in the CR0 register. There are two ways of doing this, but I'll go for the easy-to-understand version. First we move the contents of the CR0 register into the EAX register. Then we set bit 0 by making an OR operation with EAX and 1. After that, we simply move EAX into CR0 and we're done!

```
mov eax, cr0
or eax, 1
mov cr0, eax
```

After this is done, the instruction pipeline needs to be cleared. It contains garbage instructions for 16-bit mode and now that we're in 32-bit mode, we don't have any use for those. To clear the pipeline, we only need to make a far jump. This is done by jumping to a code segment and an offset. The code segment is the first segment right after the Null segment. Multiply by eight and we have our segment identifyer! We now also enter 32-bit, so we want 32-bit instructions to be compiled.

```
   jmp 08h:clear_pipe

[BITS 32]
clear_pipe:
```

This jumps to our code segment and as far in as the label 'clear_pipe'. Then we need to fill the segment registers with proper segment values. There are six segment registers: *CS, SS, DS, ES, FS* and *GS*. The CS register doesn't have to be touched, because our jump fills it with proper segment values. The SS and DS registers are the most important ones. The first is the Stack segment and the other is the Data segment, where our variables are located. We just load these with our data segment.

```
mov ax, 08h
mov ds, ax
```

```
mov ss, ax
```

Now, we have set up the first part of the stack, namely the segment part. Now to the offset part. This is stored in the ESP register. If you thought that the computer memory was cleared at a start-up, think again. The first MB is filled with different stuff you have to be careful with. Here's a table over the first MB:

Linear address range (hex)	**Memory type**	**Use**
0 - 3FF	RAM	Real mode, IVT (Interrupt Vector Table)
400 - 4FF	RAM	BDA (BIOS data area)
500 - 9FFFF	RAM	*Free memory, 7C00 used for boot sector*
A0000 - BFFFF	Video RAM	Video memory
C0000 - C7FFF	Video ROM	Video BIOS
C8000 - EFFFF	?	BIOS shadow area
F0000 - FFFFF	ROM	System BIOS

With the help of this table, we can see that setting a stack at 090000h can be a good idea for now. It's away from our code and it's large enough for now (0FFFFh). We'll not be needing the stack in this tutorial, but it's always nice to learn to do things right from the beginning.

```
mov esp, 090000h
```

In protected mode, the bios interrupts doesn't work. To be sure we accually come this far in the code, we want to print a character to the screen. As you can see, the Video RAM is located at 0xA0000 - 0xBFFFF. The most common today, is that the frame buffer is located at B8000h (color text mode, applies to CGA, EGA and VGA). If we want to target a monochrome screen, the memory location would be B80000. There are ways to detect this, but I assume that everyone today has a video card capable of color text mode.

There are two bytes for every character. The first byte is the ASCII code for the character. Then there's the attribute color byte. This didn't work very well with Bochs, when the BIOS interrupt was used. Let's give it a try with this method. I choose *Bright cyan* as foreground color and *Blue* as the background color and a non-blinking character. The bits 0-3 are used to specify the foreground color, the bits 4-6 are the background color and the 7th bit specifies blinking. Let's see... 0Bh for Bright cyan... 01h for Blue... 0 for non-blinking... That's 1Bh. I choose print a 'P' as in Protected mode.

```
mov 0B8000, 'P'
mov 0B8001, 1Bh
```

All that is left now, is to make the jump command to itself again...

