# CS444/544
# Operating Systems II
## Virtual Memory

Yeongjin Jang

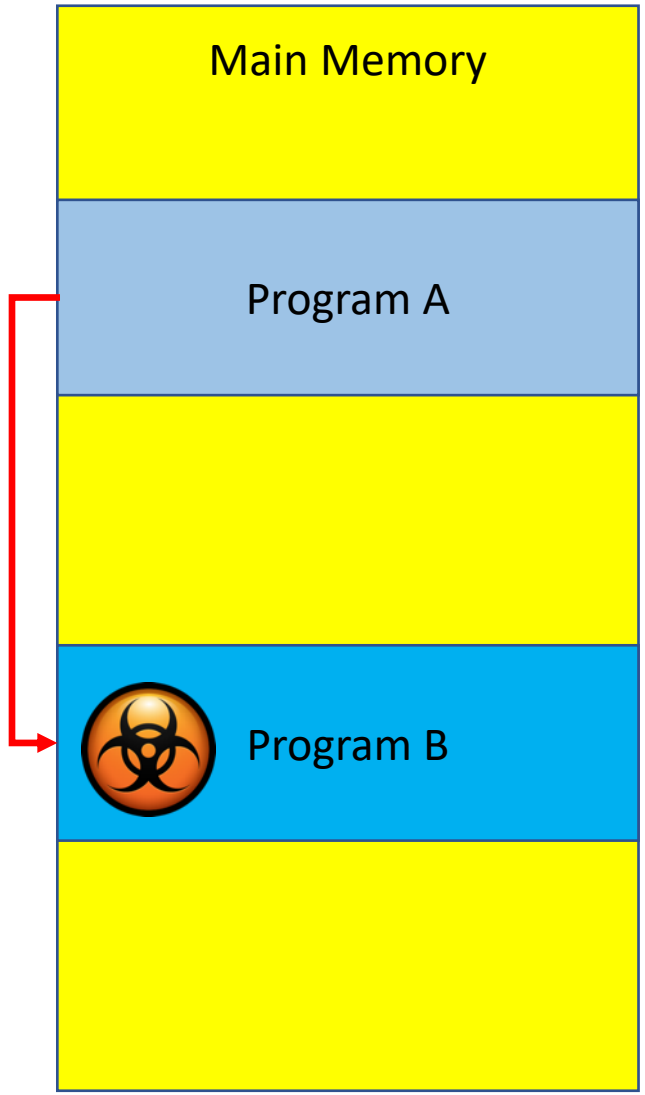
Oregon State University

# Recap - JOS Boot Sequence

- 0xf000:0xfff0 – BIOS
  - 0xf000 * 16 + 0xfff0 = 0xffff0 (REAL MODE)
- Loads boot sector – runs 0x7c00
- Enable A20
- Enable protected mode (enabling 4GB memory access)
- Read kernel ELF (Executable Linkable Format)
- …

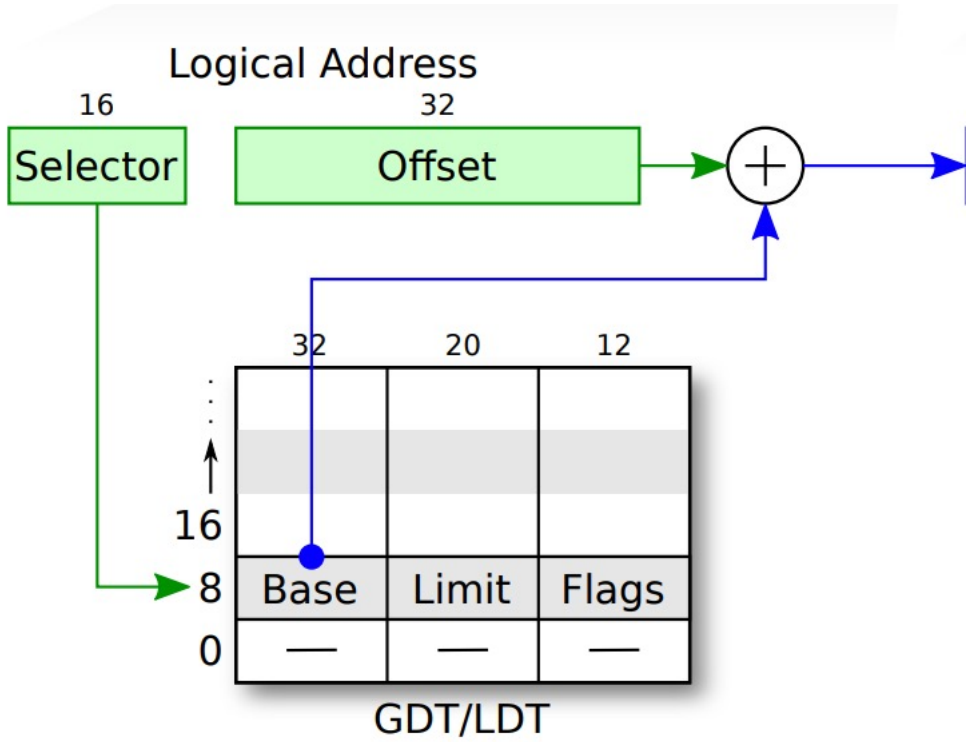
# Need for Protected Mode: No Memory Privilege in Real Mode

- 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



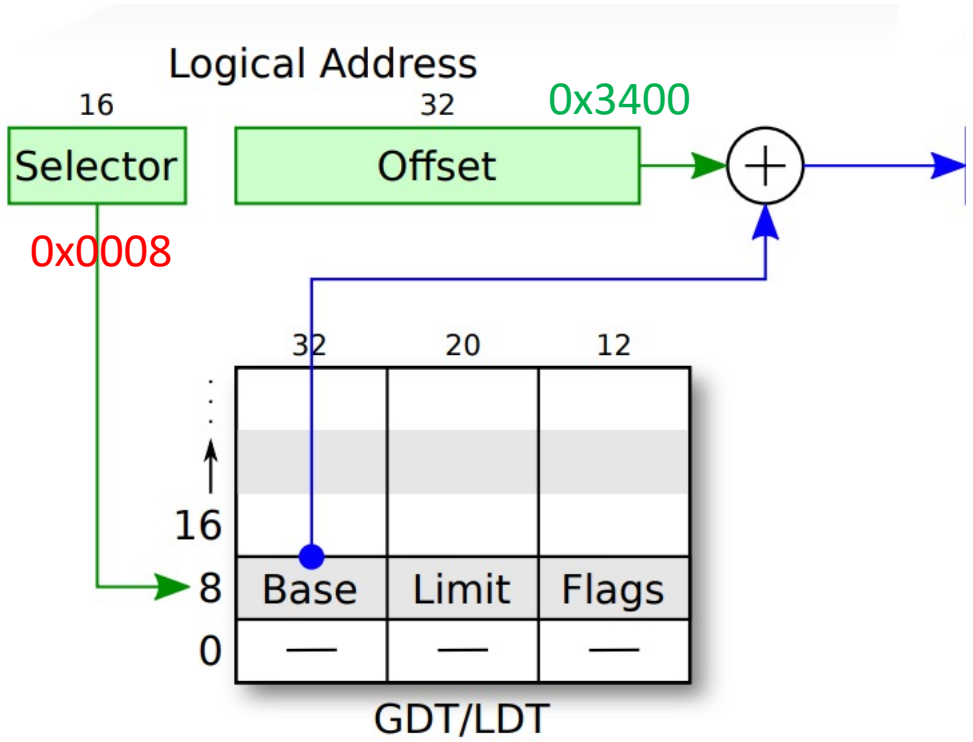

Images from: https://sipb.mit.edu/iap/6.828/files/x86_translation_and_registers.pdf

# i386 Protected Mode

- Address 0x0008:0x00003400

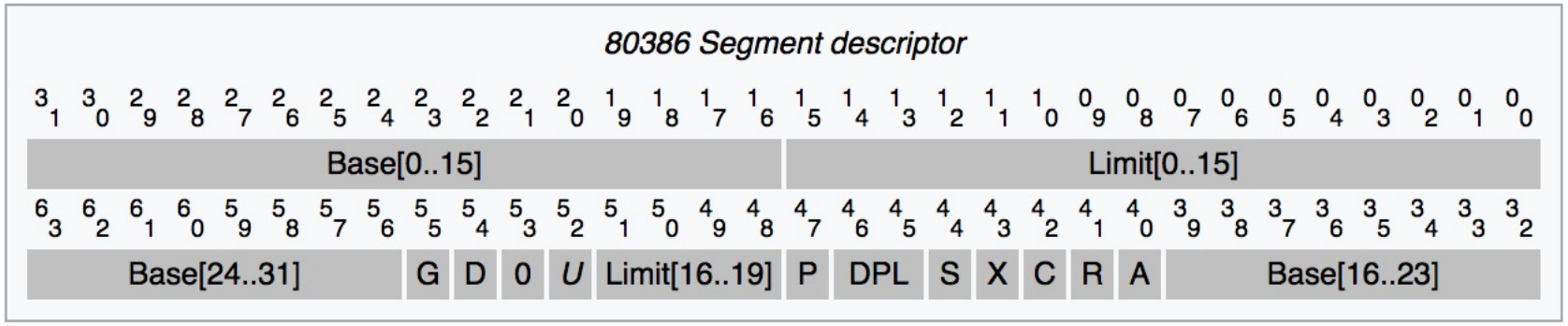
- In the real mode
  - **0x0008 * 16 + 0x3400 = 0x3480**

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




Images from: https://sipb.mit.edu/iap/6.828/files/x86_translation_and_registers.pdf

