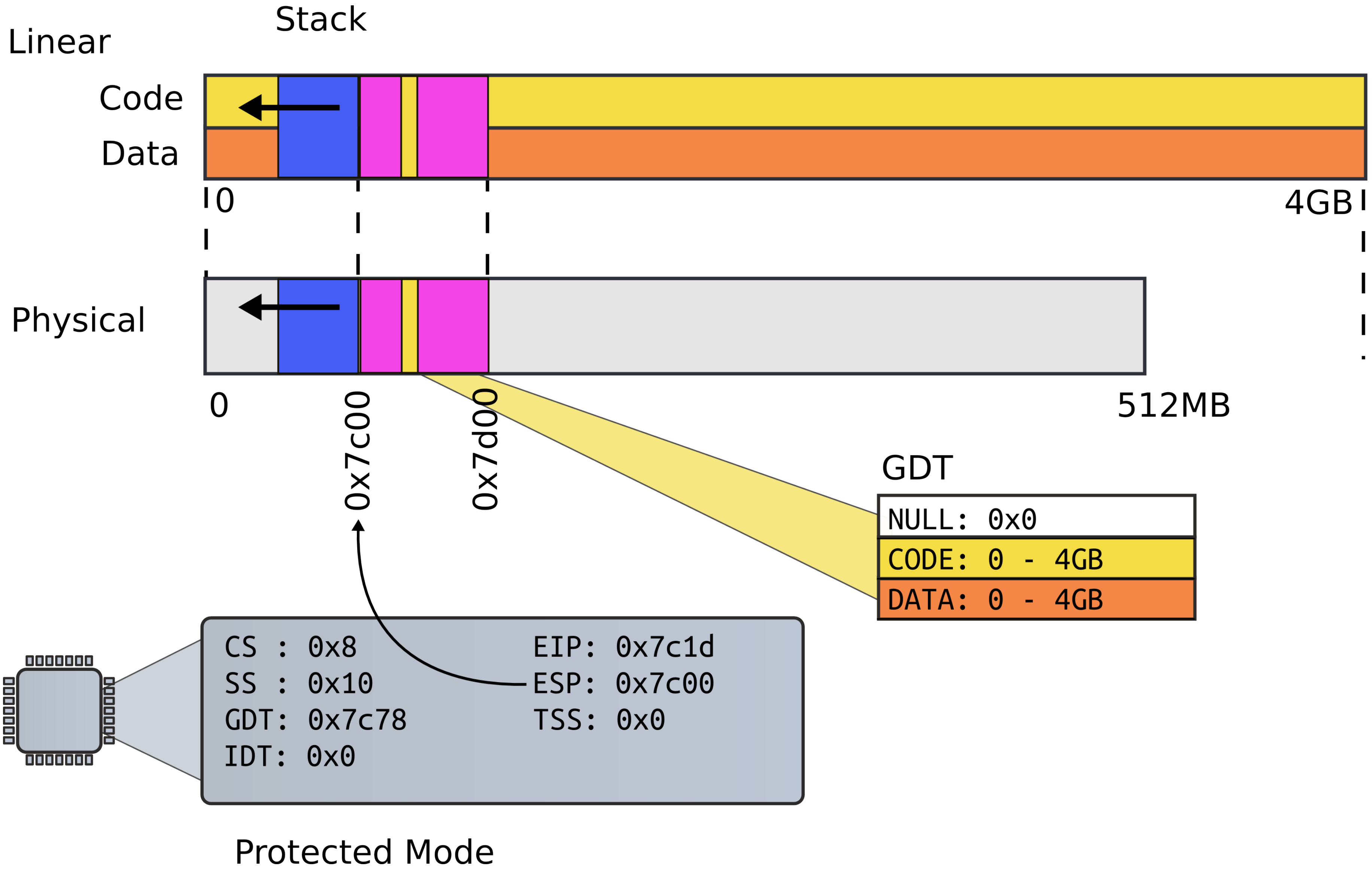
CS5460/6460: Operating Systems

Lecture 9: Finishing system boot, and system init

Anton Burtsev

January, 2014

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
```

xv6/bootasm.S

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

