143A: Principles of Operating Systems

Lecture 7: System boot

Anton Burtsev
October, 2017

Outline for today

Boot operating system

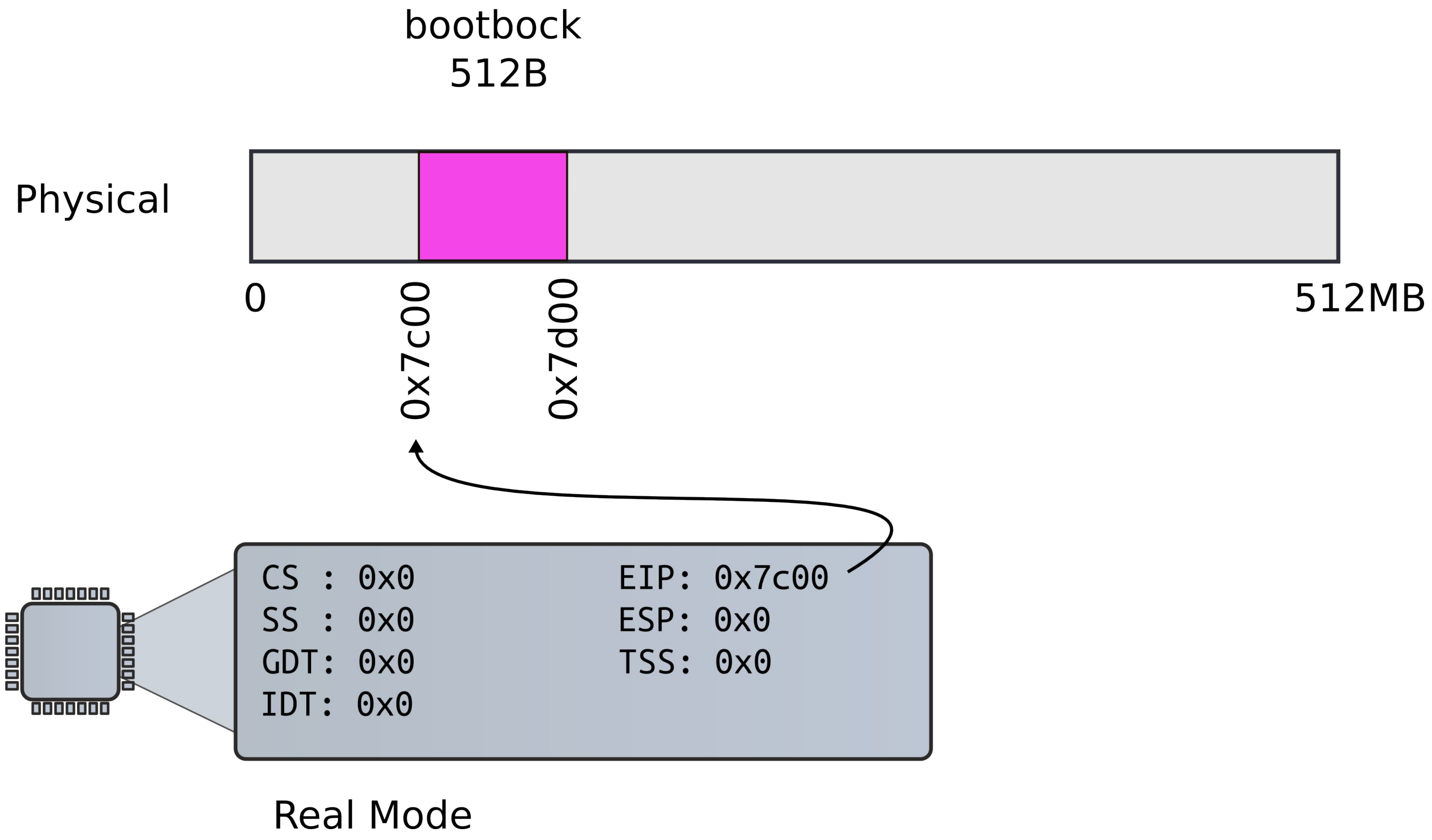
- Setup segments (data and code)
- Switch to protected mode
  - Load GDT (turn segmentation on)
- Setup stack (needed to call C functions)
- Load the kernel from disk into memory
- Setup first page table
  - 2 entries [0 : 4MB] and [2GB : (2GB + 4MB)]
- Setup high-address stack
- Jump to main()
  - Start executing kernel code

What happens when we turn on the power?

- Well it's complicated
  - Intel SGX Explained is a good start (Section 2.13 [1])
- At a high-level a sequence of software pieces initializes the platform
  - Microcode, firmware (BIOS), bootloader

- The most important thing, the OS is not the only software running on the machine
  - And not the most privileged
- Today, at least two layers sit underneath the OS/hypervizor
  - System Management Mode (SMM) (ring -2)
    - Runs below the hypervisor/OS
  - Intel Management Engine (ring -3)
    - Runs on a separate CPU

