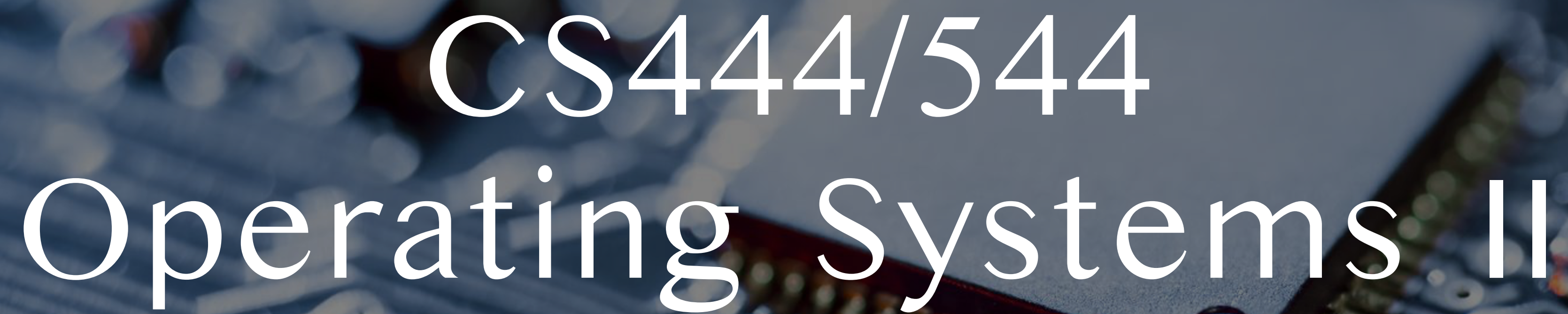


Prof. Sibin Mohan

Spring 2022 | Lec3.2: Memory Extension

Adapted from content originally created by: Prof. Yeongjin Jang

Age Old Problem in Computing | How Much Memory is “**enough**”?

16 bit register

→ need to access 1 MB memory

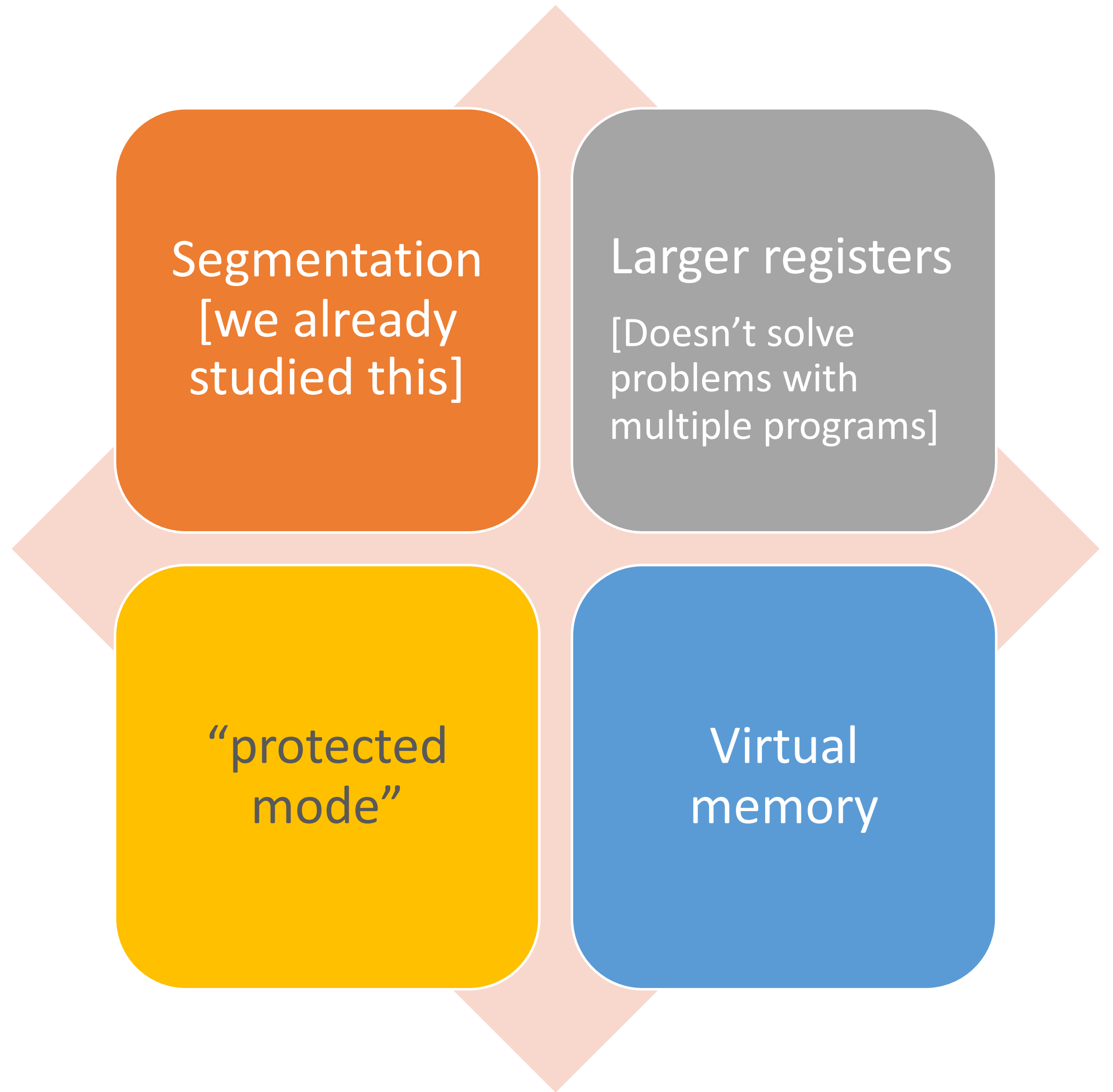
32 bit register!

→ only 16 MB available!

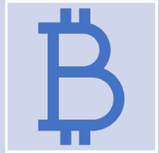
More than 1 Program!

→ All need lots of memory!

