CS444/544
Operating Systems II

Lecture 3

Virtual Memory

Protected mode

4/8/2024

Acknowledgement: Slides drawn heavily from Yeongjin Jiang

Oregon State University

Odds and Ends

- Office Hours starts this week, where?
  - Recitations: in person
  - All the rest: remote/online via discord
  - Check Canvas → Office Hours page

- Lab setup and Lab1 have been posted
  - Read document
  - Watch tutorial video + read lab slides

Public Key Error

```
Warning: Permanently added the ECDSA host key
Permission denied (publickey).
fatal: Could not read from remote repository.
```

- It means that you did not setup your ssh keys correctly

- To solve it:
  - Generate ssh key pair using Lab1_slides (slide 11 & 12)
  - Add your public key to your GitHub account
    - Instructions: https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account

Failed to bind socket: Address already in use

```
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,f
ormat=raw -serial mon:stdio -gdb tcp::29007 -D qemu.log
qemu-system-i386: -gdb tcp::29007: Failed to bind socket: Address already in use
make: *** [qemu-nox] Error 1
```

Run `$ kill-qemu`

kill-qemu

- This command will kill all running qemu instances that is owned by your account
- Please ignore the error message
  - It tries to kill qemu that is not owned by you, and has no effect to them

```
os2 ~/cs444/s21/os2-lab1-Rogersyp 118% kill-qemu
pkill: killing pid 172768 failed: Operation not permitted
pkill: killing pid 214733 failed: Operation not permitted
os2 ~/cs444/s21/os2-lab1-Rogersyp 119% make qemu-nox
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,i
ndex=0,media=disk,format=raw -serial mon:stdio -gdb tcp::2622
0 -D qemu.log
444544 decimal is XXX octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Add ~/bin to PATH in your .*shrc

- For students who typed ‘n’ on .bashrc installation,
- Please add ~/bin to your PATH environmental variable. E.g.,
  - export PATH=$PATH:~/bin

- Alternatively, you can modify the conf/env.mk file, and set

  QEMU=~/.cs444/bin/qemu-system-i386
- This will remove the errors like

```
***
*** Error: Couldn't find a working QEMU executable.
*** Is the directory containing the qemu binary in your PATH
*** or have you tried setting the QEMU variable in conf/env.mk?
***
```

Device or Resource Busy…

```
[coe_jangye@os2 (lab1) ~/jos$] make grade
make clean
make[1]: Entering directory `/nfs/stak/users/coe_jangye/jos'
rm -rf obj .gdbinit jos.in qemu.log
rm: cannot remove ‘obj/boot/.nfs00000000b4434e8600000025’: Device or resource busy
make[1]: *** [clean] Error 1
make[1]: Leaving directory `/nfs/stak/users/coe_jangye/jos'
'make clean' failed.  HINT: Do you have another running instance of JOS?
make: *** [grade] Error 1
```

- This occurs when your tmux/vim/other apps working on some of the files that is required to be deleted by our ‘make’ script
- Kill all tmux/vim sessions would remove the problem
  - Make sure that you saved all your work!

Killing tmux

- RUN
  - $ kill-all-tmux

- Killing vims
  - $ ps aux | grep vim | grep your_username_here
  - The command above will show your instance of vim
  - You can kill it selectively by running
  - $ kill -9 [pid of vim]
  - Or,
  - $ pkill vim
  - to kill all vim instances…

Some other error messages

```
X11 forwarding request failed on channel 0
```

- Please ignore this error
  - It’s about forwarding GUI applications from the server to the client
  - We don’t use GUI applications on the server

- To enable: in ~/.ssh/config, do the following:

