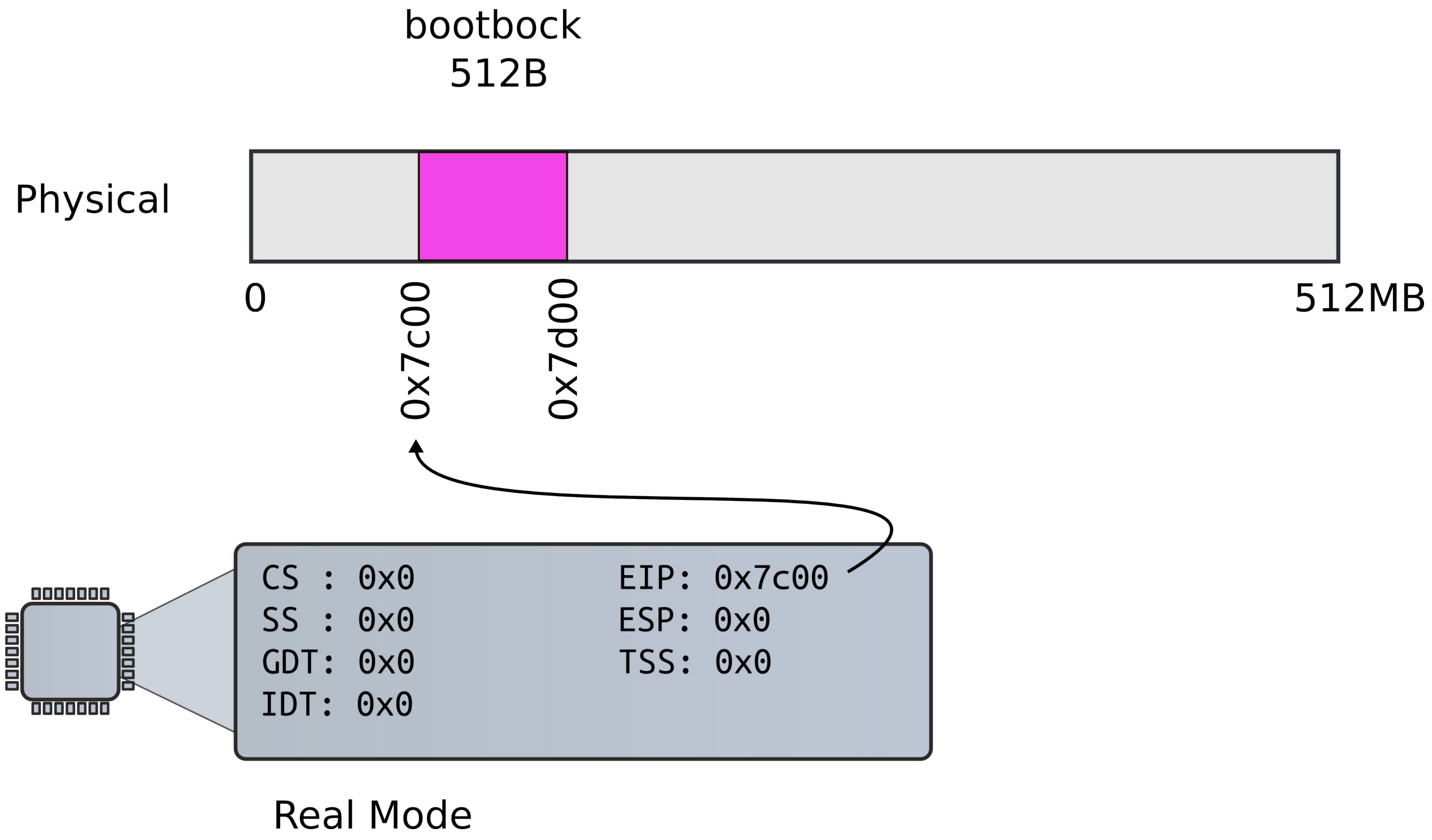
CS5460/6460: Operating Systems

Lecture 7: System boot

Anton Burtsev
January, 2014

Bootloader starts



Bootloader starts

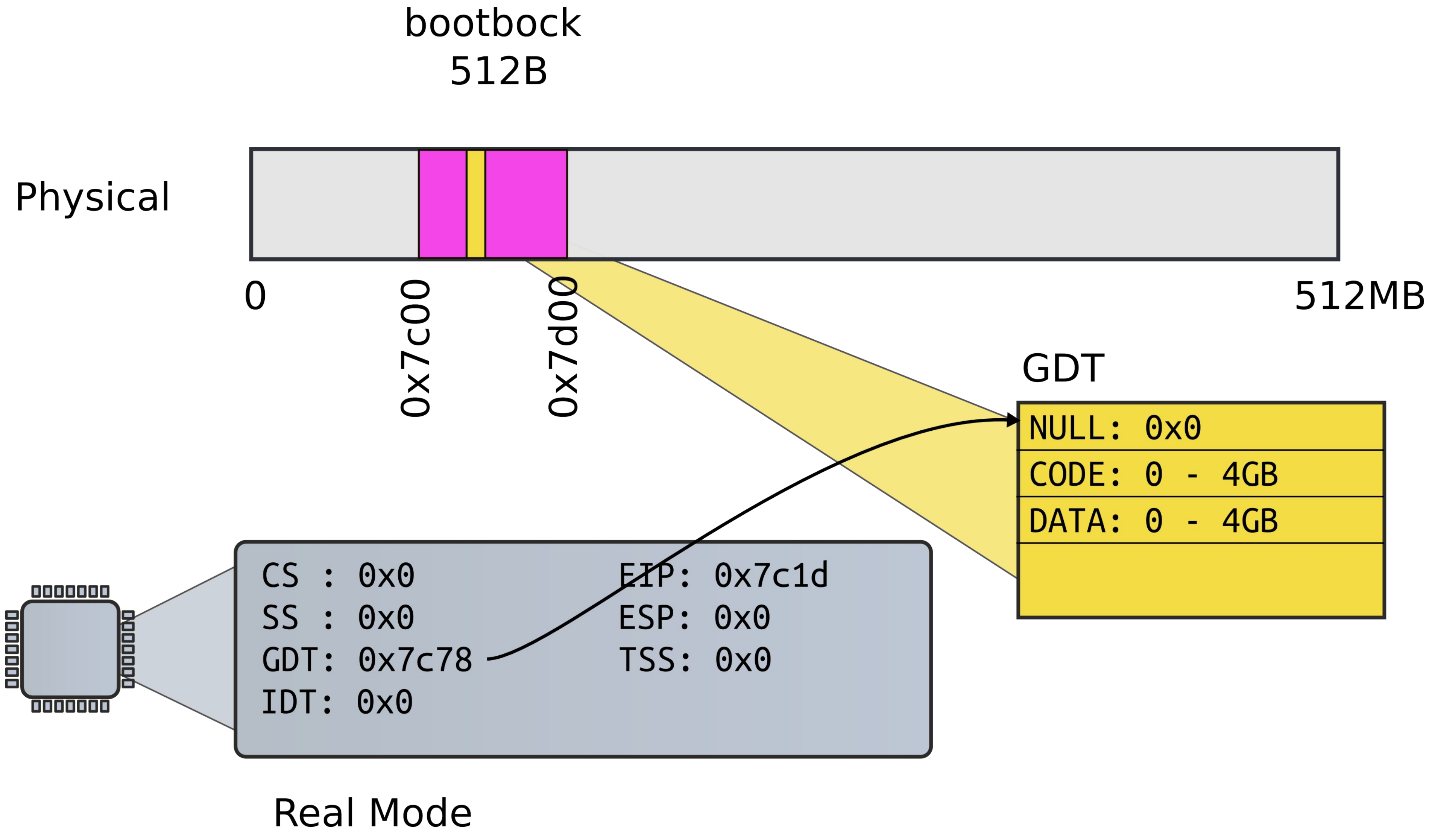
```
start:
cli # BIOS enabled interrupts; disable

# Zero 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
```

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.

```
lgdt gdtdesc
movl %cr0, %eax
orl $CR0_PE, %eax
movl %eax, %cr0
```

Load GDT


bootbock
512B
Physical
0
0x7c00
0x7d00
512MB
GDT
NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
CS : 0x0
SS : 0x0
GDT: 0x7c78
IDT: 0x0
EIP: 0x7c1d
ESP: 0x0
TSS: 0x0
Real Mode