Solutions



More about Intel x86 memory



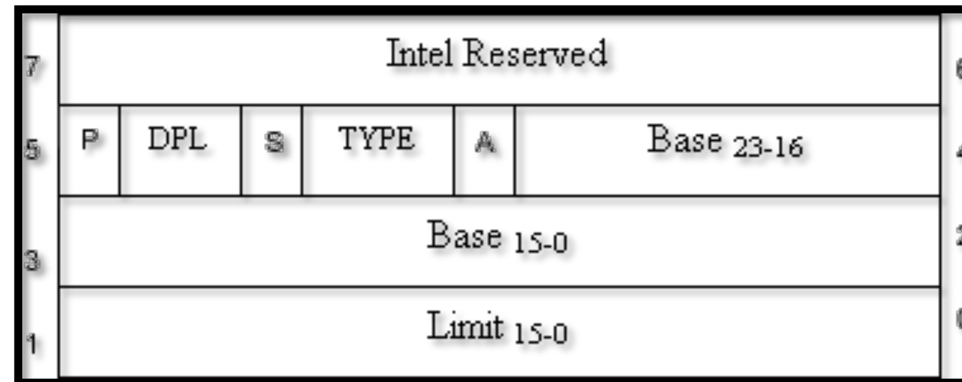
8086 [1978, 16-bit], 8088 [1979, 8-bit and 80186 [1982, 16-bit]

Uses 20-bit addressing
Real Mode segmentation

80286 [1982, **16-bit**]

24-bit (16MB) addressing via ***Protected Mode***
A different way of using segment reg
Segment register points to **Global Descriptor Table** [**GDT**]

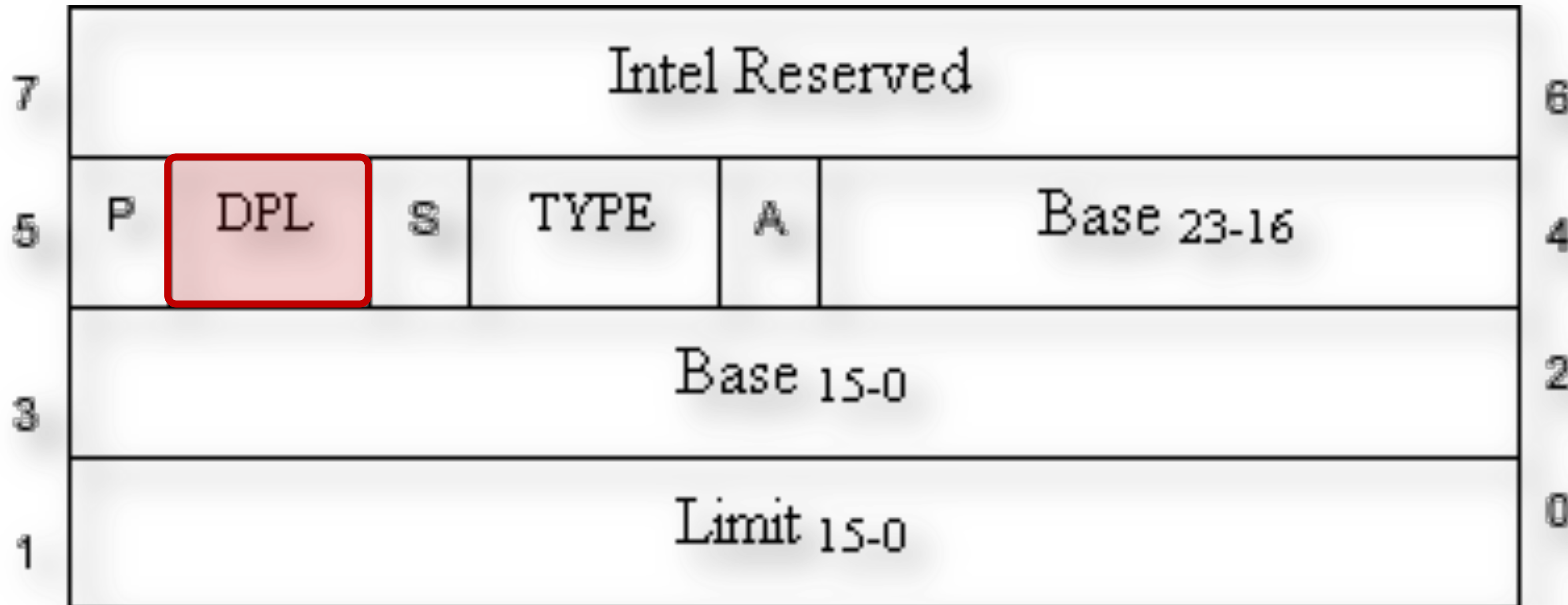
Base (24-bit) + Limit (16-bit)



The picture is from https://nptel.ac.in/courses/117104072/32

Why ‘Protected’?

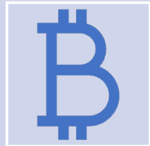
- We can **set memory privilege**!!!!
- **DPL** [Data Privilege Level]



Global Descriptor Table [GDT]

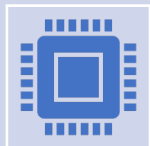
The picture is from https://nptel.ac.in/courses/117104072/32

Enter Intel i386 [80386]



32-bit processor [1985]

Supports **paging** and **virtual memory**



Major computers only equipped with approx. **16 M**B RAM!

i386 Protected Mode

- 32-bit processor, **all registers are 32 bits**
- 2^32 = 4,294,967,295 = **4GB Space**!

- Segment register now points to 32bit base addressable by 32bit offset
- **32bit base + 20bit limit**

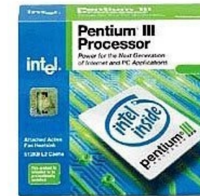
- Supports **paging** [Lab Assignment 2]
  - Virtual Memory

31 16	15 0
Base 0:15	Limit 0:15

63 56	55 52	51 48	47 40	39 32
Base 24:31	Flags	Limit 16:19	Access Byte	Base 16:23

i386 Protected Mode (cont’d)

- 80486, Pentium (P5), Pentium II (i686, P6), Pentium III
    - **Uses the same protected mode as 80386**



- Pentium 4 (Prescott, 2004)
    - Supports **64-bit (amd64)**
    - Address space: **48-bit (256TB)**