```
Host os2
    HostName os2.engr.oregonstate.edu
    User <USERNAME_HERE>
    ProxyJume access
    IdentityFile <Path_to_your_id_file>
    ForwardX11 yes
    ForwardX11Trusted yes
```

In Lab Tutorial…

- Following the boot sequence with ‘gdb’ in assembly and C code
  - Up to Exercise 6

- Learning how Intel x86 uses STACK to store a function’s local context
  - Exercise 10!

Recap – Real Mode

- Real mode segmentation, how?
  - **seg * 16 + offset**
  - [b000:b7ff] => 0xb000 * 16 + 0xb7ff = 0xbb7ff
- What is A20?
  - [f800:8001] => 0x100001?
  - [f800:8001] => 0x1?
- FYI, segment registers are:
  - %cs – code segment
  - %ds – data segment
  - %es – extra segment
  - %fs
  - %gs
  - %ss – stack segment

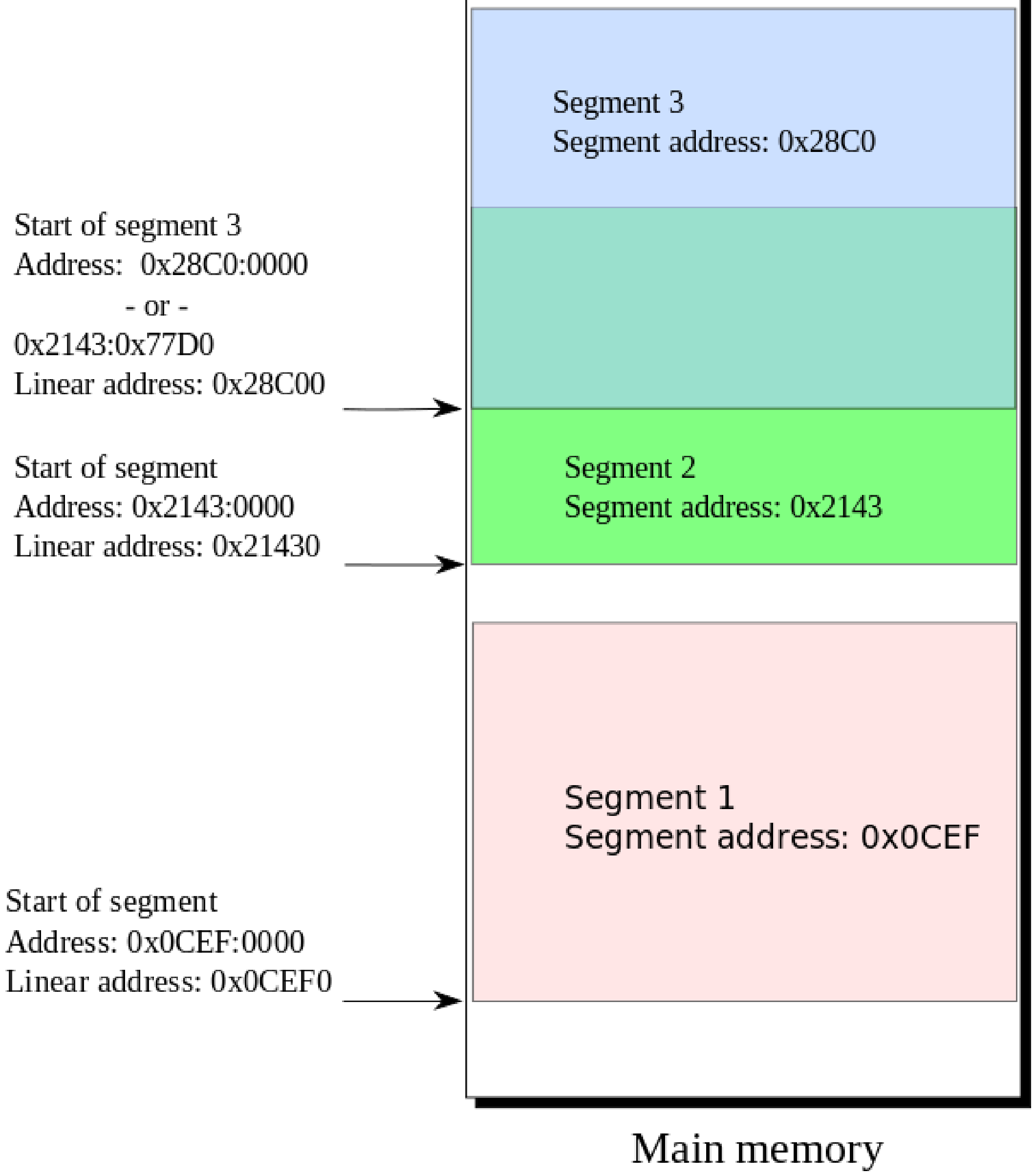
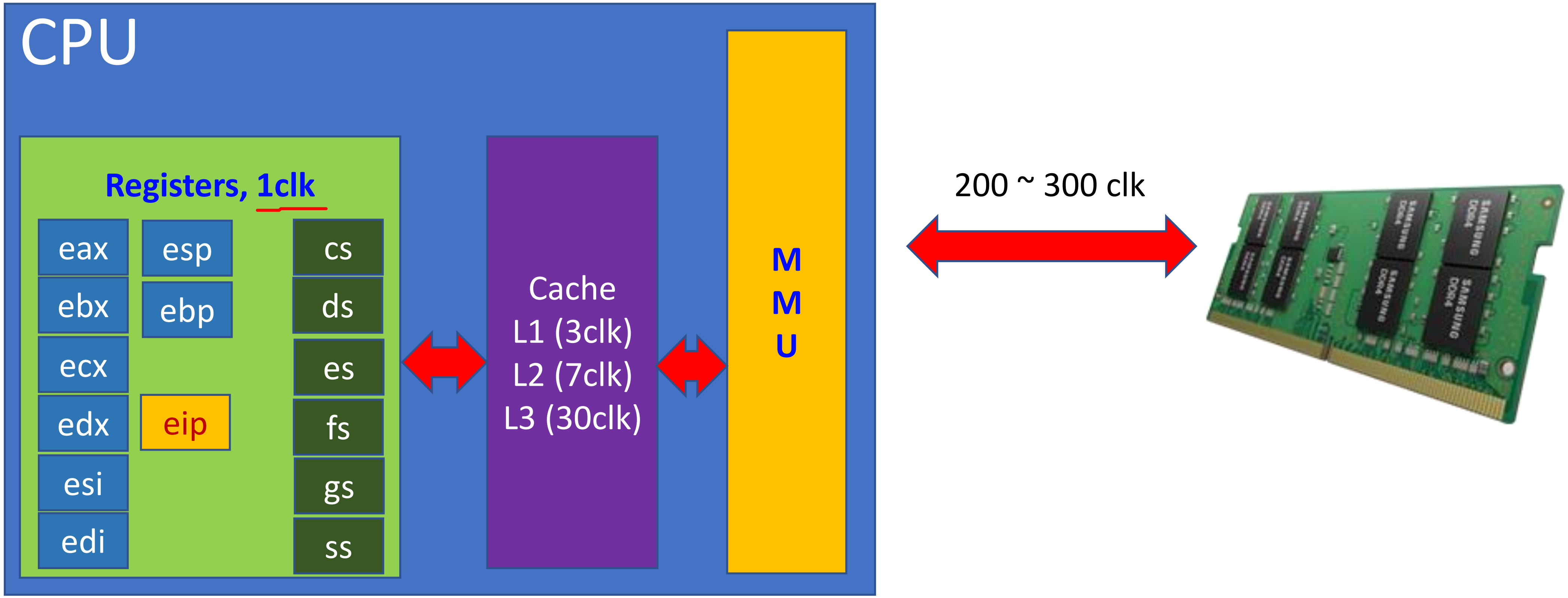
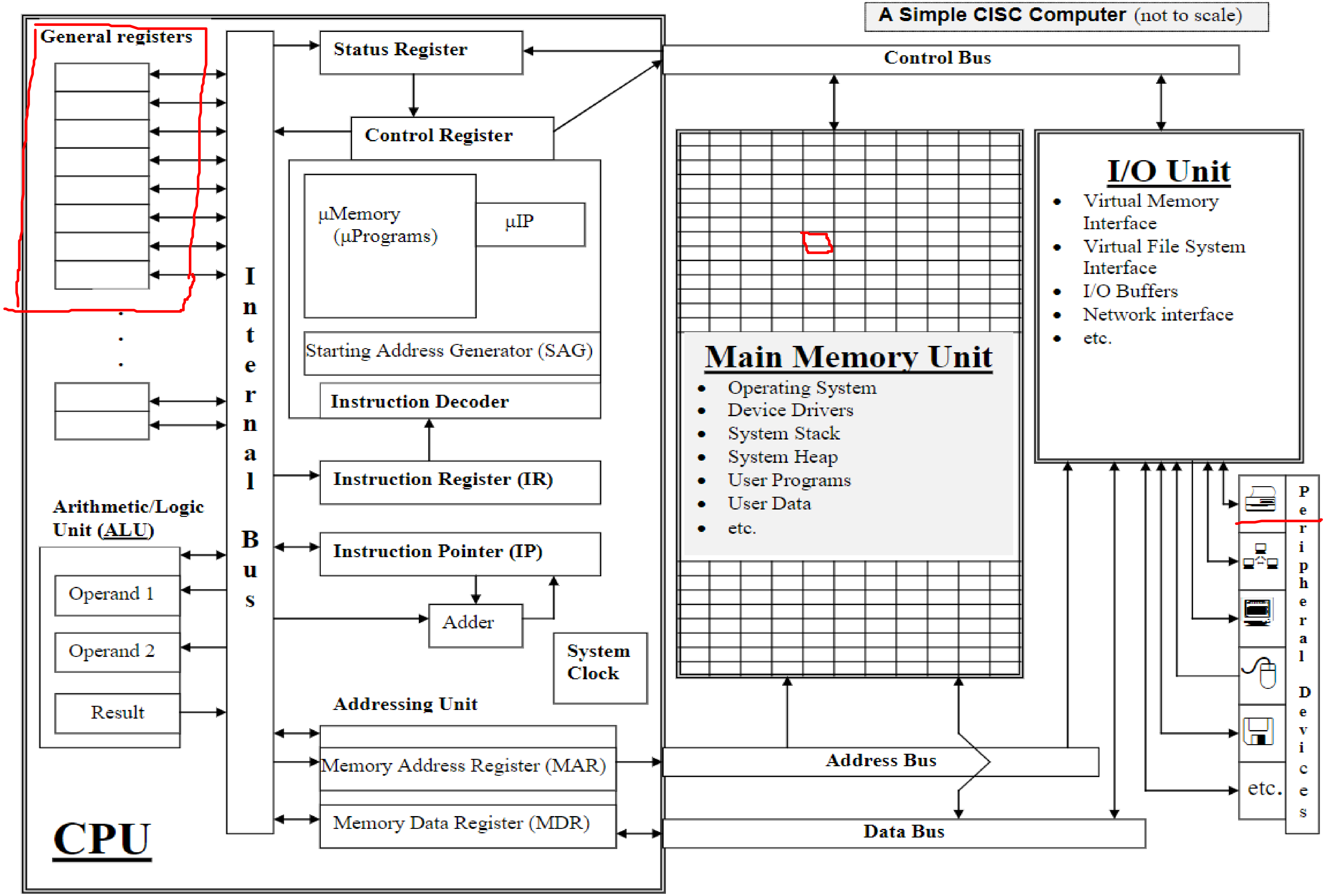


Image from: https://upload.wikimedia.org/wikipedia/commons/thumb/d/db/Overlapping_realmode_segments.svg

CPU / Registers / Memory





A Simple CISC Computer (not to scale)
General registers
Status Register
Control Register
μMemory
(μPrograms)
μIP
Starting Address Generator (SAG)
Instruction Decoder
Internal Bus
Instruction Register (IR)
Instruction Pointer (IP)
Adder
System Clock
Arithmetic/Logic Unit (ALU)
Operand 1
Operand 2
Result
Addressing Unit
Memory Address Register (MAR)
Memory Data Register (MDR)
CPU
Control Bus
Address Bus
Data Bus
Main Memory Unit
Operating System
Device Drivers
System Stack
System Heap
User Programs
User Data
etc.
I/O Unit
Virtual Memory Interface
Virtual File System Interface
I/O Buffers
Network interface
etc.
Peripheral Devices
etc.

Recap - JOS Boot Sequence

- 0xf000:0xfff0 – 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/main.c

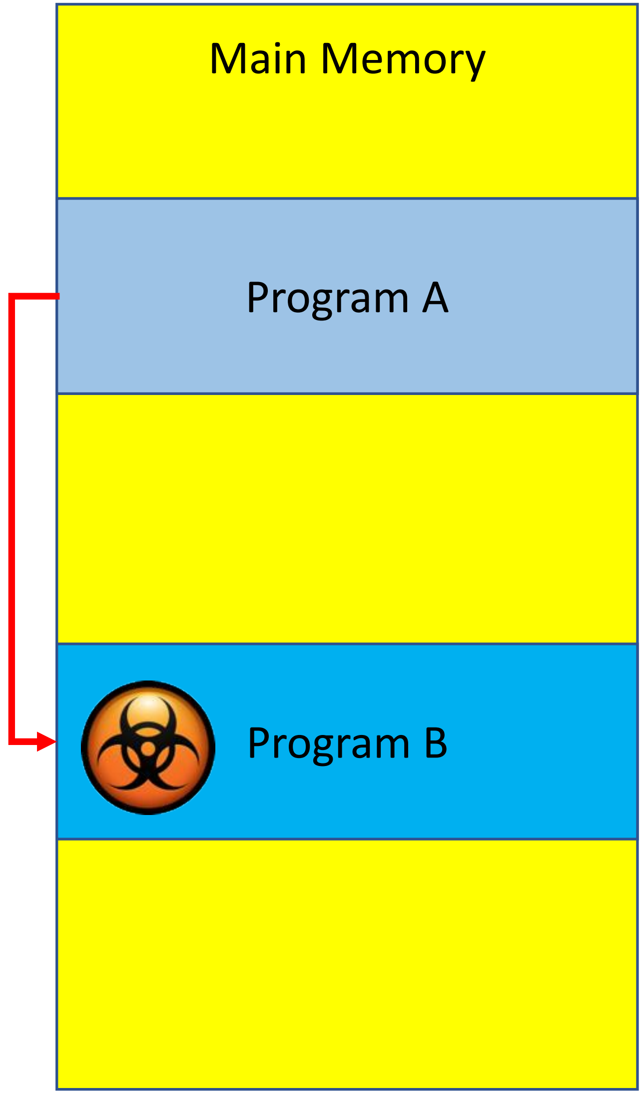
- Read kernel ELF (Executable Linkable Format)
  - https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
  - Load binary program into memory
  - Read header, map memory, copy data…

- Then, run Kernel!

Need for Protected Mode: No Memory Privilege in Real Mode

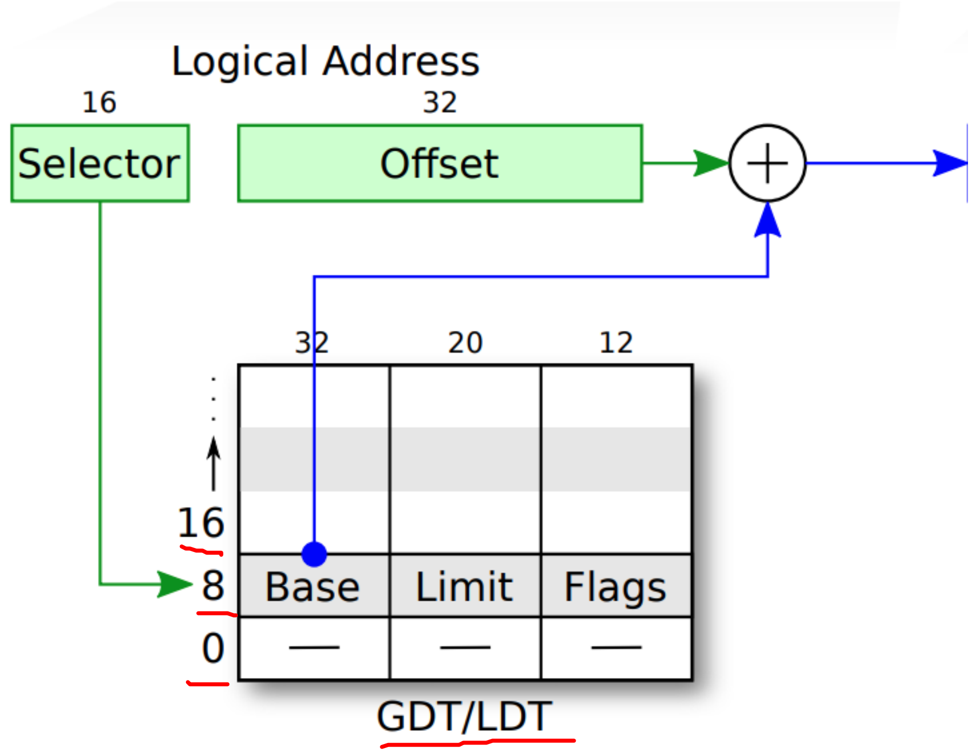
- Suppose two program runs at the same time
  - Program A attempts to modify memory used by program B

  - **No SECURITY!**



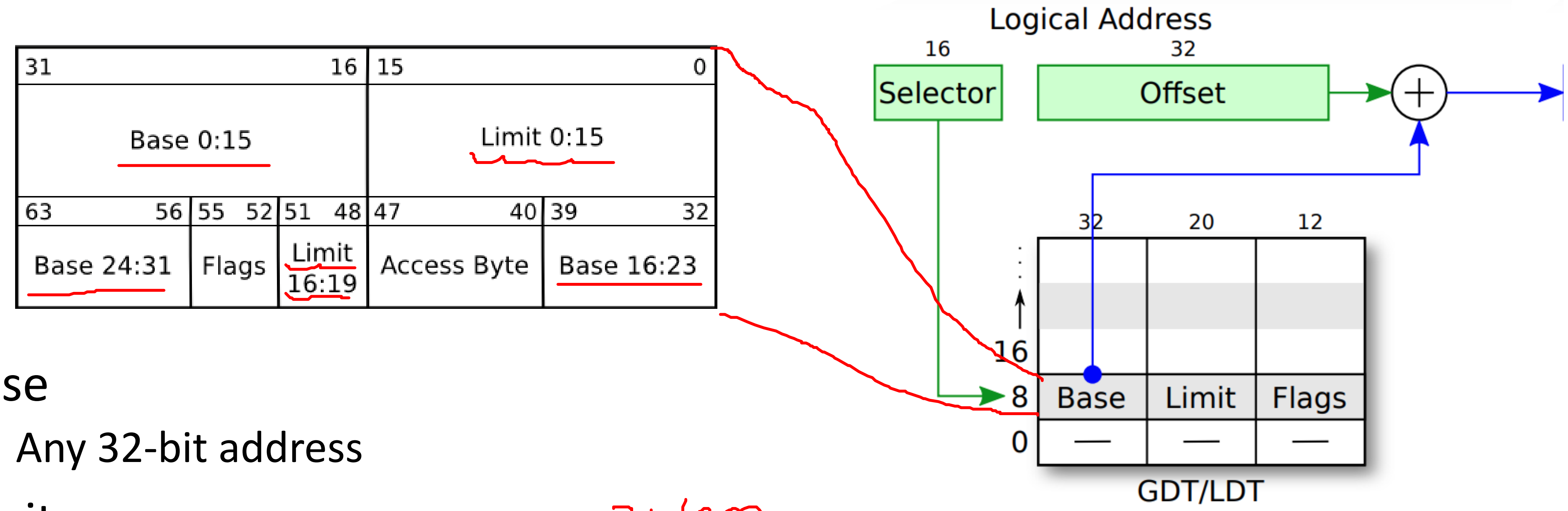
i386 Protected Mode

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



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

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.

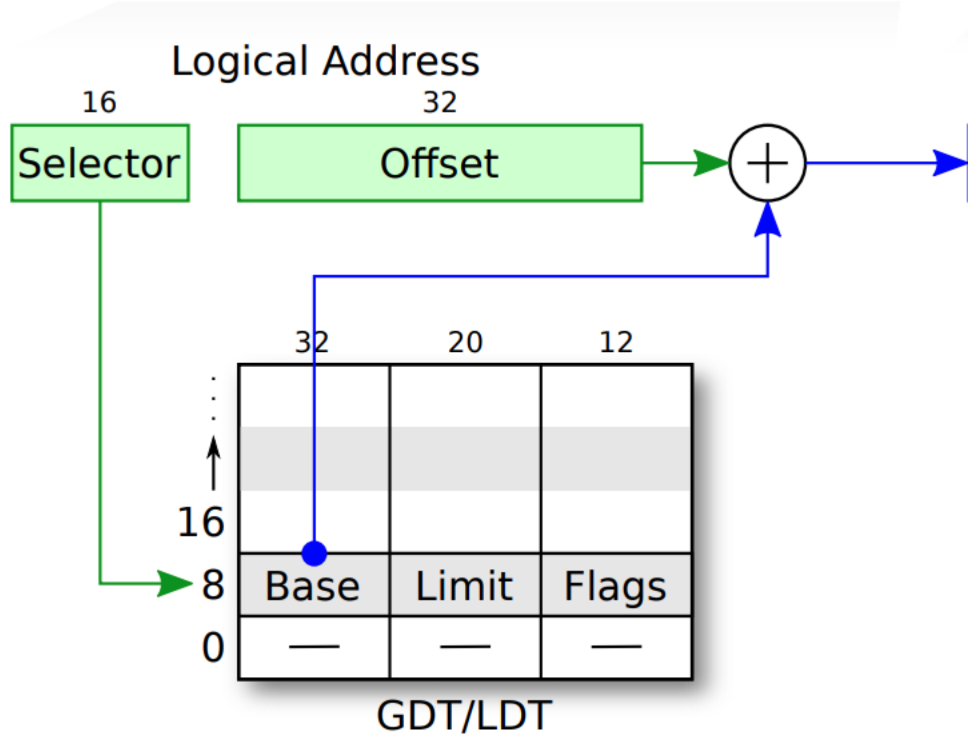
Images from: https://sipb.mit.edu/iap/6.828/files/x86_translation_and_registers.pdf
https://wiki.osdev.org/Global_Descriptor_Table

i386 Protected Mode

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

- Retrieve base address
  - **`Address = base + offset`**

  - Can access **`if (offset < limit)`** **`or`**
  - Can access **`if (offset < limit * 4096)`**
