EE488: Unix Kernel Design
Fall 2022

Booting

Youjip Won

KAIST EE

Address Mode

Address mode: **real mode** vs. protected mode

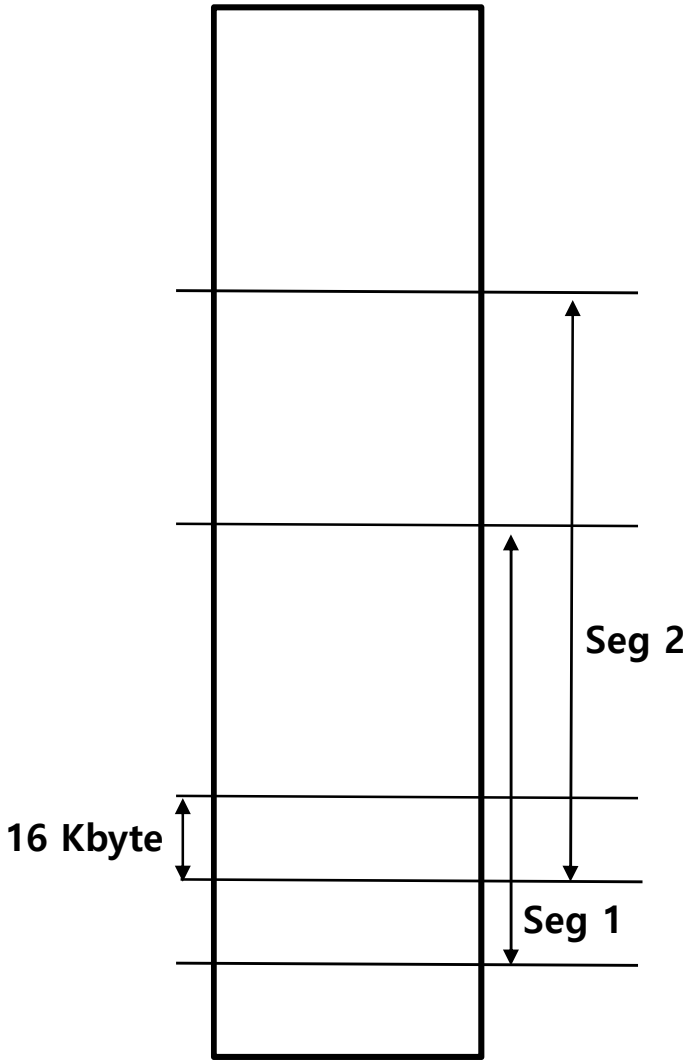
- real mode
  - Software generates the real (physical) address.
  - 8086 CPU
  - 16 bit registers
  - 20 bit address bus (maximum physical memory size: 1 Mbyte)

Address mode: **real mode** vs. protected mode

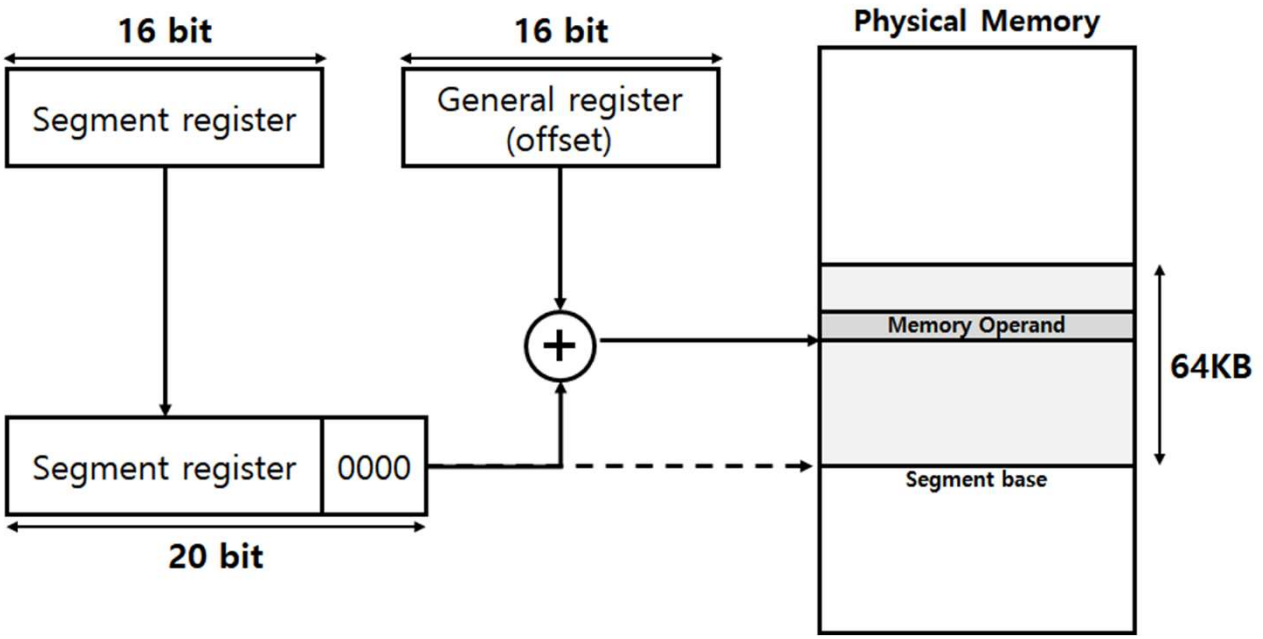
- real mode
  - 20 bit address space with 16 bit register
  - A physical address: [start: limit].
  - Address: `start<<4 + limit`
  - Four segments registers
    - CS: code segment
    - DS: data segment
    - SS: stack segment
    - ES: extra segment
  - If `CS = 0x2000, IP:0x1100`, then

    `CS:IP` refers to physical address

    `0x2000*16 + 0x1100 = 0x21100`
  - Segments can overlap.
  - No memory protection.



Address mode: **real mode** vs. protected mode



An address can access the other segment's address space.

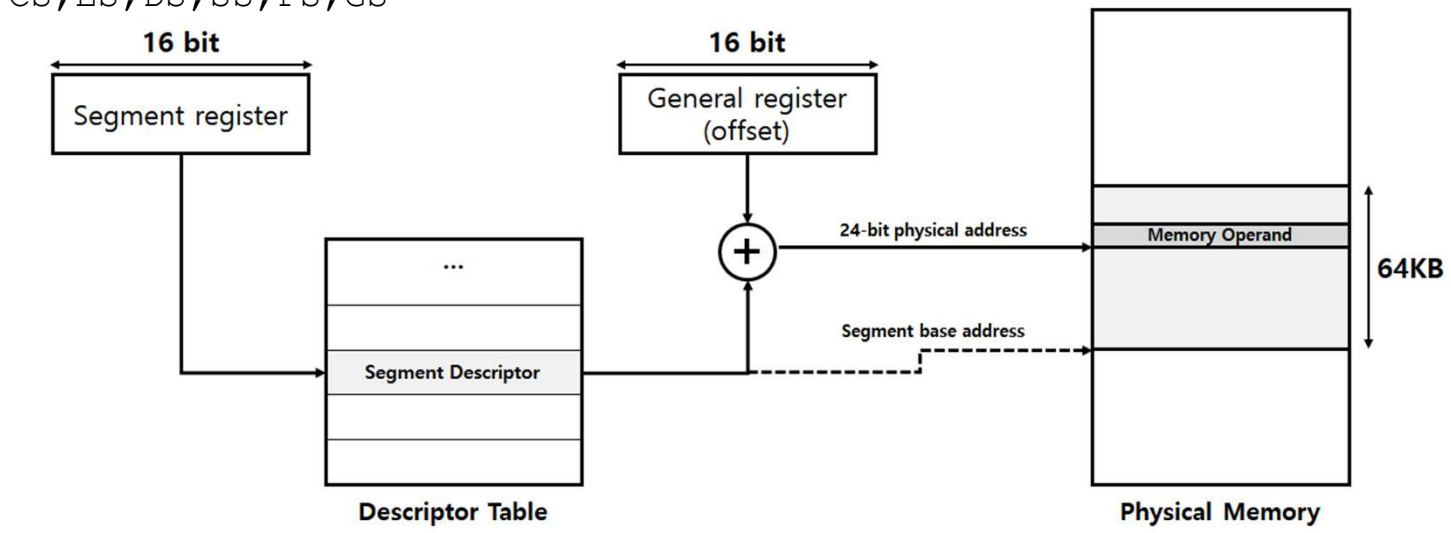
Address mode: real mode vs. **protected mode**

