Operating Systems

Lecture: System boot

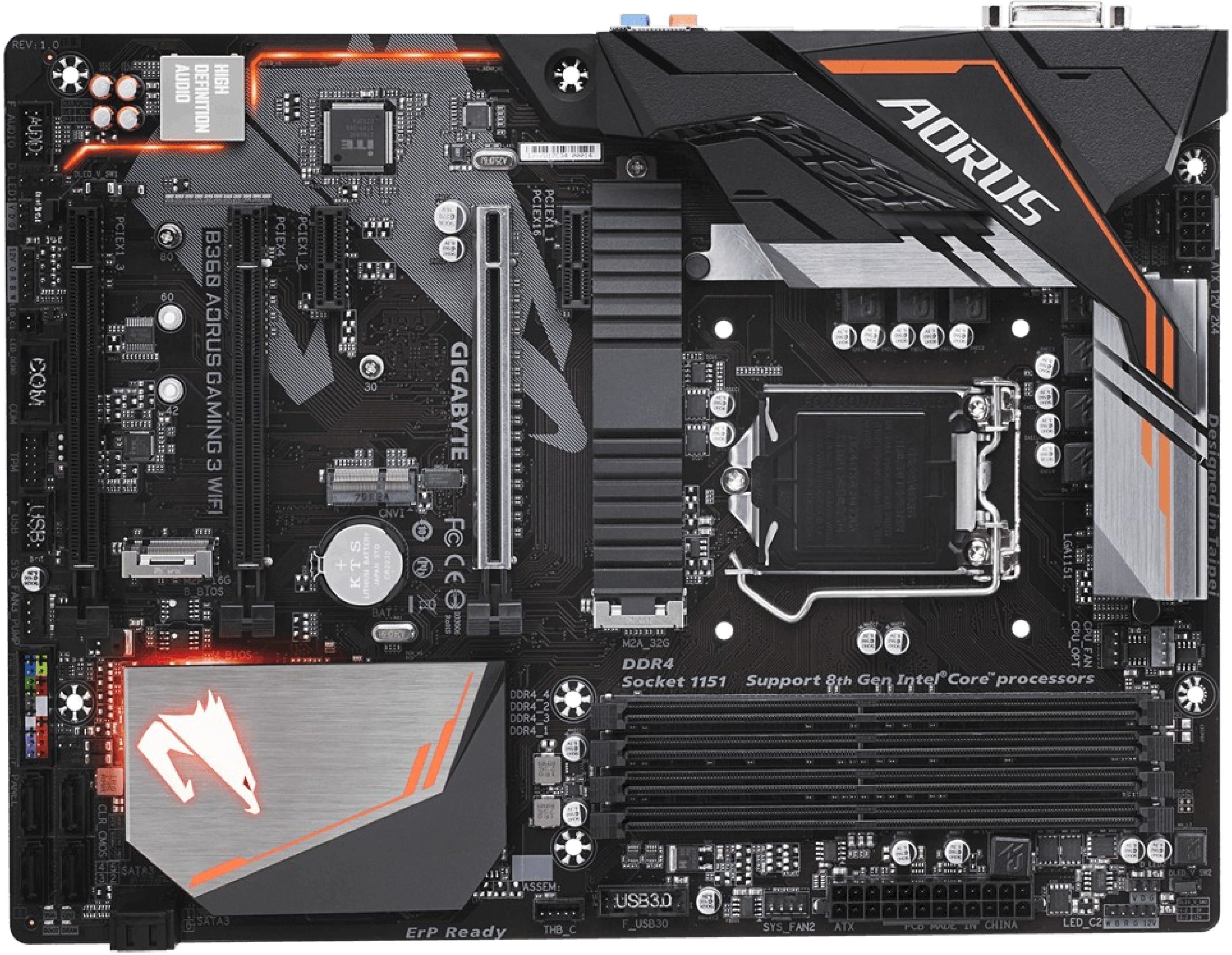
Anton Burtsev

What happens when we turn on the power?

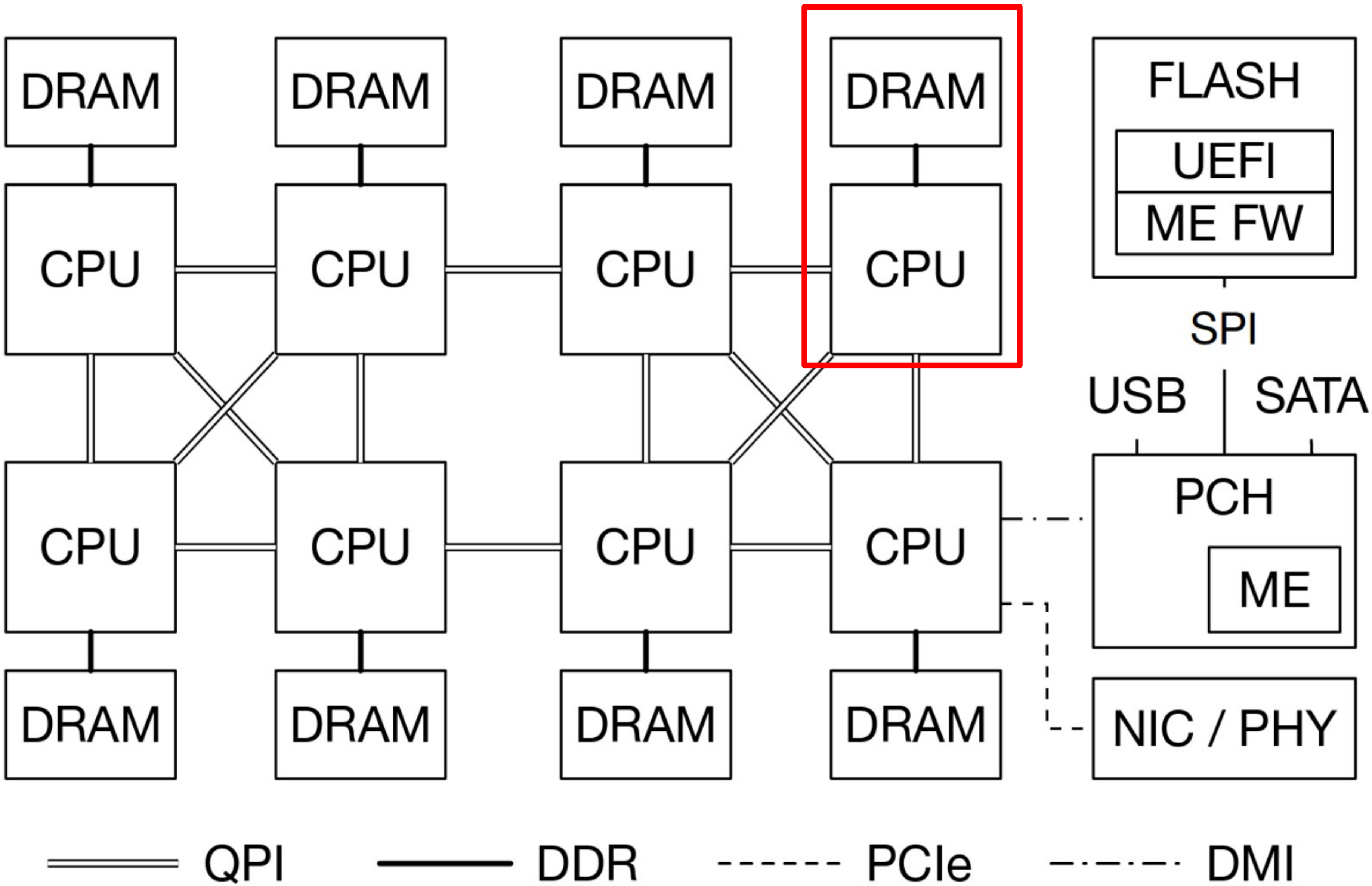
- Well it's complicated
  - Intel SGX Explained is a good start (Section 2.13 [1])
- At a high-level a sequence of software pieces initializes the platform
  - Management engine (ME), microcode, firmware (BIOS), bootloader

- 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)
    - Run on separate CPUs

