cs5460/6460: Operating Systems

Lecture 07: System boot

Anton Burtsev

February, 2026

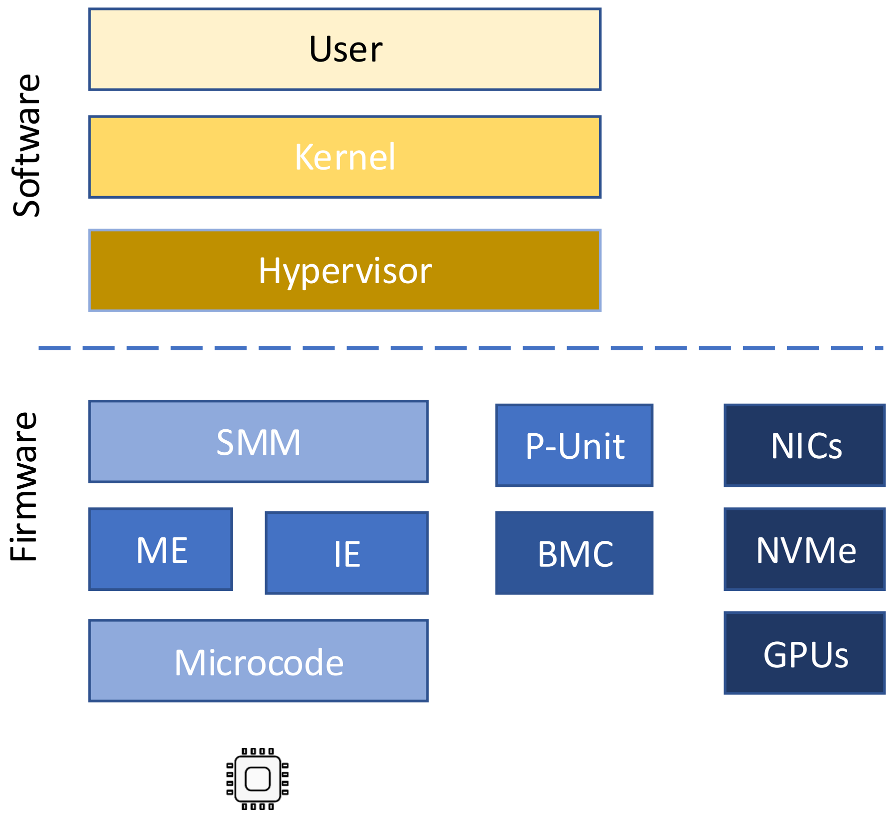
What happens when we turn on the power?

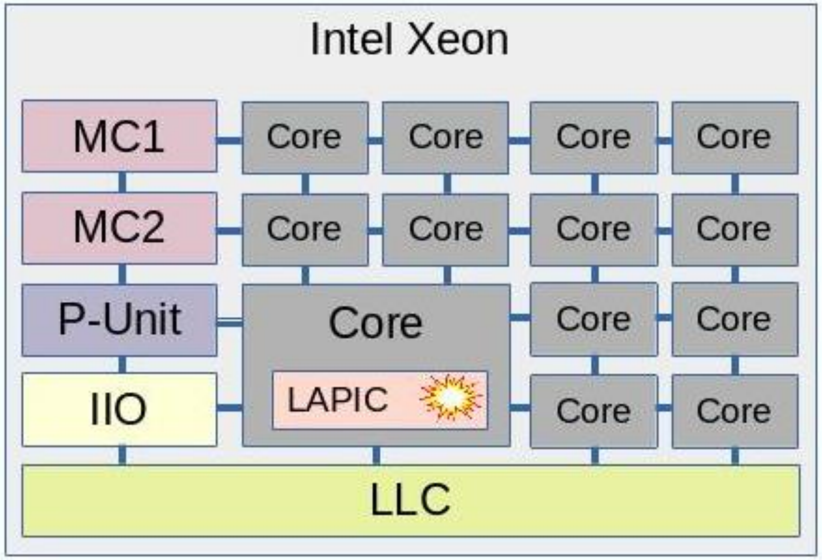
- Well it's complicated
  - Intel SGX Explained is a good start (Section 2.13 [1])
- At a high-level a sequence of software pieces initializes the platform
  - Management engine (ME), microcode, firmware (BIOS), bootloader

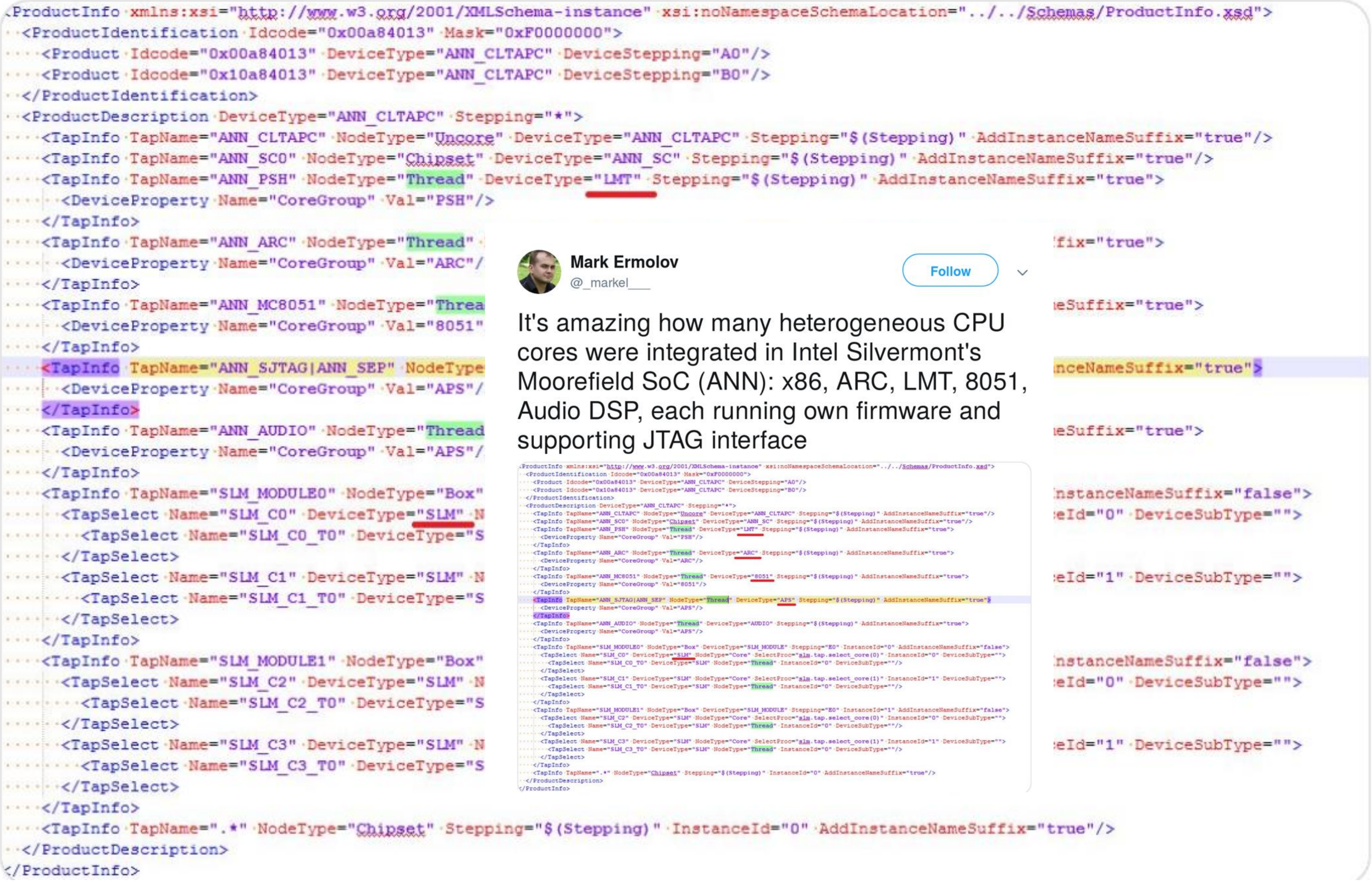
- The most important thing: the OS is not the only software running on the machine
  - And not the most privileged
- Today, at least two layers sit underneath the OS/hypervisor
- System Management Mode (SMM) (ring -2)
  - Runs below the hypervisor/OS
- Intel Management Engine and Intel Innovation Engine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




Intel Xeon
MC1
MC2
P-Unit
IIO
Core
Core
Core
Core
Core
Core
Core
Core
Core
LAPIC
Core
Core
Core
Core
LLC



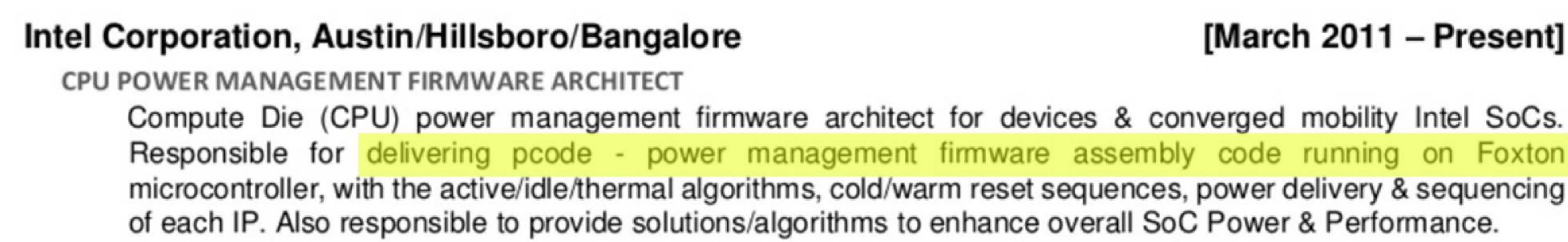
Mark Ermolov
@_markel___

Follow

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

```
<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="$(Stepping)" AddInstanceNameSuffix="true">
      <DeviceProperty Name="CoreGroup" Val="PSH"/>
    </TapInfo>
    <TapInfo TapName="ANN_ARC" NodeType="Thread" DeviceType="ARC" Stepping="$(Stepping)" AddInstanceNameSuffix="true">
      <DeviceProperty Name="CoreGroup" Val="ARC"/>
    </TapInfo>
    <TapInfo TapName="ANN_MC8051" NodeType="Thread" DeviceType="8051" Stepping="$(Stepping)" AddInstanceNameSuffix="true">
      <DeviceProperty Name="CoreGroup" Val="8051"/>
    </TapInfo>
    <TapInfo TapName="ANN_SJTAG|ANN_SEP" NodeType="Thread" DeviceType="APS" Stepping="$(Stepping)" AddInstanceNameSuffix="true">
      <DeviceProperty Name="CoreGroup" Val="APS"/>
    </TapInfo>
    <TapInfo TapName="ANN_AUDIO" NodeType="Thread" DeviceType="AUDIO" Stepping="$(Stepping)" AddInstanceNameSuffix="true">
      <DeviceProperty Name="CoreGroup" Val="APS"/>
    </TapInfo>
    <TapInfo TapName="SLM_MODULE0" NodeType="Box" DeviceType="SLM_MODULE" Stepping="E0" InstanceId="0" AddInstanceNameSuffix="false">
      <TapSelect Name="SLM_C0" DeviceType="SLM" NodeType="Core" SelectProc="slm.tap.select_core(0)" InstanceId="0" DeviceSubType="">
        <TapSelect Name="SLM_C0_T0" DeviceType="SLM" NodeType="Thread" InstanceId="0" DeviceSubType=""/>
      </TapSelect>
      <TapSelect Name="SLM_C1" DeviceType="SLM" NodeType="Core" SelectProc="slm.tap.select_core(1)" InstanceId="1" DeviceSubType="">
        <TapSelect Name="SLM_C1_T0" DeviceType="SLM" NodeType="Thread" InstanceId="0" DeviceSubType=""/>
      </TapSelect>
    </TapInfo>
    <TapInfo TapName="SLM_MODULE1" NodeType="Box" DeviceType="SLM_MODULE" Stepping="E0" InstanceId="1" AddInstanceNameSuffix="false">
      <TapSelect Name="SLM_C2" DeviceType="SLM" NodeType="Core" SelectProc="slm.tap.select_core(0)" InstanceId="0" DeviceSubType="">
        <TapSelect Name="SLM_C2_T0" DeviceType="SLM" NodeType="Thread" InstanceId="0" DeviceSubType=""/>
      </TapSelect>
      <TapSelect Name="SLM_C3" DeviceType="SLM" NodeType="Core" SelectProc="slm.tap.select_core(1)" InstanceId="1" DeviceSubType="">
        <TapSelect Name="SLM_C3_T0" DeviceType="SLM" NodeType="Thread" InstanceId="0" DeviceSubType=""/>
      </TapSelect>
    </TapInfo>
    <TapInfo TapName=".*" NodeType="Chipset" Stepping="$(Stepping)" InstanceId="0" AddInstanceNameSuffix="true"/>
  </ProductDescription>
</ProductInfo>
```

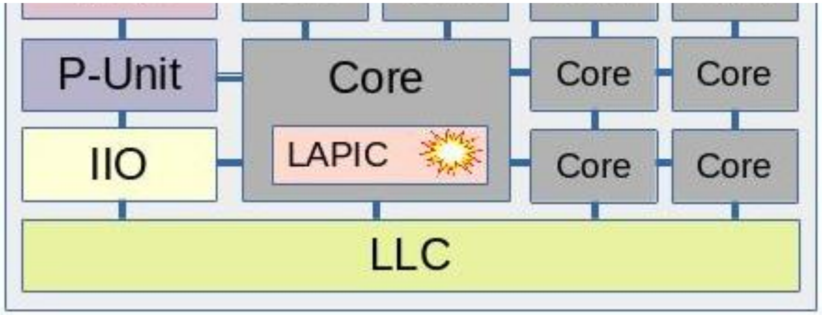
What is running on these microcontrollers?