  - **`Depending on the values in flags!`**



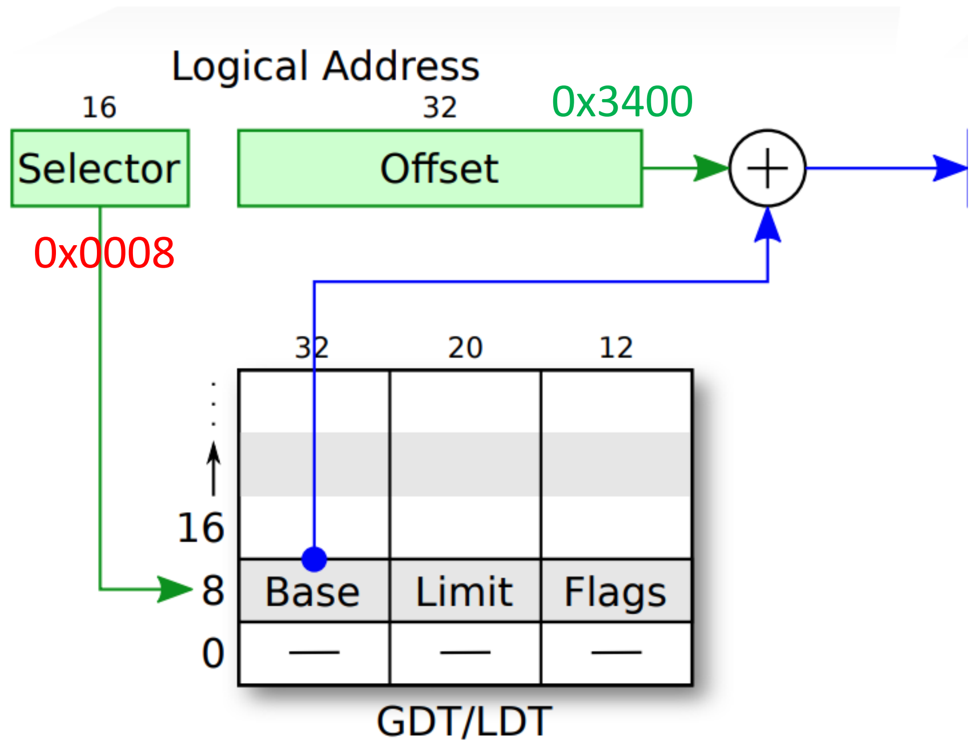
Images from: https://sipb.mit.edu/iap/6.828/files/x86_translation_and_registers.pdf

i386 Protected Mode

- Address 0x0008:0x00003400

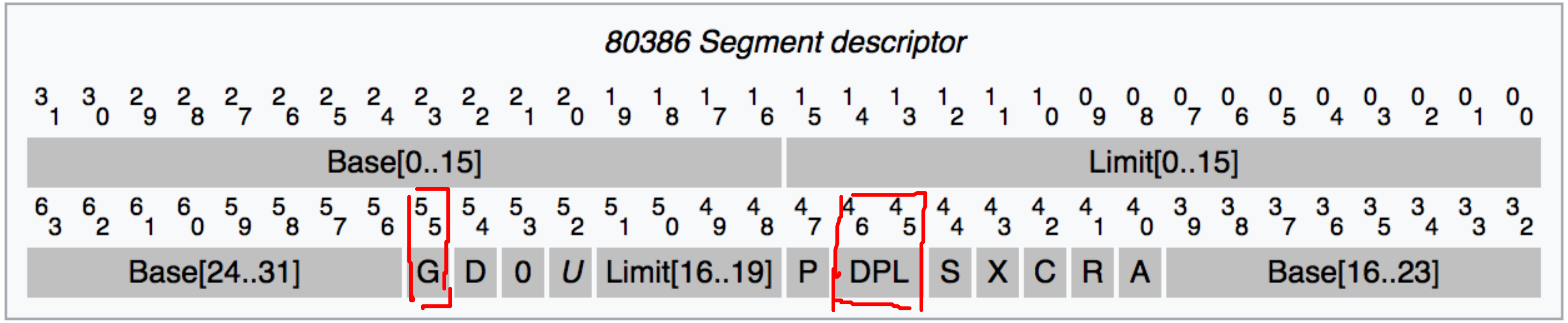
- In the real mode
  - 0x0008 * 16 + 0x3400 = 0x3480

- In the i386 protected mode
  - GDT[1].base + 0x3400
    - Access ok 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

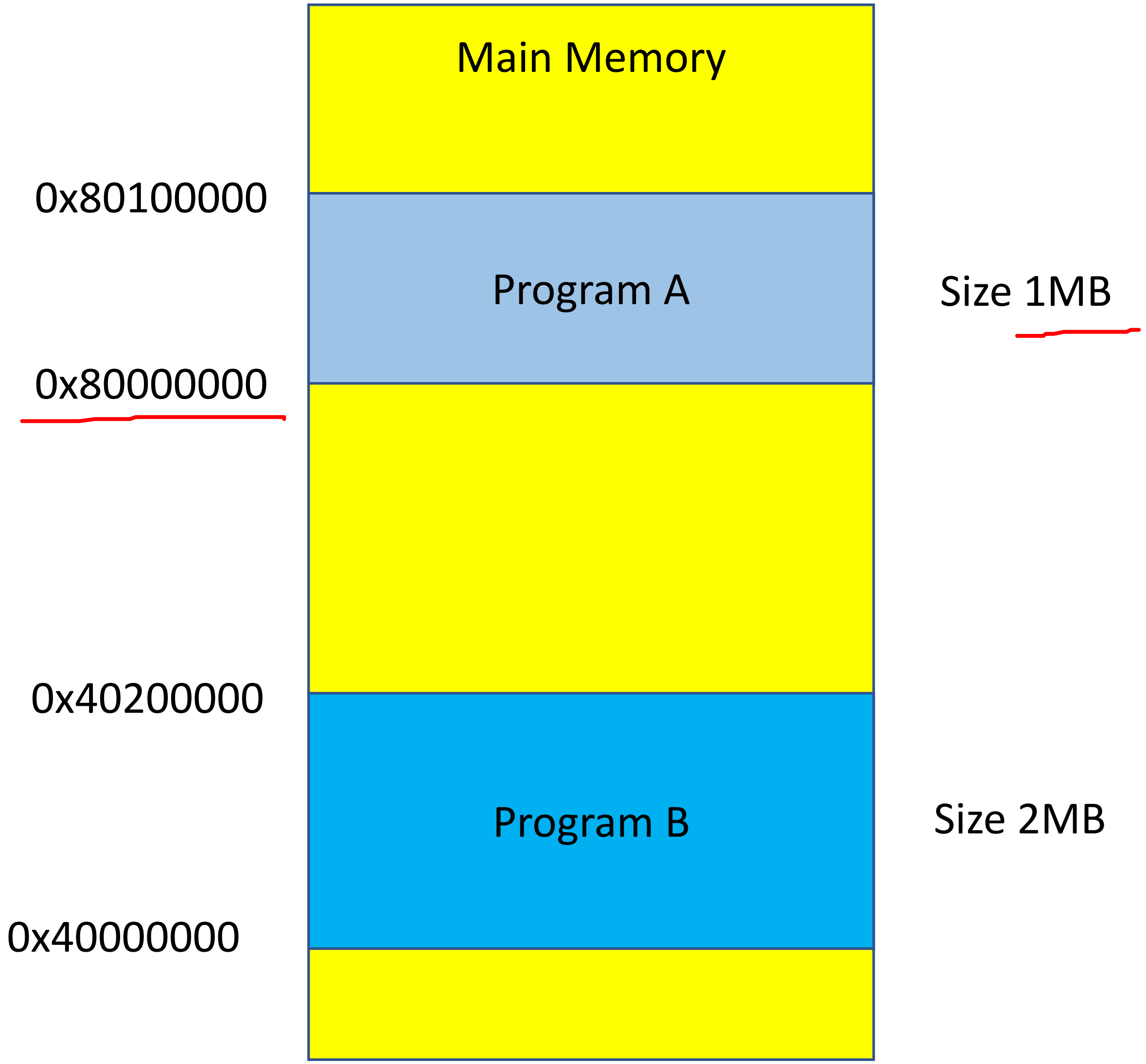
i386 Protected Mode



- 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^{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

A Segment



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

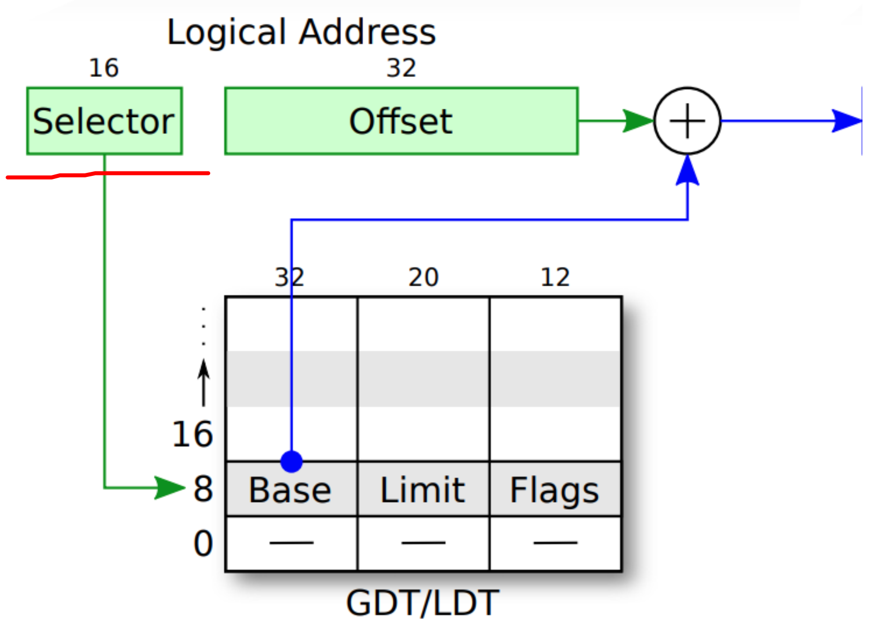
0x08:0 ~ 0x08:0x200000 are valid address for Program B
0x40000000 ~ 0x40200000

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

0x8000 × 4096 (0x1000)



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

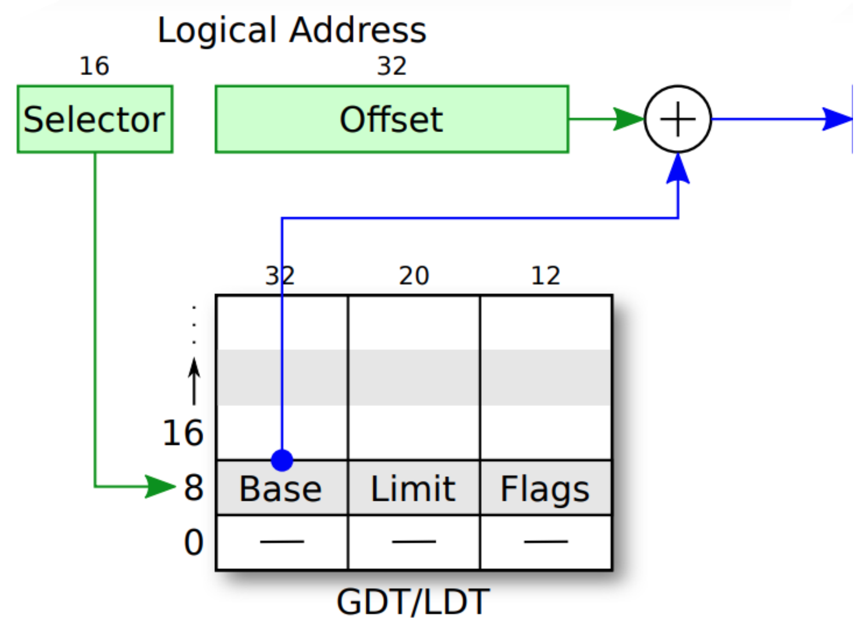
Protected Mode - Examples

16

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

  offset < limit ✓

  - 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…

80386 Segment descriptor

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>09</td><td>08</td><td>07</td><td>06</td><td>05</td><td>04</td><td>03</td><td>02</td><td>01</td><td>00</td></tr>
<tr><td colspan="16">Base[0..15]</td><td colspan="16">Limit[0..15]</td></tr>
<tr><td>63</td><td>62</td><td>61</td><td>60</td><td>59</td><td>58</td><td>57</td><td>56</td><td>55</td><td>54</td><td>53</td><td>52</td><td>51</td><td>50</td><td>49</td><td>48</td><td>47</td><td>46</td><td>45</td><td>44</td><td>43</td><td>42</td><td>41</td><td>40</td><td>39</td><td>38</td><td>37</td><td>36</td><td>35</td><td>34</td><td>33</td><td>32</td></tr>
<tr><td colspan="8">Base[24..31]</td><td>G</td><td>D</td><td>0</td><td>U</td><td colspan="4">Limit[16..19]</td><td>P</td><td colspan="2">DPL</td><td>S</td><td>X</td><td>C</td><td>R</td><td>A</td><td colspan="8">Base[16..23]</td></tr>
</table>

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…

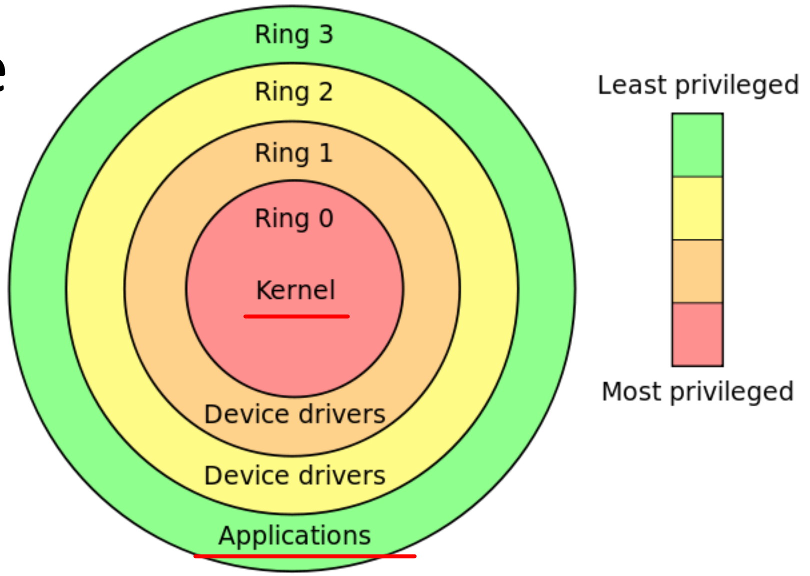


Image from: https://en.wikipedia.org/wiki/File:Priv_rings.svg

Descriptor Privilege Level Defines Ring Level

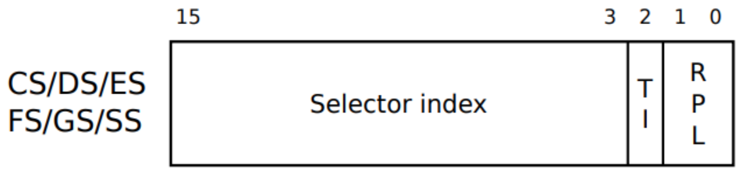
- CPL = Current Privilege Level
  - Defined in the last 2 bits of the %cs register
  - You can change %cs only via `lcall/ljmp/trap/int`

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....



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**

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?

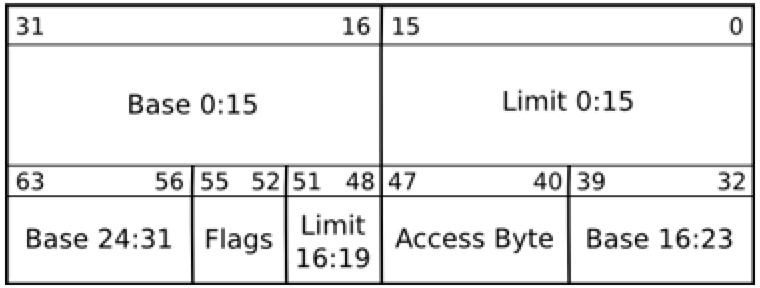
- We can switch from ring 0 to ring 3 via ljmp
  - ljmp 0x3:0x1234

- 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

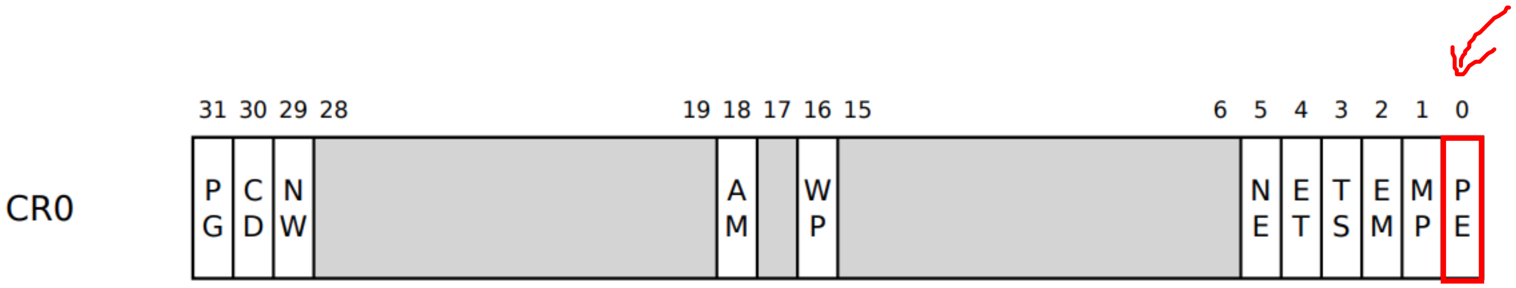



