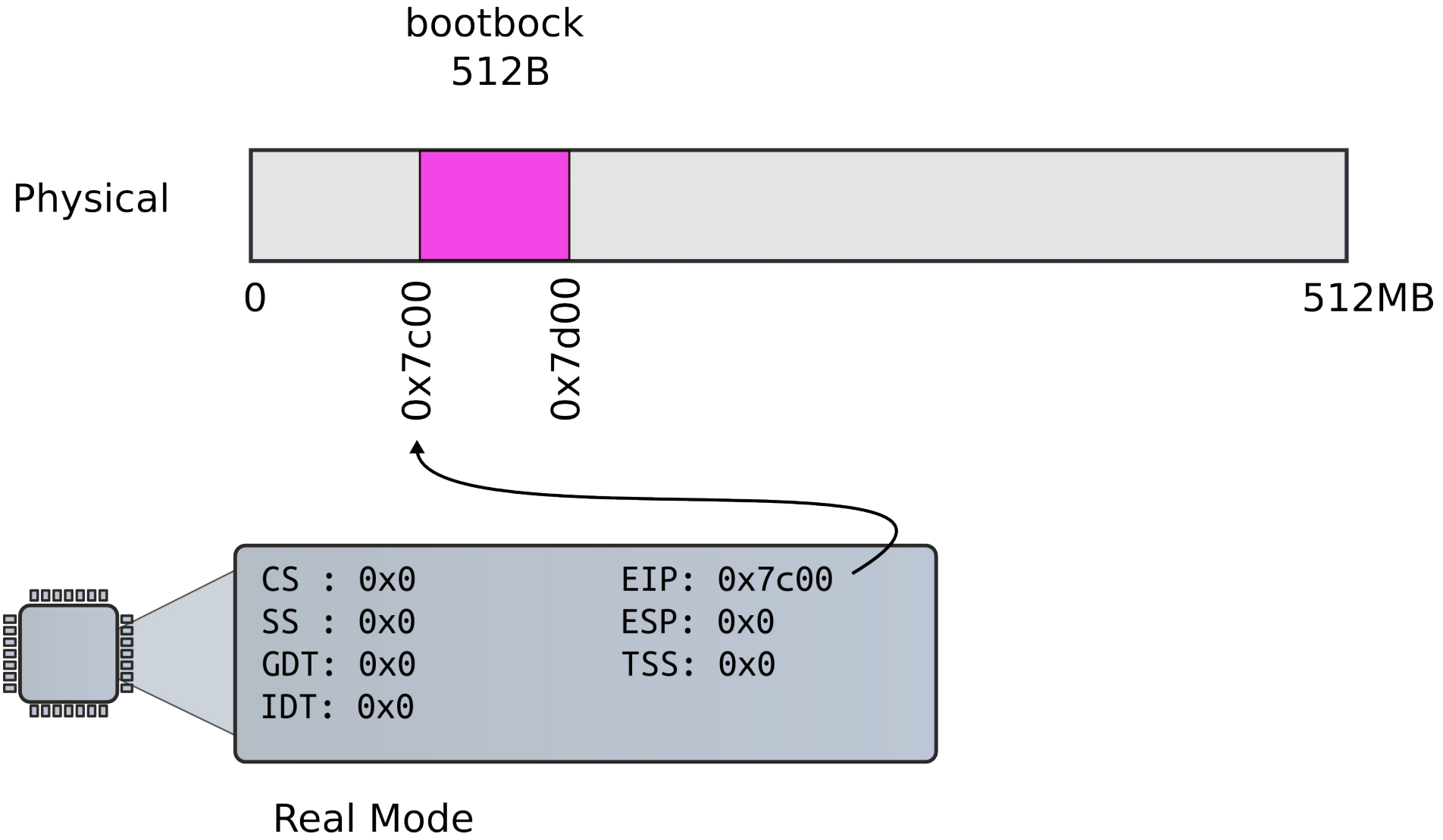
143A: Principles of Operating Systems

Lecture 7: System boot

Anton Burtsev
January, 2017

Bootloader starts



Bootloader starts

```
9111 start:
9112 cli # BIOS enabled interrupts; disable
9113
9114 # Zero data segment registers DS,ES,and SS.
9115 xorw %ax,%ax # Set %ax to zero
9116 movw %ax,%ds # -> Data Segment
9117 movw %ax,%es # -> Extra Segment
9118 movw %ax,%ss # -> Stack Segment
```