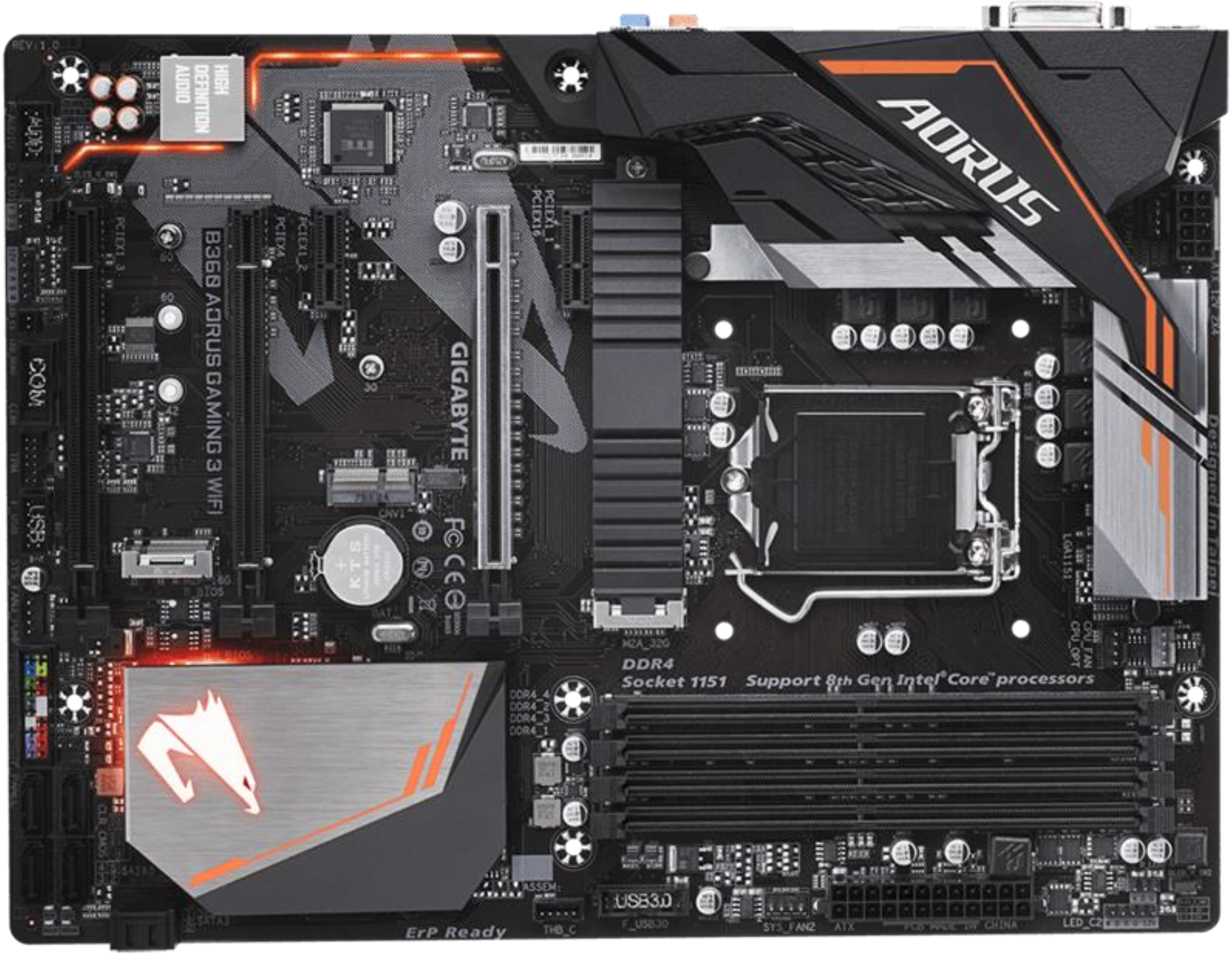
Intel Corporation, Austin/Hillsboro/Bangalore **[March 2011 – Present]**

CPU POWER MANAGEMENT FIRMWARE ARCHITECT

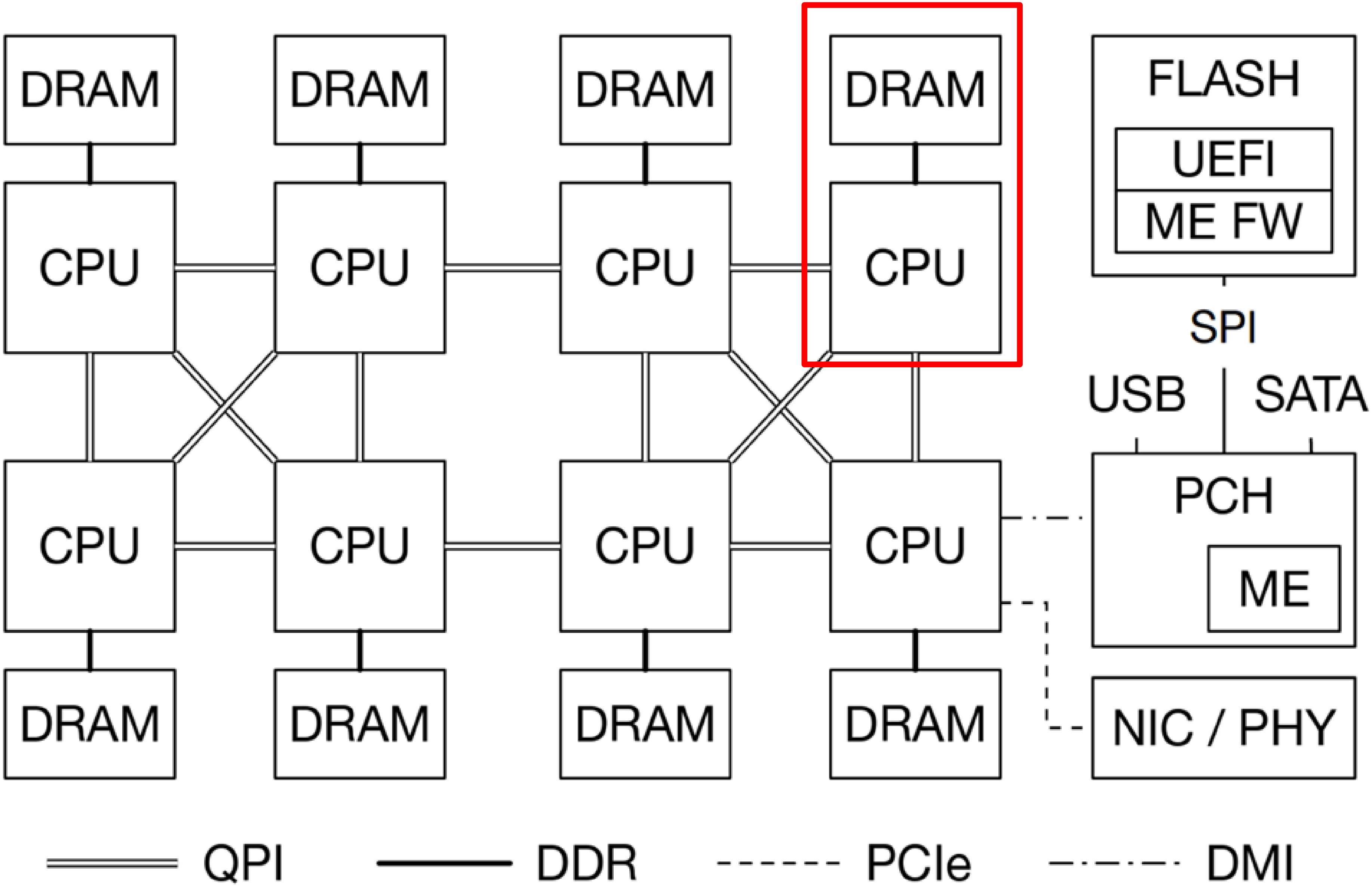
Compute Die (CPU) power management firmware architect for devices & converged mobility Intel SoCs. Responsible for delivering pcode - power management firmware assembly code running on Foxton microcontroller, with the active/idle/thermal algorithms, cold/warm reset sequences, power delivery & sequencing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