```
# Bootstrap GDT
.p2align 2                                # force 4
gdt:
  SEG_NULL                    # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff) # code seg
  SEG(STA_W, 0x0, 0xffffffff)               # data seg

.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

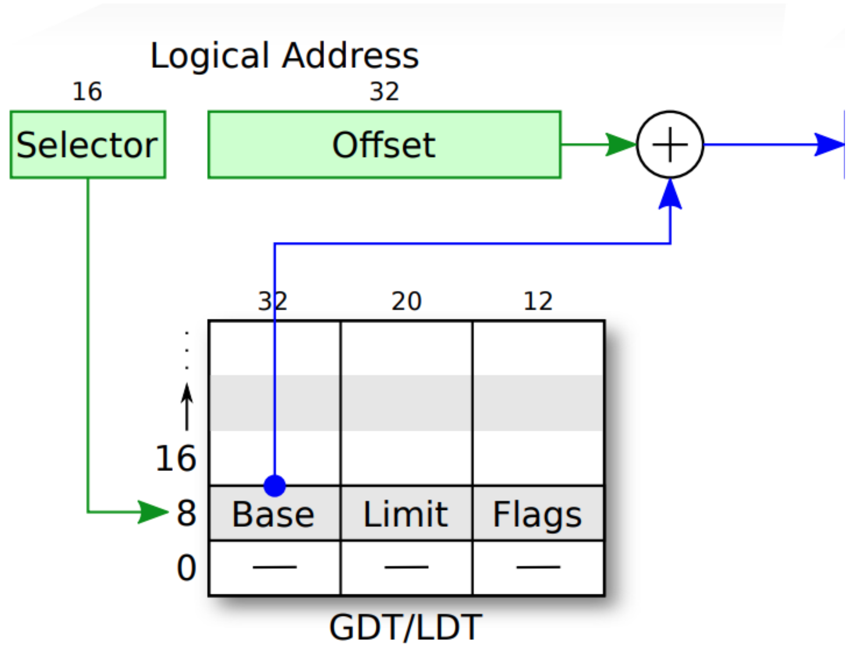
```
# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
```