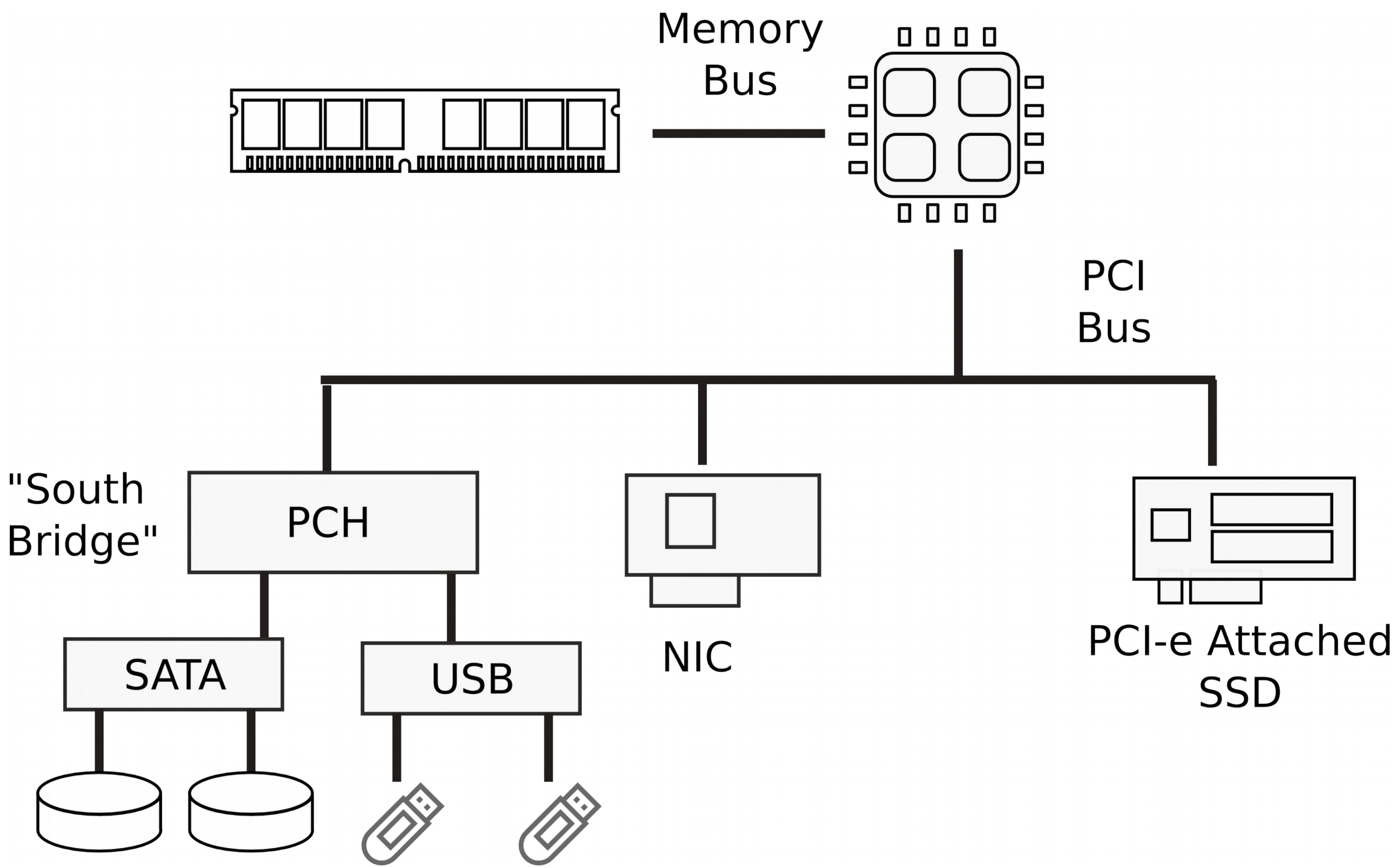
B360 AORUS Motherboard



PC motherboard components

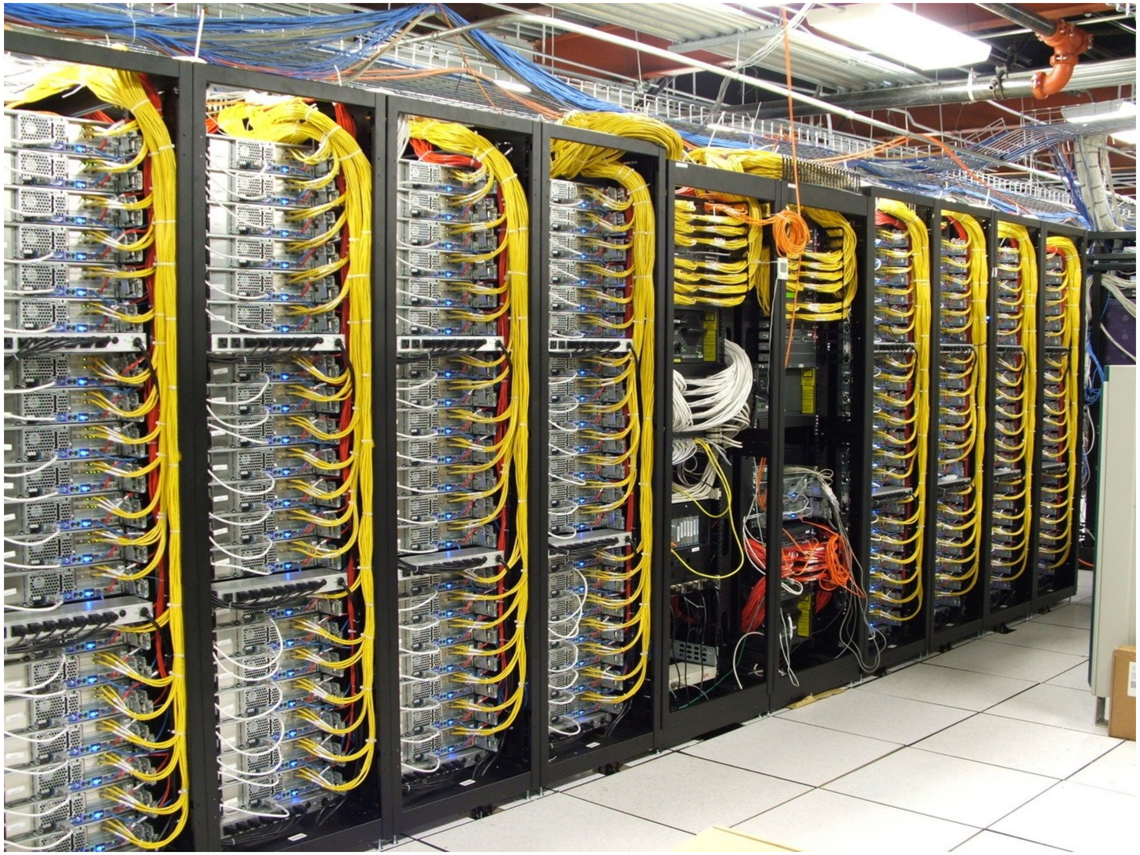


I/O Devices



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

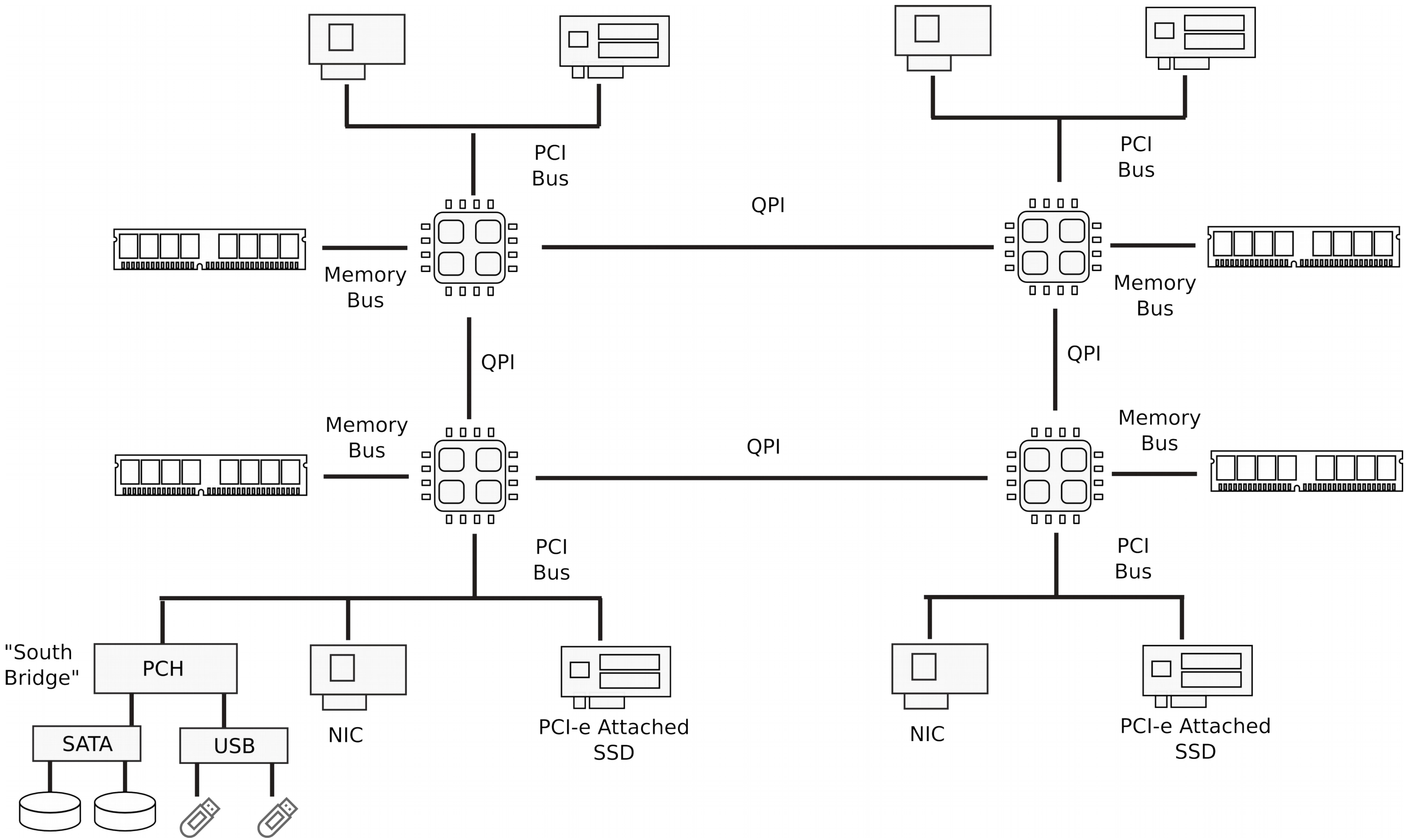
Dell R830 4-socket server



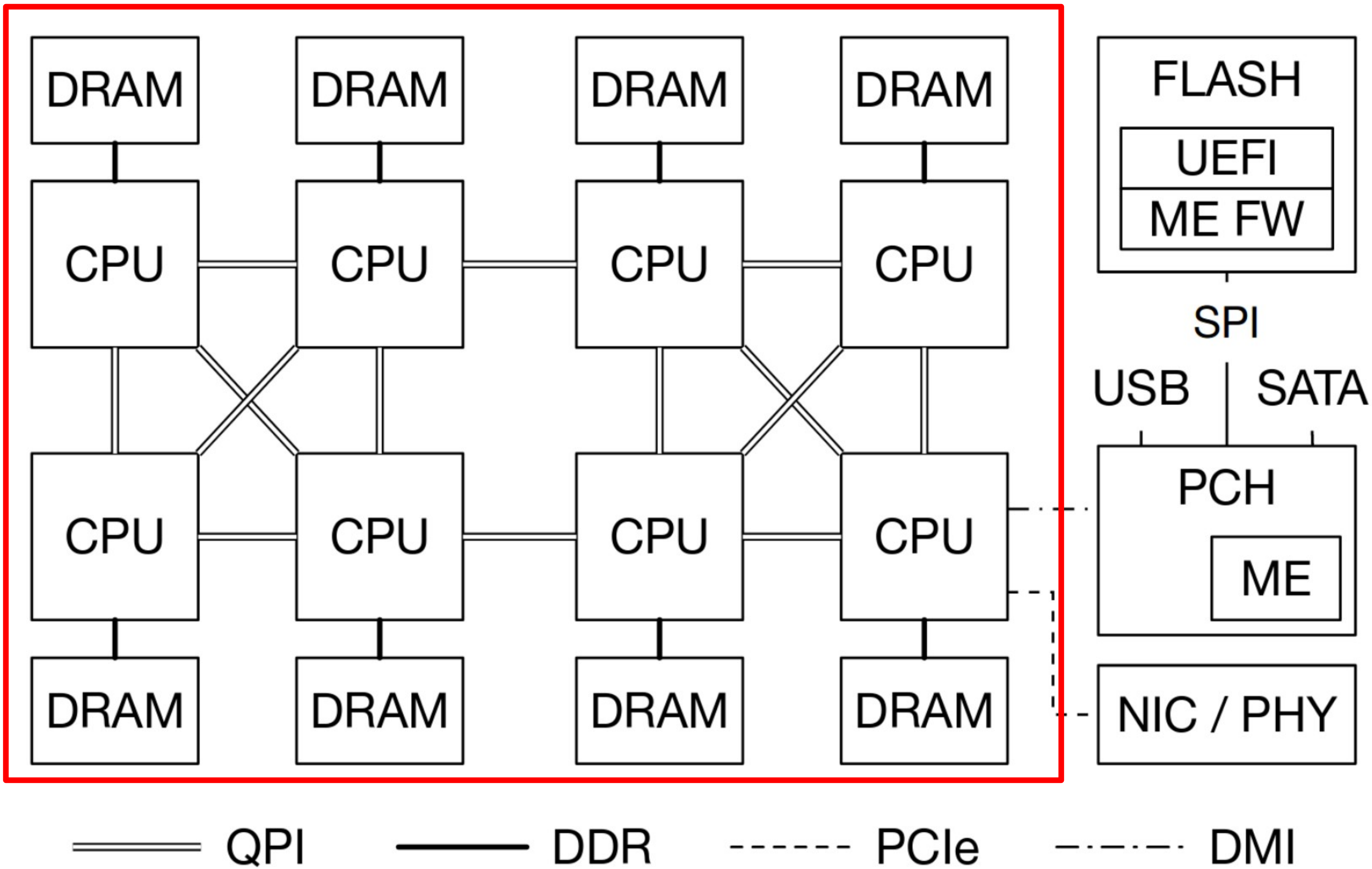
Dell Poweredge R830 System Server with 2 sockets
on the main floor and 2 sockets on the expansion