Switch to protected mode

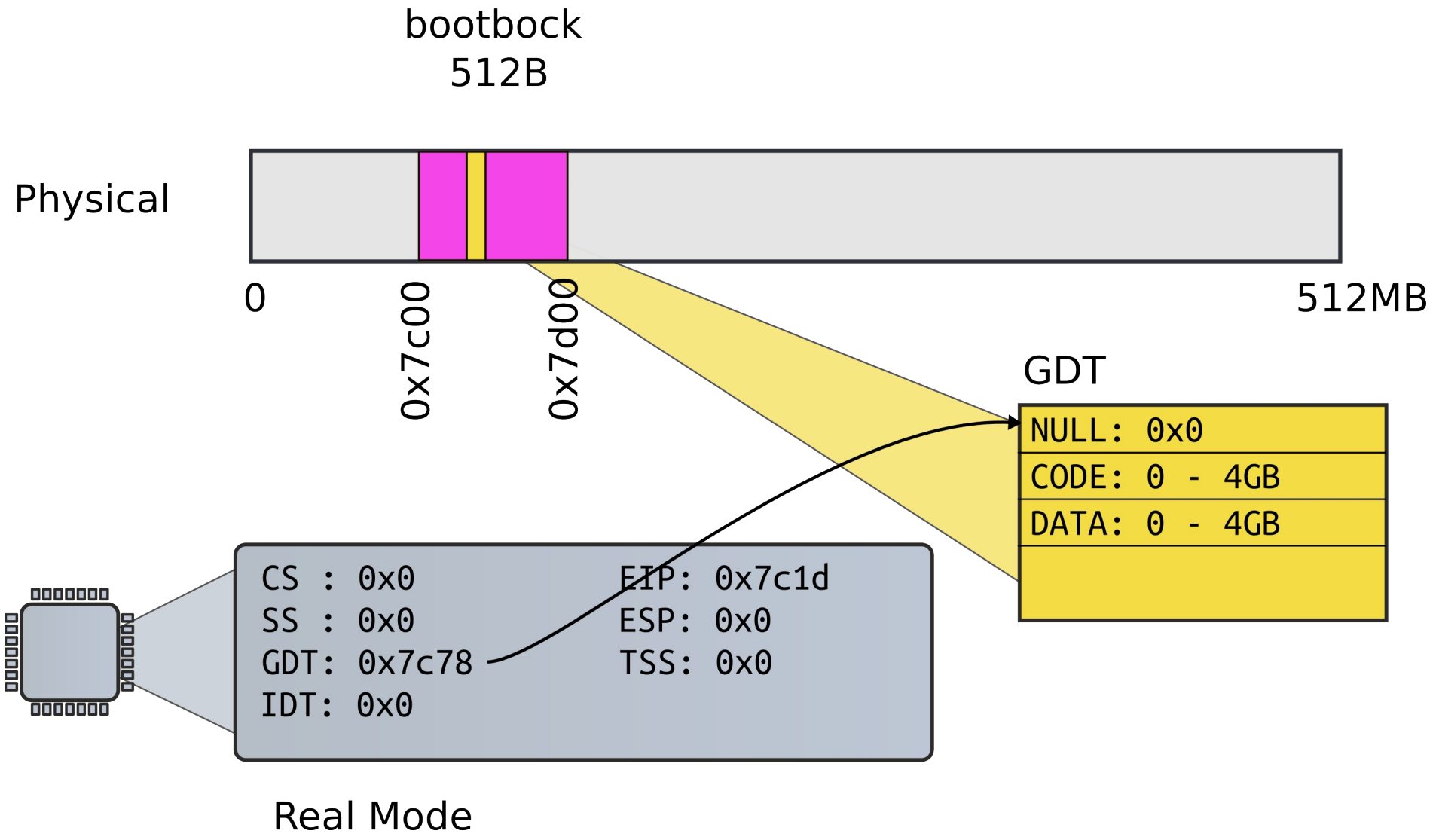
- Switch from real to protected mode
  - Use a bootstrap GDT that makes virtual addresses map directly to physical addresses so that the effective memory map doesn't change during the transition.

```
9141 lgdt gdtdesc

9142 movl %cr0, %eax

9143 orl $CR0_PE, %eax

9144 movl %eax, %cr0
```