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 < 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

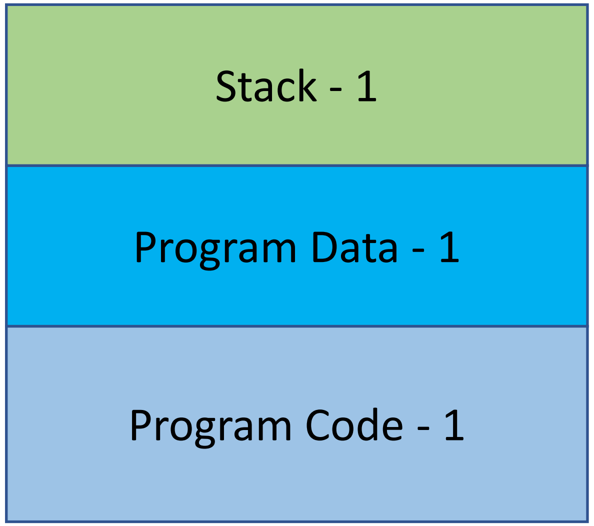


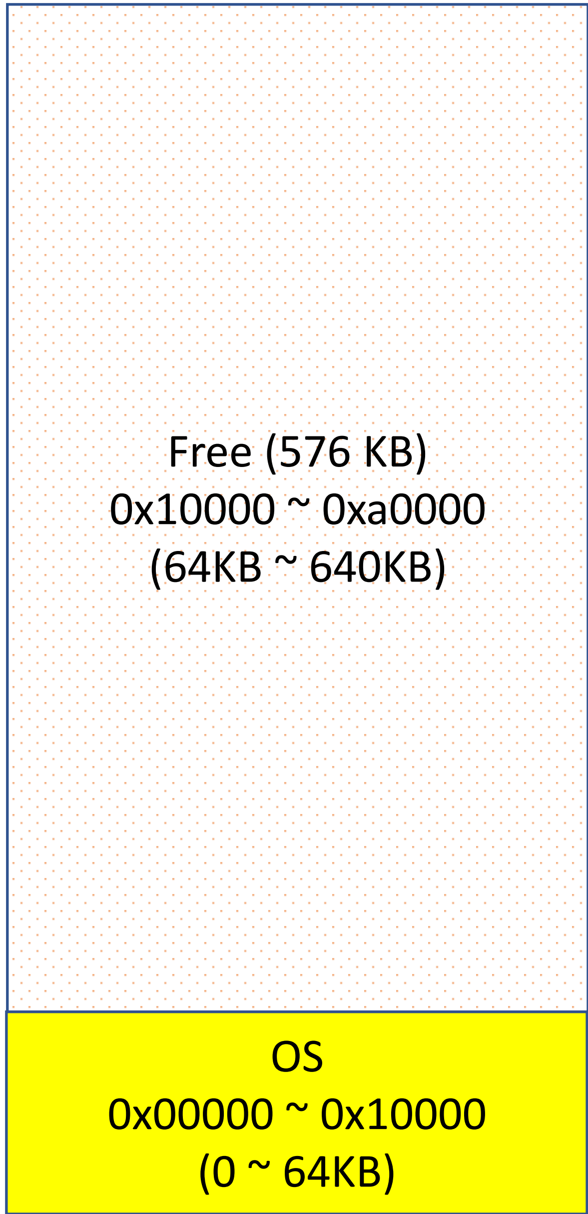
Virtual Memory

- Three goals
  - Transparency
  - Efficiency
  - Protection

Uniprogramming Environment

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

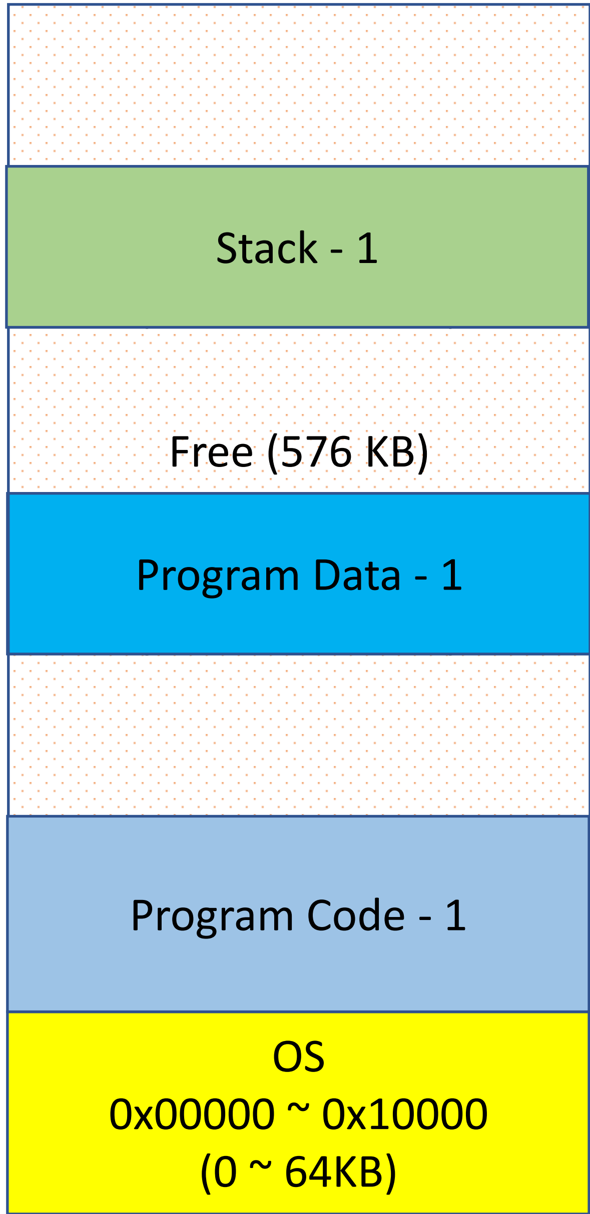




Uniprogramming Environment

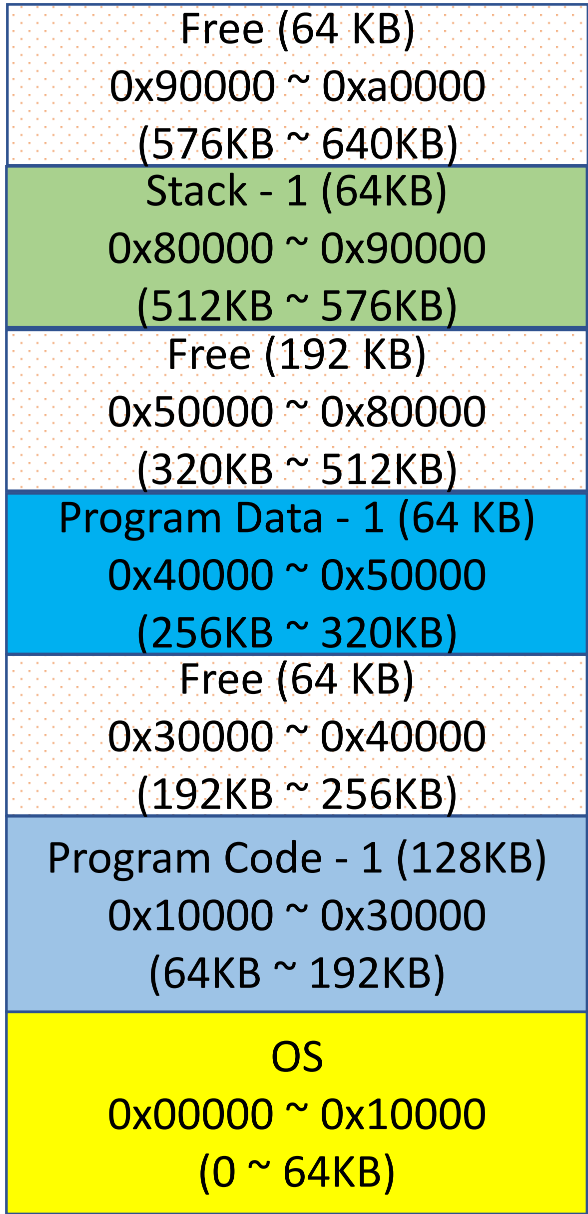
- Run one program

- The program can use memory space freely…



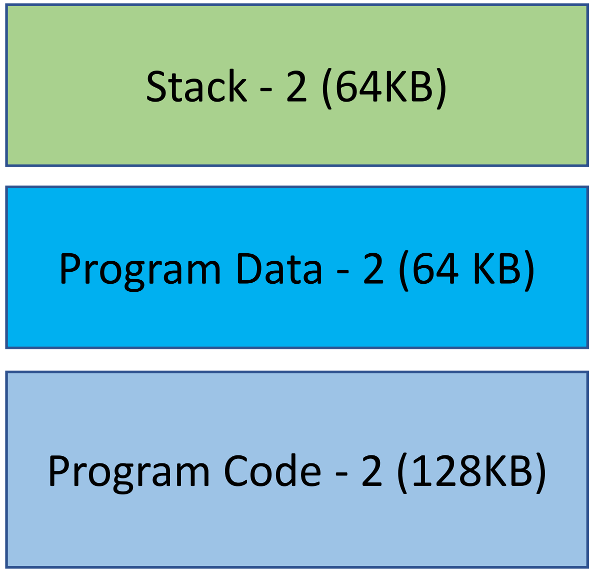
Uniprogramming Environment

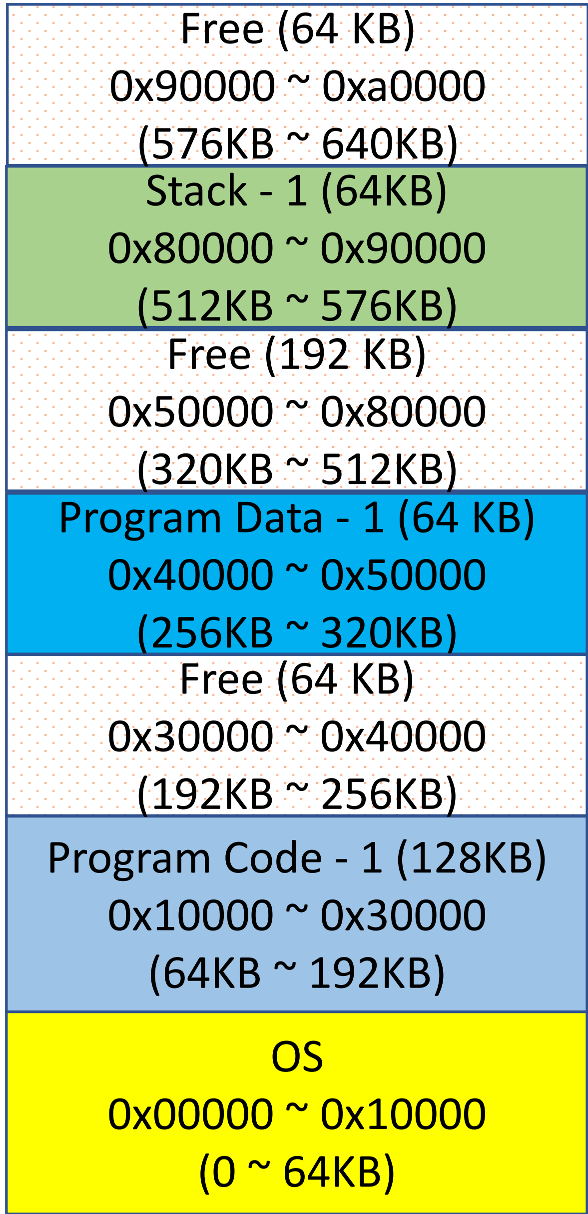
- Run one program
- The program can use memory space freely…



