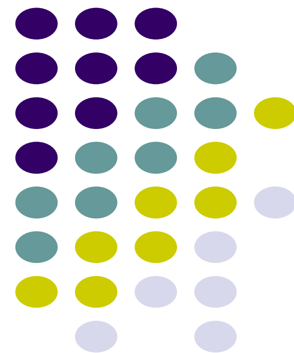
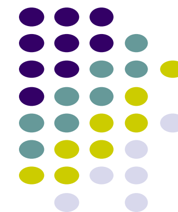
Virtual Memory Background: Address Binding & Linking

ECE 469, Jan 21

Aravind Machiry



Booting



Map code in BIOS at f000:fff0

Extended Memory (Over 1MB)

BIOS 0xf0000 ~ 0x100000 (960KB ~ 1MB)
Devices 0xc0000 ~ 0xf0000 (768KB ~ 960KB)
VGA 0xa0000 ~ 0xc0000 (640KB ~ 768KB)
Low Memory 0x00000 ~ 0xa0000 (0 ~ 640KB)

Read Master Boot Record (MBR) from the boot disk and load it at 0x7c00

Enabling Protected Mode

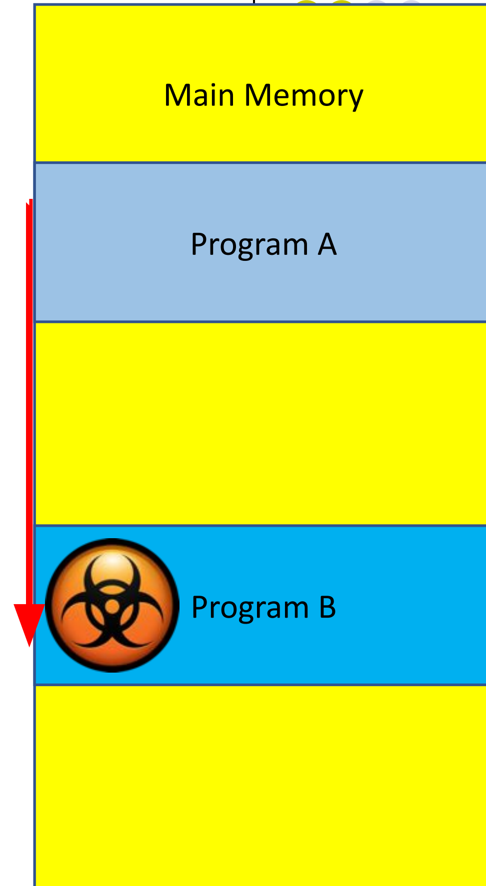
Load kernel and run!

Real mode

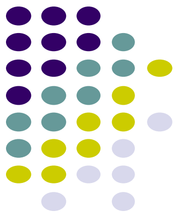
- Suppose two program runs at the same time

- Program A attempts to modify memory used by program B

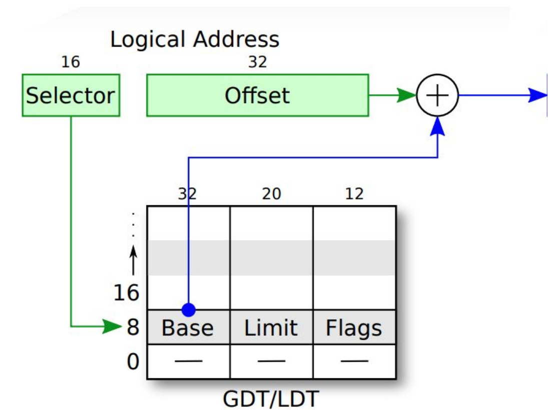
No SECURITY!



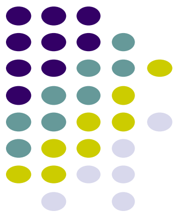
i386 Protected mode



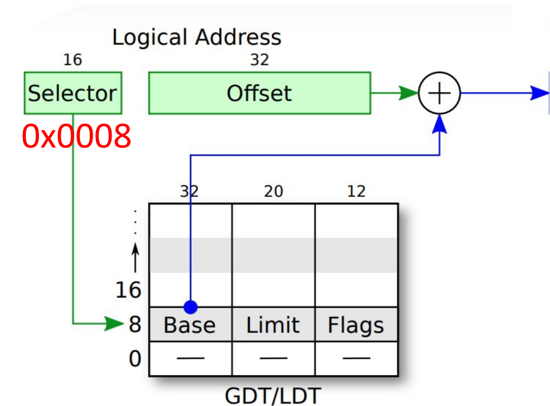
- Look at GDT (Global Descriptor Table)
  - Indexed by a segment register



