

# cs5460/6460: Operating Systems

## Lecture 07: System boot

Anton Burtsev

February, 2026

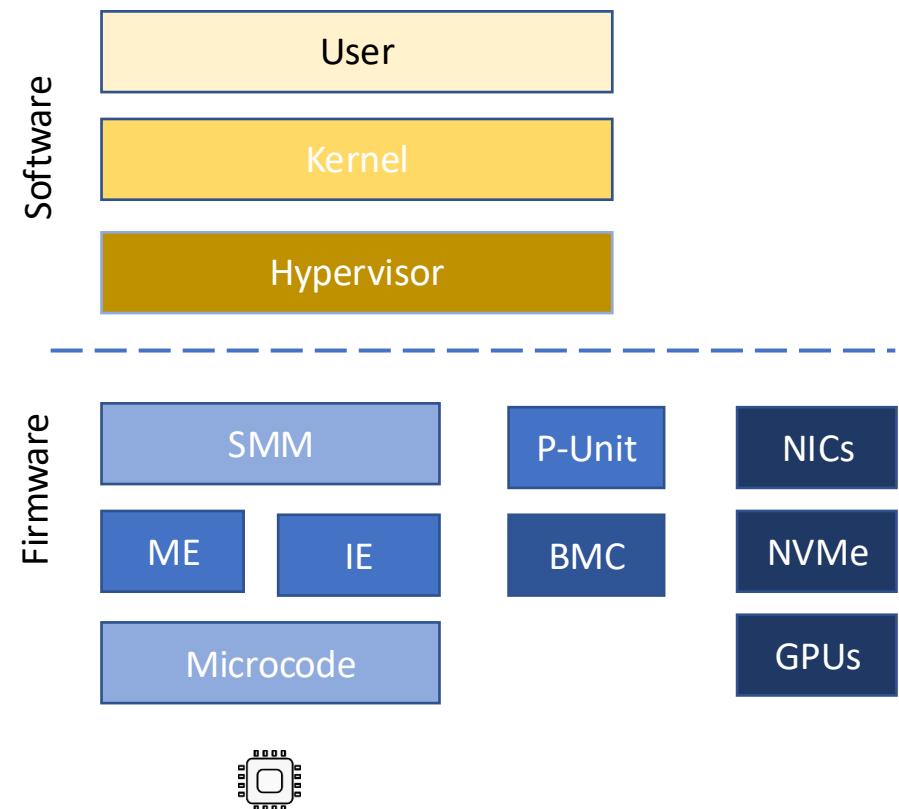
# What happens when we turn on the power?

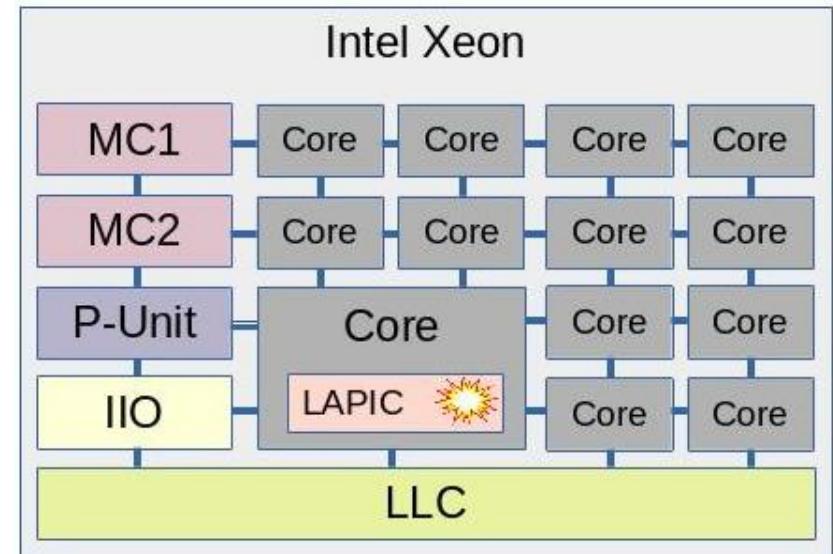
- Well it's complicated
  - Intel SGX Explained is a good start (Section 2.13 [1])
- At a high-level a sequence of software pieces initializes the platform
  - Management engine (ME), microcode, firmware (BIOS), bootloader

- 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/hypervisor
- **System Management Mode (SMM) (ring -2)**
  - Runs below the hypervisor/OS
- **Intel Management Engine** and Intel Innovation Engine (ring -3)
- And, honestly, **microcode** if you like

# Multiple layers of firmware

- Microcode
- BIOS
- Management and Innovation Engines (ME and IE)
- System Management Mode (SMM)
- Board Management Controller (BMC)
- Power microcontrollers (P-Unit and SCU)
- A ton more...
  - NICs, HDDs, NVMe, GPUs





```

<ProductInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../../../../schemas/ProductInfo.xsd">
  <ProductIdentification Idcode="0x00a84013" Mask="0xF0000000">
    <Product Idcode="0x00a84013" DeviceType="ANN_CLTAPC" DeviceStepping="A0"/>
    <Product Idcode="0x10a84013" DeviceType="ANN_CLTAPC" DeviceStepping="B0"/>
  </ProductIdentification>
  <ProductDescription DeviceType="ANN_CLTAPC" Stepping="*>
    <TapInfo TapName="ANN_CLTAPC" NodeType="Uncore" DeviceType="ANN_CLTAPC" Stepping="$(Stepping)" AddInstanceNameSuffix="true"/>
    <TapInfo TapName="ANN_SC0" NodeType="Chipset" DeviceType="ANN_SC" Stepping="$(Stepping)" AddInstanceNameSuffix="true"/>
    <TapInfo TapName="ANN_PSH" NodeType="Thread" DeviceType="LMT" Stepping="$(Stepping)" AddInstanceNameSuffix="true"/>
    <DeviceProperty Name="CoreGroup" Val="PSH"/>
  </TapInfo>
    <TapInfo TapName="ANN_ARC" NodeType="Thread">
      <DeviceProperty Name="CoreGroup" Val="ARC"/>
    </TapInfo>
    <TapInfo TapName="ANN_MC8051" NodeType="Thread">
      <DeviceProperty Name="CoreGroup" Val="8051"/>
    </TapInfo>
    <TapInfo TapName="ANN_SJTAG|ANN_SEP" NodeType="CoreGroup">
      <DeviceProperty Name="CoreGroup" Val="APS"/>
    </TapInfo>
    <TapInfo TapName="ANN_AUDIO" NodeType="Thread">
      <DeviceProperty Name="CoreGroup" Val="APS"/>
    </TapInfo>
    <TapInfo TapName="SLM_MODULE0" NodeType="Box">
      <TapSelect Name="SLM_C0" DeviceType="SLM">
        <TapSelect Name="SLM_C0_TO" DeviceType="SLM"/>
      </TapSelect>
      <TapSelect Name="SLM_C1" DeviceType="SLM">
        <TapSelect Name="SLM_C1_TO" DeviceType="SLM"/>
      </TapSelect>
    </TapInfo>
    <TapInfo TapName="SLM_MODULE1" NodeType="Box">
      <TapSelect Name="SLM_C2" DeviceType="SLM">
        <TapSelect Name="SLM_C2_TO" DeviceType="SLM"/>
      </TapSelect>
      <TapSelect Name="SLM_C3" DeviceType="SLM">
        <TapSelect Name="SLM_C3_TO" DeviceType="SLM"/>
      </TapSelect>
    </TapInfo>
    <TapInfo TapName=".*" NodeType="Chipset" Stepping="$(Stepping)" InstanceId="0" AddInstanceNameSuffix="true"/>
  </ProductDescription>
</ProductInfo>

```



Mark Ermolov  
@\_markel\_

Follow

fix="true">

ieSuffix="true">

nceNameSuffix="true">

ieSuffix="true">

nstanceNameSuffix="false">  
<Id="0" DeviceSubType="">

<Id="1" DeviceSubType="">

nstanceNameSuffix="false">  
<Id="0" DeviceSubType="">

<Id="1" DeviceSubType="">

It's amazing how many heterogeneous CPU cores were integrated in Intel Silvermont's Moorefield SoC (ANN): x86, ARC, LMT, 8051, Audio DSP, each running own firmware and supporting JTAG interface



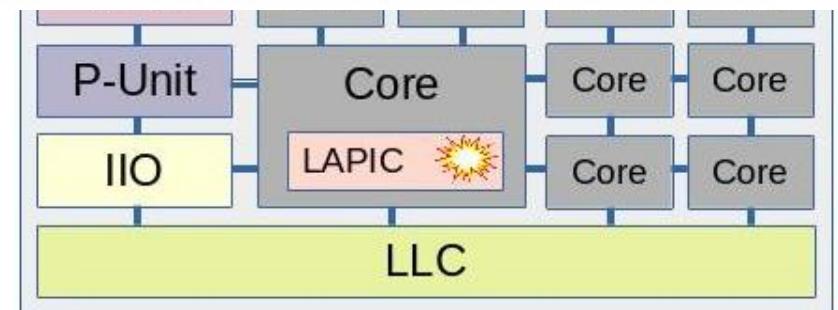
What is running on  
these  
microcontrollers?

### Intel Corporation, Austin/Hillsboro/Bangalore

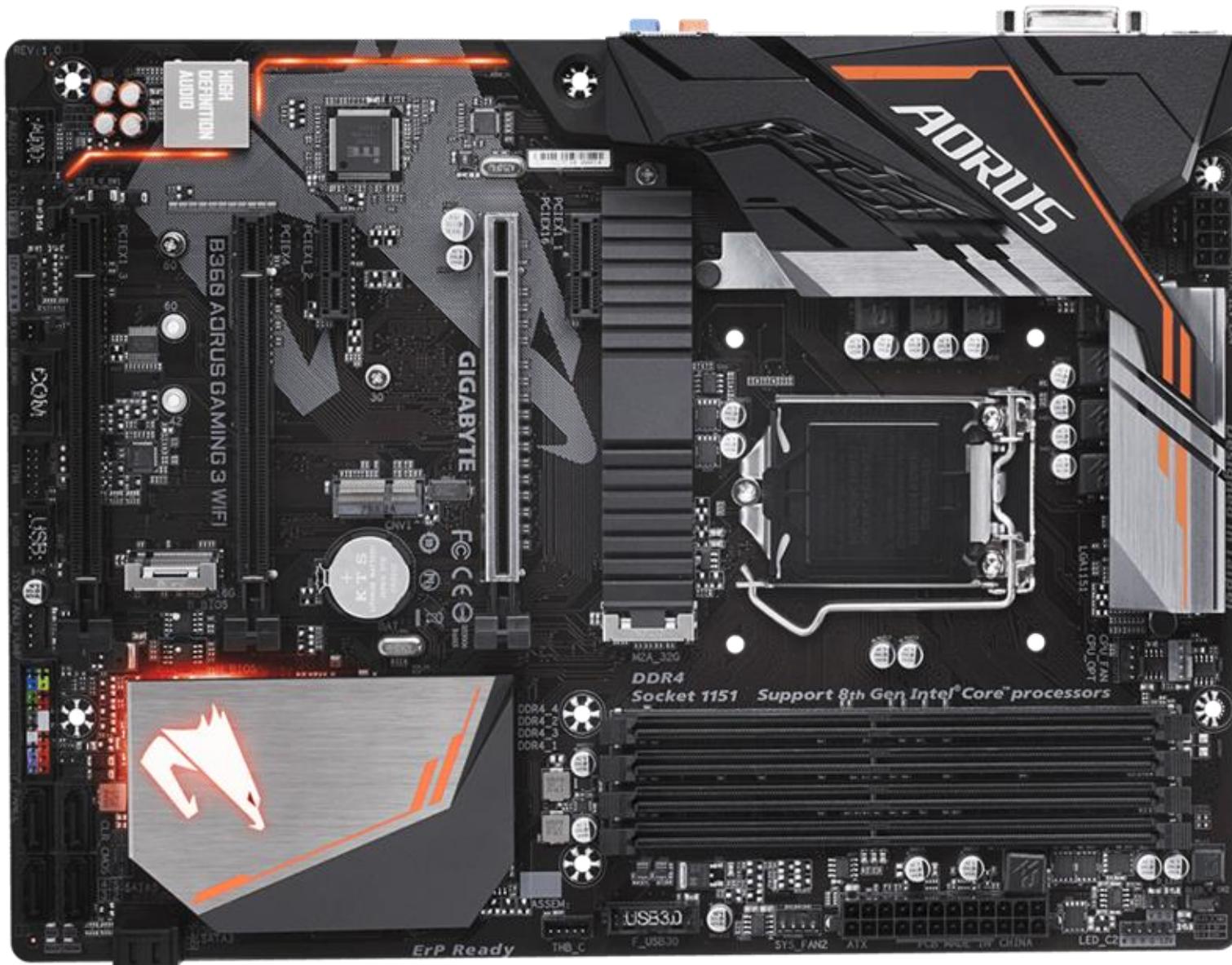
#### CPU POWER MANAGEMENT FIRMWARE ARCHITECT

Compute Die (CPU) power management firmware architect for devices & converged mobility Intel SoCs. Responsible for delivering pcode - power management firmware assembly code running on Foxton microcontroller, with the active/idle/thermal algorithms, cold/warm reset sequences, power delivery & sequencing of each IP. Also responsible to provide solutions/algorithms to enhance overall SoC Power & Performance.