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

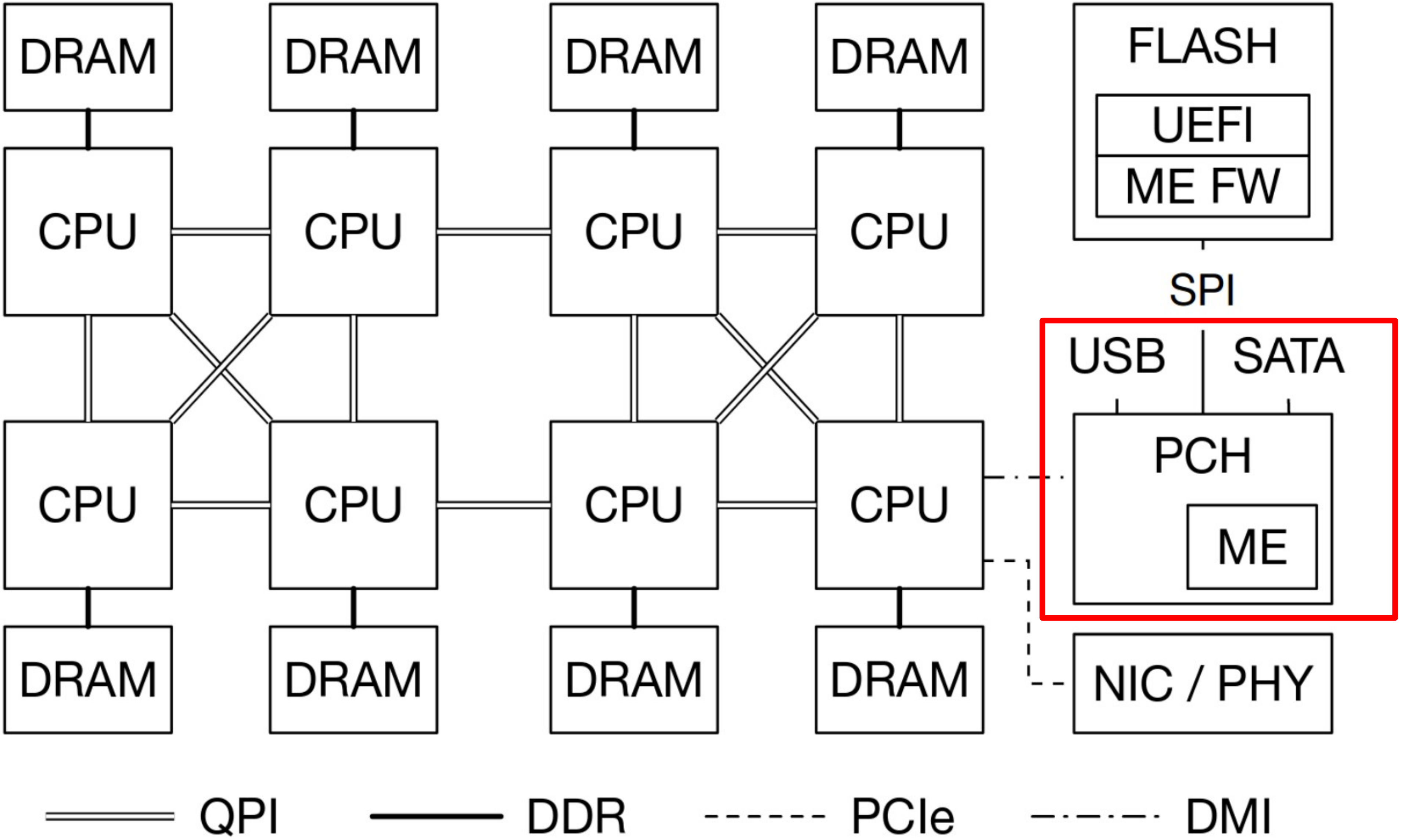
Multi-socket machines



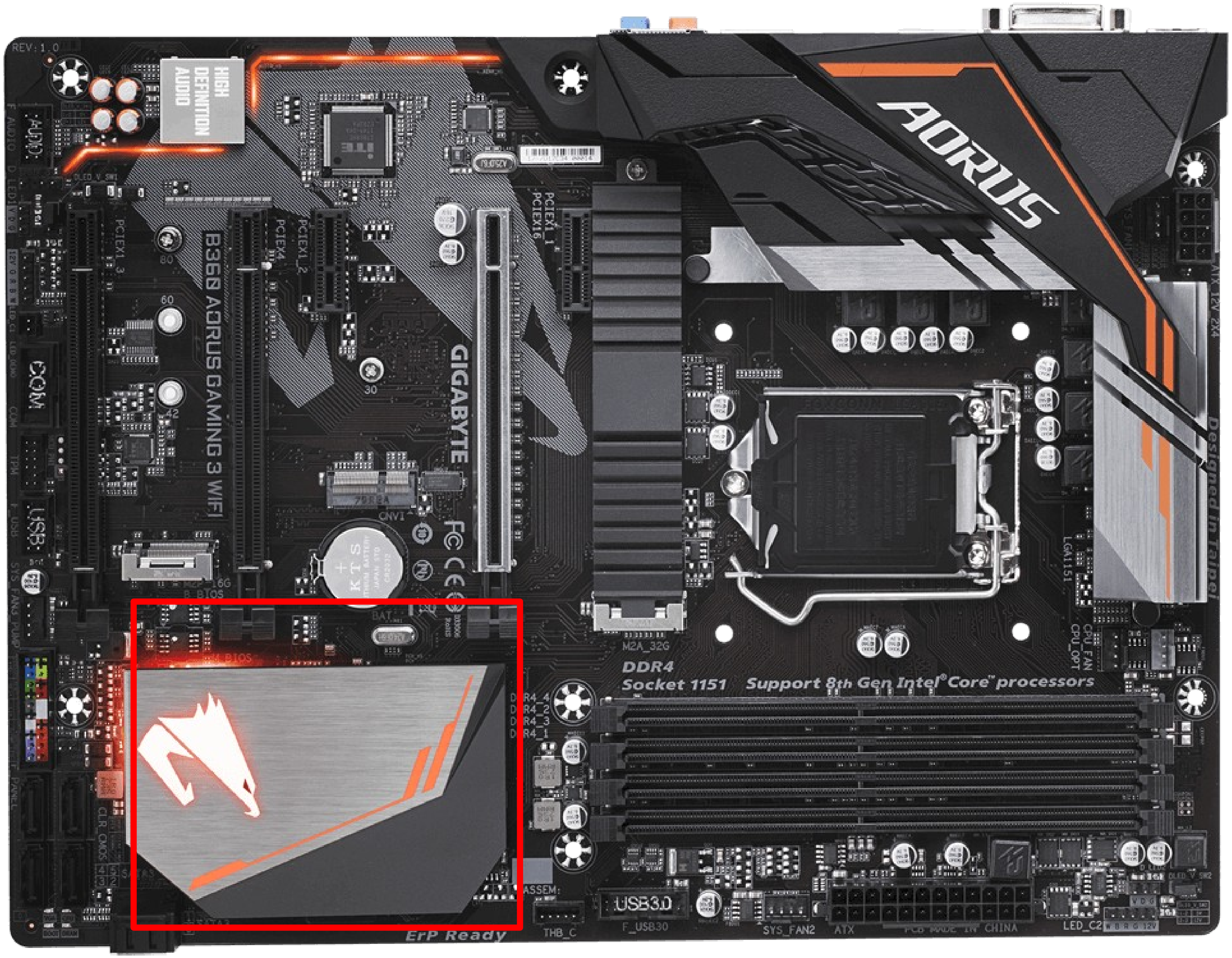
PC motherboard components



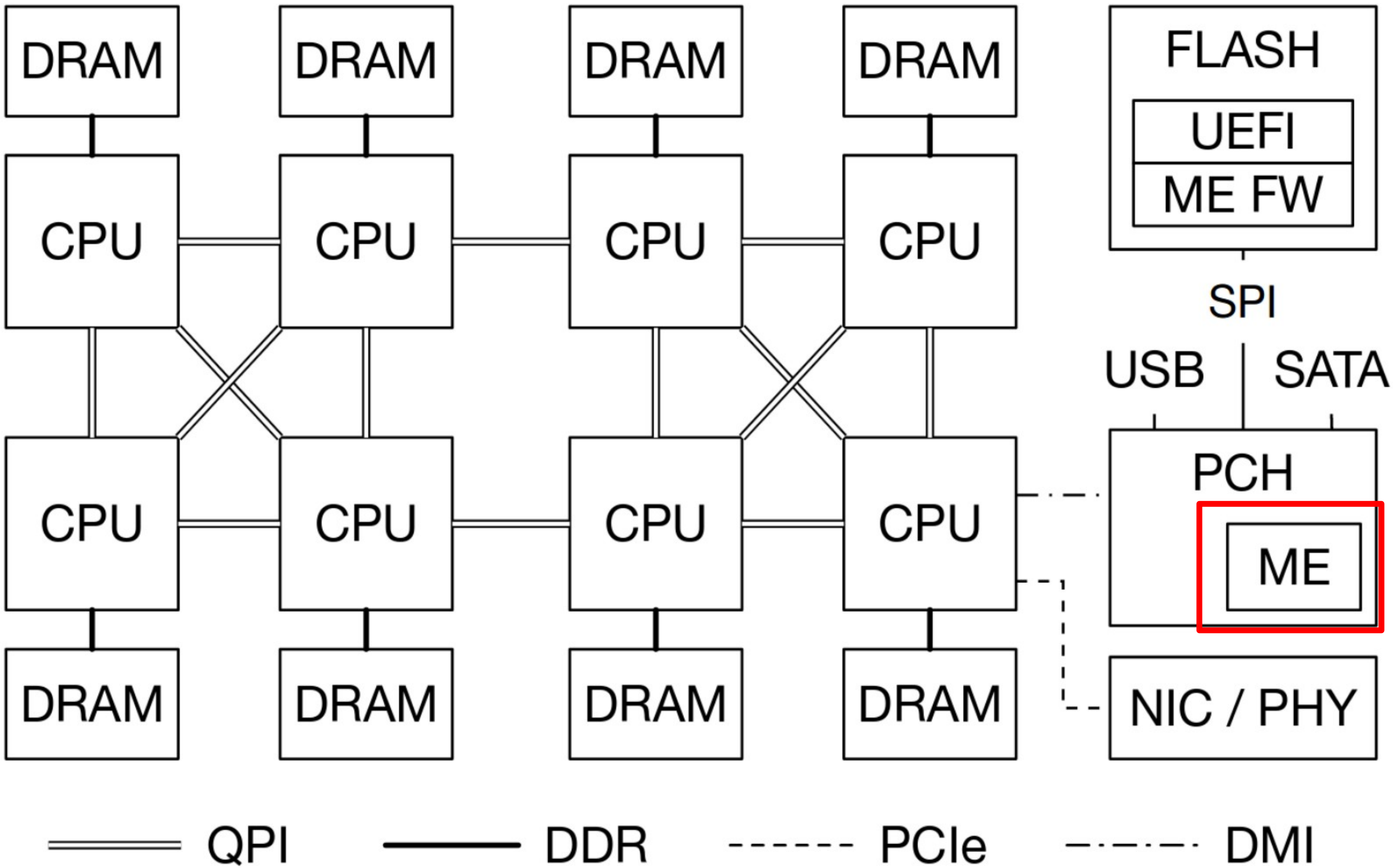
PCH – Platform Controller Hub



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?

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
     ...
```

"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); ...

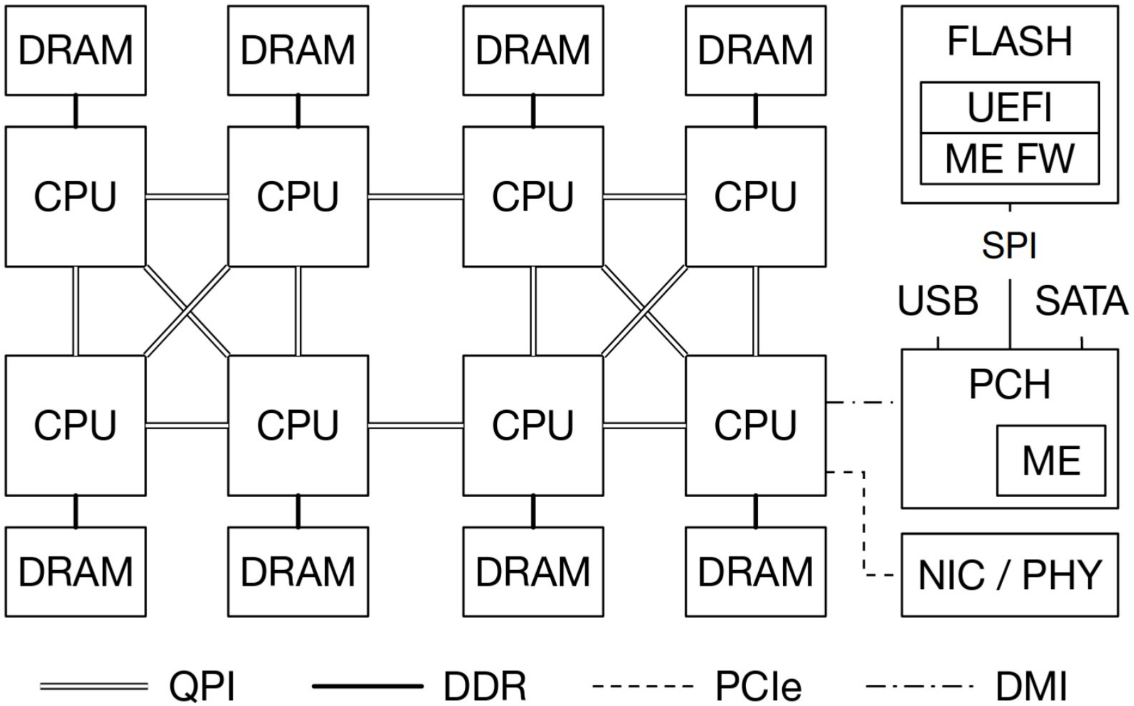
MINIX3
by Andrew Tanenbaum

```
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
                                    ...
```

```c
/* 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);
```

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
- 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

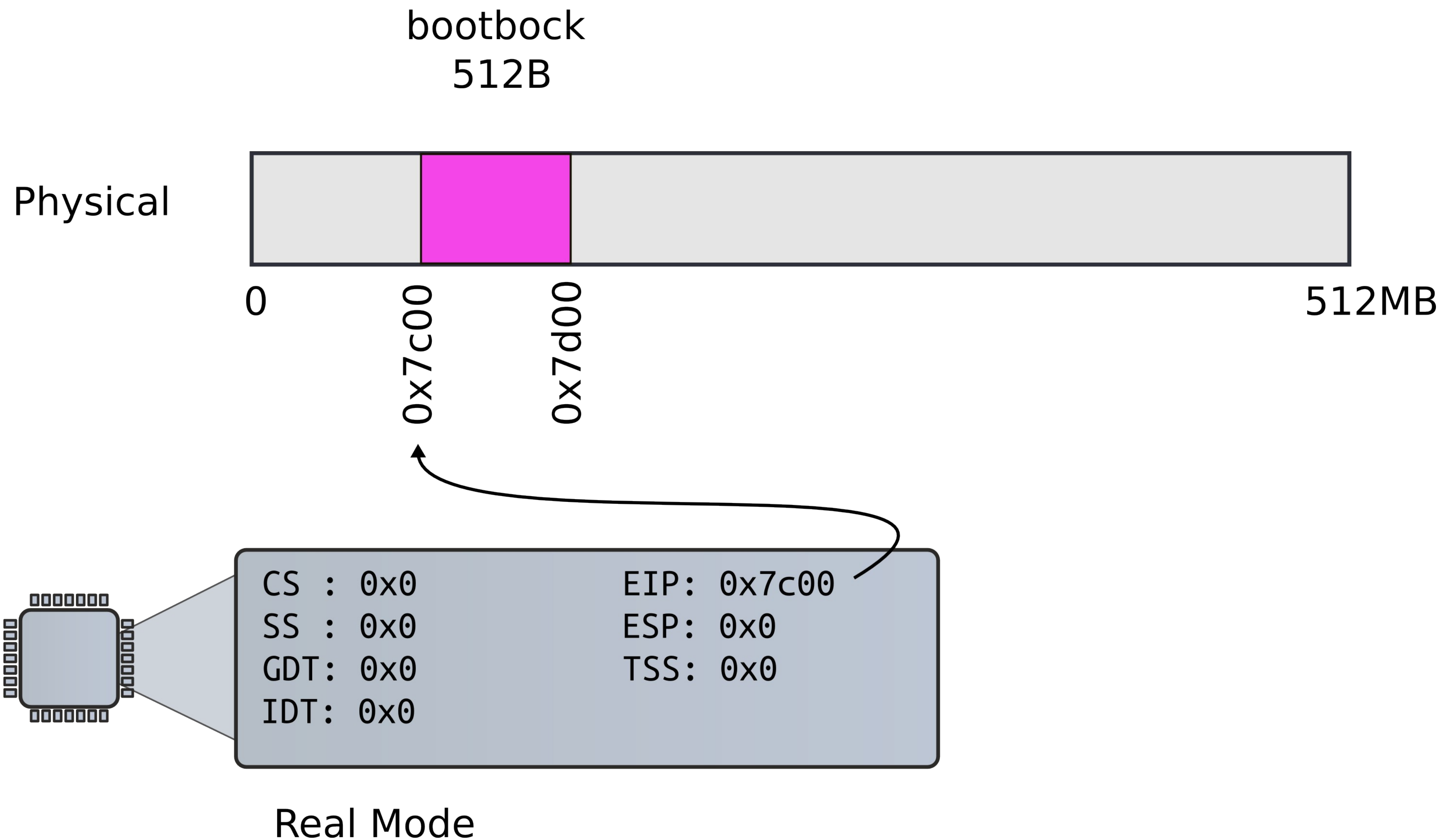
System Management Mode

- 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 or the hypervisor
  - Receives interrupts periodically, can take over the entire system any time
  - Impossible 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



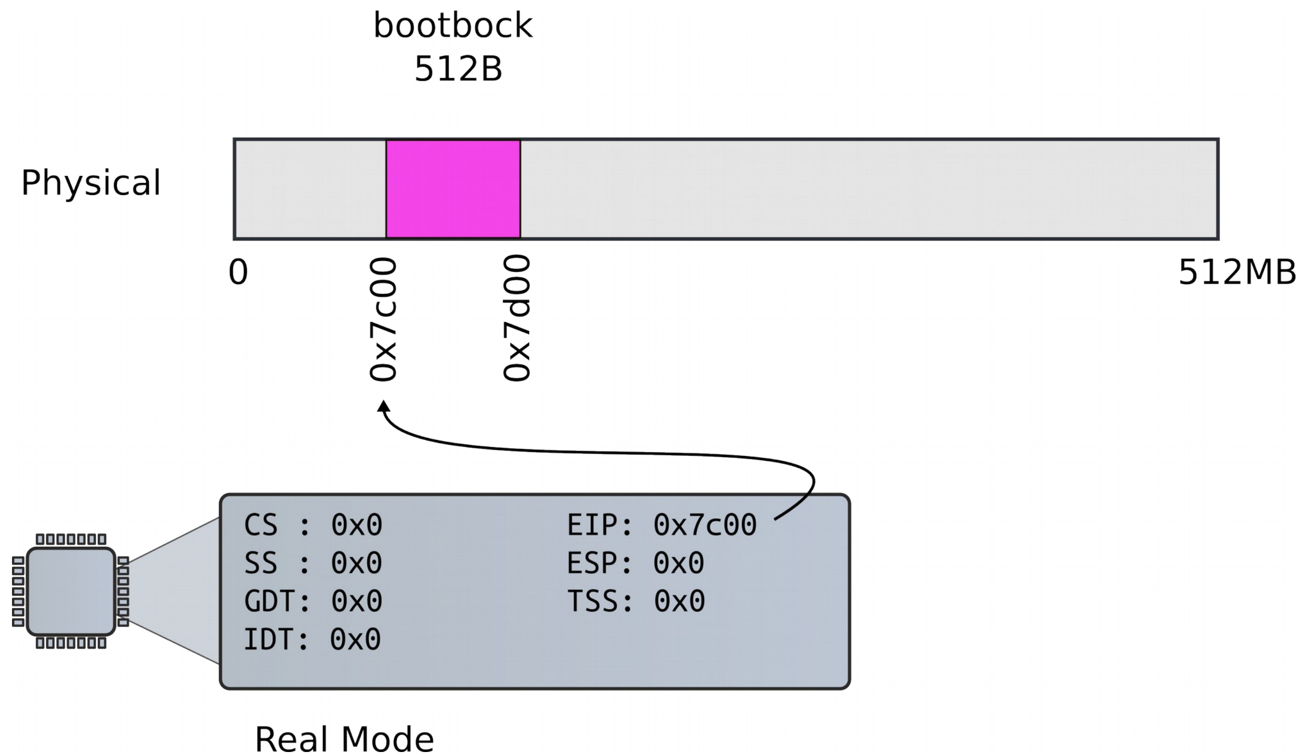
Bootloader starts

```
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?

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

```



xv6/bootasm.S [bootloader]

Linker is instructed to link the boot block code in 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
```

xv6/Makefile

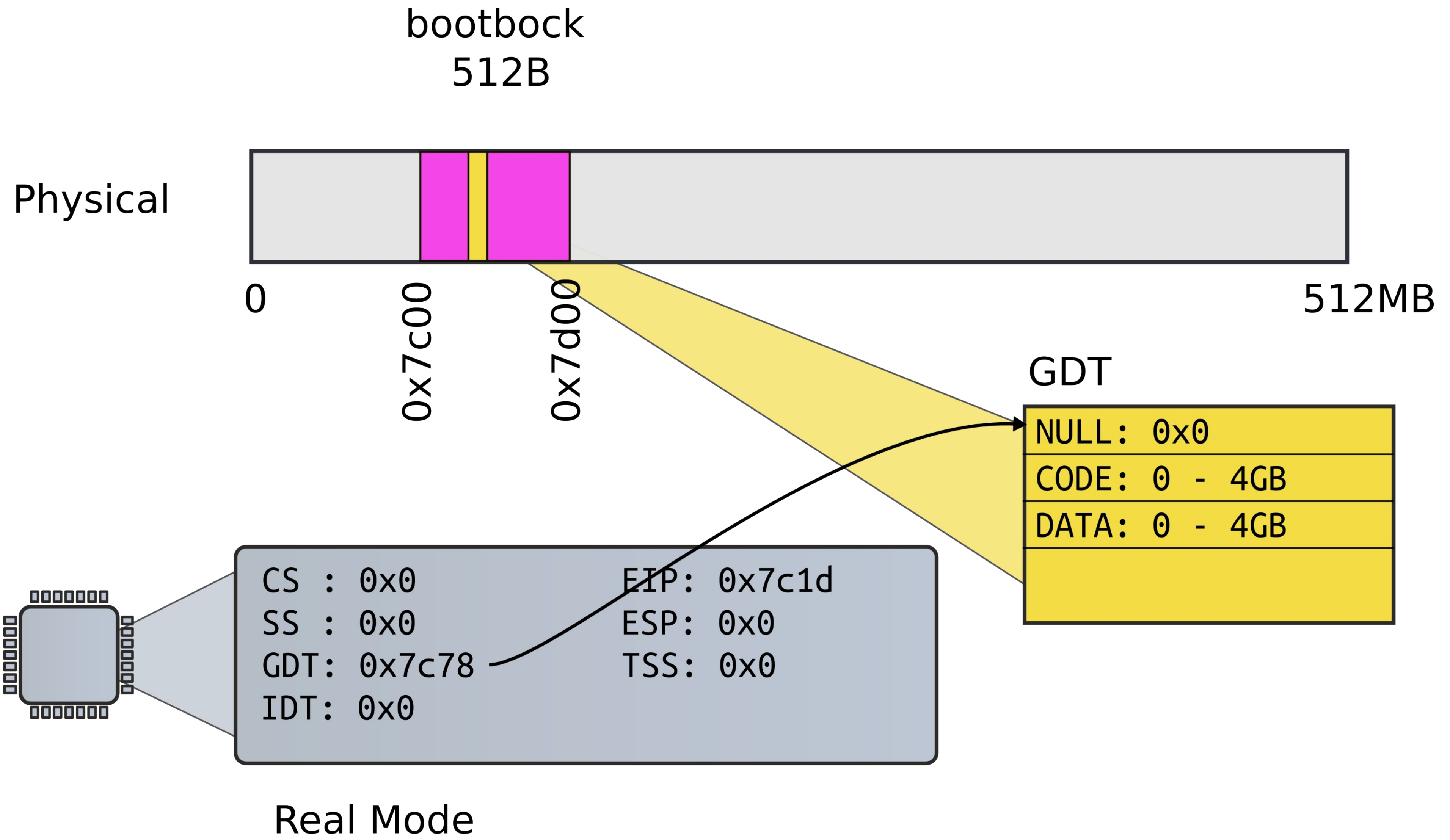
Switch to protected mode

- 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.