- Protected mode (16 bit)
  - Software generates the logical address.
  - 80286 (since 1982), 16bit CPU
  - 24 bit address bus (maximum physical memory size: 16 Mbyte)
  - Segment registers (`CS,ES,DS,SS, FS,GS`) holds the index at the segment descriptor table .

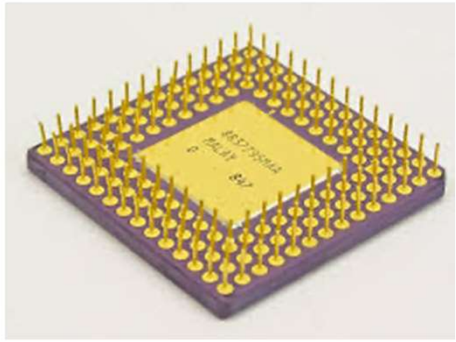
Address mode: real mode vs. **protected mode**

- Protected mode (16 bit)
  - Segment descriptor: <`base address`, `limit`> in the physical address space.
  - Segment register (segment selector register) is an index in the segment descriptor table.
  - If `%ip > limit`, abort!

`CS,ES,DS,SS,FS,GS`



Address mode: real mode vs. **protected mode**



- Protected mode (32 bit)
  - 80386, 32 bit CPU
  - Software generates the virtual address.
  - 32 bit address bus (maximum memory size: 4 Gbyte)
  - Introduction of Paging!!!

Address mode: real mode vs. **protected mode**

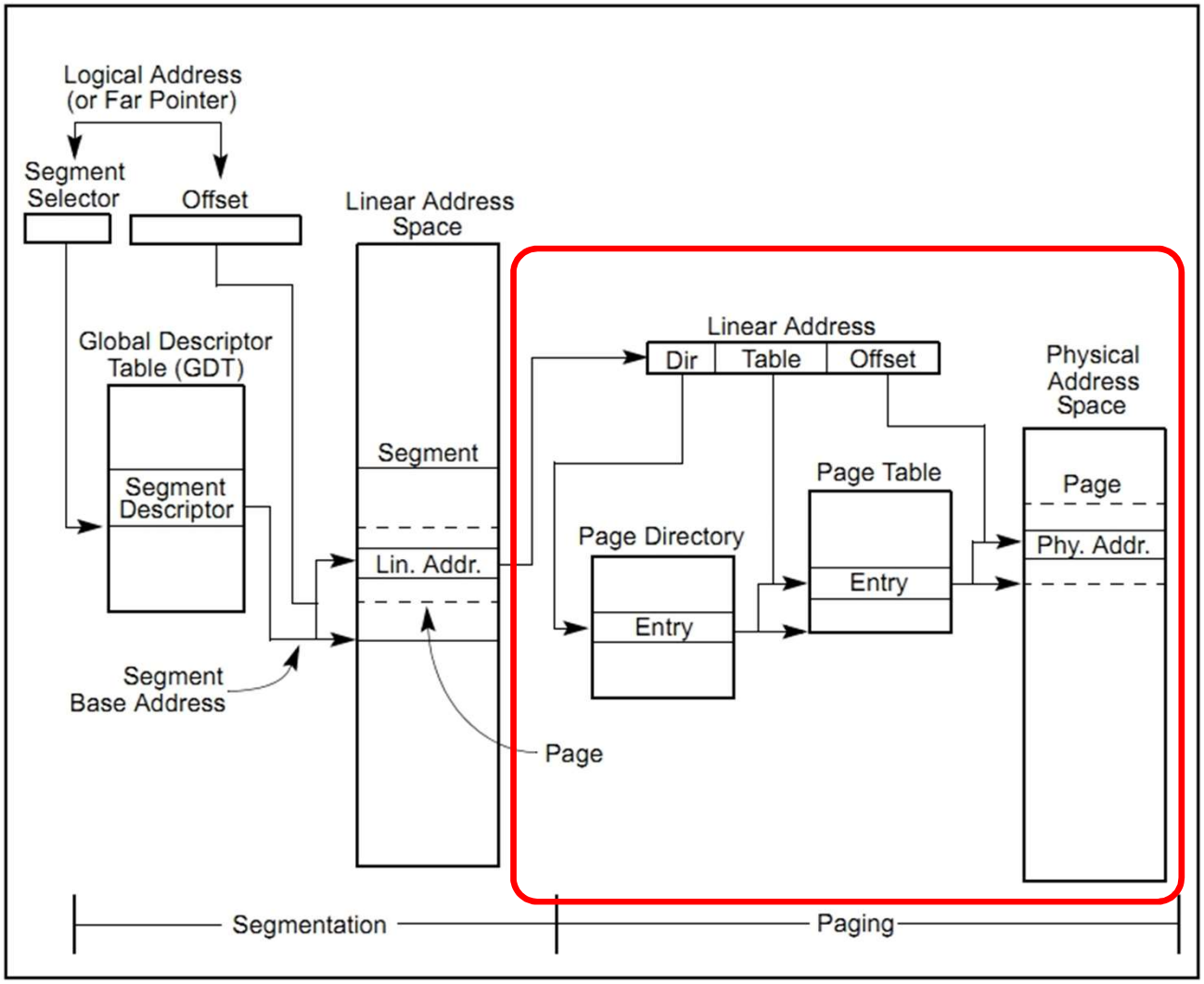
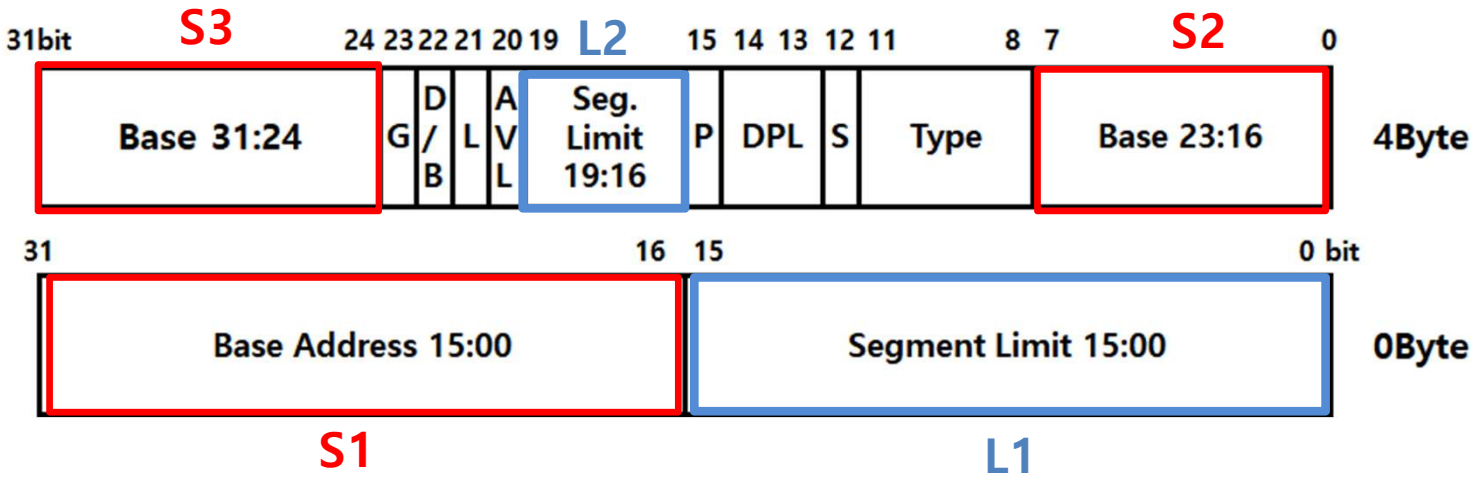