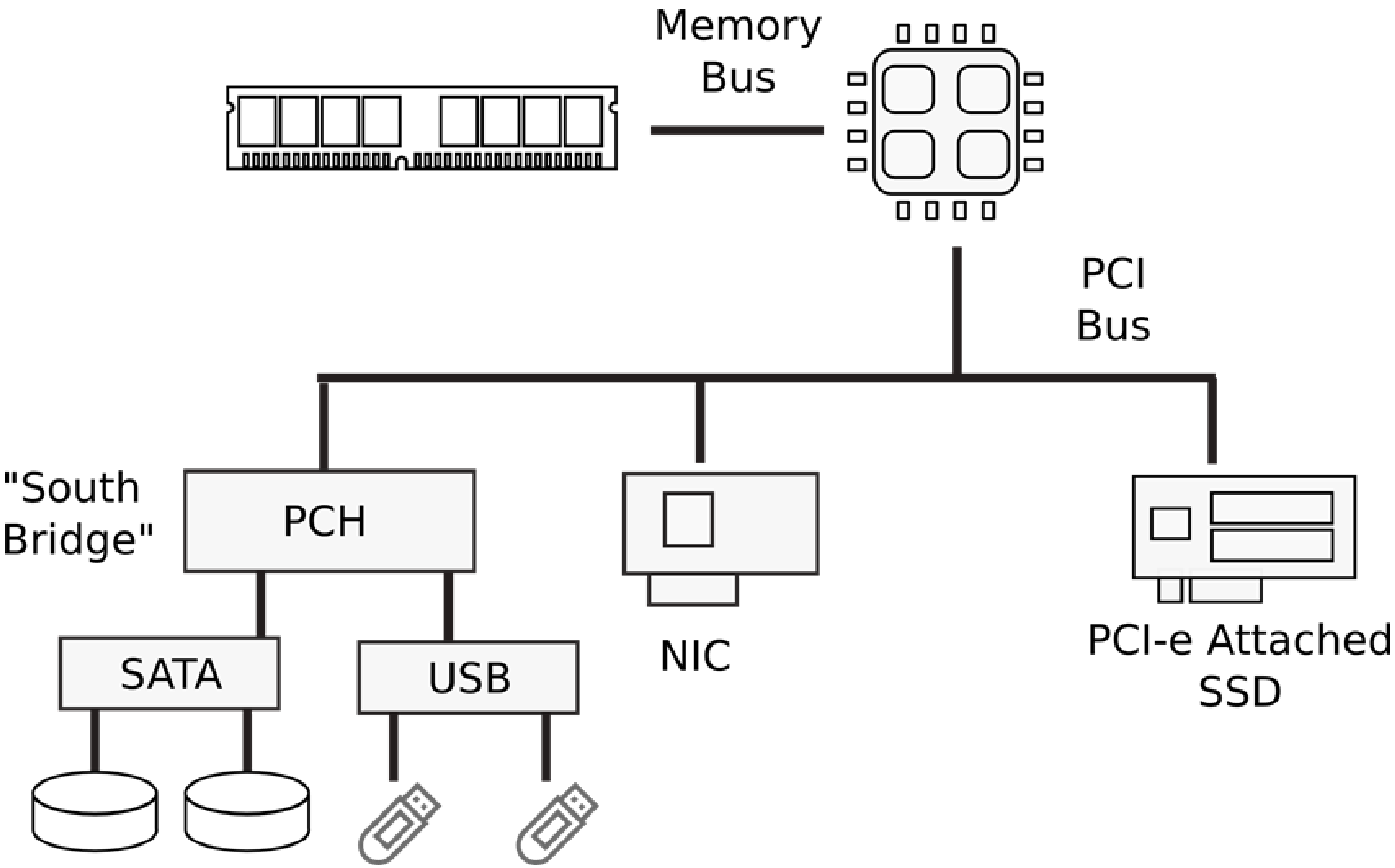
B360 AORUS Motherboard



PC motherboard components



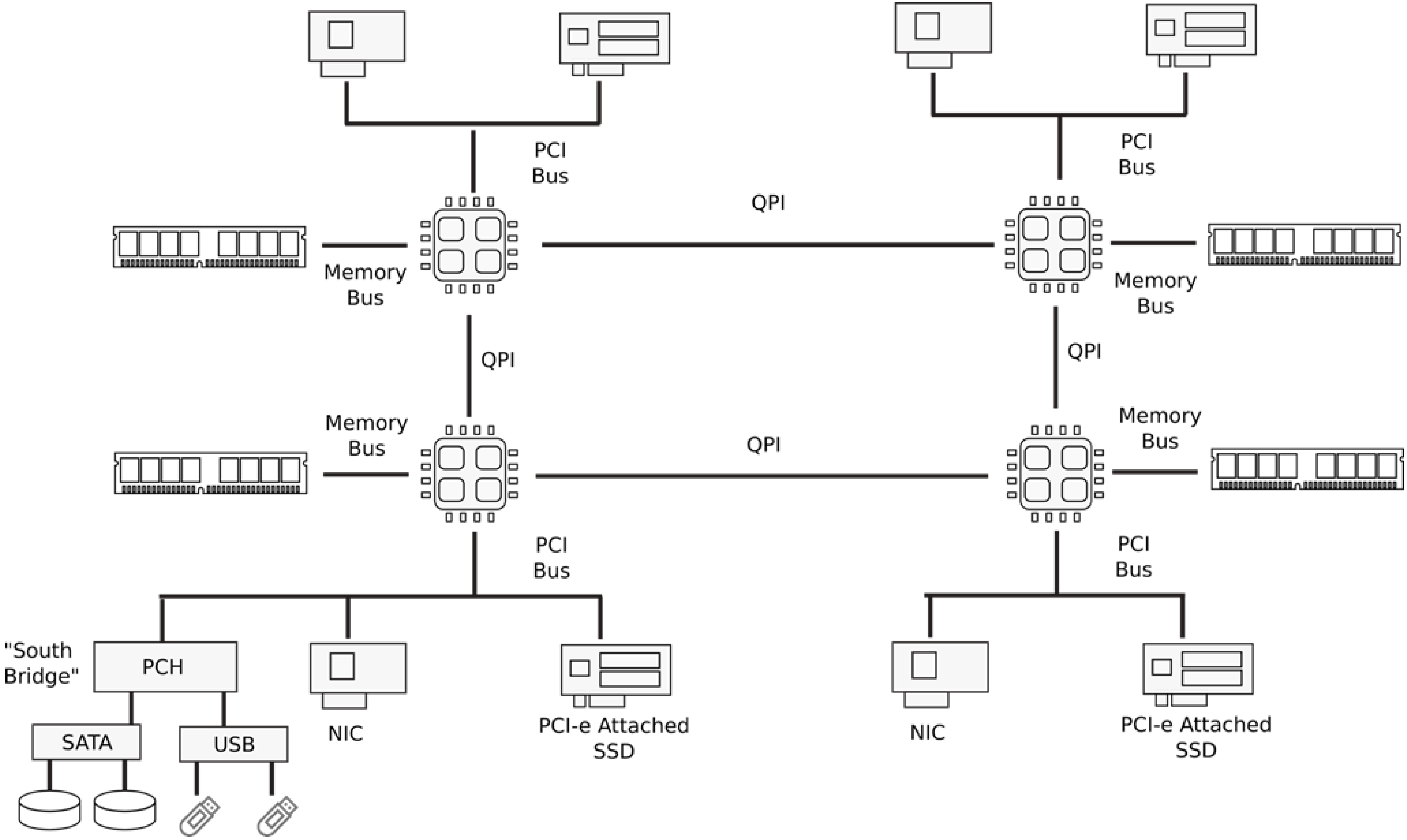
I/O Devices


Memory
Bus
PCI
Bus
"South
Bridge"
PCH
NIC
PCI-e Attached
SSD
SATA
USB

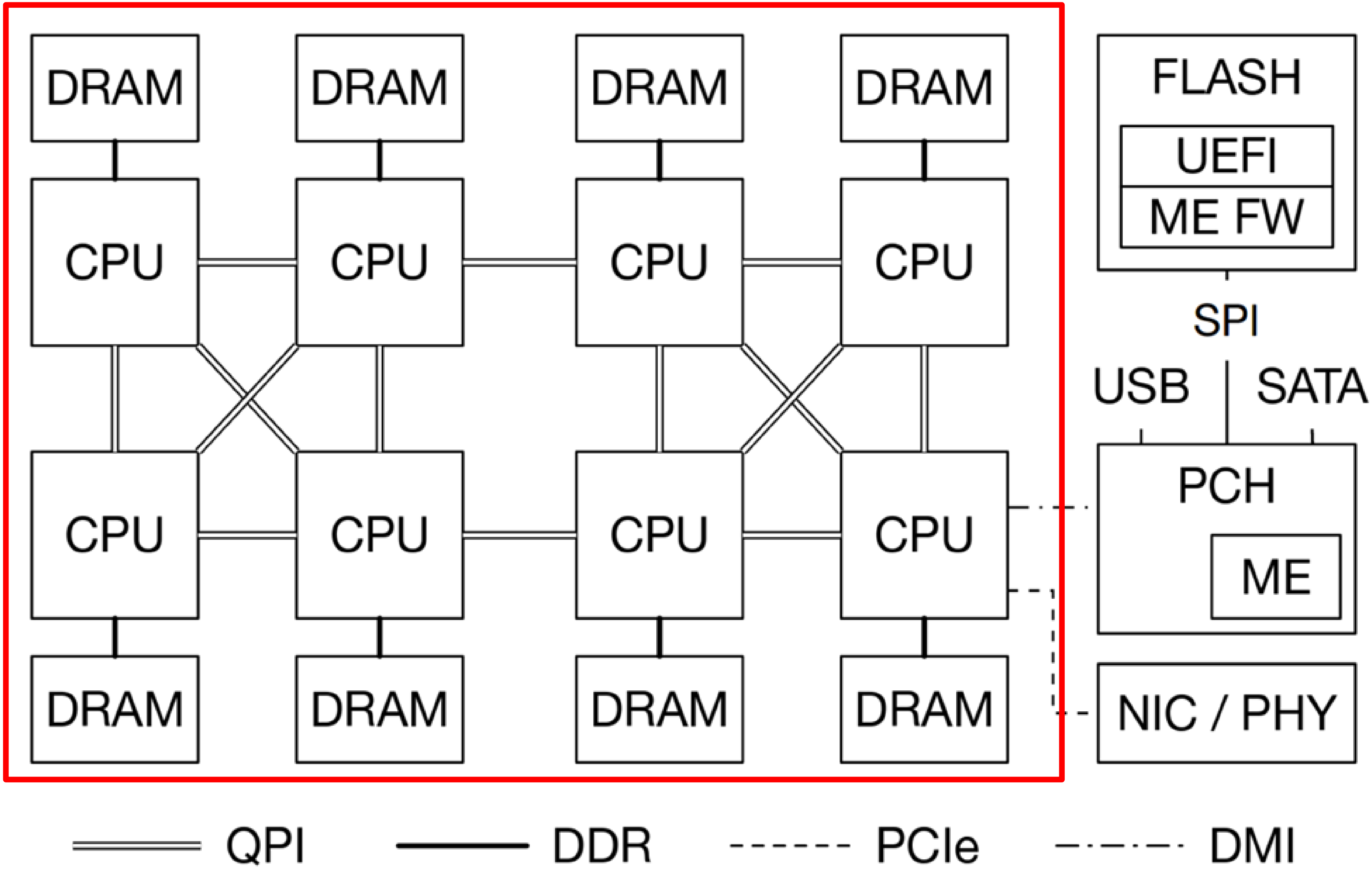
Dell R830 4-socket server

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

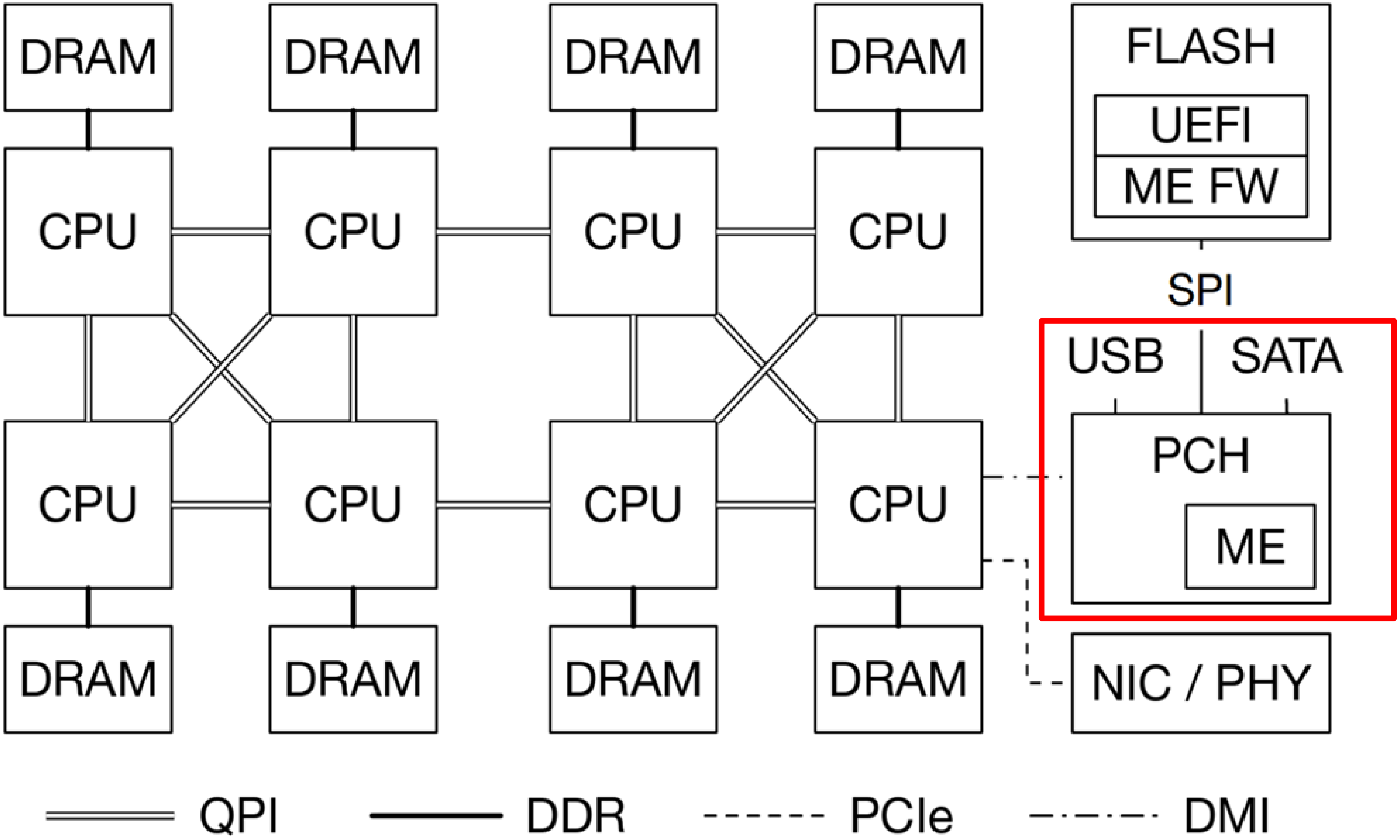
Multi-socket machines



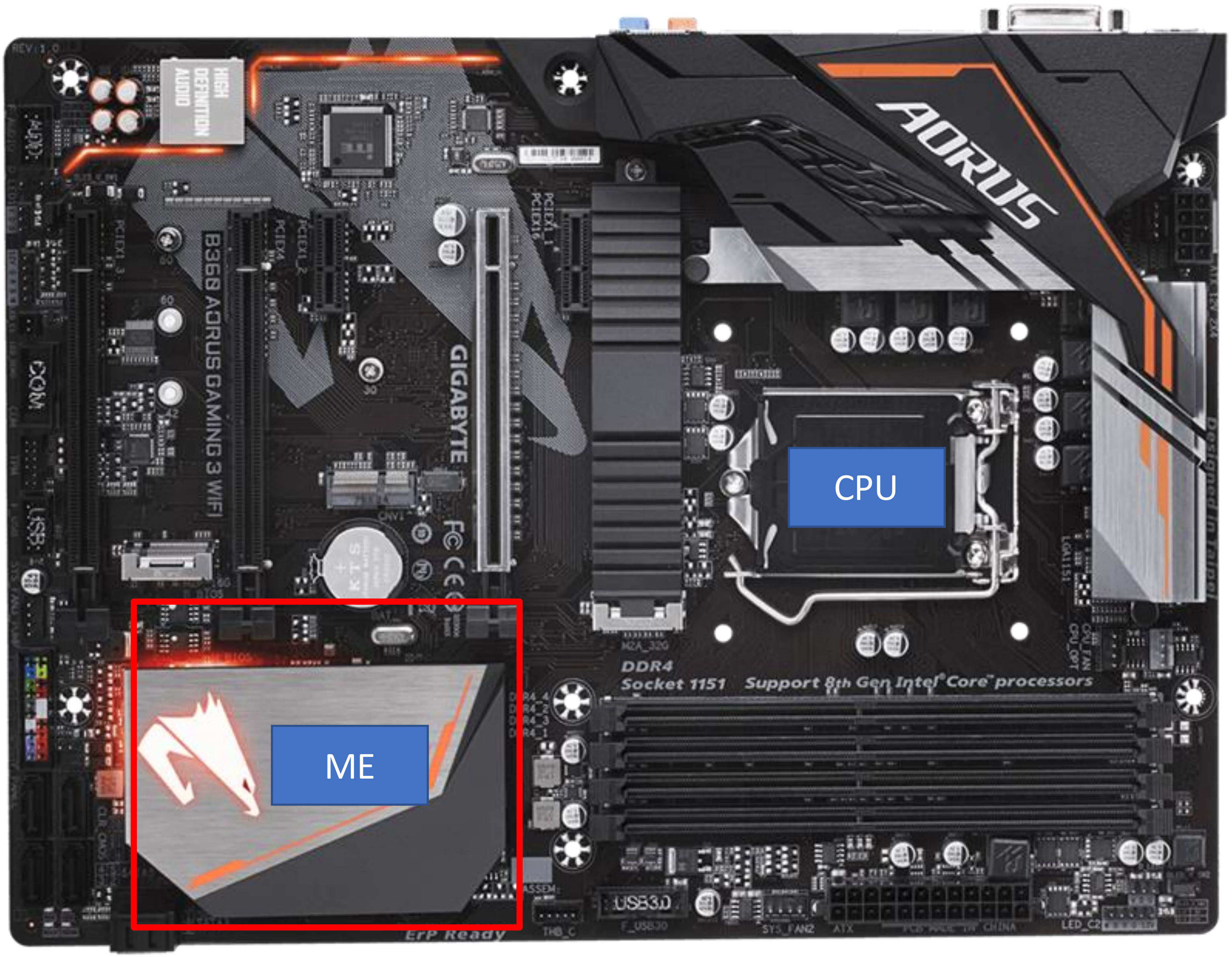
PC motherboard components



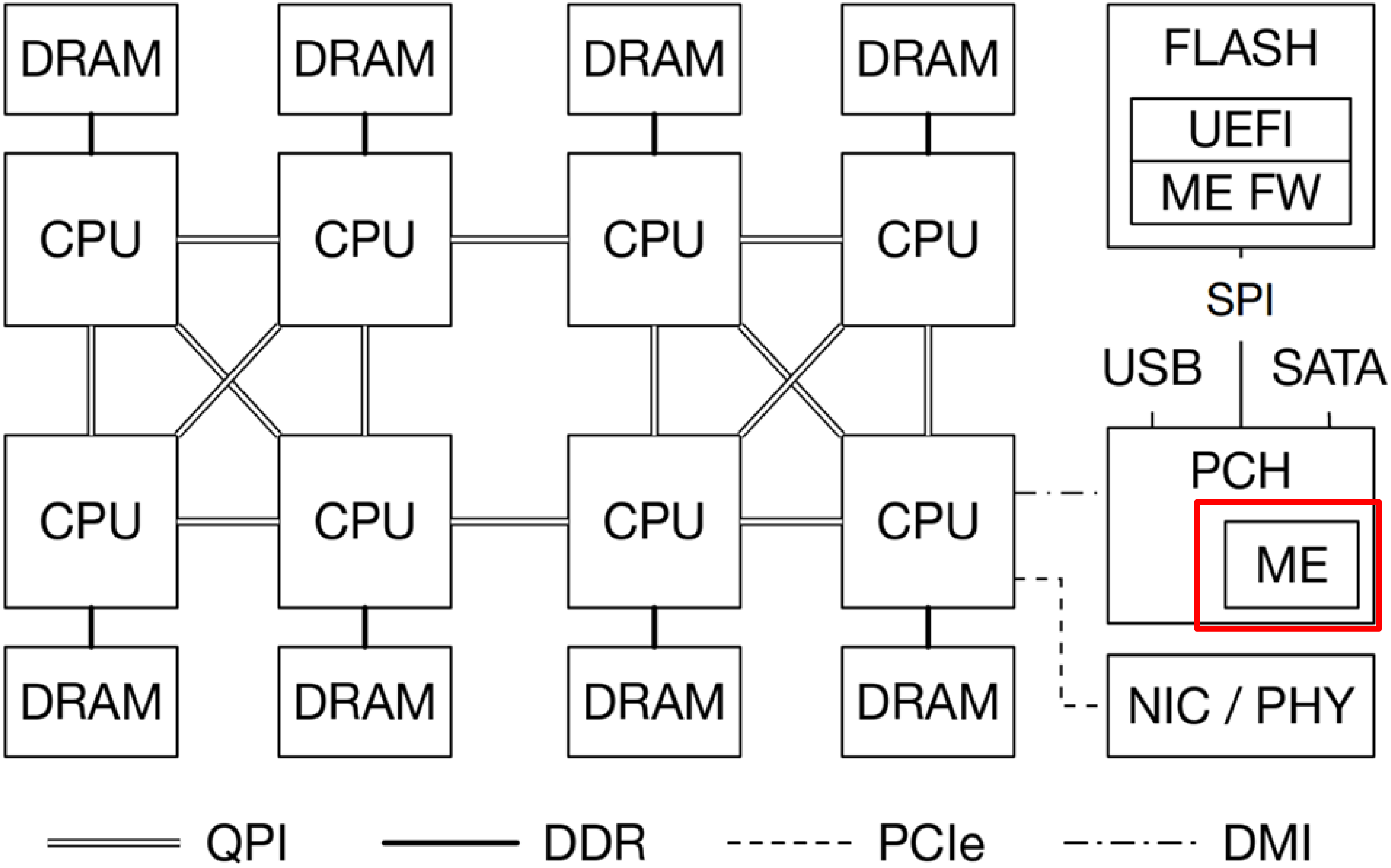
PCH – Platform Controller Hub



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 it’s 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