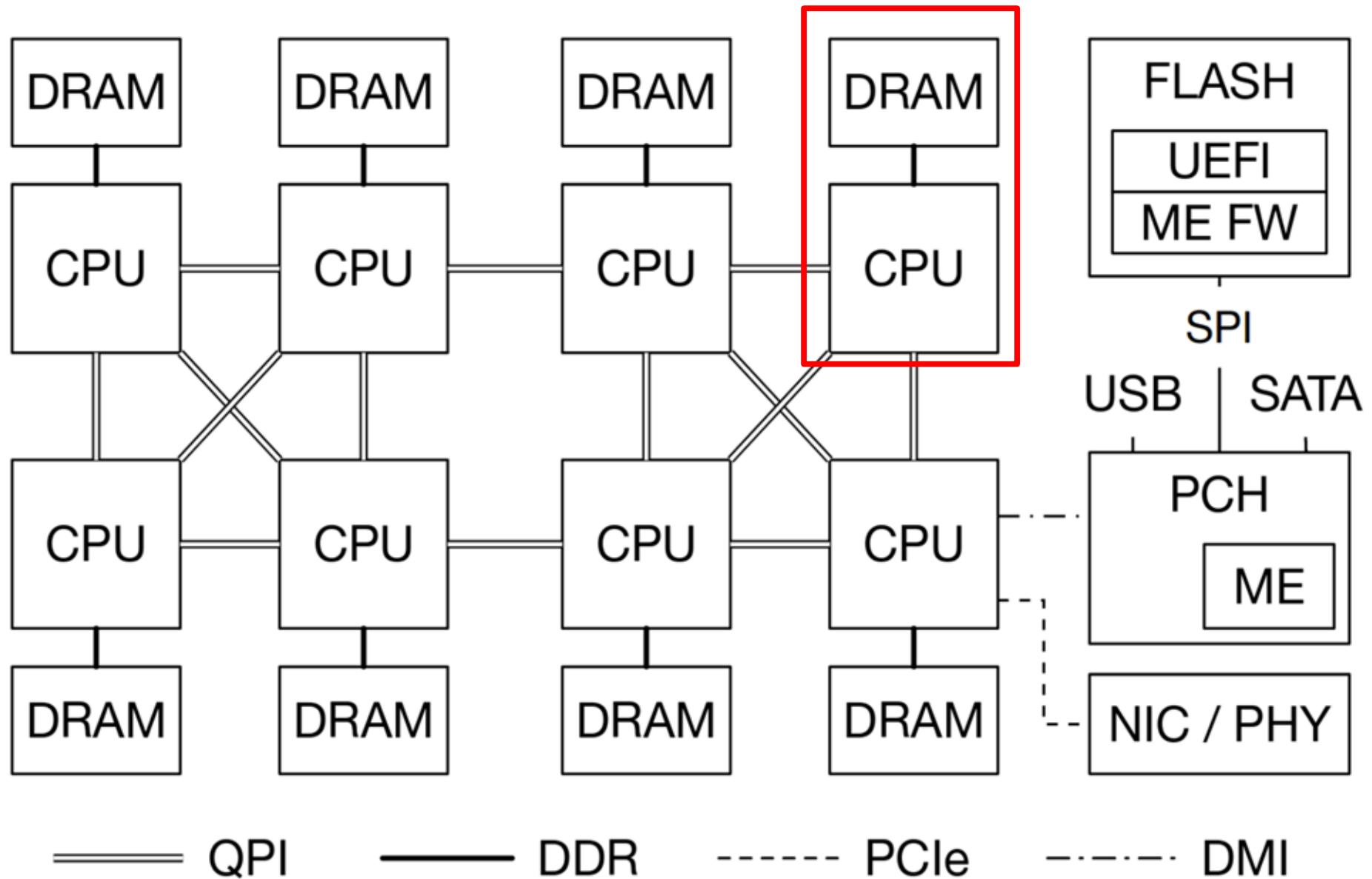
[March 2011 – Present]



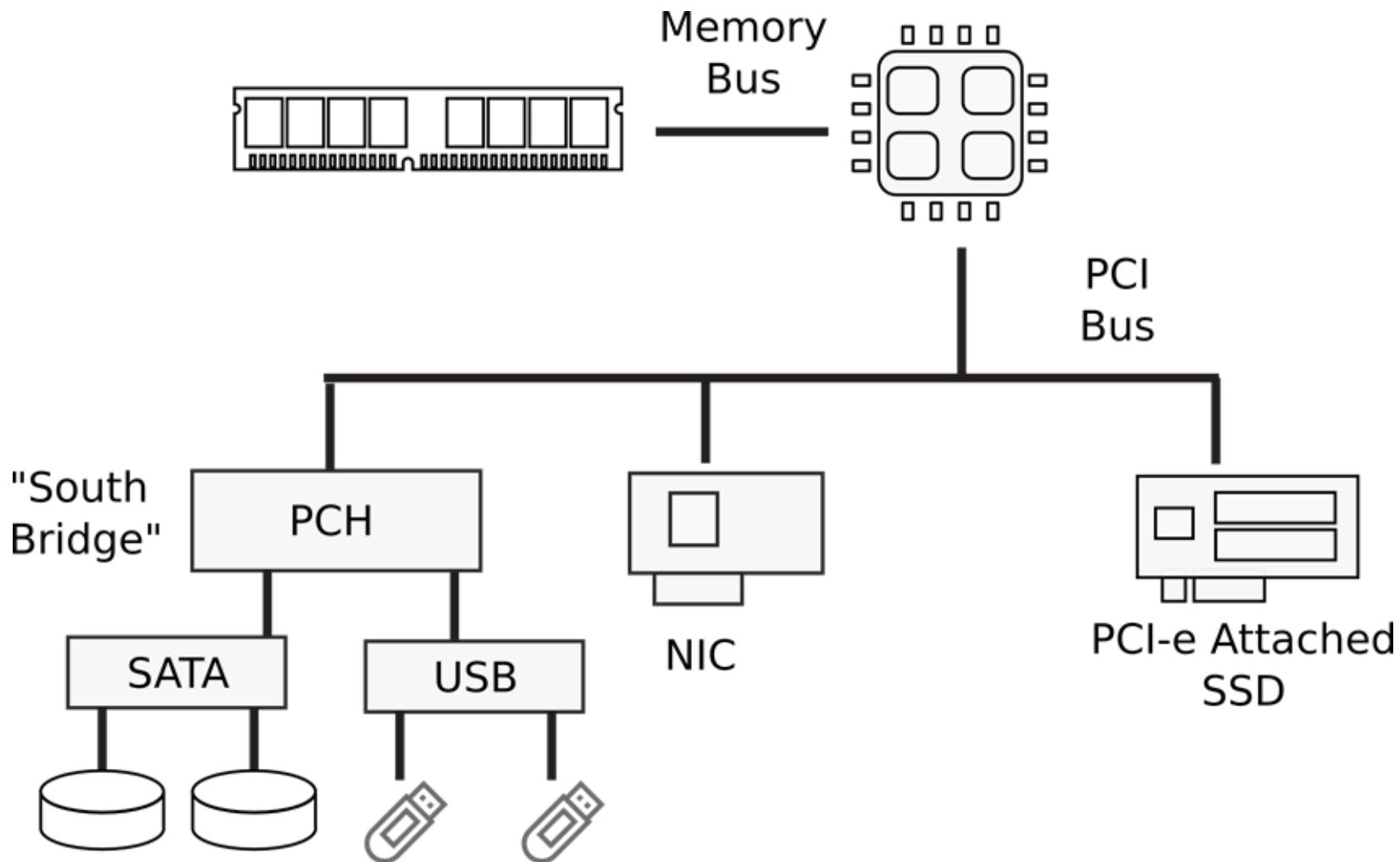
# B360 AORUS Motherboard



# PC motherboard components



# I/O Devices



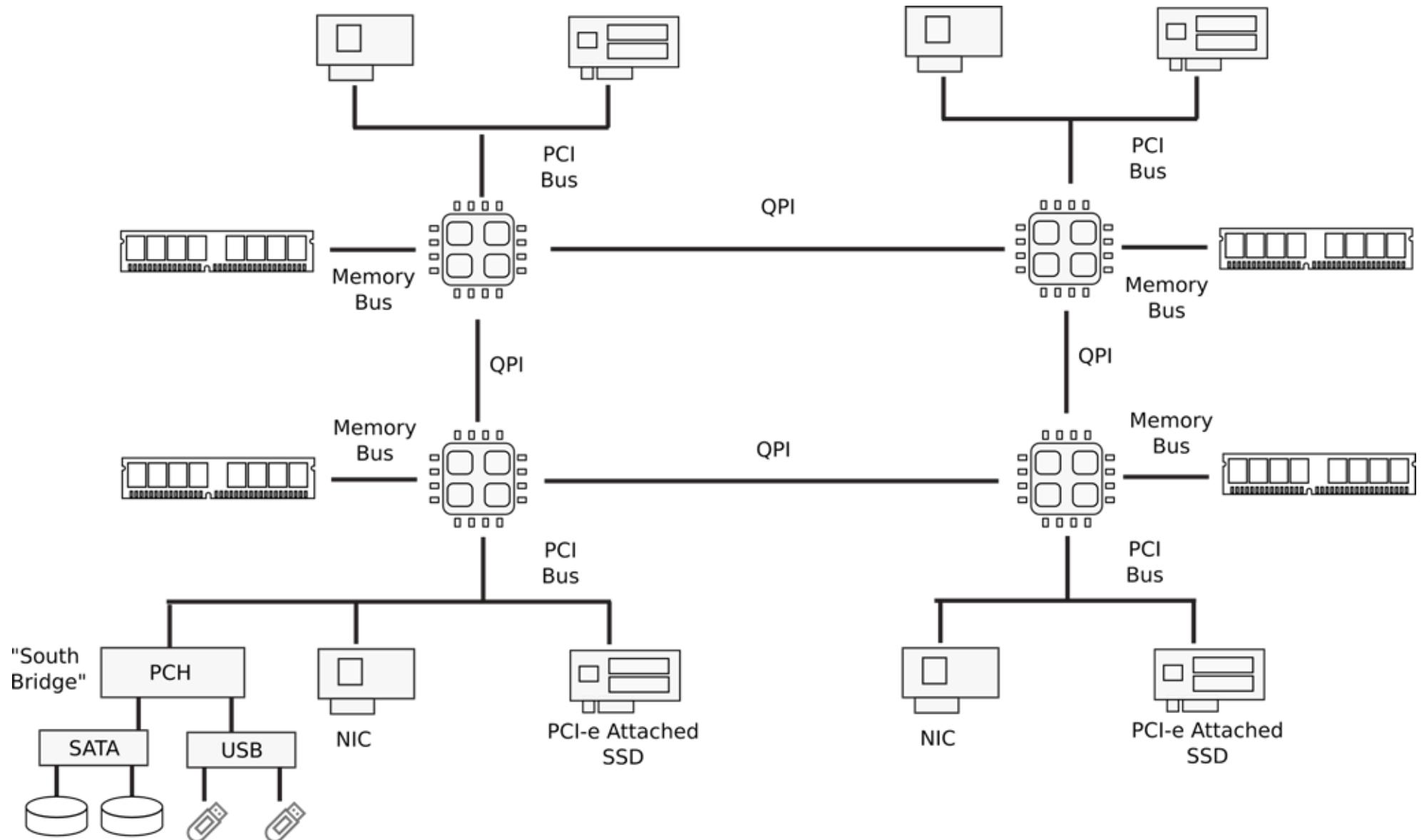
# Dell R830 4-socket server



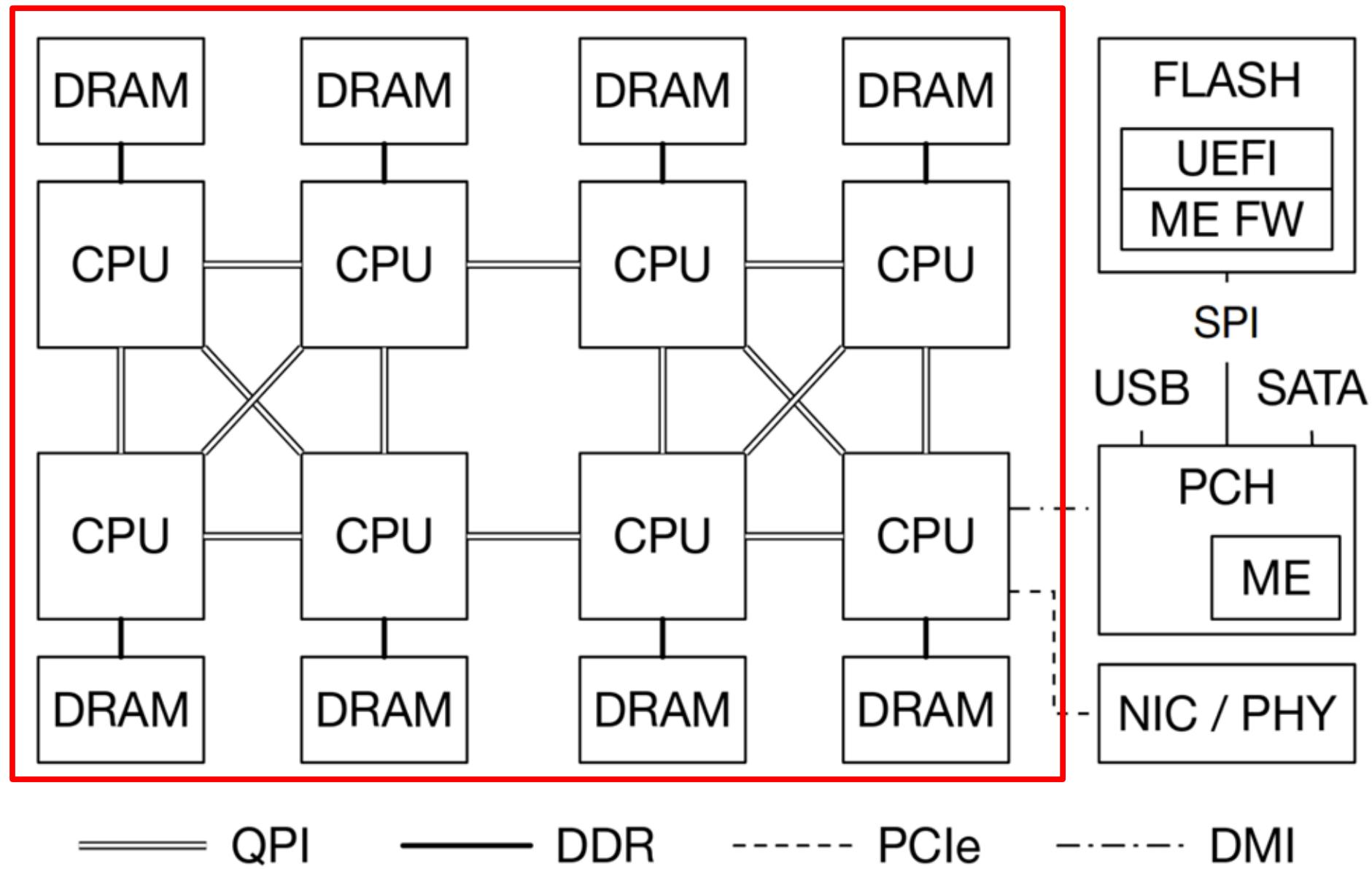
Dell Poweredge R830 System Server with 2 sockets on the main floor and 2 sockets on the expansion

[http://www.dell.com/support/manuals/us/en/19/poweredge-r830/r830\\_om/supported-configurations-for-the-poweredge-r830-system?guid=guid-01303b2b-f884-4435-b4e2-57bec2ce225a&lang=en-us](http://www.dell.com/support/manuals/us/en/19/poweredge-r830/r830_om/supported-configurations-for-the-poweredge-r830-system?guid=guid-01303b2b-f884-4435-b4e2-57bec2ce225a&lang=en-us)