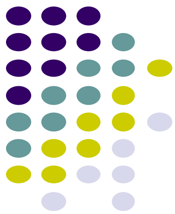
i386 Protected mode

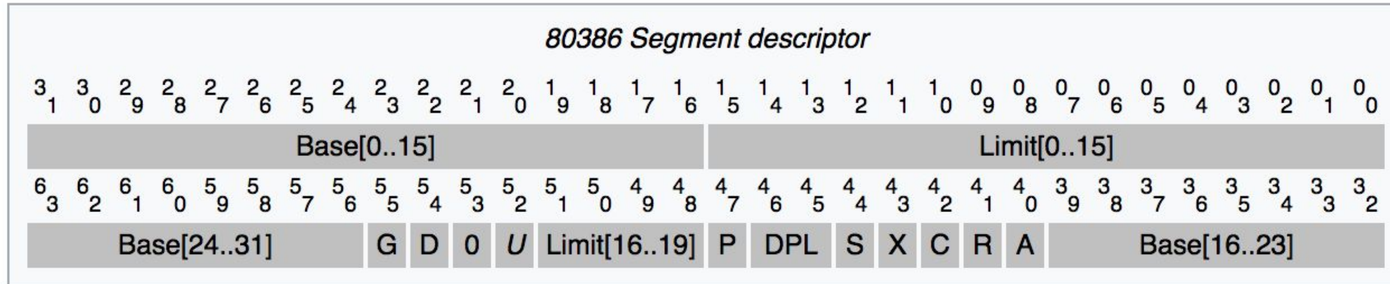


- Address 0x0008:0x00003400
- In the real mode:
  - 0x0008 * 16 + 0x3400 = 0x3480
- In the i386 protected mode
  - GDT[1].base + 0x3400
  - Access if 0x3400 is less than GDT[1].limit
  - Otherwise, raise an exception!



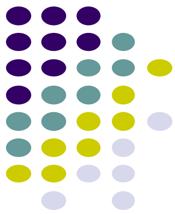
i386 Protected mode

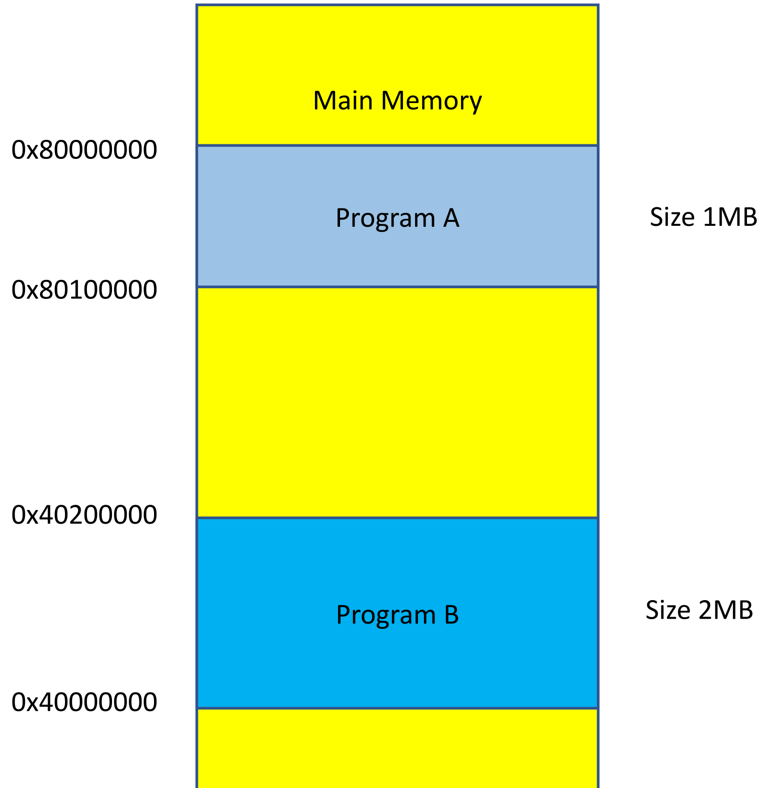




- G - Granularity (0 = byte, 1 = page)
  - 0: Limit will be byte granularity **(i.e., limit, only access 2^{20}, 1MB)**
  - 1: Limit will be page granularity **(i.e., limit * 4096, $2^{20} * 2^{12} = 2^{32}$)**
- D – Default operand size (0 = 16-bit, 1 = 32-bit)
  - Set the values of IP/SP with respect to this bit
- R,X – Readable/Executable
- DPL – Descriptor Privilege Level (a.k.a. Ring Level)
  - 0 (highest priv), 1, 2, 3 (lowest priv)