```

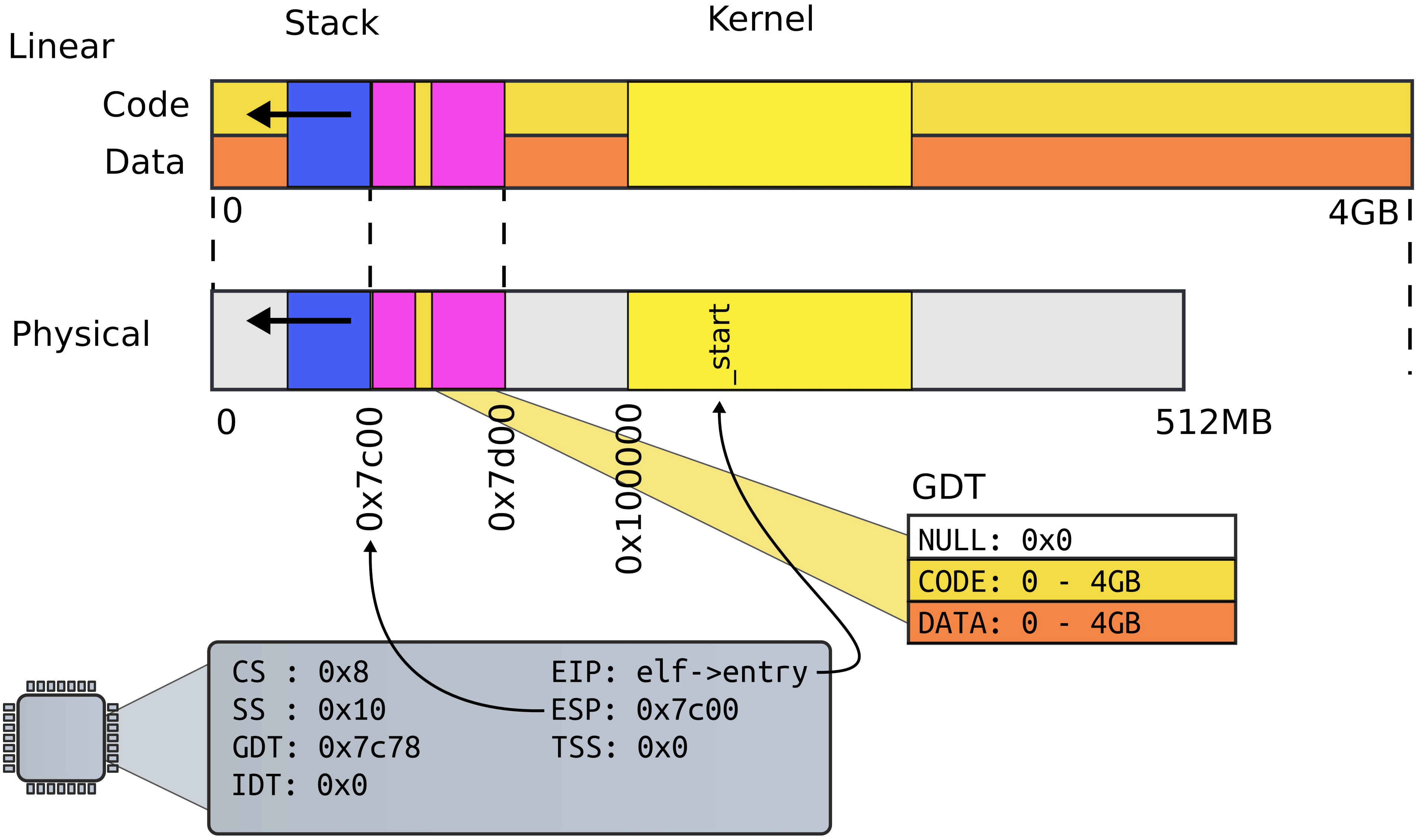
xv6/bootmain.c

bootmain(): read kernel from disk

```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();
}
```