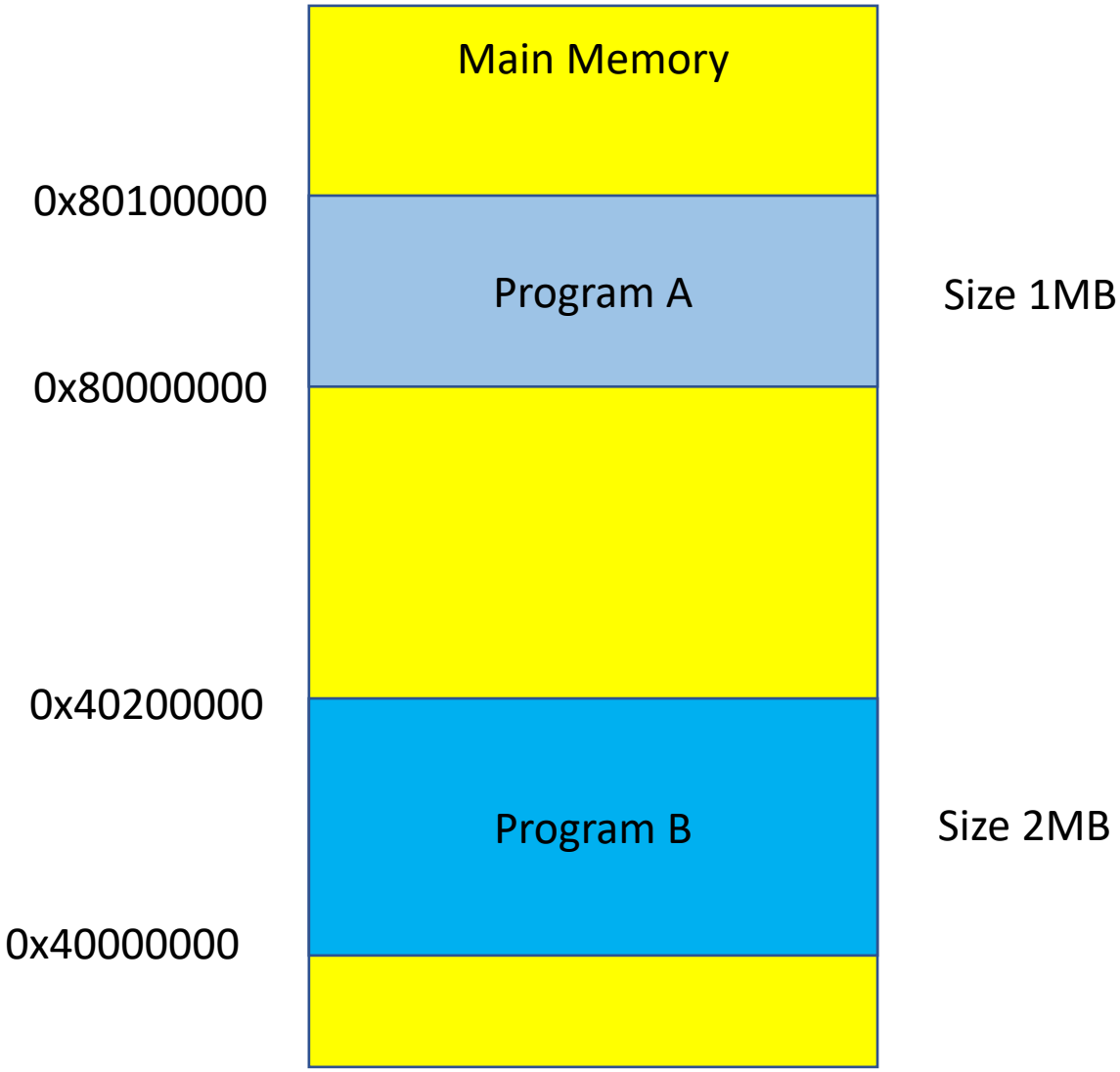
# i386 Protected Mode




- 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, $2^{20} * 2^{16} = 2^{32}$)**
- 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

# A Segment



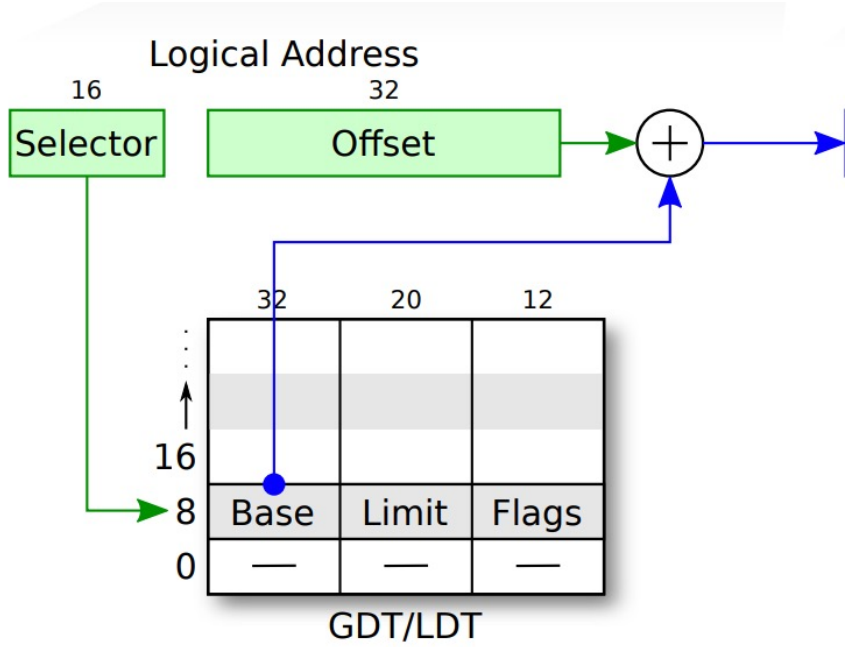

0x10:0 ~ 0x10:0x100000 are valid address for Program A
0x80000000 ~ 0x80100000

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 | 0xfffff | G=0 |
| 8 | 0x40000000 | 0x00200 | G=1 |
| 0 | 0x0 | 0x0 | G=0 |

# Protected Mode - Examples

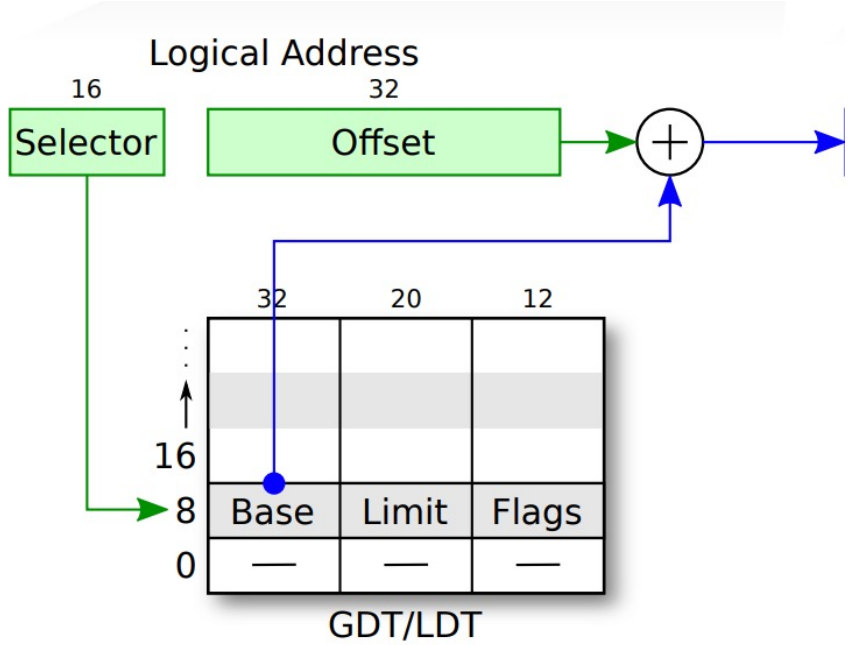
- 0x8:0x8080
  - Base: 0x40000000
  - Limit (addr): 0x8000000
  - Offset: 0x8080
  - 0x8080 < 0x8000000
  - Address: 0x40008080




| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 | 0x31310000 | 0x1000 | G=0 |
| 8 | 0x40000000 | 0x8000 | G=1 |
| 0 | 0x0 | 0x0 | G=0 |

# 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!**




| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 | 0x31310000 | 0x1000 | G=0 |
| 8 | 0x40000000 | 0x80000 | G=1 |
| 0 | 0x0 | 0x0 | G=0 |

# Protected Mode – Memory Privilege

- DPL (Descriptor Privilege Level)

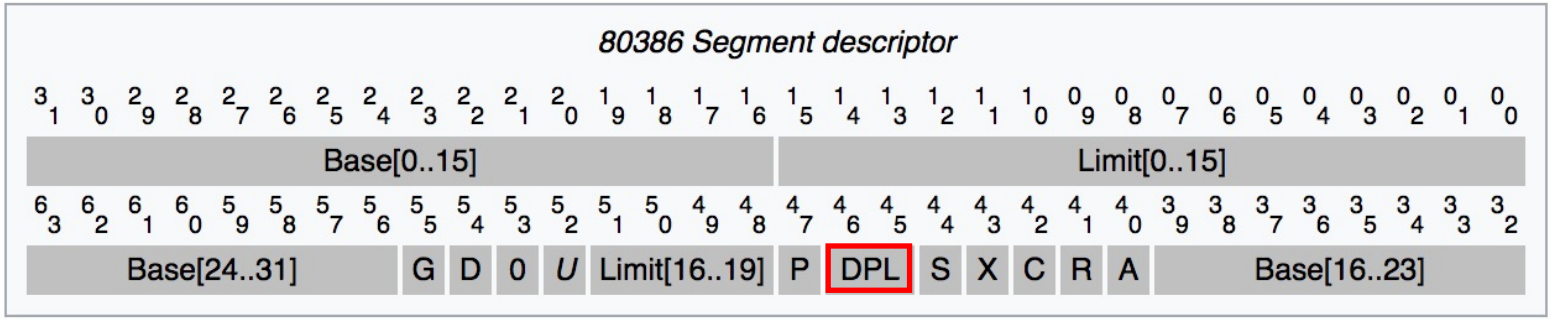
- Protected mode – four levels of memory privilege
  - 0 (00) – highest, OS kernel
  - 1 (01) – OS kernel