Figure 3-1. Segmentation and Paging

	16 bit real	16 bit protected	32 bit protected
cpu	8086	80286	80386
register	16 bit	16 bit	32 bit
Address bus	20 bit	24 bit	32 bit
Address mode	Real	Protected	protected
registers	`CS,SS,DS,ES`	`CS,SS,DS,ES, FS,GS`	
Addressing CS:IP	`CS<<4+IP`	`DT[CS]+IP`	`DT[CS]+IP`

Segment Descriptor: [base, limit, flags]

Holds location, size, permission, status of the segment.



Segment Base Address:

- 32 bit protected mode: 32 bit address bus → S3 + S2 + S1
- 16 bit protected mode: 24 bit address bus → S2 + S1

Segment Limit: 20 Bit L2 + L1

- if G == 0 → address unit = 1 Byte, Max Segment size = 1 Mbyte
- If G == 1 → address unit = 4 Kbyte, Max segment size = 4 Gbyte

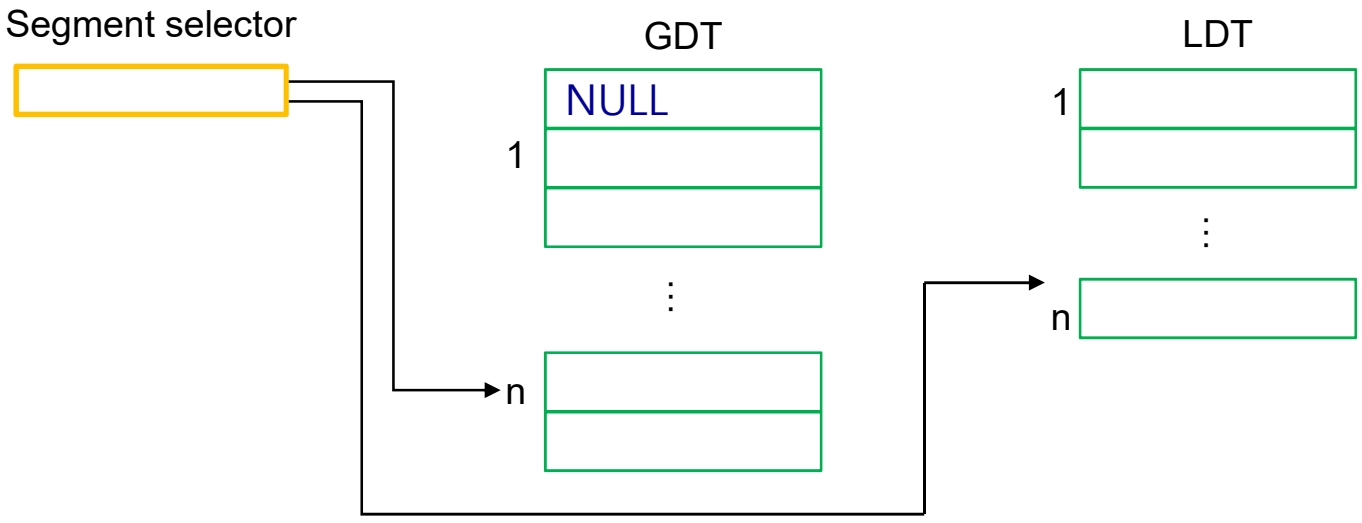
Segment Selector

- Segment Selector Register


```
Segment selector
TI: table indicator, 0 = GDT, 1 = LDT
RPL: Request Privilege Level
```

Segment descriptor table

- Kernel maintains the array of segment descriptors.
  - Per CPU: Global Descriptor Table
  - Per Process: Local Descriptor Table