Multi-programming Environment

- Run two programs

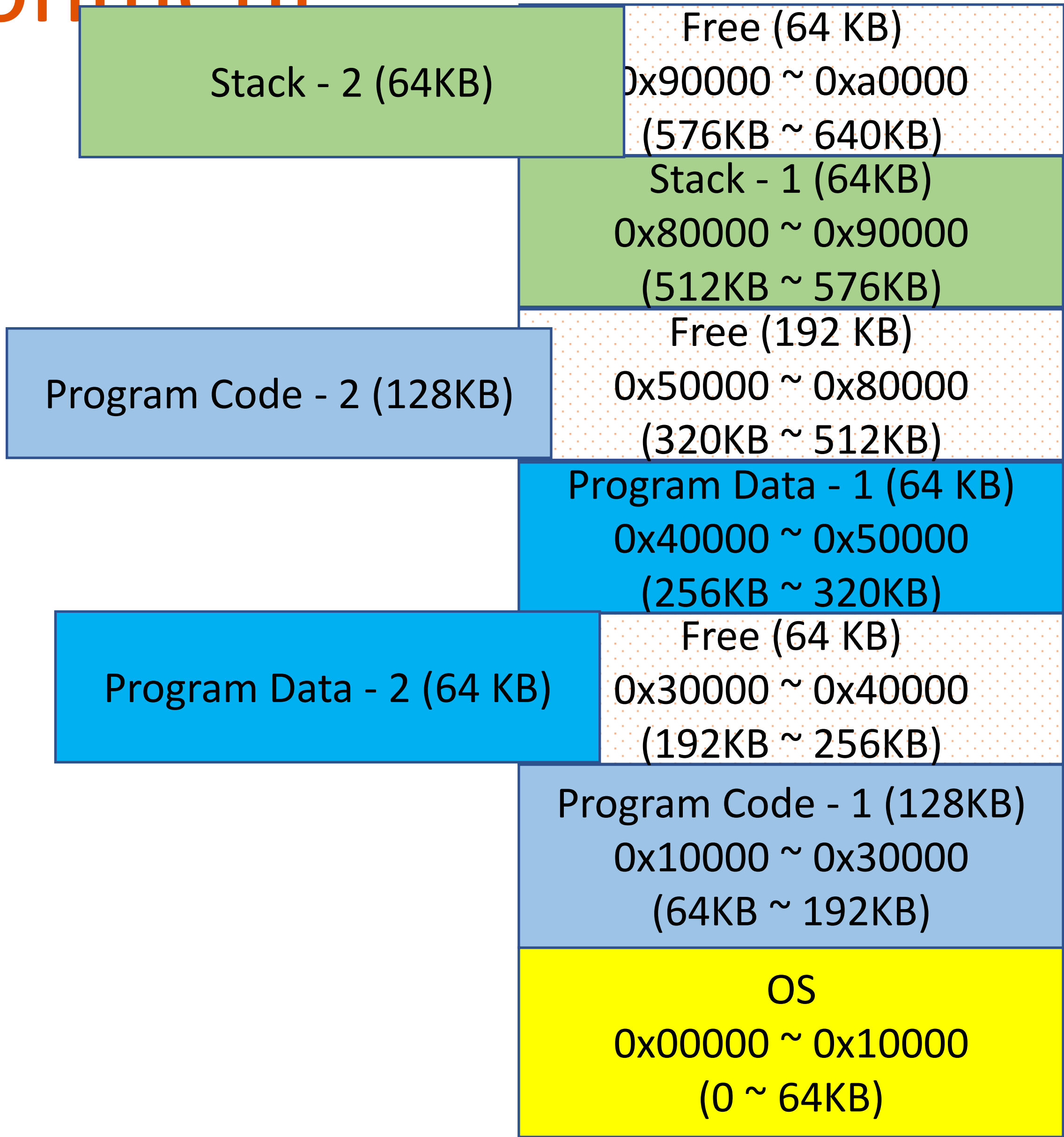




Multi-programming Environment

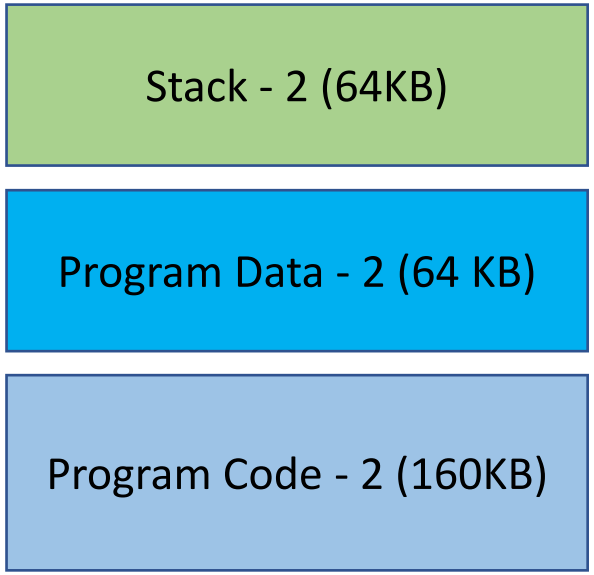
- Run two programs
- System's memory usage determines allocation
- Program need to be aware of the environment
  - Where does system loads my code?
  - You can't determine… system does..