  ---

  - 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…

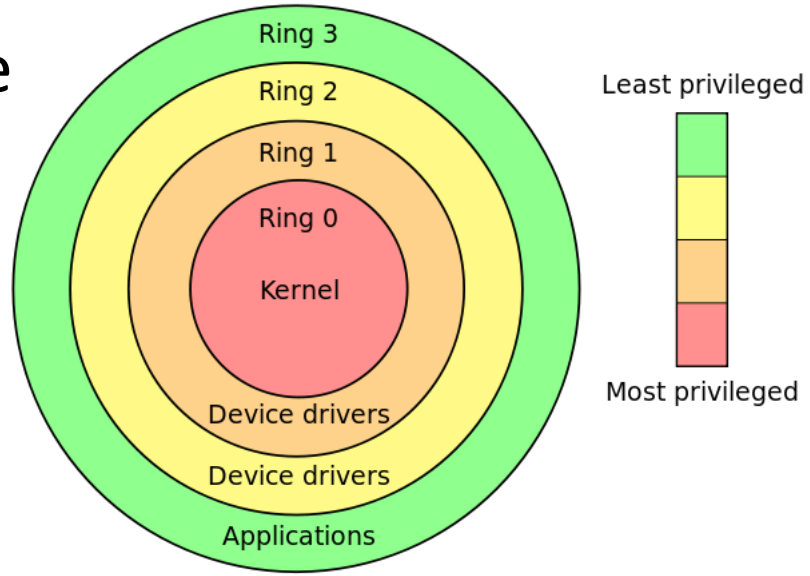



Image from: https://en.wikipedia.org/wiki/File:Priv_rings.svg

# Descriptor Privilege Level Defines Ring Level

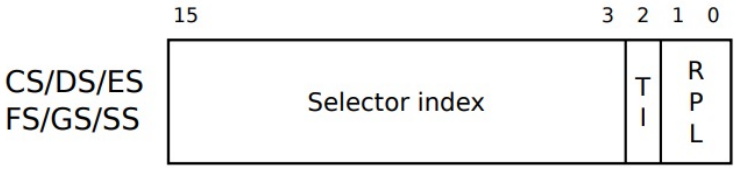
- CPL = Current Privilege Level
  - Defined in the last 2 bits of the %cs register
  - You can change %cs only via `lcall/ljmp/trap/int`

| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 **USER** | 0x31310000 | 0x1000 | G=0, **DPL=3** |
| 8 **KERNEL** | 0x40000000 | 0x80000 | G=1, **DPL=0** |
| 0 **KERNEL** | 0x0 | 0xfffff | G=1, **DPL=0** |

# 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….




| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 **USER** | 0x31310000 | 0x1000 | G=0, **DPL=3** |
| 8 **KERNEL** | 0x40000000 | 0x80000 | G=1, **DPL=0** |
| 0 **KERNEL** | 0x0 | 0xfffff | G=1, **DPL=0** |

# 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**

| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 **USER** | 0x31310000 | 0x1000 | G=0, **DPL=3** |
| 8 **KERNEL** | 0x40000000 | 0x80000 | G=1, **DPL=0** |
| 0 **KERNEL** | 0x0 | 0xfffff | G=1, **DPL=0** |

# 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)

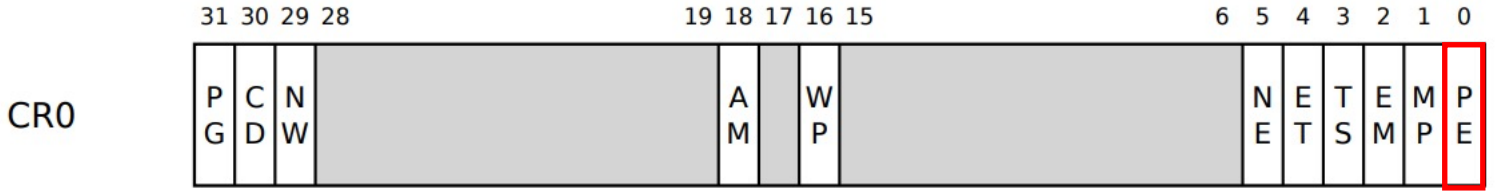
- In boot/boot.S
  - %cs to point 0 ~ 0xffffffff in DPL 0
  - %ds to point 0 ~ 0xffffffff in DPL 0
- Only kernel can access those two segment

```
# 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
```

| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 | 0x0 | 0xfffff | G=1,W **DPL=0** |
| 8 | 0x0 | 0xfffff | G=1, XR **DPL=0** |
| 0 | 0 | 0 | 0 |

# Enabling Protected Mode (part 2): Change CR0 (Control Register 0)




| | | | | | |
|---|---|---|---|---|---|
| PE | Protection enabled | ET | Extension type | NW | Not write-through |
| MP | Monitor coprocessor | NE | Numeric error | CD | Cache disable |
| EM | Emulation | WP | Write protect | PG | Paging |
| TS | Task switched | AM | Alignment mask | | |