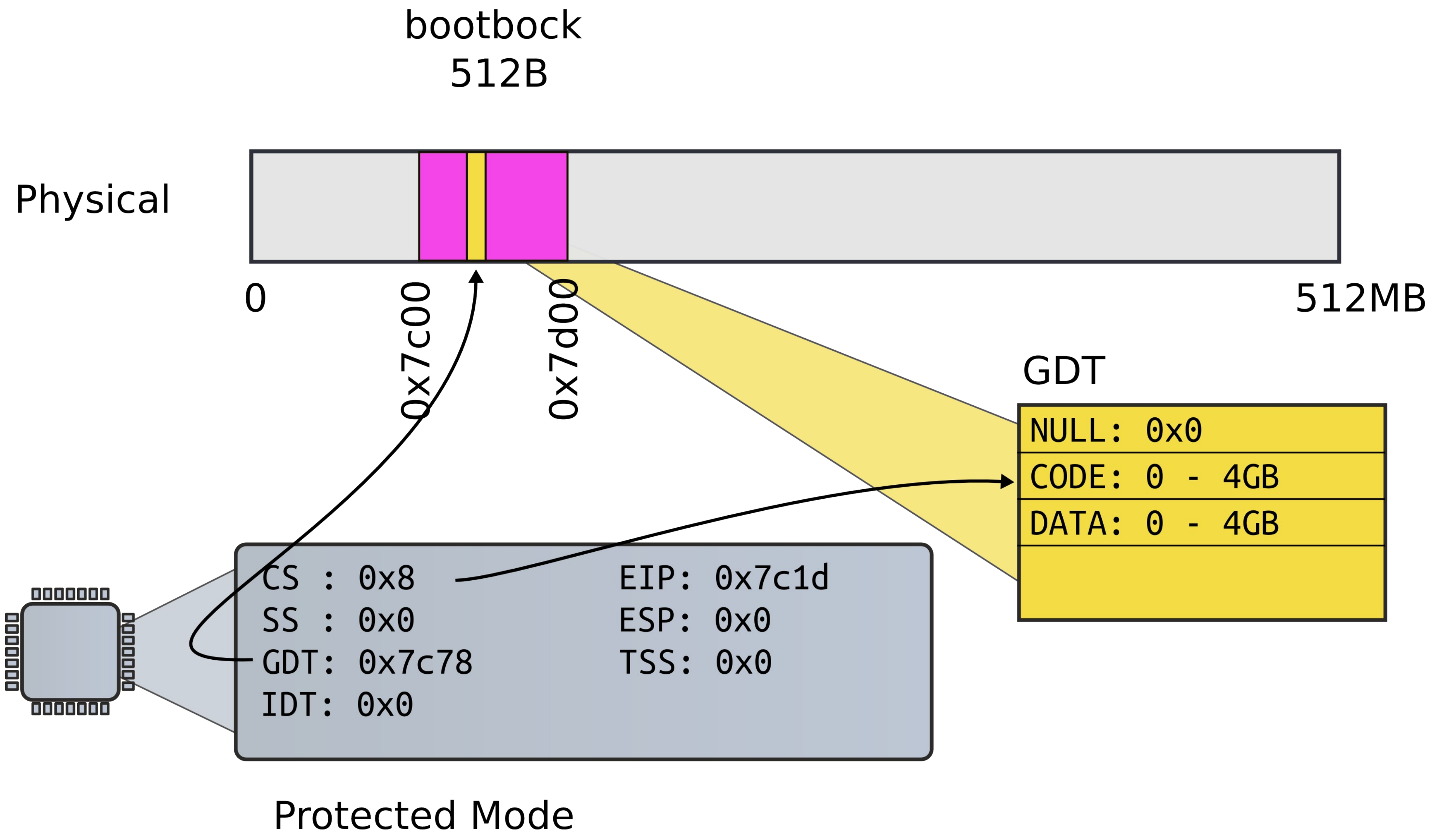
Actual switch

- Use long jump to change code segment

```
8453 ljmp $(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

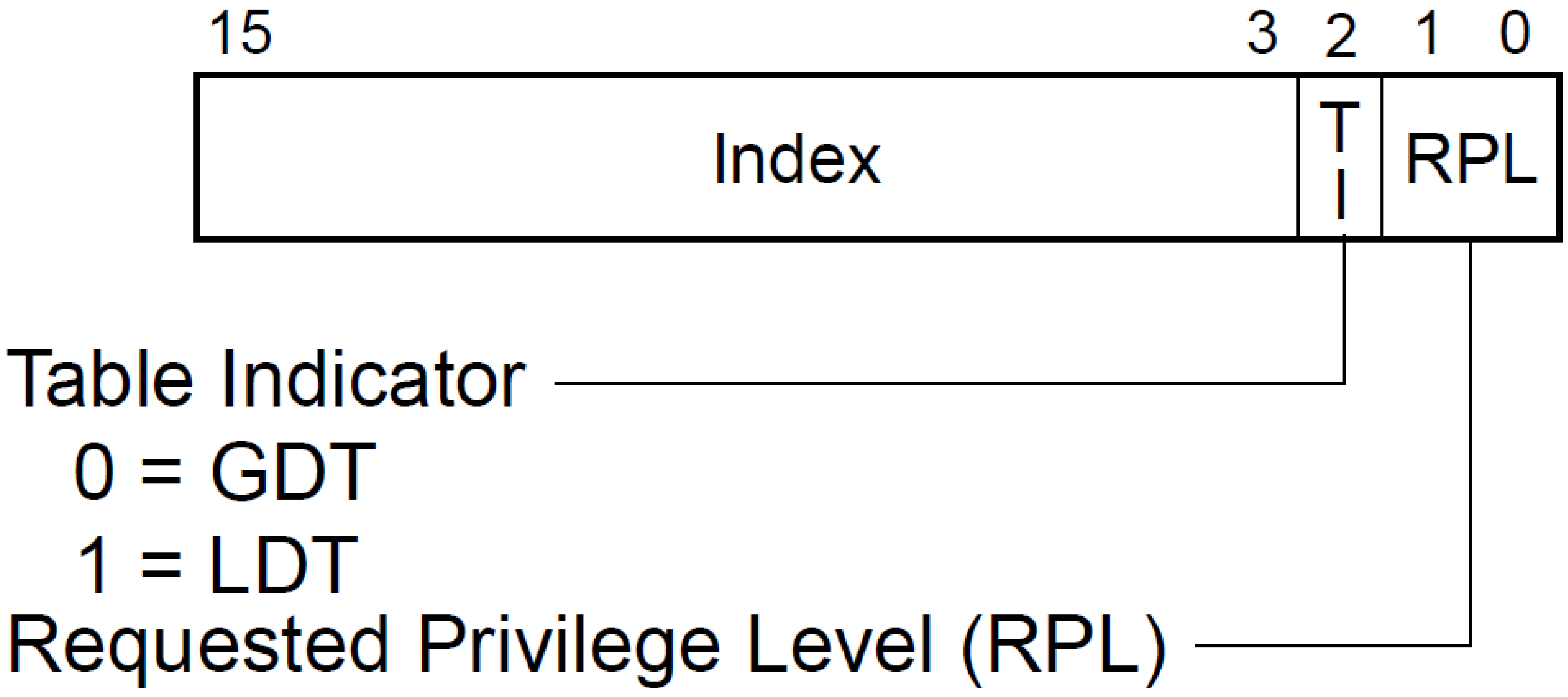