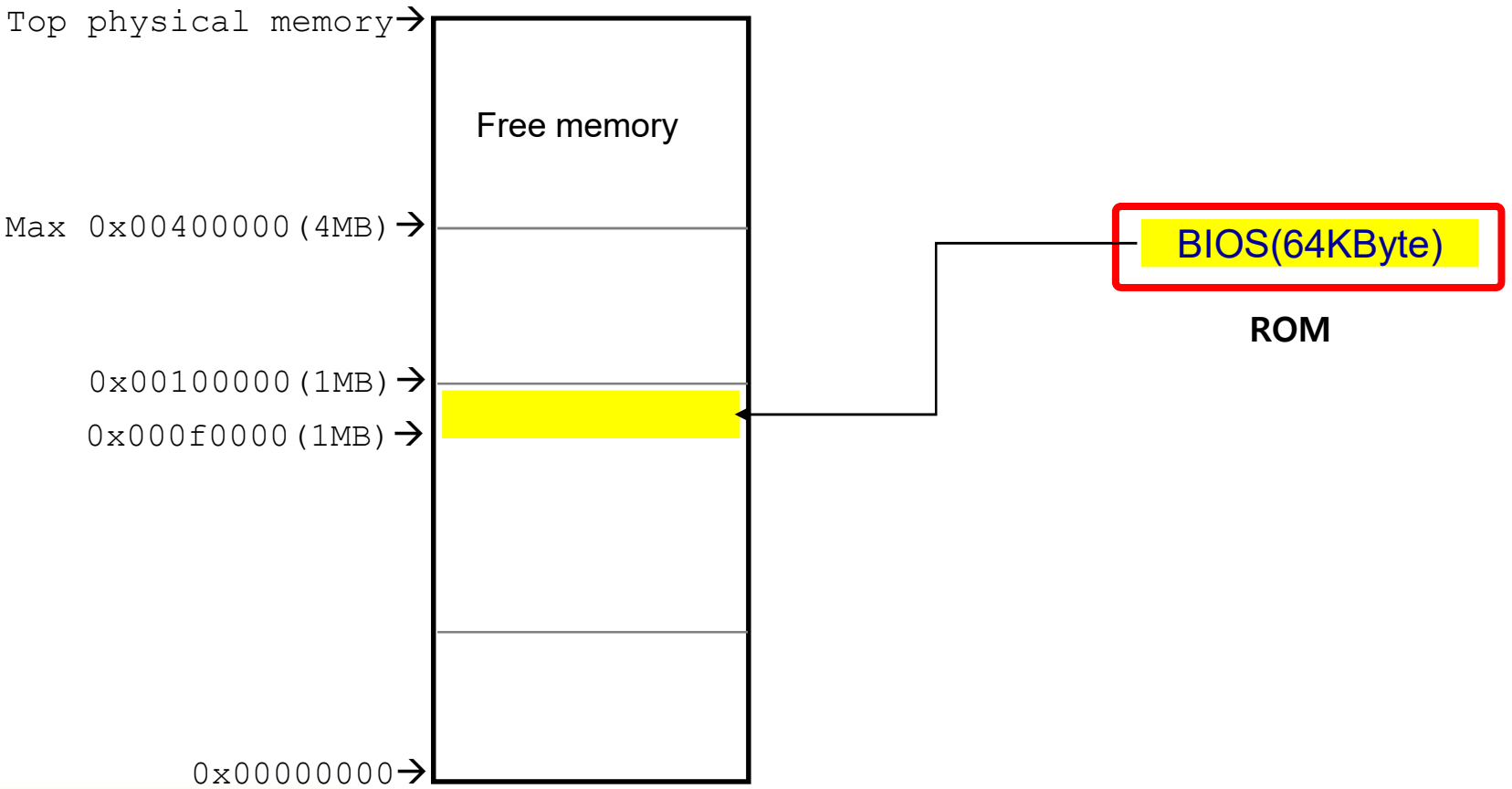
Booting-BIOS

Booting

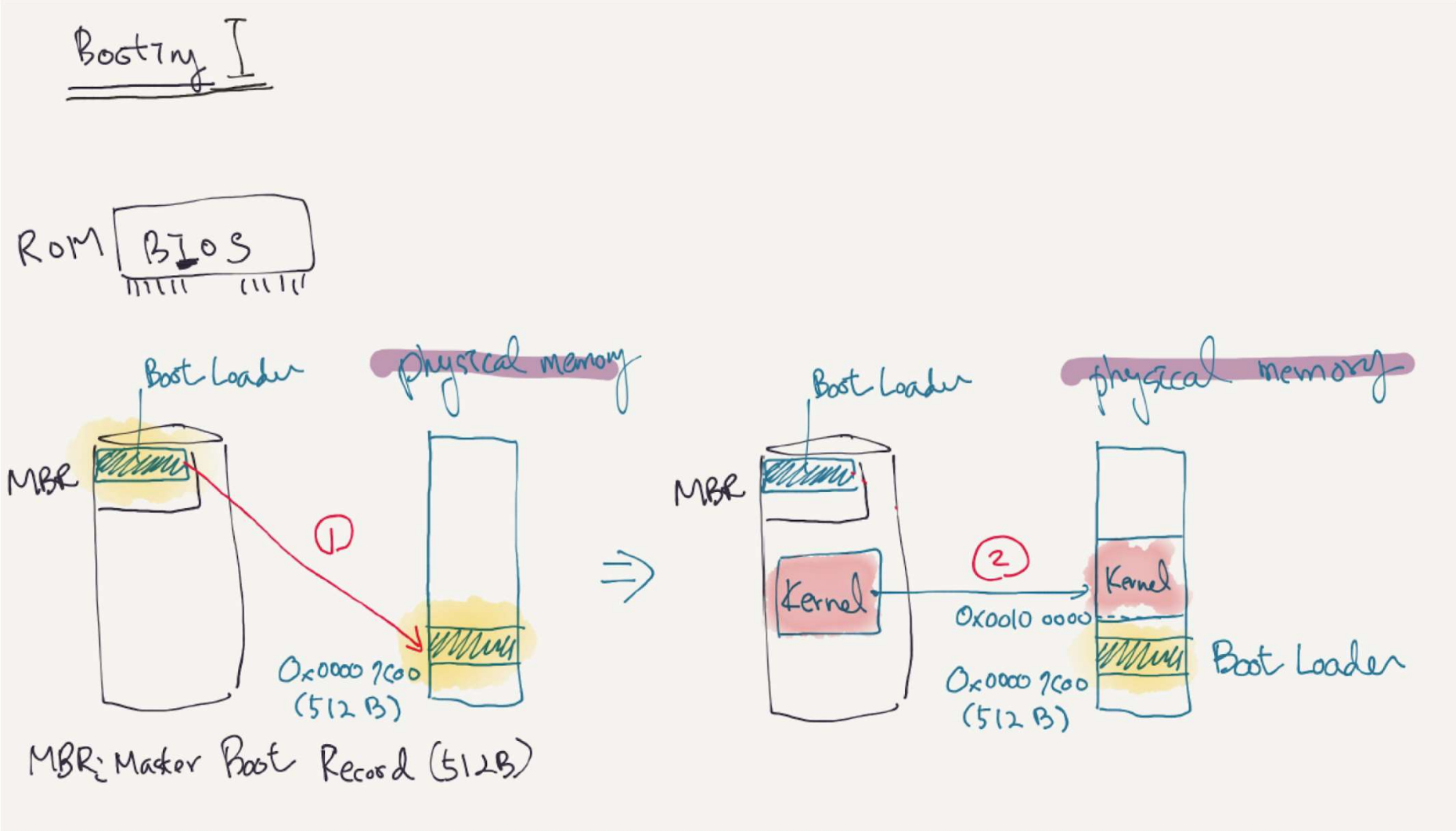
1. Load BIOS and run the BIOS.
2. Load the boot loader and run it.
3. Load the kernel and prepare the virtual memory for kernel.
4. Run the kernel.
    i. Create the first address space.
    ii. Create the first process.
    iii. Run the first process.
    iv. Run the shell.

Load BIOS

- **Power-On**
- The mainboard reads ROM and load BIOS to memory. You cannot execute program on ROM.
  - Load BIOS to `0x000f0000 - 0x000fffff(960KB-1MB) (x86 CPU).`