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

In JOS:

```
lgdt    gdtdesc
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

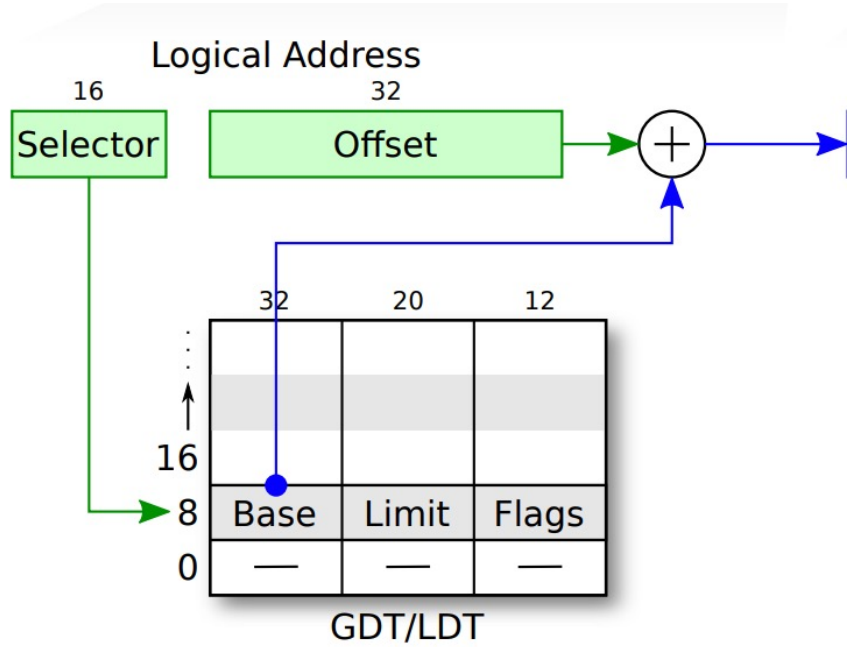
```
# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
```

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 if Offset < Limit * 4096 (if G == 1)
  - Base + Offset if 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

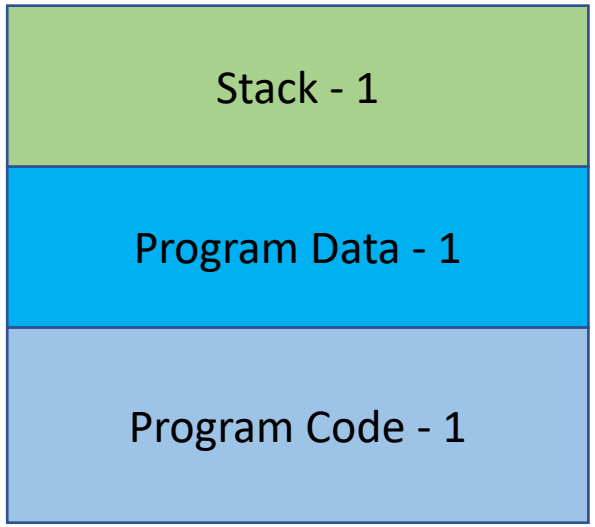


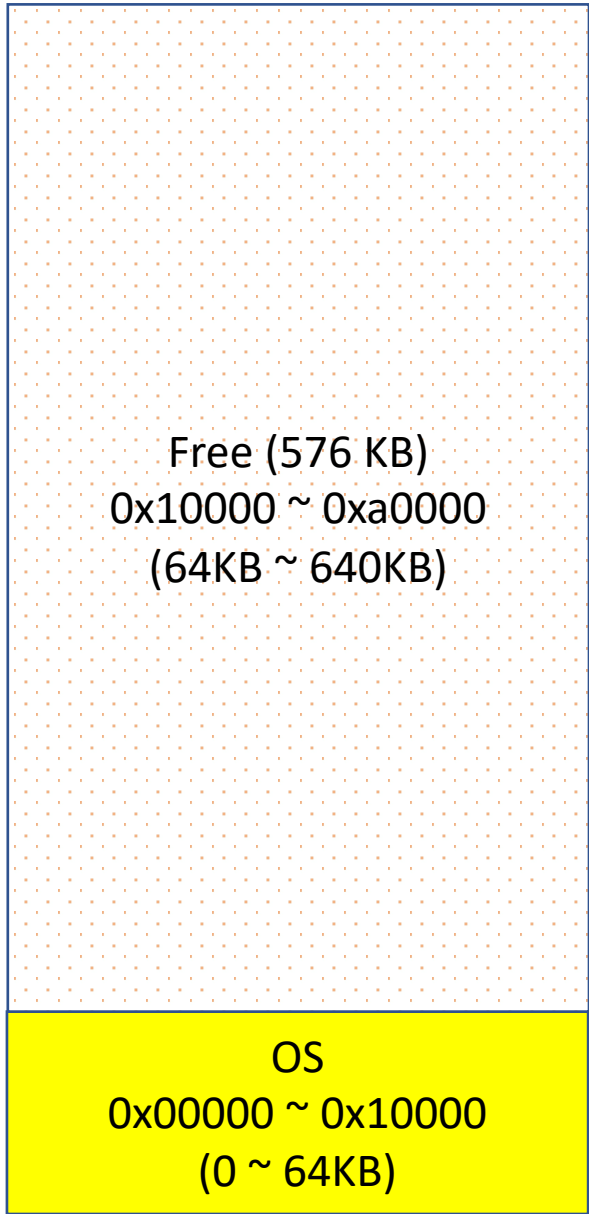
# Virtual Memory

- Three goals
  - Transparency
  - Efficiency
  - Protection

# Uniprogramming Environment

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

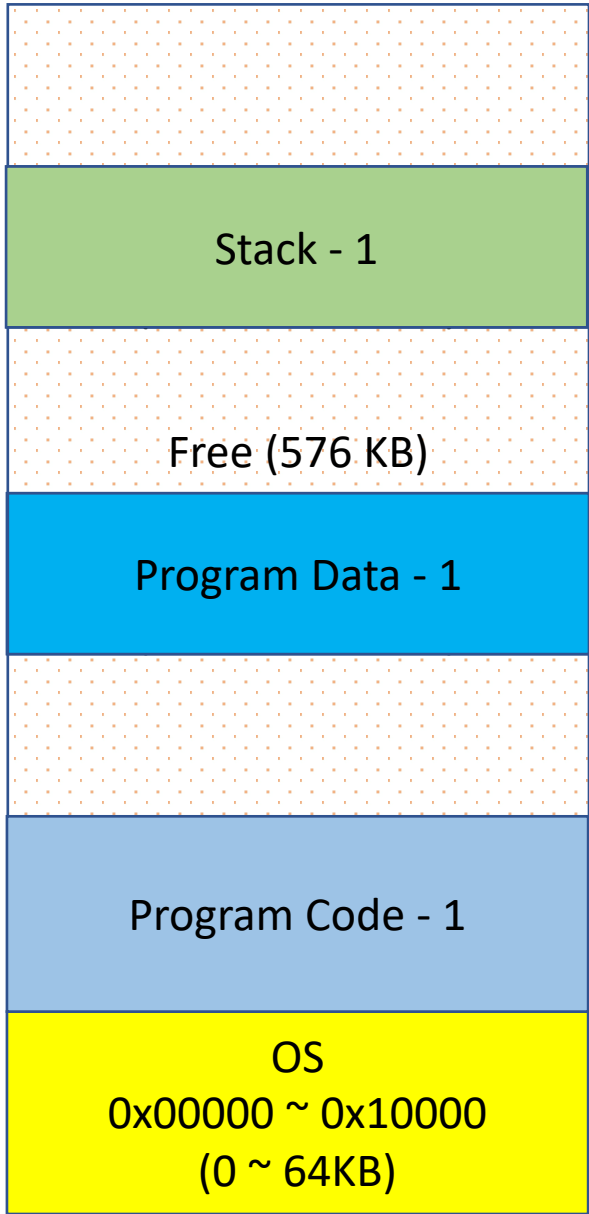


# Uniprogramming Environment

- Run one program

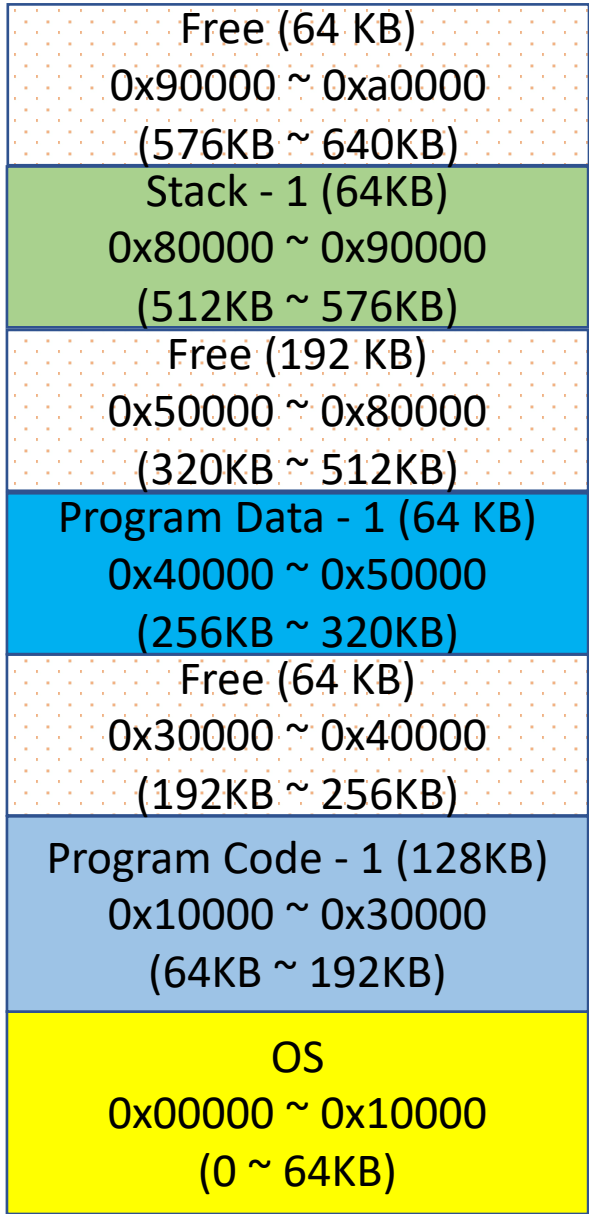
- The program can use memory space freely…



# Uniprogramming Environment

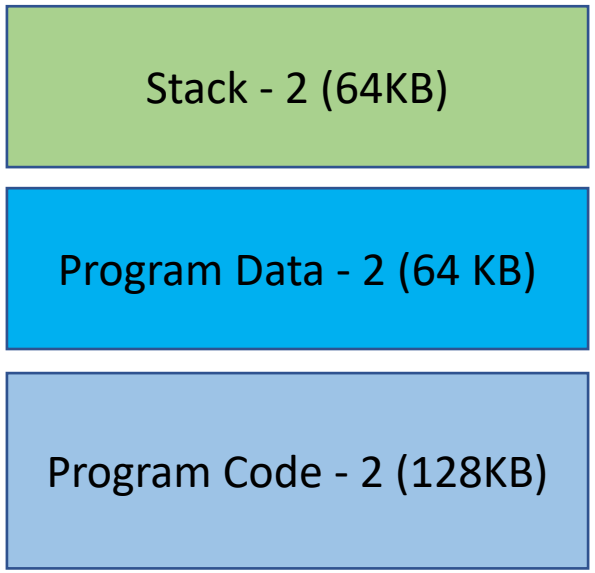