- Latest (Coffee Lake and onward)
    - Address space: **57-bit (128PB!!!)**

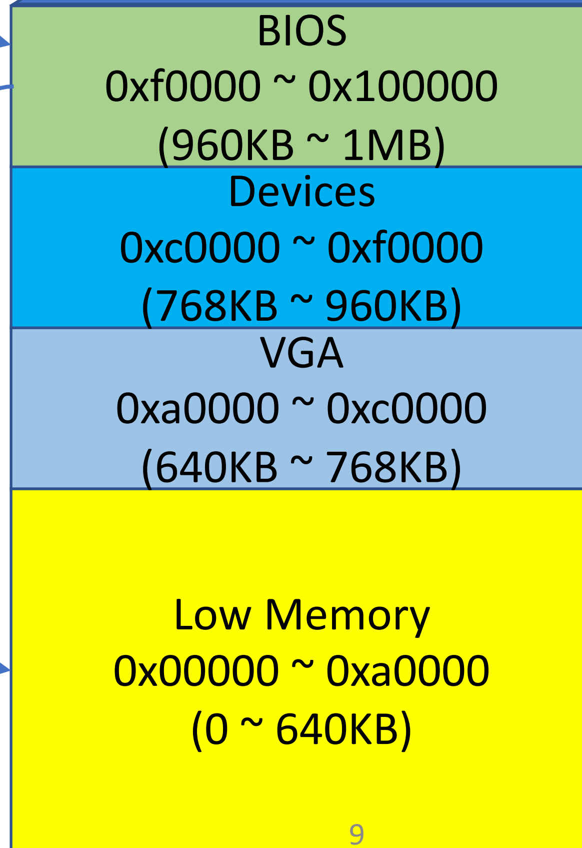
Boot memory layout

4GB for 32bit
256TB for 48bit on amd64
128PB for 57bit on amd64

Extended Memory
(Over 1MB)

Load kernel and run!

Map code in BIOS at
`f000:fff0`



Enabling Protected Mode

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

Recap | JOS Boot Sequence

0xf000:0xfff0 → BIOS

`0xf000` * 16 + `0xfff0` = `0xffff0` [REAL MODE]

Loads boot sector → runs `0x7c00`

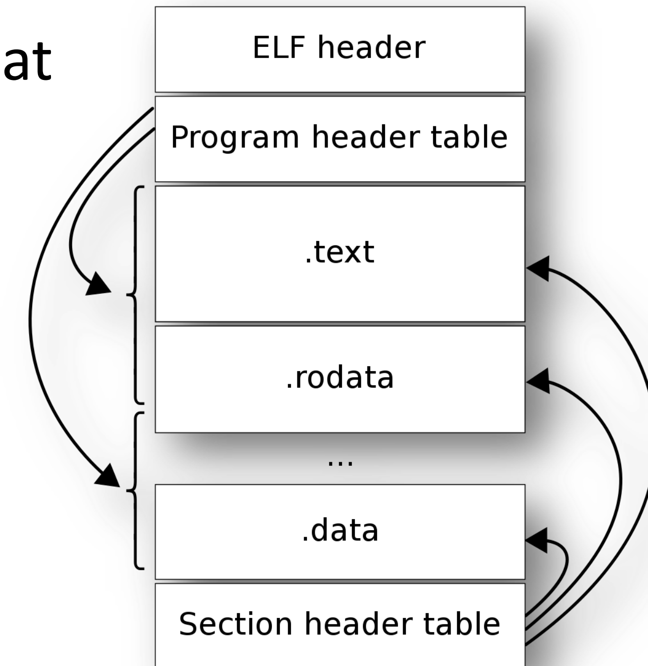
Enable **A20**

Enable protected mode → 4GB memory access

Read kernel ELF [Executable Linkable Format]

Brief Detour | ELF

- Executable and Linkable Format
- Common file format for
  - executable files, shared libraries, object code, core dumps
- Chosen as standard format
  - for Unix and Unix-like systems
  - for x86 systems
  - 1999



ELF101 a Linux executable walkthrough

ANGE ALBERTINI
CORKAMI.COM

DISSECTED FILE

```
~$ uname -m
armv7l
~$ ./simple.ARM
Hello World!
```

SIMPLE.ARM

HEADER 1/2 — TECHNICAL DETAILS FOR IDENTIFICATION AND EXECUTION

- ELF HEADER — IDENTIFY AS AN ELF TYPE, SPECIFY THE ARCHITECTURE
- PROGRAM HEADER TABLE — EXECUTION INFORMATION

SECTIONS — CONTENTS OF THE EXECUTABLE

- CODE — EXECUTABLE INFORMATION
- DATA — INFORMATION USED BY THE CODE
- SECTIONS' NAMES

HEADER 2/2 — TECHNICAL DETAILS FOR LINKING (IGNORED FOR EXECUTION)

- SECTION HEADER TABLE

HEXADECIMAL DUMP · ASCII DUMP · FIELDS · VALUES · EXPLANATION

FIELDS	VALUES	EXPLANATION
e_ident		
EI_MAG	0x7F, "ELF"	CONSTANT SIGNATURE
EI_CLASS, EI_DATA	1, 1	32 BITS, LITTLE-ENDIAN
EI_VERSION	1	ALWAYS 1
e_type	2	EXECUTABLE
e_machine	28	ARM PROCESSOR
e_version	1	ALWAYS 1
e_entry	0x8000060	ADDRESS WHERE EXECUTION STARTS
e_phoff	0x40	PROGRAM HEADERS' OFFSET
e_shoff	0x80	SECTION HEADERS' OFFSET
e_ehsize	0x34	ELF HEADER'S SIZE
e_phentsize	0x20	SIZE OF A SINGLE PROGRAM HEADER
e_phnum	1	COUNT OF PROGRAM HEADERS
e_shentsize	0x28	SIZE OF A SINGLE SECTION HEADER
e_shnum	4	COUNT OF SECTION HEADERS
e_shstrndx	3	INDEX OF THE NAMES' SECTION IN THE TABLE
p_type	1	THE SEGMENT SHOULD BE LOADED IN MEMORY
p_offset	0	OFFSET WHERE IT SHOULD BE READ
p_vaddr	0x8000000	VIRTUAL ADDRESS WHERE IT SHOULD BE LOADED
p_paddr	0x8000000	PHYSICAL ADDRESS WHERE IT SHOULD BE LOADED
p_filesz	0x90	SIZE ON FILE
p_memsz	0x90	SIZE IN MEMORY
p_flags	5	READABLE AND EXECUTABLE

ARM ASSEMBLY · EQUIVALENT C CODE

```
mov r2, #13
add r1, pc, #20
mov r0, #1
mov r7, #4
svc 0          write(STDOUT_FILENO, "Hello World!\n", len("Hello World!\n"));

mov r0, #1
mov r7, #1
svc 0          exit(1);
```

STRINGS: "Hello World!\n", 0

SECTION NAMES: "" .shrtrtab .text .rodata

SECTION HEADER TABLE

INDEX	NAME	TYPE	FLAGS	ADDRESS	OFFSET	SIZE
0	<null>	0				
1	.text	1	6	0x8000060	0x60	0x20
2	.rodata	1	2	0x8000080	0x80	0x0D
3	.shrtrtab	3			0x90	0x19

THIS IS THE WHOLE FILE. HOWEVER, MOST ELF FILES CONTAIN MANY MORE ELEMENTS. EXPLANATIONS ARE SIMPLIFIED, FOR CONCISENESS.