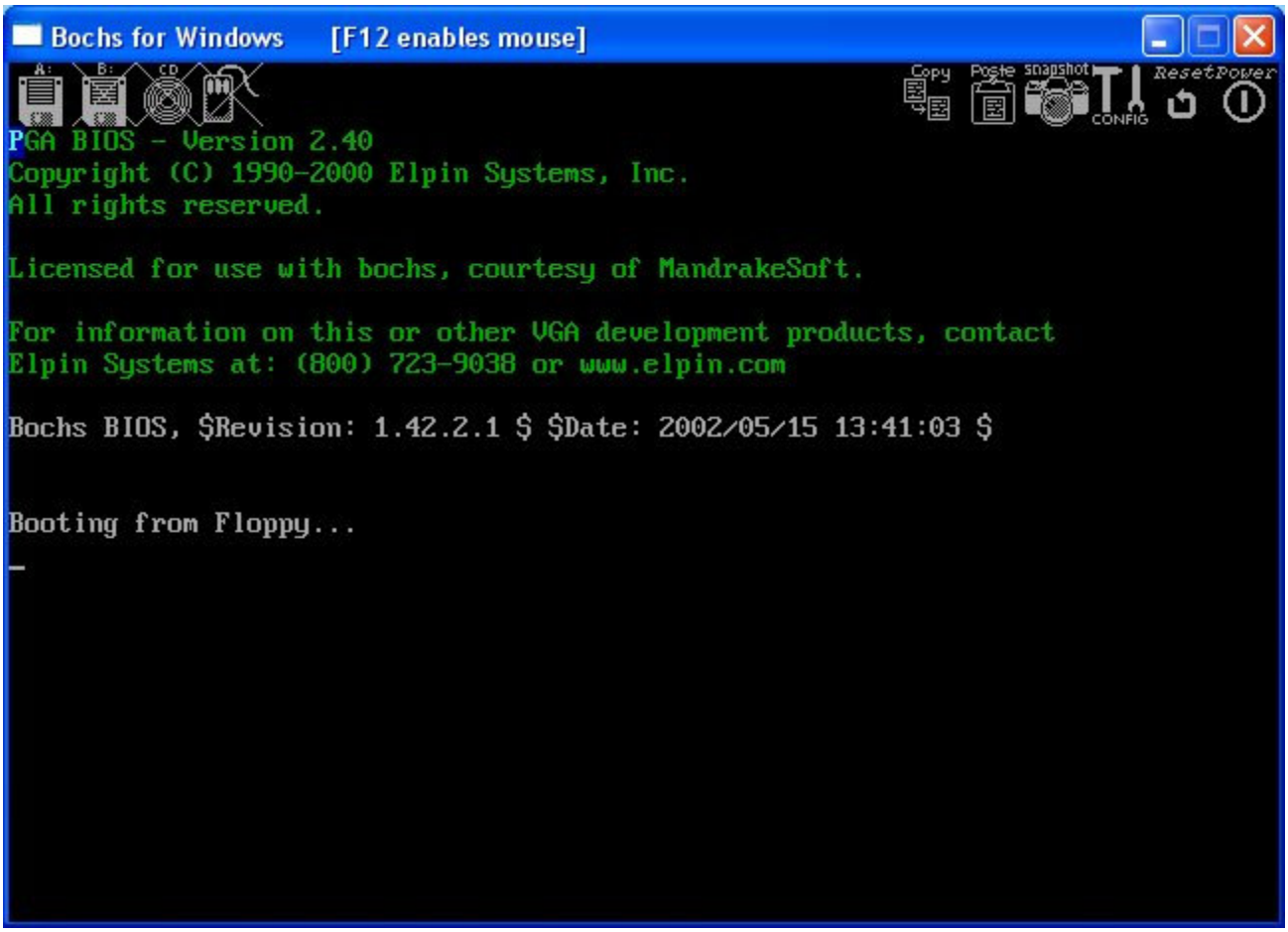
```
hang:
   jmp hang
```

...add the identifying lines...

```
times 510-($-$$) db 0
   dw 55AAh
```

...compile, copy to Bochs' directory and run!

See the colored 'P' in the top left corner?



Download the complete source for this tutorial.
Download my example configuration file for Bochs (paths to the BIOS may have to be changed, if you're using another distributions than Win32 1.4.1).

Any comments, improvments or found errors? Mail me: gregor.brunmar@home.se.

Goto the next tutorial.