- Run one program

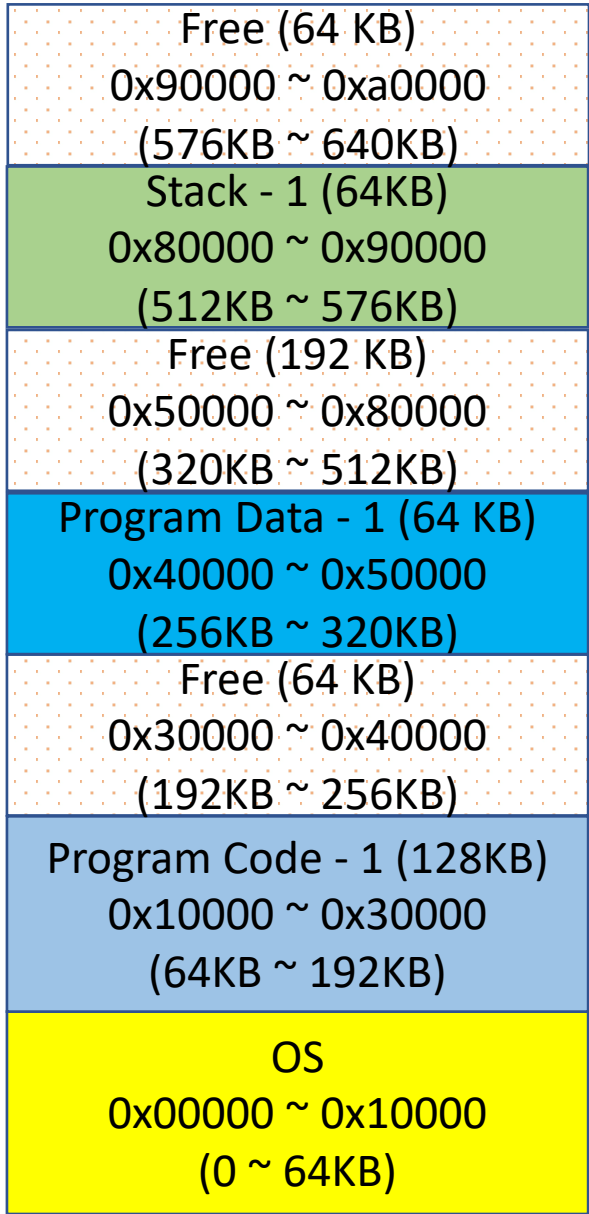
- The program can use memory space freely…



# Multi-programming Environment

- Run two programs

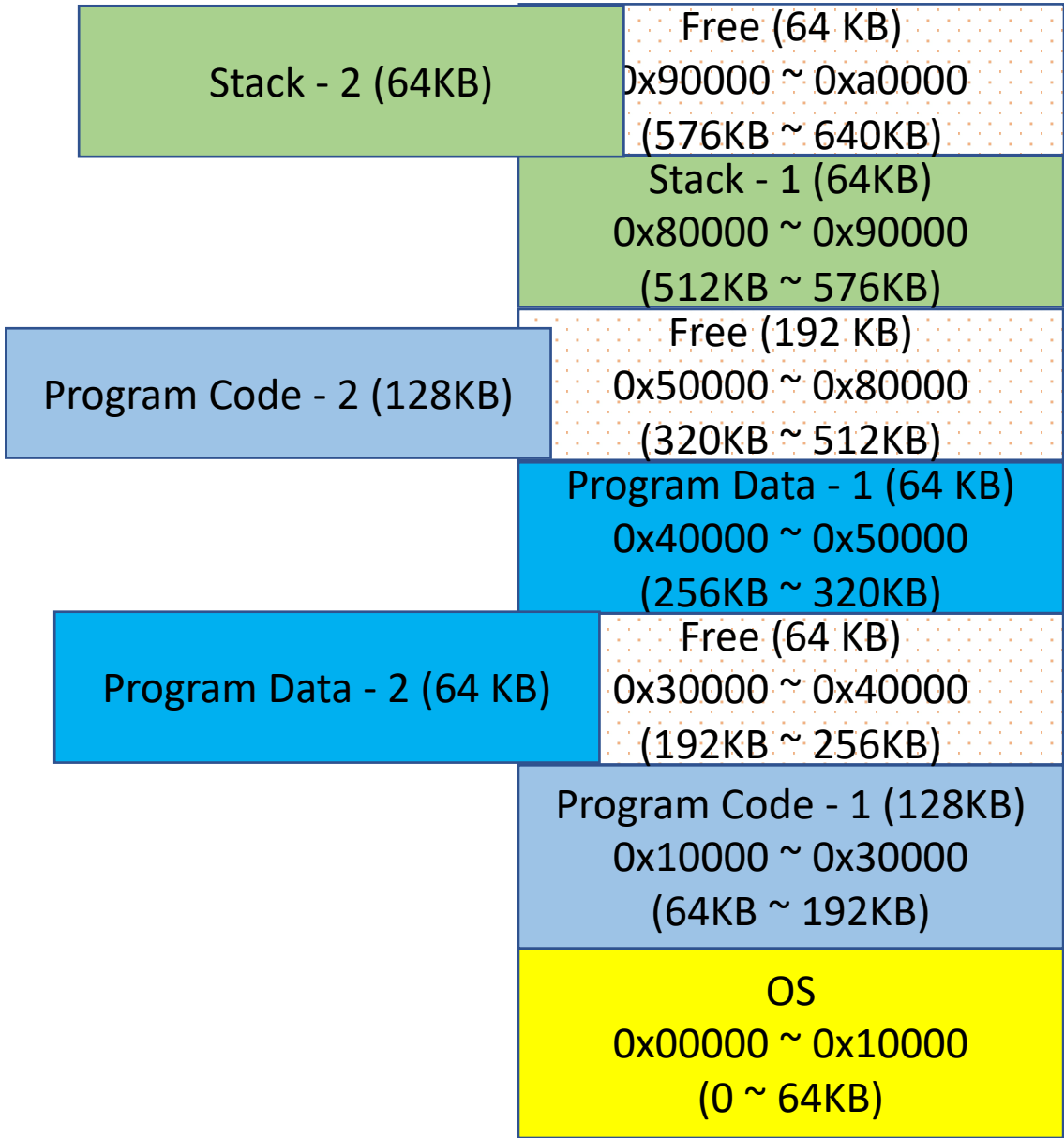


# Multi-programming Environment

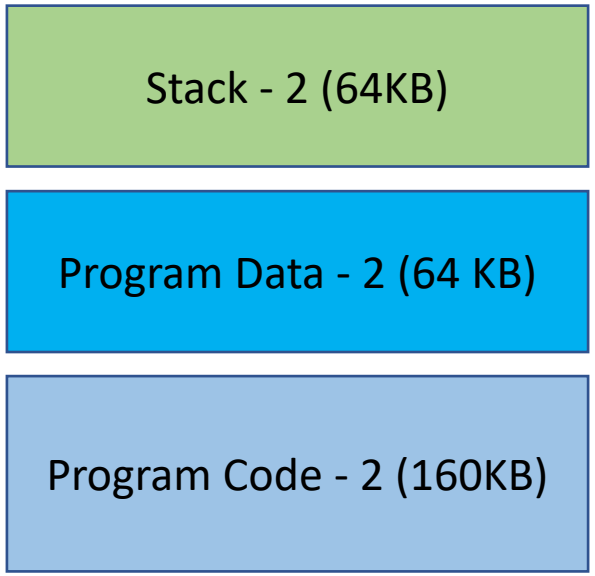
- Run two programs
- System's memory usage determines allocation
- Program need to be aware of the environment
  - Where does system loads my code?
  - You can't determine… system does..

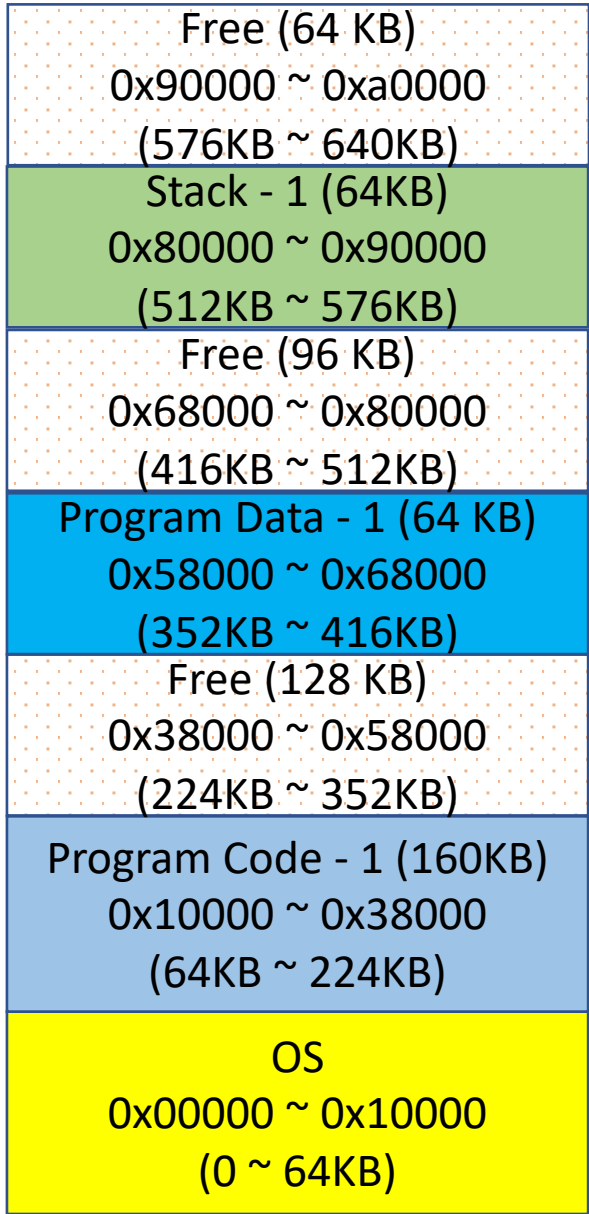
No Transparency…



# Multi-programming Environment

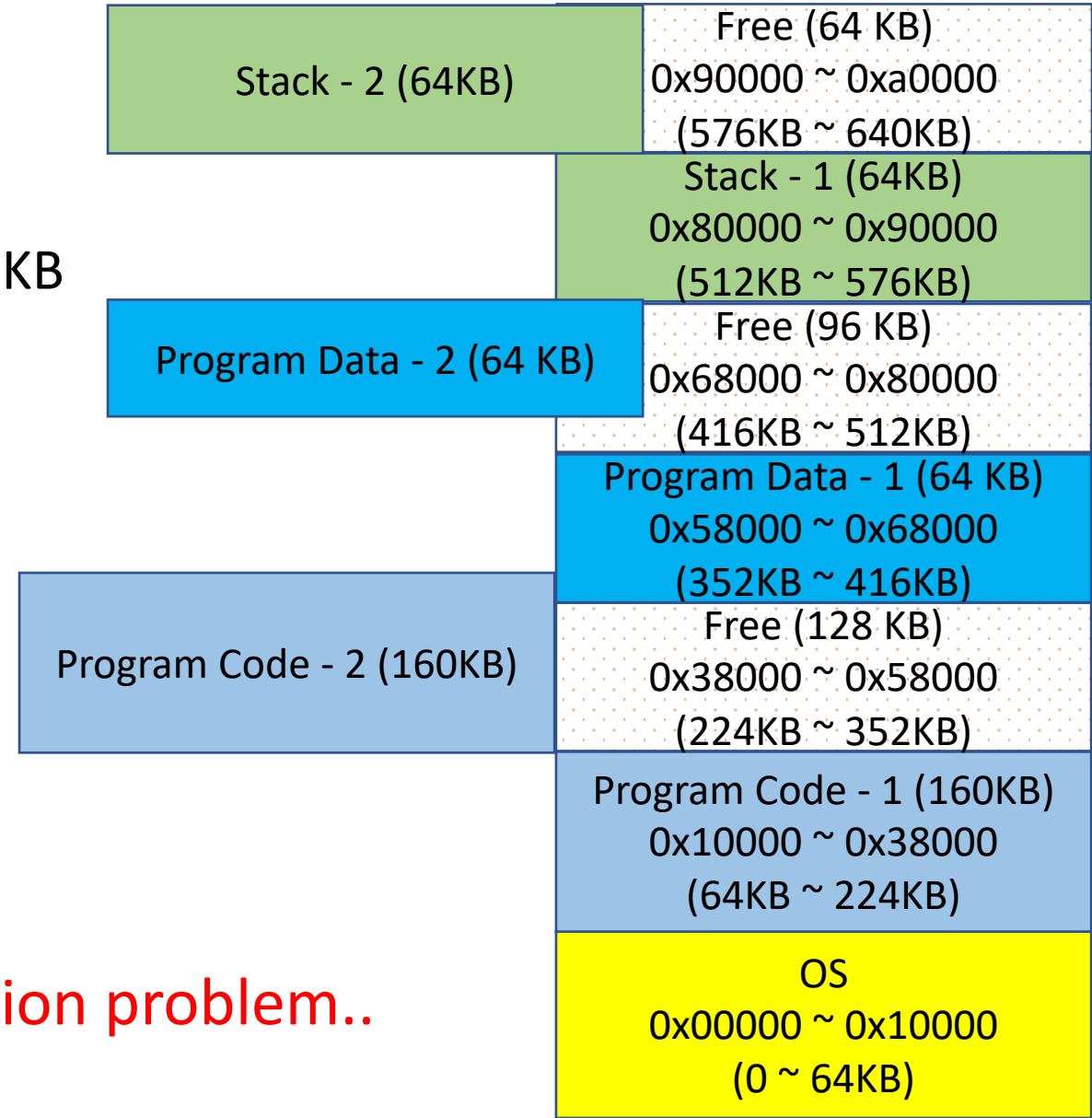
- Run two programs



# 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…

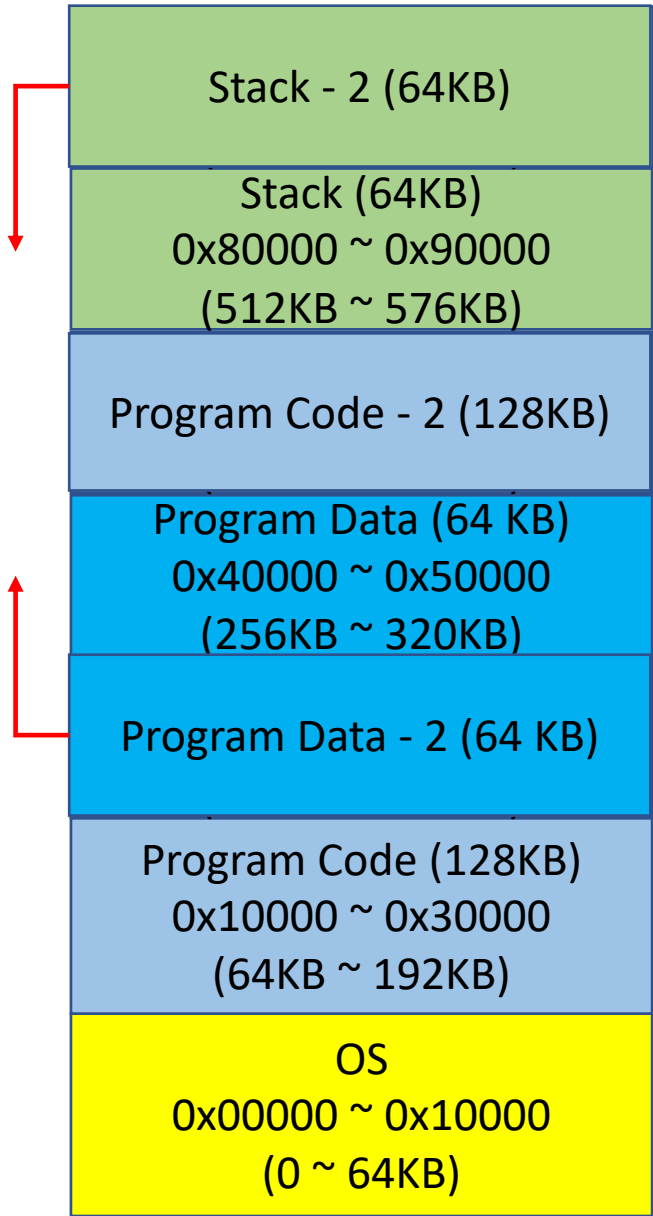



Not efficient.. Suffers memory fragmentation problem..

# 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

No isolation. Security problem.