xv6/bootmain.c

Kernel



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
```

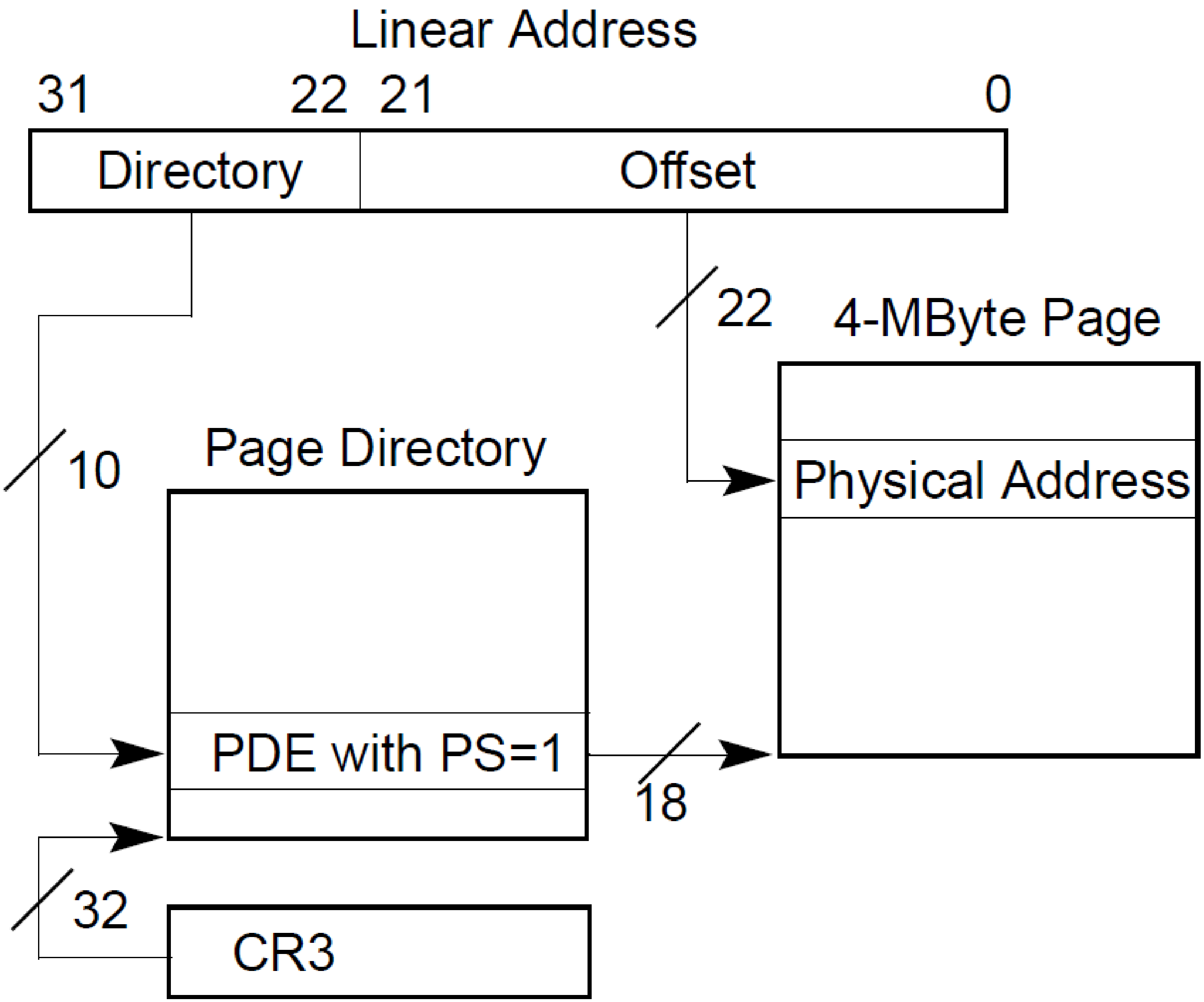
entry(): kernel ELF entry

xv6/entry.S

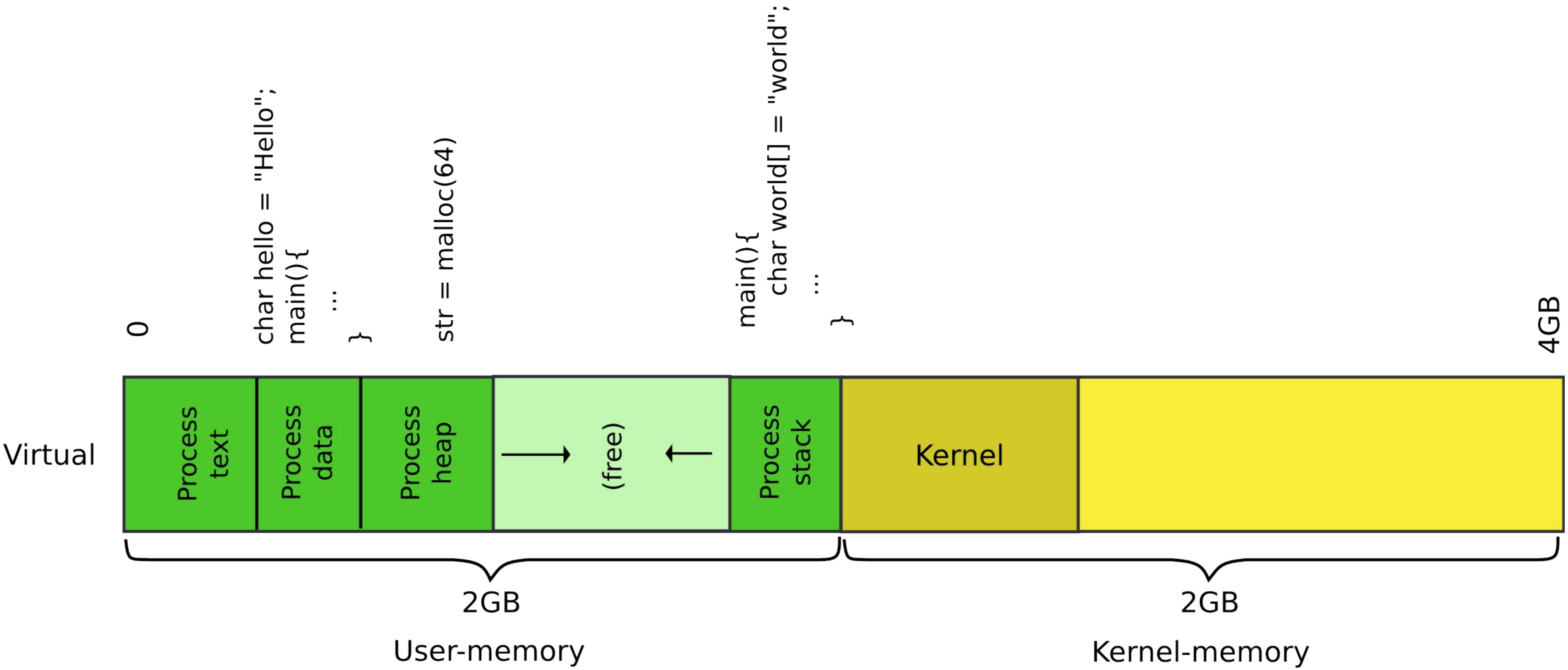