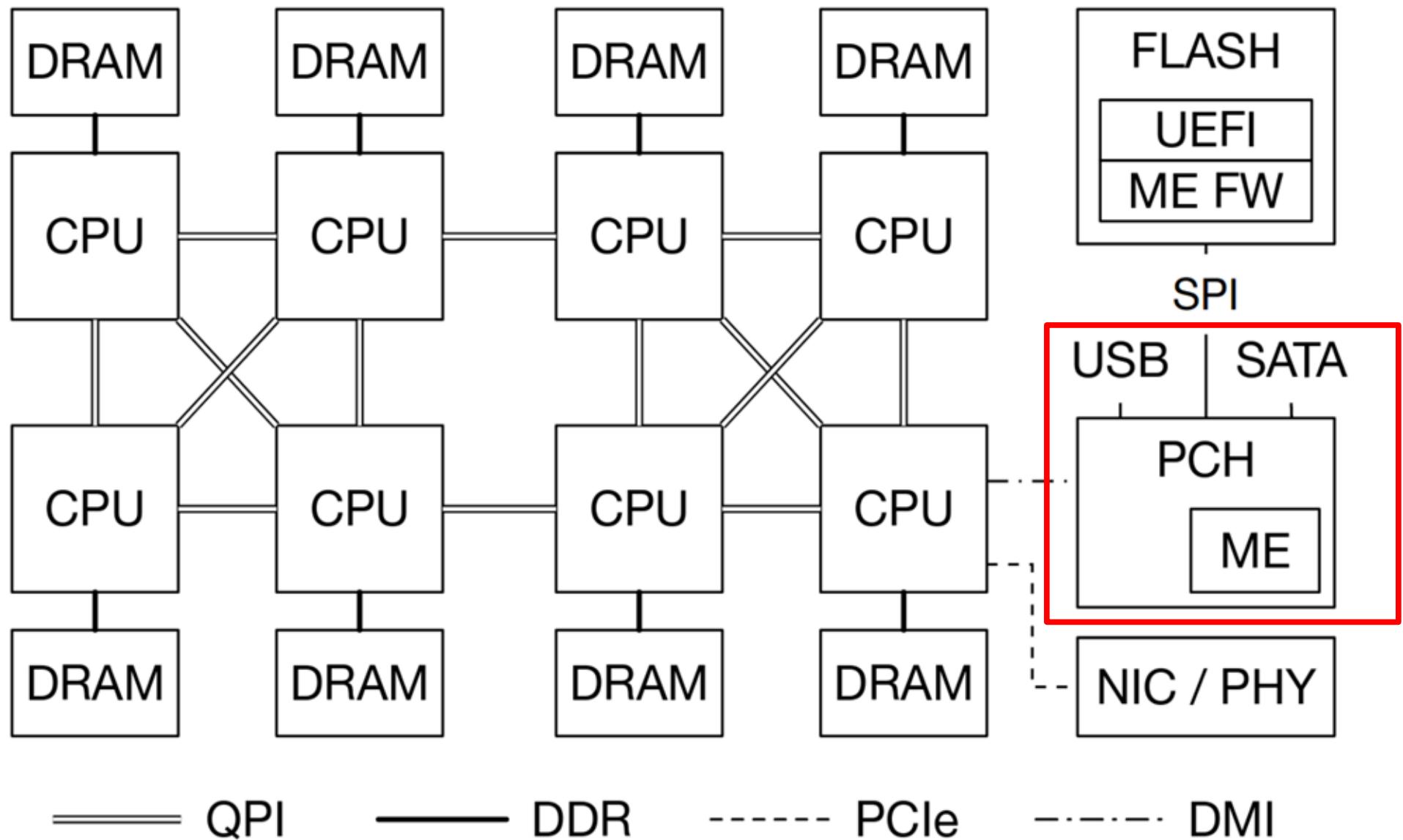
# Multi-socket machines



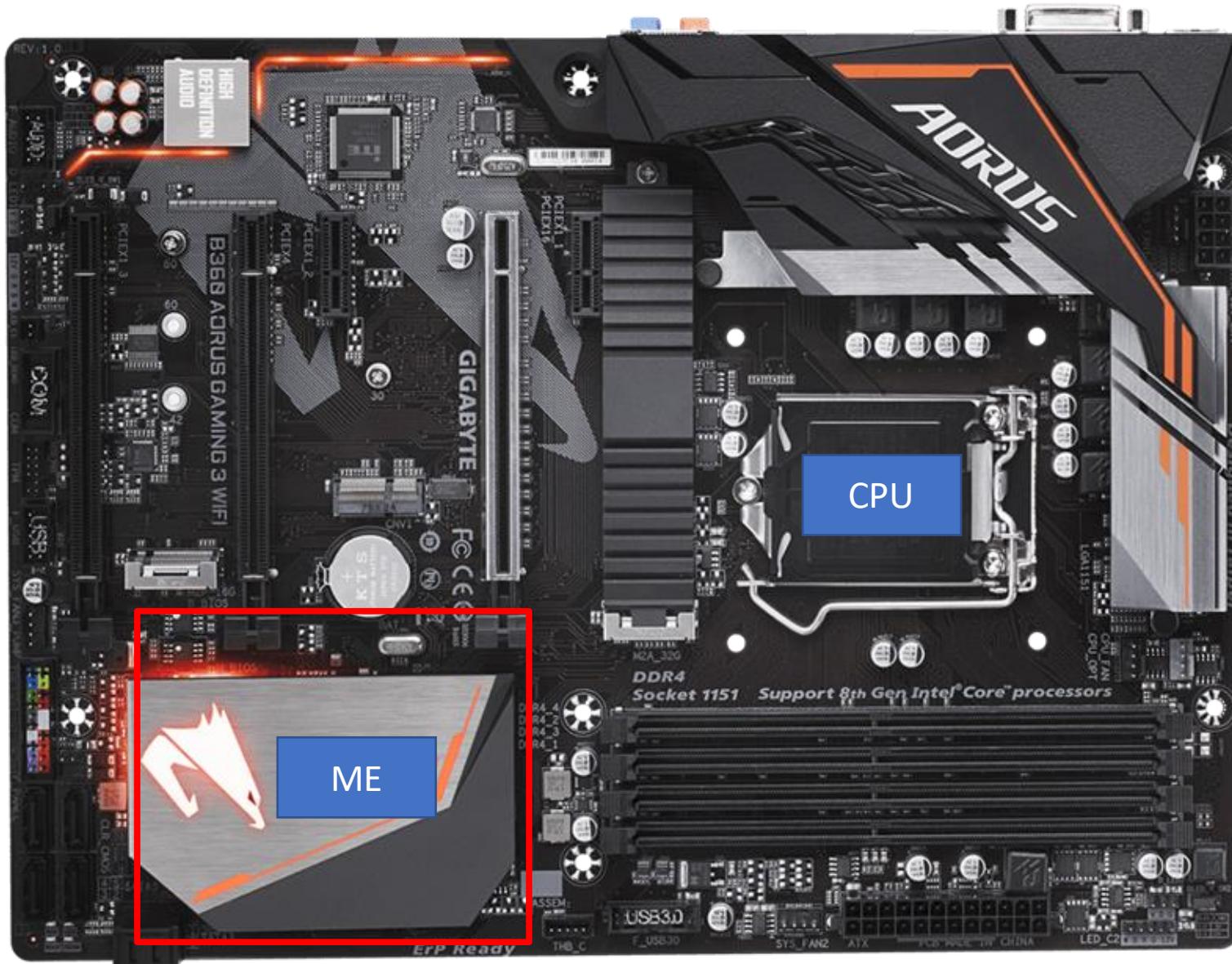
# PC motherboard components



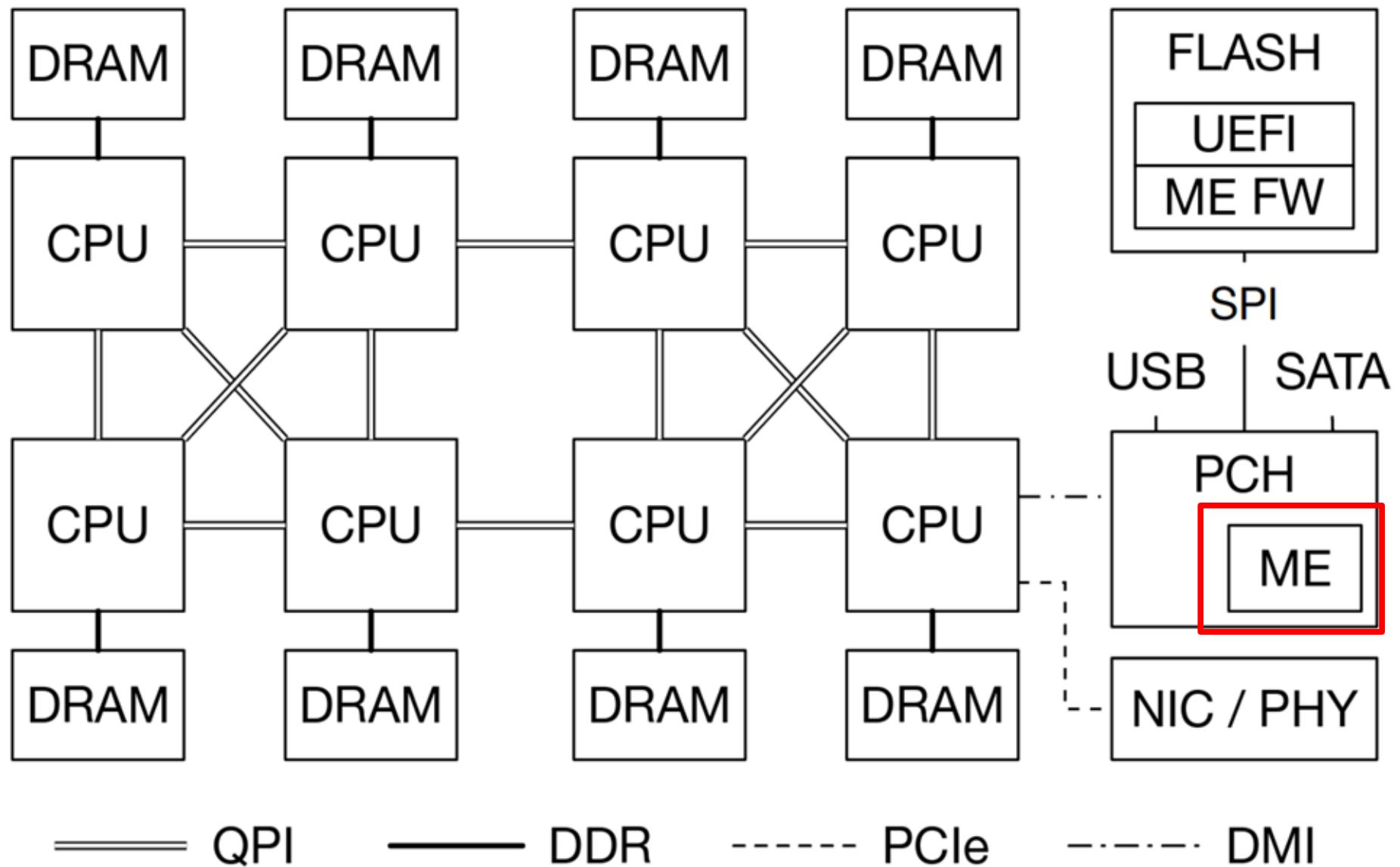
# PCH – Platform Controller Hub



# B360 AORUS Motherboard



# ME gets power before CPUs



# Intel Management Engine (ME)

- Full-featured computer
  - Intel Quark x86-based 32-bit CPU
  - Internal RAM (1.7MB)
- Can access all DRAM via DMA
- Can control boot chain
- Can access network interface (NIC) on the motherboard
  - Has its own MAC and IP address
  - Via System Management Bus (SMBus)
  - Or an ATM compatible NIC
- Connected to the power supply
  - Stays on as long as power is provided to power supply

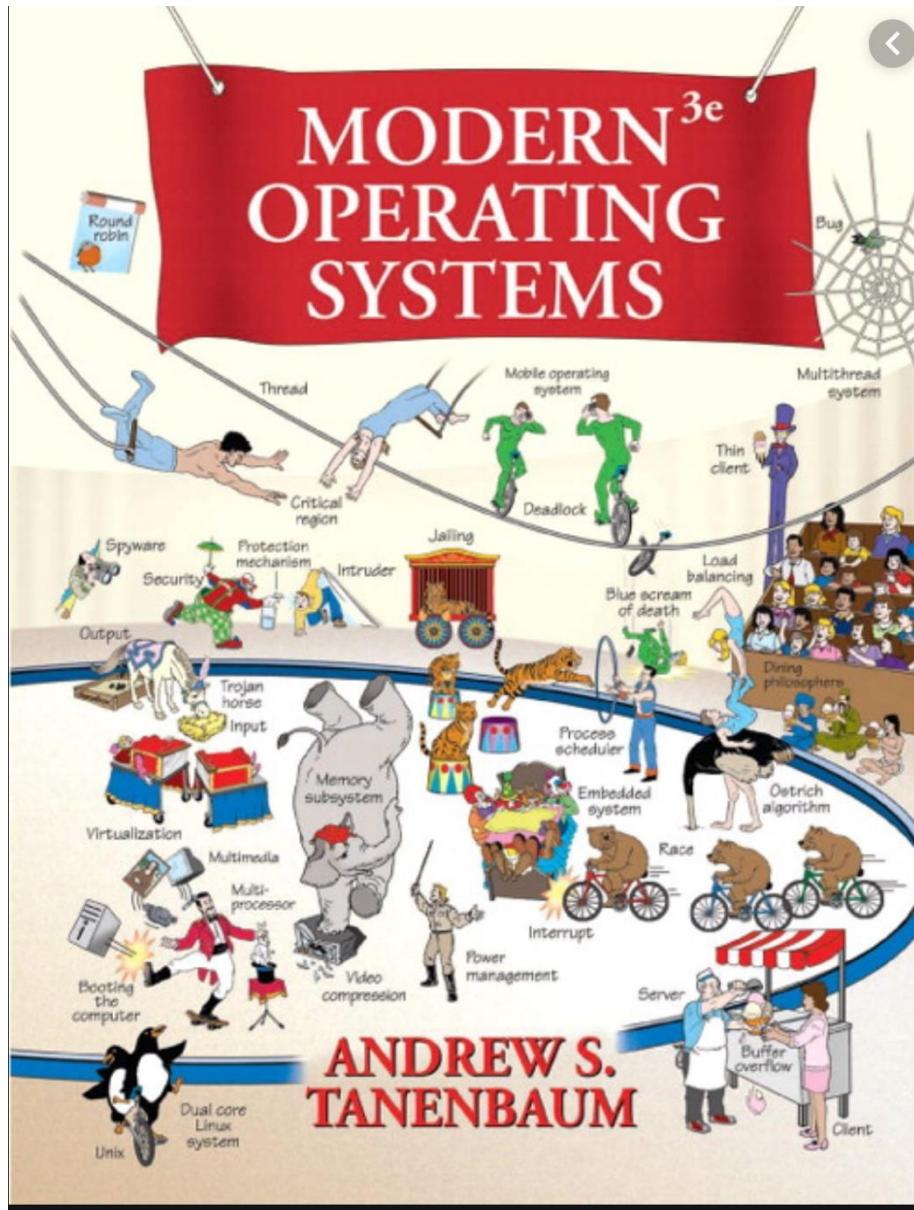
# ME: Theft prevention use-case

- In S5 (computer off) ME cannot access DRAM
  - DRAM is off
- But ME can use its internal memory
  - ME can disable a stolen laptop equipped with cellular modem remotely
- As long as power is connected
- And **cell** network has **signal**

# Intel Management Engine (ME)

- All modern motherboard chips contain ME
- Part of Active Management Technology (AMT)
- Convenient way for administrators to fix your machine remotely
  - Obviously, a huge opportunity for an attack

# What's running there?



Have you read  
“Modern  
Operating  
Systems?”

