Recap – Real Mode

• Real mode segmentation, how?
  • \texttt{seg * 16 + offset}
  • \([b000:b7ff] \Rightarrow 0xb000 * 16 + 0xb7ff = 0xbb7ff\)

• What is A20?
  • \([f800:8001] \Rightarrow 0x100001?\)
  • \([f800:8001] \Rightarrow 0x1?\)

• FYI, segment registers are:
  • \%cs – code segment
  • \%ds – data segment
  • \%es – extra segment
  • \%fs
  • \%gs
  • \%ss – stack segment
CPU / Registers / Memory

- **Registers, 1clk**
  - eax, ebx, ecx, edx, esi, edi
  - esp, ebp
  - cs, ds, es, fs, gs, ss

- **Cache**
  - L1 (3clk)
  - L2 (7clk)
  - L3 (30clk)

- **MMU**

- **eax** General-purpose registers
- **eip** Hidden register. You cannot access it
- **cs** Segment registers, stores CPL/RPL

200 ~ 300 clk
Recap - JOS Boot Sequence

• 0xf000:0xffff – BIOS

• Loads boot sector – runs 0x7c00

• Enable A20

• Enable protected mode (enabling 4GB memory access)

• Read kernel ELF (Executable Linkable Format)

• ...
JOS Bootloader (boot/main.c)

• After enabling protected mode, boot.S will run ‘ljmpl’ (long jump, far jump) to apply the new segment assigned by the GDT.
• Then, it will call bootmain in boot.c

• Read kernel ELF (Executable Linkable Format)
  • https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
  • Load binary program into memory
  • Read header, map memory, copy data...

• Then, run Kernel!
Need for Protected Mode: No Memory Privilege in Real Mode

- Suppose two program runs at the same time
  - Program A attempts to modify memory used by program B
  - **No SECURITY!**
i386 Protected Mode

• Look at GDT (Global Descriptor Table)
  • Indexed by a segment register
  • (selector)

i386 Protected Mode

- Base
  - Any 32-bit address
- Limit
  - 20-bit, but could be multiplied by 4096 bytes
  - E.g., 1 means 4096, 2 means 8192, etc.

https://wiki.osdev.org/Global_Descriptor_Table
i386 Protected Mode

• Look at GDT (Global Descriptor Table)
  • Indexed by a segment register
  • (selector)

• Retrieve base address
  • \texttt{Address} = \texttt{base} + \texttt{offset}

• Can access \texttt{if} (\texttt{offset} < \texttt{limit}) \texttt{or}
• Can access \texttt{if} (\texttt{offset} < \texttt{limit} \times 4096)
• Depending on the values in flags!

i386 Protected Mode

• Address 0x0008:0x00003400

• In the real mode
  • $0x0008 \times 16 + 0x3400 = 0x3480$

• In the i386 protected mode
  • $GDT[1].base + 0x3400$
    • Access ok if $0x3400$ is less than $GDT[1].limit$
    • Otherwise, raise an exception!

i386 Protected Mode

- **G** - Granularity (0 = byte, 1 = page)
  - 0: Limit will be byte granularity (i.e., limit, only access $2^{20}$, 1MB)
  - 1: Limit will be page granularity (i.e., limit * $4096$, $2^{20} \times 2^{12} = 2^{32}$)

- **D** – Default operand size (0 = 16-bit, 1 = 32-bit)
  - Set the values of IP/SP with respect to this bit

- **R,X** – Readable/Executable

- **DPL** – **Descriptor Privilege Level (a.k.a. Ring Level)**
  - 0 (highest priv), 1, 2, 3 (lowest priv)

For more information: [https://en.wikipedia.org/wiki/Protected_mode](https://en.wikipedia.org/wiki/Protected_mode)
A Segment

Main Memory

Program A

0x80000000

0x40000000

Program B

Size 1MB

Size 2MB

0x10:0 ~ 0x10:0x100000 are valid address for Program A
0x80000000 ~ 0x80100000

0x08:0 ~ 0x08:0x200000 are valid address for Program B
0x40000000 ~ 0x40200000

GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags
--- | --- | --- | ---
16 | 0x80000000 | 0xffff | G=0
8 | 0x40000000 | 0x00200 | G=1
0 | 0x0 | 0x0 | G=0
Protected Mode - Examples

- **0x8:0x8080**
  - Base: 0x40000000
  - Limit (addr): 0x80000000
  - Offset: 0x8080