Bootloader starts

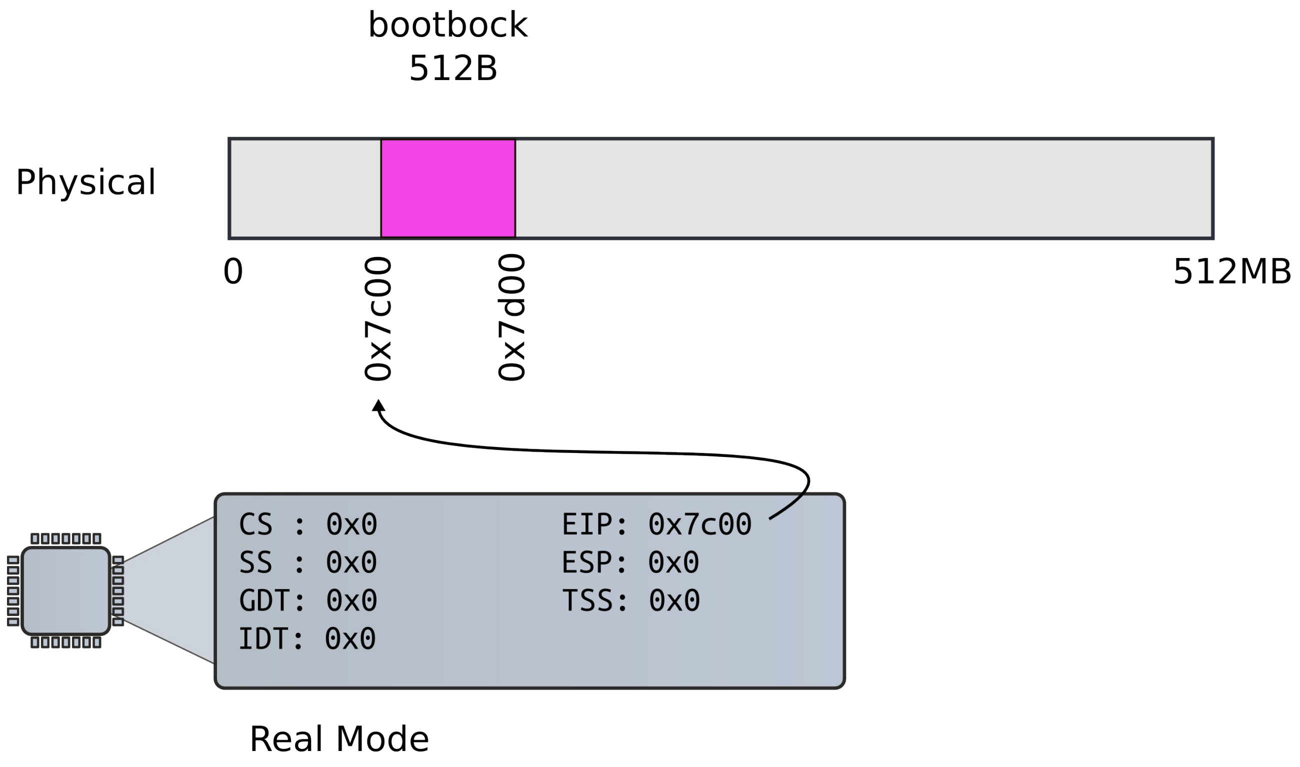


Bootloader starts

```
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
```

Why start happens to be 0x7c00?

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

```



```
CS : 0x0          EIP: 0x7c00
SS : 0x0          ESP: 0x0
GDT: 0x0          TSS: 0x0
IDT: 0x0
```

Linker are told so through the Makefile

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

```

```
bootblock: bootasm.S bootmain.c
        $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
        $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
        $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
        $(OBJDUMP) -S bootblock.o > bootblock.asm
        $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
        ./sign.pl bootblock
```

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
```

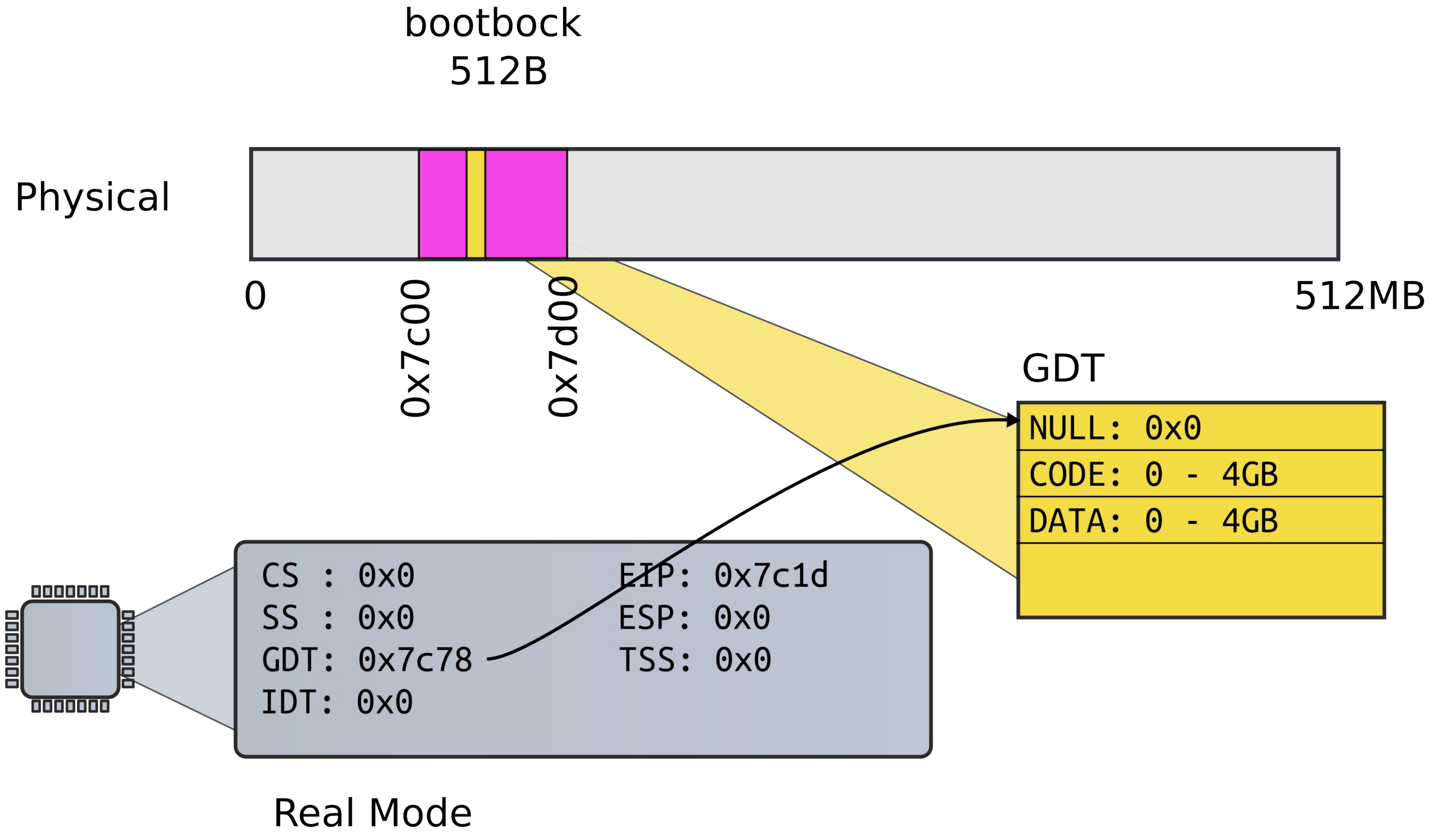
How GDT is defined

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

gdtdesc:
  .word (gdtdesc - gdt - 1) # sizeof(gdt) - 1
  .long gdt
```