```
NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
```

```
CS : 0x8          EIP: 0x7c1d
SS : 0x0          ESP: 0x0
GDT: 0x7c78       TSS: 0x0
IDT: 0x0
```

Why CS is 0x8, not 0x1?

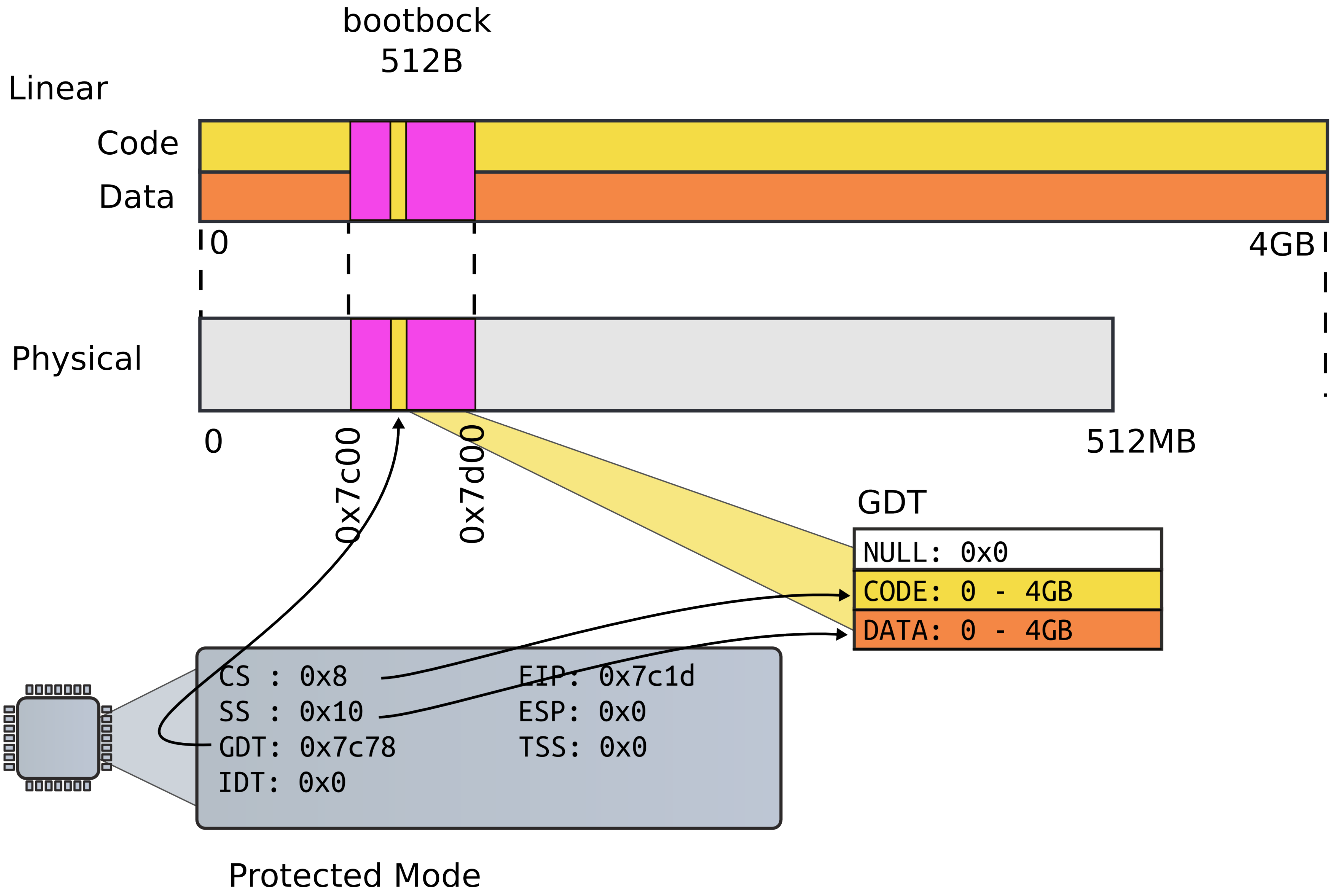
- Segment selector:



Segments