- 0x8080 < 0x80000000

- Address: 0x40008080

<table>
<thead>
<tr>
<th>GDT index</th>
<th>32-bit Base</th>
<th>20-bit Limit</th>
<th>12-bit Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0x31310000</td>
<td>0x1000</td>
<td>G=0</td>
</tr>
<tr>
<td>8</td>
<td>0x40000000</td>
<td>0x8000</td>
<td>G=1</td>
</tr>
<tr>
<td>0</td>
<td>0x0</td>
<td>0x0</td>
<td>G=0</td>
</tr>
</tbody>
</table>
Protected Mode - Examples

- **0x10:0x333**
  - Base: 0x31310000
  - Limit (addr): 0x1000
  - Offset: 0x333
  - Address: 0x31310333

- **0x10:0x8080**
  - Base: 0x31310000
  - Limit (addr): 0x1000
  - Offset: 0x8080
  - Offset > limit
  - Access denied!
Protected Mode – Memory Privilege

- DPL (Descriptor Privilege Level)

- Protected mode – four levels of memory privilege
  - 0 (00) – highest, OS kernel
  - 1 (01) – OS kernel
  - 2 (10) – highest user-level privilege
  - 3 (11) – user-level privilege

Kernel: for privileged OS operations...

User: for unprivileged applications...
Protected Mode – Memory Privilege

• No memory privilege in real mode

• Protected mode – four levels of memory privilege
  • 0 – highest, OS kernel
  • 1 – OS kernel
  • 2 – highest user-level privilege
  • 3 – user-level privilege

• Typically, 0 is for kernel, 3 is for user...

Descriptor Privilege Level Defines Ring Level

• CPL = Current Privilege Level
  • Defined in the last 2 bits of the %cs register
  • You can change %cs only via lcall/ljmp/trap/int

<table>
<thead>
<tr>
<th>GDT index</th>
<th>32-bit Base</th>
<th>20-bit Limit</th>
<th>12-bit Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 USER</td>
<td>0x31310000</td>
<td>0x1000</td>
<td>G=0, DPL=3</td>
</tr>
<tr>
<td>8 KERNEL</td>
<td>0x40000000</td>
<td>0x80000</td>
<td>G=1, DPL=0</td>
</tr>
<tr>
<td>0 KERNEL</td>
<td>0x00000000</td>
<td>0xffffffff</td>
<td>G=1, DPL=0</td>
</tr>
</tbody>
</table>
Descriptor Privilege Level Defines Ring Level

• CPL = Current Privilege Level
  • Defined in the last 2 bits of the %cs register
  • You can change %cs only via lcall/ljmp/trap/int

• Examples
  • %cs == 0x8  == 1000 in binary, last 2 bits are ZERO -> KERNEL!
  • %cs == 0x13 == 10011 in binary, last 2 bits are 3 -> USER!
  • %cs == 0x10 == 10000 in binary, last 2 bits are 0 -> KERNEL!
  • %cs == 0xb  == 1011....

<table>
<thead>
<tr>
<th>GDT index</th>
<th>32-bit Base</th>
<th>20-bit Limit</th>
<th>12-bit Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>USER</td>
<td>0x31310000</td>
<td>0x1000</td>
</tr>
<tr>
<td>8</td>
<td>KERNEL</td>
<td>0x40000000</td>
<td>0x80000</td>
</tr>
<tr>
<td>0</td>
<td>KERNEL</td>
<td>0x0</td>
<td>0xffffffff</td>
</tr>
</tbody>
</table>
Descriptor Privilege Level Defines Ring Level

- CPL = Current Privilege Level
  - Defined in the last 2 bits of the %cs register
  - You can change %cs only via lcall/ljmp/trap/int

- mov %ax, %cs ← impossible!

- Can only move down...
  - CPL==0, then ljmp 0x3:0x1234 is OK to execute
  - CPL==3, then ljmp 0x0:0x1234 is not allowed

<table>
<thead>
<tr>
<th>GDT index</th>
<th>32-bit Base</th>
<th>20-bit Limit</th>
<th>12-bit Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 USER</td>
<td>0x31310000</td>
<td>0x1000</td>
<td>G=0, DPL=3</td>
</tr>
<tr>
<td>8 KERNEL</td>
<td>0x40000000</td>
<td>0x80000</td>
<td>G=1, DPL=0</td>
</tr>
<tr>
<td>0 KERNEL</td>
<td>0x0</td>
<td>0xffff</td>
<td>G=1, DPL=0</td>
</tr>
</tbody>
</table>
OK, Kernel (Ring 0) can execute code in (Ring 3) via ljmp 0x3:0x1234

• Then, how can we go back to kernel?