Set up page directory

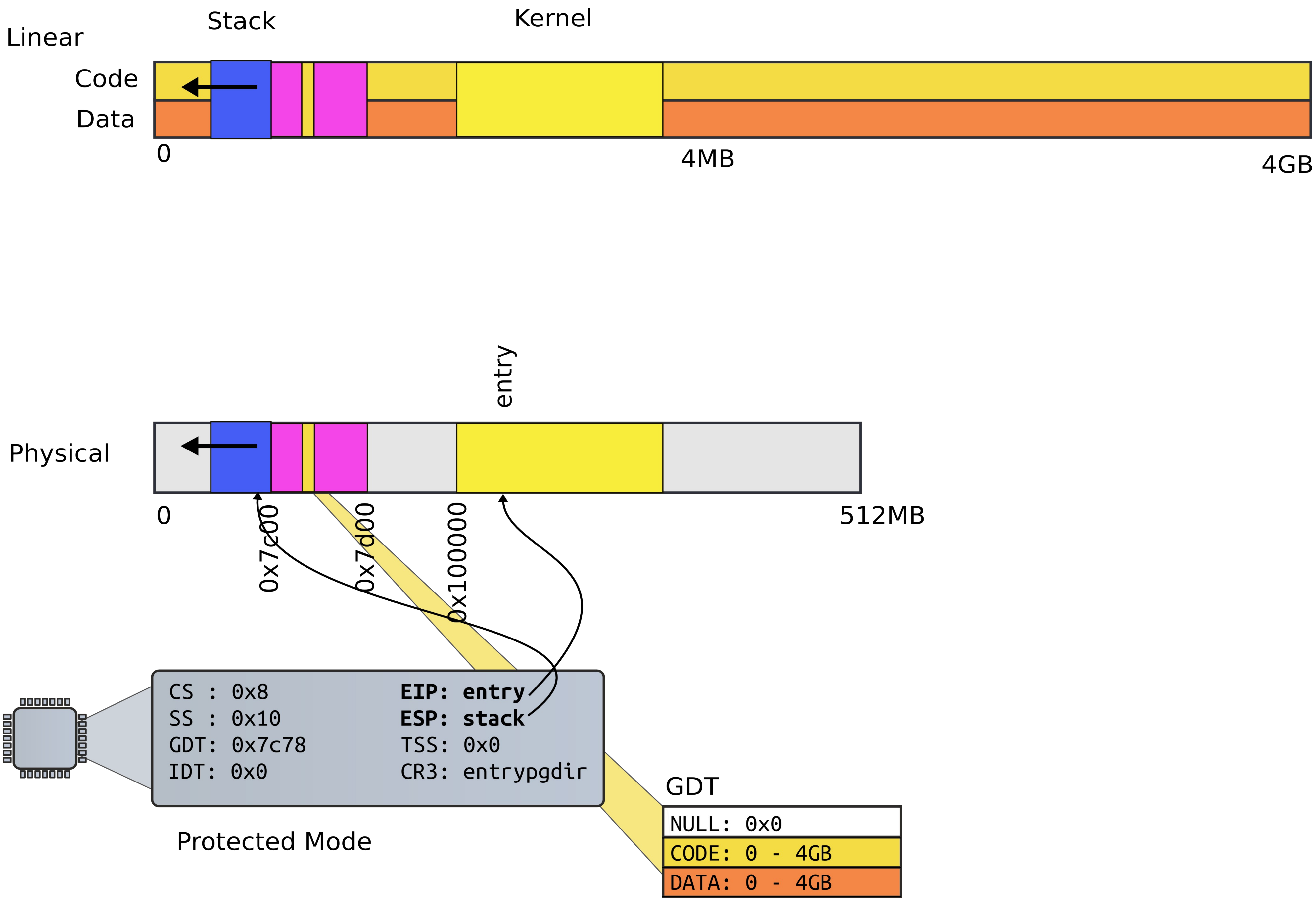
```
# Set page directory
movl $(V2P_WO(entrypgdir)), %eax
movl %eax, %cr3
```

xv6/entry.S


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

Our goal: 2GB/2GB address space





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

GDT

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

First page table

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