```
start32:
movw $(SEG_KDATA<<3), %ax # Data segment
movw %ax, %ds # -> DS: Data Segment
movw %ax, %es # -> ES: Extra Segment
movw %ax, %ss # -> SS: Stack Segment
movw $0, %ax # Zero segments not in use
movw %ax, %fs # -> FS
movw %ax, %gs # -> GS
```

Segments

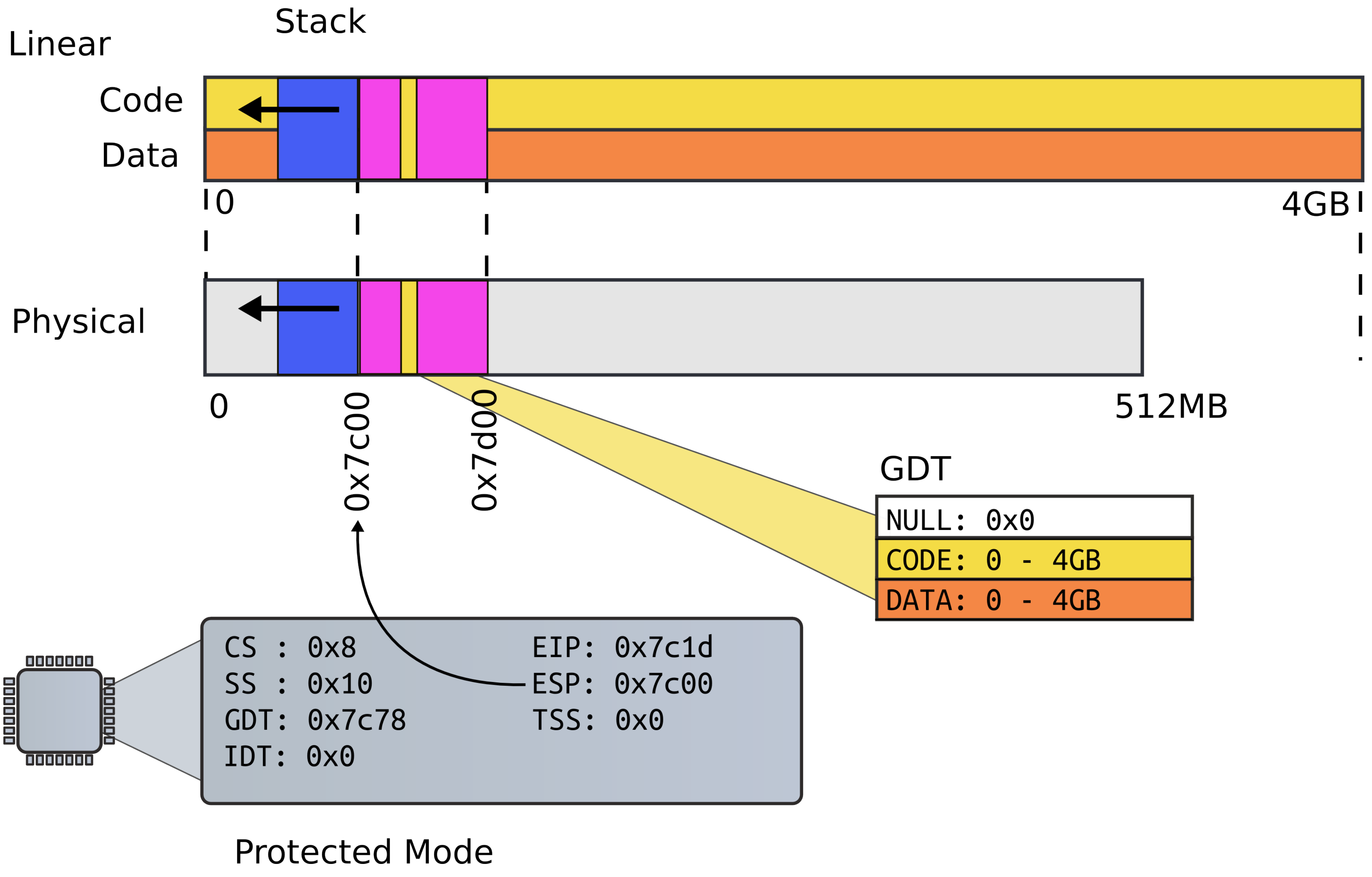

bootbock
512B
Linear