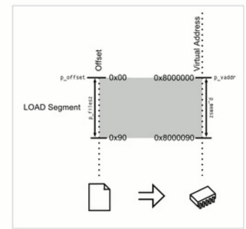
LOADING PROCESS

1 HEADER

THE ELF HEADER IS PARSED
THE PROGRAM HEADER IS PARSED
(SECTIONS ARE NOT USED)

2 MAPPING

THE FILE IS MAPPED IN MEMORY
ACCORDING TO ITS SEGMENT(S)



3 EXECUTION

ENTRY IS CALLED
SYSCALLS ARE ACCESSED VIA:
- SYSCALL NUMBER IN THE R7 REGISTER
- CALLING INSTRUCTION SVC

TRIVIA

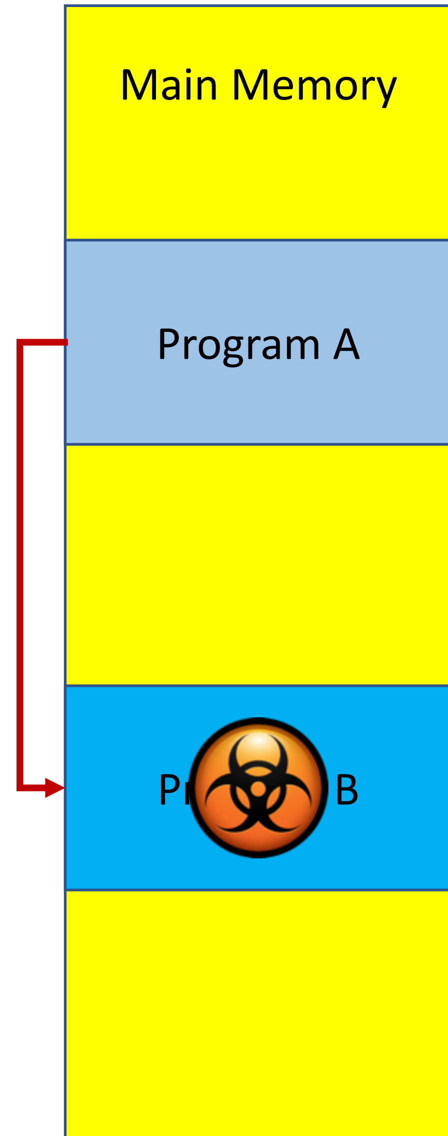
THE ELF WAS FIRST SPECIFIED BY U.S. L. AND U.I.
FOR UNIX SYSTEM V, IN 1989

THE ELF IS USED, AMONG OTHERS, IN:
- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS

VERSION 1.0A
2013/12/06
CC BY

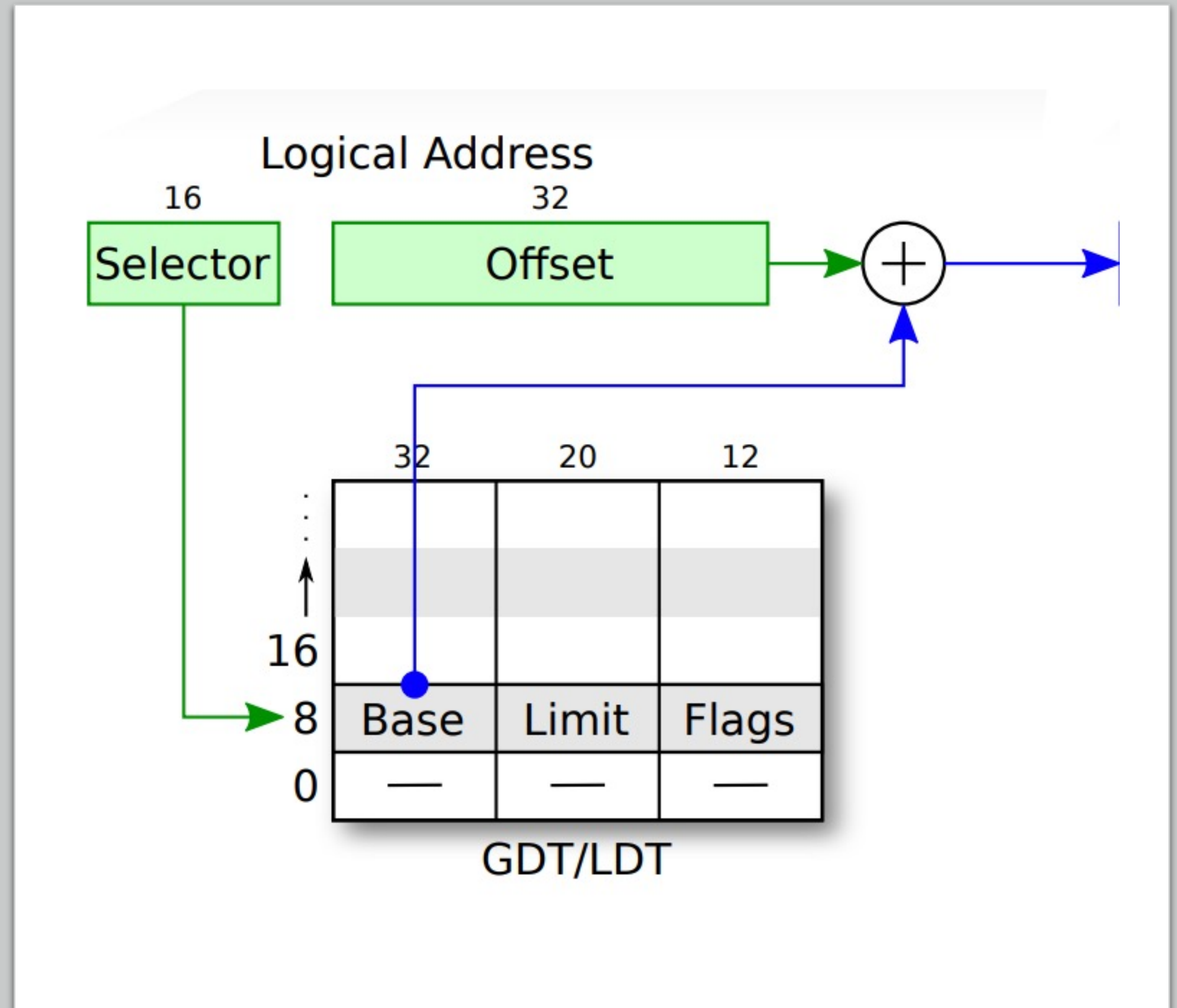
Back to Regularly Scheduled Programming

- Why do we need a “**protected**” mode to start with?
- Extending the addressing, duh! [didn’t we already go over this?]
- **Well, yes and no!** [also, why is it called the protected mode?]
- Suppose two (or more) programs run at the same time
  - Program A attempts to modify memory used by program B
- **No Protection!**