• We can switch from ring 0 to ring 3 via ljmp
  • ljmp 0x3:0x1234

• We cannot switch from ring 3 to ring 0 via ljmp
  • ljmp 0x0:0x1234 ← illegal instruction

• We use iret / sysexit / sysret to switch from ring 3 to ring 0
  • We will learn this in week 4
Enabling Protected Mode (part 1): Create Global Descriptor Table (GDT)

- In boot/boot.S
  - `%cs` to point 0 ~ 0xffffffff in DPL 0
  - `%ds` to point 0 ~ 0xffffffff in DPL 0
- Only kernel can access those two segment

```asm
# Bootstrap GDT
.p2align 2

# force 4
gdt:
    SEG_NULL      # null seg
    SEG(STA_X|STA_R, 0x0, 0xffffffff) # code seg
    SEG(STA_W, 0x0, 0xffffffff)      # data seg

.set PROT_MODE_CSEG, 0x8          # kernel code segment selector
.set PROT_MODE_DSEG, 0x10         # kernel data segment selector
```

<table>
<thead>
<tr>
<th>GDT index</th>
<th>32-bit Base</th>
<th>20-bit Limit</th>
<th>12-bit Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0x0</td>
<td>0xffffffff</td>
<td>G=1,W DPL=0</td>
</tr>
<tr>
<td>8</td>
<td>0x0</td>
<td>0xffffffff</td>
<td>G=1, XR DPL=0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Base 0:15</th>
<th>Limit 0:15</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Enabling Protected Mode (part 2): Change CR0 (Control Register 0)

Set PE (Protected enabled) to 1 will enable Protected Mode

In JOS:

```
lgdt gdt_desc
movl %cr0, %eax
orl $CR0_PE_ON, %eax
movl %eax, %cr0
```

1. Load GDT
2. Read CR0, store it to eax
3. Set PE_ON (1) on eax
4. Put eax back to CR0
   (PE_ON to CR0!!)
How to Change CPL?

- `ljmp` (instruction)
  - Long jump

```
# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
```

0x8 == 1000, Last 2 bits are zero..

```
.set PROT_MODE_CSEG, 0x8    # kernel code segment selector
.set PROT_MODE_DSEG, 0x10   # kernel data segment selector
# Bootstrap GDT
.p2align 2
# force 4

gdt:
  SEG_NULL               # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff) # code seg
  SEG(STA_W, 0x0, 0xffffffff)     # data seg
```
Protected Mode Summary

- **Segment access via GDT**
  - Base + Offset < Limit * 4096 (if G == 1)
  - Base + Offset < Limit (if G == 0)

- **Last two bits in %cs - CPL**
  - Memory Privilege - Ring level
  - 0 for OS kernel
  - 3 for user application

- **Changing CR0 to enable protected mode**
  - CR0_PE_ON == 1, set via eax

- **Changing CPL?**
  - ljmp %cs:xxxxx, set the last 2 bits of %cs as 0 for kernel, 3 for user
Virtual Memory

• Three goals

  • Transparency

  • Efficiency

  • Protection
Uniprogramming Environment

- Run one program
- The program can use memory space freely...

Diagram:

- Stack - 1
- Program Data - 1
- Program Code - 1

Legend:
- OS
  - 0x00000 ~ 0x10000 (0 ~ 64KB)
- Free (576 KB)
  - 0x10000 ~ 0xa0000 (64KB ~ 640KB)
Uniprogramming Environment

- Run one program

- The program can use memory space freely...
Uniprogramming Environment

• Run one program

• The program can use memory space freely...
# Multi-programming Environment

- Run two programs

<table>
<thead>
<tr>
<th>Stack - 2 (64KB)</th>
<th>Program Data - 2 (64 KB)</th>
<th>Program Code - 2 (128KB)</th>
</tr>
</thead>
</table>

### Memory Regions

- **OS**
  - 0x00000 ~ 0x10000 (0 ~ 64KB)

- **Program Code - 1**
  - 0x10000 ~ 0x20000 (64KB ~ 192KB)