How GDT is defined

```
9180 # Bootstrap GDT
9181 .p2align 2 # force 4 byte alignment
9182 gdt:
9183 SEG_NULLASM # null seg
9184 SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg
9185 SEG_ASM(STA_W, 0x0, 0xffffffff) # data seg
9186
9187 gdtdesc:
9188 .word (gdtdesc - gdt - 1) # sizeof(gdt) - 1
9189 .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

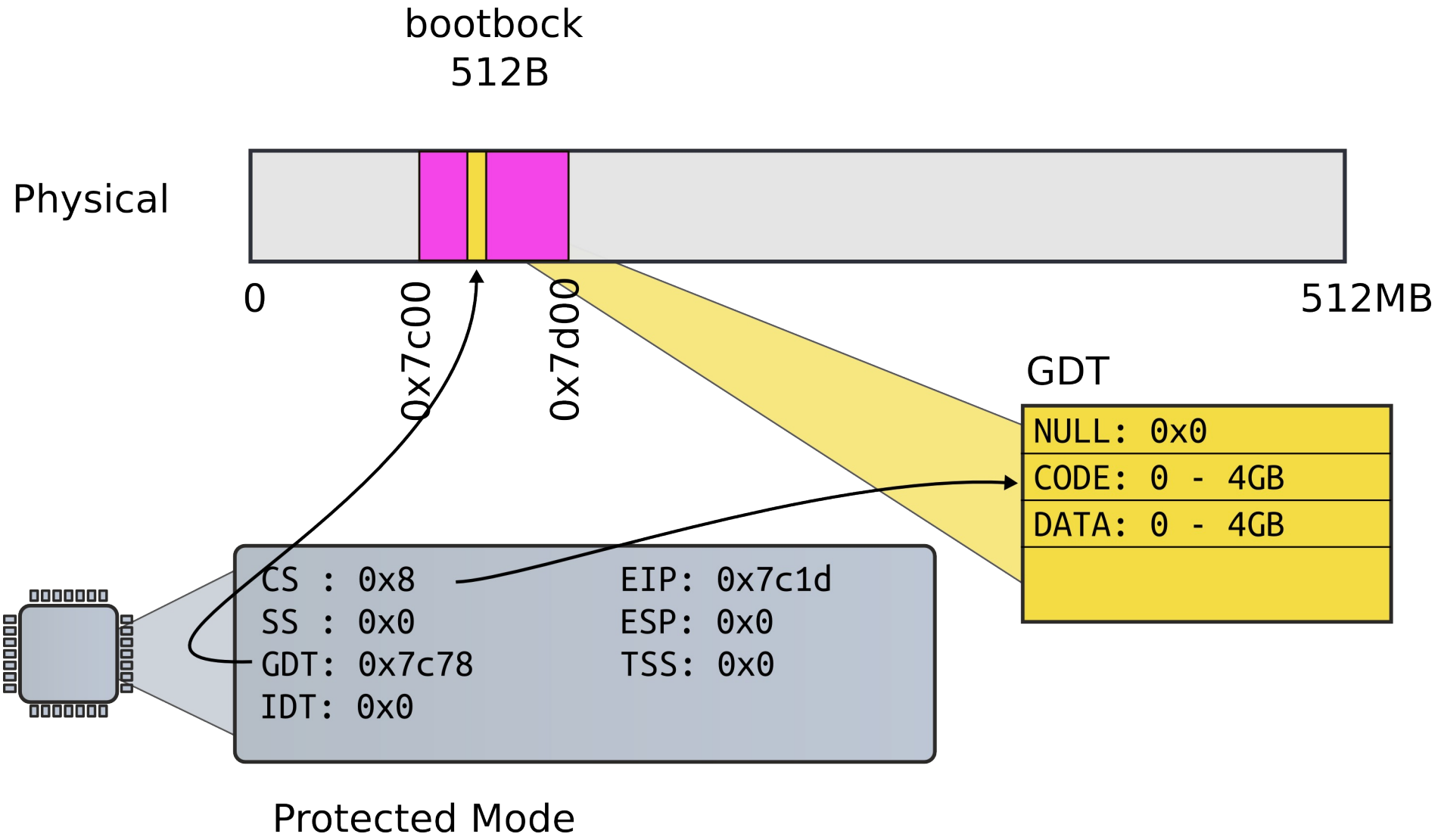
Actual switch

- Use long jump to change code segment

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

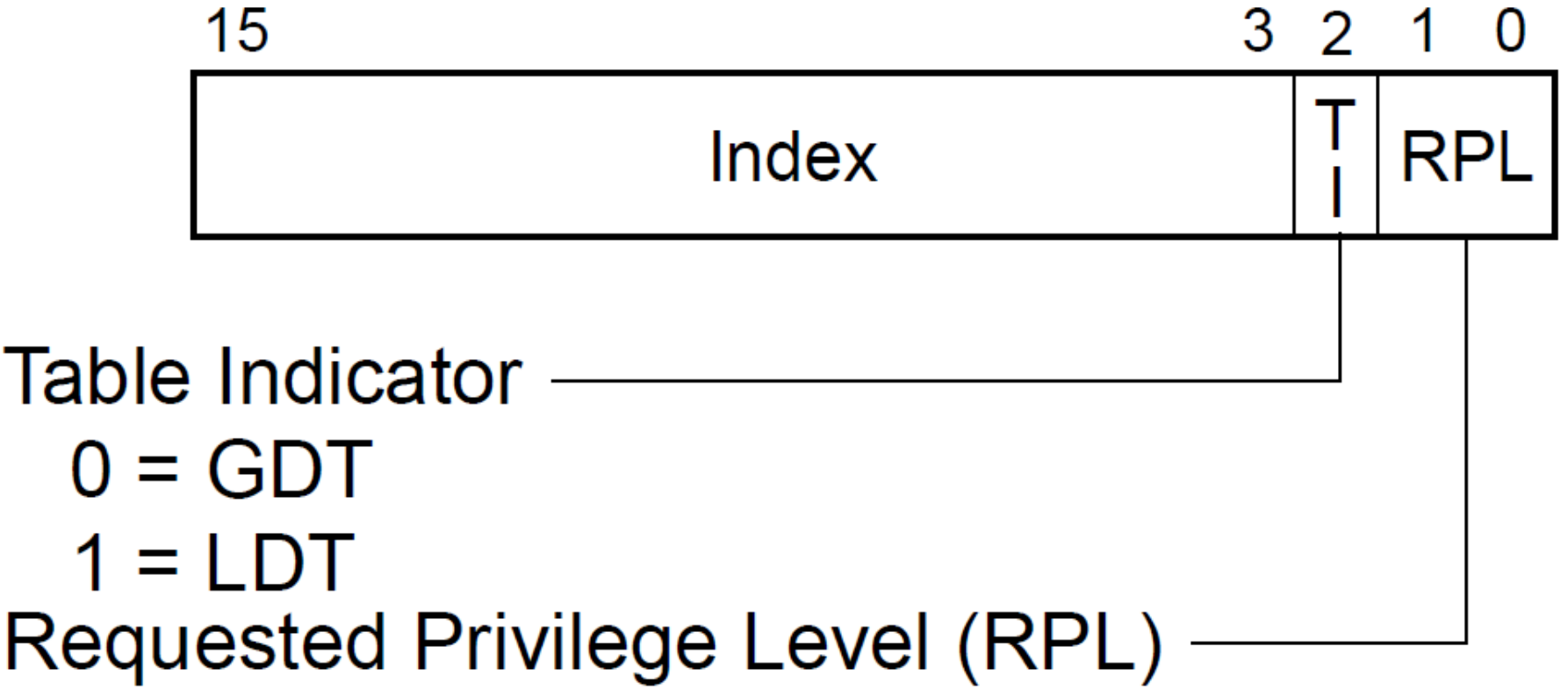
  - Explicitly specify code segment, and address
  - Segment is 0b1000 (0x8)

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?

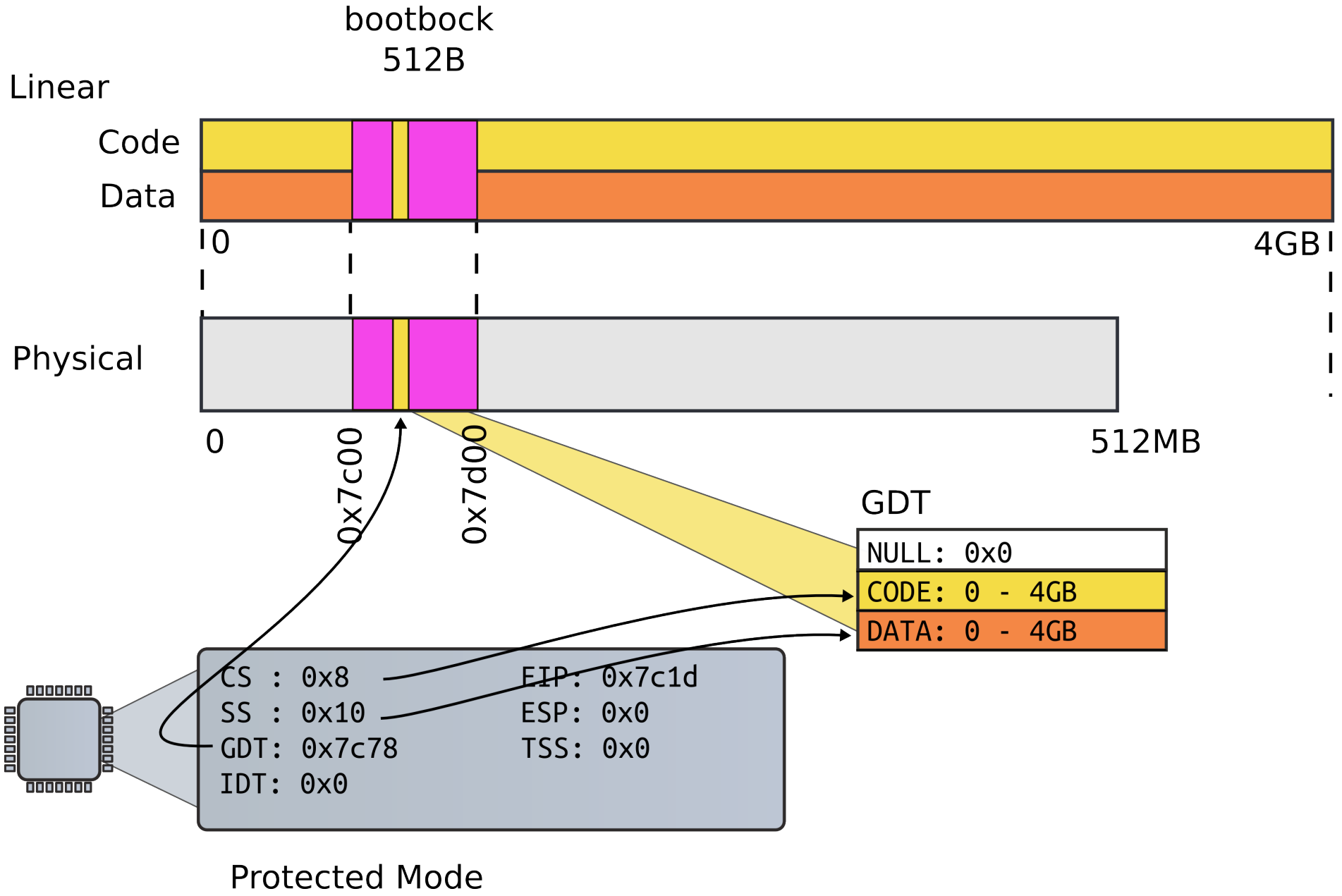
- Segment selector:



Segments