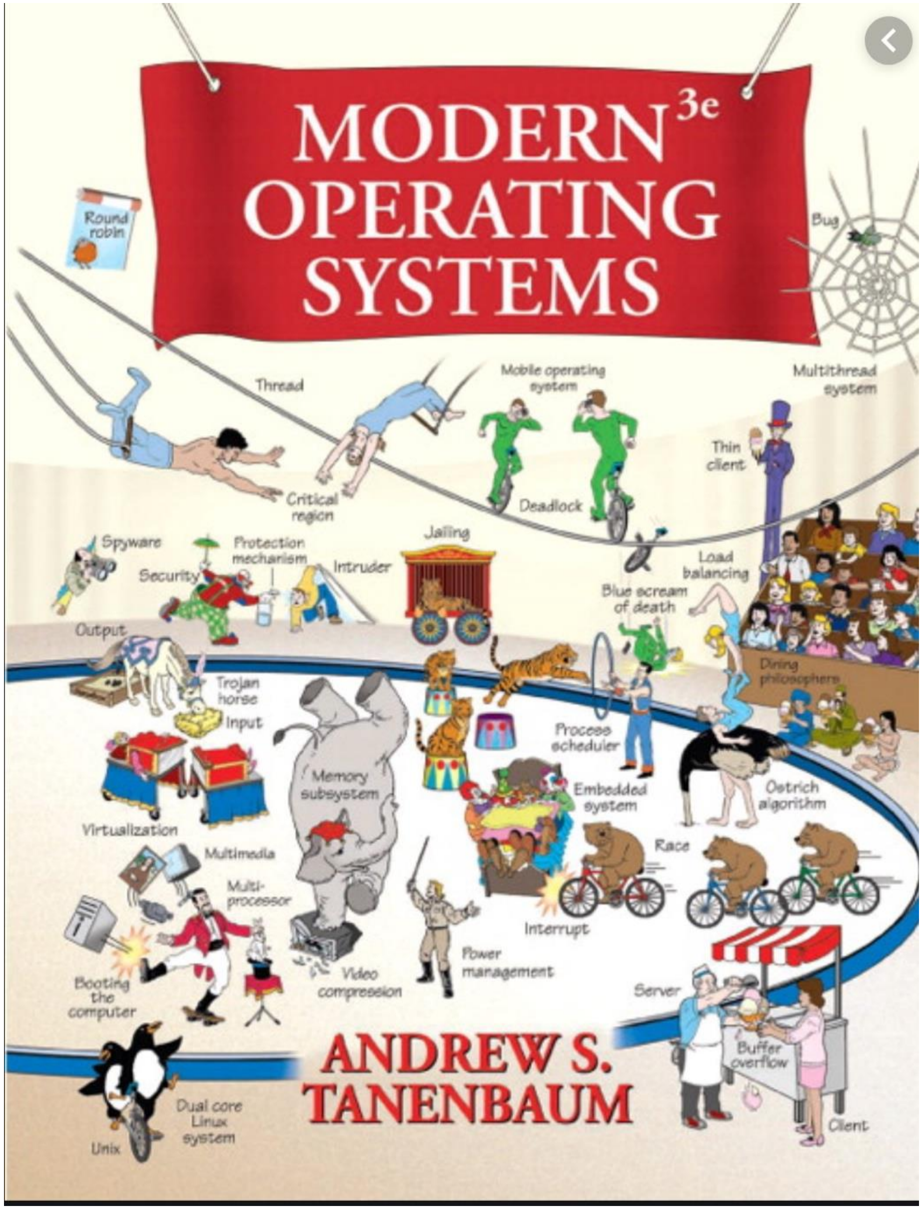
ME: Theft prevention use-case

- 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)

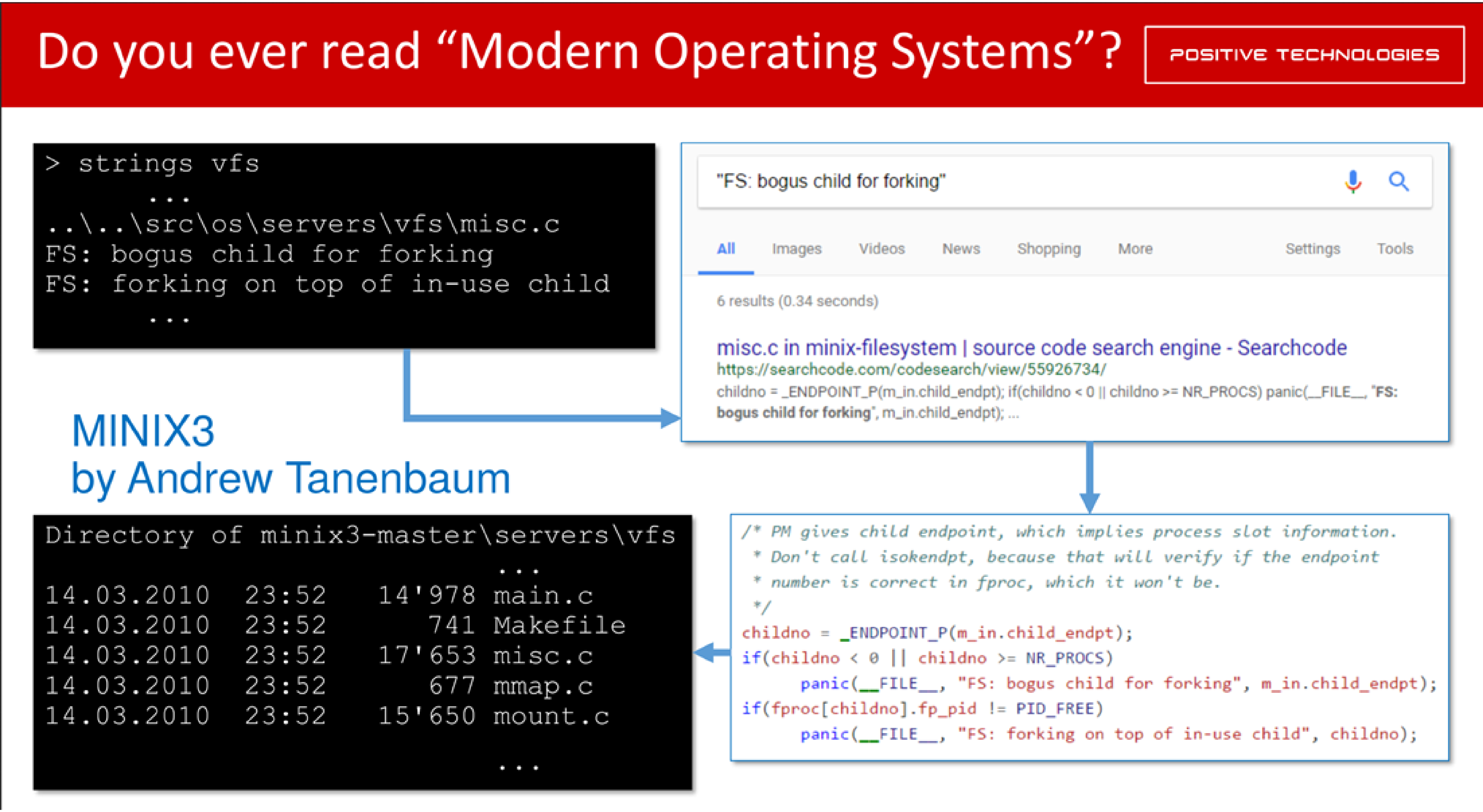
- All modern motherboard chips contain ME
- Part of Active Management Technology (AMT)
- Convenient way for administrators to fix your machine remotely
    - Obviously, a huge opportunity for an attack

What’s running there?



Have you read “Modern Operating Systems?”

What’s running there?



Do you ever read “Modern Operating Systems”?

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

"FS: bogus child for forking"

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); ...

MINIX3
by Andrew Tanenbaum

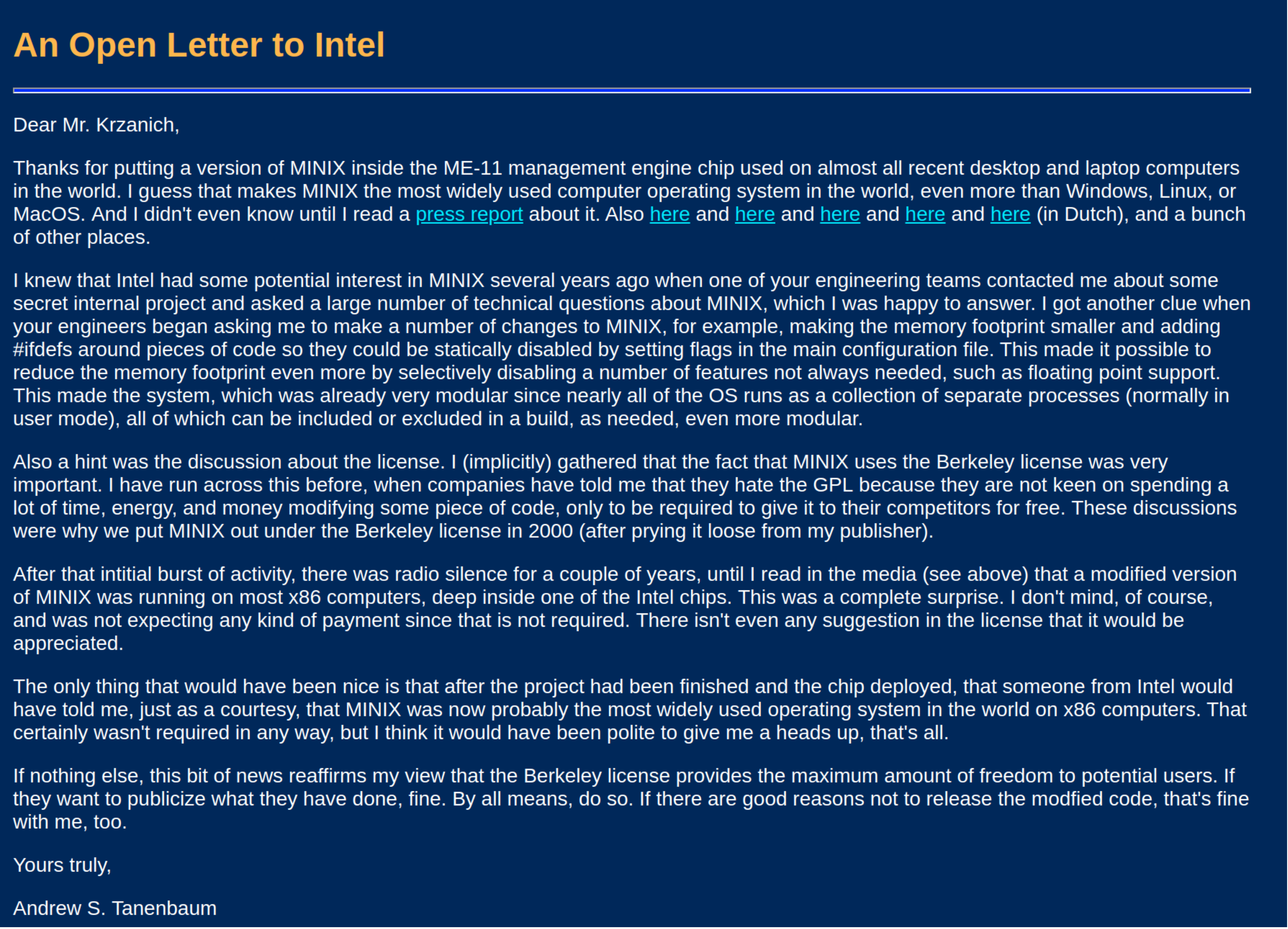
```
/* PM gives child endpoint, which implies process slot information.
 * Don't call isokendpt, because that will verify if the endpoint
 * number is correct in fproc, which it won't be.
 */
childno = _ENDPOINT_P(m_in.child_endpt);
if(childno < 0 || childno >= NR_PROCS)
      panic(__FILE__, "FS: bogus 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);
```

```
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
                              ...
```

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.

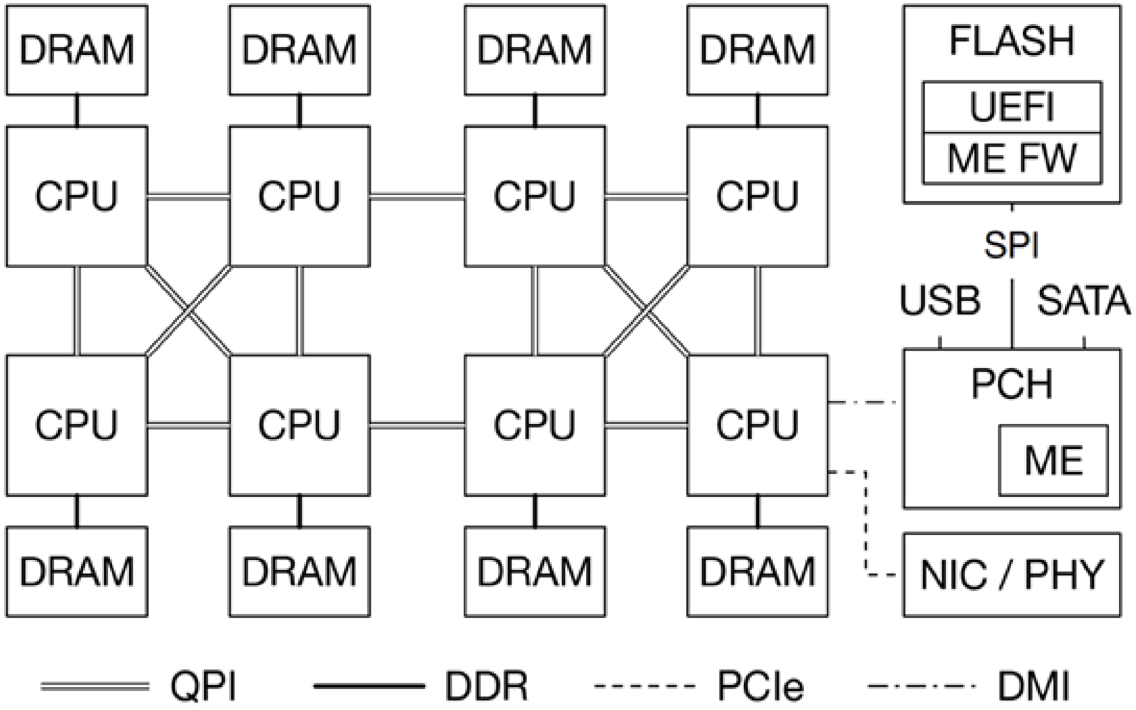
If nothing else, this bit of news reaffirms my view that the Berkeley license provides the maximum amount of freedom to potential users. If they want to publicize what they have done, fine. By all means, do so. If there are good reasons not to release the modfied code, that's fine with me, too.