Segment Example





Size 1MB

Size 2MB

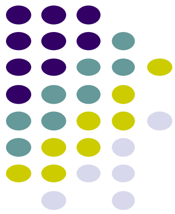
0x10:0 ~ 0x10:0x100000 are valid address for Program A

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

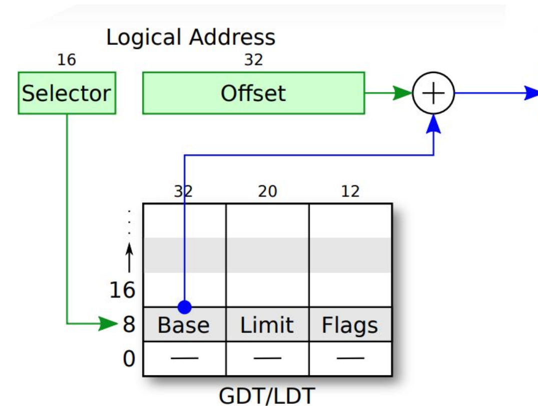


Protected mode - Examples

- 0x8:0x8080
  - Base: 0x40000000
  - Limit (addr): 0x8000000 (**0x08000 * 2^{12}**)
  - Offset: 0x8080

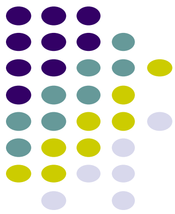
0x8080 < 0x8000000

Address: 0x40008080



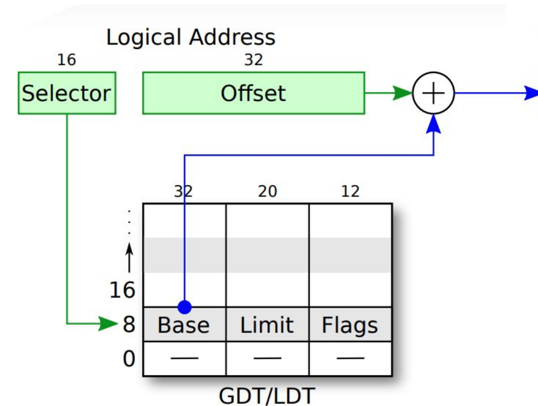
GDT index	32-bit Base	20-bit Limit	12-bit Flags
16	0x31300000	0x1000	G=0
8	0x40000000	0x08000	**G=1**
0	0x0	0x0	G=0

Protected mode - Examples

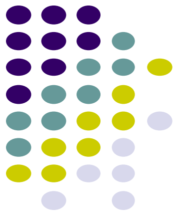


- 0x10:0x333
  - Base: 0x31300000
  - Limit (addr): 0x1000
  - Offset: 0x333

Address: 0x31310333



GDT index	32-bit Base	20-bit Limit	12-bit Flags
16	0x31300000	0x1000	G=0
8	0x40000000	0x08000	**G=1**
0	0x0	0x0	G=0

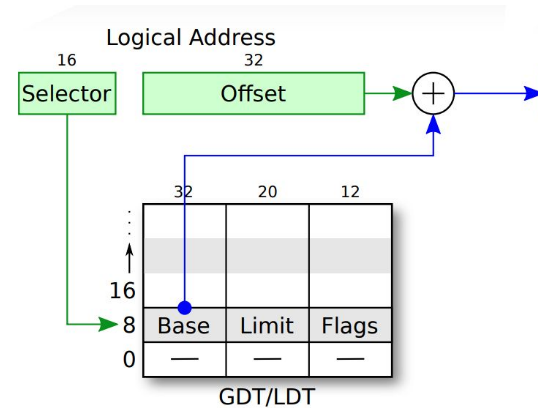


Protected mode - Examples

- 0x10:0x8080
  - Base: 0x31300000
  - Limit (addr): **0x1000**

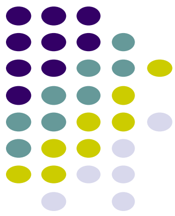
  - Offset: **0x8080**

  Offset >= limit, Access denied!