```
9155 .code32 # Tell assembler to generate 32-bit code now.
9156 start32:
9157 # Set up the protected-mode data segment registers
9158 movw $(SEG_KDATA<<3), %ax # Our data segment selector
9159 movw %ax, %ds # -> DS: Data Segment
9160 movw %ax, %es # -> ES: Extra Segment
9161 movw %ax, %ss # -> SS: Stack Segment
9162 movw $0, %ax # Zero segments not ready for use
9163 movw %ax, %fs # -> FS
9164 movw %ax, %gs # -> GS
```

Segments

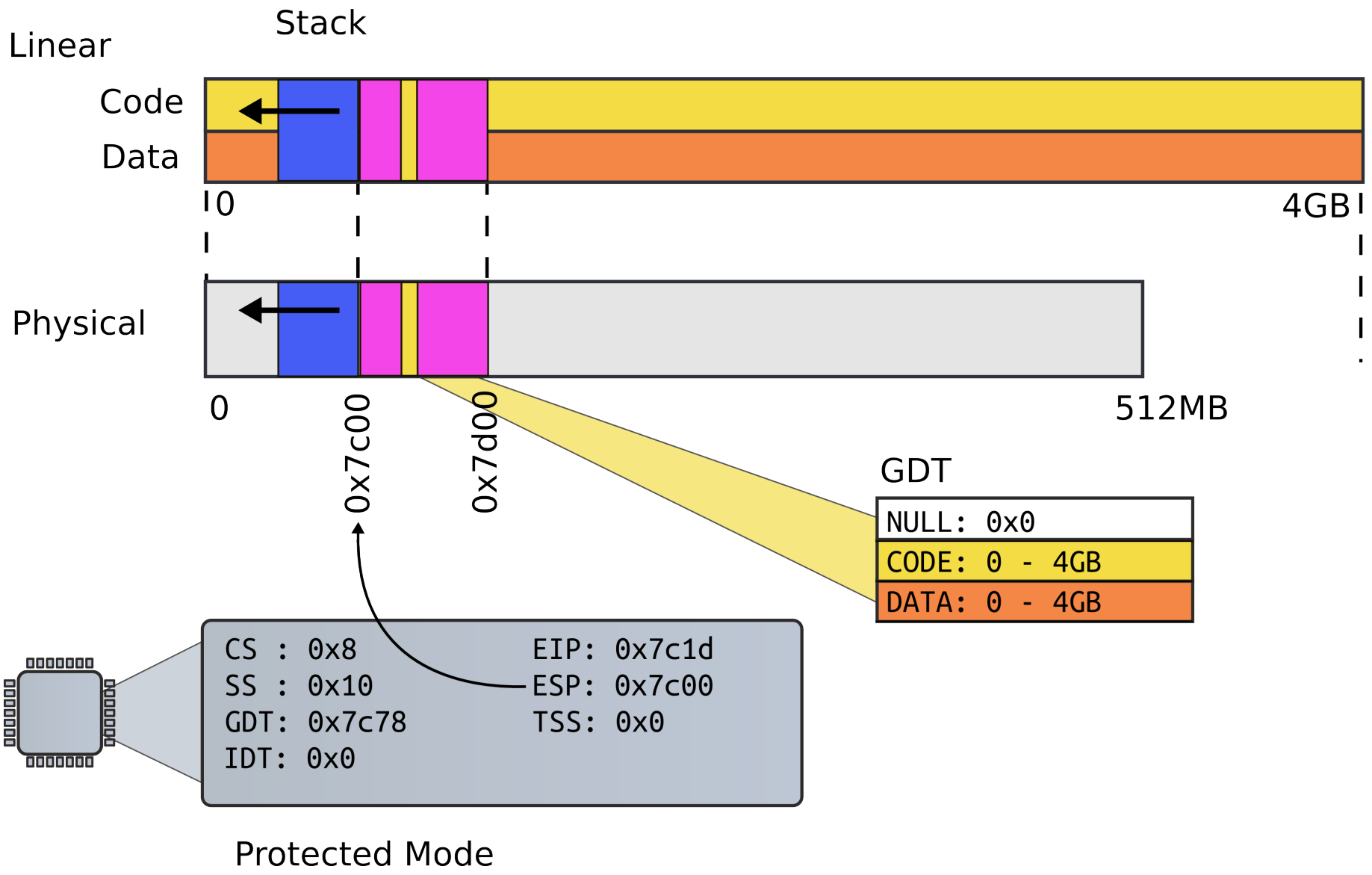

bootbock
512B
Linear
Code
Data
0
4GB
Physical
0
0x7c00
0x7d00
512MB
GDT
NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0
EIP: 0x7c1d
ESP: 0x0
TSS: 0x0
Protected Mode

Setup stack

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