How GDT is defined

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

gdtdesc:
  .word (gdtdesc - gdt - 1) # sizeof(gdt) - 1
  .long gdt
```

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

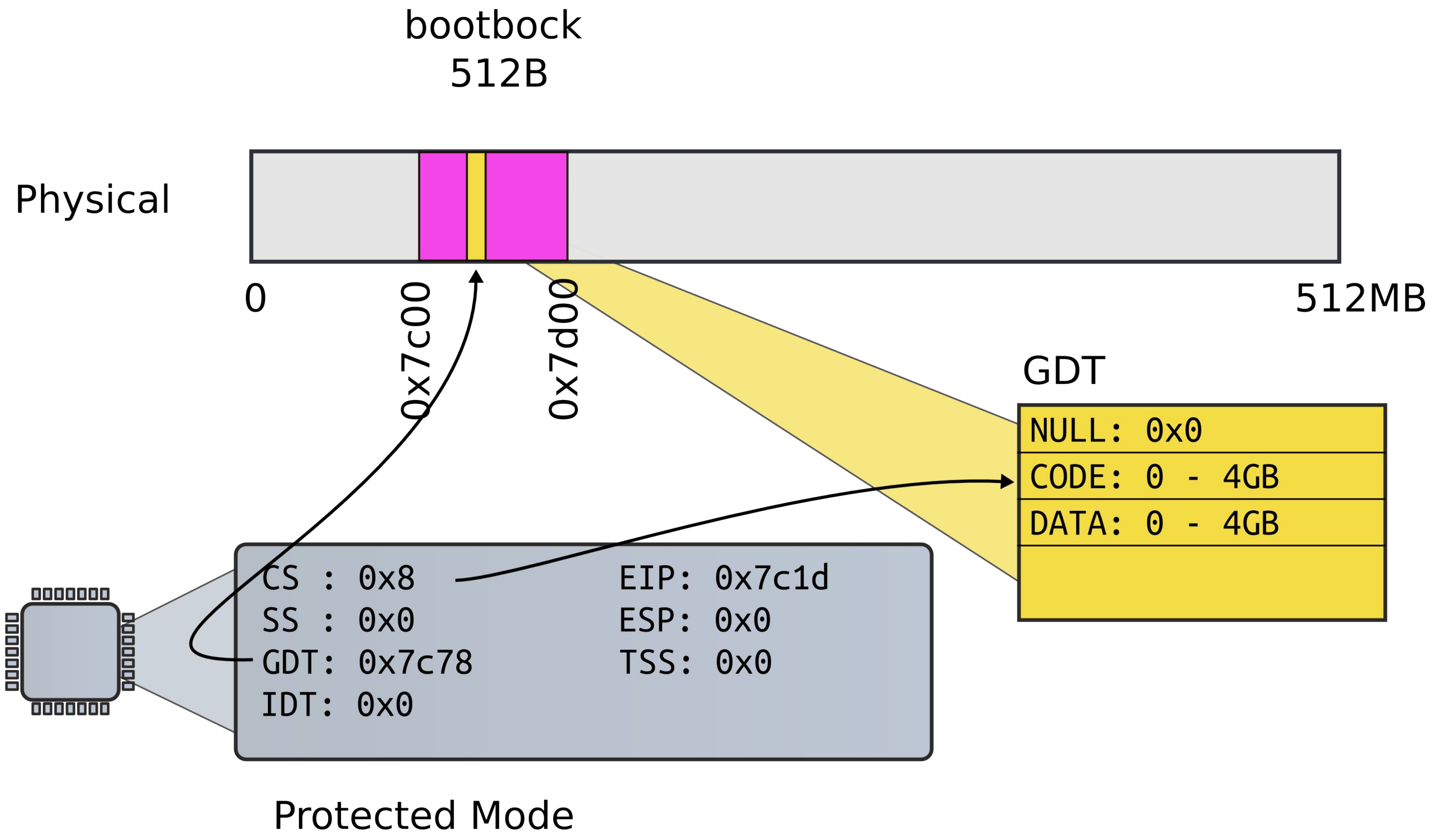
Actual switch

- Use long jump to change code segment