i386 Protected Mode

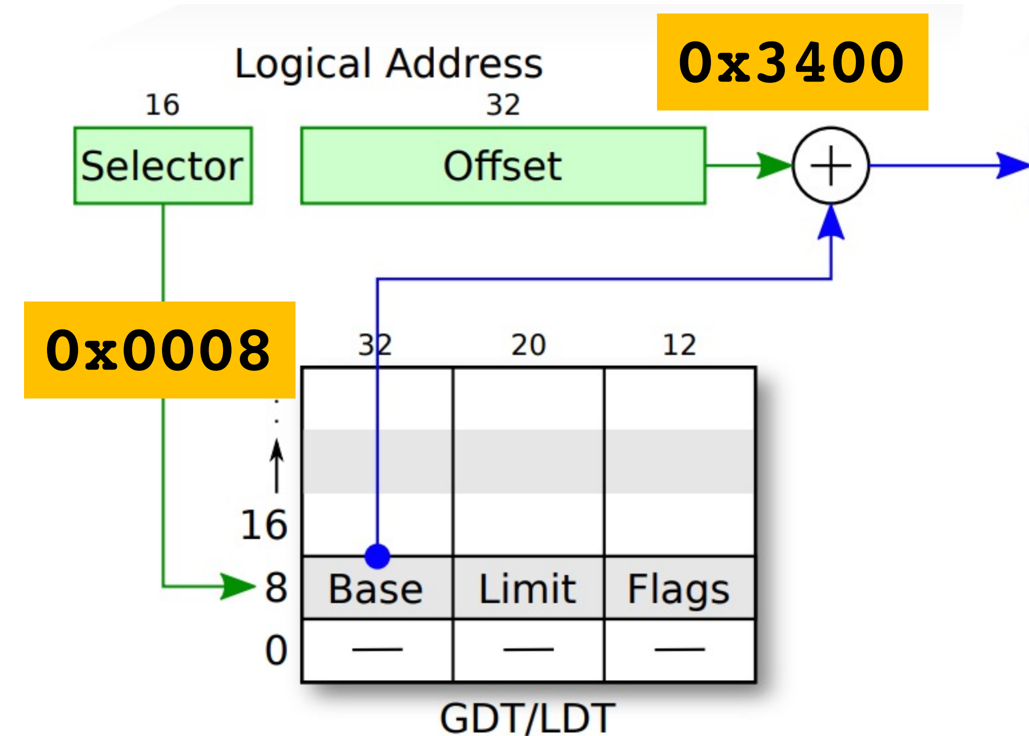
- **GDT [Global Descriptor Table]**
  - Indexed by a segment register