GDT index	32-bit Base	20-bit Limit	12-bit Flags
16	0x31300000	0x1000	G=0
8	0x40000000	0x08000	**G=1**
0	0x0	0x0	G=0

Protected mode - Memory Privilege Levels



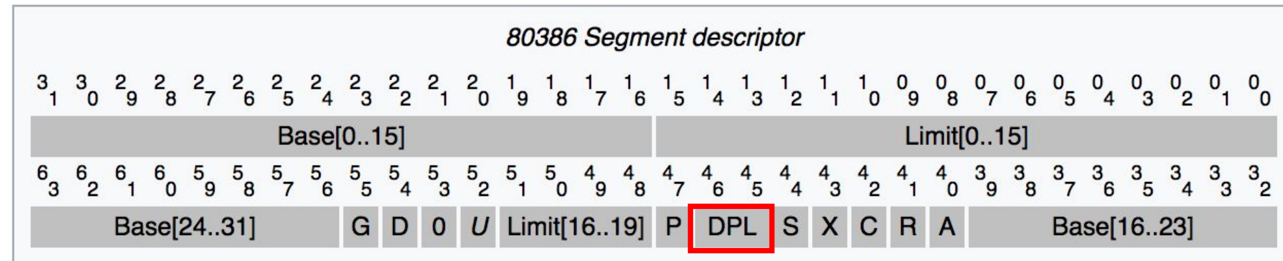
- DPL (Descriptor Privilege Level)

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

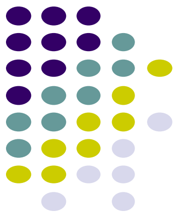
  - 2 (10) – highest user-level privilege
  - 3 (11) – user-level privilege

Kernel: for privileged OS operations…

User: for unprivileged applications…

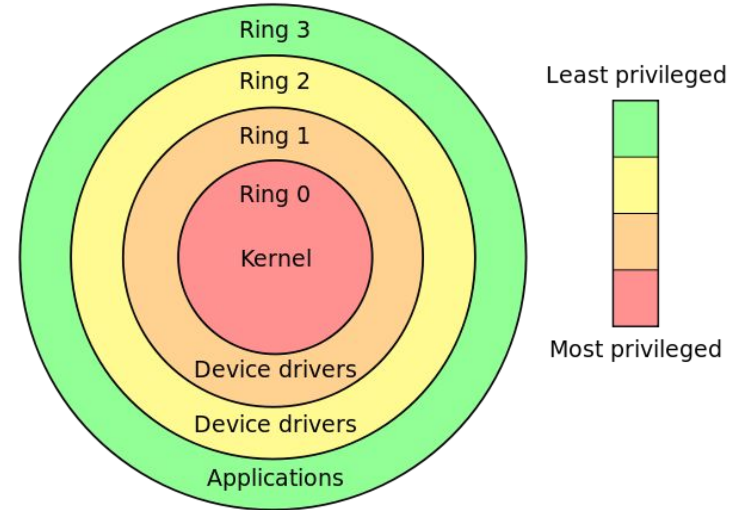


Protected mode - Memory Privilege Levels

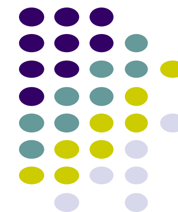


- DPL (Descriptor Privilege Level)
- Protected mode – four levels of memory privilege
  - 0 (00) – highest, OS kernel
  - 1 (01) – OS kernel

  - 2 (10) – highest user-level privilege
  - 3 (11) – user-level privilege
- Typically, 0 is for kernel, 3 is for user…

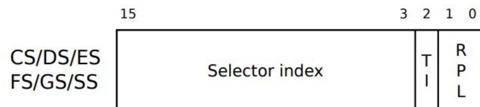


DPL Defines Ring Level



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

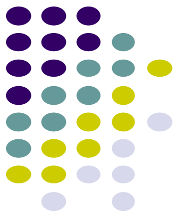
- Examples
  - %cs == 0x8 == 1000 in binary, last 2 bits are ZERO -> KERNEL!
  - %cs == 0x13 == 10011 in binary, last 2 bits are 3 -> USER!
  - %cs == 0x10 == 10000 in binary, last 2 bits are 0 -> KERNEL!
  - %cs == 0xb == 1011….



