ECE291

Lecture 14

Protected mode

Lecture outline

- Real mode review
- Protected mode theory
- Differences
- Protected mode examples

Real Mode Addressing

- Up to 1Mb of addressable memory. Address is always computed by <segment:displacement> where each of the two can be 16-bits.
- Segment register is always extended with a 0H at the right end.
- Thus up to 20 bits of address => 1Mb of memory
- Segments are always 64Kb

16-bit Default Segment + Offset address combinations:

Segment	Offset	Special Purpose
CS	IP	Instruction address
SS	SP or BP	Stack address
DS	BX, DI, SI, an 8- or 16-bit number	Data address
ES	DI for string ops	String destination address

Real Mode

32-bit Default Segment + Offset address combinations:

Segment	Offset	Special Purpose
CS	EIP	Instruction address
SS	ESP or EBP	Stack address
DS	EBX, EDI, ESI, EAX, ECX, EDX, an 8- or 32-bit number	Data address
ES	DI for string ops	String destination address
FS	No default	General address (386+)
GS	No default	General address (386+)

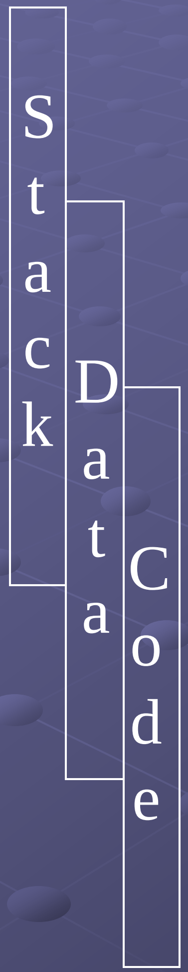
- In 386-Pentium III, never place a number > FFFFH in an offset register when operating in real mode.
- This causes the system to halt and to indicate an addressing error

Real Mode (cont.)

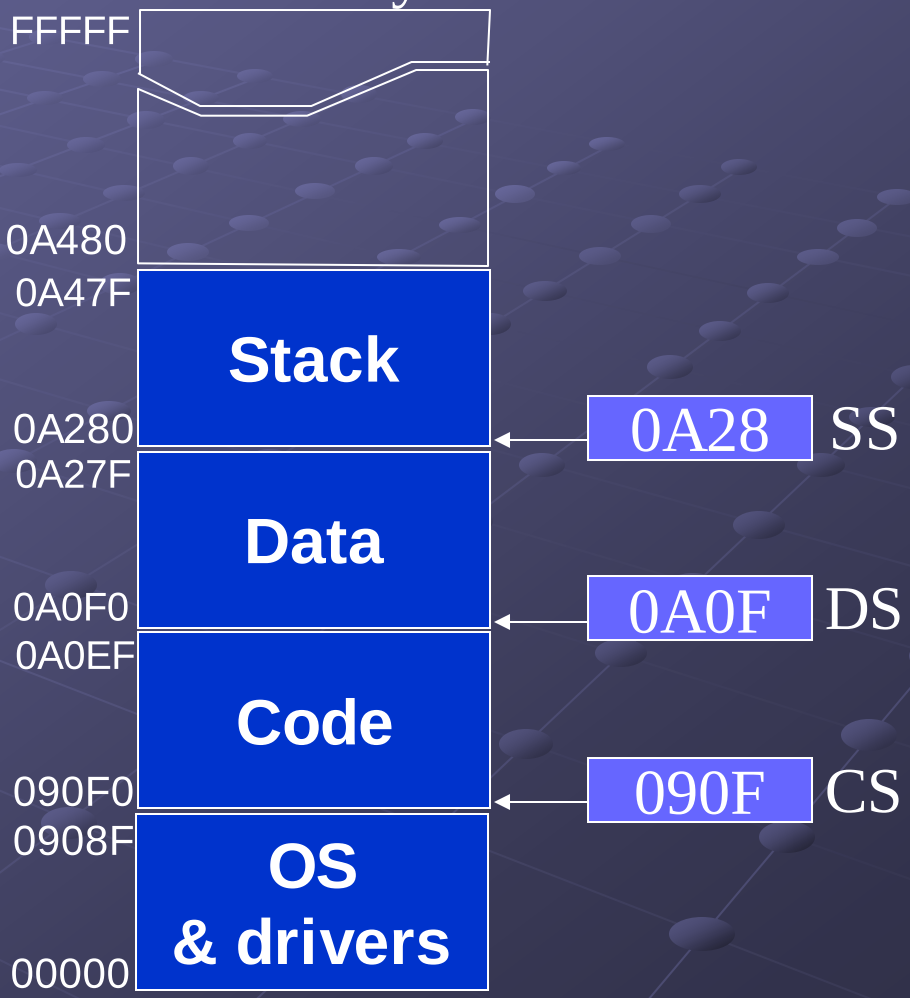
- Up to four 64Kb-segments for < x286
- Up to six segments for >= x386
- Program can use any arbitrary number of segments but only four/six can be addressed simultaneously at any given time
- If a user segment does not use all 64Kb of memory segment registers can be initialized so that segments can overlap - it's your responsibility to assure the overlap does not create unwanted side-effects!
- DOS or any other OS is responsible for linking and loading a user program, figuring out the code-data-stack segments, dynamic data area, and initializing the corresponding segment registers.