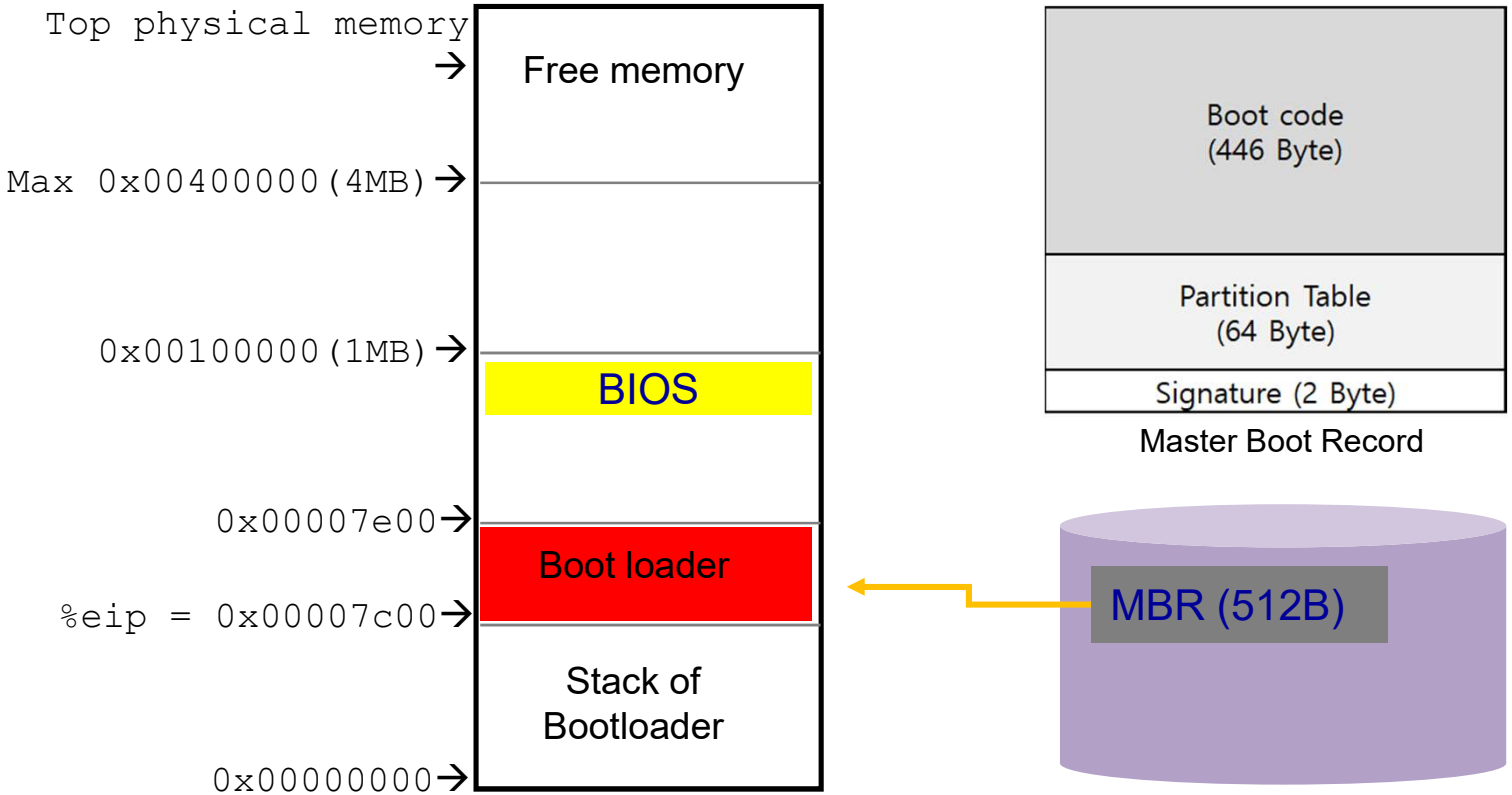
Booting 1


Booting I
ROM
BIOS
Boot Loader
physical memory
MBR
1
0x0000 7c00
(512 B)
MBR: Master Boot Record (512B)
Boot Loader
physical memory
MBR
Kernel
2
Kernel
0x0010 0000
0x0000 7c00
(512 B)
Boot Loader

Booting-Bootloader

Load Boot Loader.

- Run BIOS
  - Perform POST (Power-On-Self-Test)
  - Load the bootloader from the disk onto `0x00007c00 (28 Kbyte)- 0x7e00.`
    - Bootloader resides at the first sector of the disk (Master Boot Record).



Bootloader

- Bootloader of XV6: `bootasm.S` and `bootmain.c` .
  - `bootasm.S`: switch the mode from real to protected mode.
  - `bootmain.c`: load the kernel and initialize the virtual address space for the kernel
- Load the bootloader at `0x7c00`.

```
#Makefile
103 bootblock: bootasm.S bootmain.c
104         $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
105         $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
106         $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
107         $(OBJDUMP) -S bootblock.o > bootblock.asm
108         $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
```

Disable Interrupt and Register Reset

- Disable interrupt. (Because Interrupt handlers are not loaded yet.)
- Zeroing the Registers.

```
start:
  cli                    # BIOS enabled interrupts; disable


  # Zero-fill data segment registers DS, ES, and SS.
  xorw    %ax,%ax              # Set %ax to zero
  movw    %ax,%ds              # -> Data Segment
  movw    %ax,%es              # -> Extra Segment
  movw    %ax,%ss              # -> Stack Segment
```

Enable 21st address line.

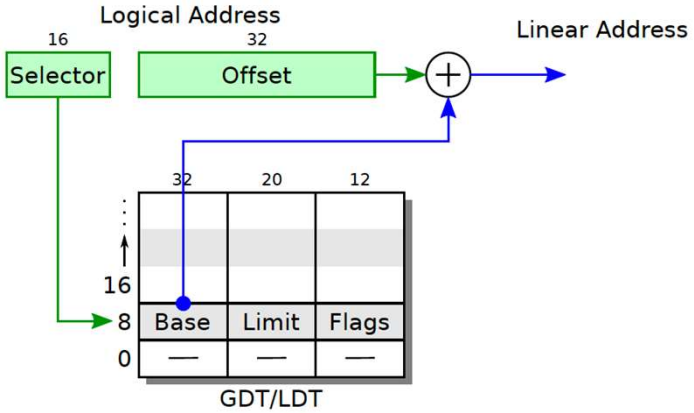
- Enable 21^st line of the address line. Initially, this line is disabled.
- Not so important. If you are interested further, please read Appendix B, the boot loader.

```
23 seta20.1:
24   inb $0x64,%al              # Wait for not busy
25   testb  $0x2,%al
26   jnz seta20.1
27
28   movb $0xd1,%al              # 0xd1 -> port 0x64
29   outb %al,$0x64
30
31 seta20.2:
32   inb $0x64,%al              # Wait for not busy
33   testb  $0x2,%al
34   jnz seta20.2
35
36   movb $0xdf,%al              # 0xdf -> port 0x60
37   outb %al,$0x60
```

Mode Switch

- Load the address of the global descriptor table to GDTR.
- Set `cr0` register to switch from real mode to protected mode.

```
lgdt    gdtdesc                 # load the address of global
                                # descriptor table to GDTR.
movl    %cr0, %eax
orl     $CR0_PE, %eax           # CR0_PE=0x00000001
movl    %eax, %cr0              # Set up control register for
                                # protected mode
```



Setup Global Descriptor Table

```
# Bootstrap GDT
.p2align 2        # power of 2 byte alignment (4 byte)
gdt:
  SEG_NULLASM                                  # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)    # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)          # data seg
```

```
#define STA_X   0x8 // Executable segment
#define STA_W   0x2 // Writeable (non-executable segments)
#define STA_R   0x2 // Readable (executable segments)
```