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
    - Initally, 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

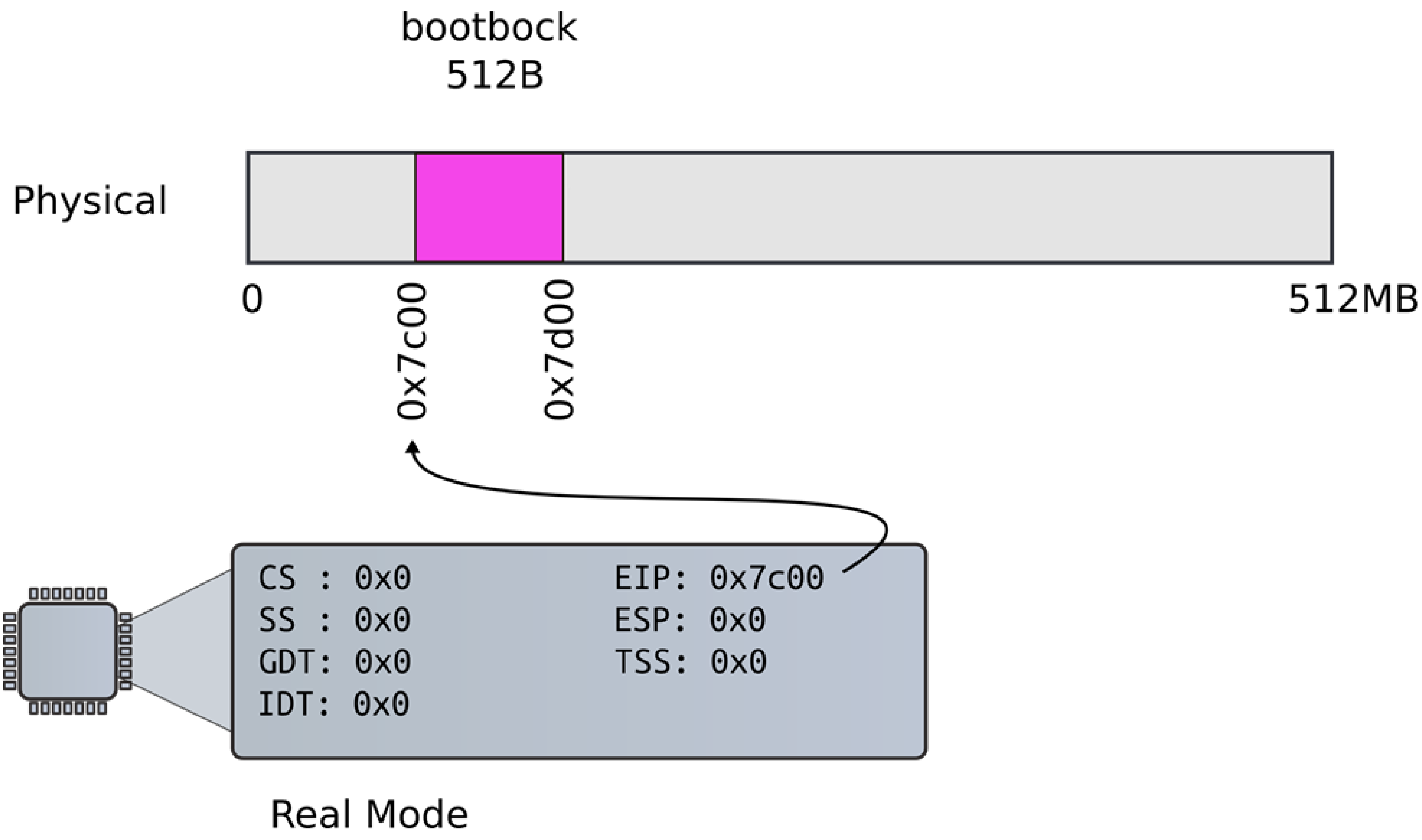
System Management Mode

- 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



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

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

```
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
```

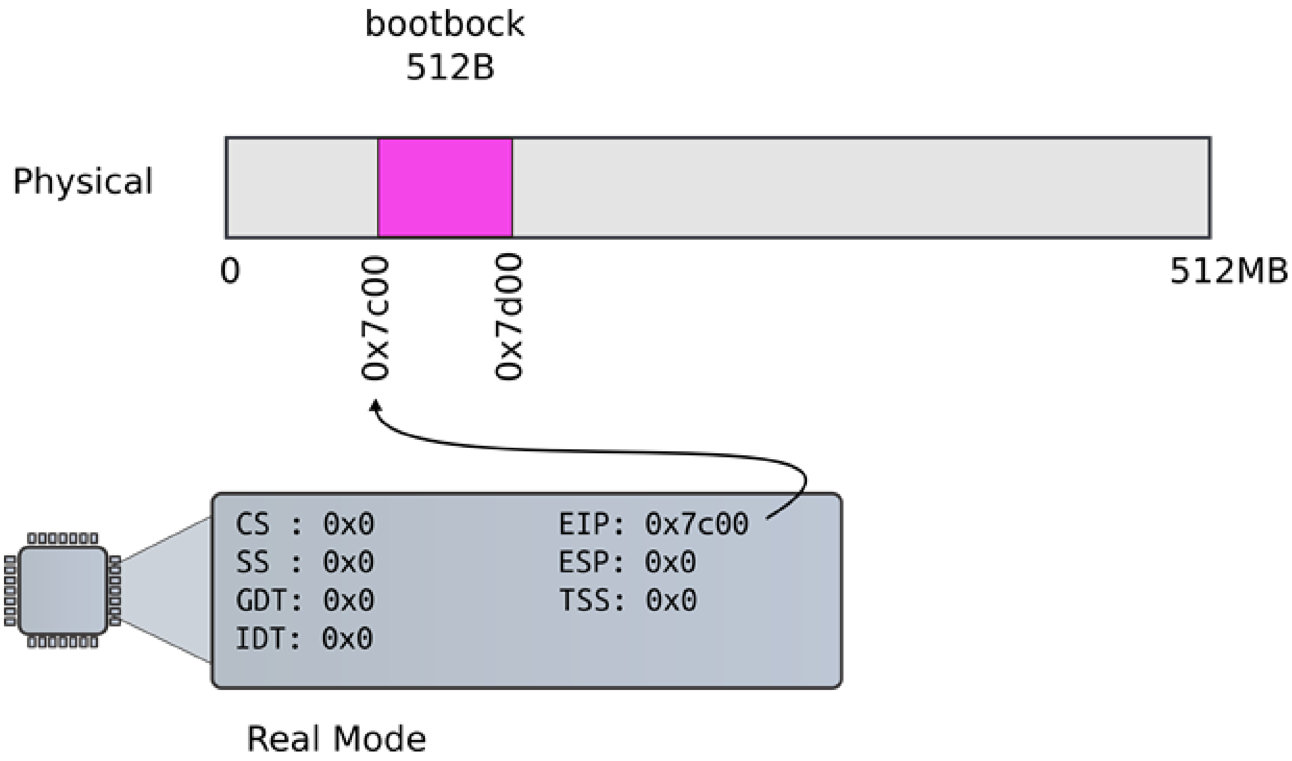
xv6/bootasm.S [bootloader]

Why start happens to be 0x7c00?

9111 start:

9112 cli # BIOS enabled interrupts; disable

9113



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

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

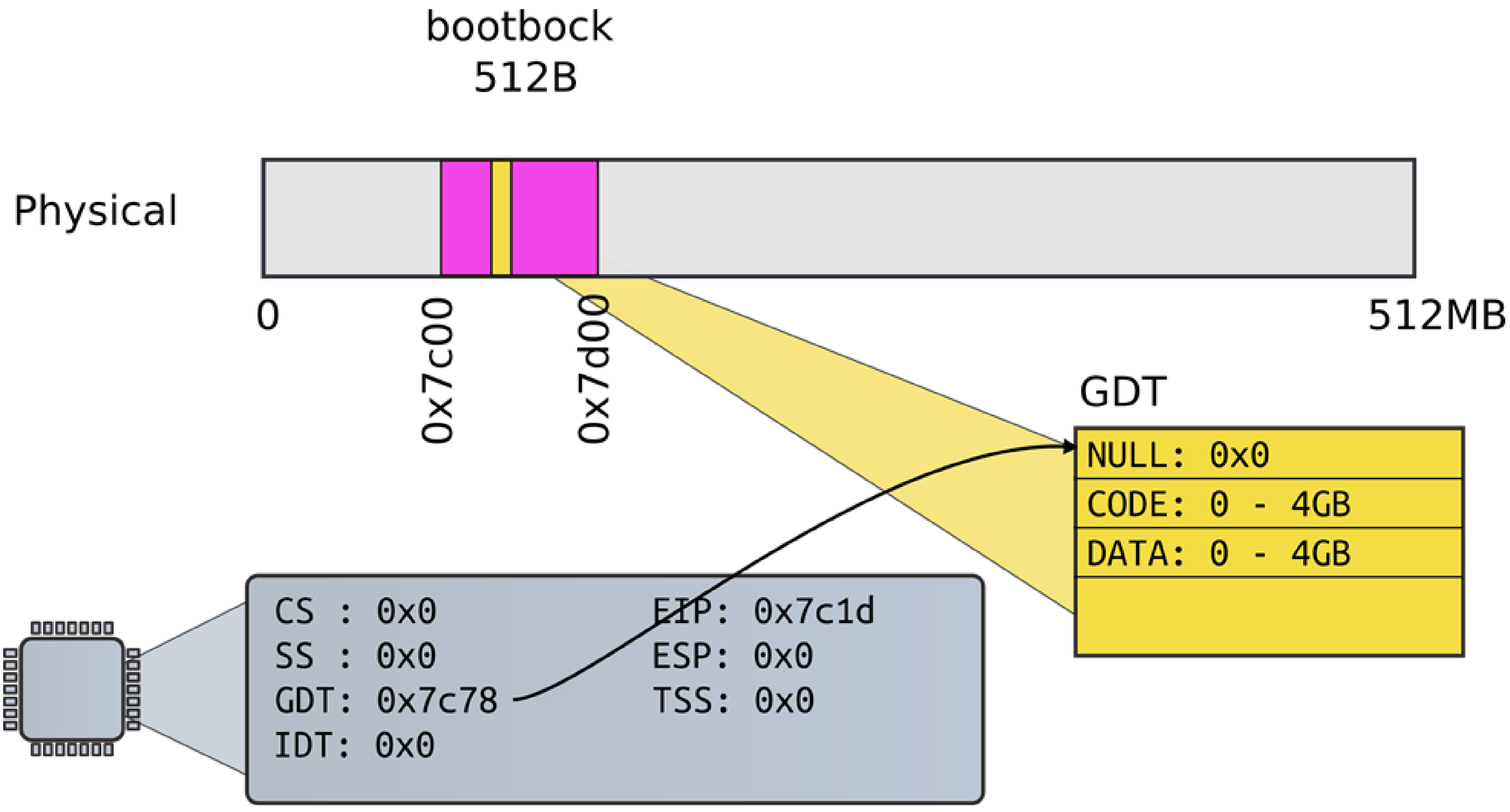
Switch to protected mode

- 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.

```
lgdt gdtdesc

movl %cr0, %eax

orl $CR0_PE, %eax

movl %eax, %cr0
```

xv6/bootasm.S [bootloader]

Load GDT

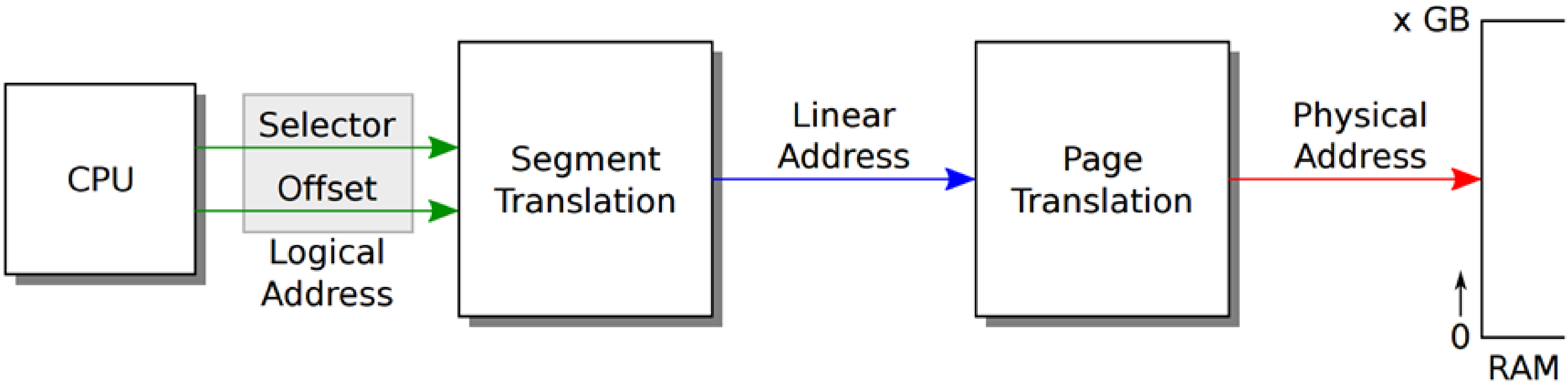


NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB

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

Real Mode

Recap: complete address translation



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
```

xv6/bootasm.S [bootloader]

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
```

xv6/bootasm.S [bootloader]