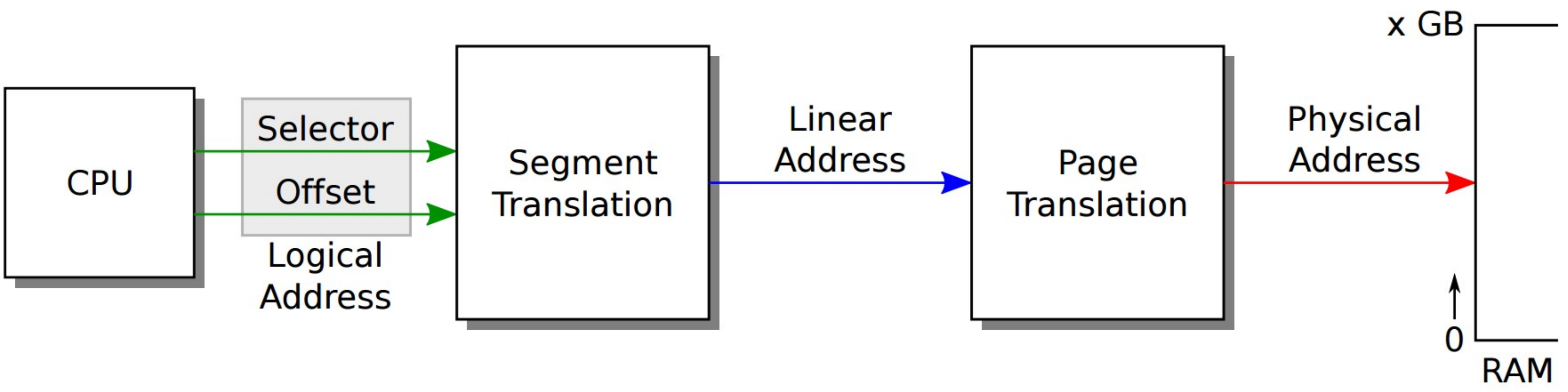
```
lgdt gdtdesc
movl %cr0, %eax
orl $CR0_PE, %eax
movl %eax, %cr0
```

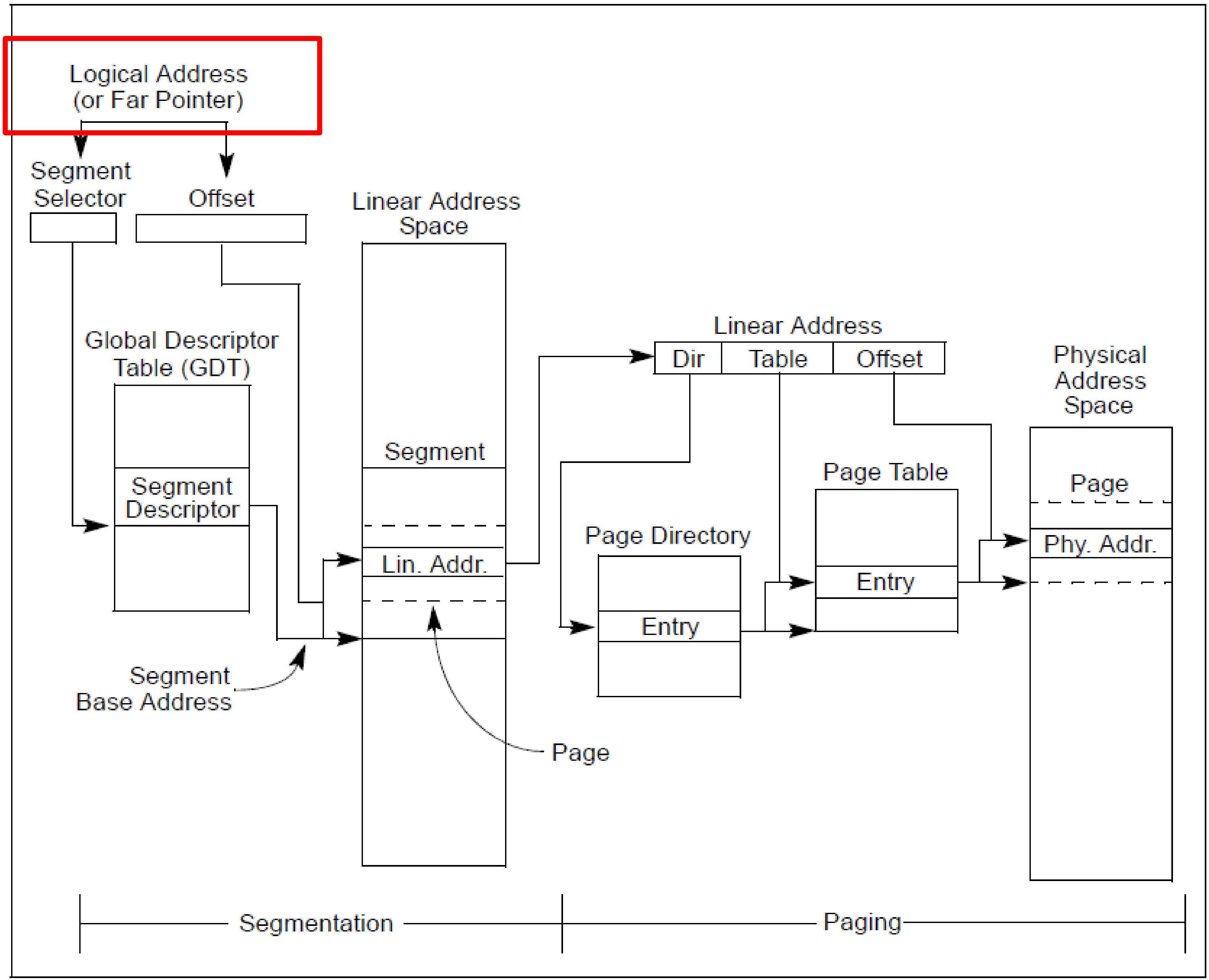
xv6/bootasm.S [bootloader]