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

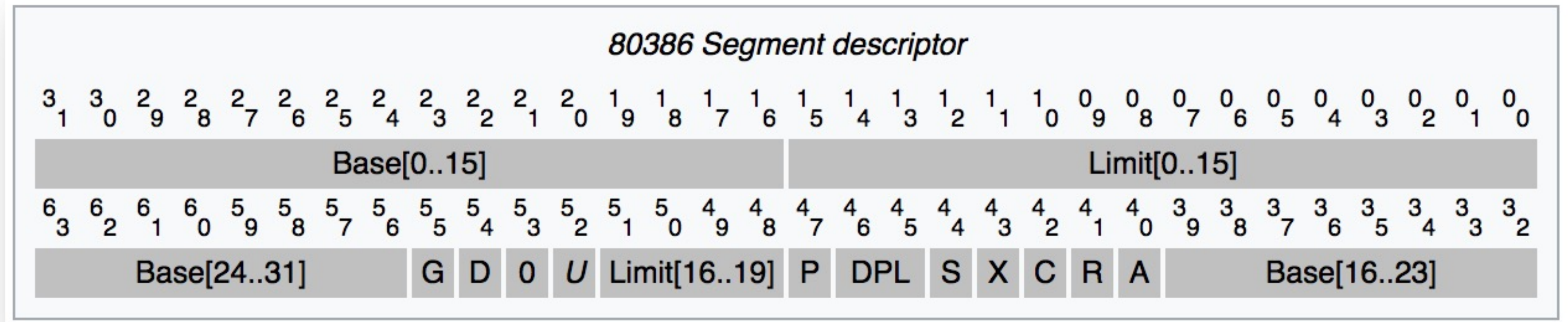
i386 Protected Mode | Example

- 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

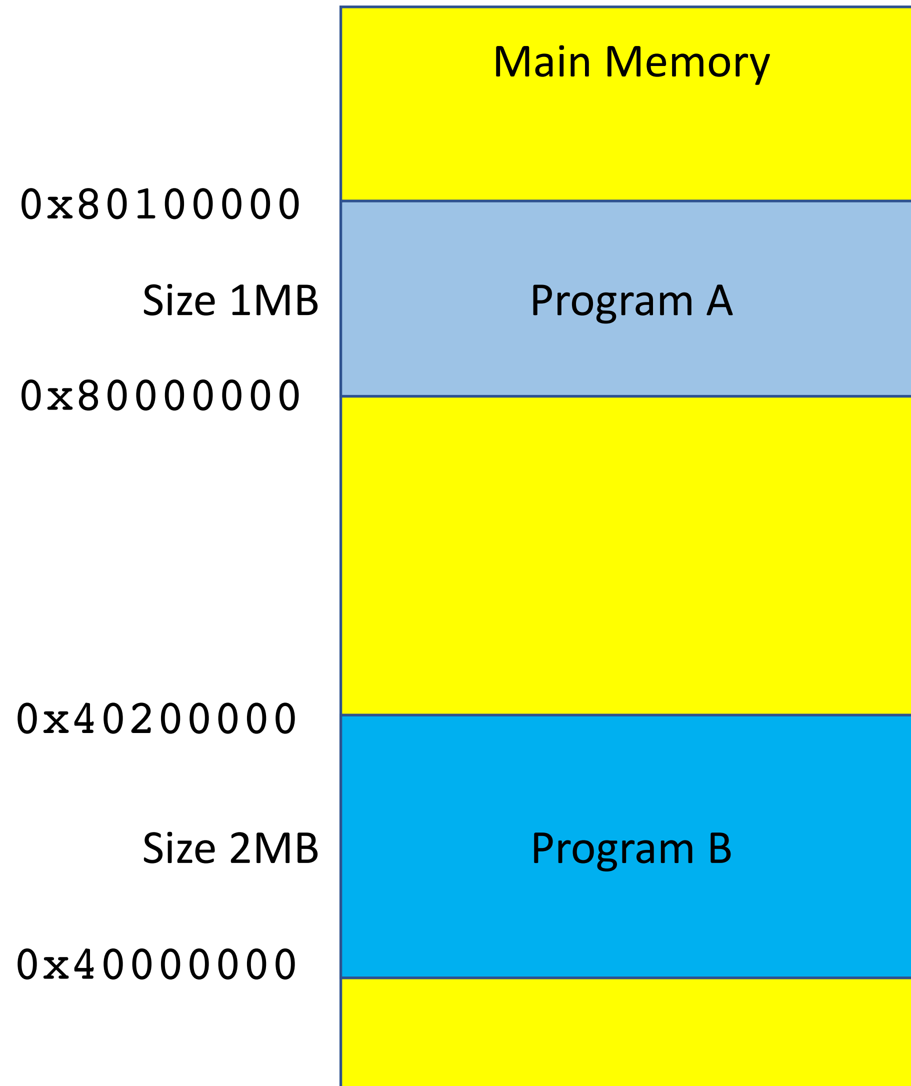
R,X → **Readable/Executable**

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



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

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

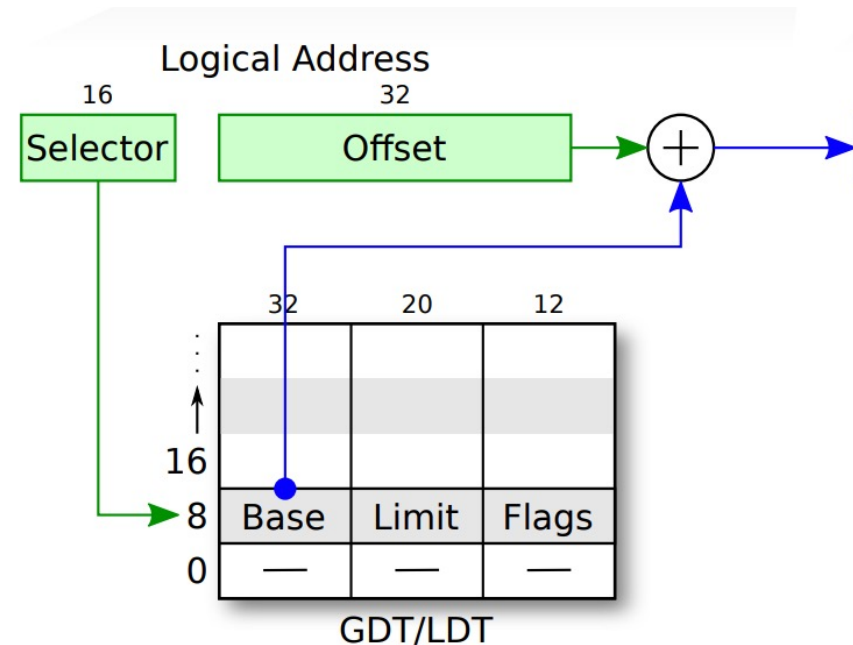
0x08:0 ~ 0x08:0x200000 are valid addresses for Program B
0x40000000 ~ 0x40200000

Protected Mode | Examples

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

    - **0x8080 < 0x8000000**

    - Address: **0x40008080**



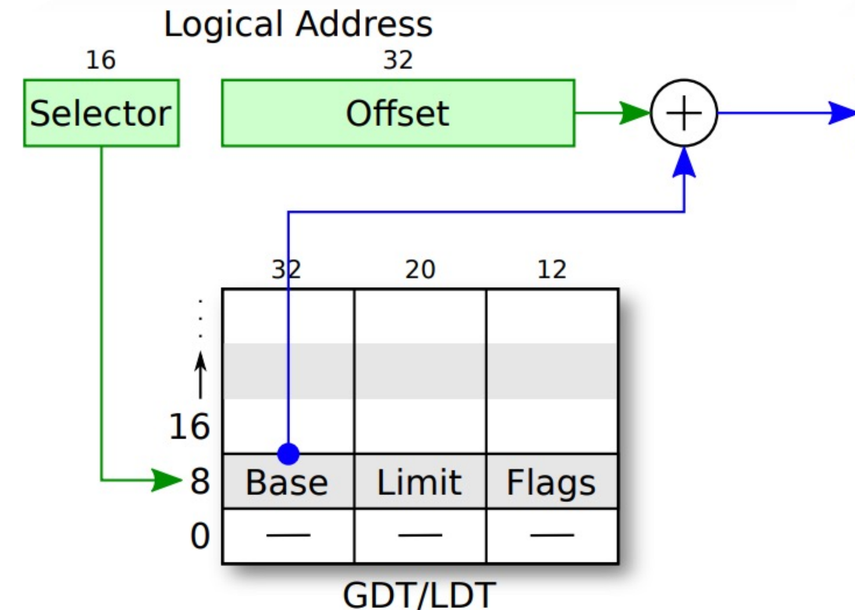
GDT index	32-bit Base	20-bit Limit	12-bit Flags
16	0x31310000	0x1000	G=0
8	0x40000000	0x8000	G=1
0	0x0	0x0	G=0