```
9153 ljmp $(SEG_KCODE<<3), $start32
```

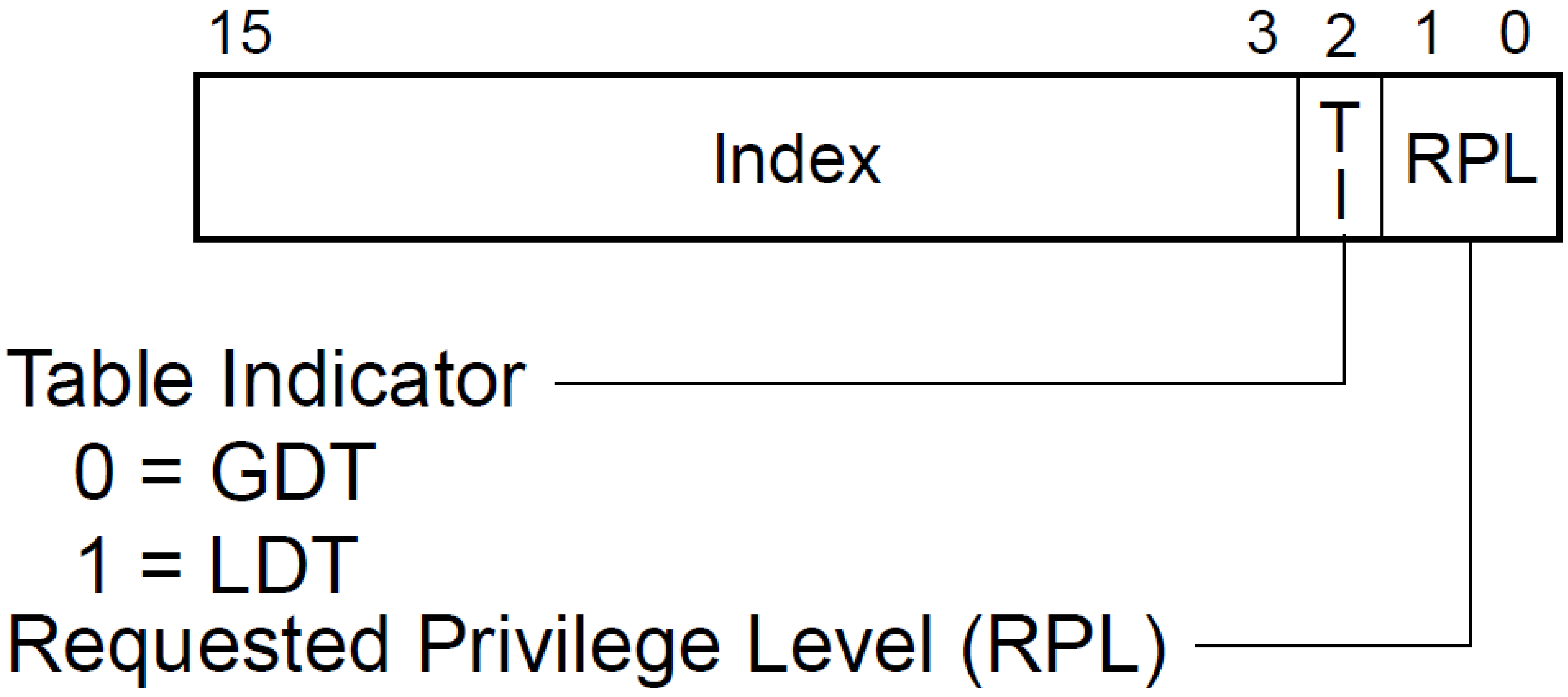
  - Explicitly specify code segment, and address
  - Segment is 0b1000 (0x8)

Long jump


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

Why CS is 0x8, not 0x1?

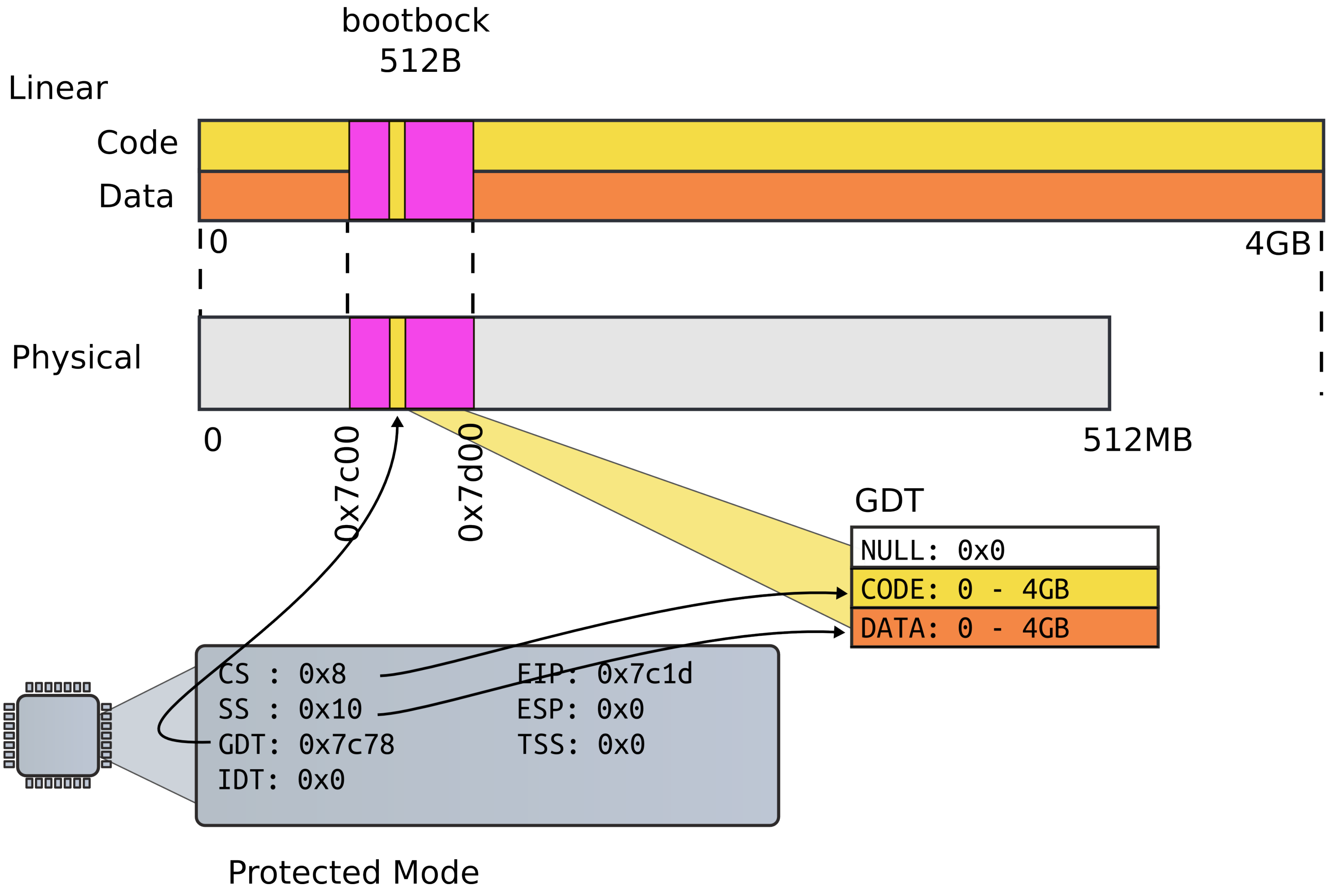
- Segment selector:



Segments

