Gabriele **Proietti Mattia**

Advanced Operating Systems and Virtualization

[1] The x86 Boot Process

Department of Computer, Control and Management Engineering "A. Ruberti", Sapienza University of Rome

gpm.name · proiettimattia@diag.uniroma1.it

A.Y. 2020/2021 · v1

Outline

1. **BIOS/UEFI**
   Actual hardware setup
2. **Bootloader Stage 1**
   Executes the stage 2 bootloader (skipped for UEFI)
3. **Bootloader Stage 2**
   Loads and starts the kernel
4. **Kernel**
   Takes control and initializes the machine (machine-dependent operations)
5. **Init (or systemd)**
   First process: basic environment initialization
6. **Runlevels/Targets**
   Initializes the user environment

Outline

[1] The x86 Boot Process

1. **Step1: BIOS/UEFI**
    1. Pre-Boot and Real Mode
    2. BIOS
2. **Step 2: Stage 1 Bootloader**
    1. MBR
    2. x86 Protected Mode
    3. x86 Memory Addressing
    4. x86 Privileges and Protection
    5. Paging
3. **Step 3: Stage 2 Bootloader**
    1. GRUB/UEFI
    2. Multi-core Support

1.1

1. x86 Boot Process

Step 1: BIOS/UEFI

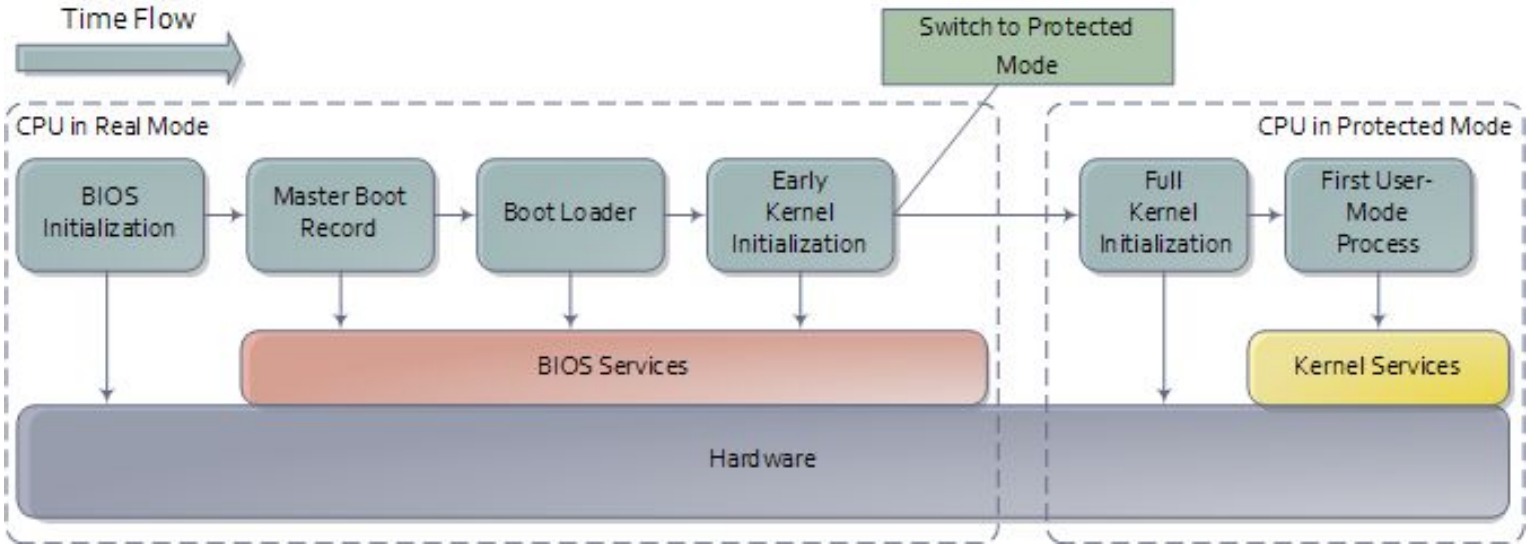
DIAG

Boot sequence

1. **BIOS/UEFI**
   Actual hardware setup
2. **Bootloader Stage 1**
   Executes the stage 2 bootloader (skipped for UEFI)
3. **Bootloader Stage 2**
   Loads and starts the kernel
4. **Kernel**
   Takes control and initializes the machine (machine-dependent operations)
5. **Init (or systemd)**
   First process: basic environment initialization
6. **Runlevels/Targets**
   Initializes the user environment