Overlapping Segments

Conceptual overlap



Memory



Real Mode (cont.)

- Segment registers allow programs to be written using only offset address and still be relocated anywhere in memory: all we need to do to move a code/data/stack segment is to change the corresponding segment register - all offset addresses remain same.
- Relocation of code and data is very important for:
  - up/downward compatibility
  - write programs without being concerned about the memory size of the particular machine they execute on
  - moving programs around in memory and allowing multiple programs to run simultaneously
- Segment registers are used to address memory in Real Mode only.
- The result is similar to Virtual Memory (in ECE 312)

Protected Mode

- In protected mode (where memory is much larger) we have yet another indirection
- Segment registers no longer point directly to memory - they point to descriptors which then point to the beginning of a segment in memory
- The drawback is a more expensive address translation mechanism
- But the benefit is that we can relocate any segment anywhere in 4Gb space, customize access rights to each segment, share segments with different programs/applications, etc.

- NOTE: Protected mode does NOT require any change in the application either (unless you customize protection rights) since the indirection is handled automatically by the linker/loader.

Protected Mode: General

- In PM, segment registers point to descriptor tables (DT) - which then give us the starting address of the segment
- Each DT contains 8K (8,192) *descriptors* where each descriptor is an 8-byte quantity that describes a memory segment. There are two DTs:
  - Global (or system) descriptor table
  - Local (or application) descriptor table
- Therefore we have up to 16K (2 x 8192) memory segments that can be addressed in PM by each application.
- Each DT resides in memory and takes up a maximum of 64Kb of memory (8B x 8K).

Format of Descriptors in PM

80286 Descriptor

00000000	00000000
Access rights	Base(B23-B16)
Base (B15-B0)	
Limit (L15-L0)	

386-Pentium III Descriptor

Base B31-B24	G	D	0	A V	Limit L19-L16
Access rights	Base(B23-B16)				
Base (B15-B0)					
Limit (L15-L0)					

- Base is the base address of segment in memory
  - in x286 it is 24-bits;
  - in x386+ it can be 32-bits. (Smallest memory granularity is 4Kb so in x386+, least significant 12 bits can be ignored in Base => 20+12=32 bit addresses)
- Limit is the last offset in a segment
  - i.e. variable size segments in PM. In x286 limit is 16-bits bit in x386+ it is 20bits.
- Examples:
  - x286: segment begins at F00000H and ends at F000FFH => it has base F00000H and has a limit of 00FFH (16 bits).
  - x386: same segment would begin at 00F00000H and will have a limit of 000FFH (20 bits)

PM: Descriptor Format

386-Pentium III Descriptor

<table>
<tr><td>Base
B31-B24</td><td>G</td><td>D</td><td>0</td><td>A
V</td><td>Limit
L19-L16</td></tr>
<tr><td>Access rights</td><td colspan="5">Base(B23-B16)</td></tr>
<tr><td colspan="6">Base (B15-B0)</td></tr>
<tr><td colspan="6">Limit (L15-L0)</td></tr>
</table>

G - granularity bit: Specifies the size of segment increments:

- G=0 => Limit specifies a segment limit of 00000H to FFFFFH
- G=1 => Limit specifies a segment limit of 00000XXXH to FFFFFXXXH

(G=1 allows a segment length of 4Kb-4Gb in increments of 4Kb)

Example 1: Starting Address = 10000000h Segment Size = 1FFh
G = 0, Base = 10000000h, Limit = 001FFh
Ending Address = Base+Limit=10000000h + 001FFh = 100001FFh

Example 2: Starting Address = 10000000h Segment Size = 1FF000
G = 1
End = Base + Limit = 10000000h + 001FFXXXh = 101FFFFFh

The extension XXX can take any value from 000 to FFF

PM: Descriptor Format (cont.)

386-Pentium III Descriptor

<table>
<tr><td>Base
B31-B24</td><td>G</td><td>D</td><td>0</td><td>A
V</td><td>Limit
L19-L16</td></tr>
<tr><td colspan="2">Access rights</td><td colspan="4">Base(B23-B16)</td></tr>
<tr><td colspan="6">Base (B15-B0)</td></tr>
<tr><td colspan="6">Limit (L15-L0)</td></tr>
</table>

AV-bit: Specifies whether the segment is available or not.