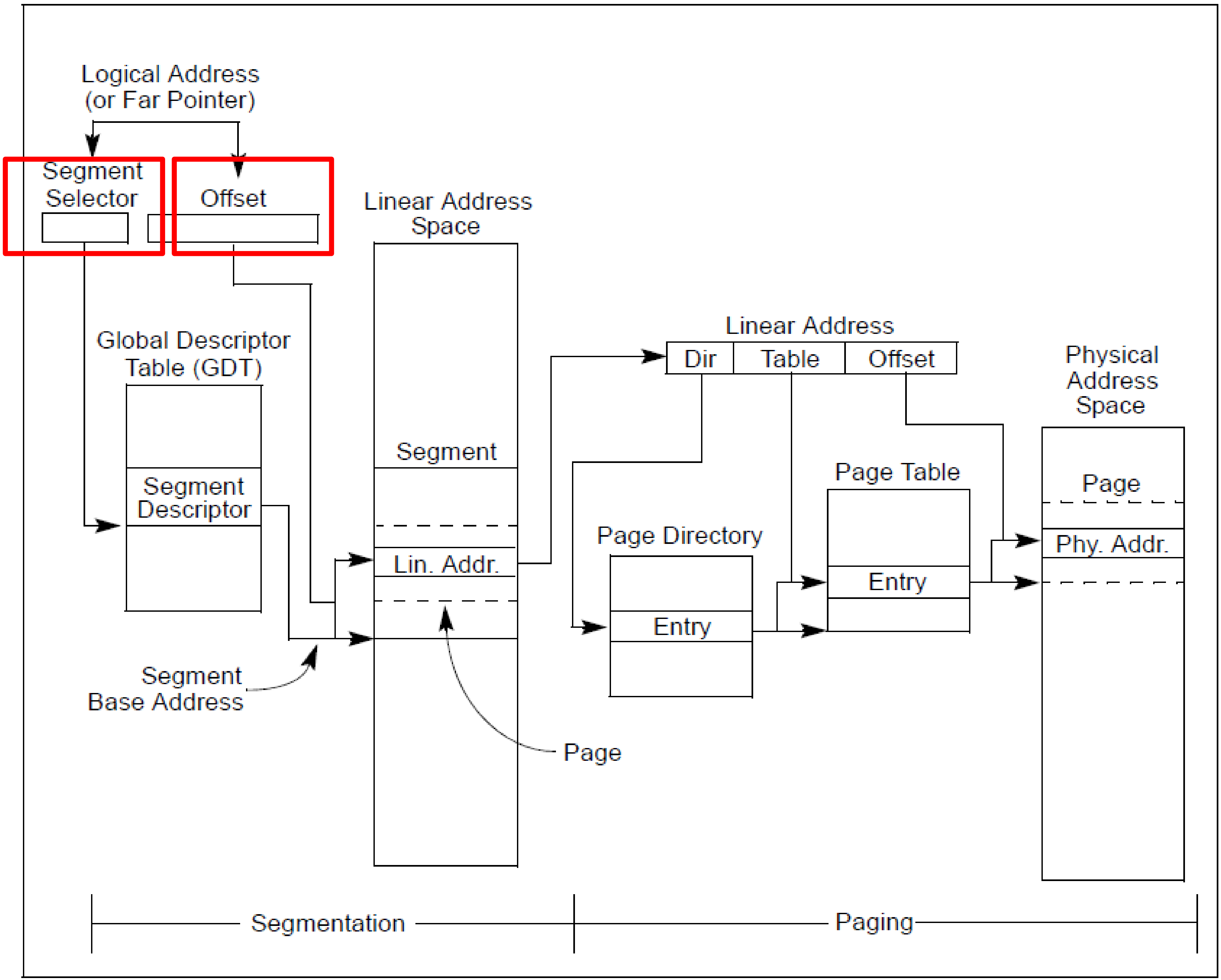
Load GDT


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

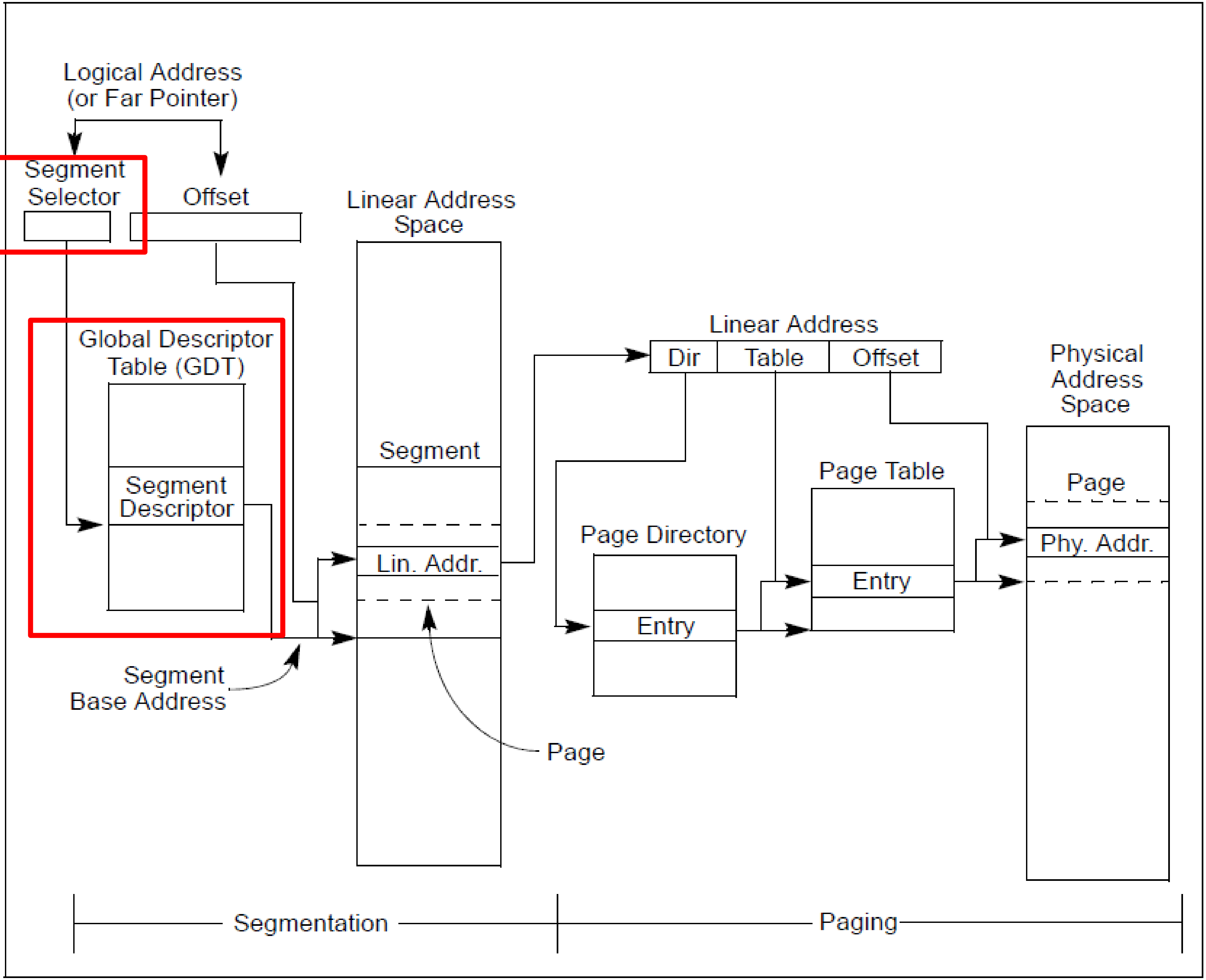
Recap: complete address translation

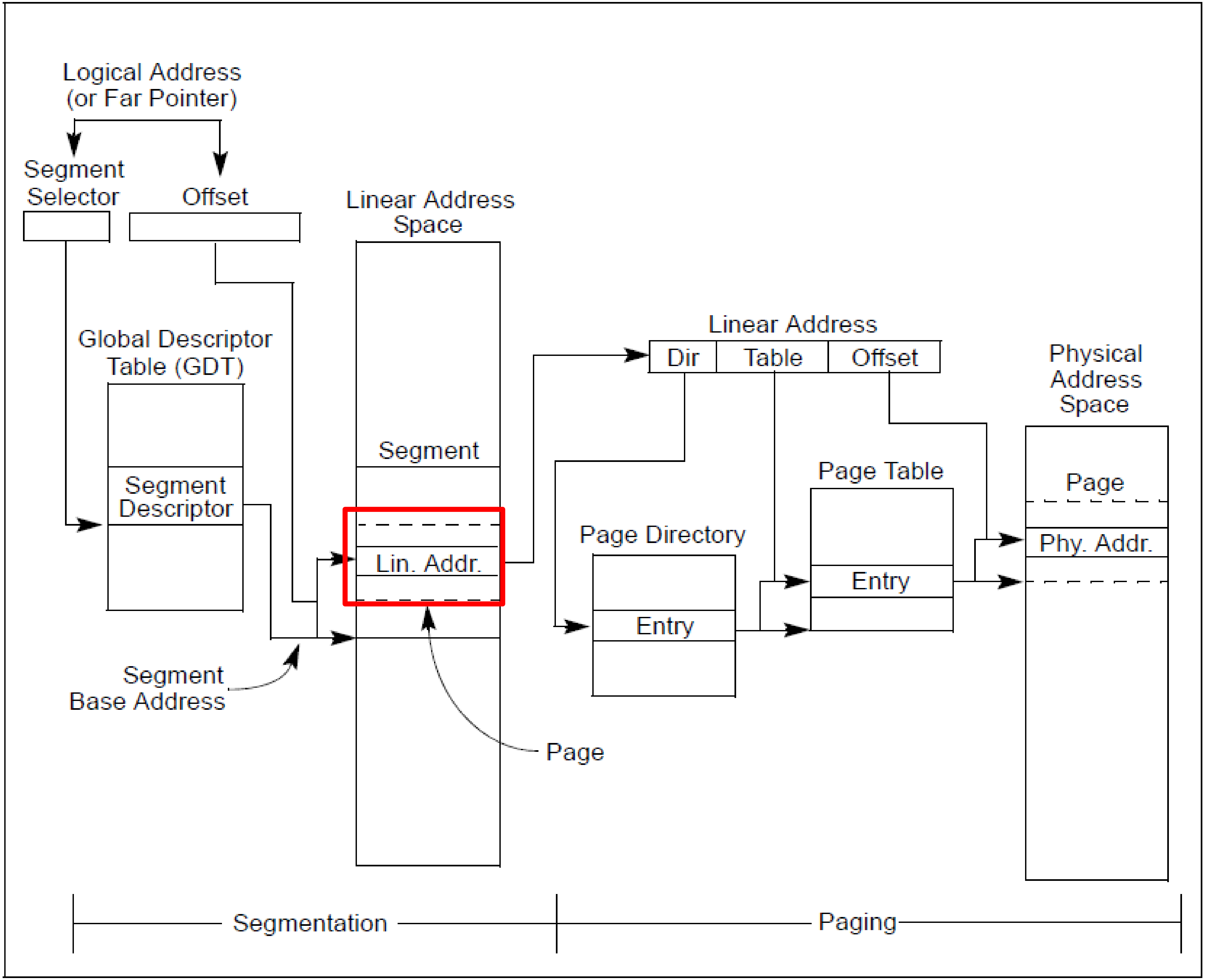



Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Physical
Address
Space
Page Directory
Entry
Page Table
Entry
Page
Phy. Addr.
Segmentation
Paging

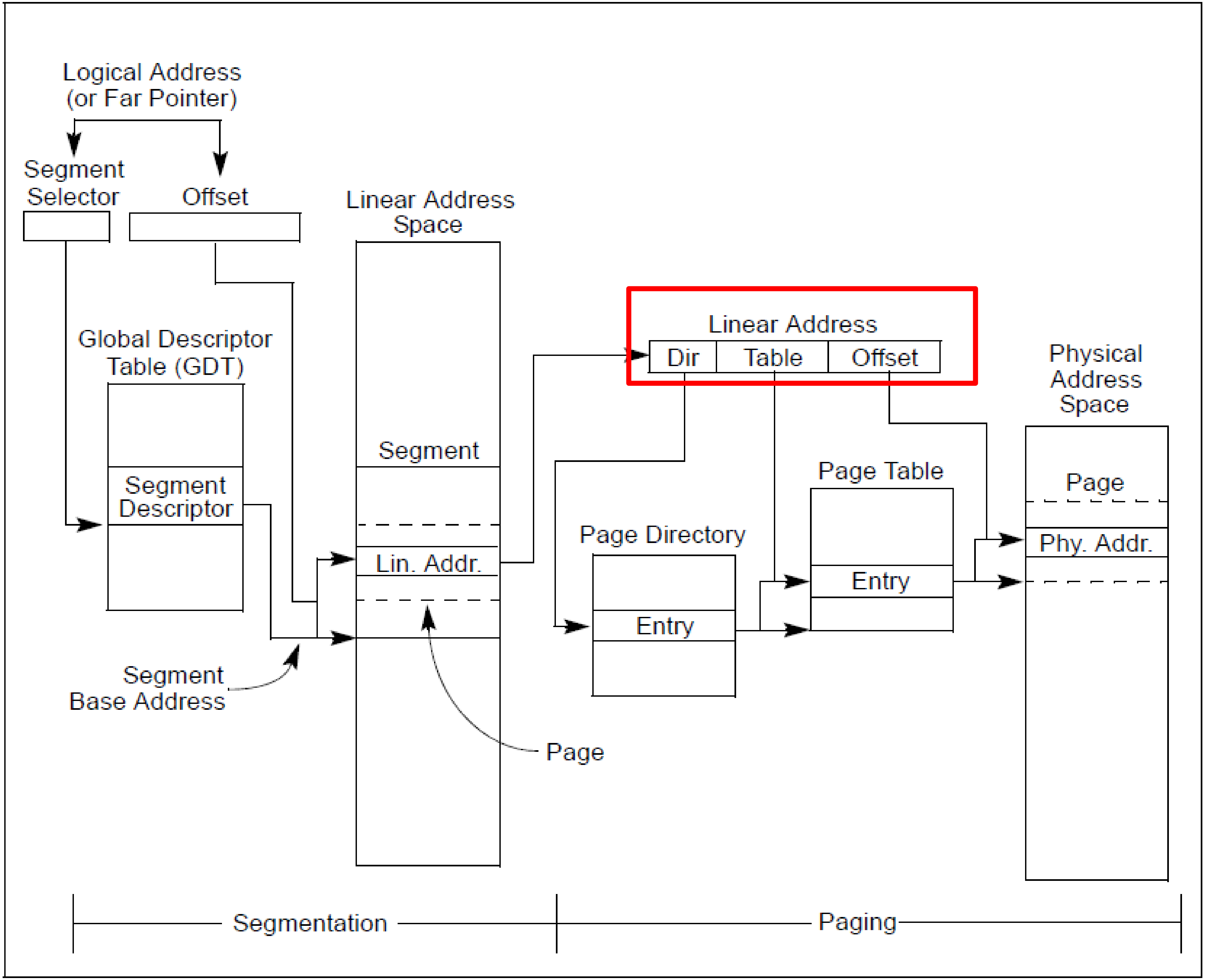


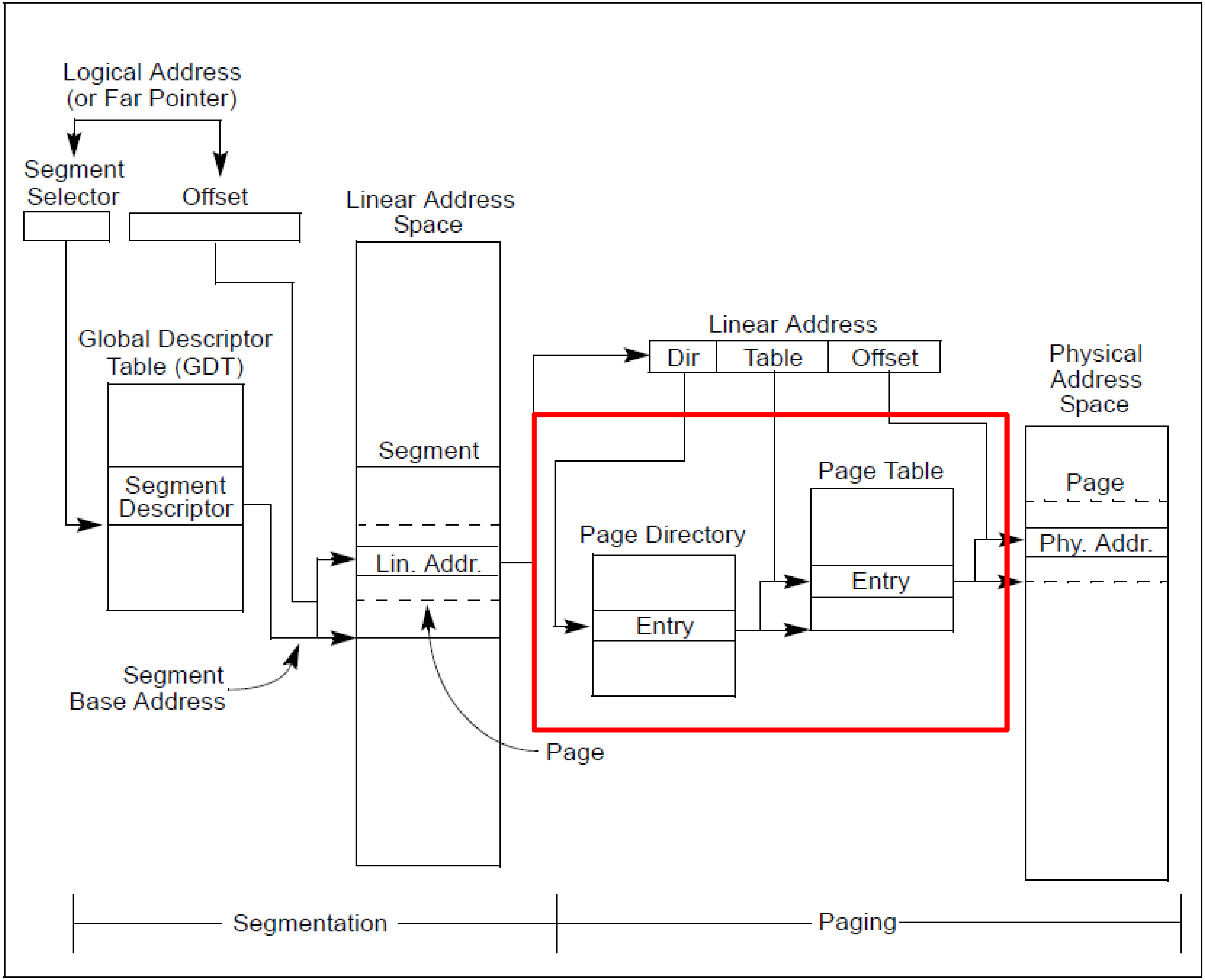
Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