```
.code32 # Tell assembler to generate 32-bit code now.
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
```

Segments



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

Setup stack

- Why do we need a stack?

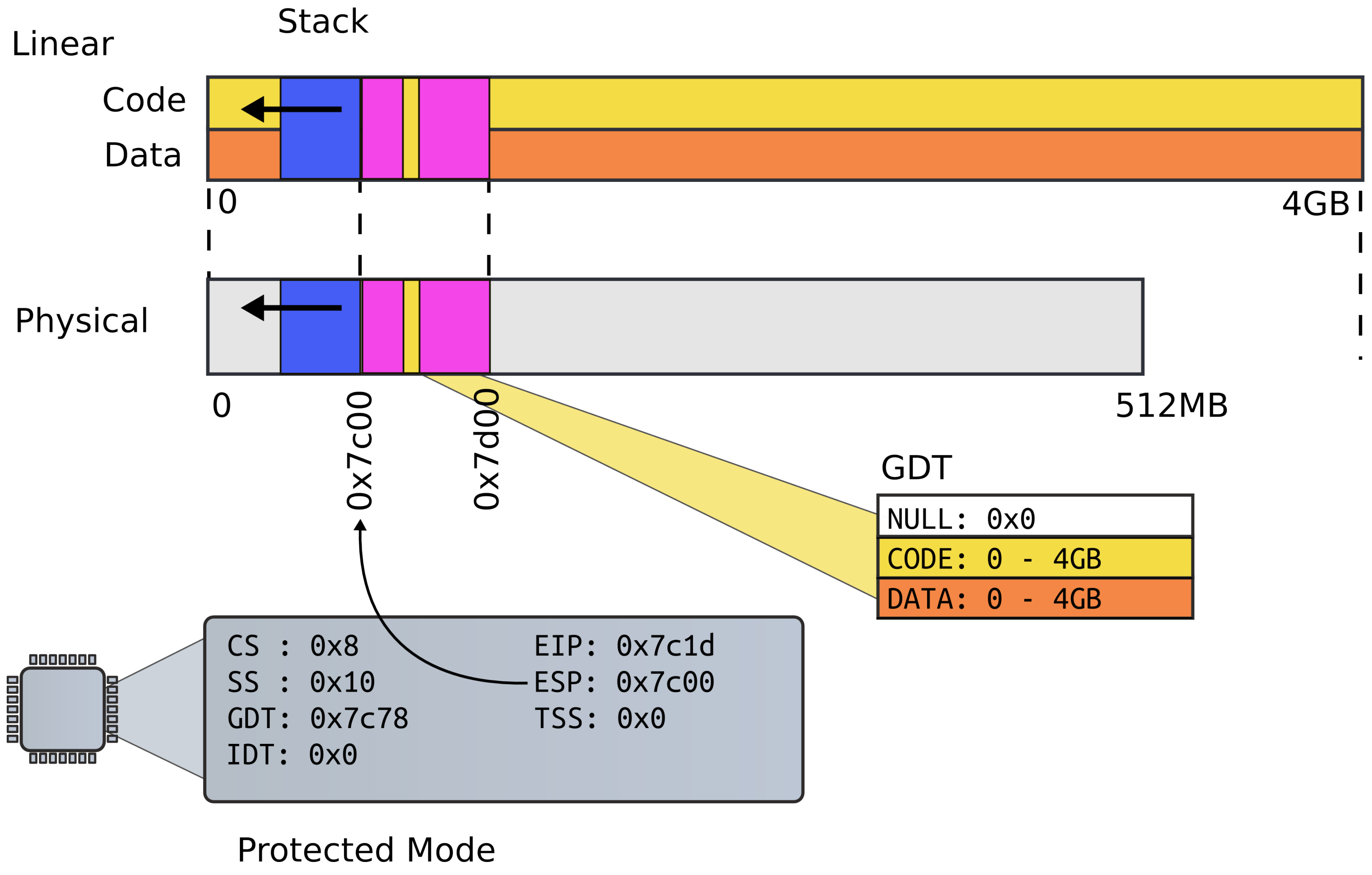
```
9166 movl $start, %esp
9167 call bootmain
```

Setup stack

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

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

First stack



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

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

Invoke first C function

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

bootmain(): read kernel from disk

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

    elf = (struct elfhdr*)0x10000; // scratch space

    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);

    // Is this an ELF executable?
    if(elf->magic != ELF_MAGIC)
        return; // let bootasm.S handle error