Boot Sequence

CPU Stages & Operations



https://manybutfinite.com/post/how-computers-boot-up/

1.1.1

1. x86 Boot Process
1. Step 1: BIOS/UEFI

Pre-Boot and Real Mode

DIAG

The Pre-Pre-Boot

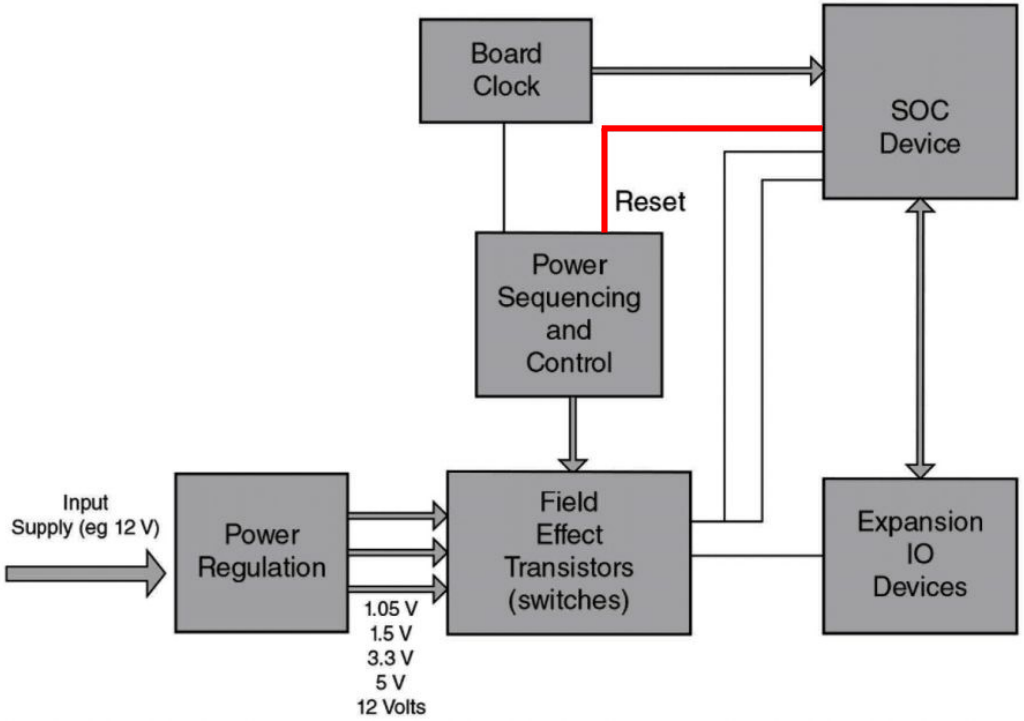
When the power button is pushed, the CPU **does not start directly** to run code (BIOS).

There are many operations that must be carried out before doing that:

- the power supply must settle down to its nominal state
- a number of derived voltages must stabilize: 1.5V, 3.3V, 5V and 12V. These voltages must be supplied in a particular sequence, this is called **power sequencing** and it is carried out by a CPLD (Complex Programmable Logic Device)
- platform clocks must be derived and this takes time

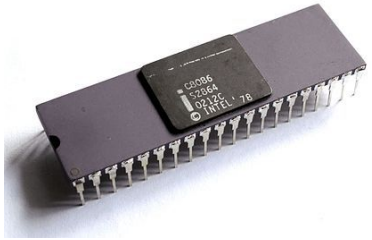
The Pre-Pre-Boot

Once the tasks have been carried out, the CLPD de-assert the reset line of the CPU.



Dice, Pete. *Quick boot: a guide for embedded firmware developers*. Walter de Gruyter GmbH & Co KG, 2017.

x86 Real Mode



Intel 8086

At this point the system is in a very basic state:

- caches are disabled
- the MMU (Memory Management Unit) is disabled
- only one CPU core can run the code (the BSP - bootstrap processor)
- the CPU runs in **Real Mode,** a compatible way with the Intel 8086 (1978, yes 1978)
- nothing is in RAM

x86 Real Mode is characterized by:

- no memory protection, no privilege levels, no multitasking
- direct access to I/O and peripheral
- memory:
  - 20 bit of a **segmented memory space** for a total of 1MB of addressable memory
  - 16 bit for instructions

Segmented Memory

Starting from the Intel 8086, the addressing of memory is segmented. This means that a memory location is referenced with two components: the segment id and the offset. Therefore, the **logical address** can be expressed as:

<seg:offset> (e.g. <A:0x10>)

There are 4 basic **16-bit** segment registers:

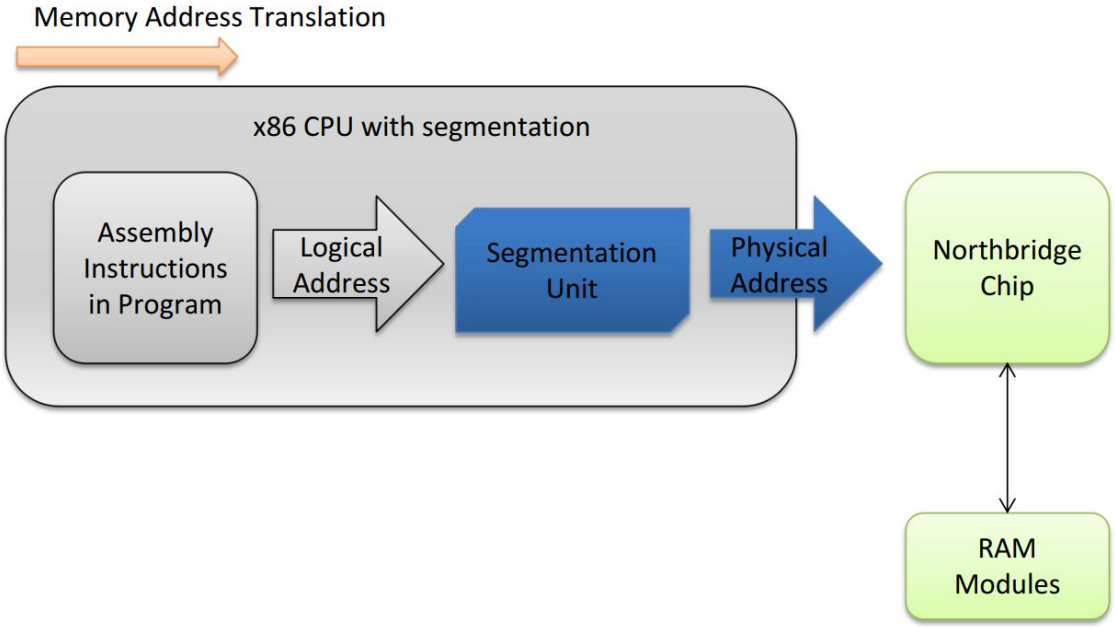
- **CS**: Code Segment
- **DS**: Data Segment
- **SS**: Stack Segment
- **ES**: Extra Segment (that can be used by the programmer)

Intel 80386 added also other two registers, FS and GS with no predefined usage.

Segmented Memory

Address Resolution

The CPU resolves addresses in the following way



Segmented Memory

Segmentation is still present nowadays and it is **always enabled**. Each assembly instruction that uses memory implicitly uses a segment register, for example:

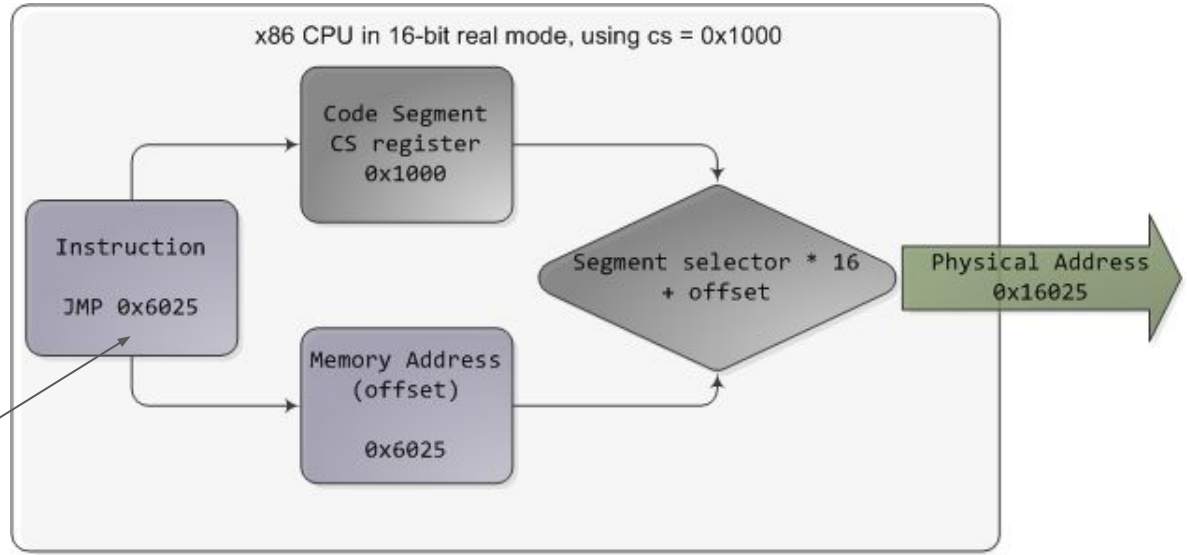
- a `jmp` uses CS
- a `push` uses SS

Most of the segment addresses can be loaded with mov instruction but CS only with `jmp` or `call`.