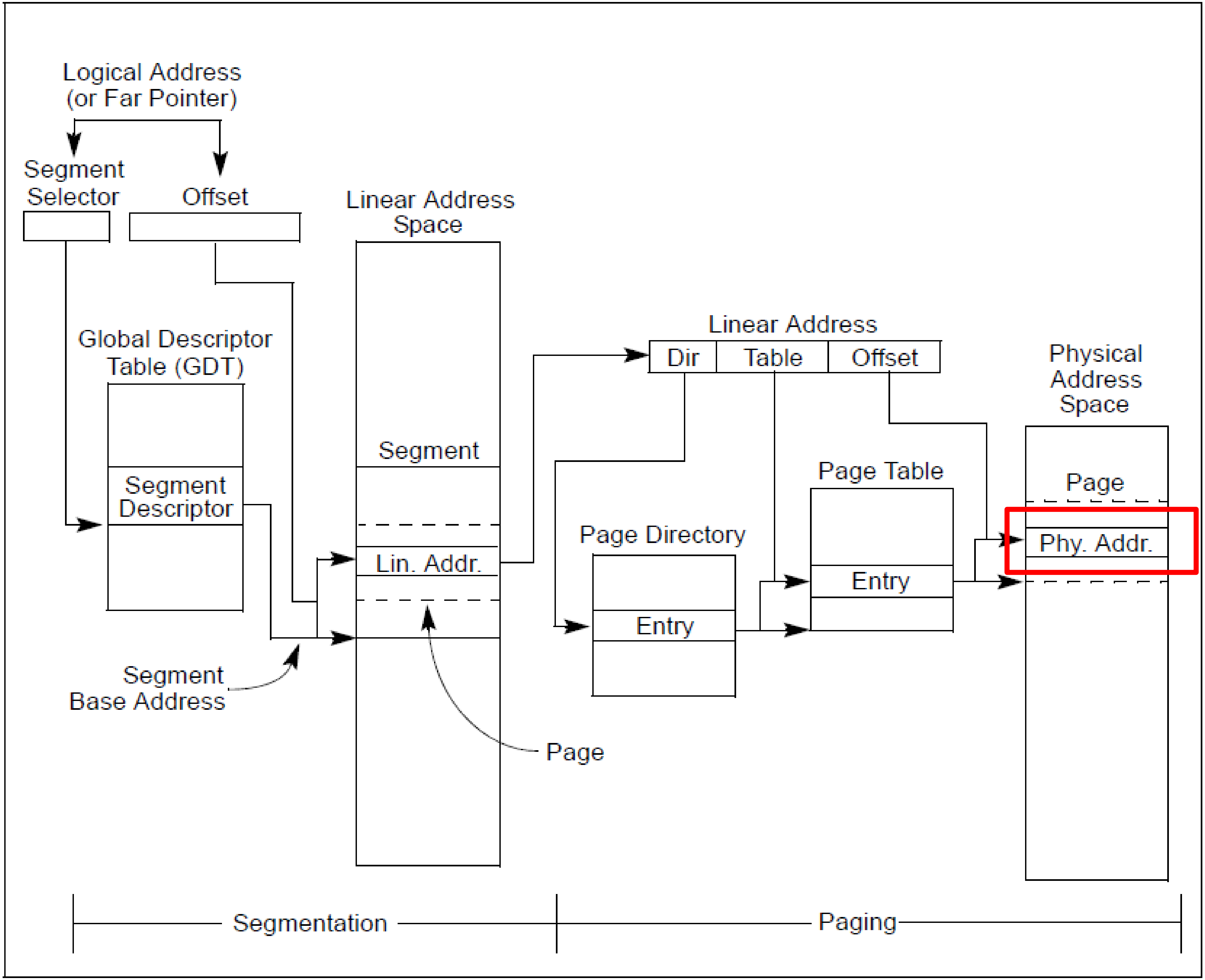


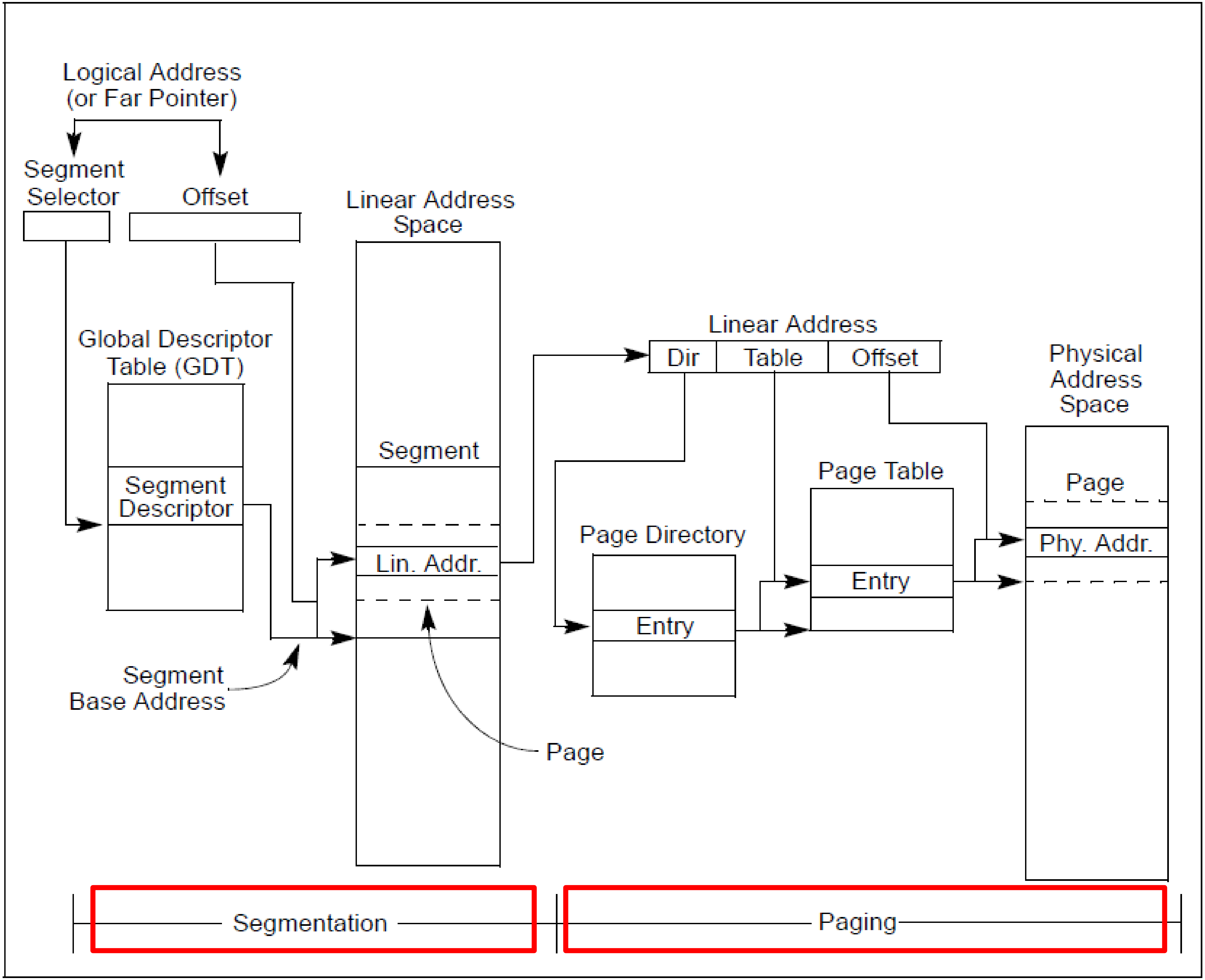
Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging



Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

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

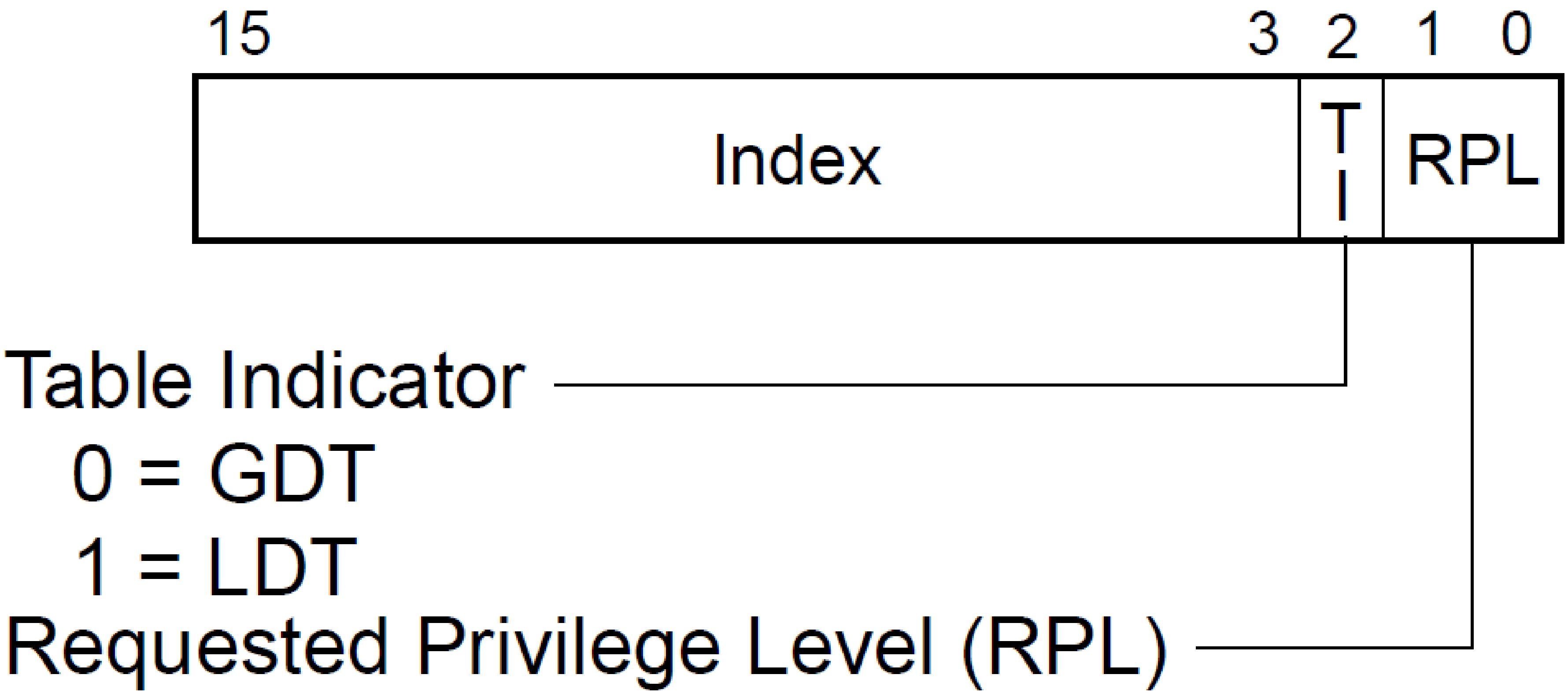
```
9153 ljmp $(SEG_KCODE<<3), $start32
```

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

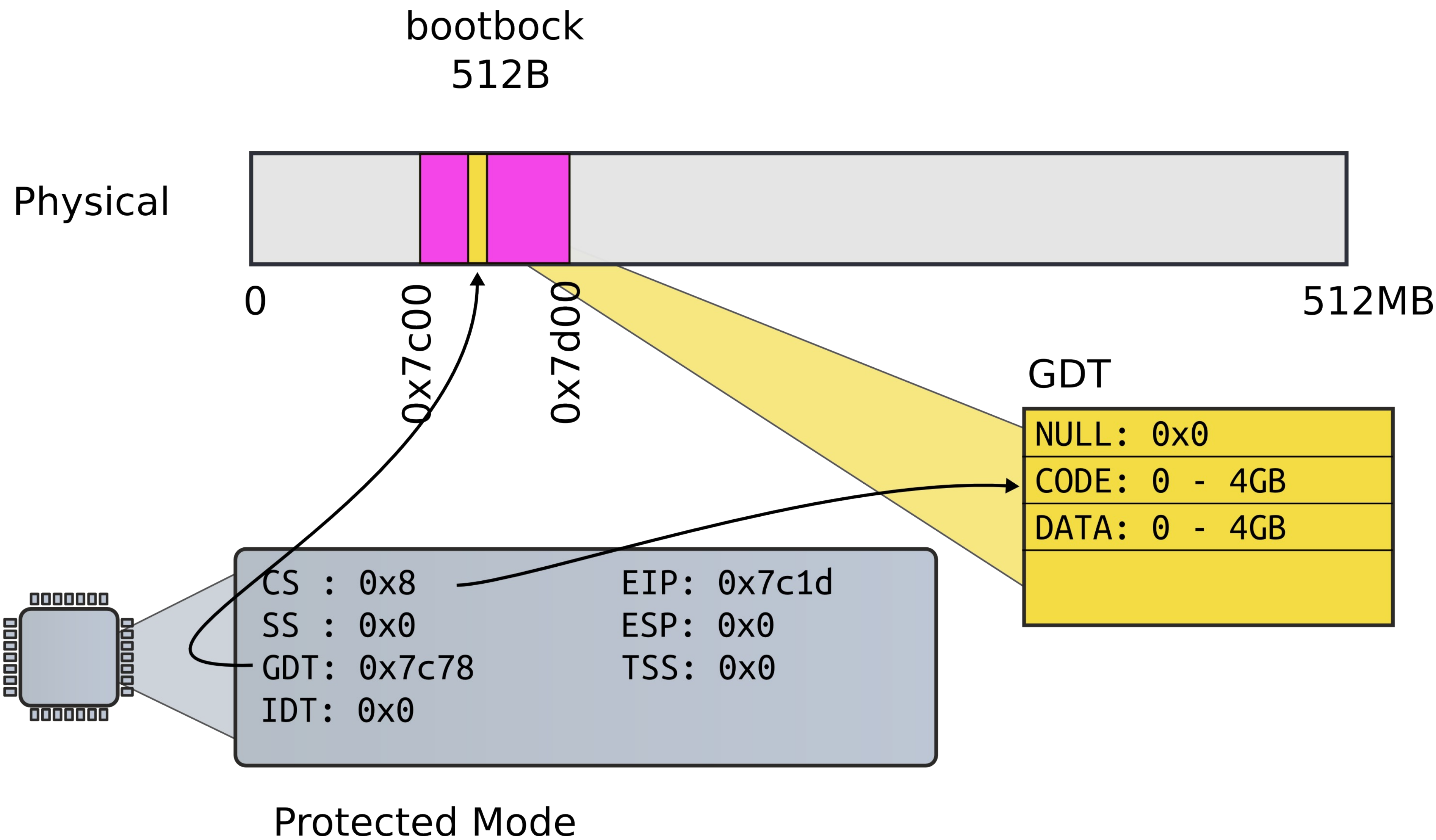
xv6/bootasm.S [bootloader]

Why CS is 0x8, not 0x1?

- Segment selector:



Long jump

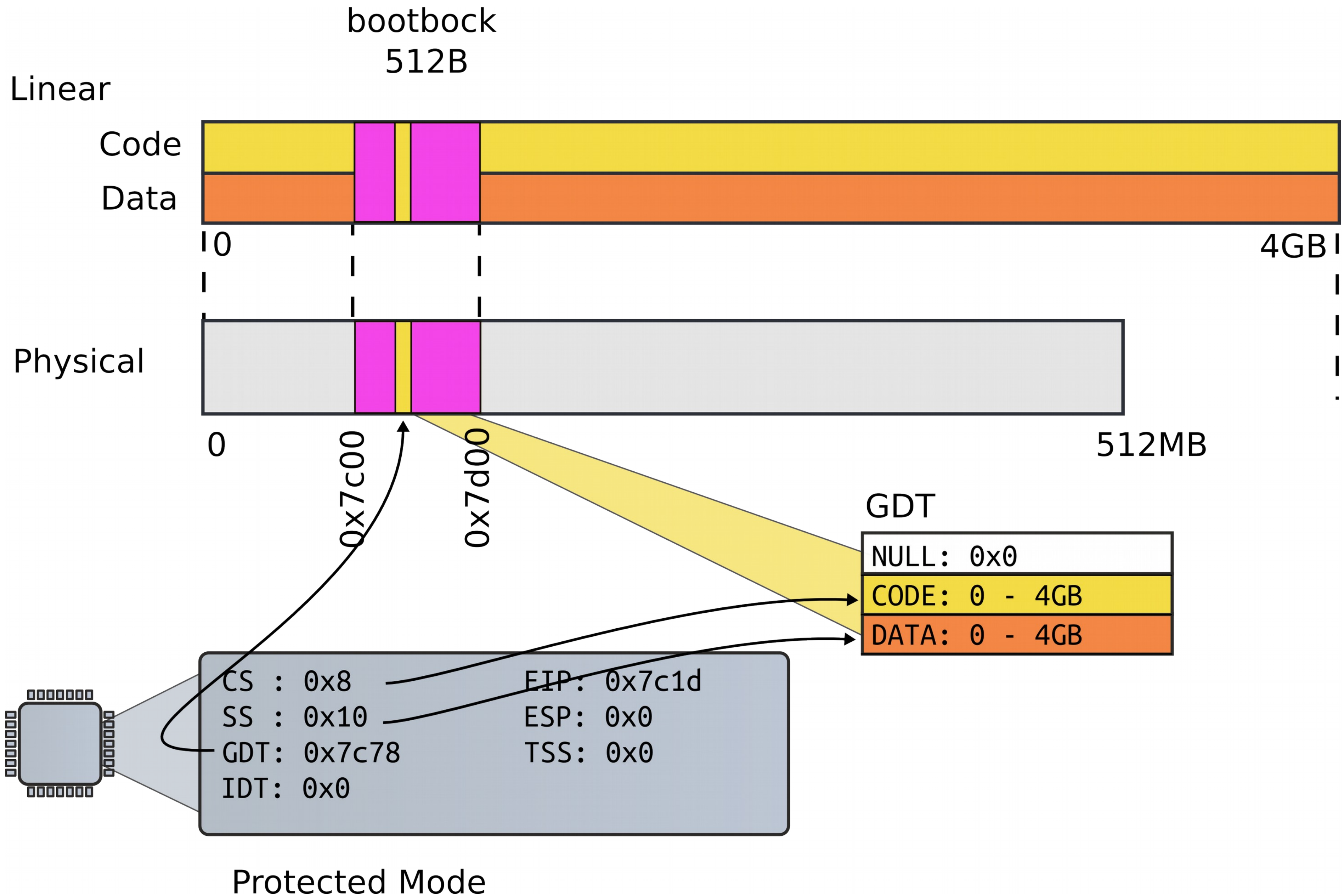

bootbock
512B
Physical
0
0x7c00
0x7d00
512MB
GDT
NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
CS : 0x8
SS : 0x0
GDT: 0x7c78
IDT: 0x0
EIP: 0x7c1d
ESP: 0x0
TSS: 0x0
Protected Mode

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?