```
9166 movl $start, %esp

9167 call bootmain
```

First stack


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

bootmain(): read kernel from disk

```
9216 void
9217 bootmain(void)
9218 {
9219     struct elfhdr *elf;
9220     struct proghdr *ph, *eph;
9221     void (*entry)(void);
9222     uchar* pa;
9223
9224     elf = (struct elfhdr*)0x10000; // scratch space
9225
9226     // Read 1st page off disk
9227     readseg((uchar*)elf, 4096, 0);
9228
9229     // Is this an ELF executable?
9230     if(elf->magic != ELF_MAGIC)
9231         return; // let bootasm.S handle error
9232
```

```
9232
9233    // Load each program segment (ignores ph flags).
9234    ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9235    eph = ph + elf->phnum;
9236    for(; ph < eph; ph++){
9237        pa = (uchar*)ph->paddr;
9238        readseg(pa, ph->filesz, ph->off);
9239        if(ph->memsz > ph->filesz)
9240            stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9241    }
9242
9243    // Call the entry point from the ELF header.
9244    // Does not return!
9245    entry = (void(*)(void))(elf->entry);
9246    entry();
9247 }
```

bootmain(): read kernel from disk

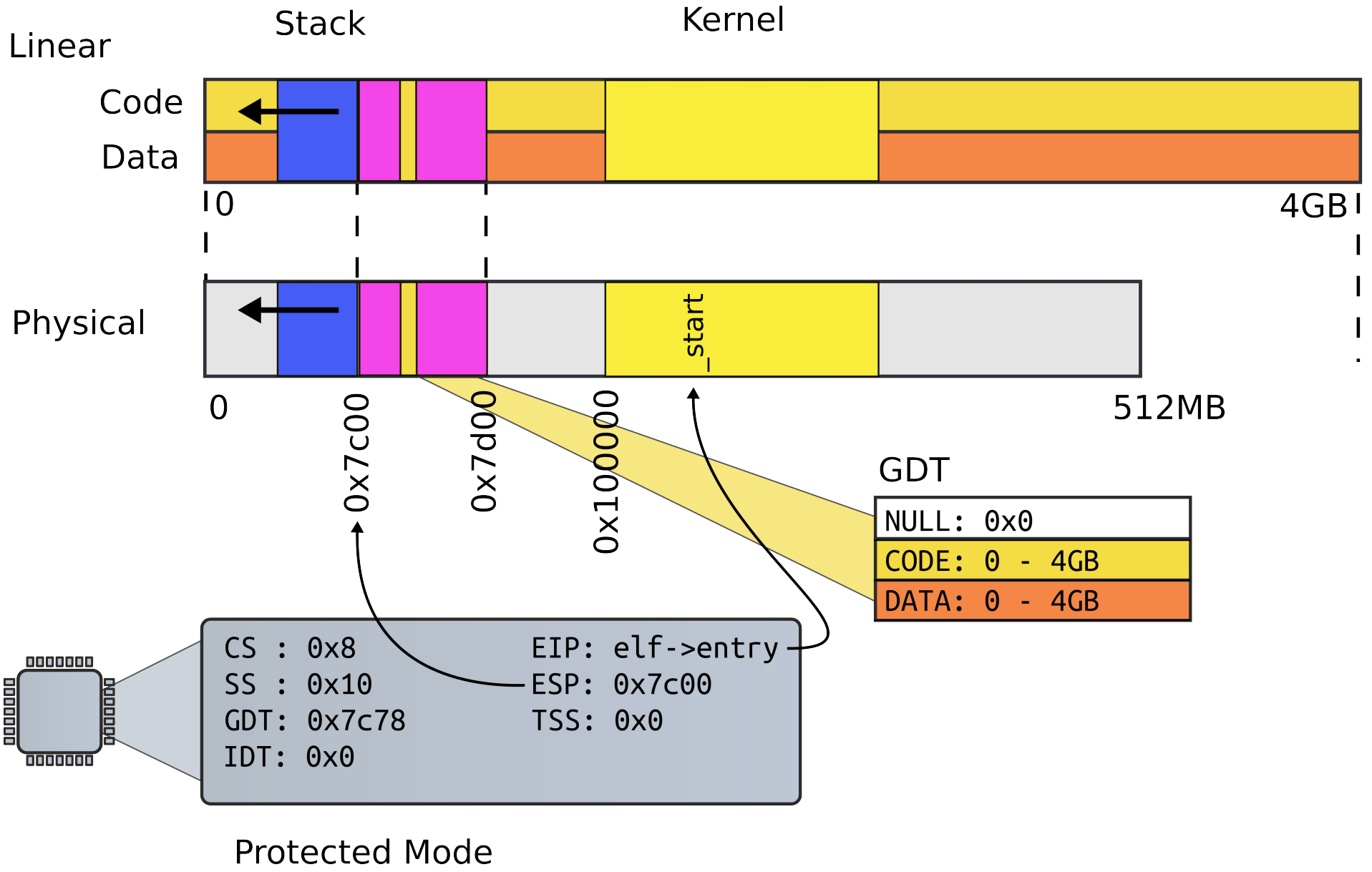
How one reads disk?