# What's running there?

Do you ever read “Modern Operating Systems”?

POSITIVE TECHNOLOGIES

```
> strings vfs
...
..\..\src\os\servers\vfs\misc.c
FS: bogus child for forking
FS: forking on top of in-use child
...
```

MINIX3  
by Andrew Tanenbaum

```
Directory of minix3-master\servers\vfs
...
14.03.2010 23:52 14'978 main.c
14.03.2010 23:52 741 Makefile
14.03.2010 23:52 17'653 misc.c
14.03.2010 23:52 677 mmap.c
14.03.2010 23:52 15'650 mount.c
...
```

"FS: bogus child for forking"

All Images Videos News Shopping More Settings Tools

6 results (0.34 seconds)

misc.c in minix-filesystem | source code search engine - Searchcode  
<https://searchcode.com/codesearch/view/55926734/>

childno = \_ENDPOINT\_P(m\_in.child\_endpt); if(childno < 0 || childno >= NR\_PROCS) panic(FILE\_, "FS: bogus child for forking", m\_in.child\_endpt); ...

```
/* PM gives child endpoint, which implies process slot information.
 * Don't call isokendpt, because that will verify if the endpoint
 * number is correct in fproc, which it won't be.
 */
childno = _ENDPOINT_P(m_in.child_endpt);
if(childno < 0 || childno >= NR_PROCS)
    panic(FILE_, "FS: bogus child for forking", m_in.child_endpt);
if(fproc[childno].fp_pid != PID_FREE)
    panic(FILE_, "FS: forking on top of in-use child", childno);
```

# Modern firmware is unreliable and insecure

## Inherent complexity

- Functionality of a partial or sometimes complete operating system
- BMC, ME and IE
  - Full-featured operating systems, processes, network drivers, etc.

# An Open Letter to Intel

---

Dear Mr. Krzanich,

Thanks for putting a version of MINIX inside the ME-11 management engine chip used on almost all recent desktop and laptop computers in the world. I guess that makes MINIX the most widely used computer operating system in the world, even more than Windows, Linux, or MacOS. And I didn't even know until I read a [press report](#) about it. Also [here](#) and [here](#) and [here](#) and [here](#) and [here](#) (in Dutch), and a bunch of other places.

I knew that Intel had some potential interest in MINIX several years ago when one of your engineering teams contacted me about some secret internal project and asked a large number of technical questions about MINIX, which I was happy to answer. I got another clue when your engineers began asking me to make a number of changes to MINIX, for example, making the memory footprint smaller and adding #ifdefs around pieces of code so they could be statically disabled by setting flags in the main configuration file. This made it possible to reduce the memory footprint even more by selectively disabling a number of features not always needed, such as floating point support. This made the system, which was already very modular since nearly all of the OS runs as a collection of separate processes (normally in user mode), all of which can be included or excluded in a build, as needed, even more modular.

Also a hint was the discussion about the license. I (implicitly) gathered that the fact that MINIX uses the Berkeley license was very important. I have run across this before, when companies have told me that they hate the GPL because they are not keen on spending a lot of time, energy, and money modifying some piece of code, only to be required to give it to their competitors for free. These discussions were why we put MINIX out under the Berkeley license in 2000 (after prying it loose from my publisher).

After that intitial burst of activity, there was radio silence for a couple of years, until I read in the media (see above) that a modified version of MINIX was running on most x86 computers, deep inside one of the Intel chips. This was a complete surprise. I don't mind, of course, and was not expecting any kind of payment since that is not required. There isn't even any suggestion in the license that it would be appreciated.

The only thing that would have been nice is that after the project had been finished and the chip deployed, that someone from Intel would have told me, just as a courtesy, that MINIX was now probably the most widely used operating system in the world on x86 computers. That certainly wasn't required in any way, but I think it would have been polite to give me a heads up, that's all.

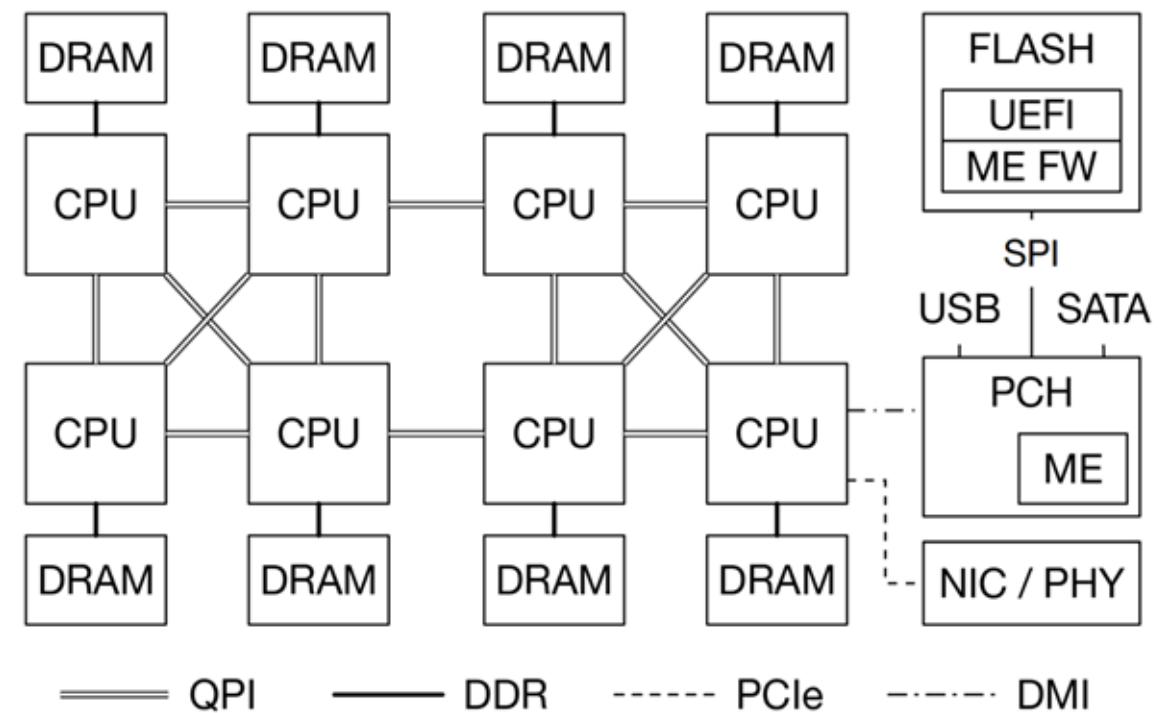
If nothing else, this bit of news reaffirms my view that the Berkeley license provides the maximum amount of freedom to potential users. If they want to publicize what they have done, fine. By all means, do so. If there are good reasons not to release the modified code, that's fine with me, too.

Yours truly,

Andrew S. Tanenbaum

# ME starts first

- Reads its initialization code from the BIOS chip
- Via the SPI bus



# Bootstrap processor (BSP)

- One of the logical processors is chosen as **bootstrap processor** (BSP)
  - Will start initialization
- Others become “**application processors**” (AP)
  - Waiting for a special interrupt from the BSP

# BSP starts reading BIOS

- Executes instructions stored in the BIOS chip
  - Initially, BIOS' ROM is **mapped into memory** address space of the CPU, so it can execute it
  - An interesting detail is that BSP starts with **DRAM disabled**
    - Hence there is no stack to call functions
    - What can be done?

# BSP starts without DRAM

- Custom-written assembly code that uses no stack
- Or a [ROMCC](#) compiler
  - Generates code from C that uses no stack
  - Used in the [coreboot](#) project

# Cache-as-RAM

- Use CPU caches as temporary replacement for RAM
  - Initialize DRAM
  - Copy BIOS firmware into DRAM and continue

# BIOS firmware

## Initialize

- Interrupt controllers
- Devices, e.g., network interfaces
- If one of PCI devices contains “option ROM”  
load and execute it
  - Network cards may contain iPXE ROM
  - Implement boot from the network host

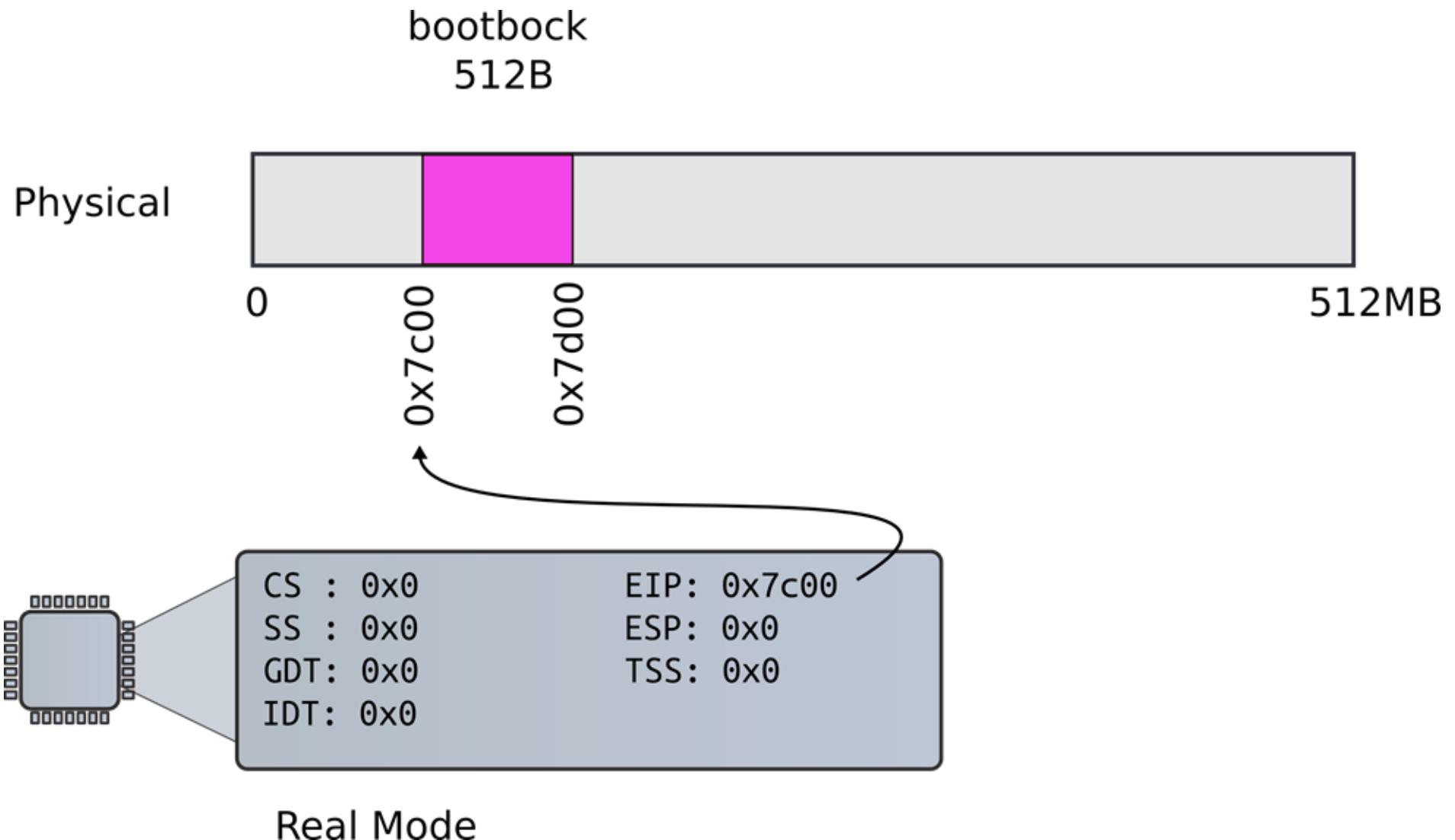
# System Management Mode

- Another compartment that runs underneath your OS or a hypervisor
  - Initialized by BIOS
  - Protected with hardware memory mechanisms
  - OS cannot access this region of memory
  - Runs under your OS and the hypervisor
  - Receives interrupts periodically, can take over the entire system any time
- **No way to disable**

# BIOS loads the boot loader

- BIOS ends by loading a **boot loader**
- Modern BIOSes can load the boot loader from a variety of sources (hard disks, USB drives, optical disks)
- Default way is to load the **first sector** (512 bytes) from disk into the memory location at **0x7c00**
- BIOS then starts executing instructions at the address 0x7c00
  - This is exactly what we see when we run xv6 under QEMU
  - QEMU emulates hardware: runs BIOS, follows the same protocol

# BIOS loads bootloader



# Outline of the boot sequence

# Outline of the boot sequence

- Setup segments (data and code)
- Switch to protected mode
- Load GDT (segmentation is on)
- Setup stack (to call C functions)
- Load kernel from disk
- Setup first page table
  - 2 entries mapping [ 0 : 1GB ] physical to two virtual regions [ 0 : 1GB ] and [8TB: 8TB + 1GB]
- Setup high-address stack
- Jump to main()