Actual switch

- Use long jump to change code segment

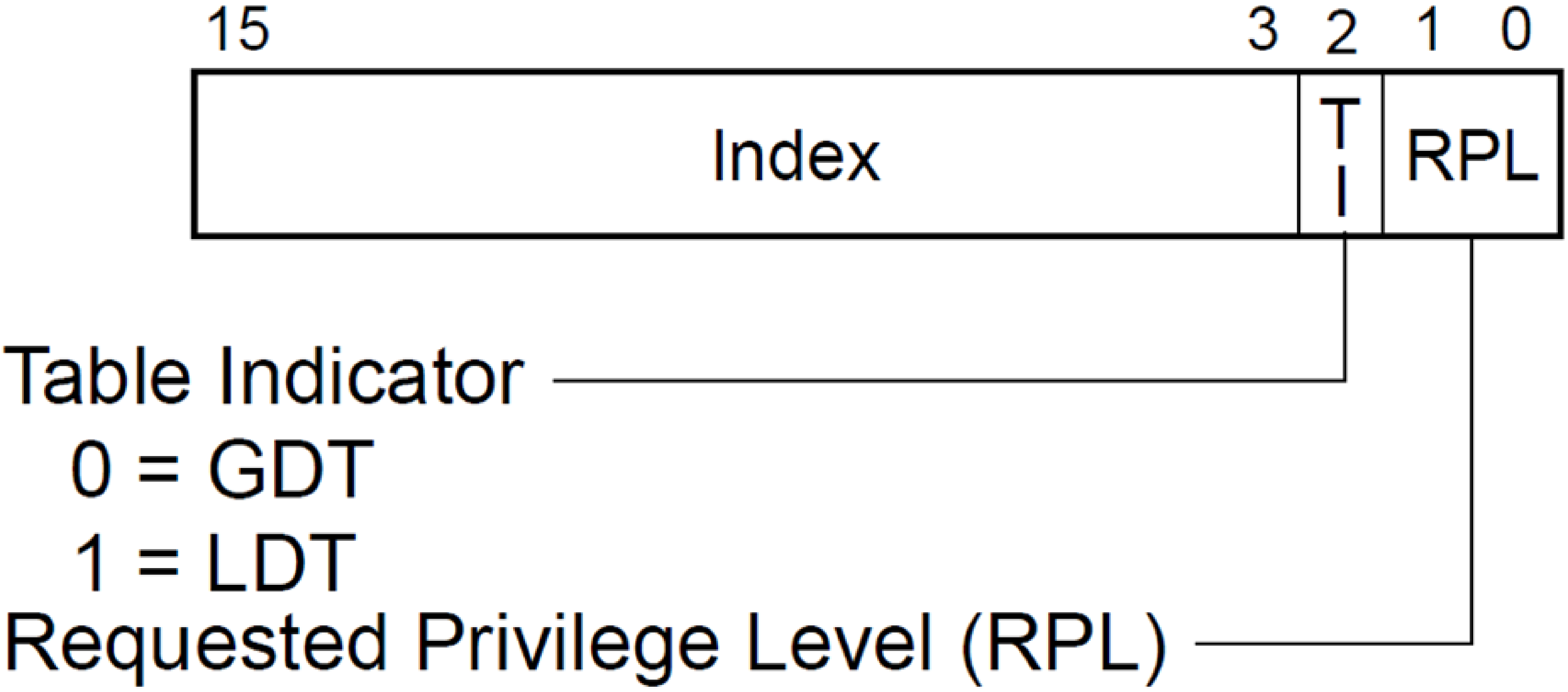
```
ljmp $(SEG_KCODE<<3), $start32
```

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

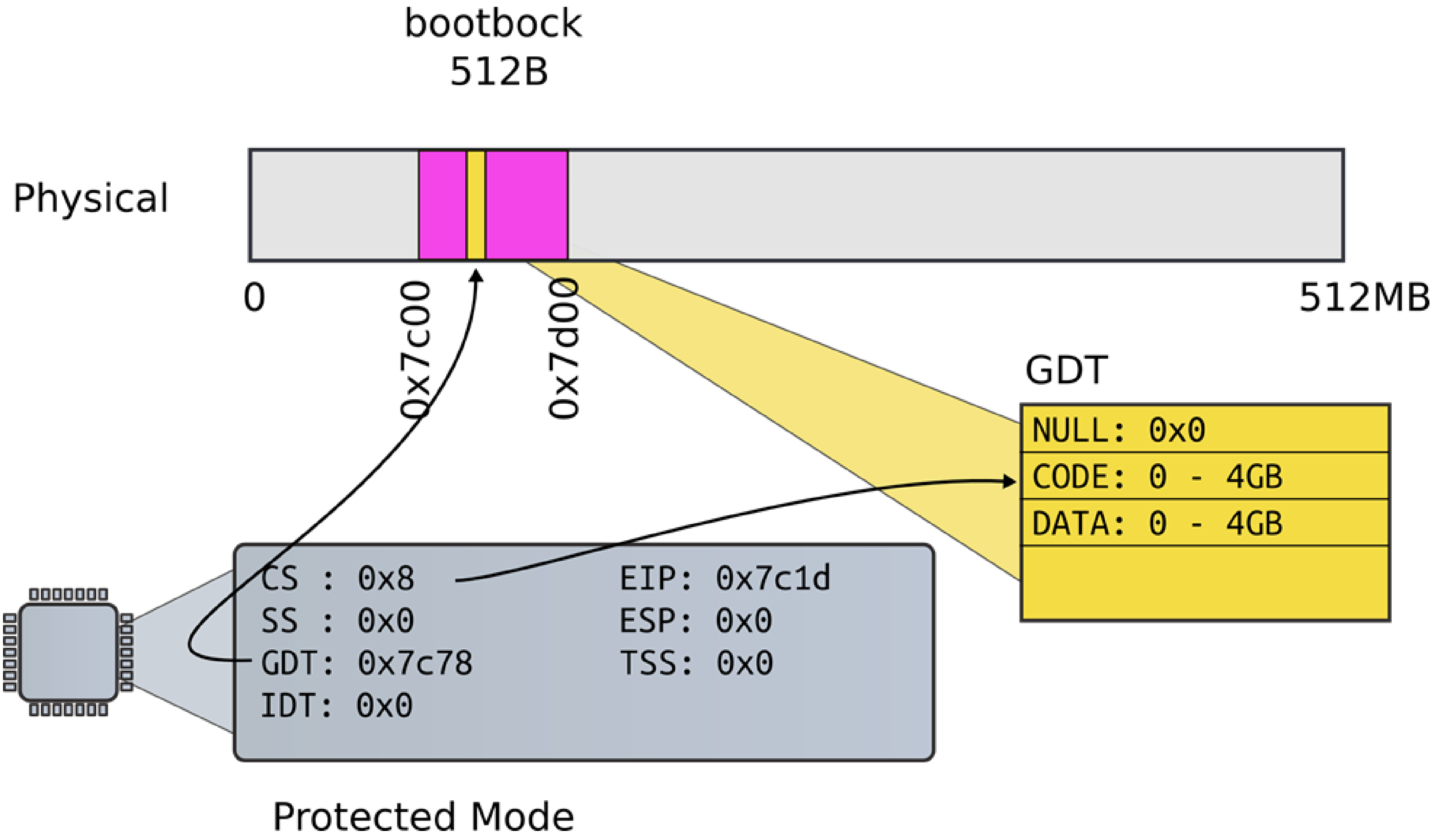
xv6/bootasm.S [bootloader]

Why CS is 0x8, not 0x1?

- Segment selector:



Long jump

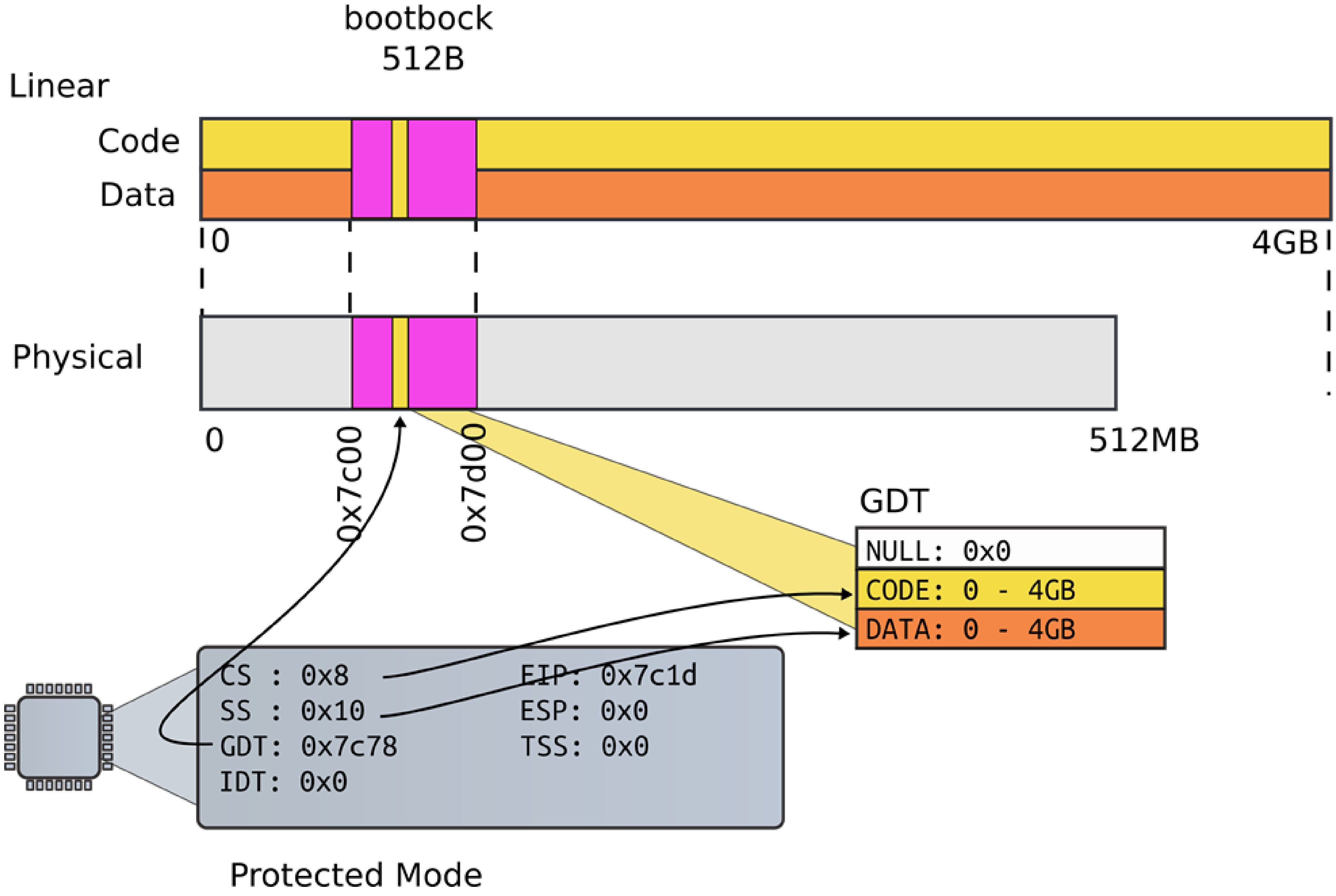

bootbock
512B
Physical
0
0x7c00
0x7d00
512MB
GDT
NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
CS : 0x8
SS : 0x0
GDT: 0x7c78
IDT: 0x0
EIP: 0x7c1d
ESP: 0x0
TSS: 0x0
Protected Mode

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
```

xv6/bootasm.S [bootloader]

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

- Why do we need a stack?

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

xv6/bootasm.S [bootloader]

Setup stack

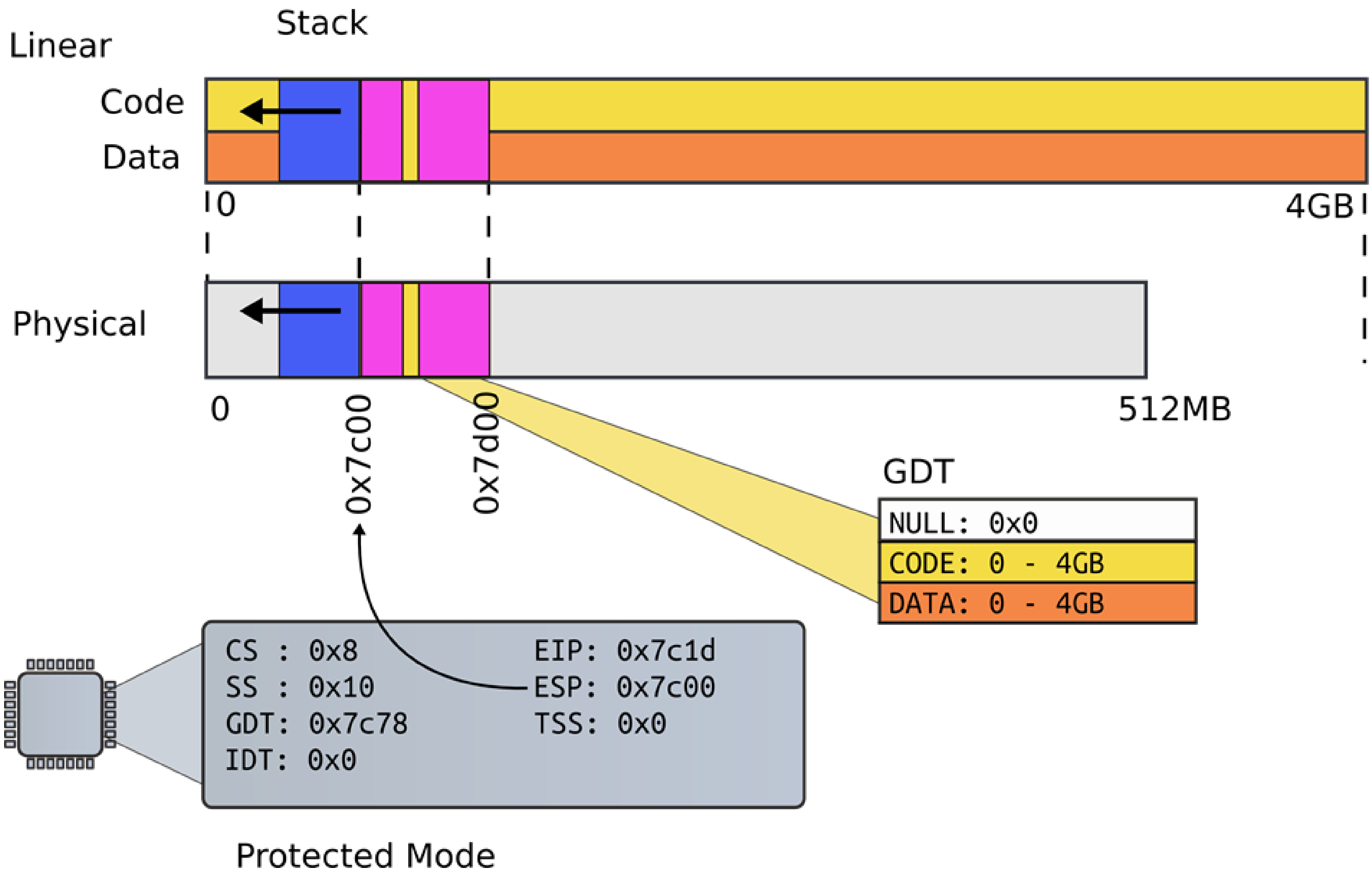
- Need stack to use C
- Function invocations
- Note, there were no stack instructions before that

7c00

```
9166 movl $start, %esp

9167 call bootmain
```

xv6/bootasm.S [bootloader]

First stack



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

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

Invoke first C function

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

xv6/bootasm.S [bootloader]

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 [bootloader]

```

  // 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

xv6/bootmain.c [bootloader]

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);
}
```

xv6/bootmain.c [bootloader]

How do we read disk (cont)?

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

```

xv6/bootmain.c [bootloader]

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);
}
```

xv6/bootmain.c [bootloader]

```

  // 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();
}
```

Call kernel entry

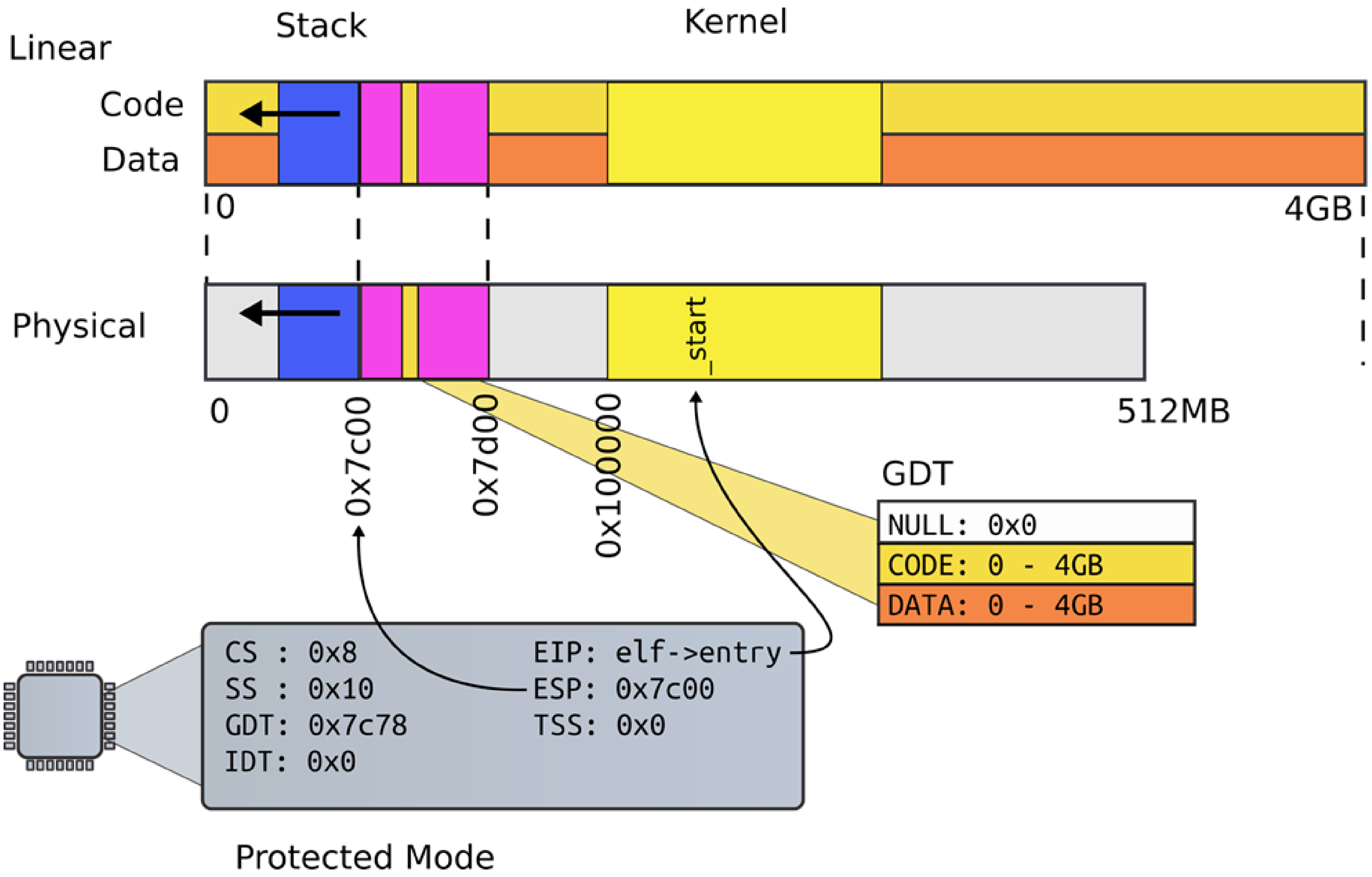
xv6/bootmain.c [bootloader]