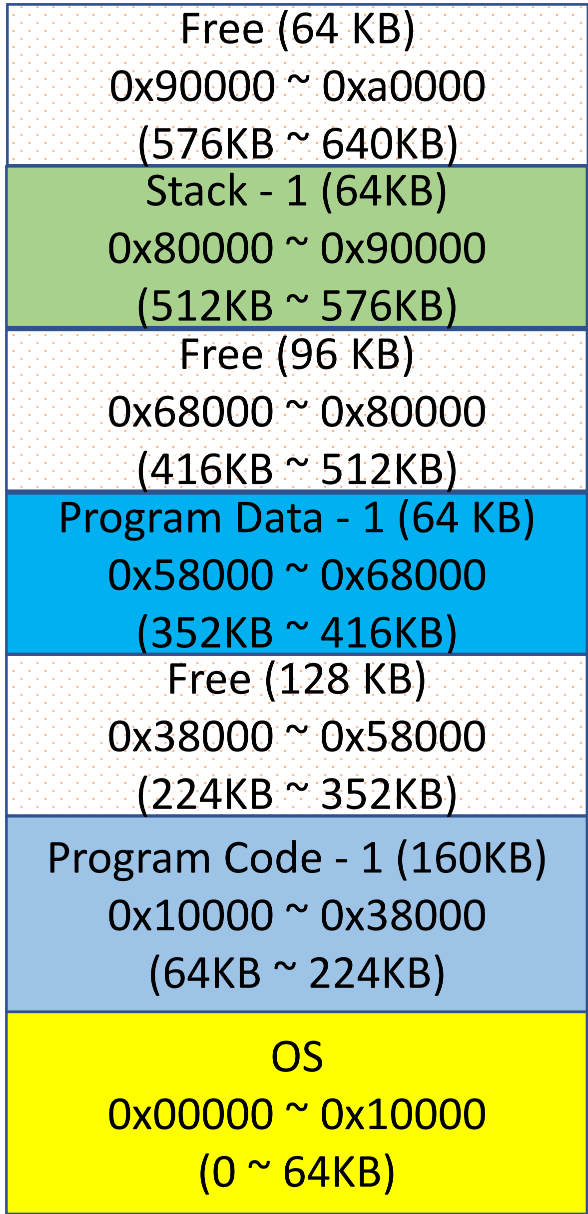
No Transparency…



Multi-programming Environment

- Run two programs

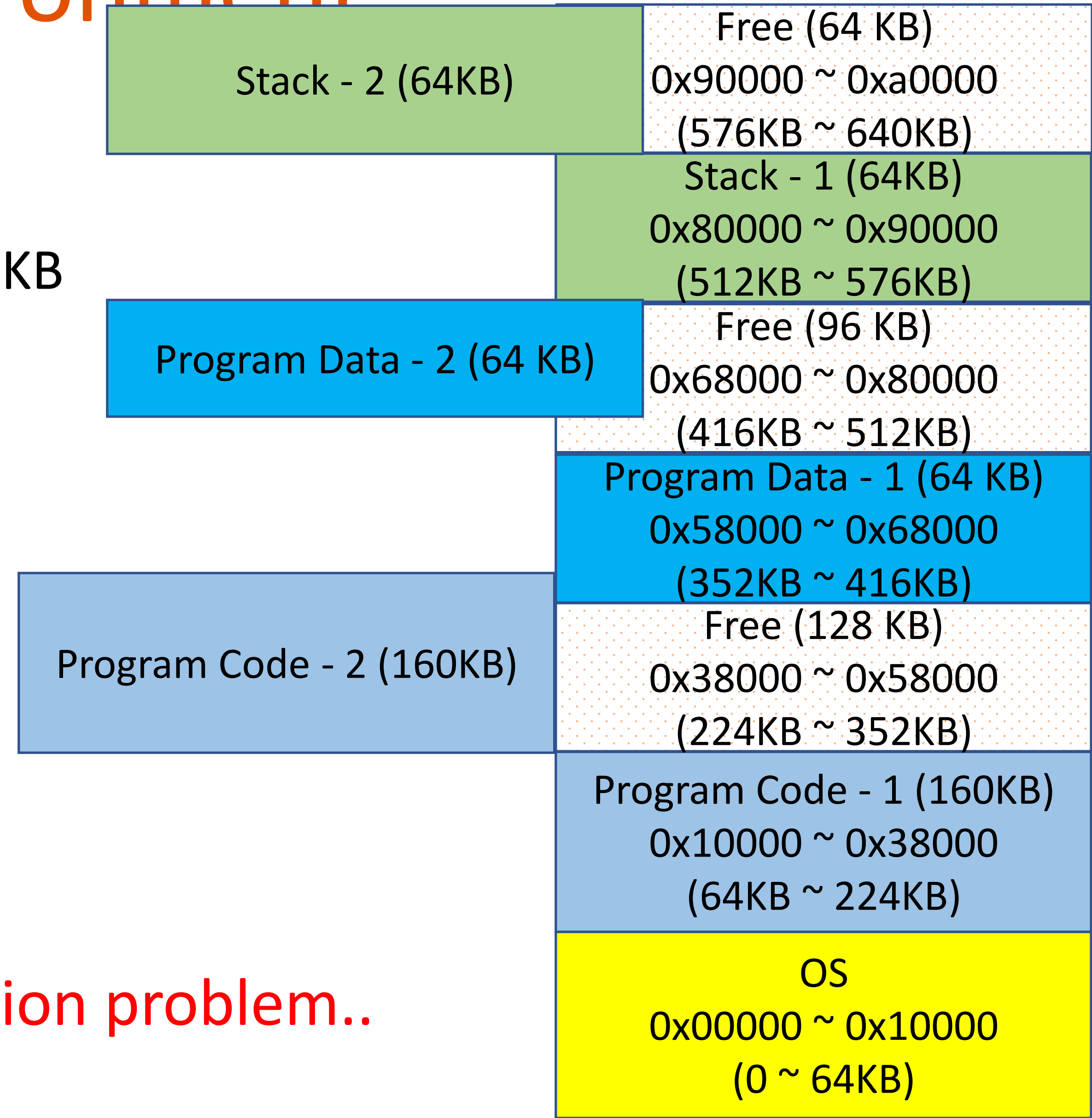




Multi-programming Environment

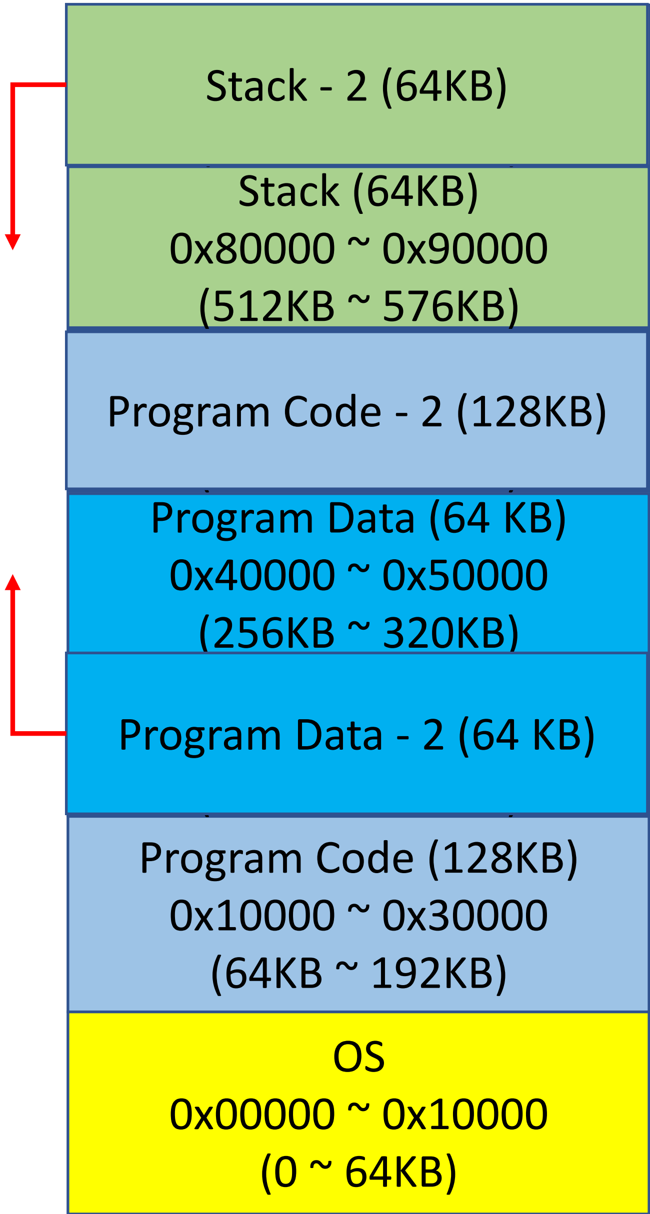
- Run two programs
  - Program size: 64KB + 64KB + 160K = 288KB
- Free mem
  - 64 + 96 + 128 = 288KB
- Cannot run Program – 2
  - Can’t fit…

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

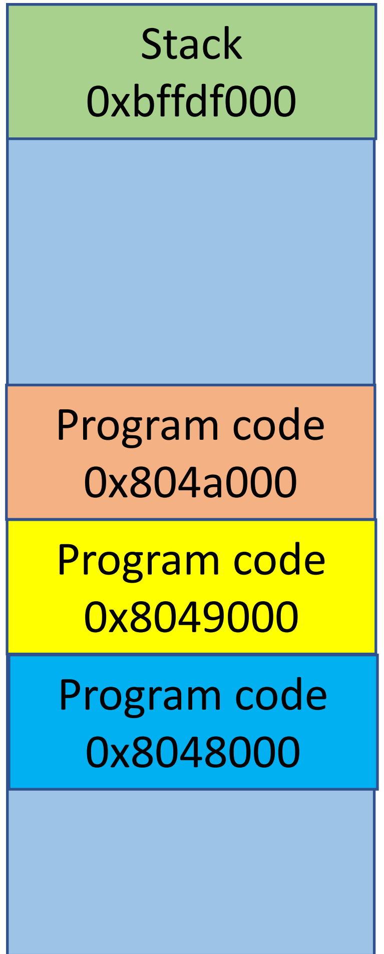
- Three goals
  - Transparency: does not need to know system’s internal state
    - Program A is loaded at 0x8048000. Can Program B be loaded at 0x8048000?
  - Efficiency: do not waste memory; manage memory fragmentation
    - Can Program B (288KB) be loaded if 288 KB of memory is free, regardless of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

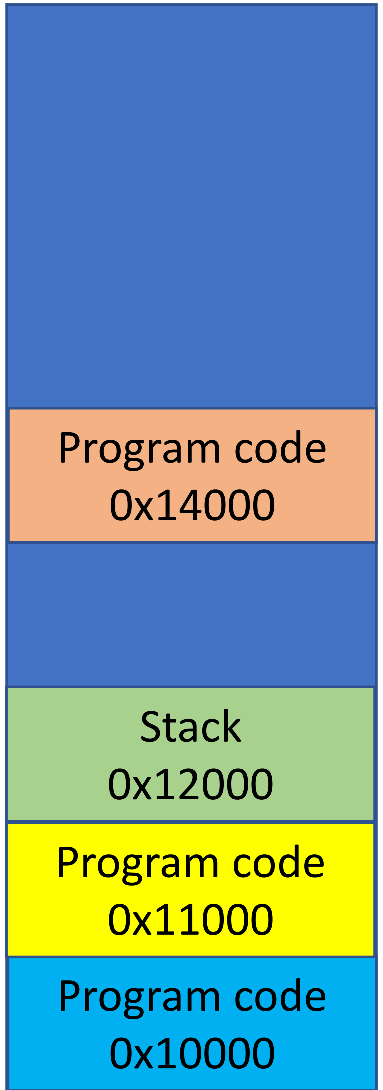
Virtual Memory - Paging

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



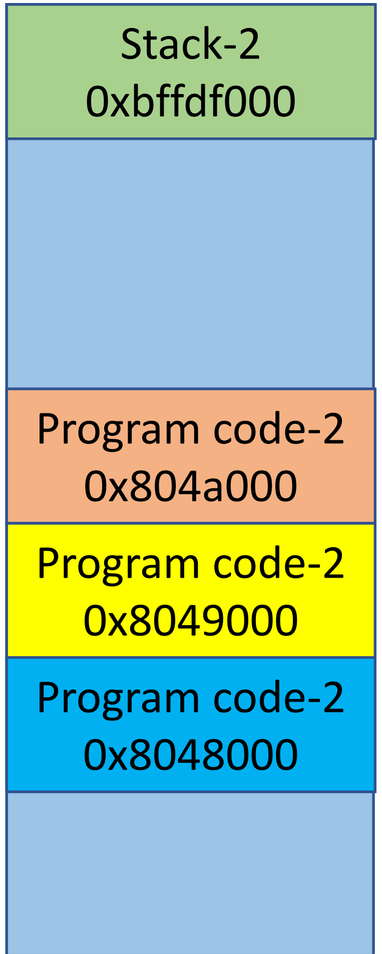
Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
…	…