# 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

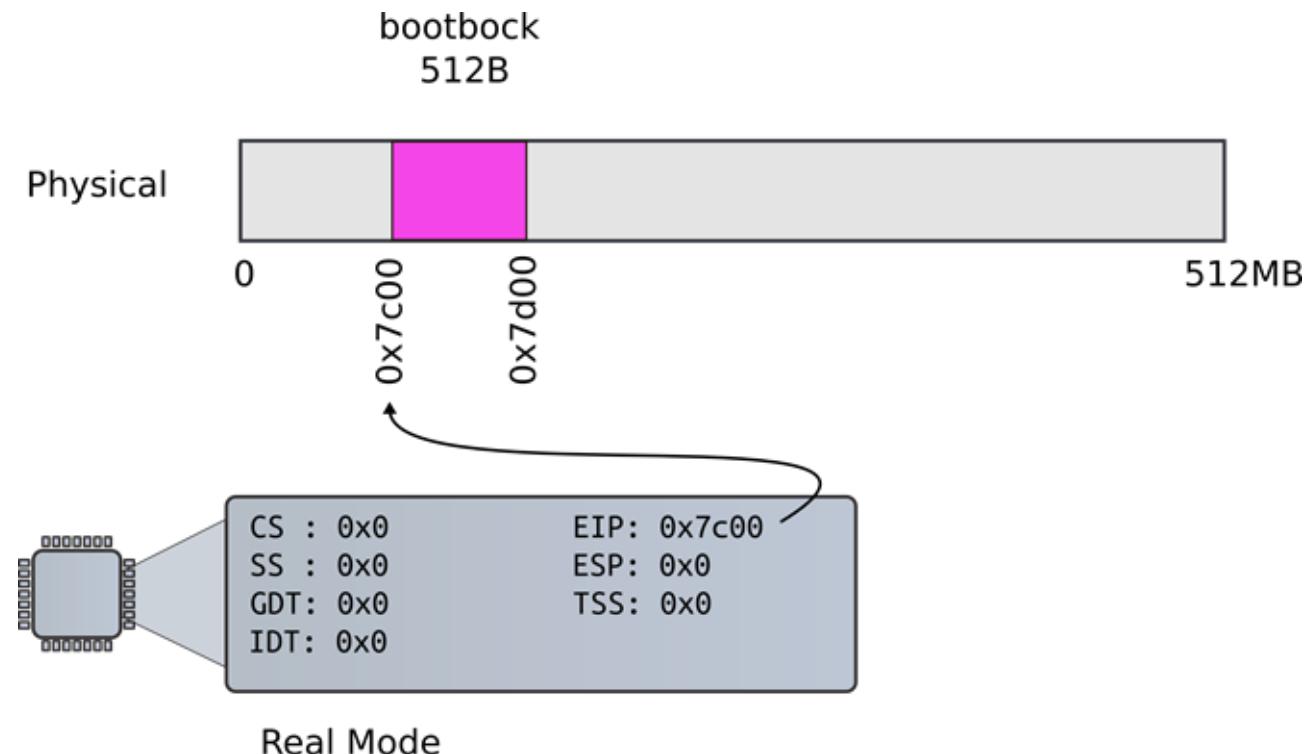
xv6/bootasm.S [bootloader]

# Why start happens to be 0x7c00?

9111 start:

9112 cli # BIOS enabled interrupts; disable

9113



xv6/bootasm.S [bootloader]

# Linker is instructed to link the boot block code in the Makefile

9111 start:

9112 cli # BIOS enabled interrupts; disable

9113

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

xv6/Makefile

# 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

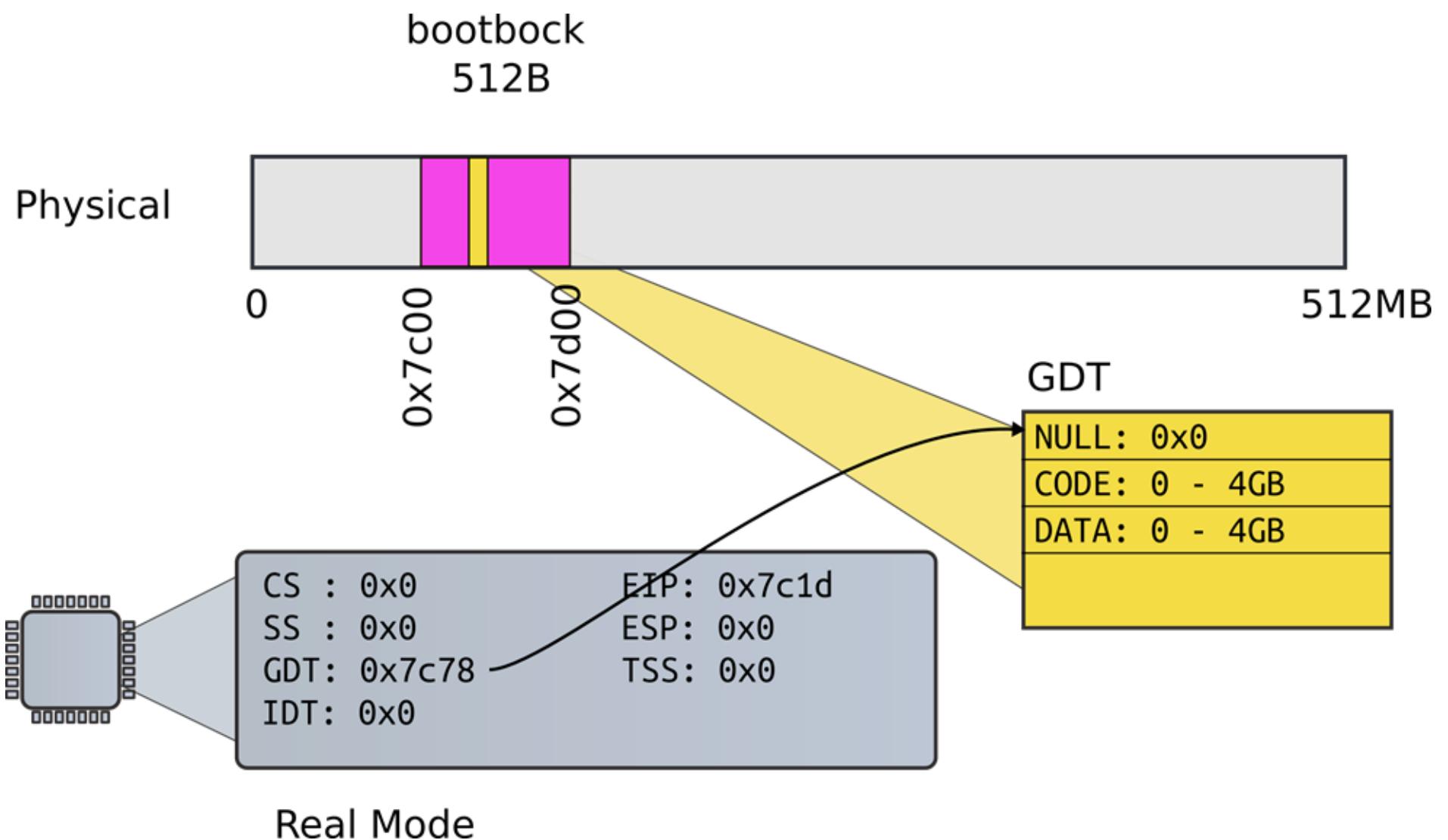
9142 movl %cr0, %eax

9143 orl \$CR0\_PE, %eax

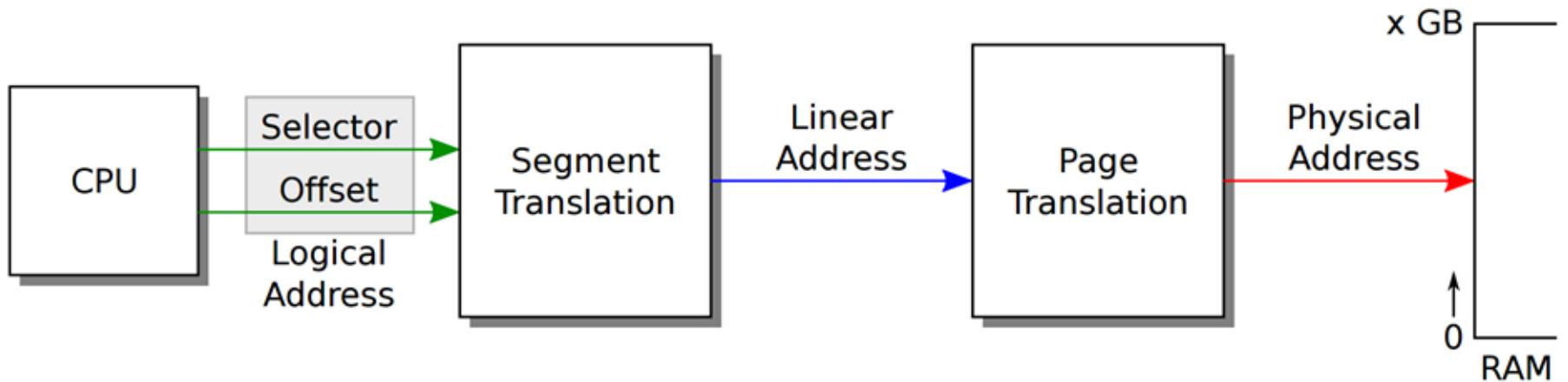
9144 movl %eax, %cr0

xv6/bootasm.S [bootloader]

# Load GDT



# Recap: complete address translation



# 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
```

xv6/bootasm.S [bootloader]

# 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
```

xv6/bootasm.S [bootloader]

# Actual switch

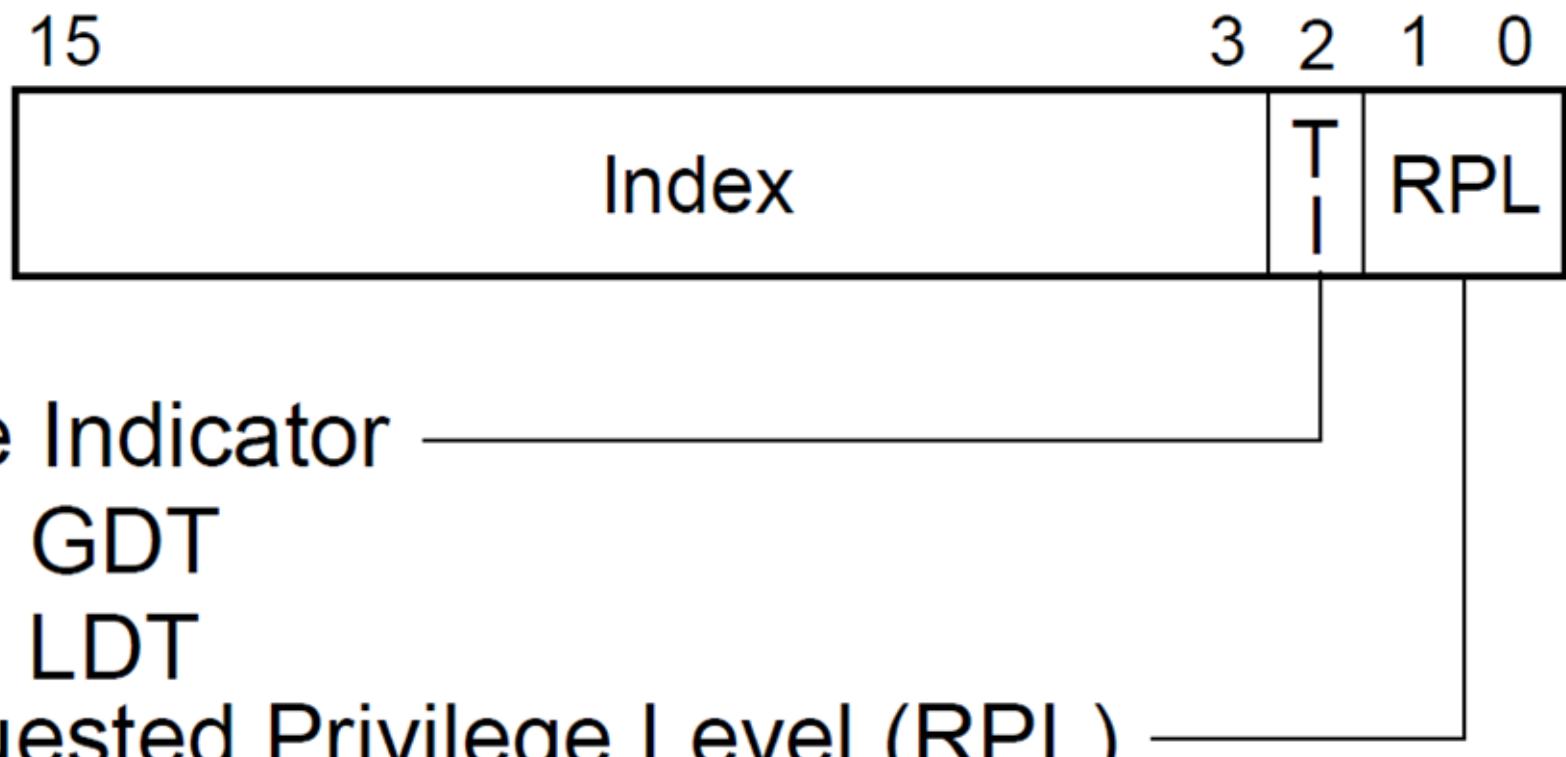
- Use long jump to change code segment

9153 ljmp \$(SEG\_KCODE<<3), \$start32

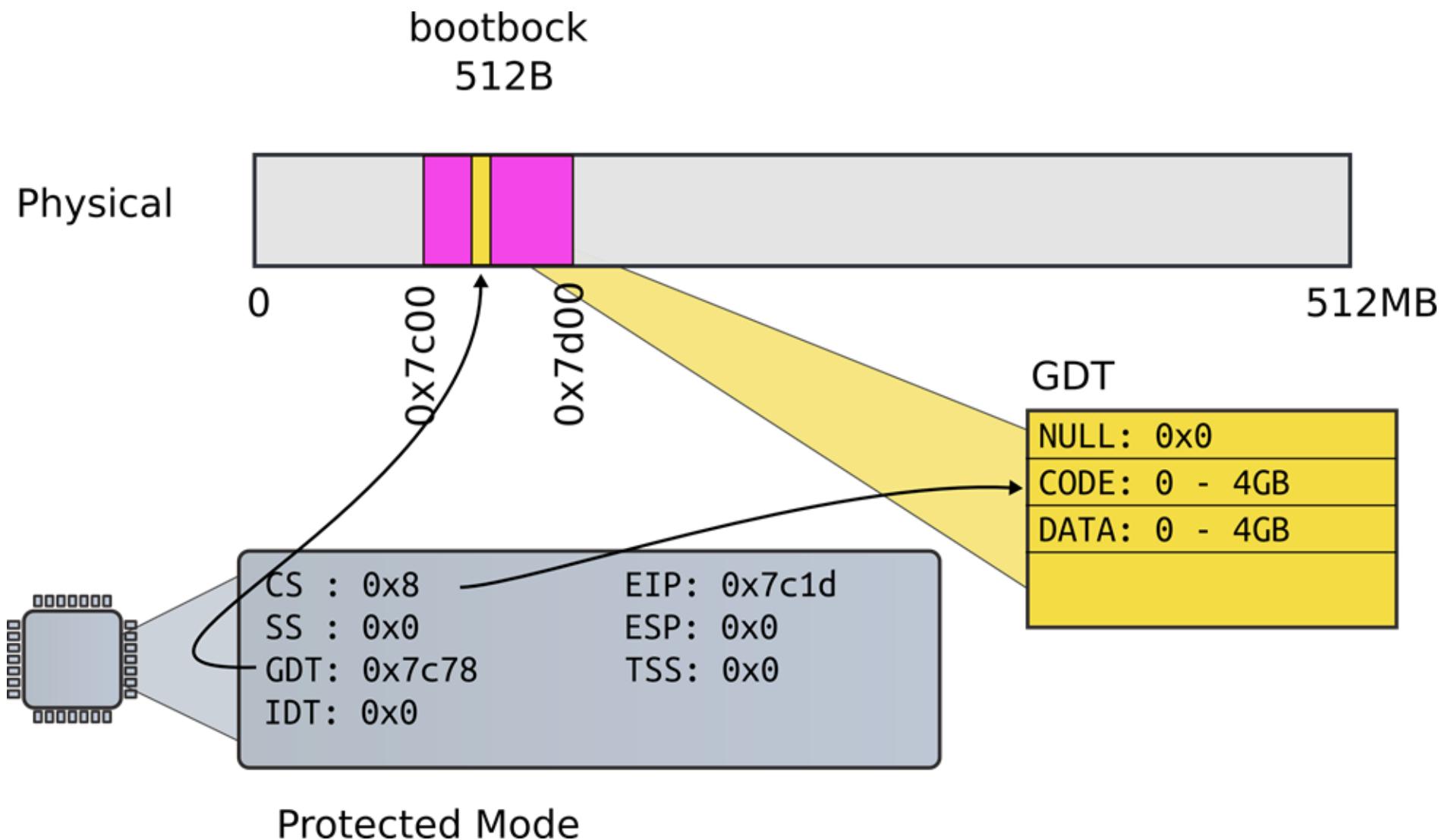
- Explicitly specify code segment, and address
- Segment is 0b1000 (0x8)

