Bootloader starts

bootbock
512B

0x7c00

Real Mode

CS : 0x0
SS : 0x0
GDT: 0x0
IDT: 0x0

EIP: 0x7c00
ESP: 0x0
TSS: 0x0

Physical

0 0x7c00 0x7d00

512MB
Bootloader starts

8411 start:

8412 cli # BIOS enabled interrupts; disable

8413

8414 # Zero data segment registers DS, ES, and SS.
8415 xorw %ax, %ax # Set %ax to zero
8416 movw %ax, %ds # -> Data Segment
8417 movw %ax, %es # -> Extra Segment
8418 movw %ax, %ss # -> Stack Segment
Switch to protected mode

- Switch from real to protected mode
  - Use a bootstrap GDT that makes virtual addresses map directly to physical addresses so that the effective memory map doesn’t change during the transition.

```
8441 lgdt gdtdesc
8442 movl %cr0, %eax
8443 orl $CR0_PE, %eax
8444 movl %eax, %cr0
```
Load GDT

bootbock
512B

Physical

0 0x7c00 0x7d00 512MB

Real Mode

CS : 0x0
SS : 0x0
GDT: 0x7c78
IDT: 0x0

FIP: 0x7c1d
ESP: 0x0
TSS: 0x0

GDT

NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
Actual switch

- Use long jump to change code segment

```
8453 1jmp $(SEG_KCODE<<3), $start32
```

- Explicitly specify code segment, and address

- Segment is 0b1000 (0x8)

- Also the segment has 32bit flag

- CPU will switch to 32 bit when segment is loaded, and PE flag is set in CR0
Long jump

bootbock
512B

Physical

0

0x7c00

0x7d00

512MB

GDT

NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB

Protected Mode

CS : 0x8
SS : 0x0
GDT: 0x7c78
IDT: 0x0

EIP: 0x7c1d
ESP: 0x0
TSS: 0x0
Why CS is 0x8, not 0x1?

- Segment selector:

  ![Segment Selector Diagram]

  - Table Indicator
    - 0 = GDT
    - 1 = LDT
    - Requested Privilege Level (RPL)
Segments

8456 start32:

8458 movw $(SEG_KDATA<<3), %ax # Data segment
8459 movw %ax, %ds # -> DS: Data Segment
8460 movw %ax, %es # -> ES: Extra Segment
8461 movw %ax, %ss # -> SS: Stack Segment
8462 movw $0, %ax # Zero segments not in use
8463 movw %ax, %fs # -> FS
8464 movw %ax, %gs # -> GS
Setup stack

- Need stack to use C
  - Function invocations
  - Note, there were no stack instructions before that

8467 movl $start, %esp
8468 call bootmain
First stack

Linear Stack

Code

Data

Physical Stack

0x7c00

0x7d00

GDT

NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB

Protected Mode

CS: 0x8
SS: 0x10
GDT: 0x7c78
IDT: 0x0

EIP: 0x7c1d
ESP: 0x7c00
TSS: 0x0
First page table

- Two 4MB entries (large pages)
- Entry #0
  - 0x0 – 4MB → 0x0:0x400000
- Entry #960
  - 0x0 – 4MB → 0x8000000:0x80400000
The diagram illustrates the process of converting a linear address to a physical address in a 4-MByte page. The linear address is divided into two parts: Directory and Offset.

- The Directory is used to select a page directory.
- The Offset is used to select a page within the page directory.

The page directory contains a PDE (Page Directory Entry) with PS=1, indicating a page table entry. The PDE is used to translate the page directory entry into a physical address.

The CR3 register is used to select the page directory. The 32 bits of CR3 are used to index into the page directory, and the 18 bits of the page directory entry are used to calculate the physical address.

The 22 bits of the Offset are used to select a 4-MByte page within the page directory.

The diagram shows a flow from the Linear Address to the Physical Address, passing through the Directory, Offset, Page Directory, PDE with PS=1, and CR3.
First page table

1310 __attribute__((aligned__(PGSIZE)))
1311 pde_t entrypgdir[NPDENTRIES] = {
1312 // Map VA’s [0, 4MB) to PA’s [0, 4MB)
1313 [0] = (0) | PTE_P | PTE_W | PTE_PS,
1314 // Map VA’s [KERNBASE, KERNBASE+4MB) to PA’s [0, 4MB)
1315 [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1316 };
entry:

# Turn on page size extension for 4Mbyte pages
movl %cr4, %eax
orl $(CR4_PSE), %eax
movl %eax, %cr4

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

# Turn on paging.
movl %cr0, %eax
orl $(CR0_PG|CR0_WP), %eax
movl %eax, %cr0
High address stack

Linear

Stack

Kernel

Code

Data

0

4MB

2GB

4GB

Virtual

Physical

Page table

CS: 0x8
SS: 0x10
GDT: 0x7c78
IDT: 0x0
EIP: 0x10001a
ESP: stack

0 - 4MB
0x0
...
2GB - 2GB + 4MB
...

Protected Mode

GDT

NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
1053 # Set up the stack pointer.
1054 movl $(stack + KSTACKSIZE), %esp
1055
1056 # Jump to main(), and switch to executing at
1057 # high addresses. The indirect call is needed because
1058 # the assembler produces a PC-relative instruction
1059 # for a direct jump.
1060 mov $main, %eax
1061 jmp *%eax
1062
1063 .comm stack, KSTACKSIZE
Thank you!