TI Table index (0=GDT, 1=LDT)
RPL Requester privilege level

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**

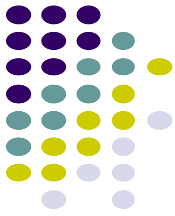
DPL Defines Ring Level



- 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**

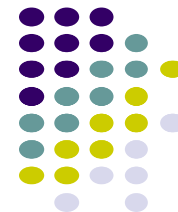
GDT index	32-bit Base	20-bit Limit	12-bit Flags
16 **USER**	0x31310000	0x1000	G=0, **DPL=3**
8 **KERNEL**	0x40000000	0x80000	G=1, **DPL=0**
0 **KERNEL**	0x0	0xfffff	G=1, **DPL=0**

Ring 0 (Kernel) can go to Ring 3 (User)



- 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: Create GDT



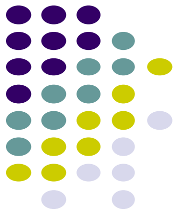
- In boot/boot.S
  - %cs to point 0 ~ 0xffffffff in DPL 0
  - %ds to point 0 ~ 0xffffffff in DPL 0

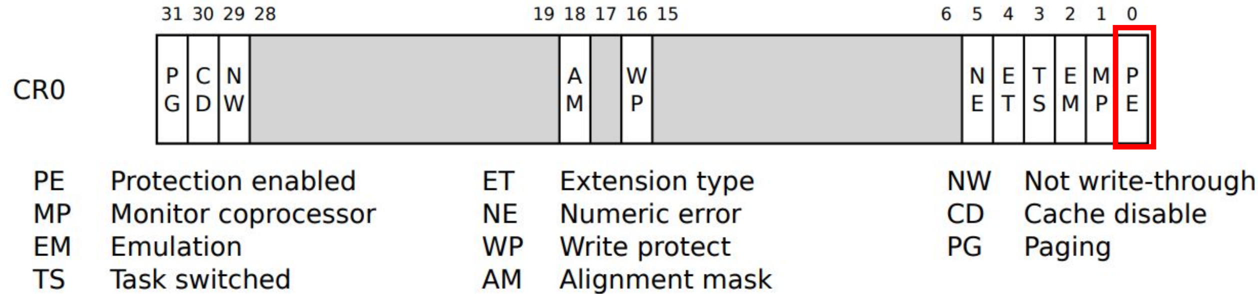
- Only kernel can access those two segment

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

```
# 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
```