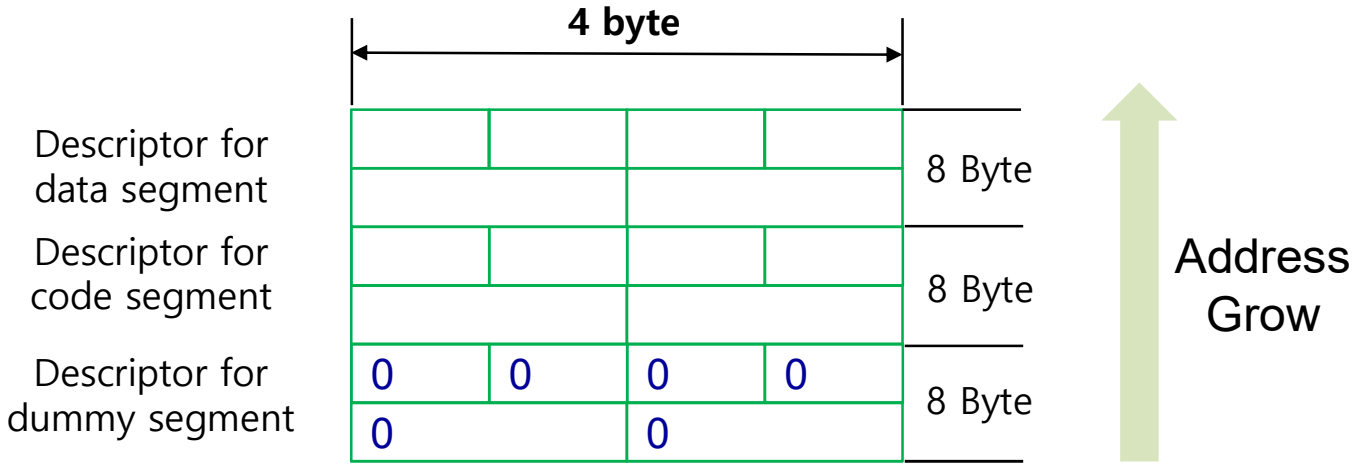
- No translation and G =1
  - code segment: `base address: 0, limit: 4 Gbyte`
  - Data segment: `base address: 0, limit: 4 GByte`

create x86 segments(asm.h)

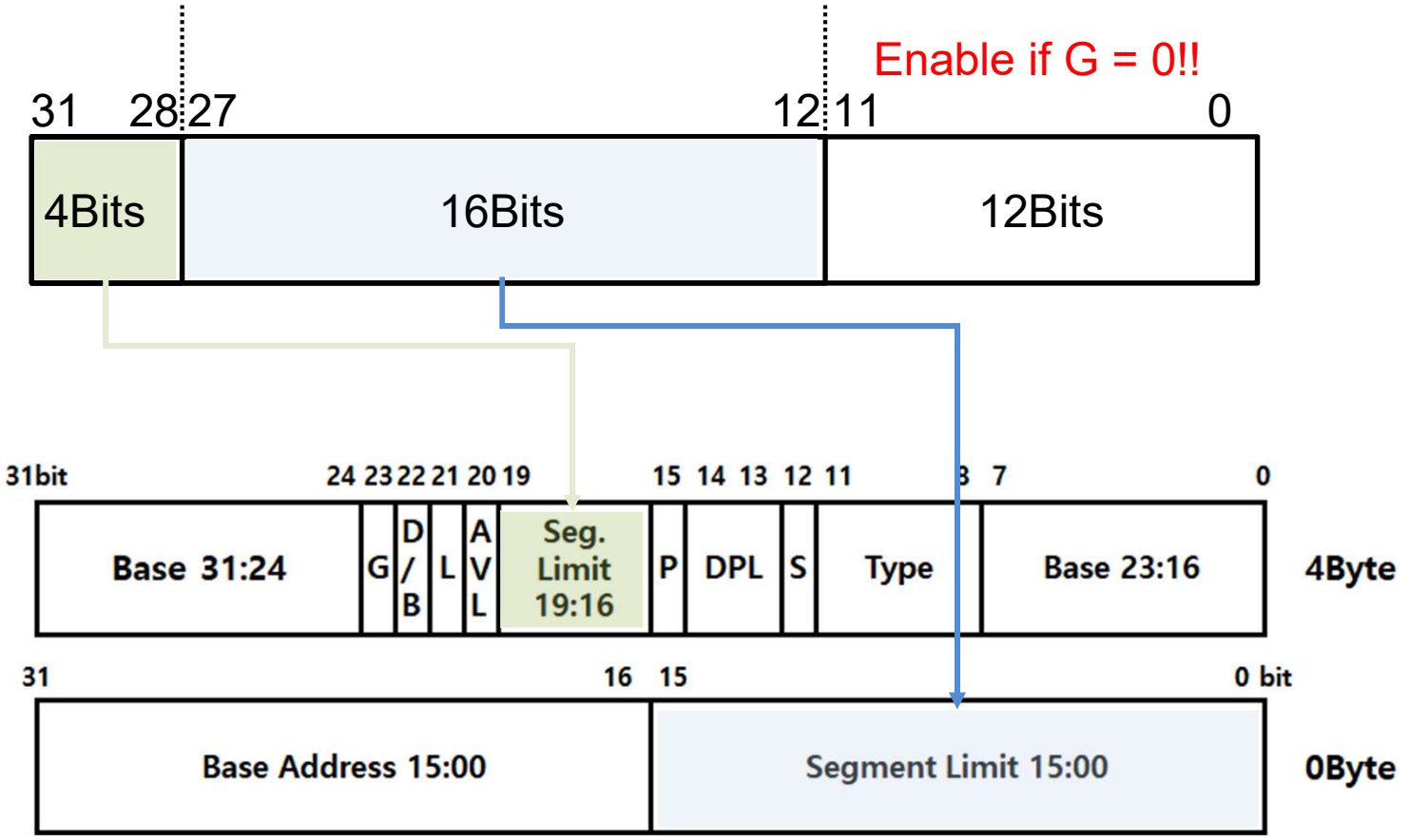
- `.word` : truncate the values of comma separated expressions to word.
- `.byte` : generate the initialized bytes of the command separated expressions.

```
#define SEG_NULLASM     \
        .word 0, 0;     \
        .byte 0, 0, 0, 0
```

Why is the word 16 bit?