# 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

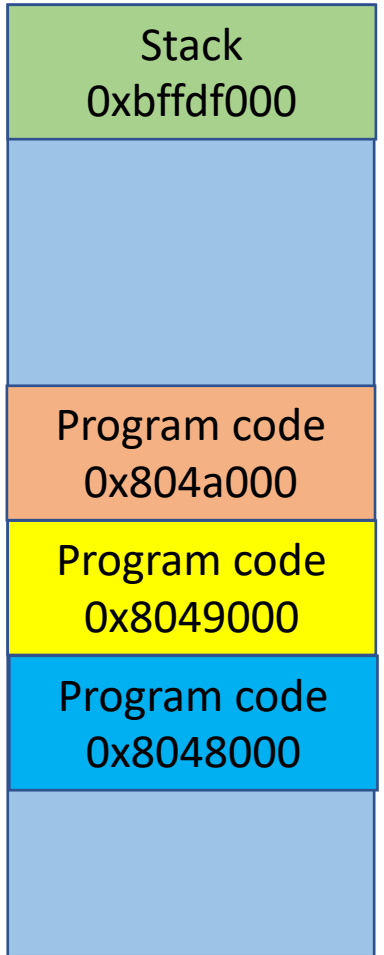
- A method of implementing virtual memory

- 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.

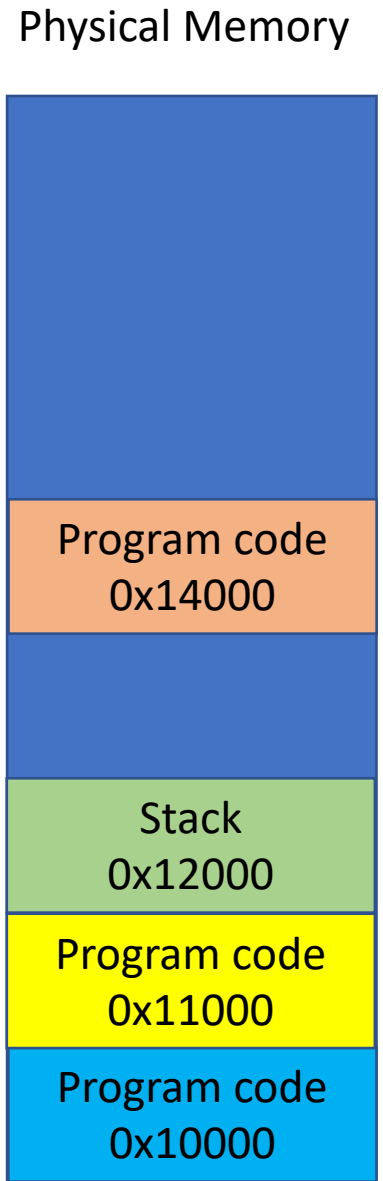
- Having an indirect map between virtual page and physical page
  - Set an arbitrary virtual address for a page, e.g., 0x81815000
  - Set a physical address to that page as a map, e.g., 0x32000
  - 0x81815000 ~ 0x81815fff will be translated into
  - 0x32000 ~ 0x32fff

# Virtual Memory - Paging

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

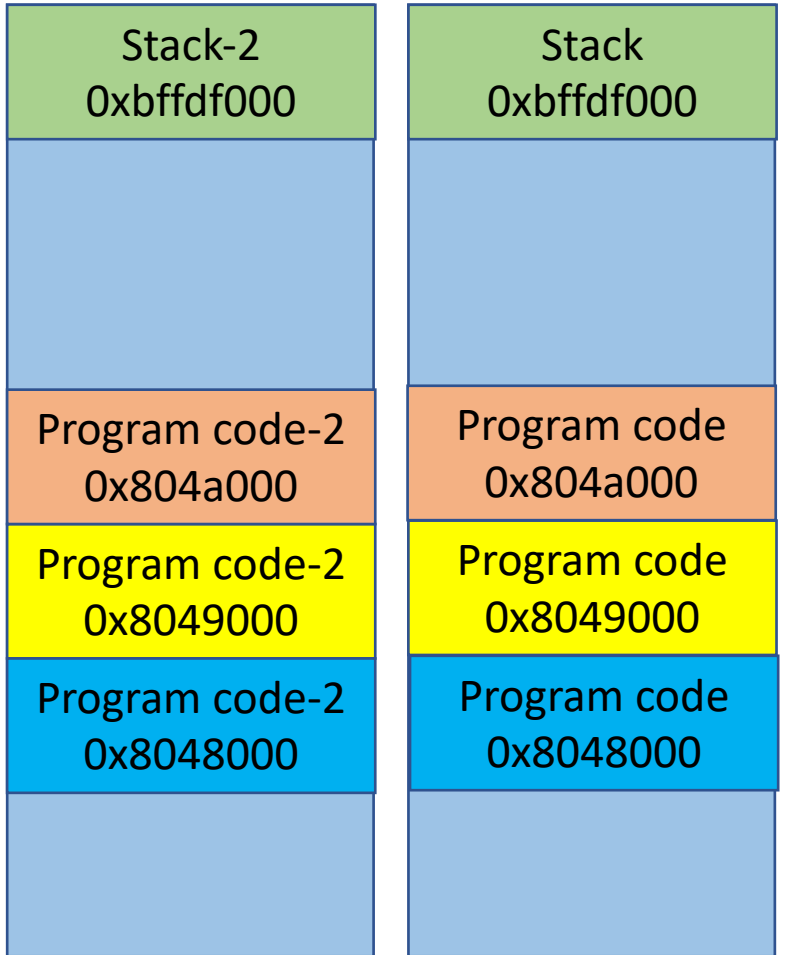



| Virtual | Physical |
|---|---|
| 0x8048000 | 0x10000 |
| 0x8049000 | 0x11000 |
| 0x804a000 | 0x14000 |
| 0xbffdf000 | 0x12000 |
| … | … |



# Paging: Virtual Memory

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



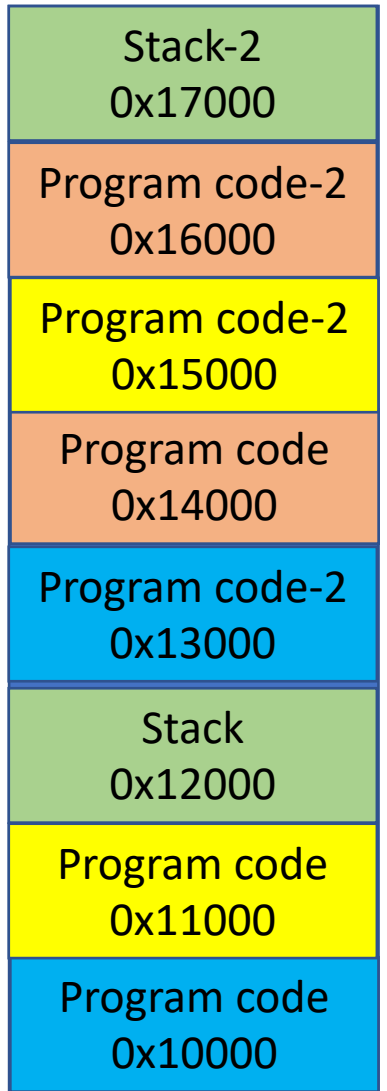