Enabling Protected Mode: Change CR0





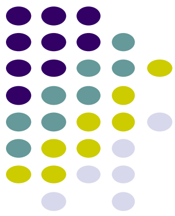
Set PE (Protected enabled) to **1** will enable Protected Mode

```
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!!)

Protected Mode Summary

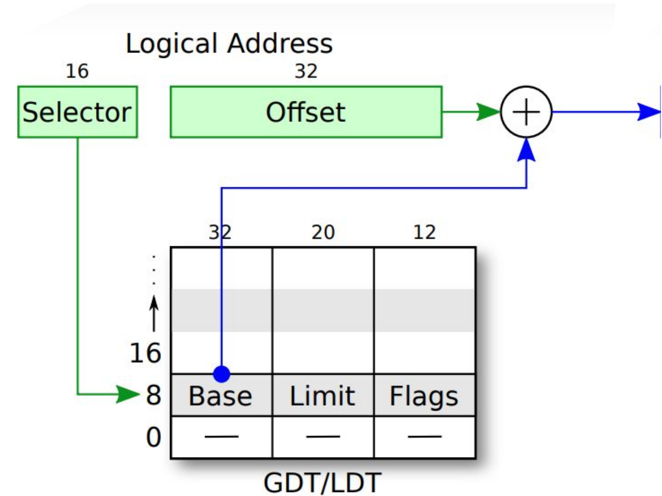


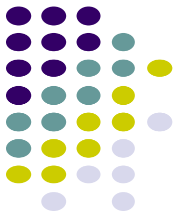
- Segment access via GDT
  - Base + Offset if Offset < Limit * 4096 (if G == 1)
  - Base + Offset if Offset < Limit (if G == 0)

- 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

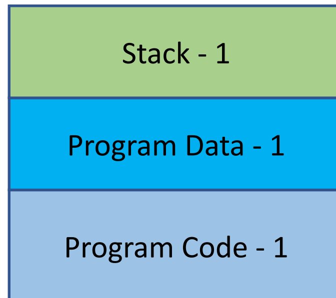
- Changing CPL?
  - ljmp %cs:xxxxx, set the last 2 bits of %cs as 0 for kernel, 3 for user





Uniprogramming Environment

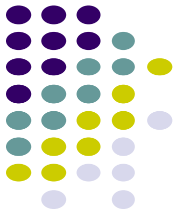
- Run one program
- The program can use memory space freely…



Free (576 KB)
0x10000 ~ 0xa0000
(64KB ~ 640KB)

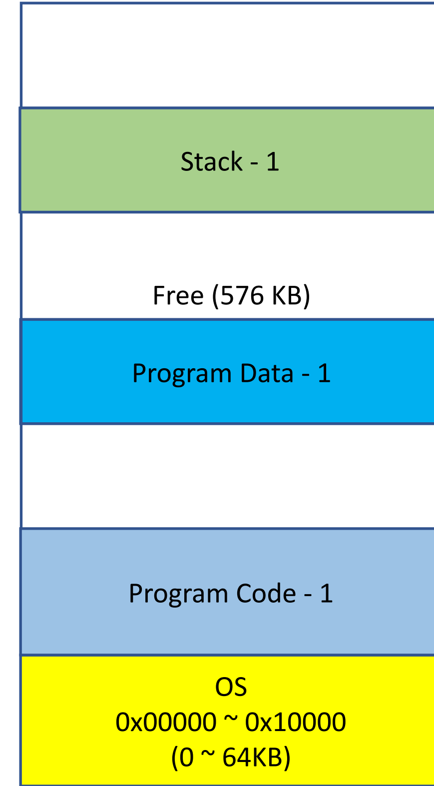