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

xv6/bootasm.S [bootloader]

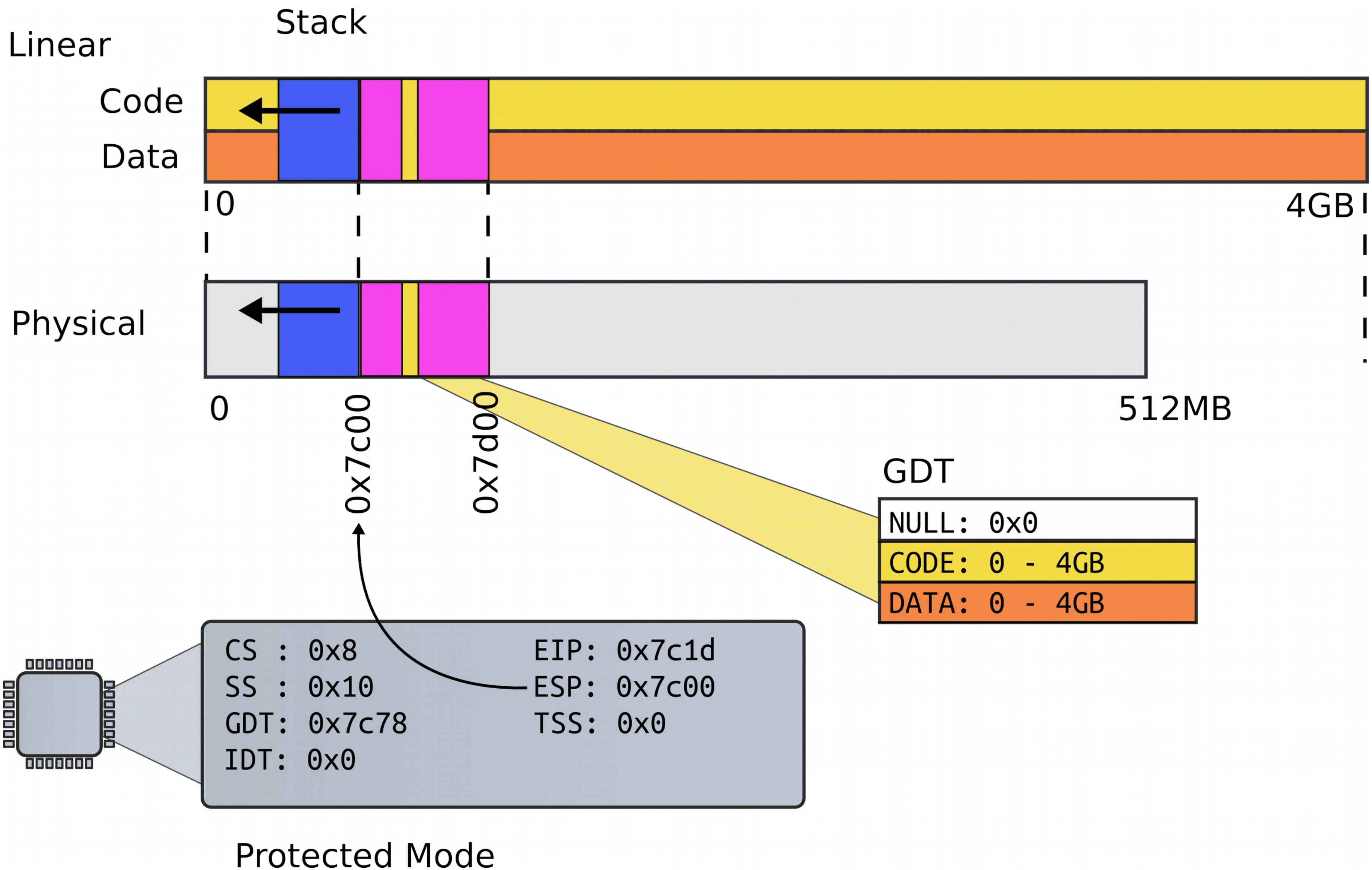
Setup stack

- 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


Stack
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: 0x7c00
TSS: 0x0
Protected Mode

Invoke first C function

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

xv6/bootasm.S [bootloader]

bootmain(): read kernel from disk

```c
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]

```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

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]

```

    // 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

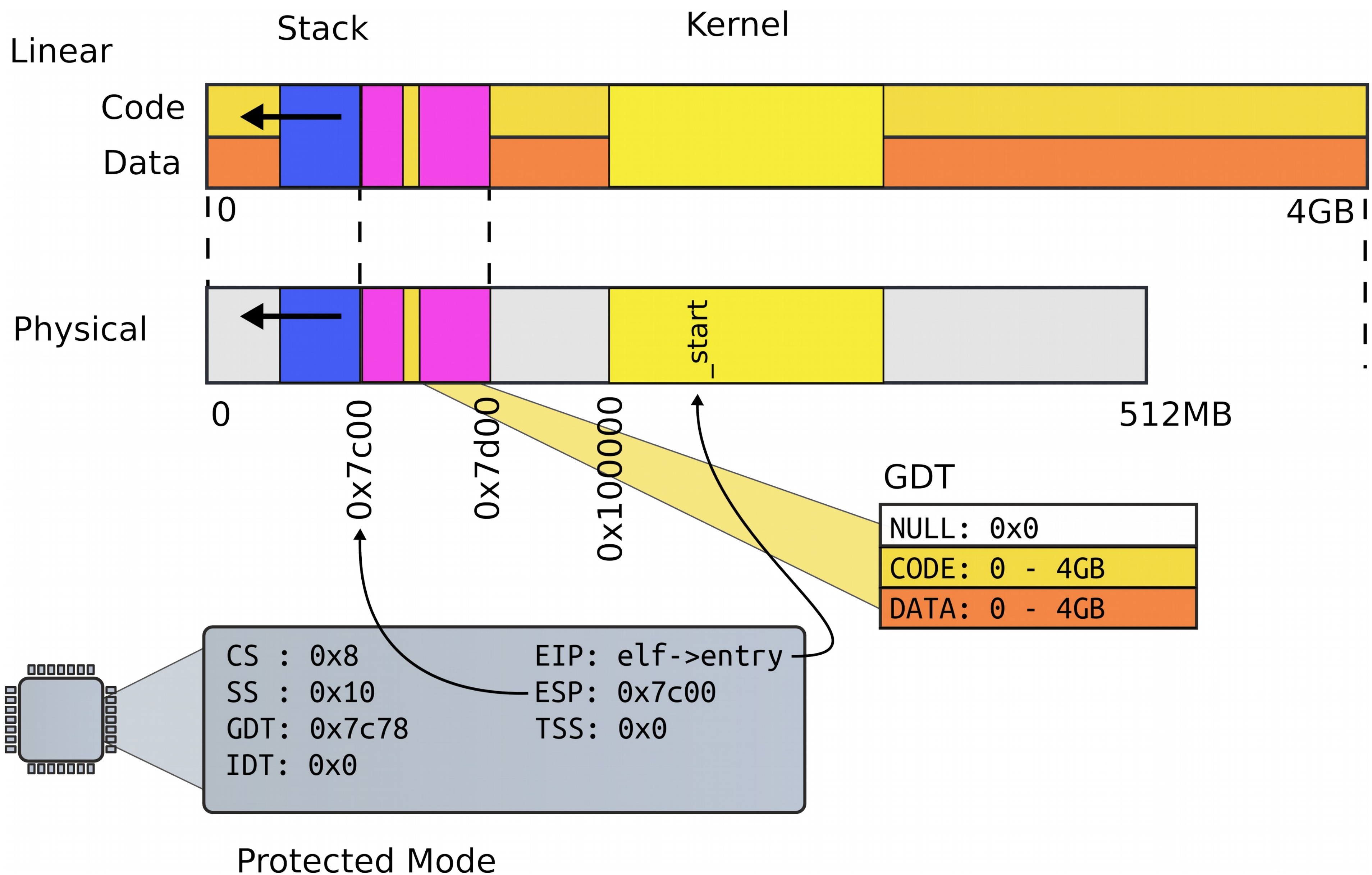
xv6/bootmain.c [bootloader]

```
.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:
# 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 [kernel]

Kernel


Stack
Kernel
Linear
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
```

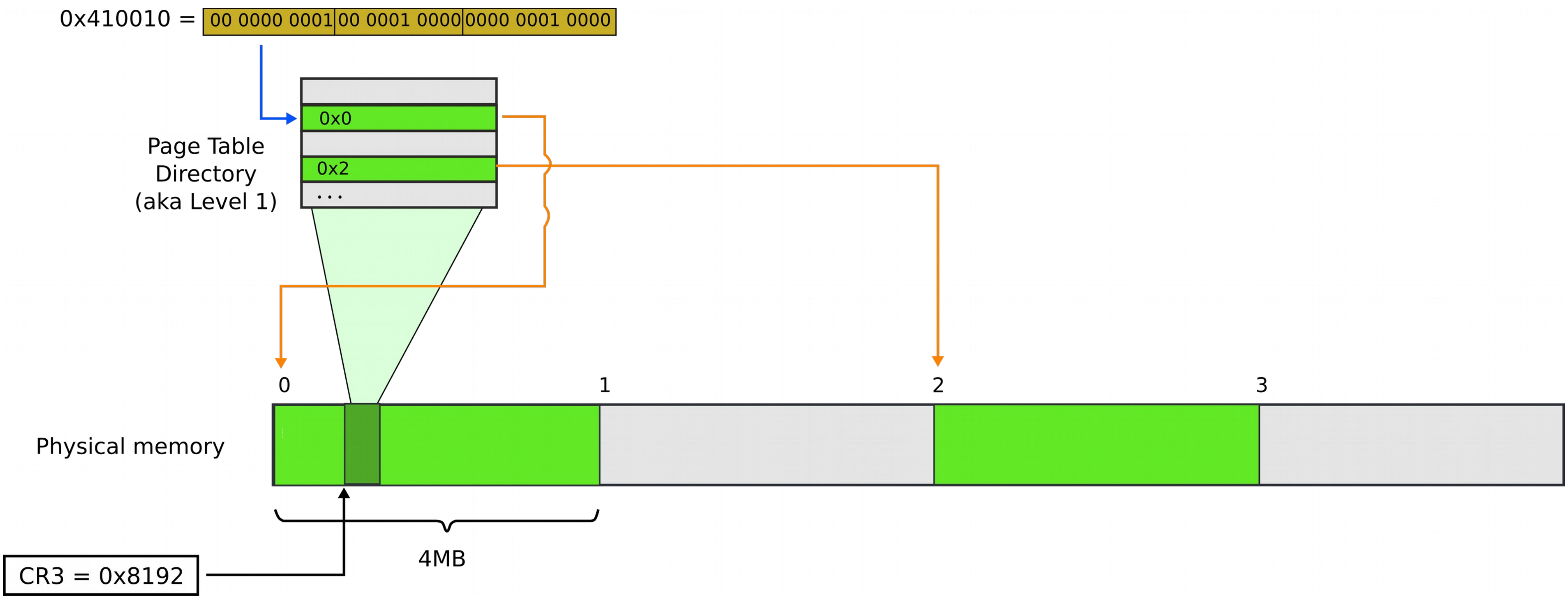
entry(): kernel ELF entry

xv6/entry.S [kernel]

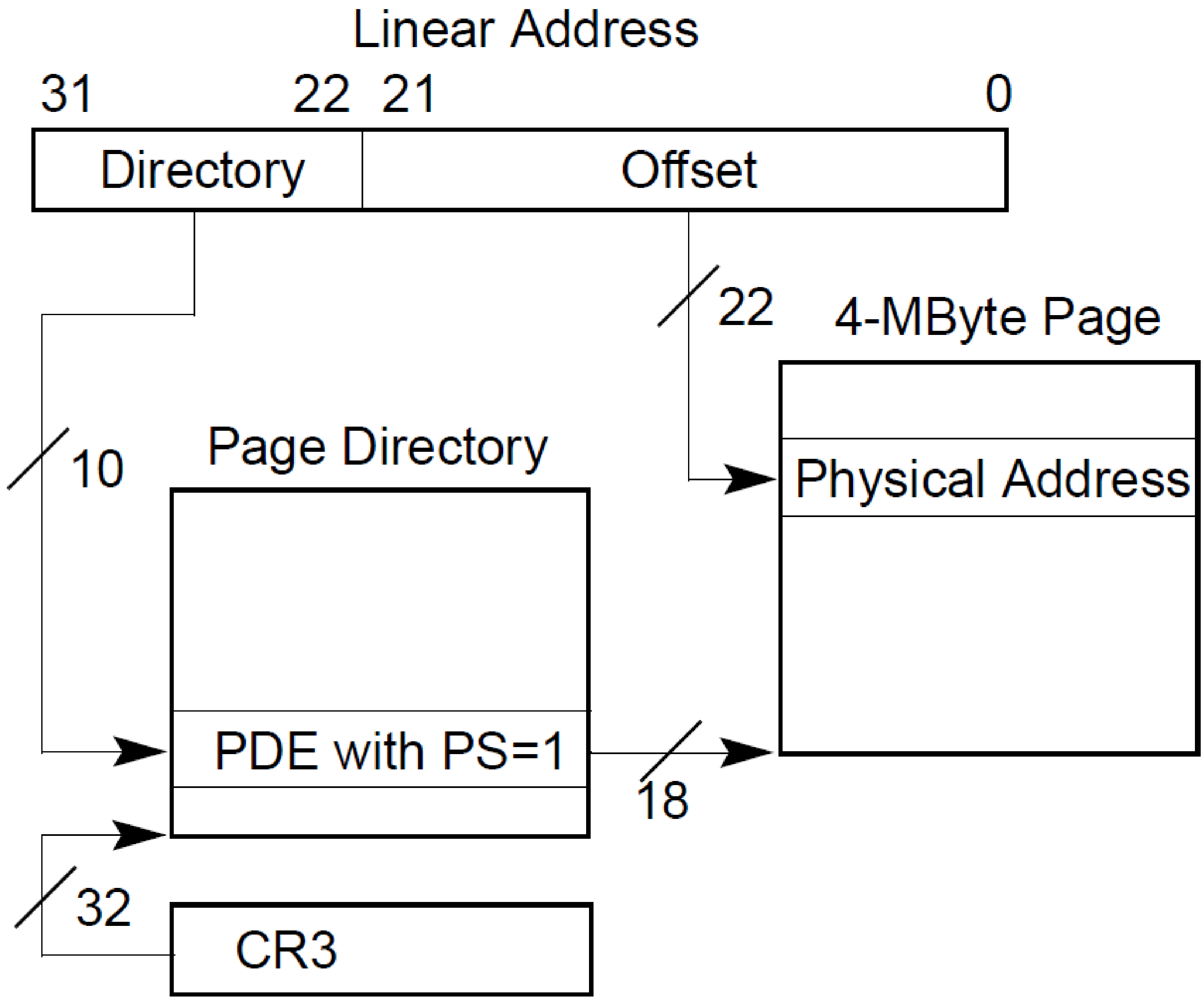
32bit x86 supports two page sizes

- 4KB pages
- 4MB pages

Page translation for 4MB pages



Page translation for 4MB pages

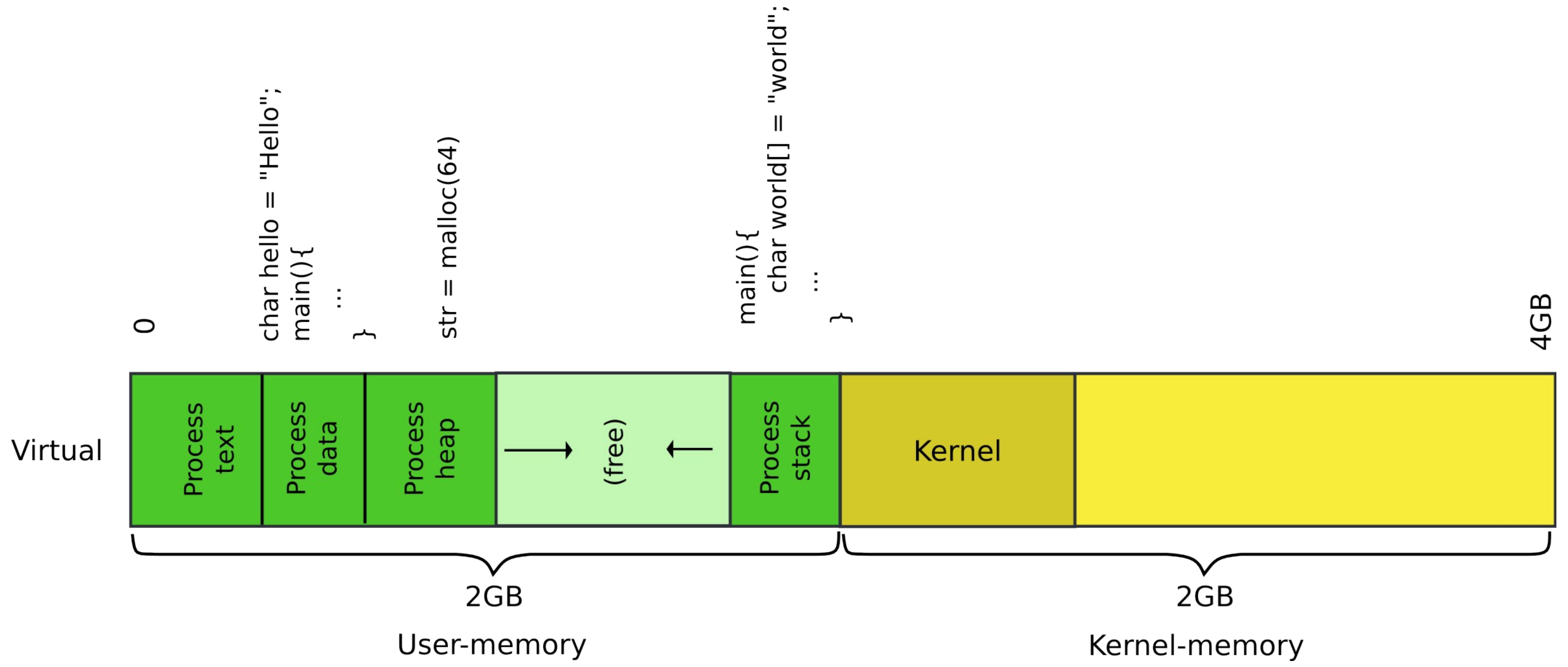


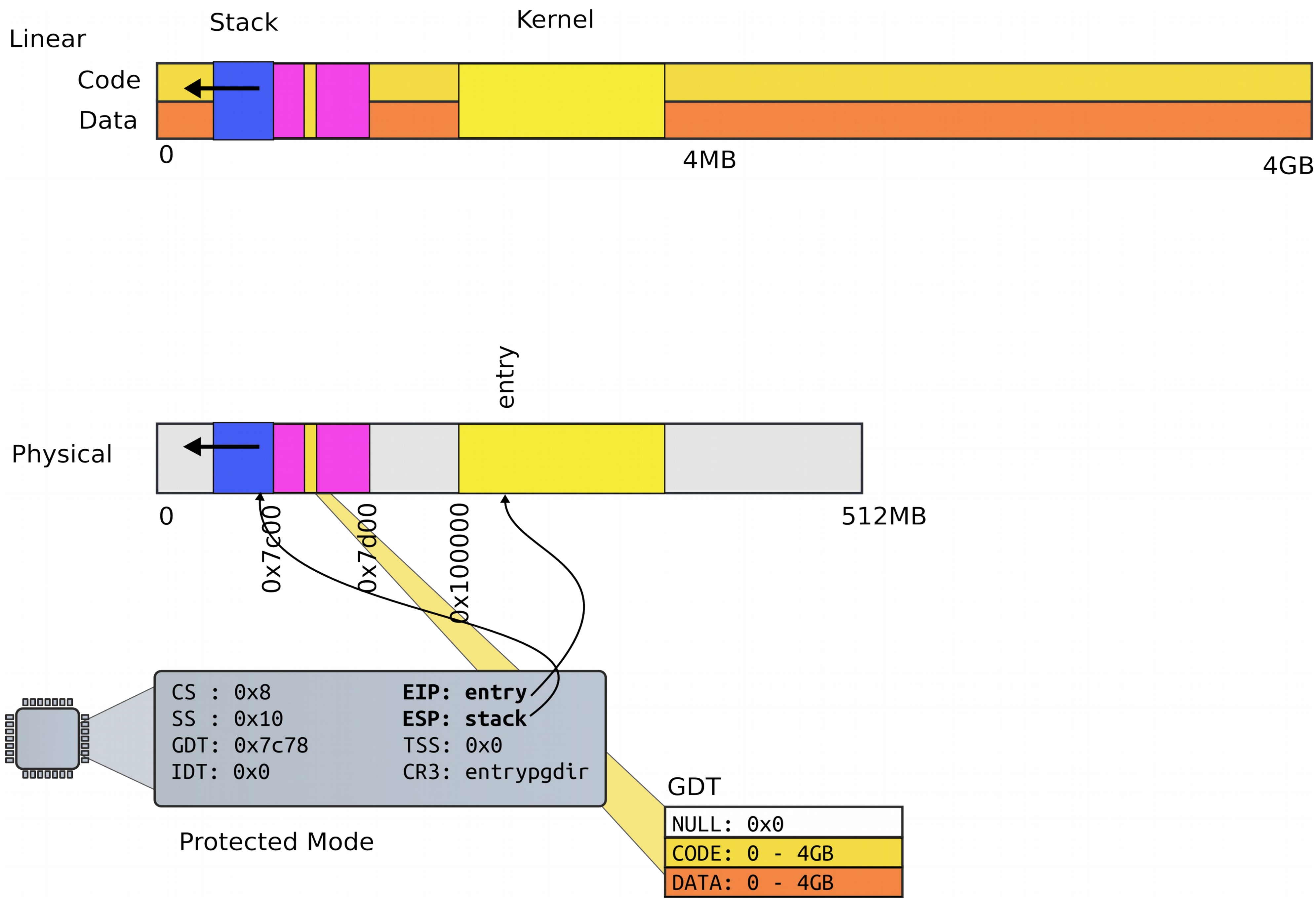
Set up page directory

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

xv6/entry.S [kernel]

Our goal: 2GB/2GB 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
GDT
NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
Protected Mode

First page table

- Two 4MB entries (large pages)
- Entry #0
  - 0x0 – 4MB → 0x0:0x400000
- Entry #512
  - 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