```
.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:
  # Enable physical-address extensions (PAE).
  movl %cr4, %eax
  orl $(CR4_PAE), %eax
  movl %eax, %cr4
```

entry(): kernel ELF entry

xv6/entry.S [kernel]

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

Protected Mode

entry(): kernel ELF entry

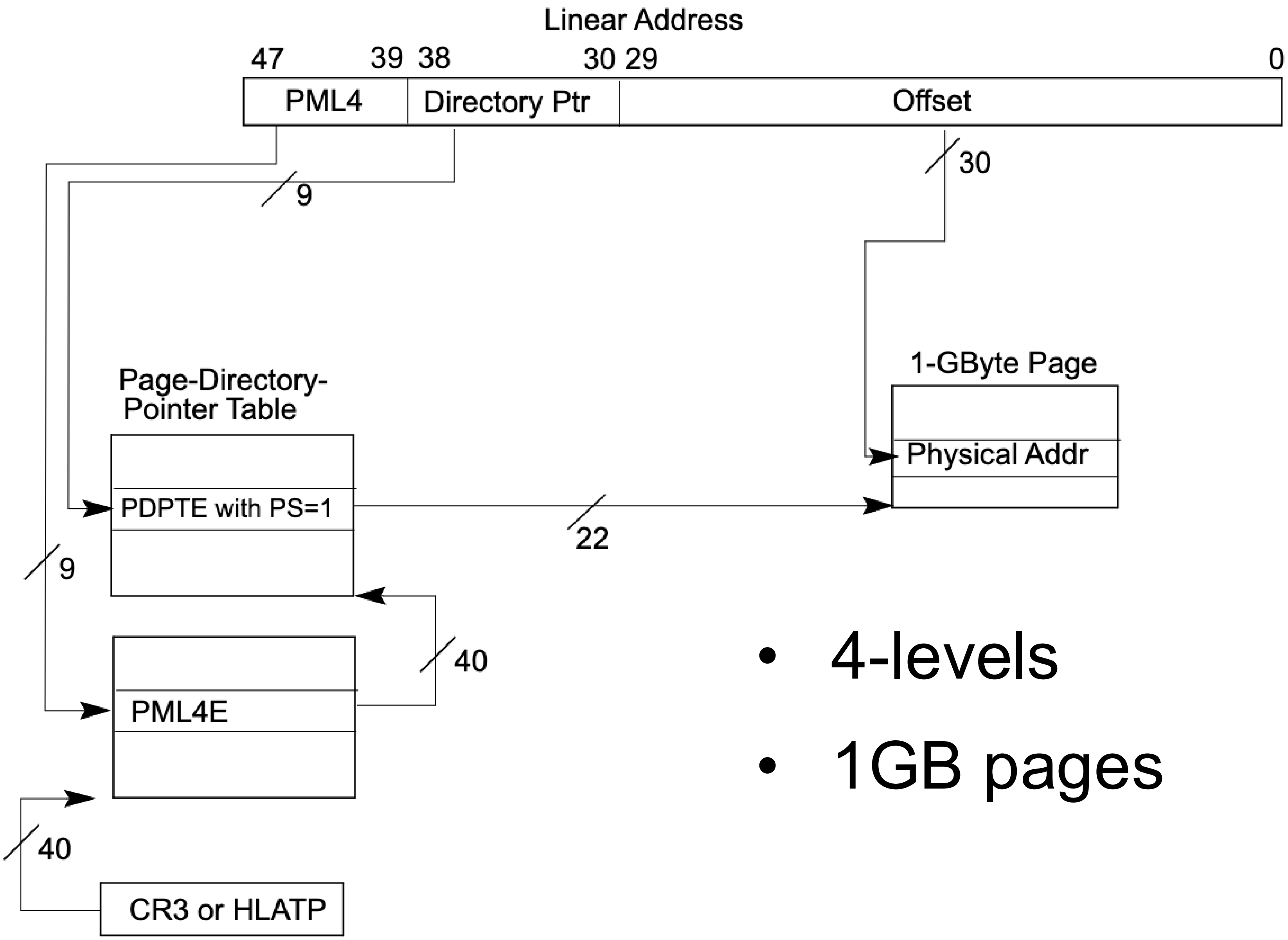
xv6/entry.S [kernel]

```
.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:
  # Enable physical-address extensions (PAE).
  movl %cr4, %eax
  orl $(CR4_PAE), %eax
  movl %eax, %cr4
```

64bit x86 supports three page sizes

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

Page translation for 1GB pages



- 4-levels
- 1GB pages

Set up page directory

```
# Load CR3 with the physical address of PML4.
movl $(V2P_WO(entrypml4)), %eax
movl %eax, %cr3
```

xv6/entry.S [kernel]

Our goal: a split address space

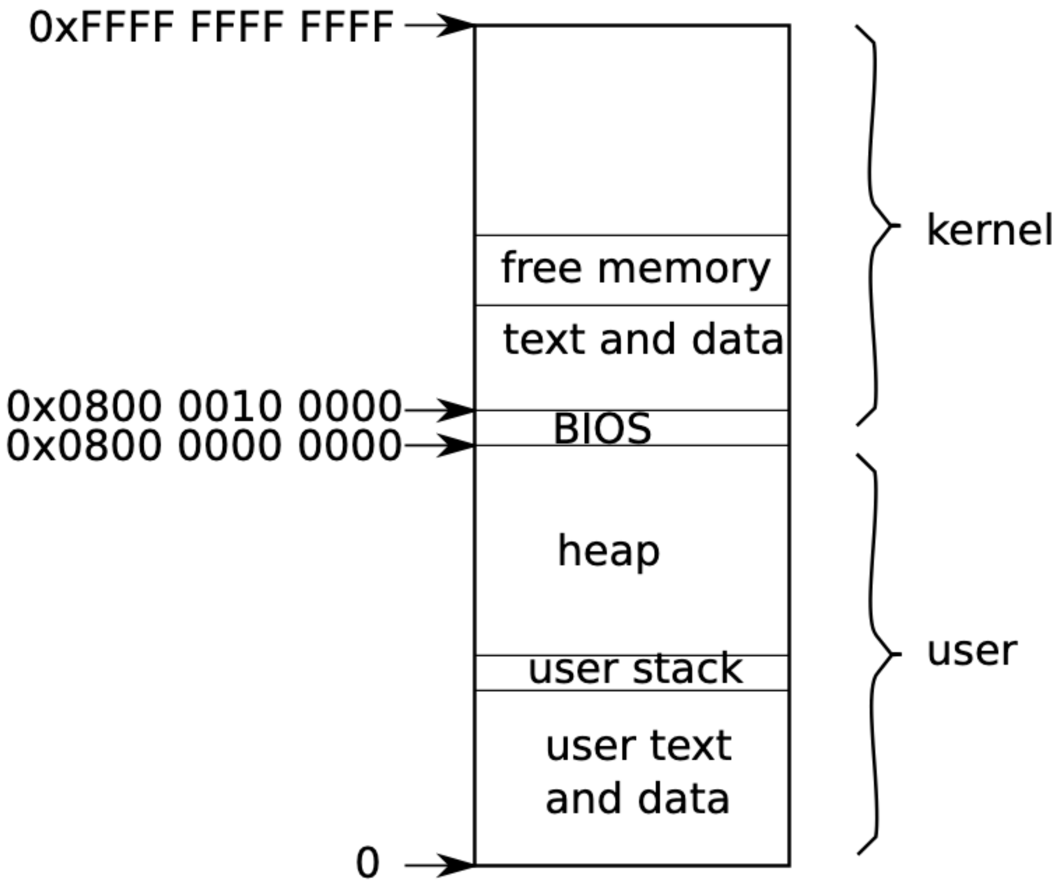
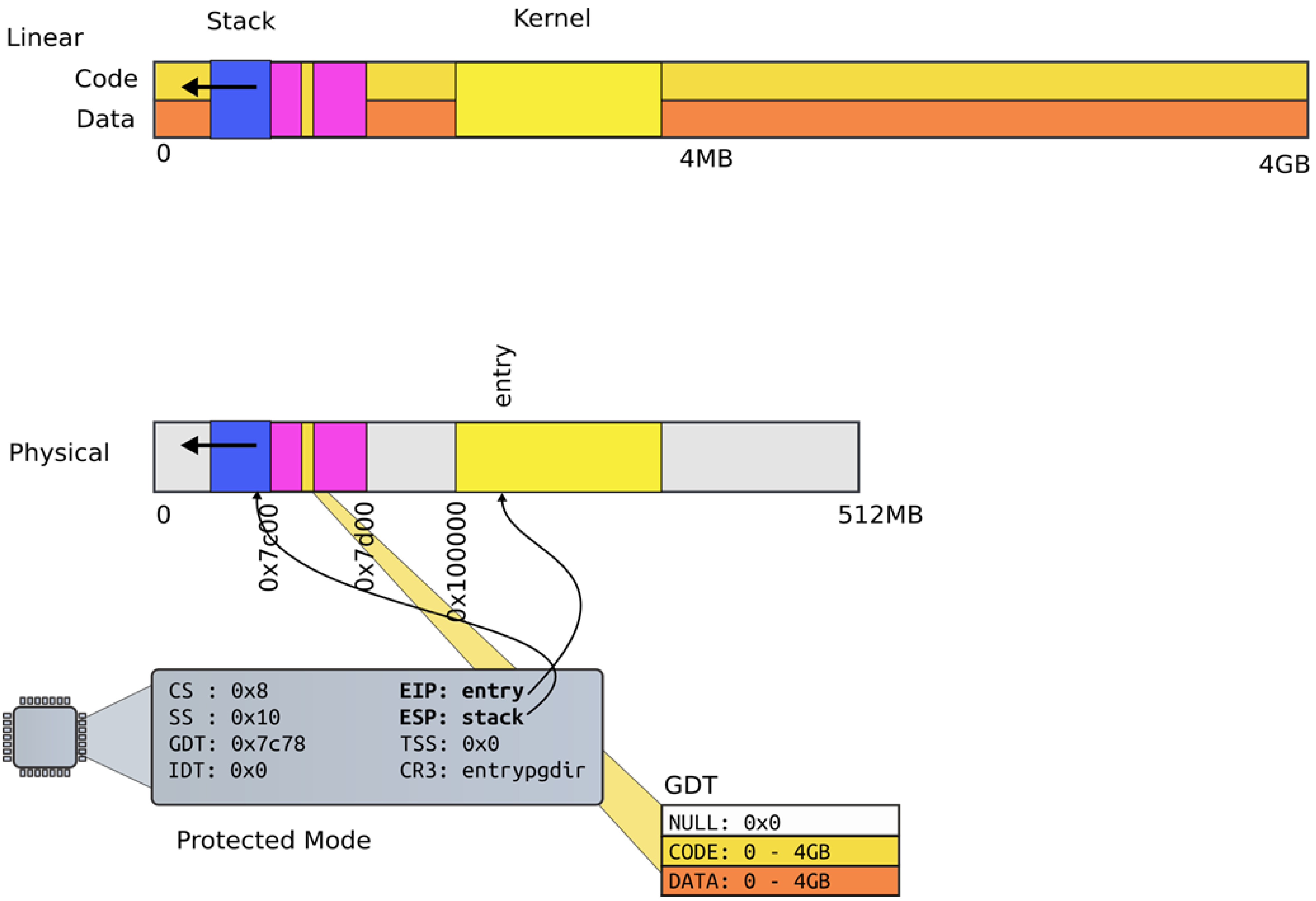


Figure 1-2. Layout of a virtual address space


Linear
Stack
Kernel
Code
Data
0
4MB
4GB
entry
Physical
0
0x7c00
0x7d00
0x100000
512MB
CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0
EIP: entry
ESP: stack
TSS: 0x0
CR3: entrypgdir
Protected Mode
GDT
NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB

Boot-time page table

- Two 1GB entries (large pages)
- Entry #0
  - 0x0 – 1GB (virt) → 0x0 - 1GB (phys)
- Entry #256
  - 0x8TB – 8TB + 1GB (virt) → 0x0 - 1GB (phys)

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

First page table

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

First page table

First page table (cont)

```
// Page directory and page table constants.
#define NPML4ENTRIES 512 // # page map level 4 entries
```

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

First page table

KERNBASE 8TB

```
#define KERNBASE 0x80000000000      // First kernel virtual address
```

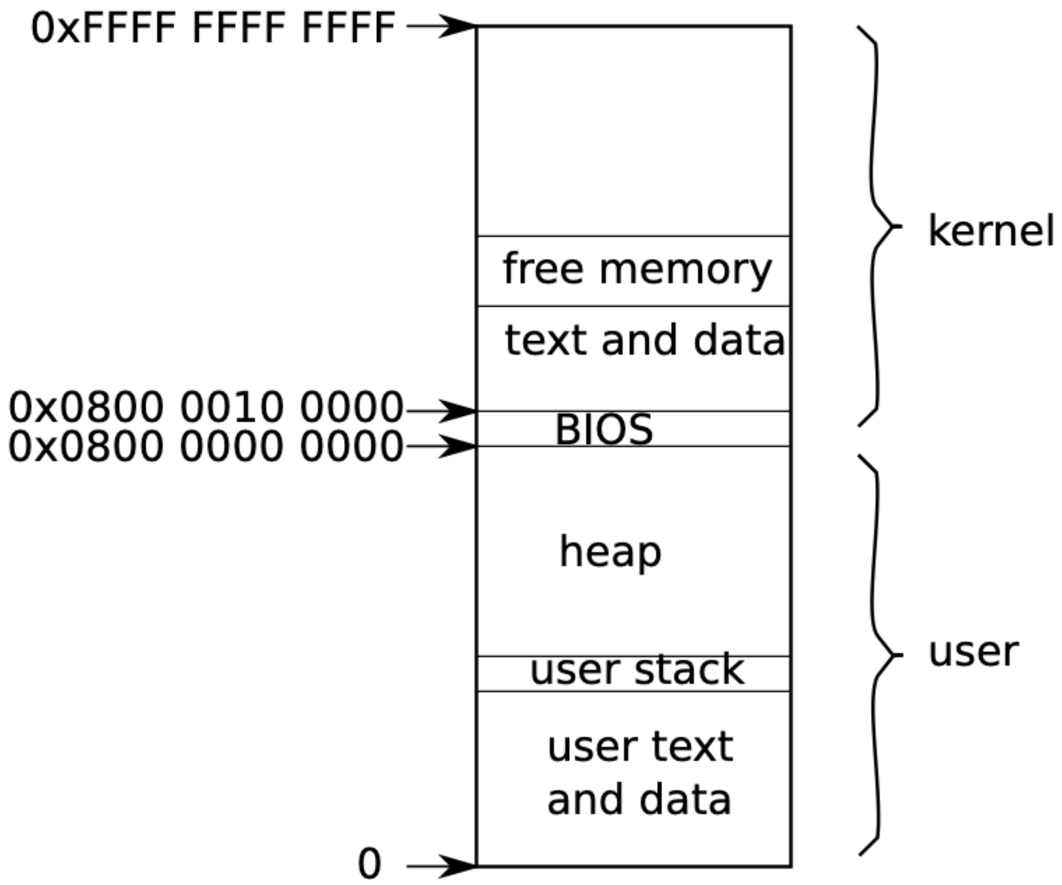


Figure 1-2. Layout of a virtual address space

First page table (cont)