```
9257
9258 // Read a single sector at offset into dst.
9259 void
9260 readsect(void *dst, uint offset)
9261 {
9262     // Issue command.
9263     waitdisk();
9264     outb(0x1F2, 1); // count = 1
9265     outb(0x1F3, offset);
9266     outb(0x1F4, offset >> 8);
9267     outb(0x1F5, offset >> 16);
9268     outb(0x1F6, (offset >> 24) | 0xE0);
9269     outb(0x1F7, 0x20); // cmd 0x20 - read sectors
9270
9271     // Read data.
9272     waitdisk();
9273     insl(0x1F0, dst, SECTSIZE/4);
9274 }
```

How one reads disk (cont)?

```
9250 void
9251 waitdisk(void)
9252 {
9253     // Wait for disk ready.
9254     while((inb(0x1F7) & 0xC0) != 0x40)
9255     ;
9256 }
9257
```

Kernel

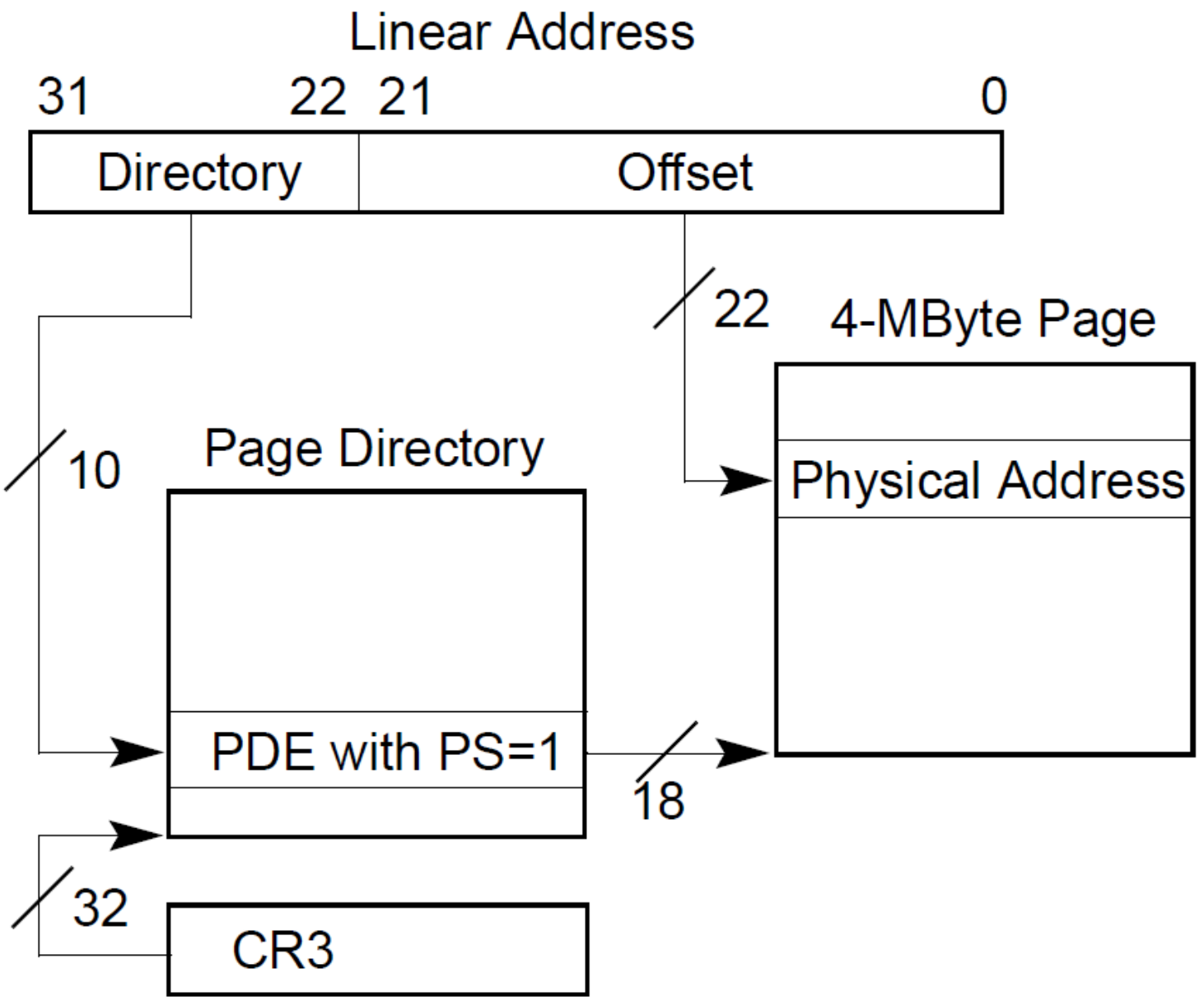

Linear
Stack
Kernel
Code
Data
0
4GB
Physical
_start
0
0x7c00
0x7d00
0x100000
512MB
GDT
NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0
EIP: elf->entry
ESP: 0x7c00
TSS: 0x0
Protected Mode