x86 Real Mode

Address Resolution

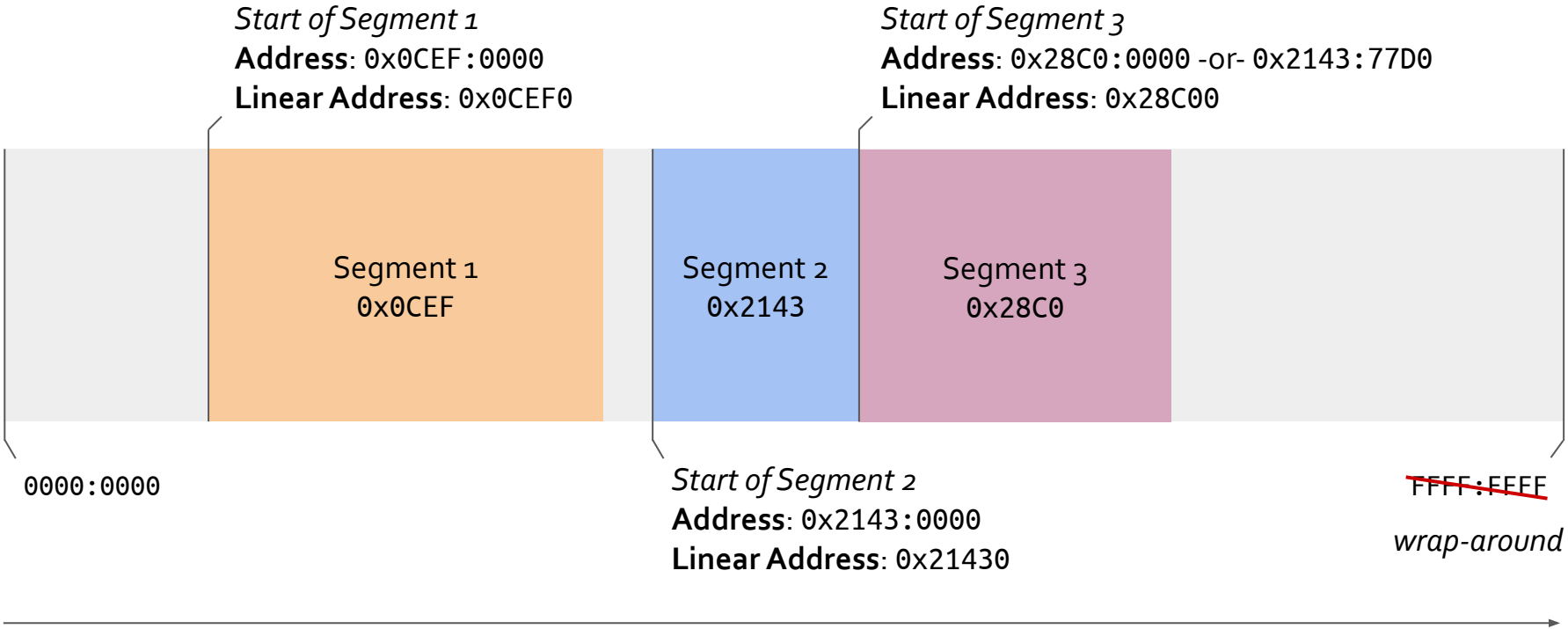
```
  0000 0110 1110 1111 0000 Segment, 16 bits, shifted 4 bits left (or multiplied by 0x10)
+      0001 0010 0011 0100 Offset,    16 bits
-----------------------------------
  0000 1000 0001 0010 0100 Address, 20 bits
```



https://manybutfinite.com/post/memory-translation-and-segmentation

x86 Real Mode

Segmented Memory



1.1.2

1. x86 Boot Process

1. Step 1: BIOS/UEFI

BIOS

DIAG

The First Fetched Instruction

Once the CLPD de-assert the reset line, newer processors load a microcode update for example for patching vulnerabilities. This, obviously, must be done before executing any program. After that the CPU starts executing instructions located at a precise memory address, called the **reset vector**. For Intel x86 the reset vector is at:

```
0xF000:FFF0
```

Only 16 bytes from the top memory boundary. On IBM PCs that specific memory area is bound to a ROM, the so-called **BIOS**. The first fetched instruction is

```
ljmp $0xf000,$0xe05b
```