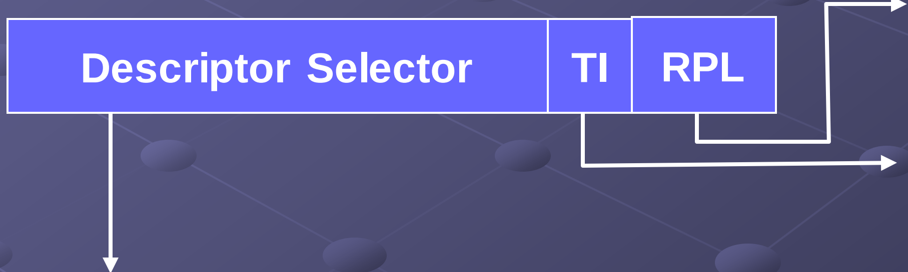
D-bit: Specifies how memory is accessed in RM or PM

- D=0 => Default: 16-bit instructions, offsets and registers
- D=1 => Default: 32-bit instructions, offsets and registers

NOTE: The default of register size and offset size can be overridden in both 16- and 32-bit instruction modes.

Access Rights byte: Specifies access rights, access violation actions, direction of growth (for data segments) - e.g., shared code segments

Segment Register in PM:



RPL=requested privilege level
(00-highest, 11-lowest)

TI=0 => Global descriptor table
TI=1 => Local descriptor table

Selects one of 8,192 descriptors in global or local description tables

DPMI

- The Dos Protected Mode Interface
  - Windows 2000 runs in PMode and encapsulates your programs so they can't crash the computer (theoretically)
  - This encapsulation prevents direct access to some of the processor's protected mode capabilities
  - DPMI is a standard way to access these capabilities outside the box while still having Windows provide some protection

DPMI

- DPMI is accessed via INT 31h
- See the DPMI reference on the References Page for great detail
- For most things you will use our special Protected Mode Library
- One of the ints you'll use directly involves invoking DOS interrupts in PMode. This is function 300h.
- Function 300h is used to call any ISR which accesses memory

How to write PMode Programs

```
; Simple Real Mode Program

STKSEG SEGMENT STACK
RESB 512
STKTOP

SEGMENT CODE
   myvar1 db 45h
   myvar2 resb 1
..start
   mov ax, cs
   mov ds, ax

   ret
```

```
; Simple Protected Mode Program

SECTION .data
   myvar1 db 45h

SECTION .bss
   myvar2 resb 1

SECTION .text
   GLOBAL _main
   _main

   ret
```

Other differences – program structure

- SECTION is essentially the same as SEGMENT
- .data contains initialized variables
- .bss contains uninitialized variables that take up no space in your executable but get allocated and initialized to zero when your program executes
- .text is the same as CODE
- _main must be declared global and serves the same purpose as ..start
- All offsets change from 16 bit to 32 bit

Other differences - addressing

- Memory addressing now uses extended registers EBX, EBP, ESI, EDI instead of the 16 bit equivalents
- You can *also* use ***any*** 32-bit register to address memory (including EAX, ECX, EDX, ESP)
- Add *any* two of these registers
- One may be multiplied by a scaling factor of 1, 2, 4, or 8.
  - mov ax, [disp]
  - mov ax, [eax + disp]
  - mov ax, [eax + 4*ebp + disp]		;defaults to DS
  - mov ax, [ebp + 4*eax + disp]		;defaults to SS

Example MMX Program

```
GLOBAL _main
SECTION .bss        ;=================
                    ; Uninitialized data
SECTION .data       ;=================
  Array_1   db  01h, 02h, 03h, 04h, 05h, 06h, 07h, 08h
            db  01h, 01h, 01h, 01h, 01h, 01h, 01h, 01h
            db  08h, 09h, 0Ah, 0Bh, 0Ch, 0Dh, 0Eh, 0Fh

  Array_2   db  09h, 09h, 09h, 09h, 09h, 09h, 09h, 09h
            db  00h, 01h, 02h, 03h, 04h, 05h, 06h, 07h
            db  80h, 90h, 0A0h, 0B0h, 0C0h, 0D0h, 0E0h, 0F0h

  ClearMMX    dd   00h, 00h
```

Example MMX Program

```
SECTION .text         ;==================
_main
   mov  ebx, Array_1
   mov  edx, Array_2
   mov   ecx, 3

   call  _MMX_Reset     ;Reset MMX registers to zero

 .Add_Array
   movq    mm0, [ebx+8*ecx-8]
   paddb   mm0, [edx+8*ecx-8]
   movq    qword [edx+8*ecx-8], mm0   ;Store the sum
   movq    mm1, [edx+8*ecx-8]
   dec     ecx
   cmp     ecx, 0
   ja      .Add_Array
   ;loop   .Add_Array

movq    mm5, [Array_2]      ;Move sums to MMX
 movq    mm6, [Array_2+8]
 movq    mm7, [Array_2+16]
 emms
ret

_MMX_Reset
   movq  mm0, [ClearMMX]
   movq  mm1, mm0
   movq  mm2, mm0
   movq  mm3, mm0
   movq  mm4, mm0
   movq  mm5, mm0
   movq  mm6, mm0
   movq  mm7, mm0
ret
```