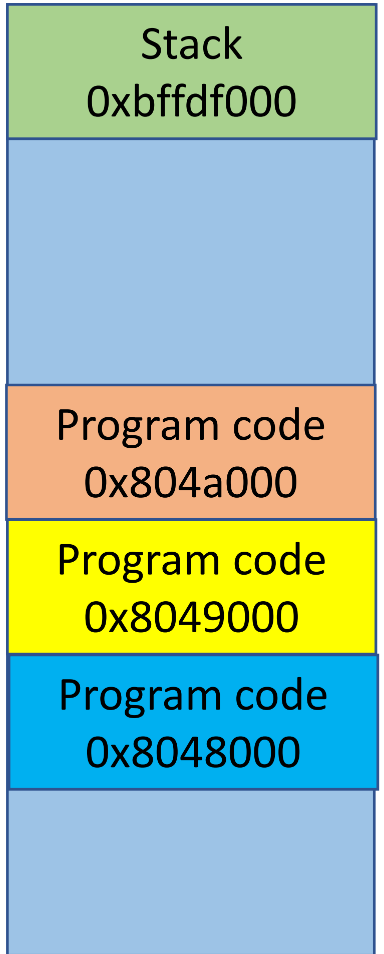
Physical Memory



Paging: Virtual Memory

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

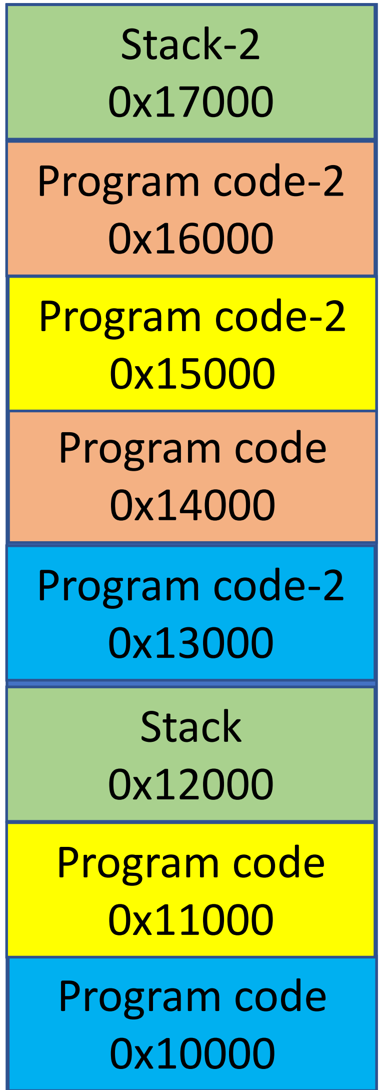




Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
…	…

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

Physical Memory

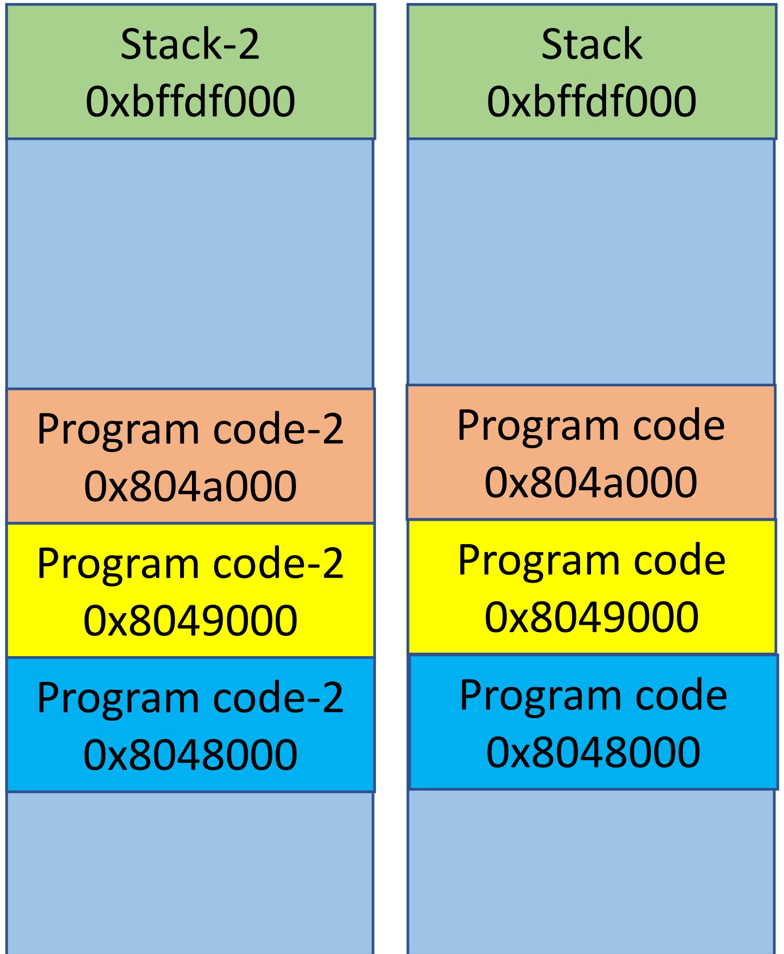


Transparency: does not need to know system's internal state

Program A is loaded at 0x8048000.

Can Program B be loaded at 0x8048000?

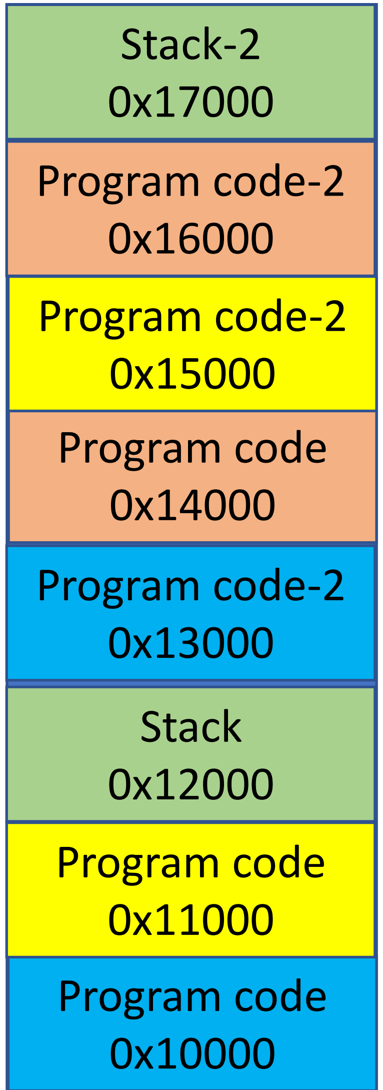
- Having an indirect table that maps virt-addr to phys-addr



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

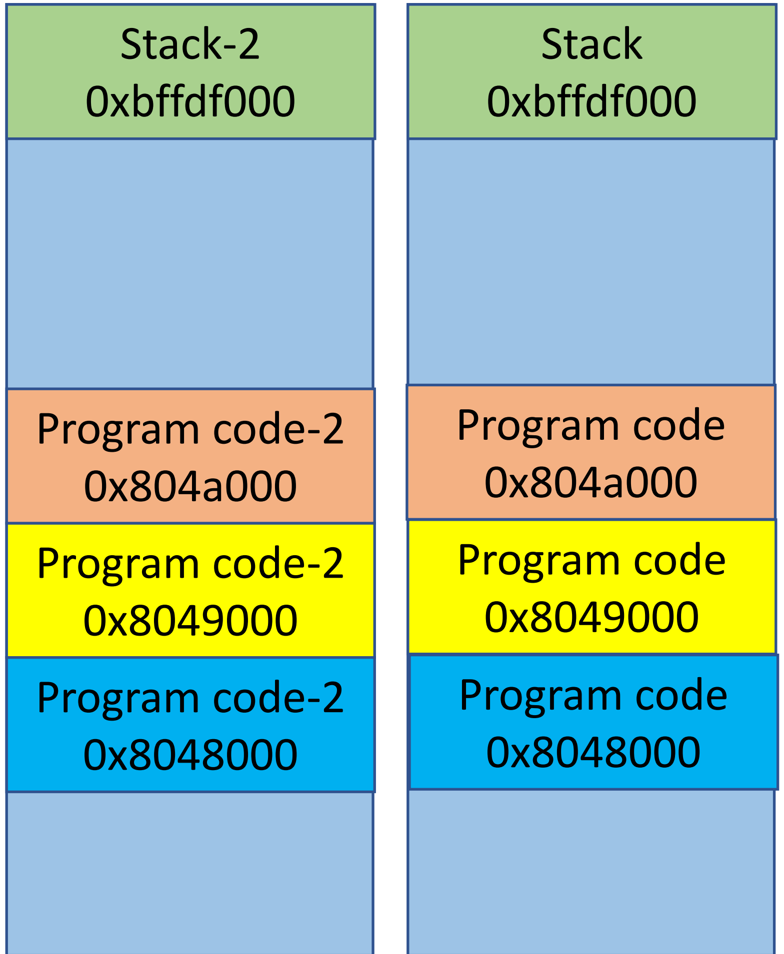
Virtual-2	Physical-2
0x8048000	0x13000
0x8049000	0x15000
0x804a000	0x16000
0xbffdf000	0x17000
…	…

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?

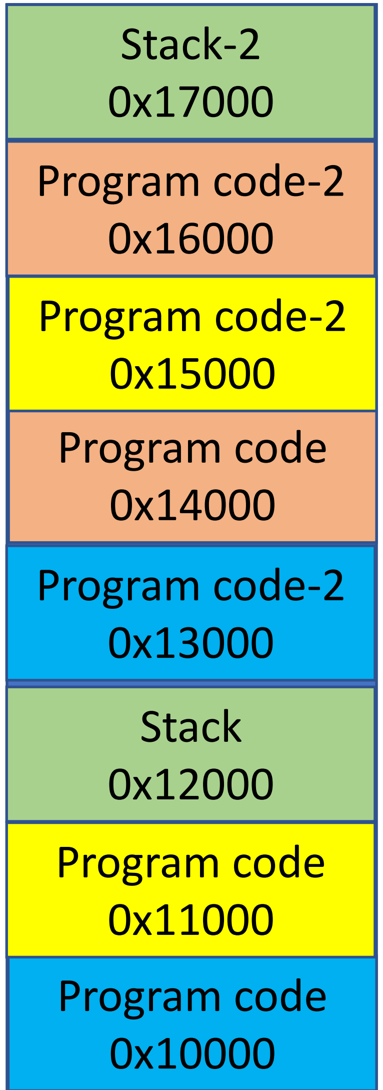
- Having an indirect table that maps virt-addr to phys-addr



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

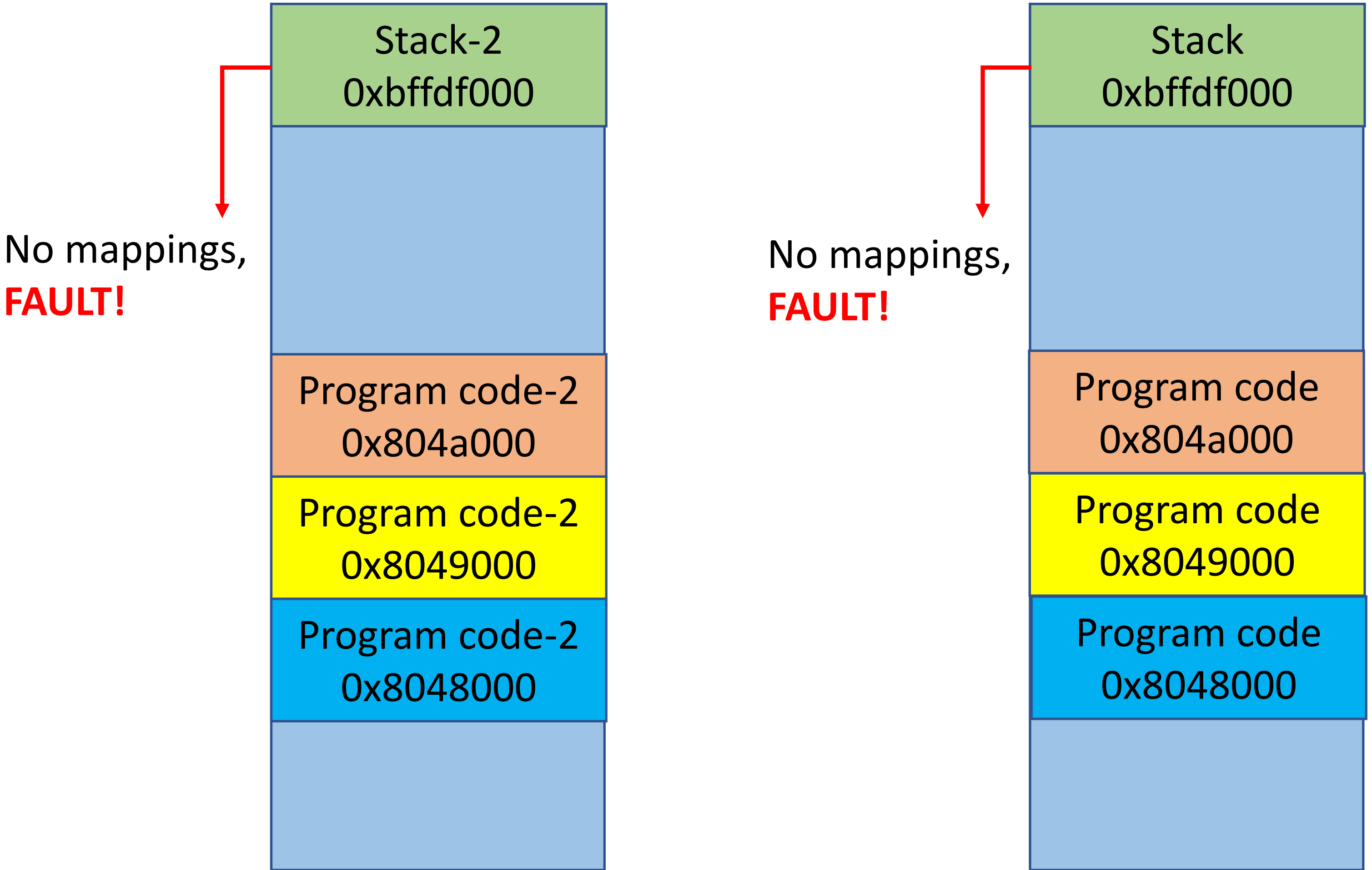
Virtual-2	Physical-2
0x8048000	0x13000
0x8049000	0x15000
0x804a000	0x16000
0xbffdf000	0x17000
…	…

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