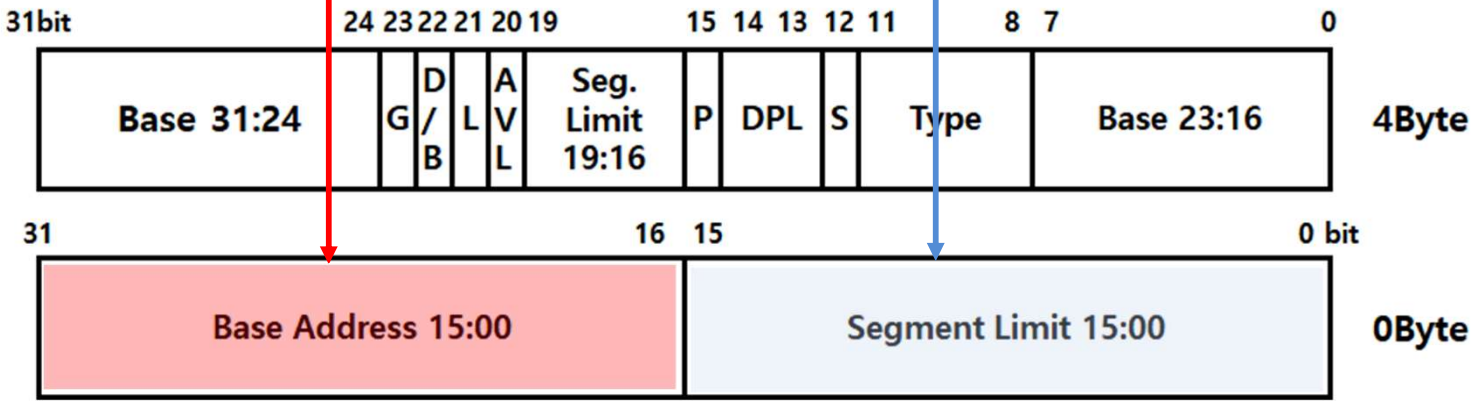


Segment Limit



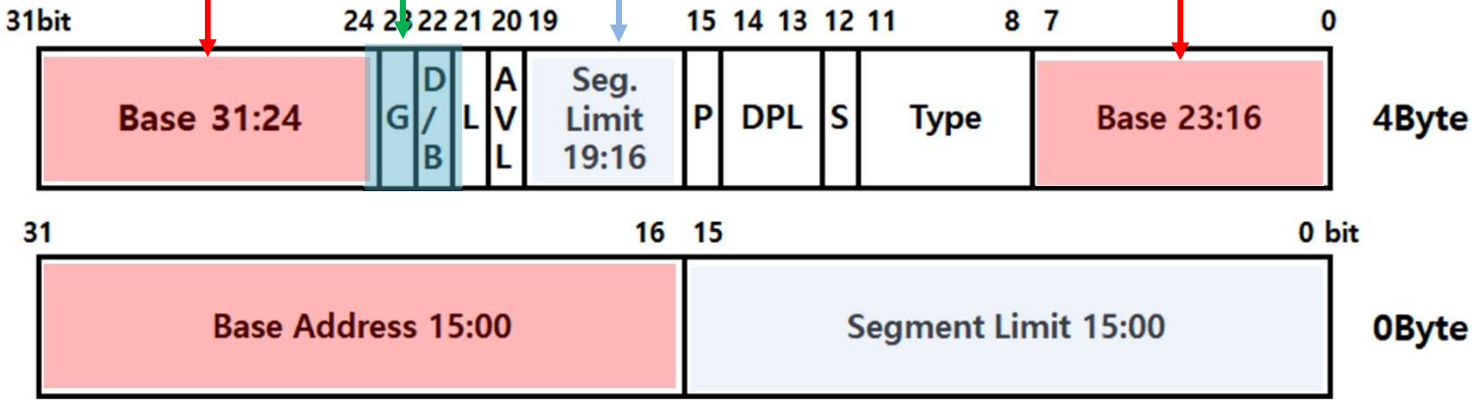
SEG_ASM(type,base,lim)

```
// The 0xC0 means the limit is in 4096-byte units
// and (for executable segments) 32-bit mode.
#define SEG_ASM(type,base,lim)                                  \
.word (((lim) >> 12) & 0xffff), ((base) & 0xffff))              \
.byte (((base) >> 16) & 0xff), (0x90 | (type)),                 \
        (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```



SEG_ASM(type,base,lim)

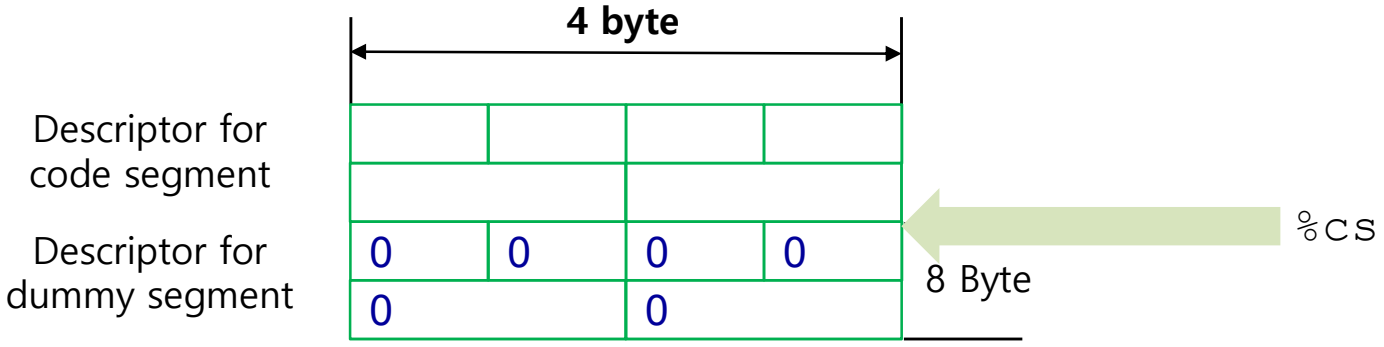
```
// The 0xC0 means the limit is in 4096-byte units
// and (for executable segments) 32-bit mode.
#define SEG_ASM(type,base,lim)                                  \
.word (((lim) >> 12) & 0xffff), ((base) & 0xffff))  \
.byte (((base) >> 16) & 0xff), (0x90 | (type)),     \
        (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) &
0xff)
```



Mode switch

- In real mode, CS register contains physical address.
- In protected mode, CS register should contain the index to the descriptor table. → To change to protected mode, reload `%cs` and `%eip`.

```
ljmp    $(SEG_KCODE<<3), $start32
        # %CS is set to 0x0008 (second entry in GDT).
        # The second descriptor in GDT is for KERNEL CODE.
.code32 # now the mode change completes!!!
#define SEG_KCODE 1
#define SEG_KDATA 2
```



Initialize segment registers after mode Switch.

- Initialize the segment registers.

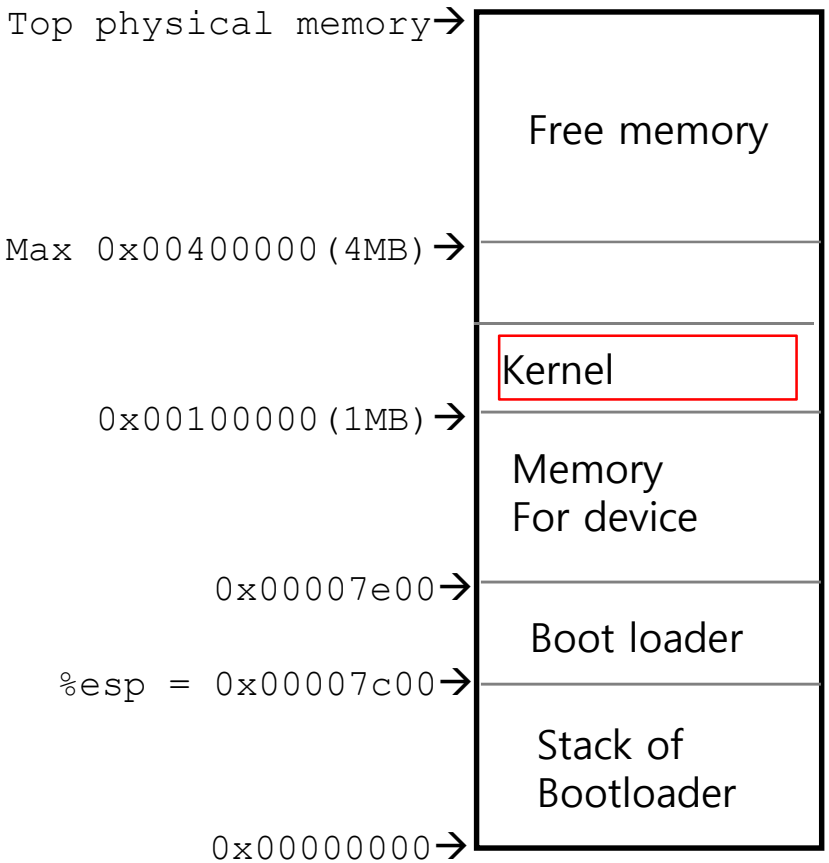
```
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds        # -> DS: Data Segment
  movw    %ax, %es        # -> ES: Extra Segment
  movw    %ax, %ss        # -> SS: Stack Segment
  movw    $0,  %ax        # Zero segments not ready for use
  movw    %ax, %fs        # -> FS
  movw    %ax, %gs        # -> GS
```

- To use C function, we need a stack.
- Set up the stack pointer and call into C (`bootmain.C`).