Protected Mode - Examples

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

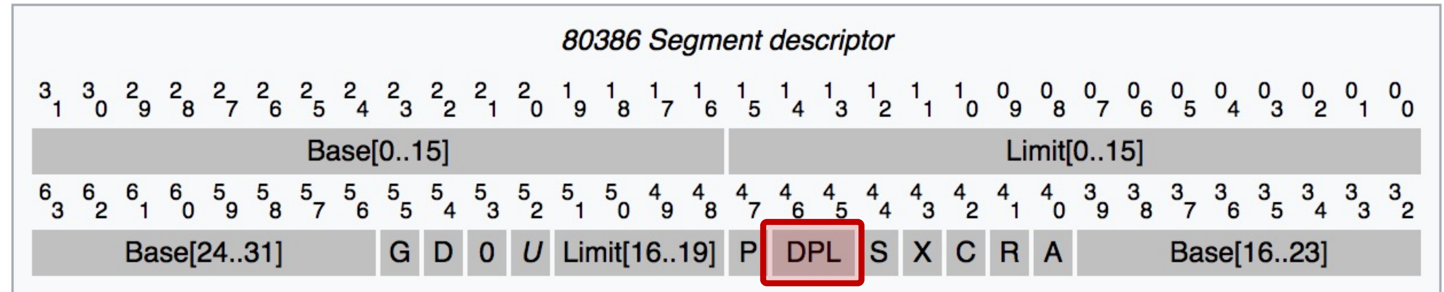
  - Address: **`0x31310333`**

- `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

Protected Mode | Memory Privilege

- **No memory privilege in real mode**

- Protected mode – four levels
  - **0 – highest, OS kernel**
  - 1 – OS kernel

  - 2 – highest user-level privilege
  - **3 – user-level privilege**

only these are used

Typically, 0 is for kernel, 3 is for user

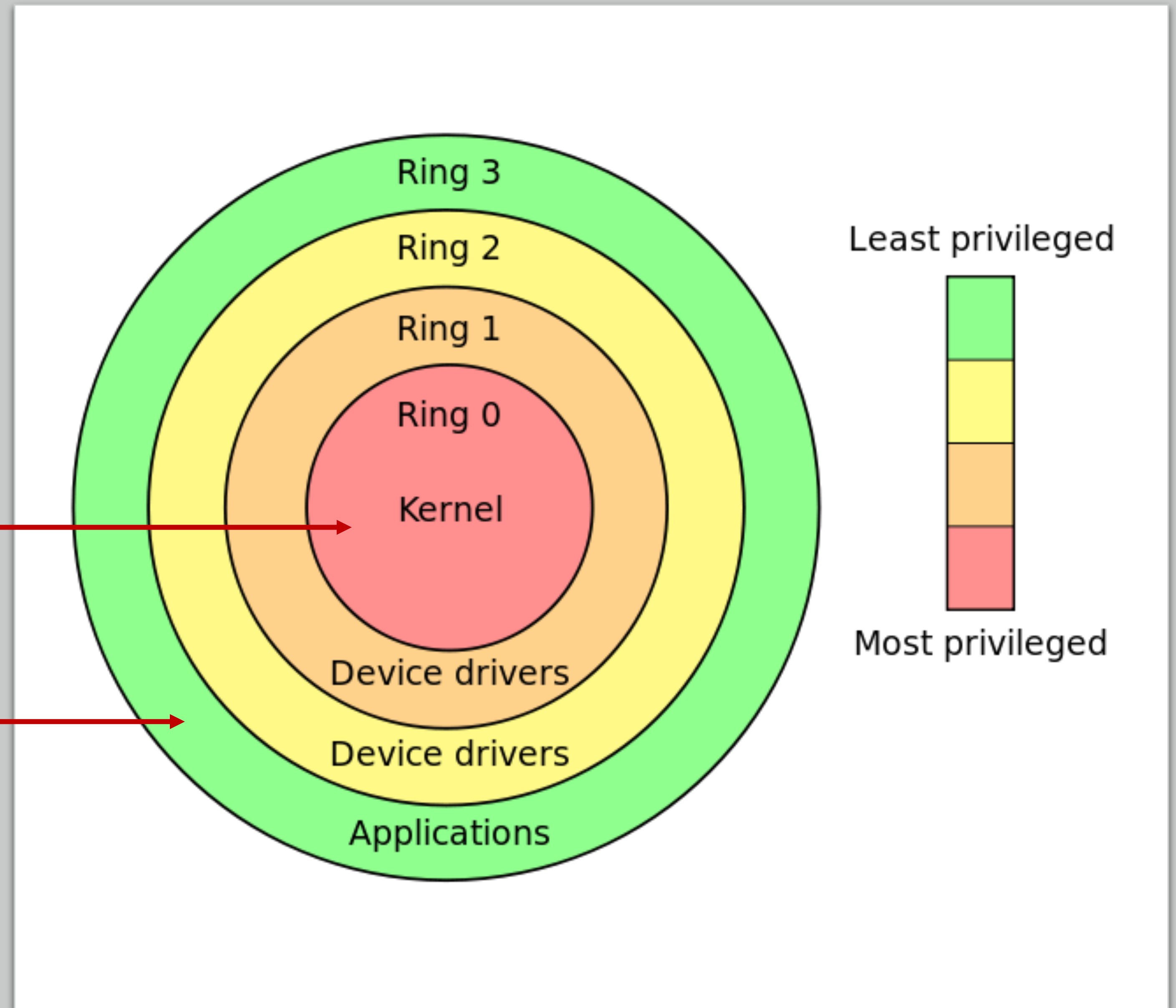


Image from: https://en.wikipedia.org/wiki/File:Priv_rings.svg

DPL→ Defines Ring Level

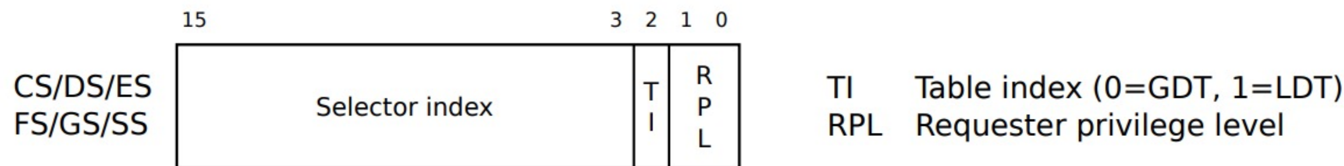
- **CPL** [**Current** Privilege Level]
  - Defined in the **last 2 bits** of the `%cs` register
  - Can change `%cs` only via: `lcall/ljmp/trap/int`

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**

DPL→ Defines Ring Level

- **CPL** [**Current** Privilege Level]
  - Defined in the **last 2 bits** of the **`%cs`** register
  - 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**

DPL→ Defines Ring Level

- **CPL** [**Current** Privilege Level]
  - Defined in the **last 2 bits** of the **`%cs`** register
  - Can change **`%cs`** only via: `lcall/ljmp/trap/int`
  - `mov %ax, %cs` ← **impossible!**

- **Can only move downwards in privilege levels!**
  - CPL==0, then `ljmp 0x3:0x1234` → **OK to execute**
  - CPL==3, then `ljmp 0x0:0x1234` → **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**

Kernel (Ring 0) can execute code in (Ring 3) via **`ljmp 0x3:0x1234`**

How can we go **back** to the kernel?