OS
0x00000 ~ 0x10000
(0 ~ 64KB)

Uniprogramming Environment

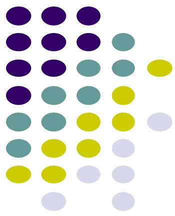


- Run one program

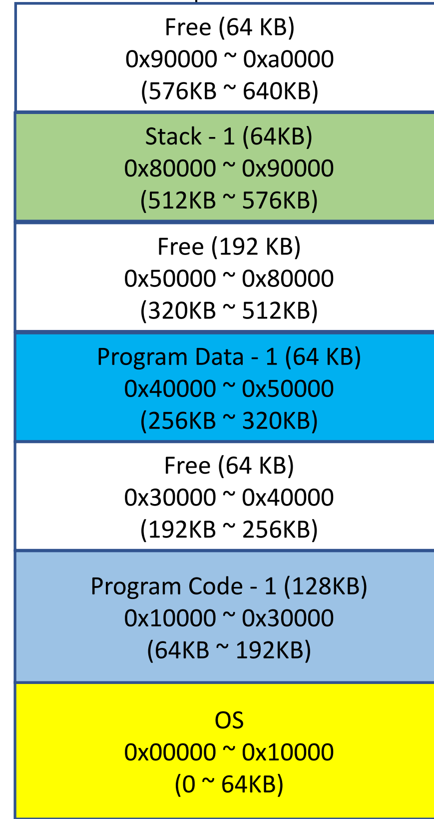
- The program can use memory space freely…

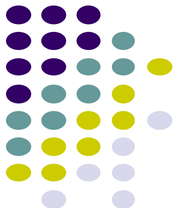


Uniprogramming Environment



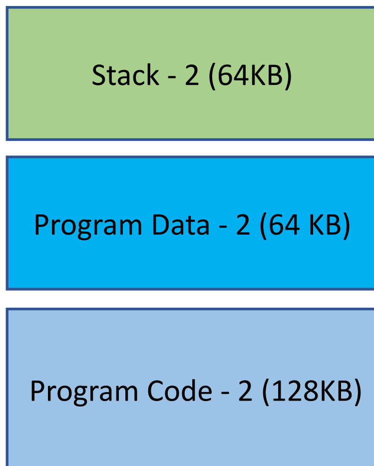
- Run one program
- The program can use memory space freely…

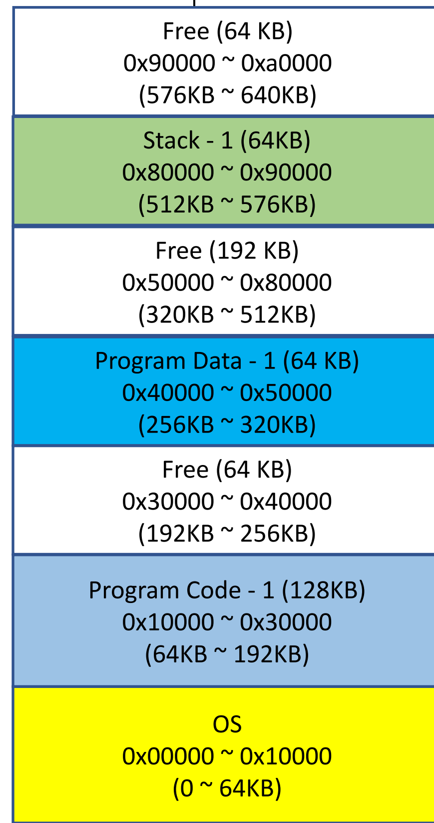




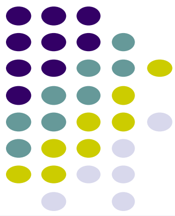
Multi-programming Environment

- Run two programs



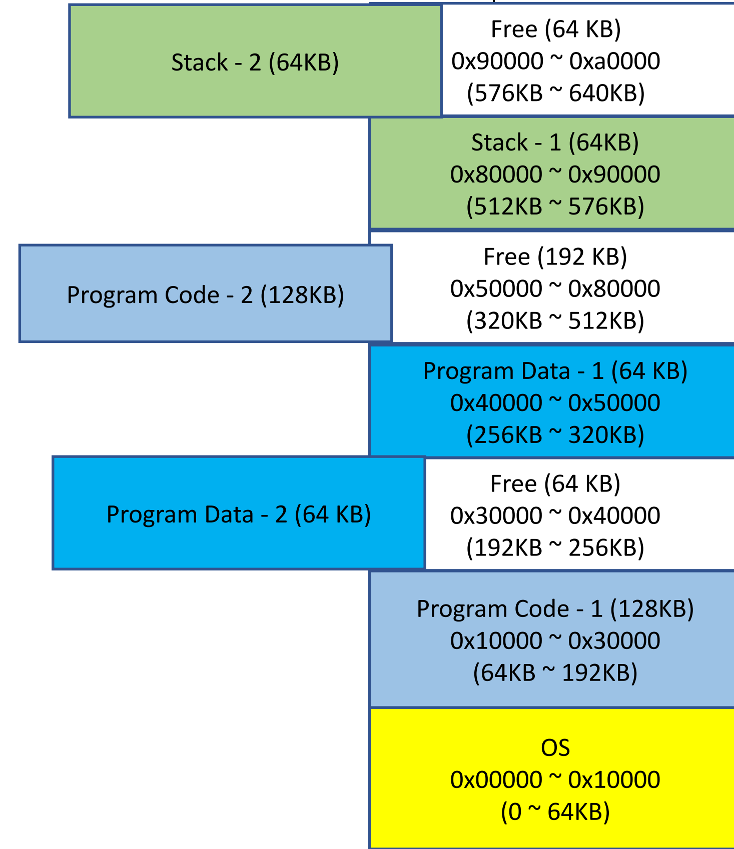


Multi-programming Environment

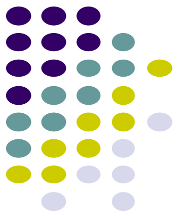


- Run two programs
- System's memory usage determines allocation
- Program need to be aware of the environment
  - Where does system loads my code?
  - You can't determine… system does..