```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

```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

```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

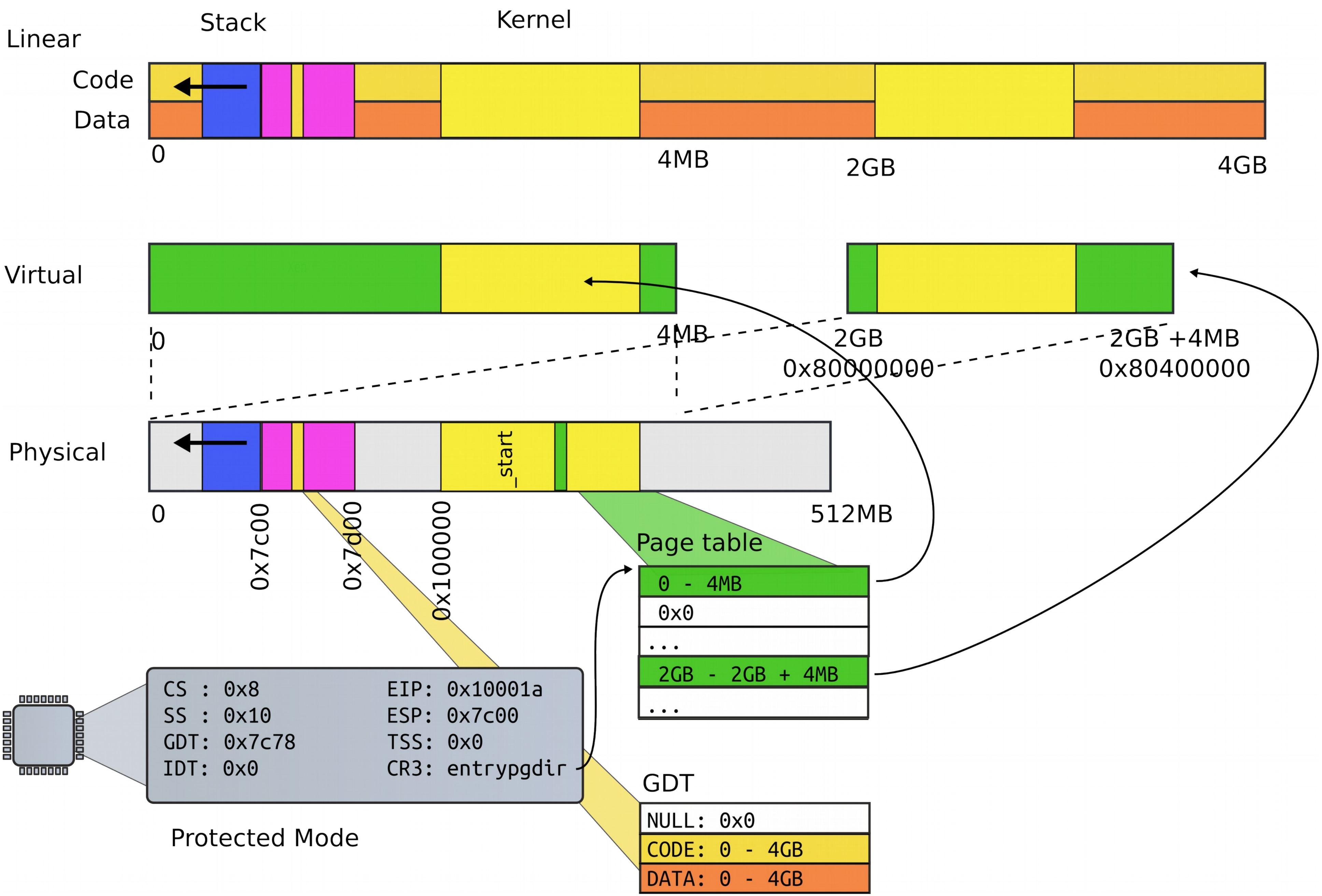
```
// The boot page table used in entry.S and entryother.S.
// Page directories (and page tables) must start on page boundaries,
// hence the __aligned__ attribute.
// PTE_PS in a page directory entry enables 4Mbyte pages.

__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



Turn on paging

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

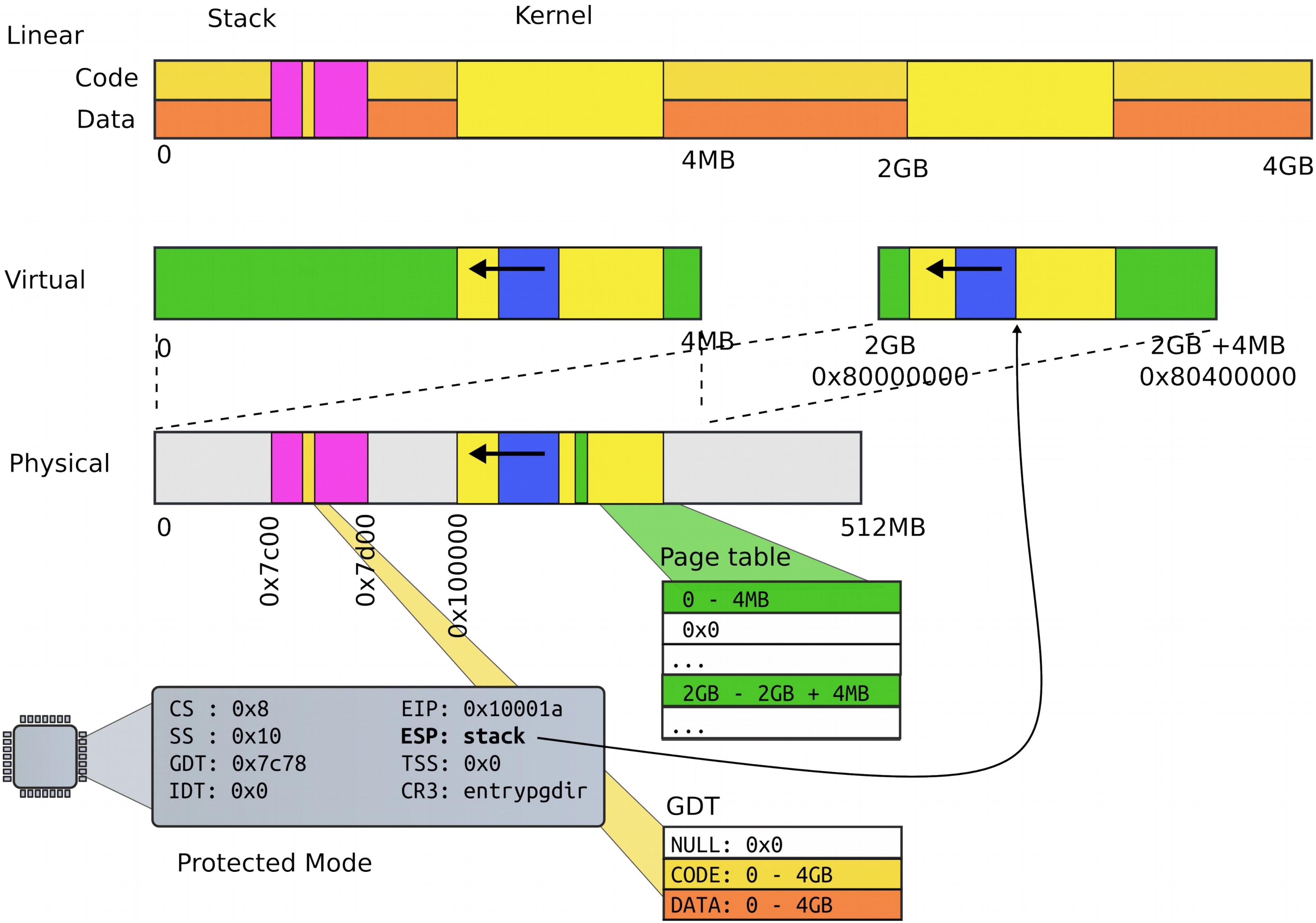
xv6/entry.S [kernel]

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
```

xv6/entry.S [kernel]

High address stack (4K)



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
```

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]

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

Function pointers

```c
#include <stdio.h>

void func_a(void){
    printf("func_a\n");
    return;
}

void func_b(void) {
    printf("func_b\n");
     return;
}

int main(int ac, char **av)
{
    void (*fp)(void);

    fp = func_b;
    fp();
    return;
}
```

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    $0xfffffff0,%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    $0xfffffff0,%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
```