```
// page map level 4 index
#define PML4X(va) (((uint64)(va) >> PML4SHIFT) & 0x1FF)
```

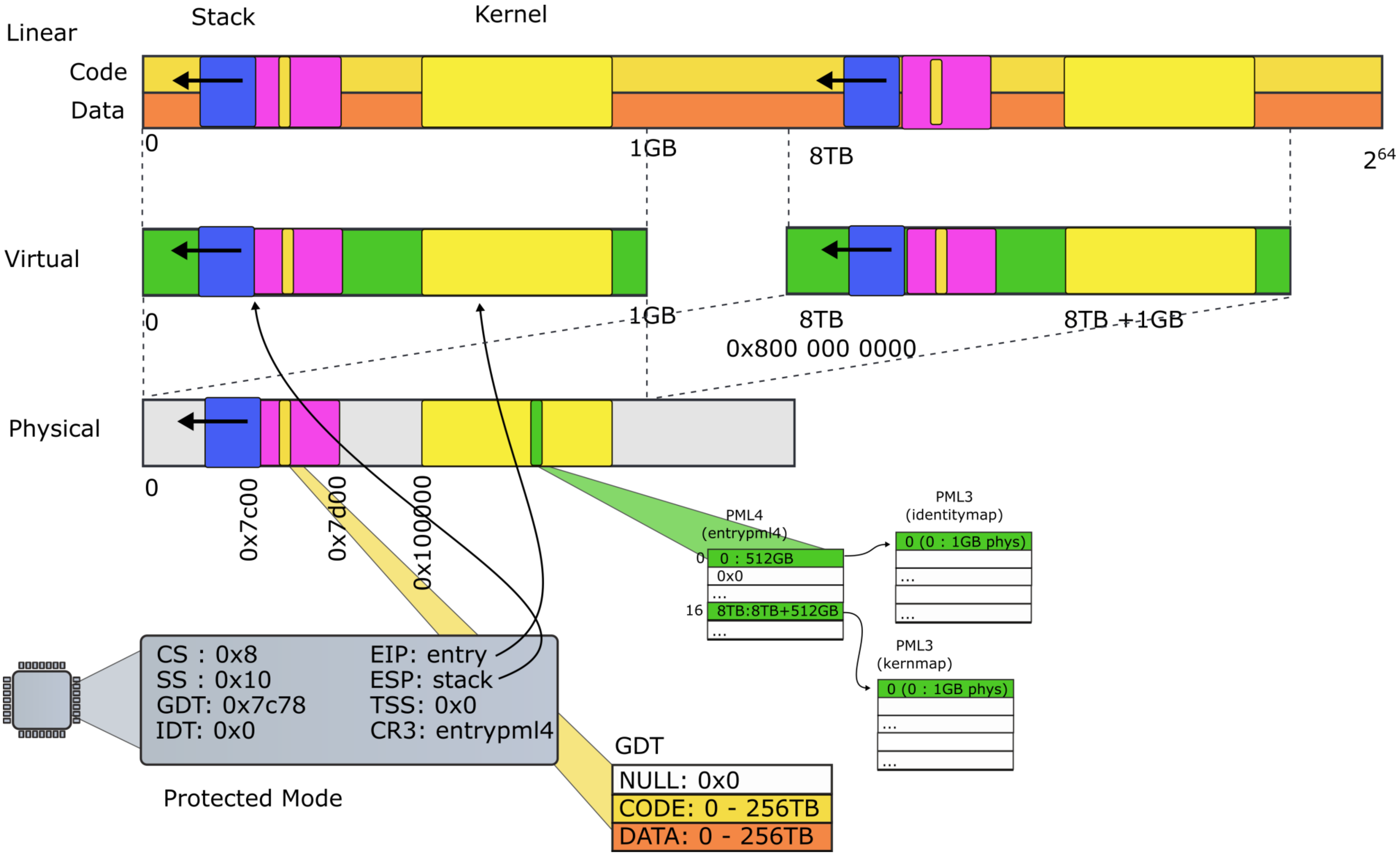
```
__attribute__((__aligned__(PGSIZE)))
pml4e_t entrypml4[NPML4ENTRIES] = {
// Flags below should be added with "|" and not with "+",
// however, "|" seems to complex for the link editor,
// so the compiler refuses to compile the code.
// The use of "+" is valid since PTE_* are only single bits.
[0]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

First page table

```
__attribute__((__aligned__(PGSIZE)))
pdpte_t identitymap[NPDPTENTRIES] = {
  // Map VA's [0, 1GB) to PA's [0, 1GB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
};

__attribute__((__aligned__(PGSIZE)))
pdpte_t kernmap[NPDPTENTRIES] = {
  // Map VA's [KERNBASE, KERNBASE+1GB) to PA's [0, 1GB)
  [PDPTX(KERNBASE)] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

Two 1GB pages

First page table



Turn on paging

```
# Enable IA-32e mode by setting IA32_EFER.LME = 1.
movl $EFER_MSR, %ecx
rdmsr
orl $EFER_MSR_LME, %eax
wrmsr

# Enable paging.
# This causes the processor to set the IA32_EFER.LMA bit to 1.
movl %cr0, %eax
orl $CR0_PG, %eax
movl %eax, %cr0
```

xv6/entry.S [kernel]

Switch to 64bit mode

```
# We are now in the 32-bit compatibility submode of IA-32e mode.
# To complete the transition to 64-bit submode, we have to load
# a gdt with the 64-bit flag set (in the code segment), and then
# use a far jump to reload %cs and %rip.
lgdt (V2P_WO(gdt64desc))
ljmp $(SEG_KCODE<<3), $(V2P_WO(start64))
```

xv6/entry.S [kernel]

Switch to 64bit mode

```
.data
…
.p2align 4 # force 16 byte alignment
gdt64:
  SEG_NULLASM # null seg
  SEG64_ASM(STA_X|STA_R, SEG64_CODE) # code seg
  SEG64_ASM(STA_W, SEG64_OTHER) # data seg

gdt64desc:
  .word (gdt64desc - gdt64 - 1) # sizeof(gdt64) - 1
  .quad V2P_WO(gdt64) # address gdt64
```

xv6/entry.S [kernel]

Jump to main()

```
.p2align 4
.code64 # Tell assembler to generate 64-bit code now.
start64:
  # 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.
  movabs $(stack + KSTACKSIZE), %rsp
  movabs $main, %rax
  jmp *%rax
```

xv6/entry.S [kernel]

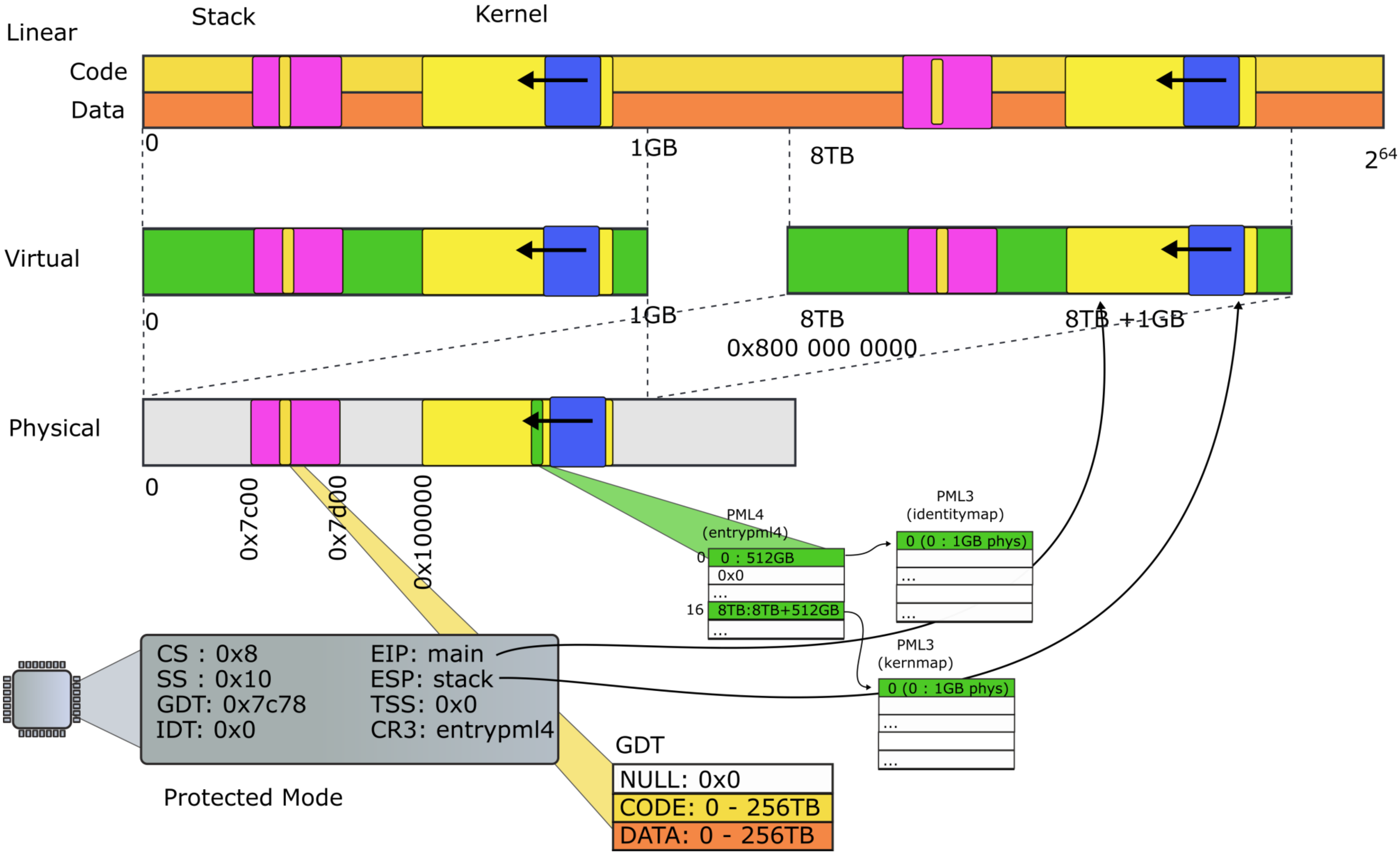
High address stack (4K)

```
1186 .comm stack, KSTACKSIZE
```

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

xv6/entry.S [kernel]

High address stack (4K)



CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0

EIP: main
ESP: stack
TSS: 0x0
CR3: entrypml4

GDT

NULL: 0x0
CODE: 0 - 256TB
DATA: 0 - 256TB

Jump to main()

```
.p2align 4
.code64 # Tell assembler to generate 64-bit code now.
start64:
  # 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.
  movabs $(stack + KSTACKSIZE), %rsp
  movabs $main, %rax
  jmp *%rax
```

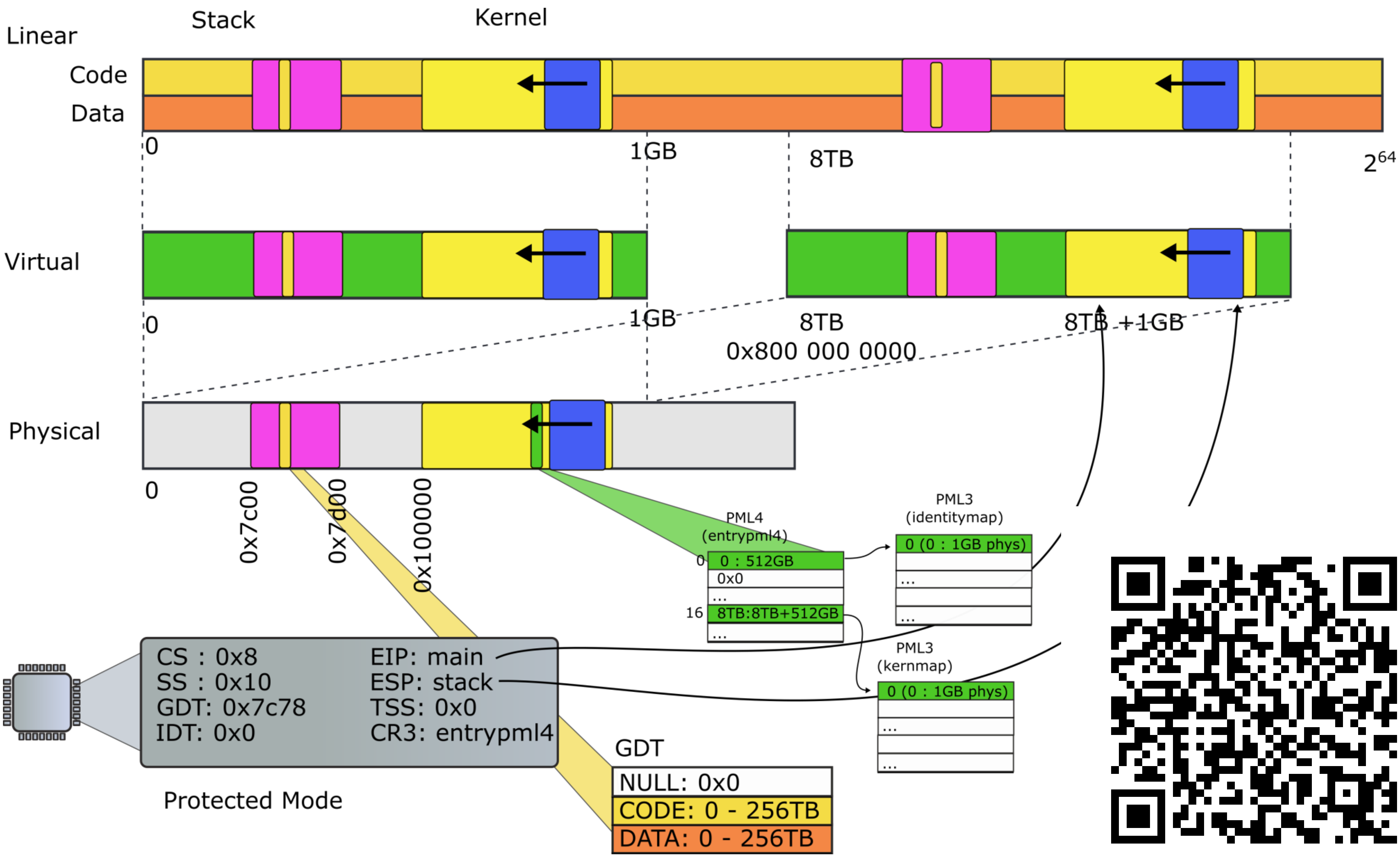
xv6/entry.S [kernel]

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());
...
}
```

xv6/main.c [kernel]

Poll



CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0

EIP: main
ESP: stack
TSS: 0x0
CR3: entrypml4

Protected Mode

GDT

NULL: 0x0
CODE: 0 - 256TB
DATA: 0 - 256TB

https://pollev.com/cs5460

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