- **Program Data - 1**
  - 0x20000 ~ 0x30000 (192KB ~ 256KB)

- **Free (64 KB)**
  - 0x30000 ~ 0x40000 (256KB ~ 320KB)

- **Free (192 KB)**
  - 0x40000 ~ 0x50000 (320KB ~ 512KB)

- **Stack - 1**
  - 0x50000 ~ 0x60000 (512KB ~ 640KB)

- **Free (64 KB)**
  - 0x60000 ~ 0x70000 (640KB ~ 768KB)

- **Program Code - 2**
  - 0x70000 ~ 0x80000 (768KB ~ 960KB)

- **Free (192 KB)**
  - 0x80000 ~ 0x90000 (960KB ~ 1024KB)

- **Program Data - 1**
  - 0x90000 ~ 0xa0000 (1024KB ~ 1152KB)

- **Stack - 2**
  - 0xa0000 ~ 0xb0000 (1152KB ~ 1280KB)

- **Free (128 KB)**
  - 0xb0000 ~ 0xc0000 (1280KB ~ 1344KB)

- **Free (576 KB)**
  - 0xc0000 ~ 0xd0000 (1344KB ~ 1456KB)

- **Free (128 KB)**
  - 0xd0000 ~ 0xe0000 (1456KB ~ 1536KB)

- **Free (320 KB)**
  - 0xe0000 ~ 0xf0000 (1536KB ~ 1664KB)
Multi-programming Environment

- Run two programs
- System’s memory usage determines allocation
- Program need to be aware of the environment
  - Where does system loads my code?
  - You can’t determine... system does..

No Transparency...

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000 ~ 0x010000</td>
<td>Program Code - 1</td>
<td>128KB</td>
</tr>
<tr>
<td>0x010000 ~ 0x020000</td>
<td>Program Data - 1</td>
<td>64KB</td>
</tr>
<tr>
<td>0x020000 ~ 0x040000</td>
<td>Stack - 1</td>
<td>64KB</td>
</tr>
<tr>
<td>0x040000 ~ 0x050000</td>
<td>Program Code - 2</td>
<td>128KB</td>
</tr>
<tr>
<td>0x050000 ~ 0x060000</td>
<td>Program Data - 2</td>
<td>64KB</td>
</tr>
<tr>
<td>0x060000 ~ 0x080000</td>
<td>Stack - 2</td>
<td>64KB</td>
</tr>
<tr>
<td>0x080000 ~ 0x0a0000</td>
<td>Program Code - 1</td>
<td>128KB</td>
</tr>
<tr>
<td>0x0a0000 ~ 0x0c0000</td>
<td>Program Data - 2</td>
<td>64KB</td>
</tr>
<tr>
<td>0x0c0000 ~ 0x0e0000</td>
<td>Stack - 2</td>
<td>64KB</td>
</tr>
<tr>
<td>0x0e0000 ~ 0x100000</td>
<td>OS</td>
<td>64KB</td>
</tr>
</tbody>
</table>

Note: The memory locations and sizes are hypothetical and for demonstration purposes. Actual memory allocation can vary.
Multi-programming Environment

- Run two programs

Stack - 2 (64KB)

Program Data - 2 (64 KB)

Program Code - 2 (160KB)
Multi-programming Environment

• Run two programs
  • Program size: 64KB + 64KB + 160K = 288KB

• Free mem
  • 64 + 96 + 128 = 288KB

• Cannot run Program – 2
  • Can’t fit…

Not efficient.. Suffers memory fragmentation problem..
Multi-programming Environment

• Run two programs

• What if Program-2’s stack underflows?

• What if Program-2’s data overflows?

• Without virtual memory
  • Programs can affect to the other’s execution

Virtual Memory

• Three goals
  • Transparency: does not need to know system’s internal state
    • Program A is loaded at 0x8048000. Can Program B be loaded at 0x8048000?
  
  • Efficiency: do not waste memory; manage memory fragmentation
    • Can Program B (288KB) be loaded if 288 KB of memory is free, regardless of its allocation?

  • Protection: isolate program’s execution environment
    • Can we prevent an overflow from Program A from overwriting Program B’s data?
Paging

• A method of implementing virtual memory

• Split memory into multiple 4,096 byte blocks (12-bit)
  • Last 3 digits of page address are ZERO (in hexadecimal)
  • E.g., 0x0, 0x1000, 0x2000, ..., 0x8048000, 0x804a000, ..., 0x7fffe000, etc.