```

```c

    // Load each program segment (ignores ph flags).
    ph = (struct proghdr*)((uchar*)elf + elf->phoff);
    eph = ph + elf->phnum;
    for(; ph < eph; ph++){
        pa = (uchar*)ph->paddr;
        readseg(pa, ph->filesz, ph->off);
        if(ph->memsz > ph->filesz)
            stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
    }

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

bootmain(): read kernel from disk

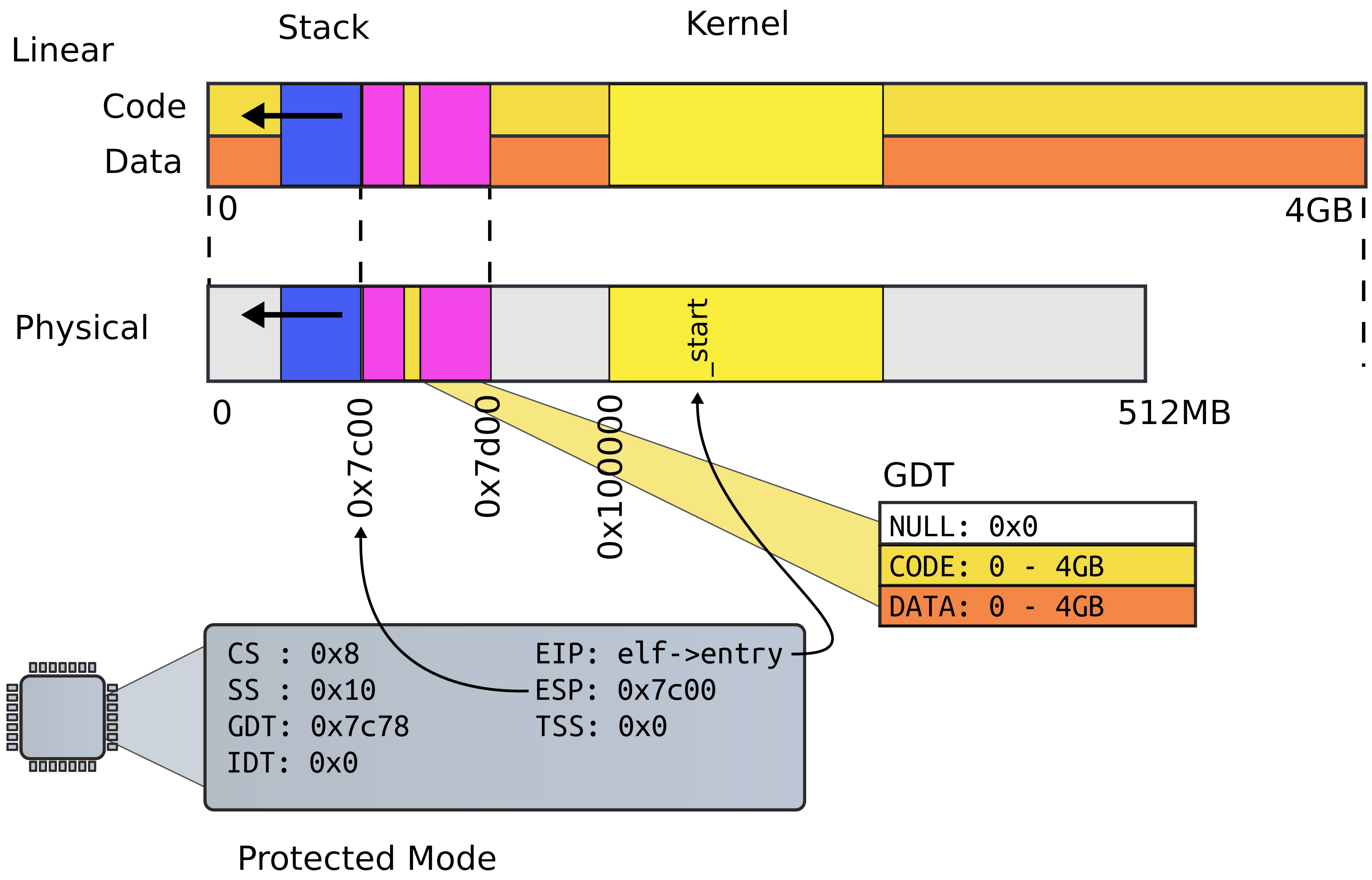
How do we read disk?

```

// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
    // Issue command.
    waitdisk();
    outb(0x1F2, 1); // count = 1
    outb(0x1F3, offset);
    outb(0x1F4, offset >> 8);
    outb(0x1F5, offset >> 16);
    outb(0x1F6, (offset >> 24) | 0xE0);
    outb(0x1F7, 0x20); // cmd 0x20 - read sectors

    // Read data.
    waitdisk();
    insl(0x1F0, dst, SECTSIZE/4);
}
```

How do we read disk (cont)?