```
// 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

```
// 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

```
// 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

```
// 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

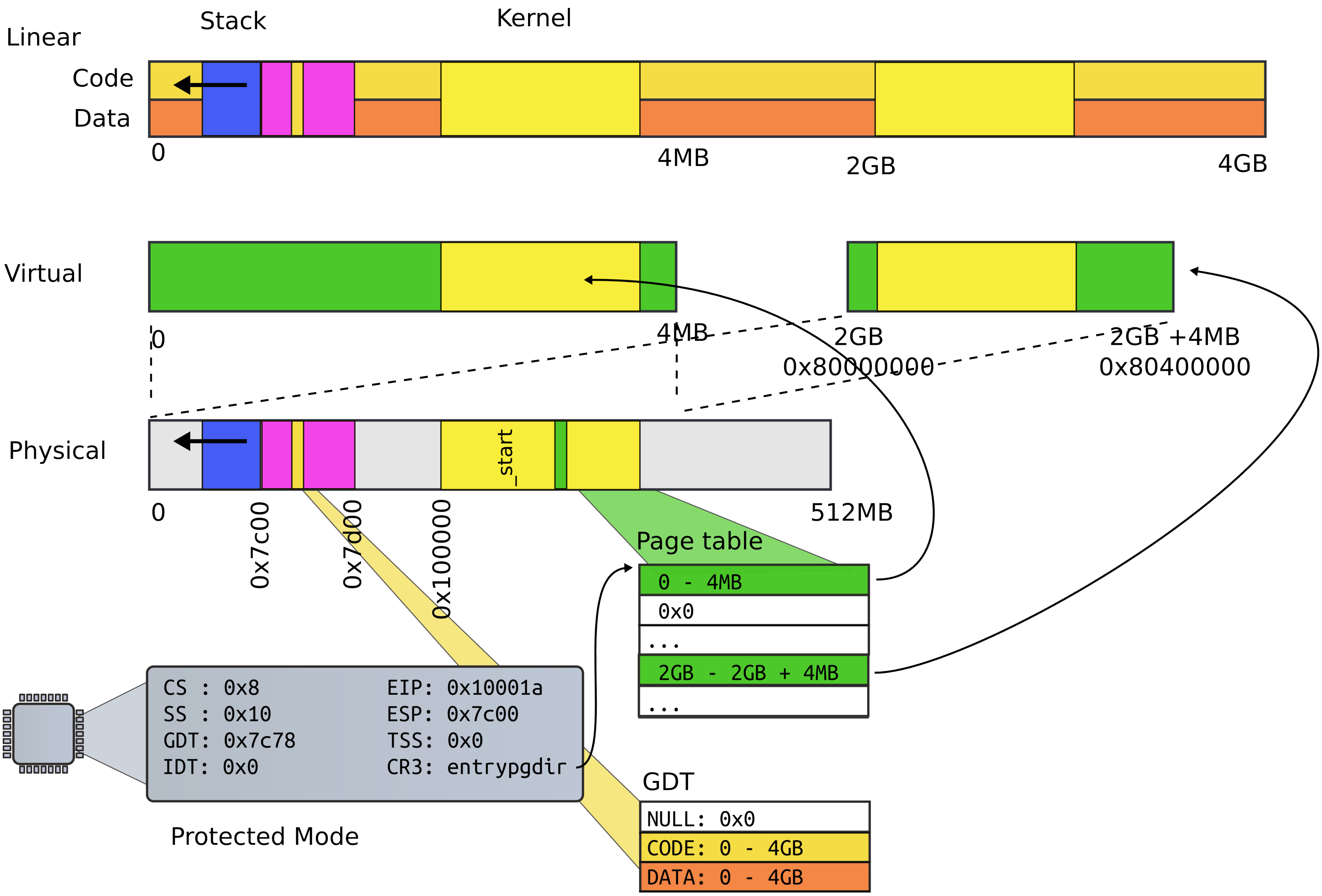
```
// The boot page table used in entry.S and entryother.S.
// Page directories (and page tables) must start on page boundaries,
// hence the __aligned__ attribute.
// PTE_PS in a page directory entry enables 4Mbyte pages.