```
  movl    $start, %esp
  call    bootmain
```

Load the kernel.

- The boot loader loads the kernel into physical address `0x00100000` (1MByte).
- Jump to `entry();`



bootmain.c

```
#include "elf.h"
#include "x86.h"
#include "memlayout.h"

#define SECTSIZE  512

void readseg(uchar*, uint, uint);

void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
```

< Read kernel image from disk to memory. >

```
  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

Load the kernel.

- The boot loader loads the kernel into physical address `0x00100000` (1MByte).
- Jump to `entry();`

```
28   readseg((uchar*)elf, 4096, 0);
35   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
36   eph = ph + elf->phnum;
37   for(; ph < eph; ph++){
38     pa = (uchar*)ph->paddr;
39     readseg(pa, ph->filesz, ph->off);
40     if(ph->memsz > ph->filesz)
41       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
42   }
46   entry = (void(*)(void))(elf->entry);
47   entry();
```

entry()

```
entry:
1045 # Turn on page size extension
     # for 4Mbyte pages.
1046 movl %cr4, %eax
1047 orl $(CR4_PSE), %eax
1048 movl %eax, %cr4

1049 # Set page directory
1050 movl $(V2P_WO(entrypgdir)), %eax
1051 movl %eax, %cr3

1052 # Turn on paging.
1053 movl %cr0, %eax
1054 orl $(CR0_PG|CR0_WP), %eax
1055 movl %eax, %cr0

1057 # Set up the stack pointer to
       point high address.
1058 movl $(stack + KSTACKSIZE), %esp

1063 # jump to main.
1064 mov $main, %eax
1065 jmp *%eax
```

Set page size to 4 Mbyte.

entry()

```
entry:
1045 # Turn on page size extension
     # for 4Mbyte pages.
1046 movl %cr4, %eax
1047 orl $(CR4_PSE), %eax
1048 movl %eax, %cr4

1049 # Set page directory
1050 movl $(V2P_WO(entrypgdir)), %eax
1051 movl %eax, %cr3

1052 # Turn on paging.
1053 movl %cr0, %eax
1054 orl $(CR0_PG|CR0_WP), %eax
1055 movl %eax, %cr0

1057 # Set up the stack pointer to
       point high address.
1058 movl $(stack + KSTACKSIZE), %esp

1063 # jump to main.
1064 mov $main, %eax
1065 jmp *%eax
```

1. Get the virtual address of page directory.
2. Subtract `KERNBASE`, start of kernel address.
3. Set it to the address of page directory.

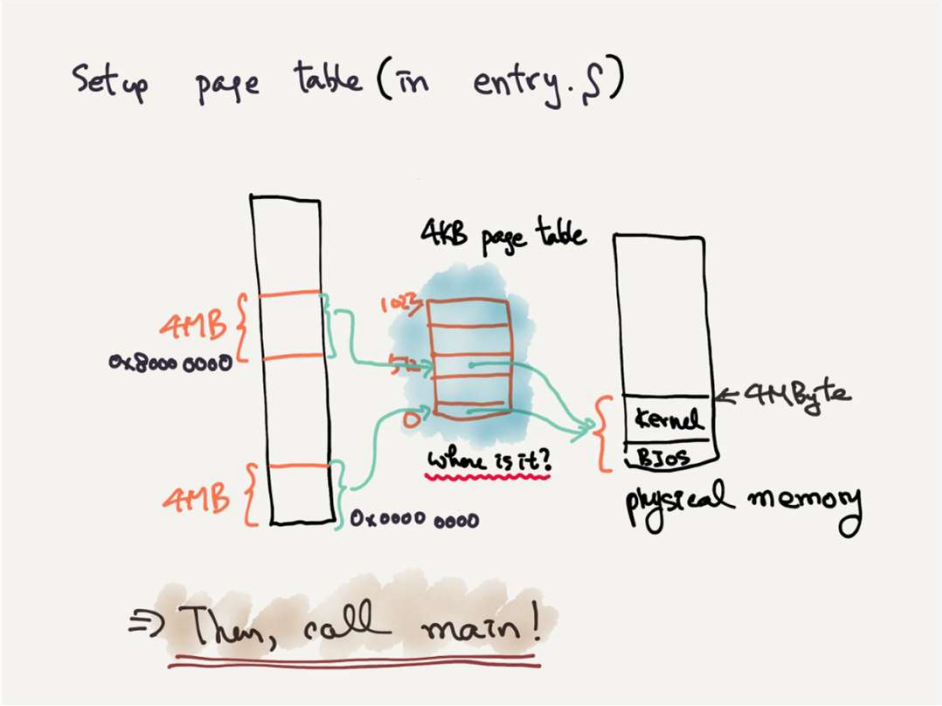
Page directory set up in `main.c()`

```
// The boot page table used in entry.S and entryother.S.
// PTE_PS in a page directory entry enables 4Mbyte pages.

pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS, //512th entry
};
```

22

```
KERNBASE = 0x80000000
1000 0000 0000 0000 0000 0000 0000 0000
```



entry()

```
entry:
1045 # Turn on page size extension
     # for 4Mbyte pages.
1046 movl %cr4, %eax
1047 orl $(CR4_PSE), %eax
1048 movl %eax, %cr4

1049 # Set page directory
1050 movl $(V2P_WO(entrypgdir)), %eax
1051 movl %eax, %cr3

1052 # Turn on paging.
1053 movl %cr0, %eax
1054 orl $(CR0_PG|CR0_WP), %eax
1055 movl %eax, %cr0

1057 # Set up the stack pointer to
       point high address.
1058 movl $(stack + KSTACKSIZE), %esp

1063 # jump to main.
1064 mov $main, %eax
1065 jmp *%eax
```

1. Turn on paging.

Booting

- Bootloader starts to execute in real mode.
- Change the address mode.
  1. Enable the address line 21.
  2. Initialize the global descriptor table.
  3. Set the bit 0 of the control register (CR0).
  4. Set the segment selector (cs) register and instruction pointer (eip).
- Initialize the segment selector registers.
- Loads the kernel.
- Prepare the initial page table for the kernel to run.
- Jump to kernel.

entry()

```
entry:
1045 # Turn on page size extension
     # for 4Mbyte pages.
1046 movl %cr4, %eax
1047 orl $(CR4_PSE), %eax
1048 movl %eax, %cr4

1049 # Set page directory
1050 movl $(V2P_WO(entrypgdir)), %eax
1051 movl %eax, %cr3

1052 # Turn on paging.
1053 movl %cr0, %eax
1054 orl $(CR0_PG|CR0_WP), %eax
1055 movl %eax, %cr0

1057 # Set up the stack pointer to
       point high address.
1058 movl $(stack + KSTACKSIZE), %esp

1063 # jump to main.
1064 mov $main, %eax
1065 jmp *%eax

.comm stack, KSTACKSIZE
```

1. Setup stack pointer.

   (KSTACKSIZE=4096)

summary

- Booting
  - Turn on the computer
  - Run BIOS.
  - Run Bootloader.
    - Change the address mode from real mode to protected mode.
    - Load kernel.
    - Setup page table for the kernel.
    - Jump to `main()`.