```
void
waitdisk(void)
{
    // Wait for disk ready.
    while((inb(0x1F7) & 0xC0) != 0x40)
    ;
}

```

Kernel



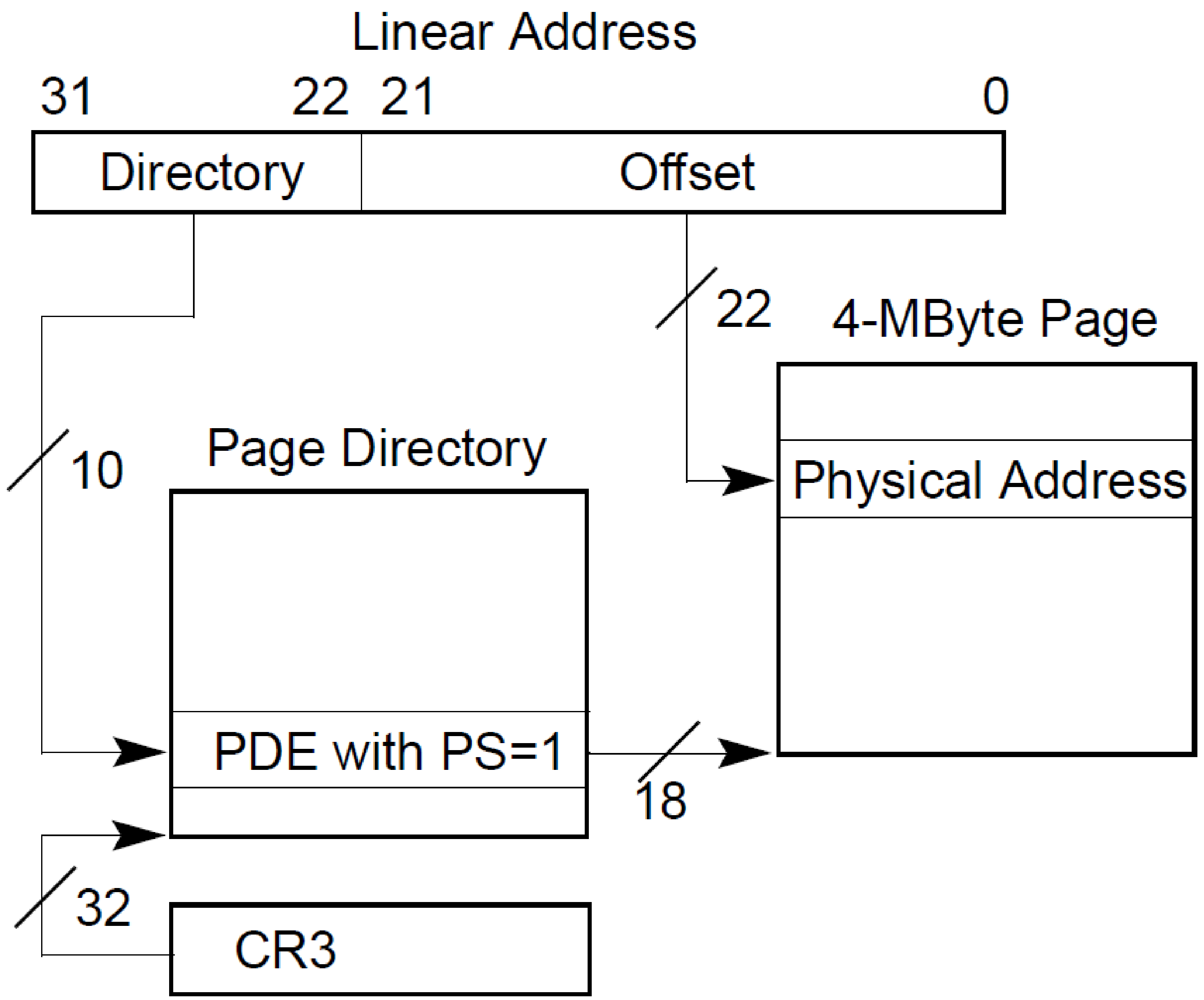
Linear
Stack
Kernel
Code
Data
0
4GB
Physical
_start
0
0x7c00
0x7d00
0x100000
512MB
GDT
NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0
EIP: elf->entry
ESP: 0x7c00
TSS: 0x0
Protected Mode

```
.globl entry
# By convention, the _start symbol specifies the ELF entry point.
# Since we haven’t set up virtual memory yet, our entry point is
# the physical address of ’entry’.
.globl _start
_start = V2P_WO(entry)

# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
# Turn on page size extension for 4Mbyte pages
   movl %cr4, %eax
   orl $(CR4_PSE), %eax
   movl %eax, %cr4
```

entry(): kernel ELF entry

Set up page directory

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


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

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

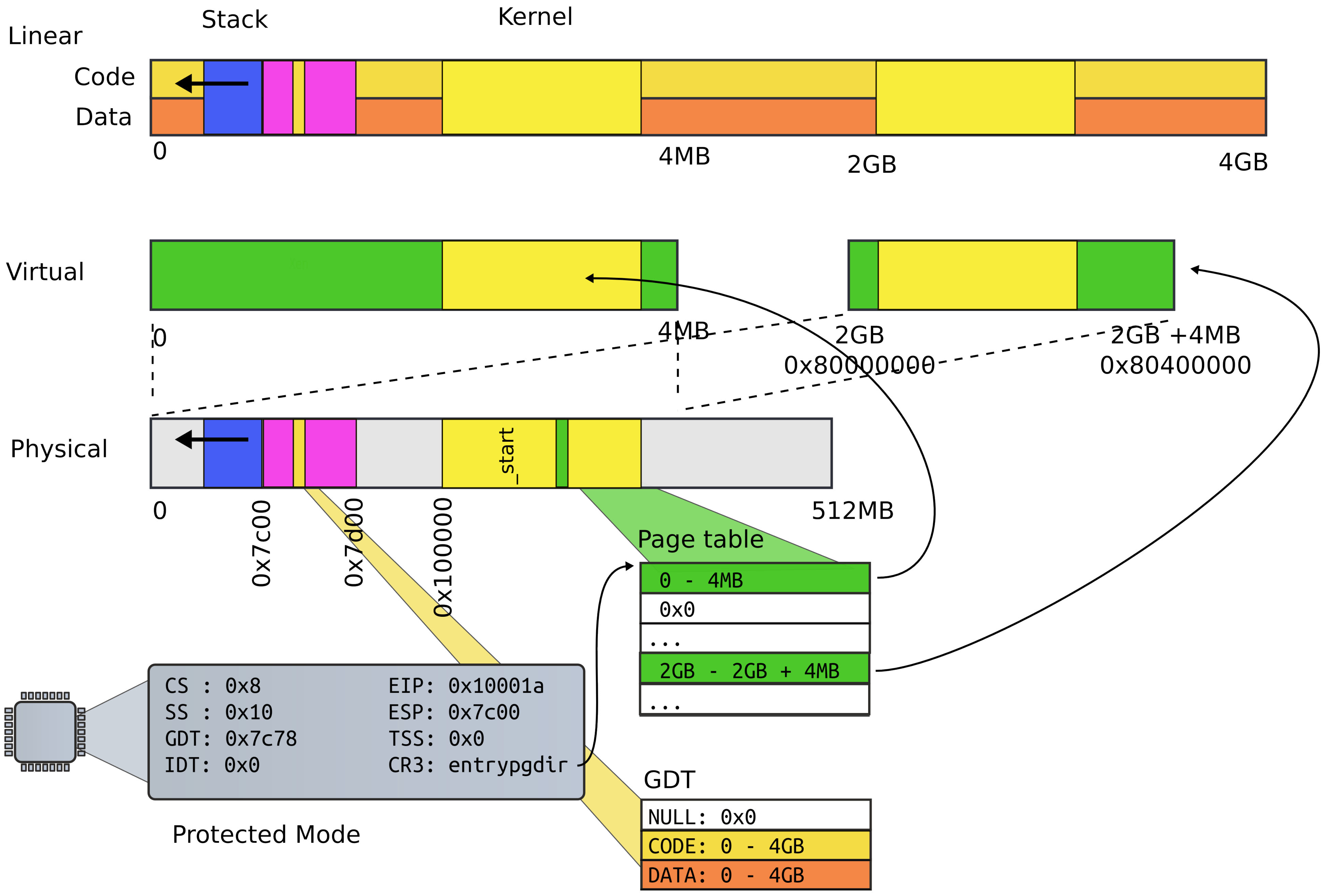
```c
// The boot page table used in entry.S and entryother.S.
// Page directories (and page tables) must start on page boundaries,
// hence the __aligned__ attribute.
// PTE_PS in a page directory entry enables 4Mbyte pages.