# Why CS is 0x8, not 0x1?

- Segment selector:



# Long jump



# 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

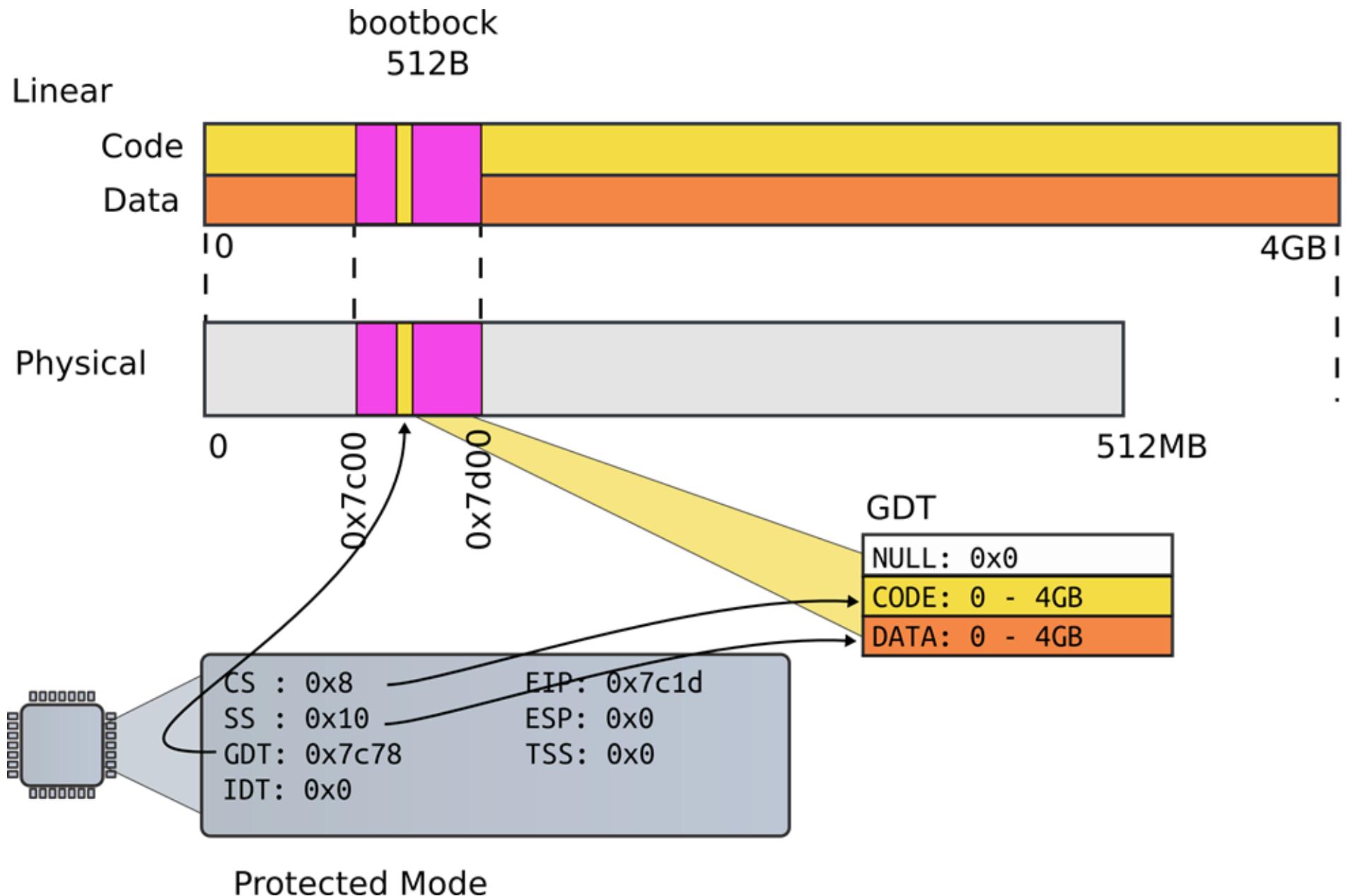
9162 movw \$0, %ax # Zero segments not ready for use

9163 movw %ax, %fs # -> FS

9164 movw %ax, %gs # -> GS

xv6/bootasm.S [bootloader]

# Segments



# Setup stack

- Why do we need a stack?

9166 movl \$start, %esp

9167 call bootmain

xv6/bootasm.S [bootloader]

# Setup stack

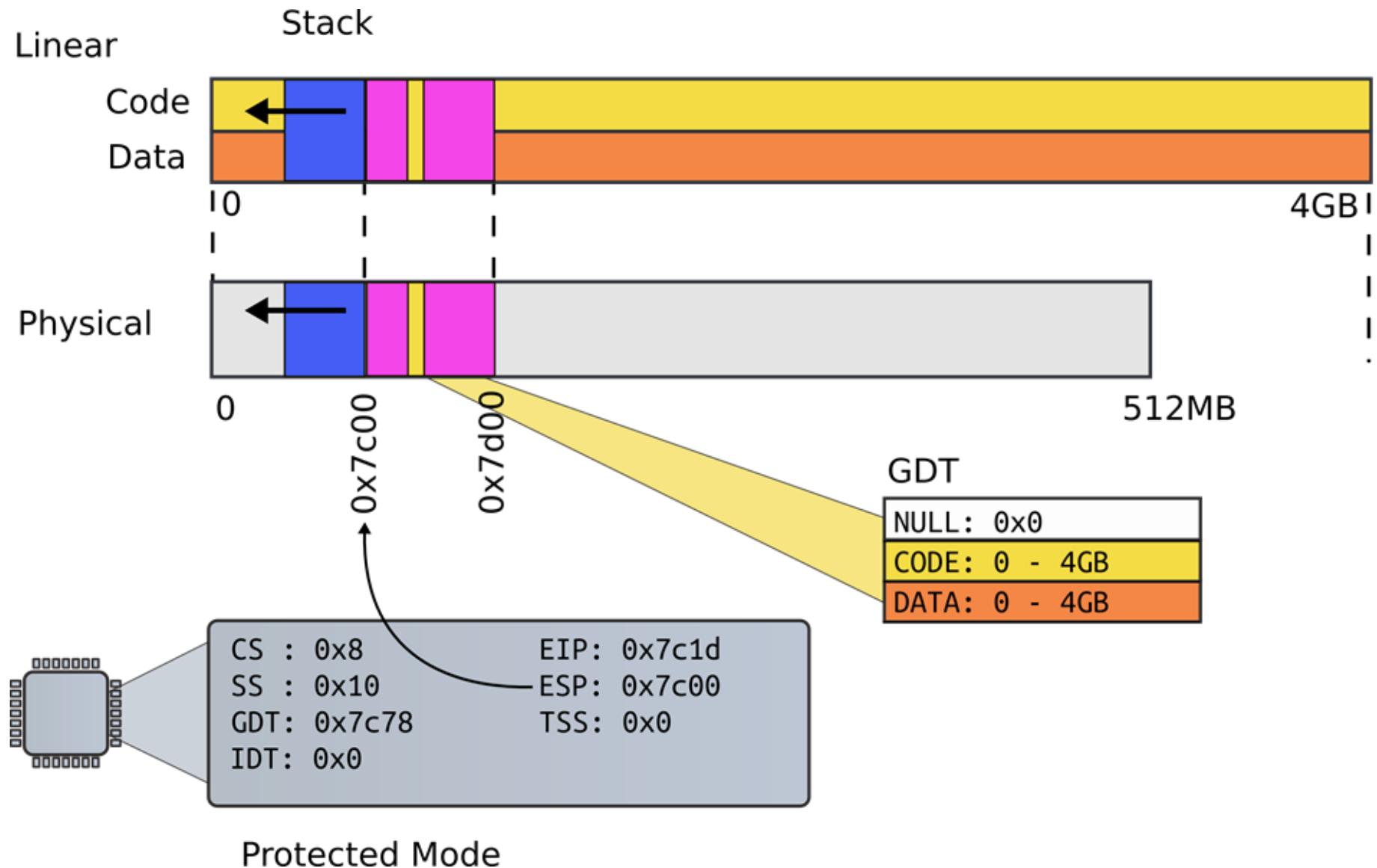
- Need stack to use C
- Function invocations
- Note, there were no stack instructions before that

9166 movl \$start, %esp

9167 call bootmain

xv6/bootasm.S [bootloader]

# First stack



# Invoke first C function

9166 movl \$start, %esp

9167 call bootmain

# 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
```

xv6/bootmain.c [bootloader]

```
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**

xv6/bootmain.c [bootloader] **from disk**

# How do we read disk?

9257 // 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 do we read disk (cont)?

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

# How do we read disk?

9257 // 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 }

xv6/bootmain.c [bootloader]

```
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 }
```

## Call kernel entry

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

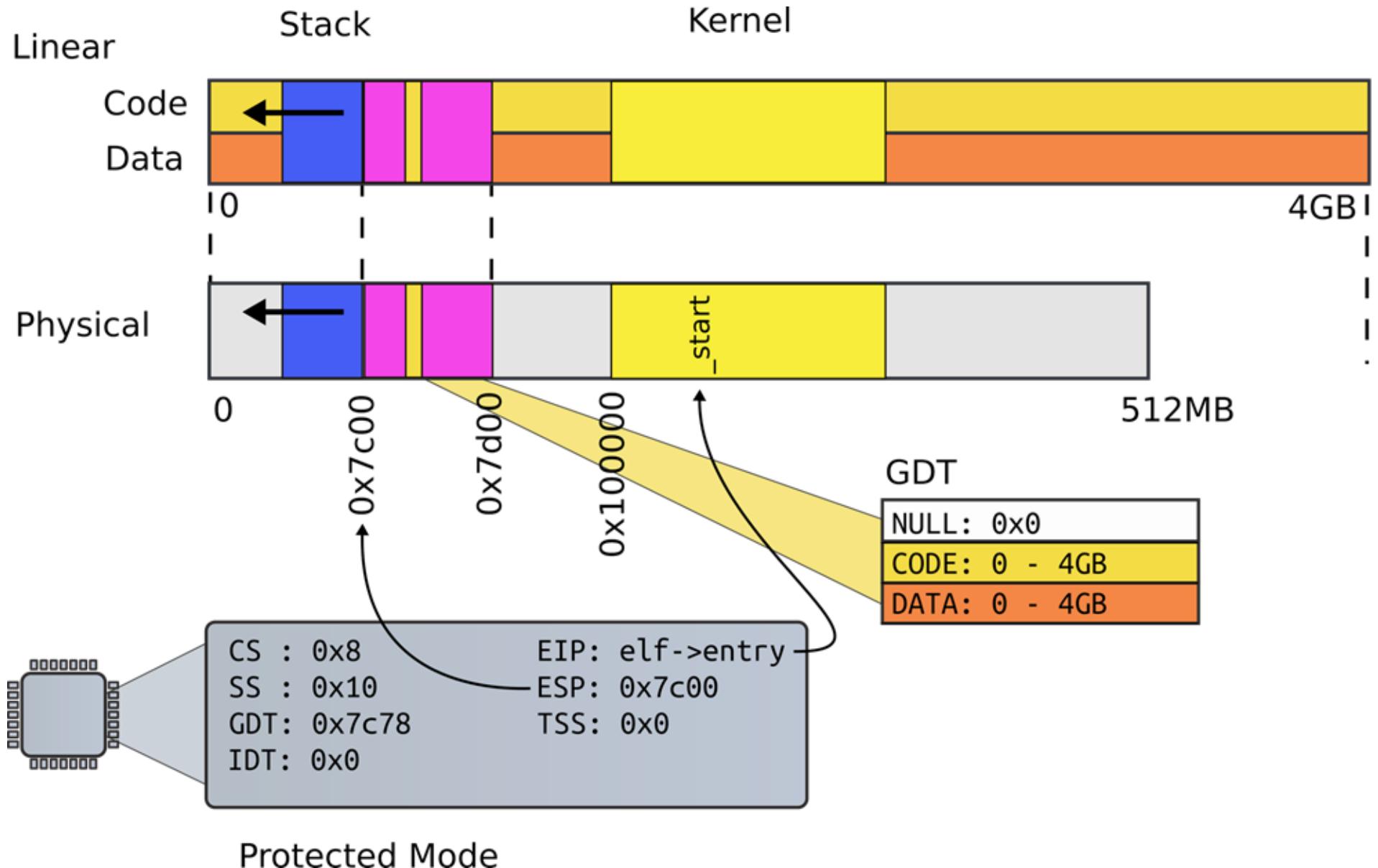
1146 movl %cr4, %eax

1147 orl \$(CR4\_PSE), %eax

1148 movl %eax, %cr4

entry(): kernel ELF entry  
xv6/entry.S [kernel]