• Having an indirect map between virtual page and physical page
  • Set an arbitrary virtual address for a page, e.g., 0x81815000
  • Set a physical address to that page as a map, e.g., 0x32000
  • 0x81815000 ~ 0x81815fff will be translated into
  • 0x32000 ~ 0x32fff
Virtual Memory - Paging

- Having an indirect table that maps virt-addr to phys-addr

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048000</td>
<td>0x10000</td>
</tr>
<tr>
<td>0x8049000</td>
<td>0x11000</td>
</tr>
<tr>
<td>0x804a000</td>
<td>0x14000</td>
</tr>
<tr>
<td>0xbffdf000</td>
<td>0x12000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Paging: Virtual Memory

- Having an indirect table that maps virt-addr to phys-addr

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048000</td>
<td>0x10000</td>
</tr>
<tr>
<td>0x8049000</td>
<td>0x11000</td>
</tr>
<tr>
<td>0x804a000</td>
<td>0x14000</td>
</tr>
<tr>
<td>0xbffdf000</td>
<td>0x12000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual-2</th>
<th>Physical-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048000</td>
<td>0x13000</td>
</tr>
<tr>
<td>0x8049000</td>
<td>0x15000</td>
</tr>
<tr>
<td>0x804a000</td>
<td>0x16000</td>
</tr>
<tr>
<td>0xbffdf000</td>
<td>0x17000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Transparency: does not need to know system’s internal state
Program A is loaded at 0x8048000.
Can Program B be loaded at 0x8048000?

- Having an indirect table that maps virt-addr to phys-addr

<table>
<thead>
<tr>
<th>Stack-2 0xbffdf000</th>
<th>Stack 0xbffdf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program code-2 0x804a000</td>
<td>Program code 0x804a000</td>
</tr>
<tr>
<td>Program code-2 0x8049000</td>
<td>Program code 0x8049000</td>
</tr>
<tr>
<td>Program code-2 0x8048000</td>
<td>Program code 0x8048000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048000</td>
<td>0x10000</td>
</tr>
<tr>
<td>0x8049000</td>
<td>0x11000</td>
</tr>
<tr>
<td>0x804a000</td>
<td>0x14000</td>
</tr>
<tr>
<td>0xbffdf000</td>
<td>0x12000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual-2</th>
<th>Physical-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048000</td>
<td>0x13000</td>
</tr>
<tr>
<td>0x8049000</td>
<td>0x15000</td>
</tr>
<tr>
<td>0x804a000</td>
<td>0x16000</td>
</tr>
<tr>
<td>0xbffdf000</td>
<td>0x17000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Physical Memory

Stack 0x17000
Program code-2 0x16000
Program code-2 0x15000
Program code 0x14000
Program code-2 0x13000
Stack 0x12000
Program code 0x11000
Program code 0x10000
**Efficiency:** do not waste memory
Can Program B (288KB) be loaded if only 288 KB of memory is free, regardless of its allocation?

- Having an indirect table that maps virt-addr to phys-addr

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048000</td>
<td>0x10000</td>
</tr>
<tr>
<td>0x8049000</td>
<td>0x11000</td>
</tr>
<tr>
<td>0x804a000</td>
<td>0x14000</td>
</tr>
<tr>
<td>0xbffdf000</td>
<td>0x12000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual-2</th>
<th>Physical-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048000</td>
<td>0x13000</td>
</tr>
<tr>
<td>0x8049000</td>
<td>0x15000</td>
</tr>
<tr>
<td>0x804a000</td>
<td>0x16000</td>
</tr>
<tr>
<td>0xbffdf000</td>
<td>0x17000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Physical Memory**

- Stack 0x17000
- Program code 2 0x16000
- Program code 2 0x15000
- Program code 0x14000
- Program code 2 0x13000
- Stack 0x12000
**Protection**: isolate program’s execution environment

Can we prevent an **overflow from Program A** from overwriting **Program B’s data**?
Protected-Mode Address Translation

CPU → Selector → Offset → Logical Address

Segment Translation → Linear Address → Page Translation

Physical Address → x GB

Logical Address

Selector 32 Offset

16

10 10 12 Linear Address

Dir Table Offset

20 12 Physical Address

PPN Offset

1023

GDT/LDT

8 16 Base Limit Flags

16

CR3

Page Directory