- can switch from ring 0 to ring 3 via **ljmp**
  - `ljmp 0x3:0x1234`
- **cannot** switch from ring 3 to ring 0 via ljmp
  - `ljmp 0x0:0x1234` ← **illegal instruction**
- use **`iret / sysexit / sysret`**
  - to switch from ring 3 to ring 0
  - will learn this in later lectures

Enabling Protected Mode (part 1): Create Global Descriptor Table (GDT)

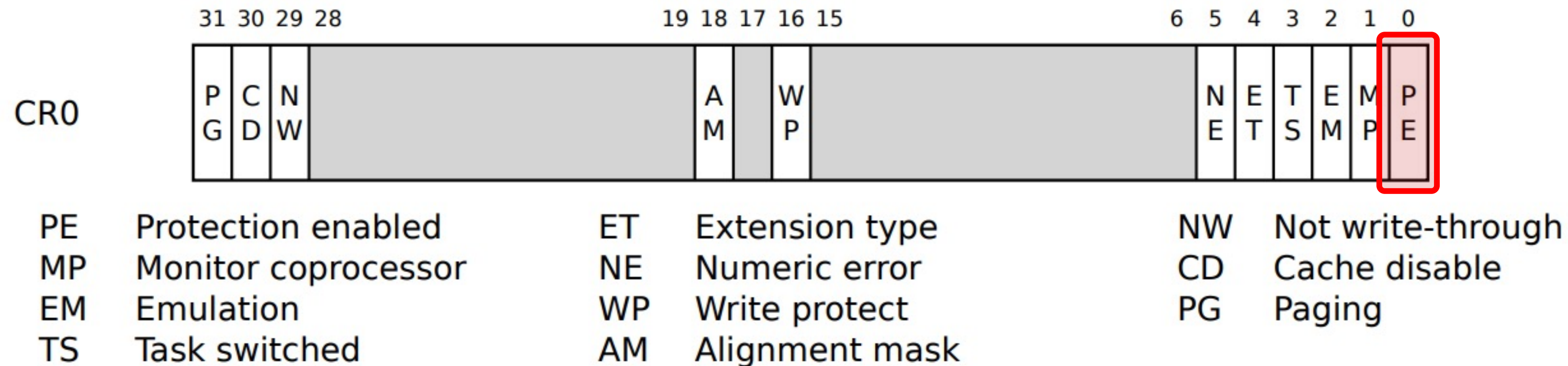
- In **boot/boot.S**
  - `%cs` to point to **0** ~ **0xffffffff** in DPL 0
  - `%ds` to point to **0** ~ **0xffffffff** in DPL 0
- **Only kernel can access these two segments**

```
# 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
```

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

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

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



Set **PE** [Protected Enabled] to **1** → 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
```

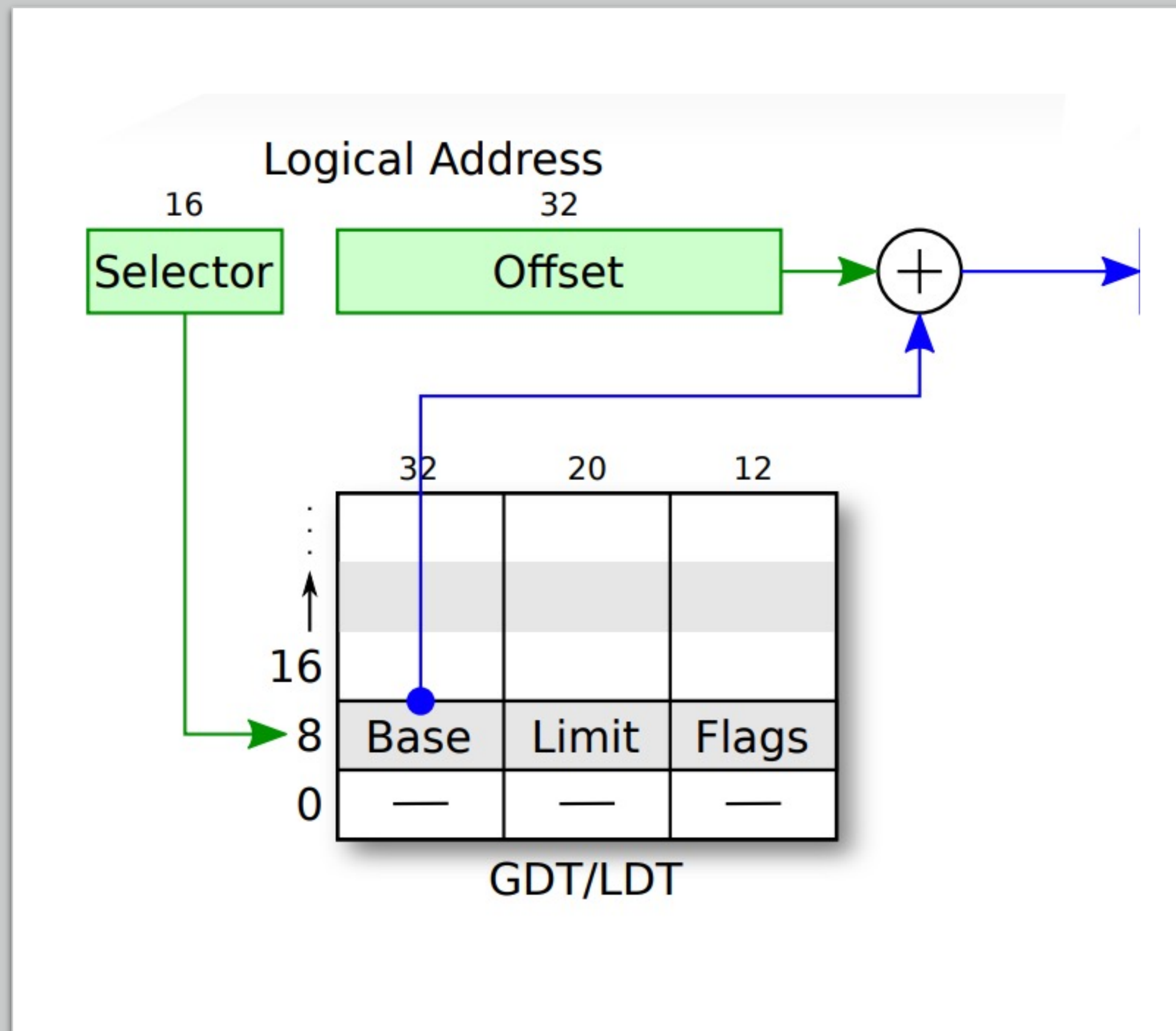
0x8 == 1000, Last 2 bits are zero

```
.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 applications
- 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



Additional Reading

- ELF Format
  https://en.wikipedia.org/wiki/Executable_and_Linkable_Format