Code
Data
0
4GB
Physical
0
0x7c00
0x7d00
512MB
GDT
NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0
EIP: 0x7c1d
ESP: 0x0
TSS: 0x0
Protected Mode

Setup stack

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

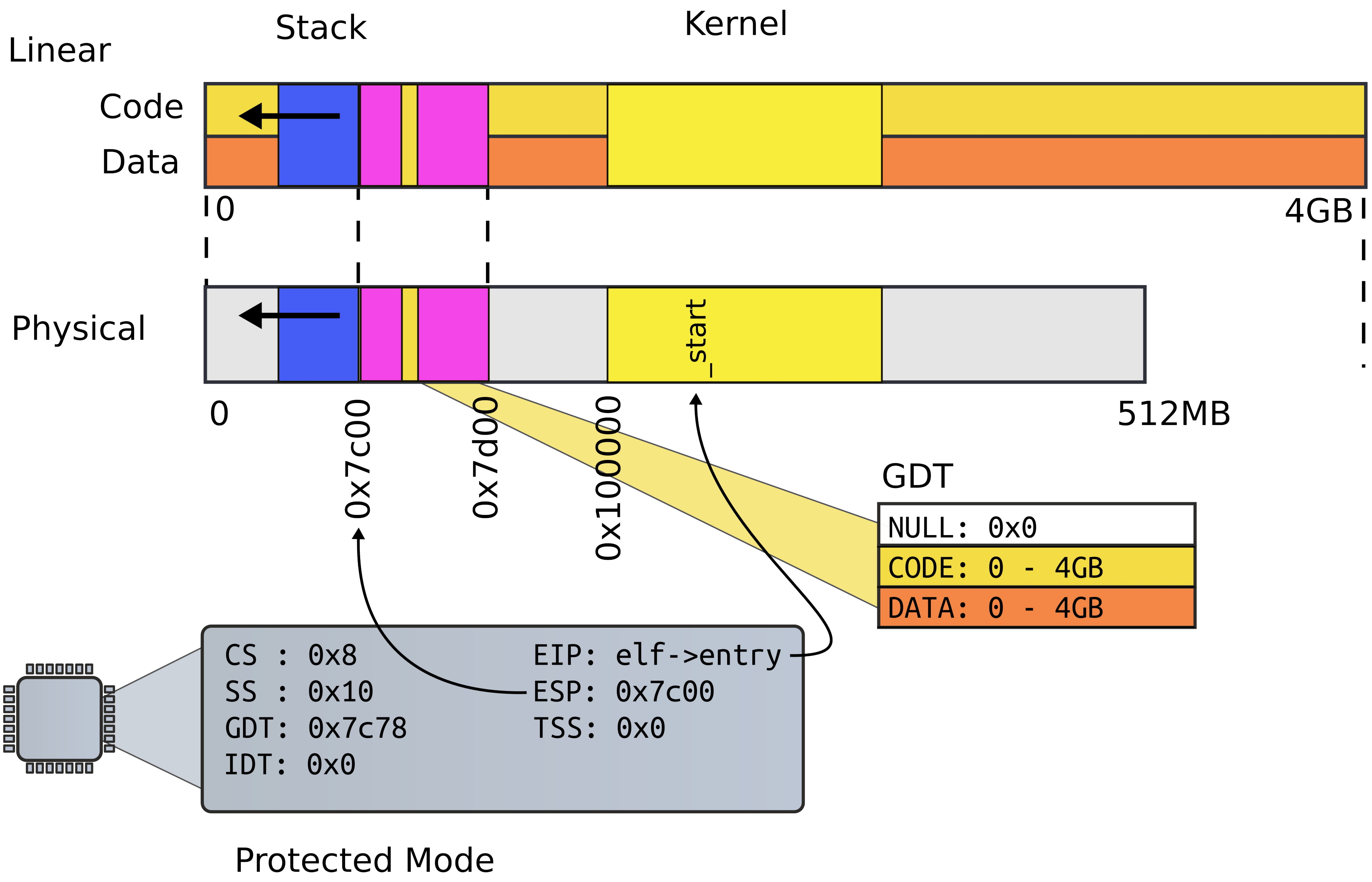
```
8467 movl $start, %esp
8468 call bootmain
```

First stack