__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

First page table (cont)

```
// Page directory and page table constants.
#define NPDENTRIES 1024
```

First page table



```
CS : 0x8        EIP: 0x10001a
SS : 0x10       ESP: 0x7c00
GDT: 0x7c78     TSS: 0x0
IDT: 0x0        CR3: entrypgdir
```

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

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

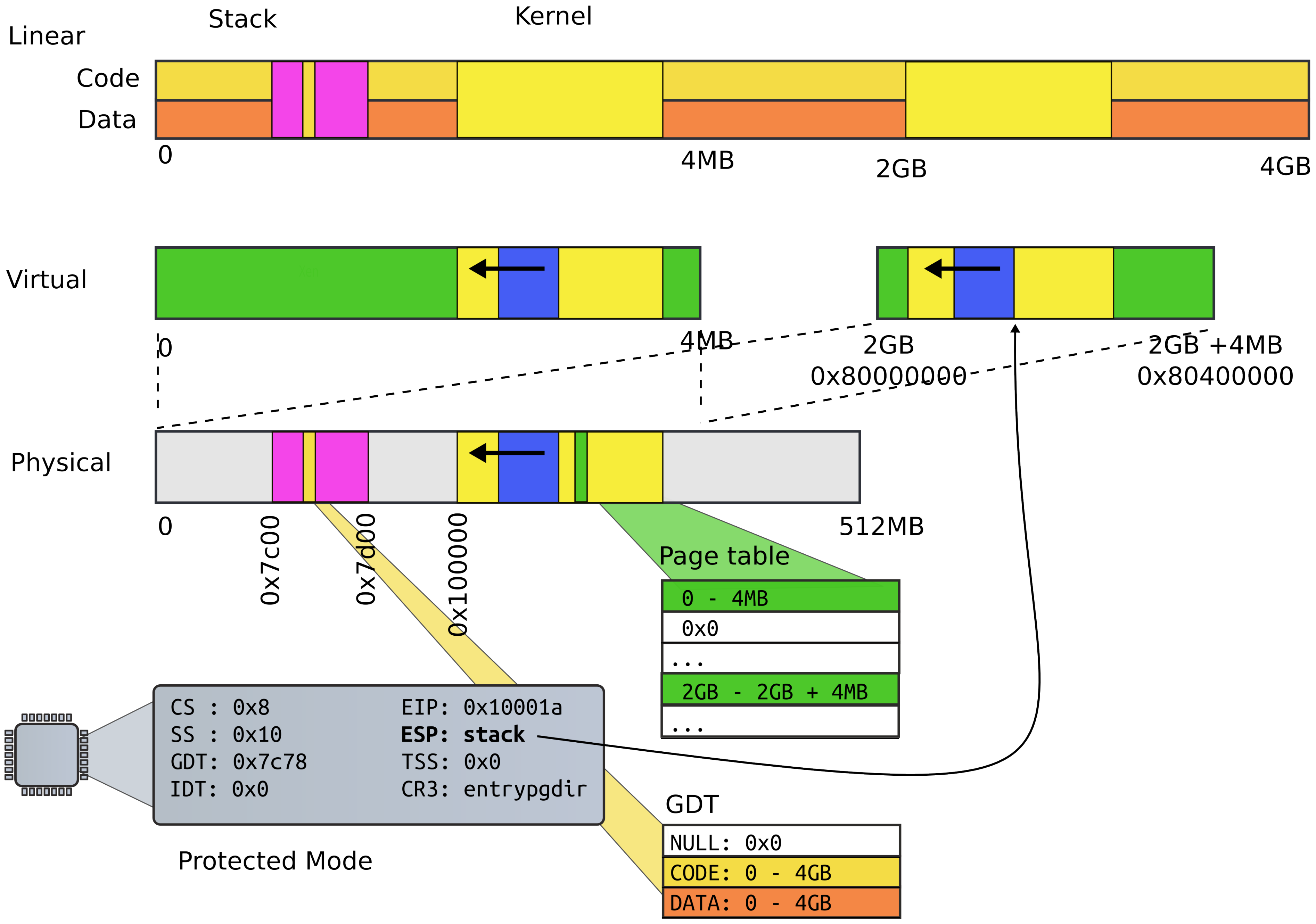
Turn on paging

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

High address stack (4K)

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

...
.comm stack, KSTACKSIZE


#define KSTACKSIZE 4096 // size of
                per-process kernel stack
```

High address stack (4K)



```
CS : 0x8             EIP: 0x10001a
SS : 0x10            ESP: stack
GDT: 0x7c78          TSS: 0x0
IDT: 0x0             CR3: entrypgdir
```

Page table

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

GDT

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

Jump to main()

```
# 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

```

Running in main()

```c
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    cprintf("\ncpu%d: starting xv6\n\n", cpunum());
...
}
```

References

- [1] Costan, Victor, and Srinivas Devadas. "Intel SGX Explained." IACR Cryptology ePrint Archive 2016 (2016): 86. https://eprint.iacr.org/2016/086.pdf

Questions?