This starts the actual BIOS code, the long-jump also sets CS to `0xf0000`

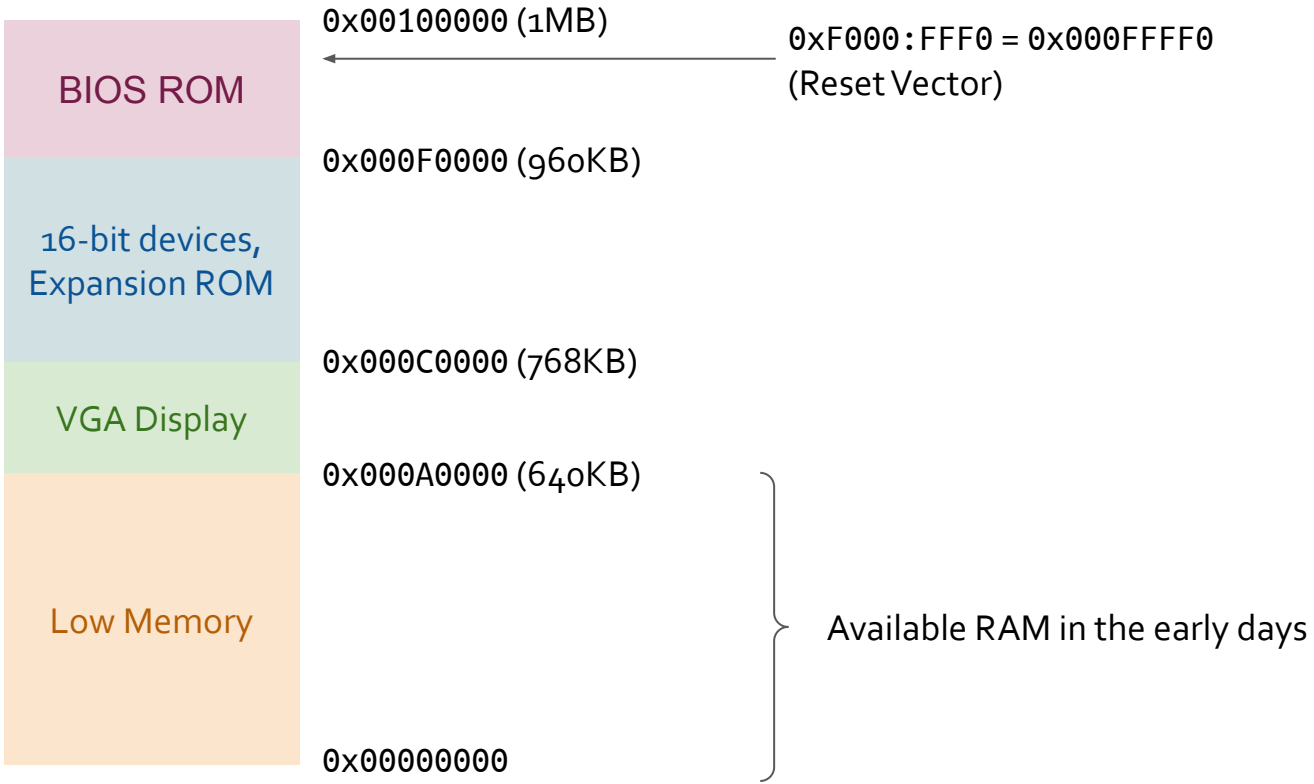
BIOS Operations

The usual operations carried out by the BIOS are:

- looking for **video adapters** that need to be run specific routines, these ROMs are mapped from C000:0000 to C780:0000
- **POST** (Power-on Self-Test) does peripheral check (mouse, keyboard), also checks RAM consistency, initialize the Video Card
- loads the **boot order configuration**, from the CMOS (64bytes)
- **copying** itself in RAM for a faster access (*shadowing*)
- **identifying** the Stage 1 Bootloader (512bytes) using the specified boot order and loading it in RAM at address `0000:7c00`
- finally the **control** is given with the instruction `ljmp $0x0000,$0x7c00`

BIOS Operations

RAM after BIOS startup



1.2

1. x86 Boot Process

Step 2: Stage 1 Bootloader

DIAG

Boot sequence

1. **BIOS/UEFI**
   Actual hardware setup
2. **Bootloader Stage 1**
   Executes the stage 2 bootloader (skipped for UEFI)
3. **Bootloader Stage 2**
   Loads and starts the kernel
4. **Kernel**
   Takes control and initializes the machine (machine-dependent operations)
5. **Init (or systemd)**
   First process: basic environment initialization
6. **Runlevels/Targets**
   Initializes the user environment

1.2.1

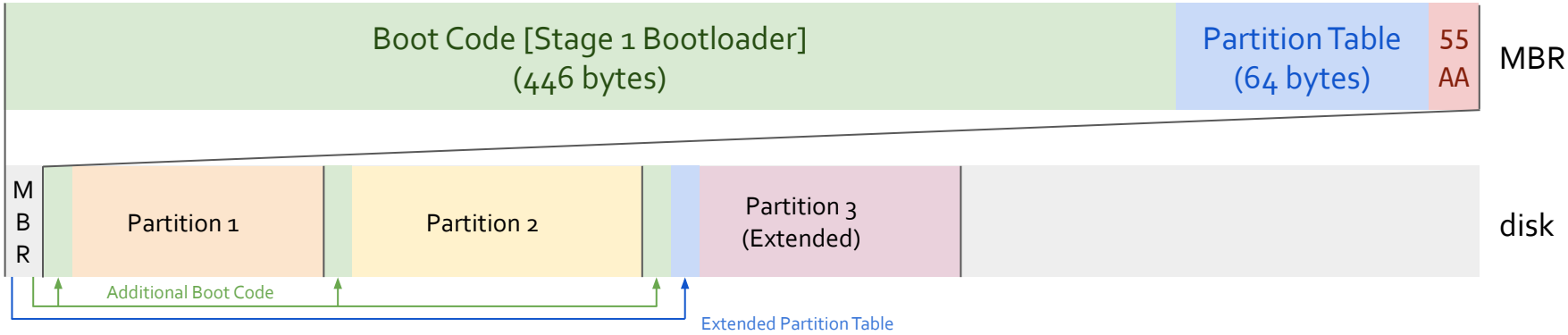
1. x86 Boot Process
2. Step 2: Stage 1 Bootloader

MBR

DIAG

The Master Boot Record (MBR)

The first sector (512 bytes) of the disk contains the Master Boot Record, which stores **executable code** and the **partition table** of the disk.