```
CS : 0x8          EIP: 0x7c1d
SS : 0x10         ESP: 0x7c00
GDT: 0x7c78       TSS: 0x0
IDT: 0x0
```

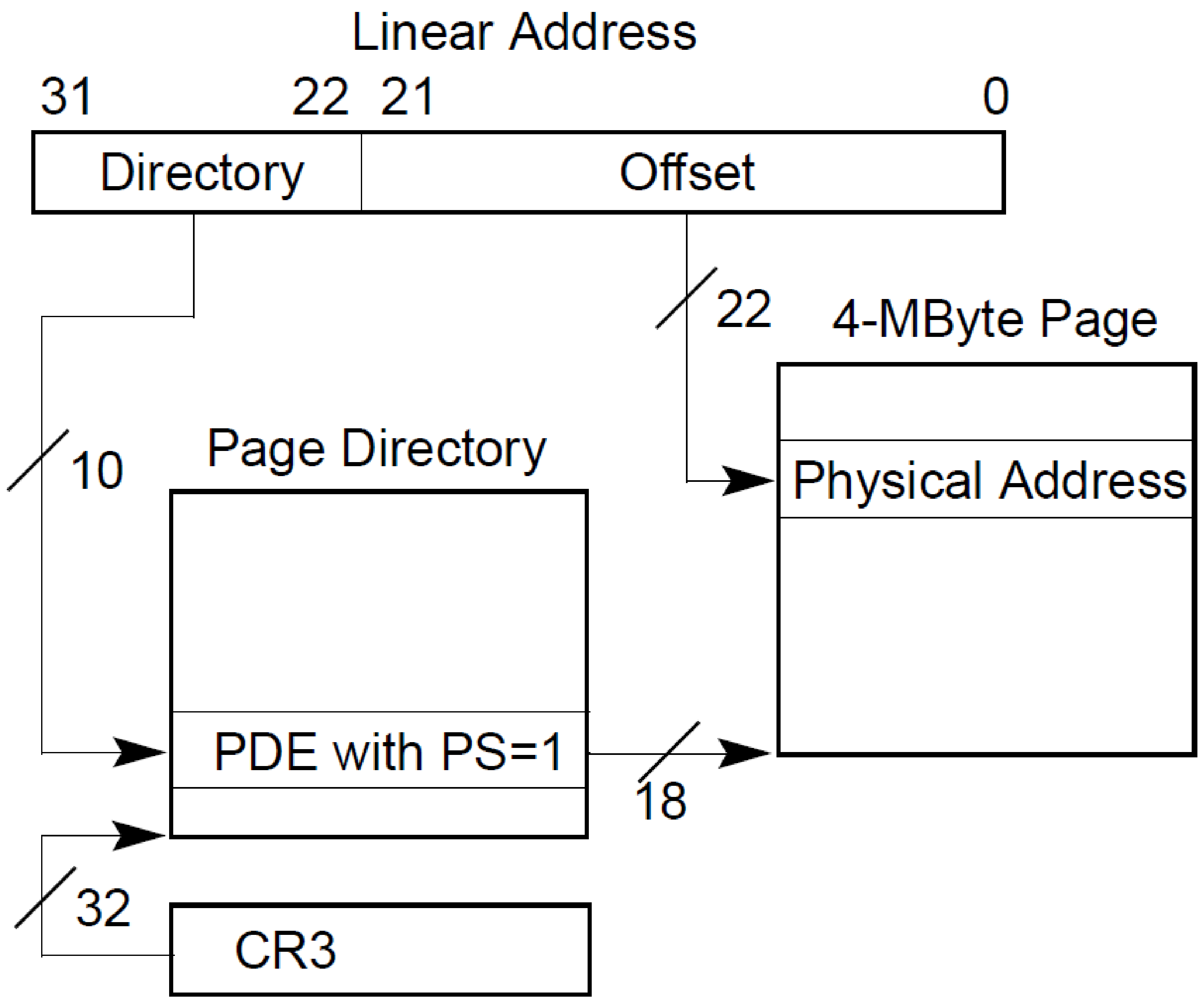
Kernel