```
1039 .globl entry
1136 # By convention, the _start symbol specifies the ELF entry point.
1137 # Since we haven't set up virtual memory yet, our entry point is
1138 # the physical address of 'entry'.
1139 .globl _start
1140 _start = V2P_WO(entry)
1141
1142 # Entering xv6 on boot processor, with paging off.
1143 .globl entry
1144 entry:
1145 # Turn on page size extension for 4Mbyte pages
1146    movl %cr4, %eax
1147    orl $(CR4_PSE), %eax
1148    movl %eax, %cr4
```

entry(): kernel ELF entry

Set up page directory

```
1149 # Set page directory
1150 movl $(V2P_WO(entrypgdir)), %eax
1151 movl %eax, %cr3
```


Linear Address
31
22
21
0
Directory
Offset
22
4-MByte Page
Physical Address
10
Page Directory
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

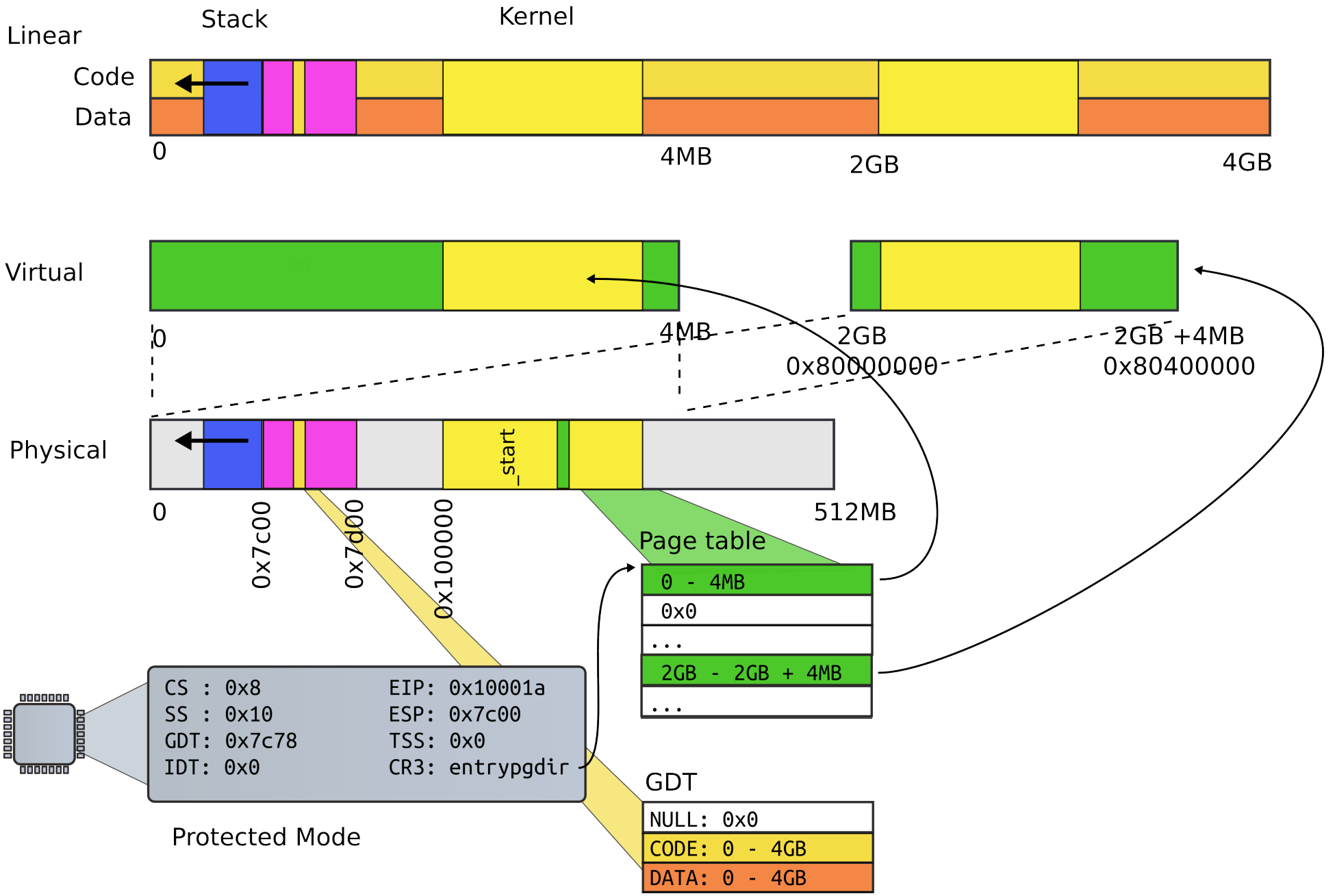
```
1406 // The boot page table used in entry.S and entryother.S.
1407 // Page directories (and page tables) must start on page
        boundaries,
1408 // hence the __aligned__ attribute.
1409 // PTE_PS in a page directory entry enables 4Mbyte pages.
1410
1411 __attribute__((__aligned__(PGSIZE)))
1412 pde_t entrypgdir[NPDENTRIES] = {
1413 // Map VA's [0, 4MB) to PA's [0, 4MB)
1414 [0] = (0) | PTE_P | PTE_W | PTE_PS,
1415 // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1416 [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1417 };
```

First page table

First page table (cont)

```
0870 // Page directory and page table constants.