| Virtual | Physical |
|---|---|
| 0x8048000 | 0x10000 |
| 0x8049000 | 0x11000 |
| 0x804a000 | 0x14000 |
| 0xbffdf000 | 0x12000 |
| … | … |

| Virtual-2 | Physical-2 |
|---|---|
| 0x8048000 | 0x13000 |
| 0x8049000 | 0x15000 |
| 0x804a000 | 0x16000 |
| 0xbffdf000 | 0x17000 |
| … | … |

Physical Memory

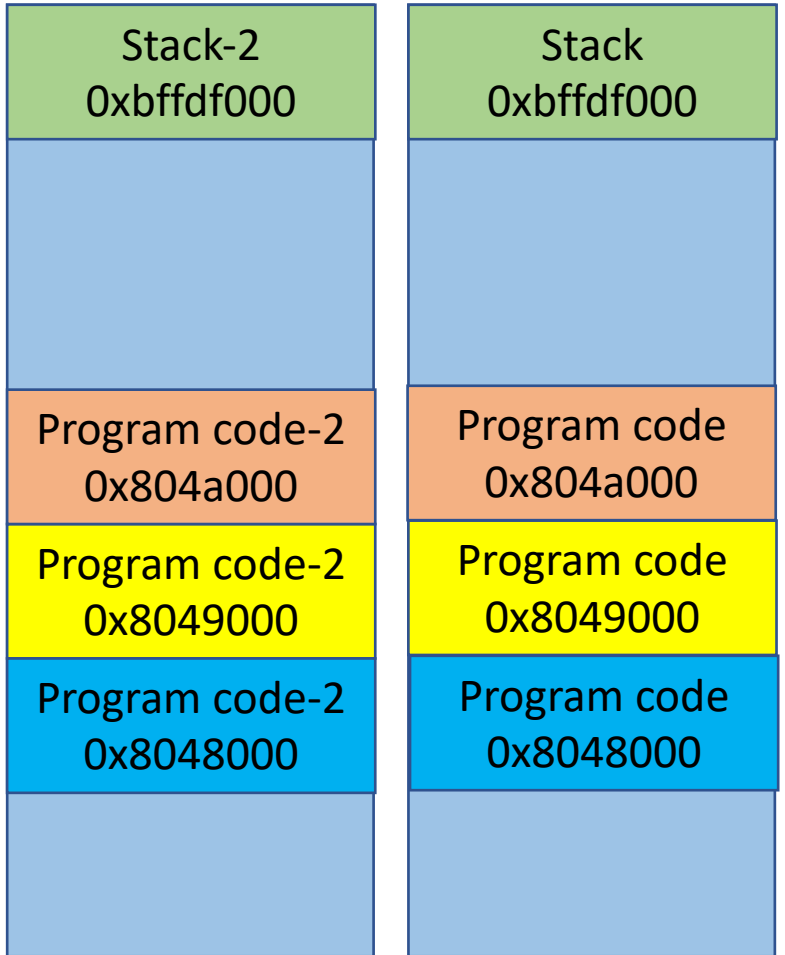


Transparency: does not need to know system's internal state

Program A is loaded at 0x8048000.

Can Program B be loaded at 0x8048000?

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



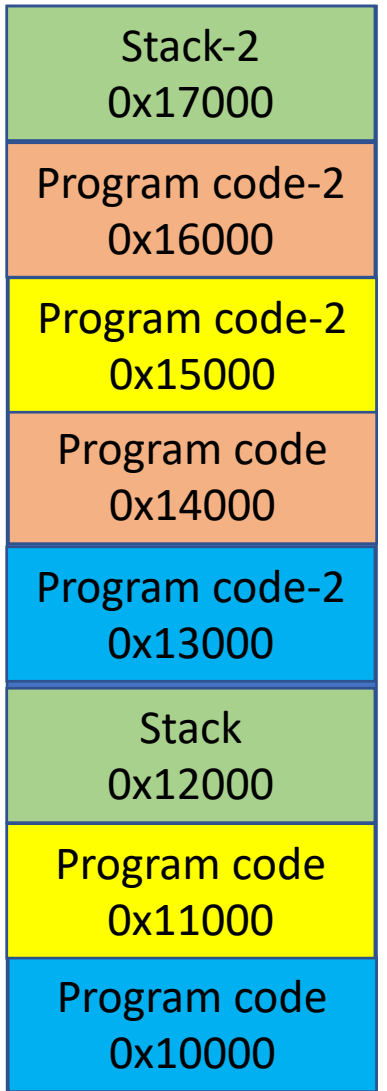




| Virtual | Physical |
|---|---|
| 0x8048000 | 0x10000 |
| 0x8049000 | 0x11000 |
| 0x804a000 | 0x14000 |
| 0xbffdf000 | 0x12000 |
| … | … |

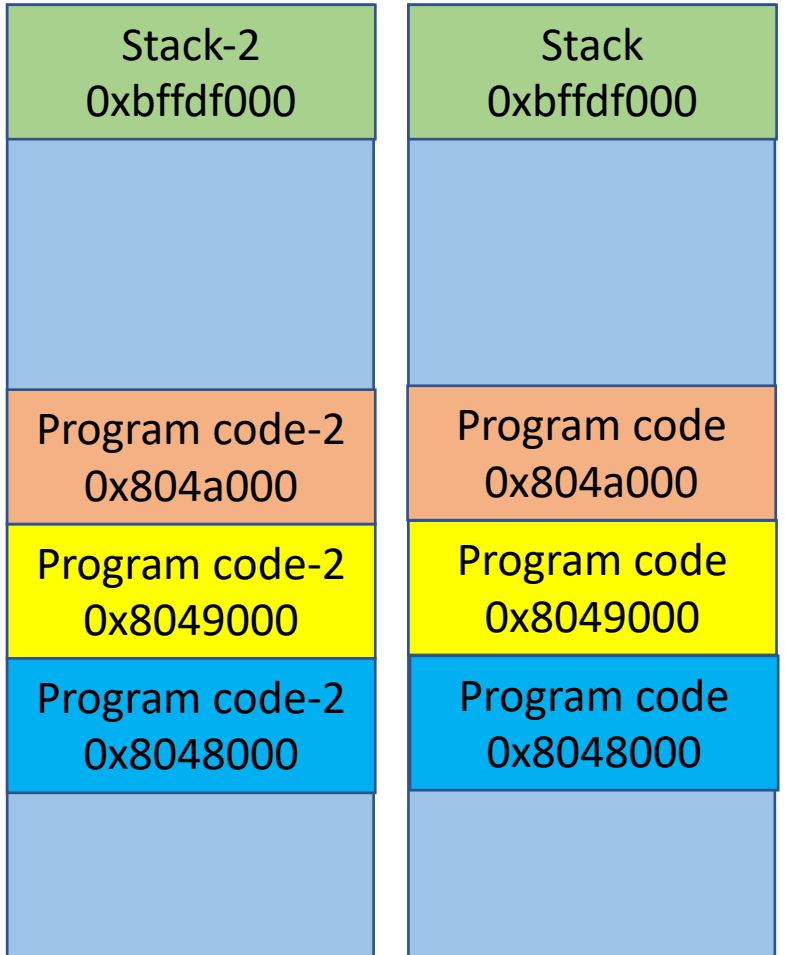
| Virtual-2 | Physical-2 |
|---|---|
| 0x8048000 | 0x13000 |
| 0x8049000 | 0x15000 |
| 0x804a000 | 0x16000 |
| 0xbffdf000 | 0x17000 |