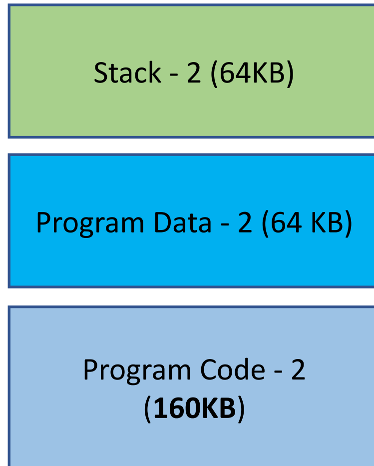
No Transparency…

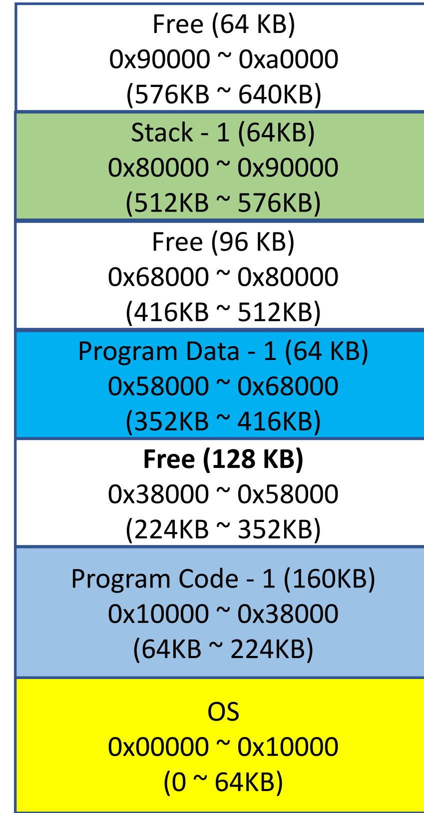


Multi-programming Environment

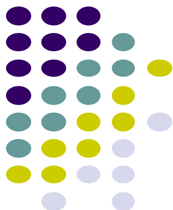


- Run two programs

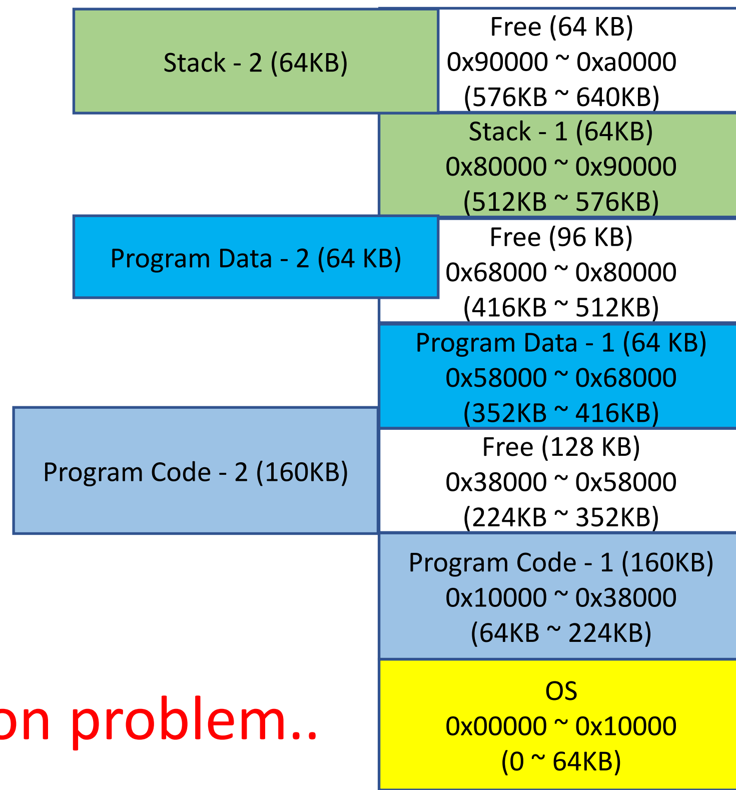




Multi-programming Environment

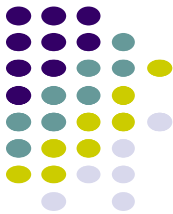


- Run two programs
  - Program size: 64KB + 64KB + 160K = 288KB
- Free mem: 64 + 96 + 128 = 288KB
- Cannot run Program – 2
  - Can't fit…



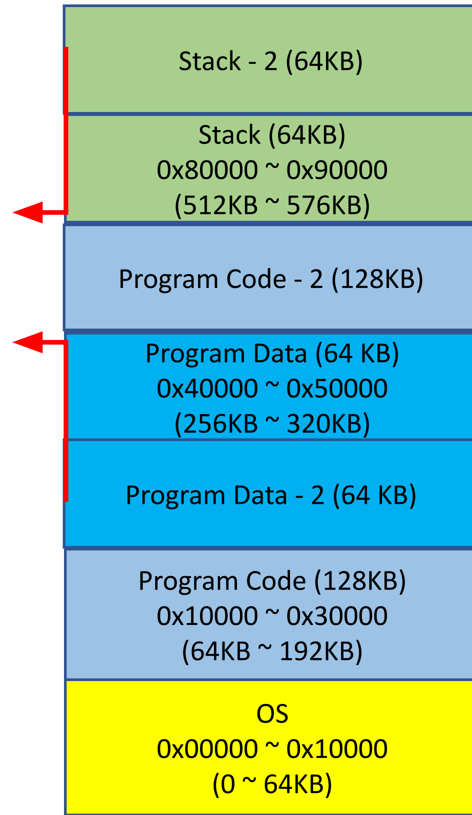
Not efficient.. Suffers memory fragmentation problem..

Multi-programming Environment

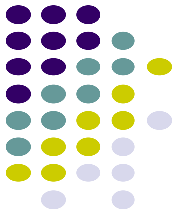


- Run two programs
- What if Program-2's stack underflows?
- What if Program-2's data overflows?
- Without virtual memory
  - Programs can affect to the other's execution

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?