__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
```

Protected Mode

Page table

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

GDT

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

Turn on paging

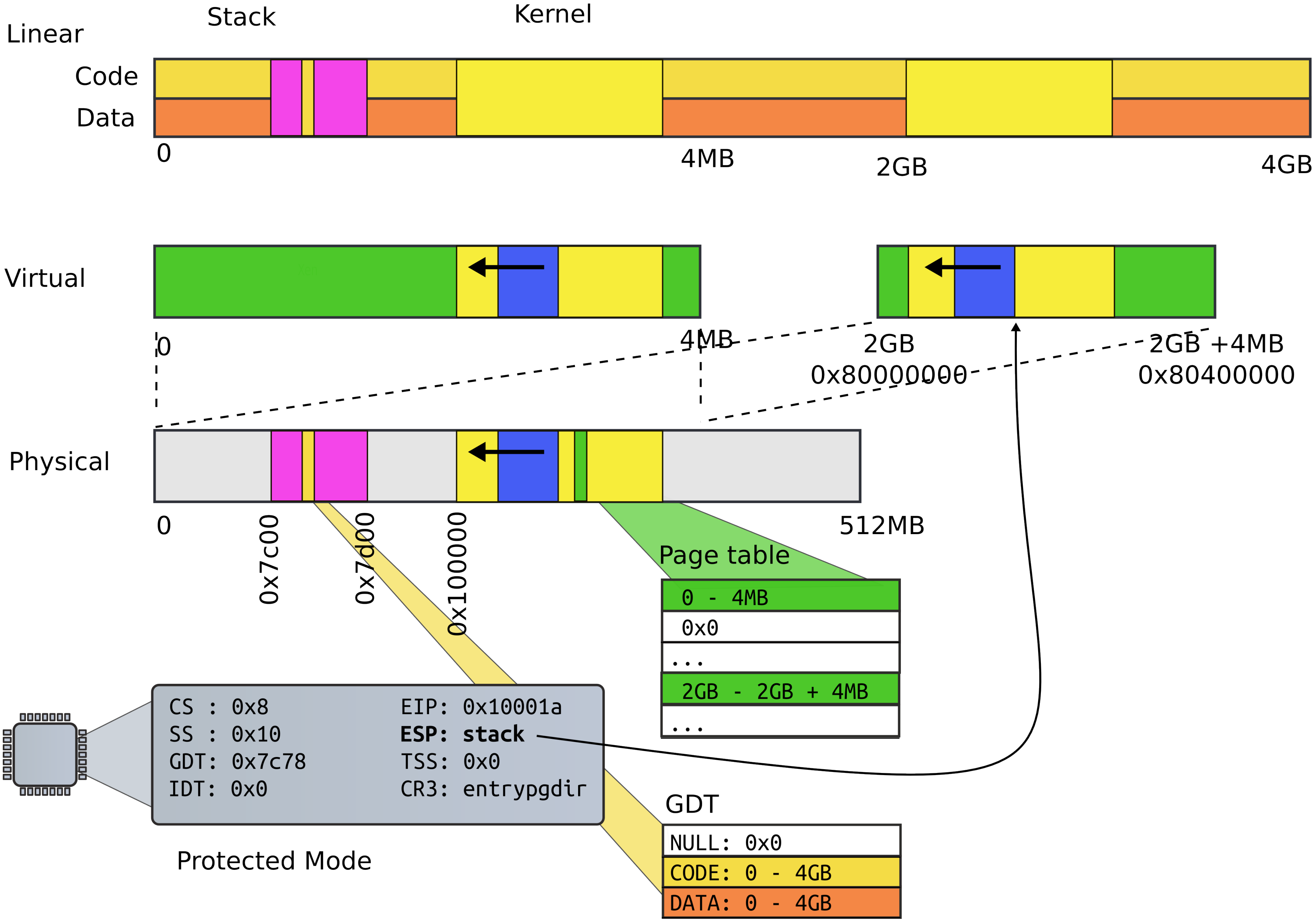
```
# Turn on paging.
movl %cr0, %eax
orl $(CR0_PG|CR0_WP), %eax
movl %eax, %cr0
```

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)



Linear

Stack

Kernel

Code

Data

0

4MB

2GB

4GB

Virtual

0

4MB

2GB
0x80000000

2GB +4MB
0x80400000

Physical

0

0x7c00

0x7d00

0x100000

512MB

Page table

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

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

GDT

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

Protected Mode

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()

```
// 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());
...
}
```

Recap of the boot sequence

- Setup segments (data and code)
- Switched to protected mode
  - Loaded GDT (segmentation is on)
- Setup stack (to call C functions)
- Loaded kernel from disk
- Setup first page table
  - 2 entries [0 : 4MB] and [2GB : (2GB + 4MB)]
- Setup high-address stack
- Jumped to main()

Conclusion

- We've booted
  - We're running in main()

- Next time:
  - 2-level page tables
  - Process and kernel address space

Thank you!

Thank you!