| … | … |

Physical Memory



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



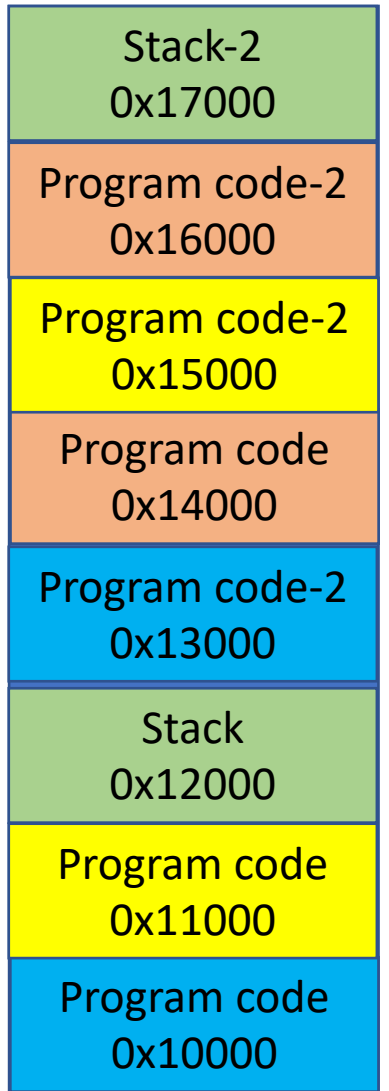




| Virtual | Physical |
|---|---|
| 0x8048000 | 0x10000 |
| 0x8049000 | 0x11000 |
| 0x804a000 | 0x14000 |
| 0xbffdf000 | 0x12000 |
| … | … |

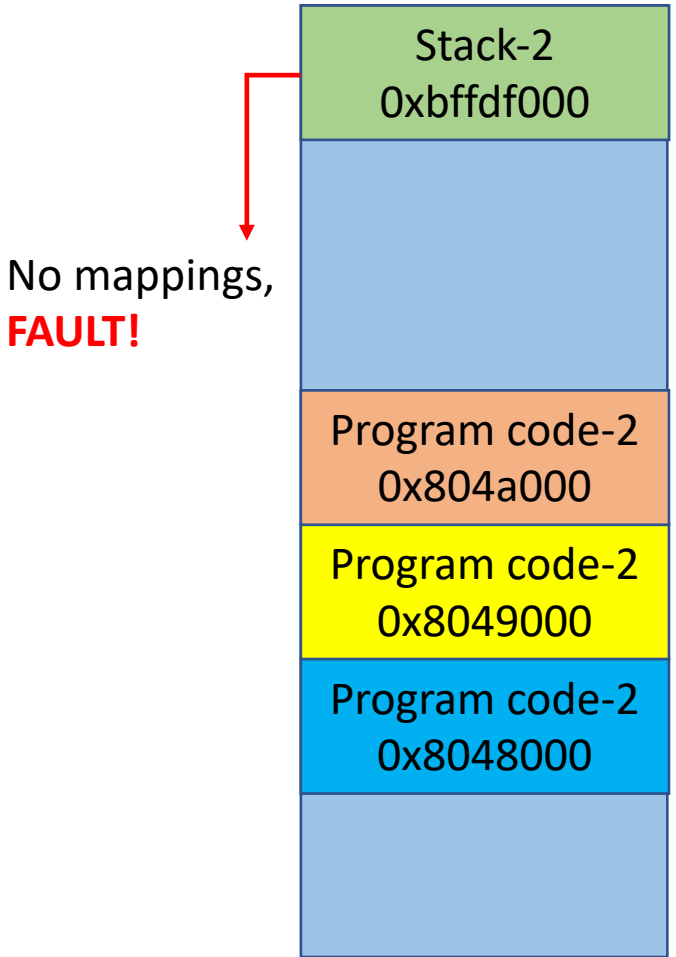
| Virtual-2 | Physical-2 |
|---|---|
| 0x8048000 | 0x13000 |
| 0x8049000 | 0x15000 |
| 0x804a000 | 0x16000 |
| 0xbffdf000 | 0x17000 |
| … | … |

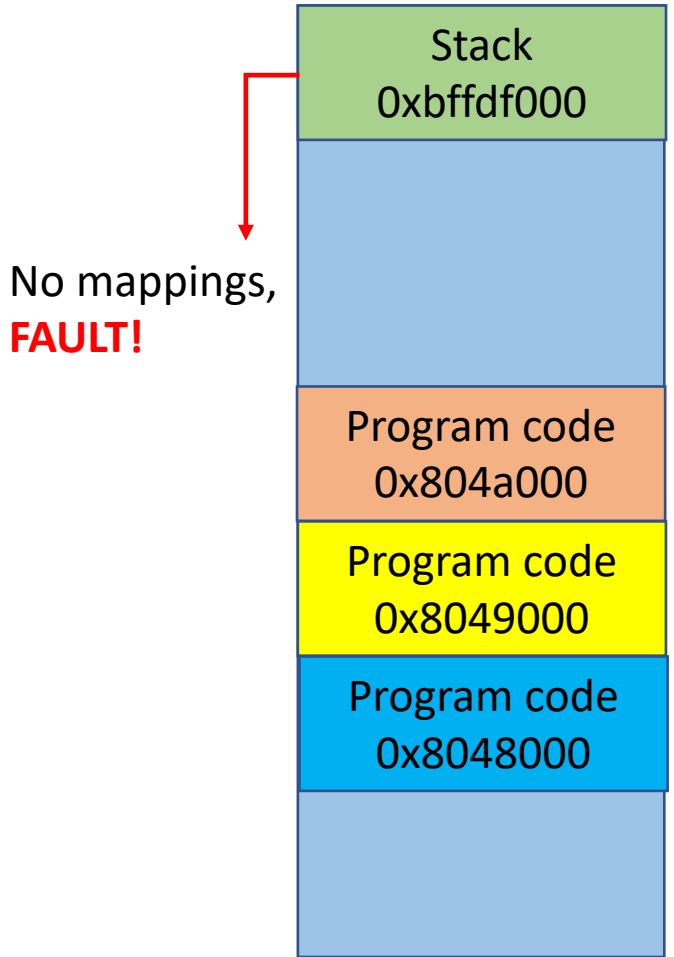
Physical Memory

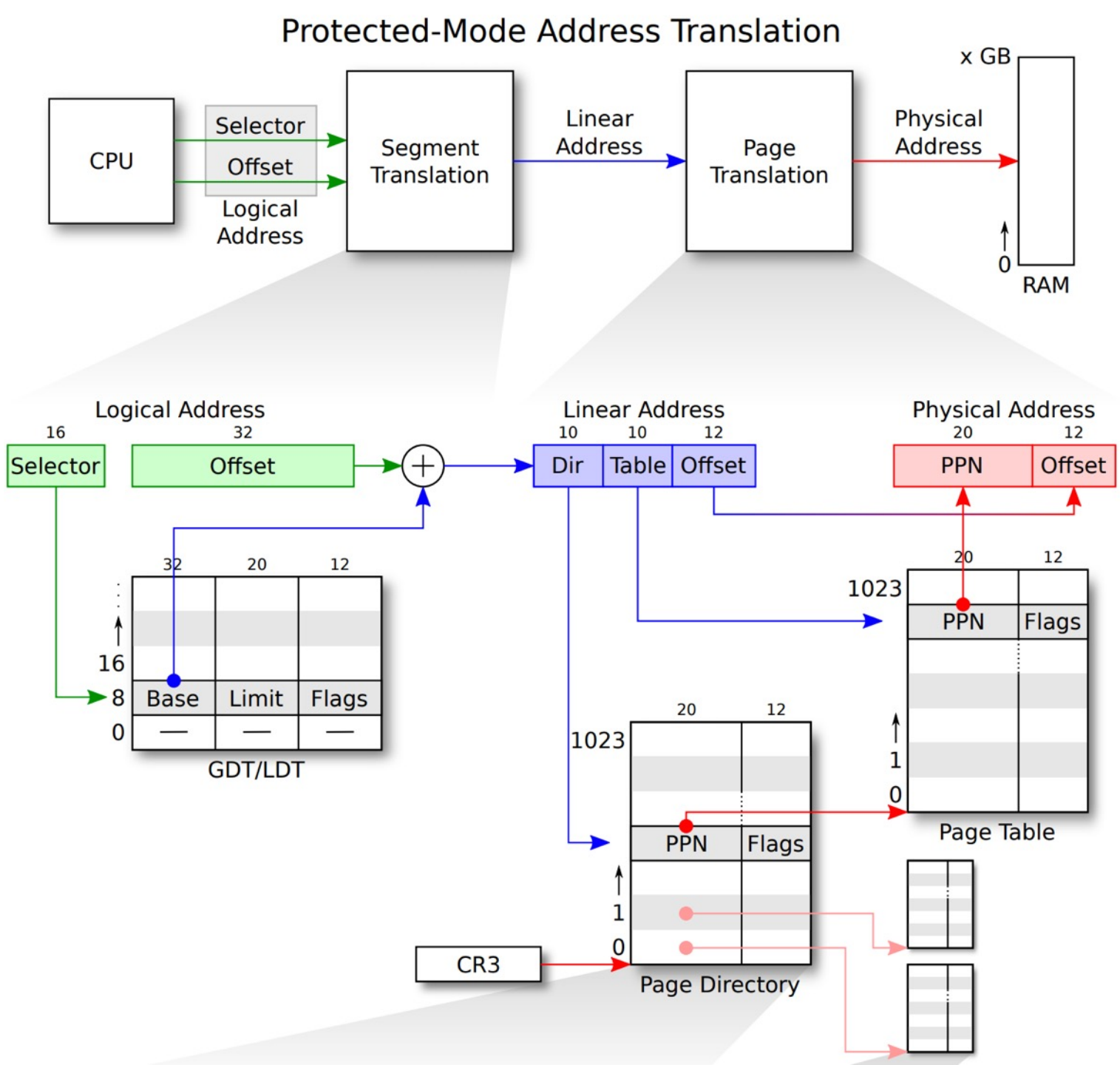


Protection: isolate program's execution environment

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
0
RAM
Logical Address
16
32
Selector
Offset
32
20
12
16
8
0
Base
Limit
Flags
GDT/LDT
Linear Address
10
10
12
Dir
Table
Offset
1023
0
CR3
Page Directory
Physical Address
20
12
PPN
Offset
20
12
1023
PPN
Flags
1
0
Page Table