Partition Table

The partition table contains up to 4 entries but it can be extended to multiple sectors of the disk in order to address more partitions.

Nowadays, with UEFI, MBR has been replaced with GPT which will we see later.

The Master Boot Record (MBR)

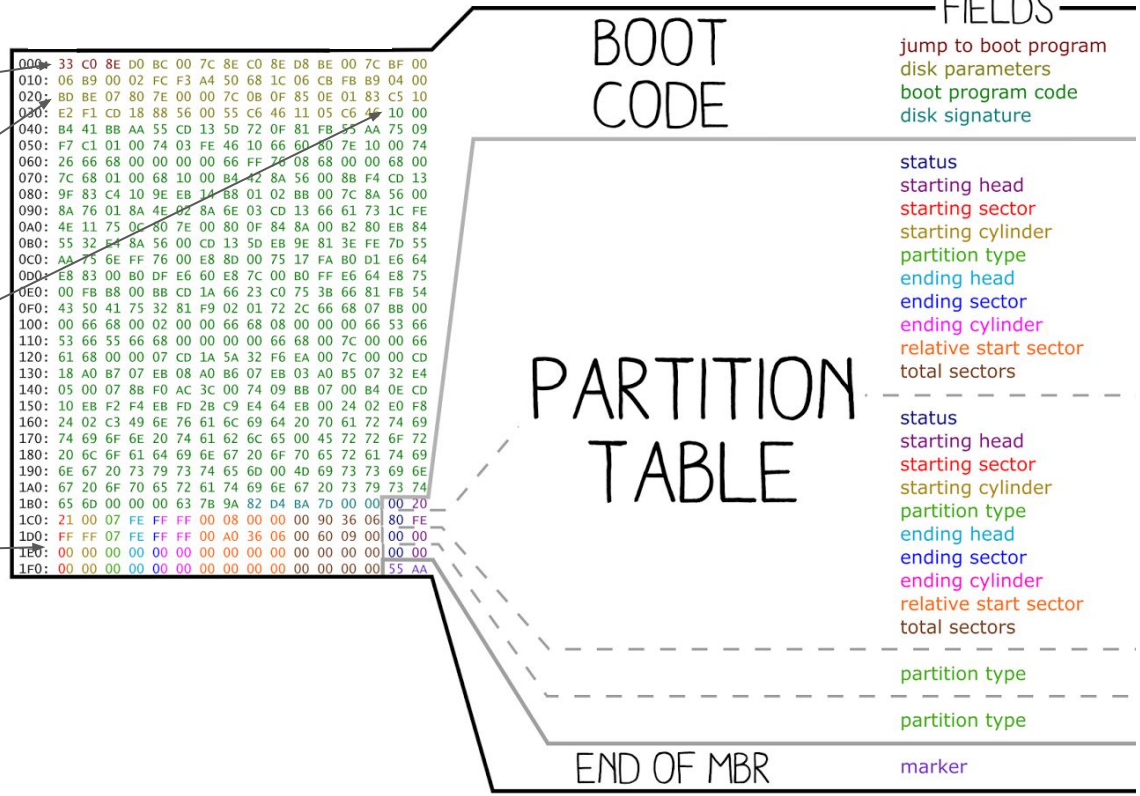
Anatomy

This is a **jump** to the executable code

This **area** is the BIOS Parameter Block (BPB) a data structure that contains the **physical data of the disk**

Executable **code** starts here

Each partition is described with 16 bytes of data, we have space for 4 partitions



Moral of the story

We have only a **384 bytes** program for starting the OS!