# Kernel



# entry(): kernel ELF entry

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

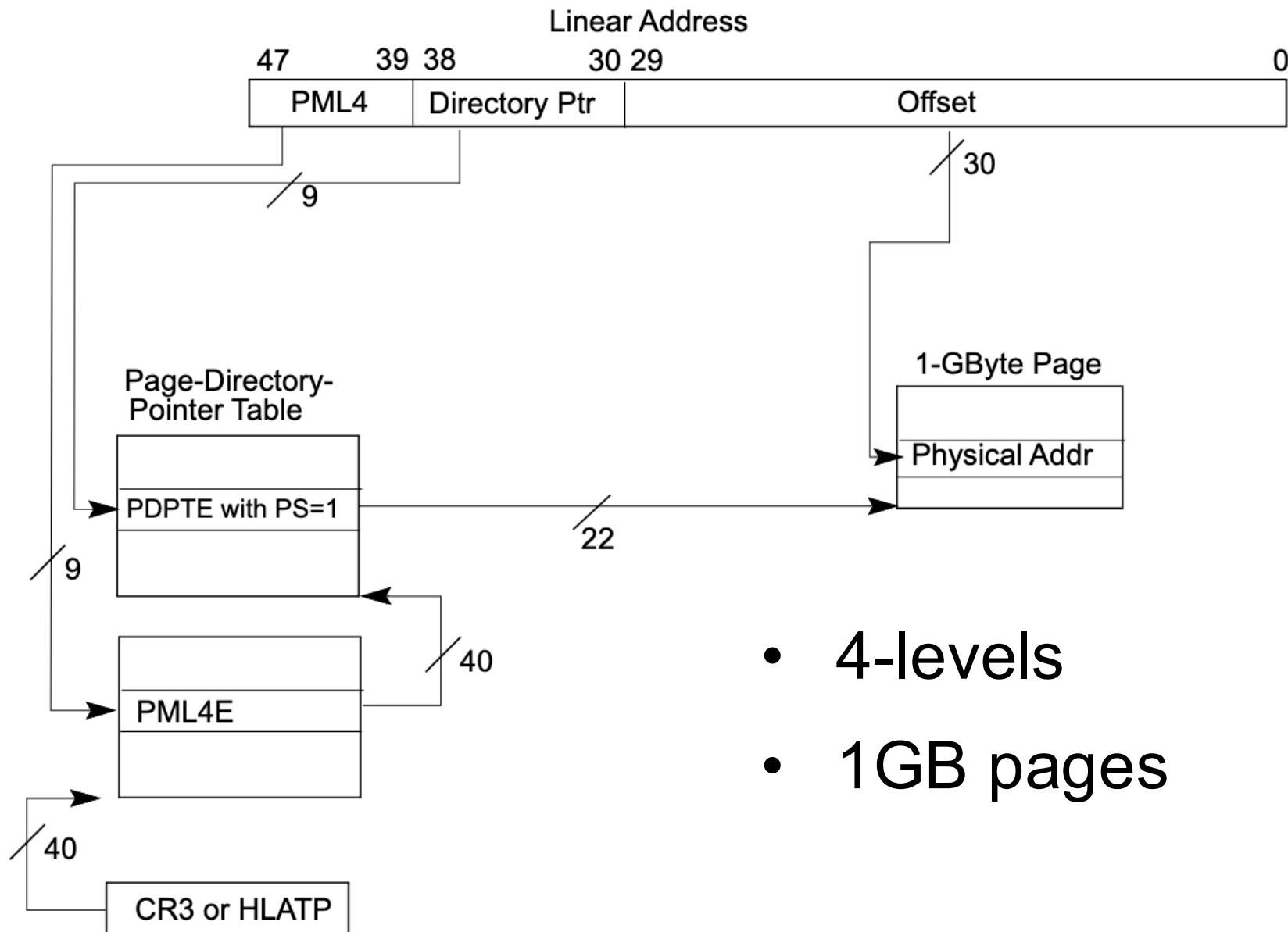
1148 movl %eax, %cr4

xv6/entry.S [kernel]

# 64bit x86 supports three page sizes

- 4KB pages
- 2MB pages
- 1GB pages

# Page translation for 1GB pages



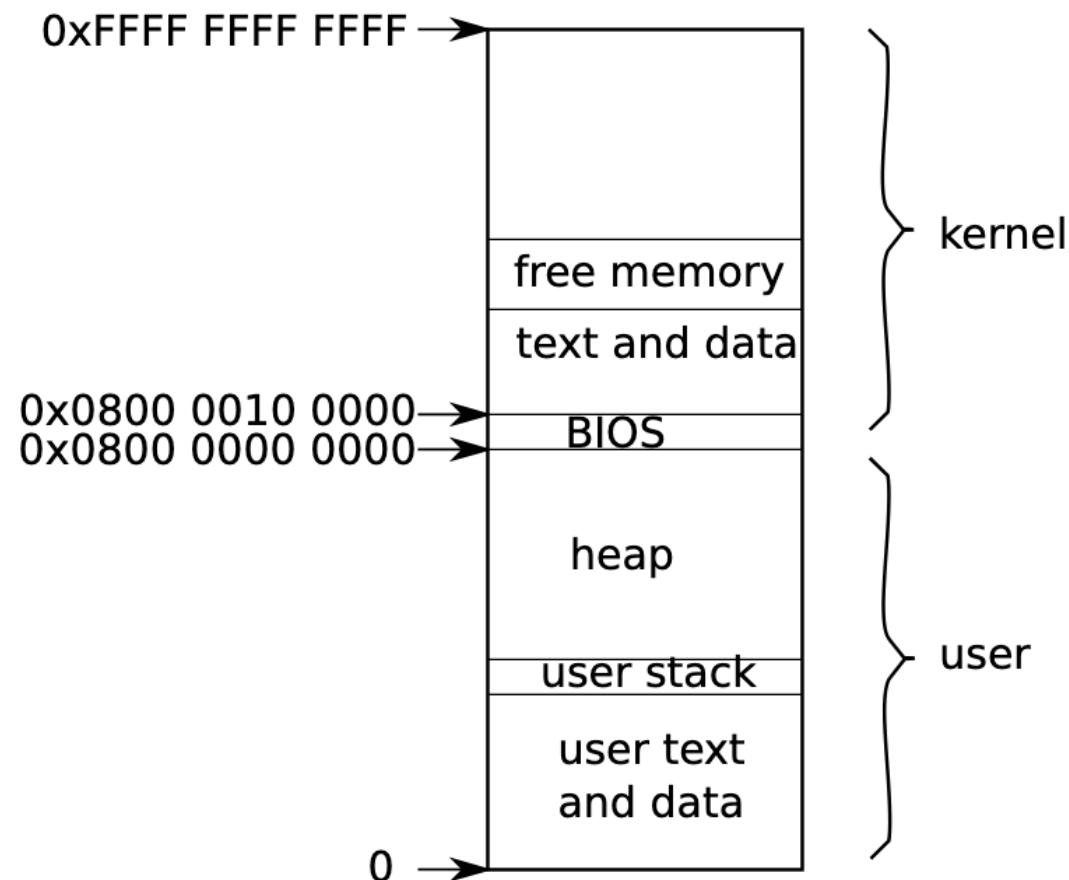
# Set up page directory

1150 # Load CR3 with the physical address of PML4.

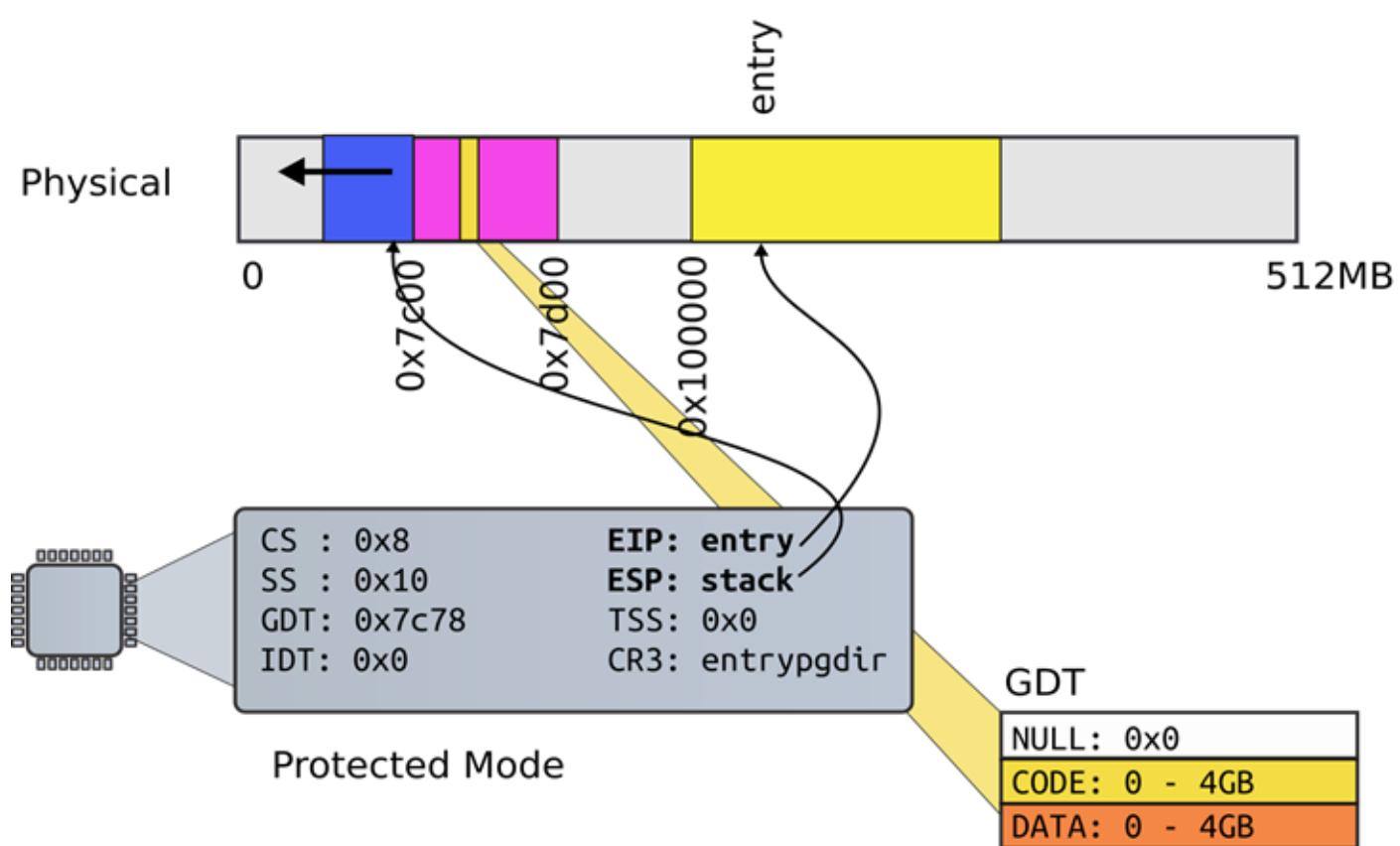
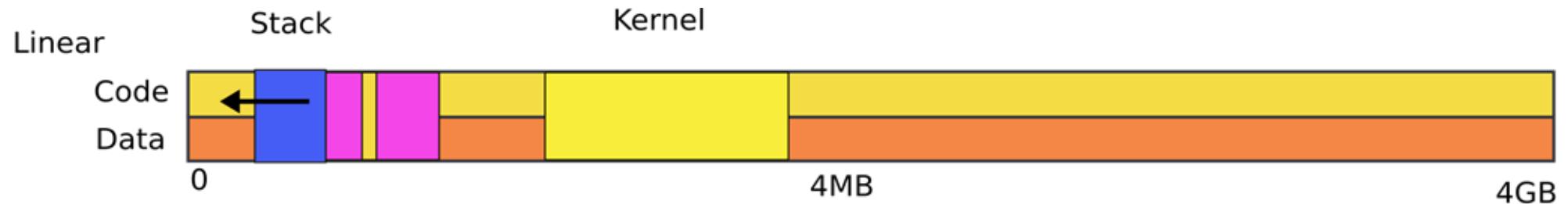
1151 movl \$(V2P\_WO(entrypml4)), %eax

1152 movl %eax, %cr3

# Our goal: a split address space



**Figure 1-2.** Layout of a virtual address space



# Boot-time page table

- Two 1GB entries (large pages)
- Entry #0
  - $0x0 - 1GB$  (virt)  $\rightarrow 0x0 - 1GB$  (phys)
- Entry #256
  - $0x8GB - 8GB + 1GB$  (virt)  $\rightarrow 0x0 - 1GB$  (phys)

```
1466 __attribute__((__aligned__(PGSIZE)))  
  
1467 pml4e_t entrypml4[NPML4ENTRIES] = {  
  
1468 // Flags below should be added with " | " and not with "+",  
  
1469 // however, " | " seems to complex for the link editor,  
  
1470 // so the compiler refuses to compile the code.  
  
1471 // The use of "+" is valid since PTE_* are only single bits.  
  
1472 [0] = V2P(identitymap) + PTE_P + PTE_W,  
  
1473 [PML4X(KERNBASE)] = V2P(kernmap) + PTE_P + PTE_W,  
  
1474 };
```

# First page table

```
1466 __attribute__((__aligned__(PGSIZE)))  
1467 pml4e_t entrypml4[NPM_4ENTRIES] = {  
1468 // Flags below should be added with " | " and not with "+",  
1469 // however, " | " seems to complex for the link editor,  
1470 // so the compiler refuses to compile the code.  
1471 // The use of "+" is valid since PTE_* are only single bits.  
1472 [0] = V2P(identitymap) + PTE_P + PTE_W,  
1473 [PML4X(KERNBASE)] = V2P(kernmap) + PTE_P + PTE_W,  
1474 };
```

# First page table

# First page table (cont)

0870 // Page directory and page table constants.