0871 #define NPDENTRIES 1024
```

First page table



Turn on paging

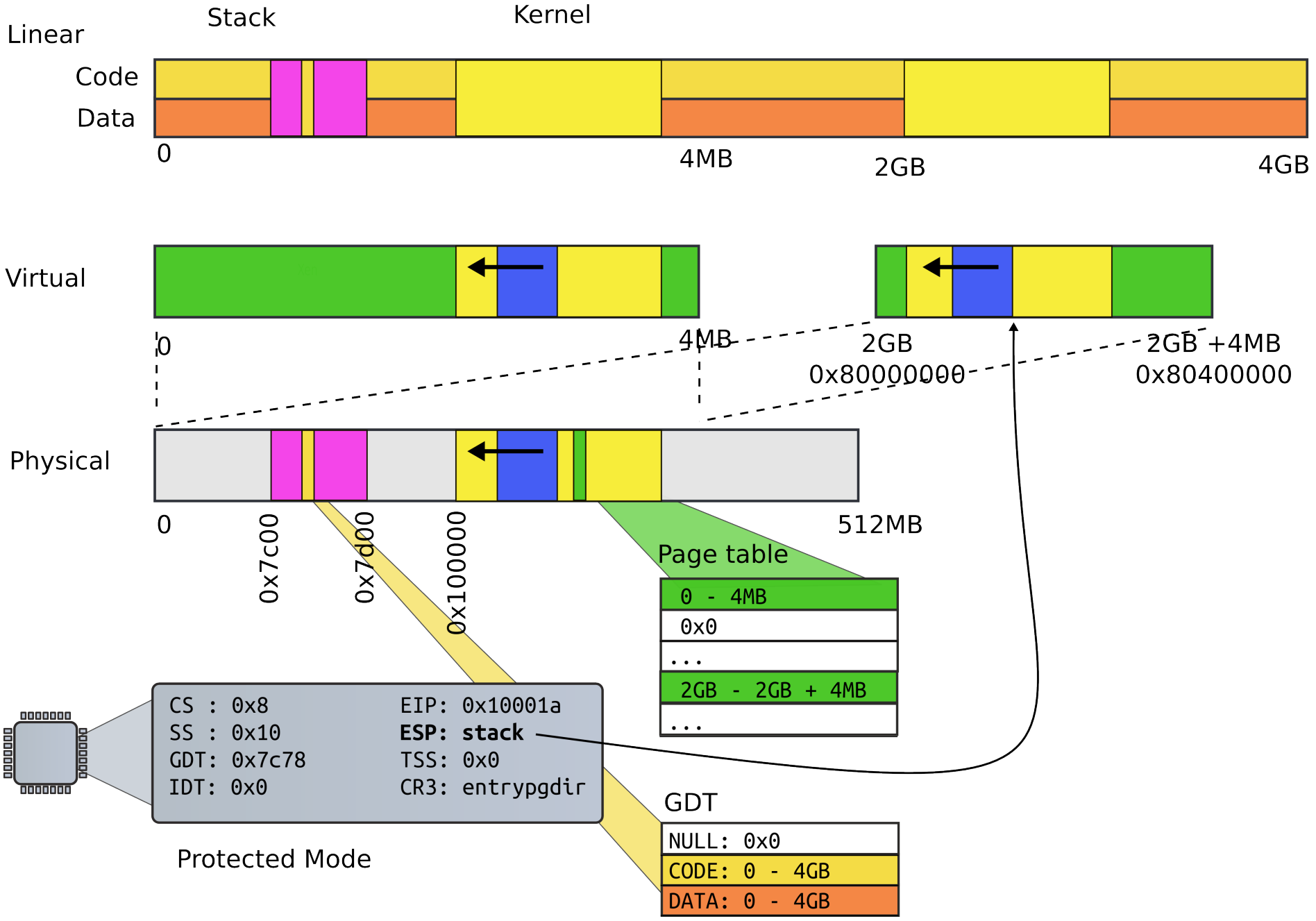
```
1152 # Turn on paging.
1153 movl %cr0, %eax
1154 orl $(CR0_PG|CR0_WP), %eax
1155 movl %eax, %cr0
```

High address stack (4K)

```
1157 # Set up the stack pointer.
1158 movl $(stack + KSTACKSIZE), %esp
1159
...
1167 .comm stack, KSTACKSIZE
```

```
0151 #define KSTACKSIZE 4096 // size of
                per-process kernel stack
```

High address stack (4K)



Jump to main()

```
1160 # Jump to main(), and switch to executing at
1161 # high addresses. The indirect call is
needed
       because
1162 # the assembler produces a PC-relative
       instruction
1163 # for a direct jump.
1164 mov $main, %eax
1165 jmp *%eax
1166
```

Running in main()

```
1313 // Bootstrap processor starts running C code here.
1314 // Allocate a real stack and switch to it, first
1315 // doing some setup required for memory allocator to work.
1316 int
1317 main(void)
1318 {
1319     kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320     kvmalloc(); // kernel page table
1321     mpinit(); // detect other processors
1322     lapicinit(); // interrupt controller
1323     seginit(); // segment descriptors
1324     cprintf("\ncpu%d: starting xv6\n\n", cpunum());
...
1340 }
```

Questions?