The Master Boot Record (MBR)

The actual code

```
.code16
.text

.globl _start;

_start:
jmp stage1_start

OEMLabel:  .string "BOOT"
BytesPerSector:  .short 512
SectorsPerCluster: .byte 1
ReservedForBoot: .short 1
NumberOfFats:  .byte 2
RootDirEntries:  .short 224
LogicalSectors:  .short 2880
MediumByte:  .byte 0x0F0
SectorsPerFat:  .short 9
SectorsPerTrack: .short 18
Sides:  .short 2
HiddenSectors:  .int 0
LargeSectors:  .int 0
DriveNo:  .short 0
Signature:  .byte 41   #41 = floppy
VolumeID:  .int 0x00000000   # any value
VolumeLabel:  .string "myOS       "
FileSystem:  .string "FAT12   "

.stage1_start:
cli # Disable interrupts
xorw %ax,%ax # Segment zero
movw %ax,%ds
movw %ax,%es
movw %ax,%ss

...
```

http://web.archive.org/web/20200607220642/http://polytimenerd.blogspot.com/2012/06/write-your-own-kernel-bootloader-stub.html

Stage 1 Bootloader

Tasks

The stage 1 bootloader must:

- enable the **A20 line**
- switch to **32-bit protected mode**
- setup a **stack**
- load the kernel, but for doing that we need to navigate the filesystem so this must be done by the *Stage 2 bootloader*

Stage 1 Bootloader

The A20 line

The intel 80286, the successor of the 8086, increased the addressable memory to 16MB, that means **24 bits** for addresses.

For maintaining the compatibility with the programs written for the 8086 the 21th bit is forced to zero, in this way, the memory "wrap-around" when exceeds the 1MB limit. For example:

`0xF800:8000` ⟶ `0x00100000`

⟶ `0x0000 0000 0001 0000 0000 0000 0000 0000`

By forcing the 21th to 0 (line A20) the address starts from the beginning of the memory

⟶ `0x0000 0000 0000 0000 0000 0000 0000 0000`

Stage 1 Bootloader

The A20 line

For forcing the A20 to zero the IBM decided to make a modification on the motherboard, in particular by using a a spare pin of the 8042 keyboard controller. The pin has been routed to the A20 line, so called Gate A20.

The A20 is disabled by default when the CPU starts and it must be **enabled** before entering in *protected mode*.

```
call wait_for_8042
movb $0xd1, %al #command write
outb %al, $0x64
call wait_for_8042
movb $0xdf, %al # Enable A20
outb %al, $0x60
call wait_for_8042
```

```
wait_for_8042:
  inb %al, $0x64
  testb $2,%al # Bit2 set=busy
  jnz wait_for_8042
  ret
```

1.2.2

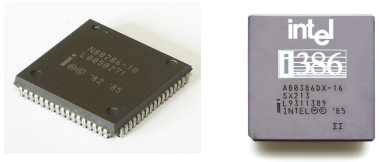
1. x86 Boot Process
2. Step 2: Stage 1 Bootloader