```
CS : 0x8          EIP: elf->entry
SS : 0x10         ESP: 0x7c00
GDT: 0x7c78       TSS: 0x0
IDT: 0x0
```

First page table

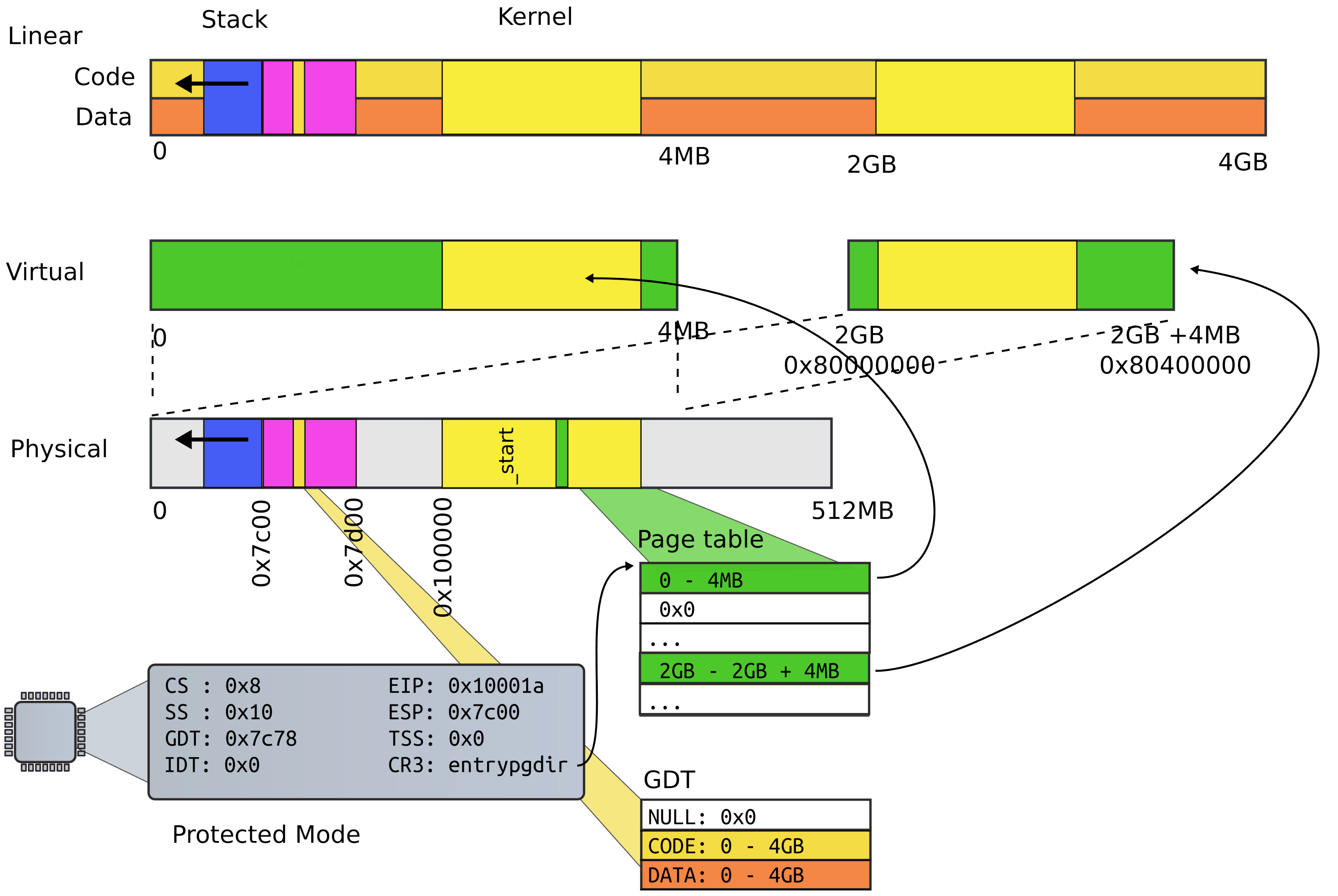
- Two 4MB entries (large pages)
- Entry #0
  - 0x0 – 4MB → 0x0:0x400000
- Entry #960
  - 0x0 – 4MB → 0x8000000:0x80400000


Linear Address
31
22
21
0
Directory
Offset
22
4-MByte Page
10
Page Directory
Physical Address
PDE with PS=1
18
32
CR3

First page table

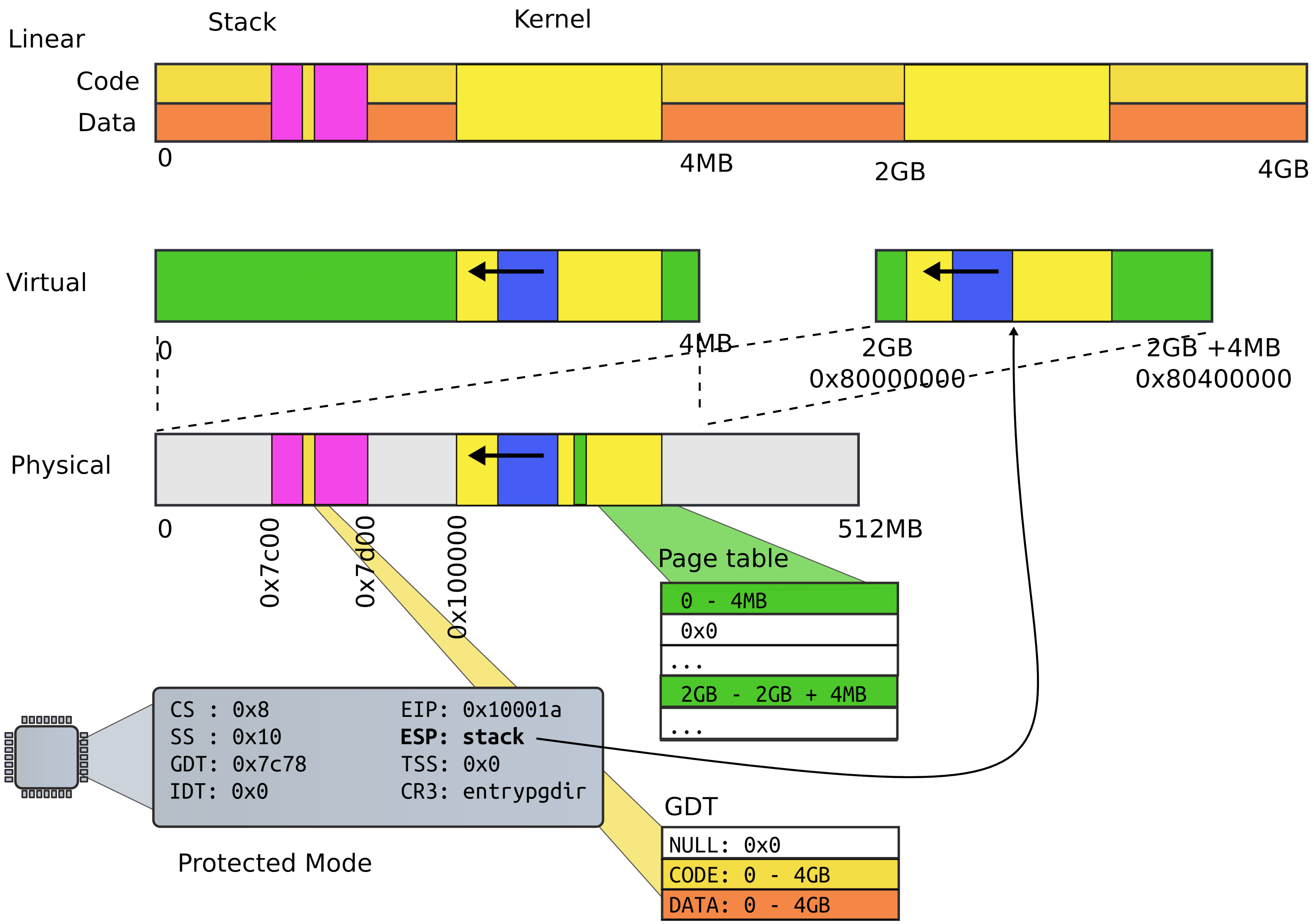
```
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
// Map VA's [0, 4MB) to PA's [0, 4MB)
[0] = (0) | PTE_P | PTE_W | PTE_PS,
// Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
[KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

First page table



```
.globl entry
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



```
# Set up the stack pointer.
movl $(stack + KSTACKSIZE), %esp

# Jump to main(), and switch to executing at
# high addresses. The indirect call is needed because
# the assembler produces a PC-relative instruction
# for a direct jump.
mov $main, %eax
jmp *%eax

.comm stack, KSTACKSIZE
```

Thank you!