0871 #define NPML4ENTRIES 512 // # page map level 4 entries

```
1466 __attribute__((__aligned__(PGSIZE)))  
  
1467 pml4e_t entrypml4[NPML4ENTRIES] = {  
  
1468 // Flags below should be added with " | " and not with "+",  
  
1469 // however, " | " seems to complex for the link editor,  
  
1470 // so the compiler refuses to compile the code.  
  
1471 // The use of "+" is valid since PTE_* are only single bits.  
  
1472 [0] = V2P(identitymap) + PTE_P + PTE_W,  
1473 [PML4X(KERNBASE)] = V2P(kernmap) + PTE_P + PTE_W,  
  
1474 };
```

# First page table

# First page table (cont)

0850 // page map level 4 index

0851 #define PML4X(va) (((uint64)(va) >> PML4SHIFT) & 0x1FF)

```
1466 __attribute__((__aligned__(PGSIZE)))  
  
1467 pml4e_t entrypml4[NPML4ENTRIES] = {  
  
1468 // Flags below should be added with " | " and not with "+",  
  
1469 // however, " | " seems to complex for the link editor,  
  
1470 // so the compiler refuses to compile the code.  
  
1471 // The use of "+" is valid since PTE_* are only single bits.  
  
1472 [0] = V2P(identitymap) + PTE_P + PTE_W,  
  
1473 [PML4X(KERNBASE)] = V2P(kernelmap) + PTE_P + PTE_W,  
  
1474 };
```

# First page table

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,

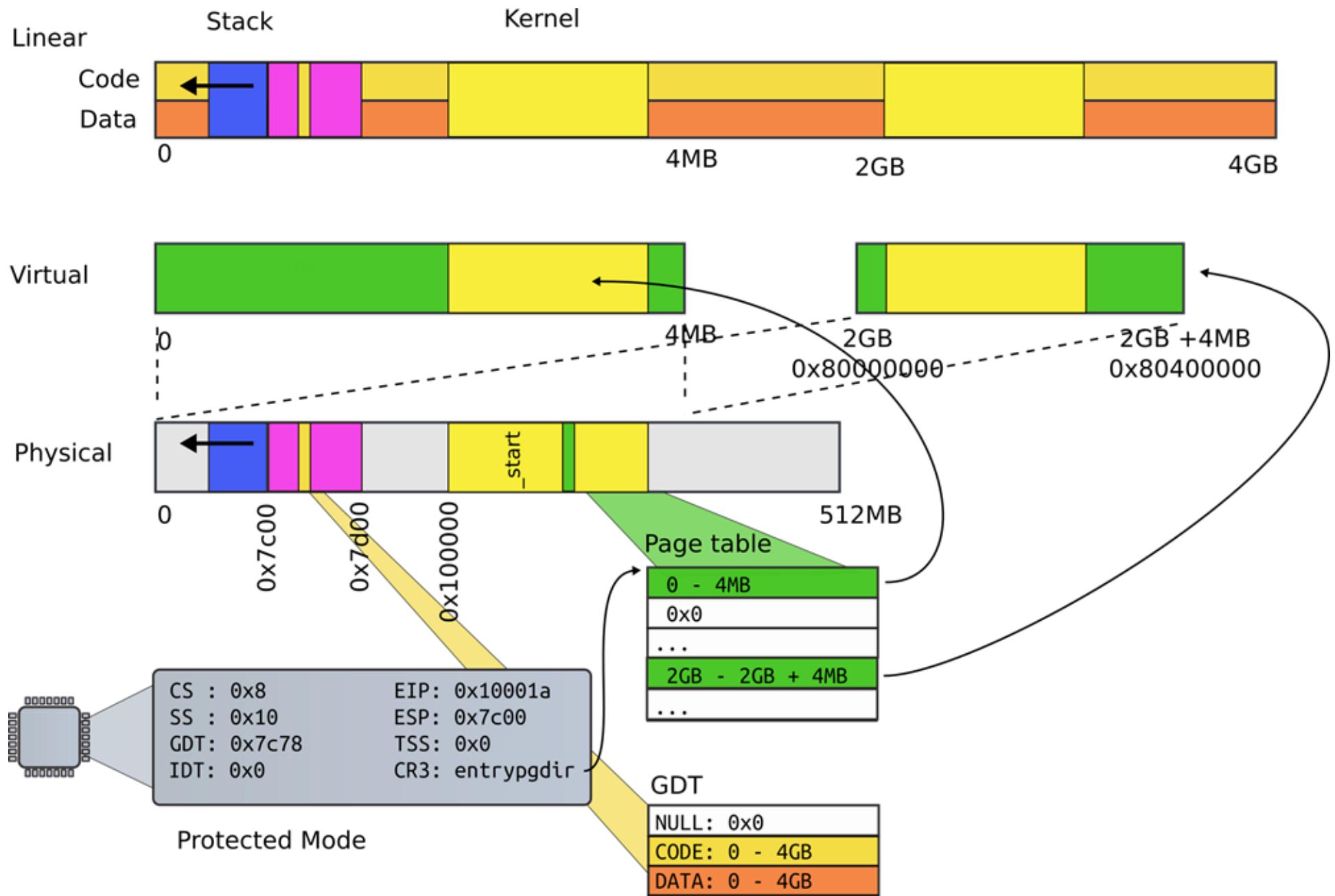
1415 // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)

1416 [KERNBASE>>PDXSHIFT] = (0) | PTE\_P | PTE\_W | PTE\_PS,

1417};

# First page table

# First page table



# Turn on paging

1152 # Turn on paging.

1153 movl %cr0, %eax

1154 orl \$(CRO\_PG | CRO\_WP), %eax

1155 movl %eax, %cr0

xv6/entry.S [kernel]

# High address stack (4K)

1157 # Set up the stack pointer.

1158 movl \$(stack + KSTACKSIZE), %esp

1159

...

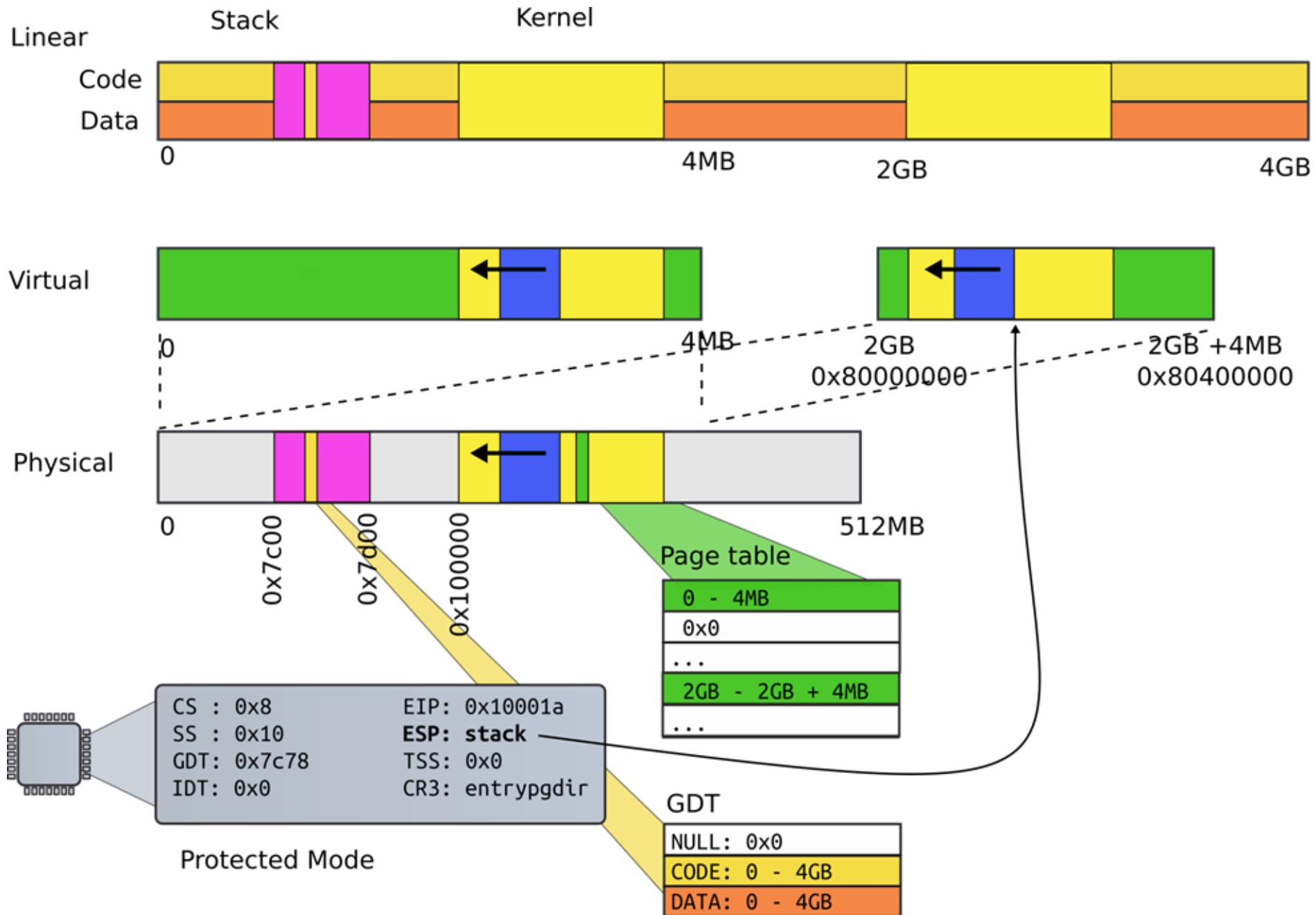
1167 .comm stack, KSTACKSIZE

0151 #define KSTACKSIZE 4096 // size of

per-process kernel stack

xv6/entry.S [kernel]

# 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

xv6/entry.S [kernel]

# 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

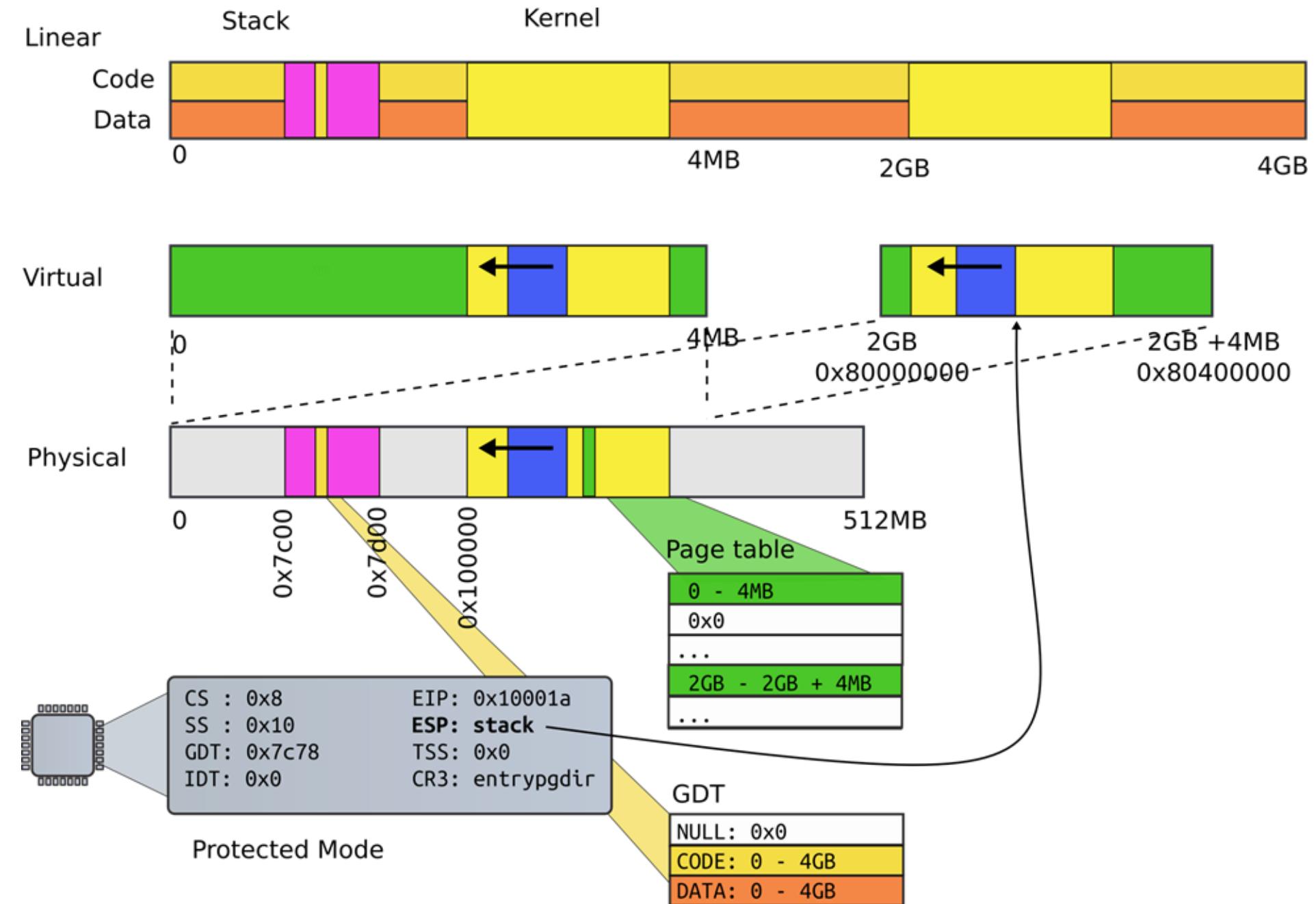
1324   cprintf("\ncpu%d: starting xv6\n\n", cpunum());

...

1340 }

**xv6/main.c [kernel]**

# Poll: [PollEv.com/antonburtsev](https://PollEv.com/antonburtsev)



# 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()

Thank you!

# References

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

```
1. #include <stdio.h>

2. void func_a(void){
3.     printf("func_a\n");
4.     return;
5. }

6. void func_b(void) {
7.     printf("func_b\n");
8.     return;
9. }

10. int main(int ac, char **av)
11. {
12.     void (*fp)(void);

13.     fp = func_b;
14.     fp();
15.     return;
16. }
```

# Function pointers

08048432 <func\_b>:

```
8048432: 55          push %ebp
8048433: 89 e5        mov %esp,%ebp
8048435: 83 ec 18     sub $0x18,%esp
8048438: c7 04 24 07 85 04 08  movl $0x8048507,(%esp)
804843f: e8 ac fe ff ff  call 80482f0 <puts@plt>
8048444: 90          nop
8048445: c9          leave
8048446: c3          ret
```

08048447 <main>:

```
8048447: 55          push %ebp
8048448: 89 e5        mov %esp,%ebp
804844a: 83 e4 f0     and $0xffffffff0,%esp
804844d: 83 ec 10     sub $0x10,%esp
                                # Load pointer to func_p on the stack
8048450: c7 44 24 0c 32 84 04  movl $0x8048432,0xc(%esp)
8048457: 08
8048458: 8b 44 24 0c    mov 0xc(%esp),%eax
804845c: ff d0        call *%eax
804845e: 90          nop
804845f: c9          leave
8048460: c3          ret
```

# Function pointers

08048432 <func\_b>:

```
8048432: 55          push %ebp
8048433: 89 e5        mov %esp,%ebp
8048435: 83 ec 18     sub $0x18,%esp
8048438: c7 04 24 07 85 04 08  movl $0x8048507,(%esp)
804843f: e8 ac fe ff ff  call 80482f0 <puts@plt>
8048444: 90          nop
8048445: c9          leave
8048446: c3          ret
```

08048447 <main>:

```
8048447: 55          push %ebp
8048448: 89 e5        mov %esp,%ebp
804844a: 83 e4 f0     and $0xffffffff0,%esp
804844d: 83 ec 10     sub $0x10,%esp
                      # Load pointer to func_p on the stack
8048450: c7 44 24 0c 32 84 04  movl $0x8048432,0xc(%esp)
8048457: 08          # Move func_b into %eax
8048458: 8b 44 24 0c    mov 0xc(%esp),%eax
804845c: ff d0        call *%eax # Call %eax
804845e: 90          nop
804845f: c9          leave
8048460: c3          ret
```

# Function pointers