x86 Protected Mode

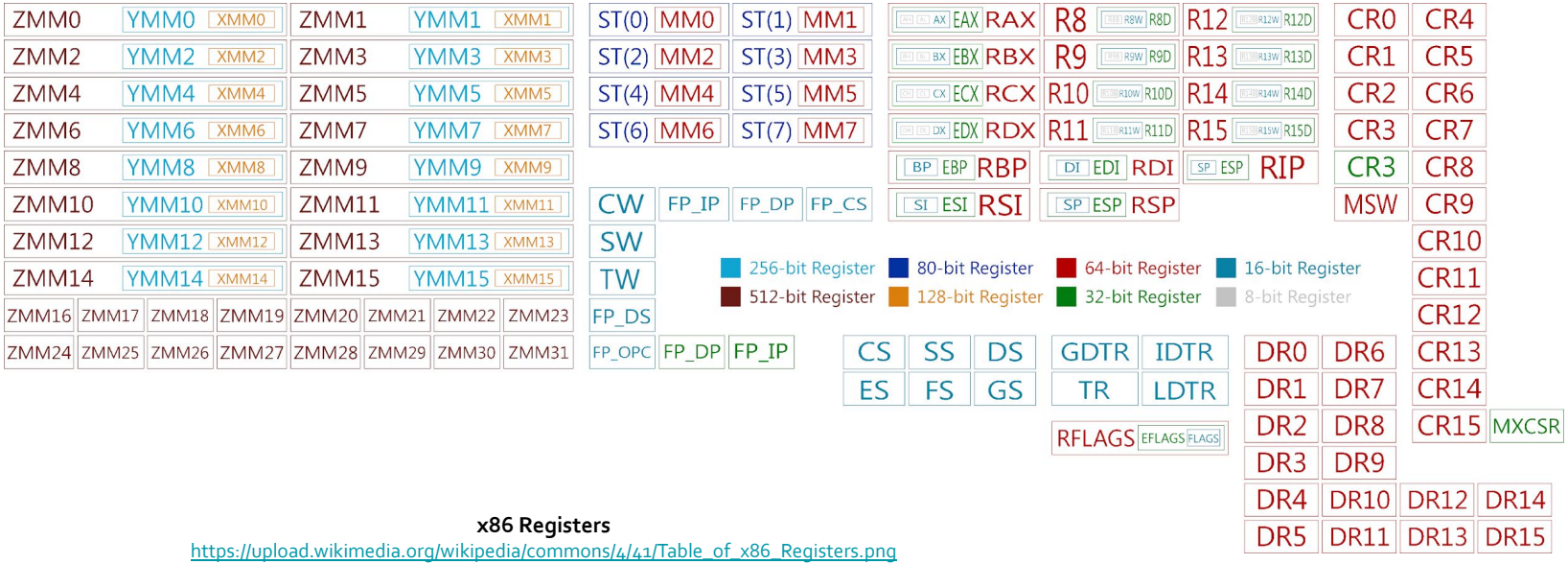
DIAG

x86 Protected Mode

The x86 protected mode was introduced with the 80286 (1982) and it was extended with memory paging in the 80386 (1985). Still today, modern PCs starts in Real Mode for backward compatibility, therefore the Protected Mode **must be enabled** during the startup.



From left, Intel 80286 and 80386



x86 Registers

https://upload.wikimedia.org/wikipedia/commons/4/41/Table_of_x86_Registers.png

x86 Protected Mode

CR0 Register

The CR0 register is 32 bits long on the 386 and higher processors. On x64 processors in long mode, it (and the other control registers) is 64 bits long. CRo has various control flags that modify the basic operation of the processor.

	Bit	Name	Full Name	Description
→	0	PE	Protected Mode Enable	If 1, system is in protected mode, else system is in real mode
	1	MP	Monitor co-processor	Controls interaction of WAIT/FWAIT instructions with TS flag in CR0
	2	EM	Emulation	If set, no x87 floating-point unit present, if clear, x87 FPU present
	3	TS	Task switched	Allows saving x87 task context upon a task switch only after x87 instruction used
	4	ET	Extension type	On the 386, it allowed to specify whether the external math coprocessor was an 80287 or 80387
	5	NE	Numeric error	Enable internal x87 floating point error reporting when set, else enables PC style x87 error detection
→	16	WP	Write protect	When set, the CPU can't write to read-only pages when privilege level is 0
	18	AM	Alignment mask	Alignment check enabled if AM set, AC flag (in EFLAGS register) set, and privilege level is 3
	29	NW	Not-write through	Globally enables/disable write-through caching
→	30	CD	Cache disable	Globally enables/disable the memory cache
→	31	PG	Paging	If 1, enable paging and use the § CR3 register, else disable paging.

https://en.wikipedia.org/wiki/Control_register

x86 Protected Mode

Entering Basic Protected Mode

The first action to do for entering protected mode is to set the bit 0 (PE) of CR0 to 1, but this is not enough for enabling all of the facilities. We need to set the CS and the only way to do this is to use a far jump (`ljmp`), then the code will execute in 32/64 bit mode.

```
ljmp 0x0000, PE_mode

.code32

PE_mode:
 # Set up the protected-mode data segment registers
 movw $PROT_MODE_DSEG, %ax
 movw %ax, %ds
 movw %ax, %es
 movw %ax, %fs
 movw %ax, %gs
 movw %ax, %ss
```

1.2.3

1. x86 Boot Process
2. Step 2: Stage 1 Bootloader

x86 Memory Addressing

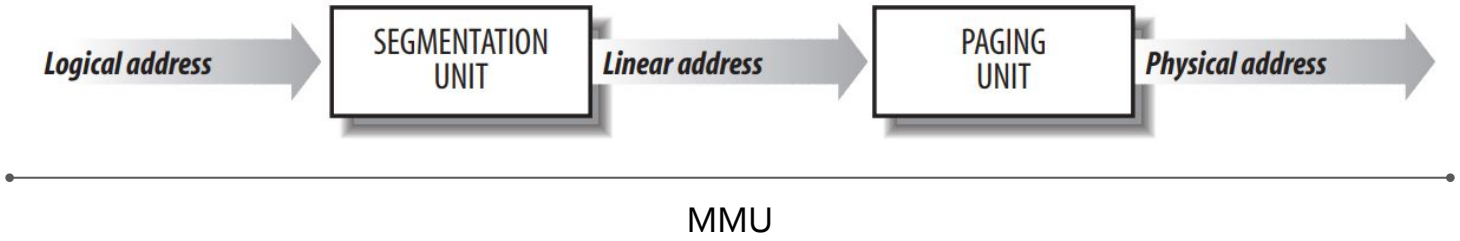
DIAG

Memory Addresses

The 8086 defined three kinds of memory addresses:

- a **logical address** that is used in the ASM code is always composed by two parts: a *segment* (selector) and an *offset* within the segment (e.g. `0xFFFF:FFFF`)
- a **linear address** that in a 32bit architecture is a 32bit unsigned integer and can be used to address up to 4GB (e.g. `0x00000000` - `0xFFFFFFFF`)
- a **physical address** that is used to address memory cells in memory chips, they correspond to the electrical signal sent along the address pins of the cpu to the memory bus.

Address are translated by the MMU (Memory Management Unit) set of circuits.



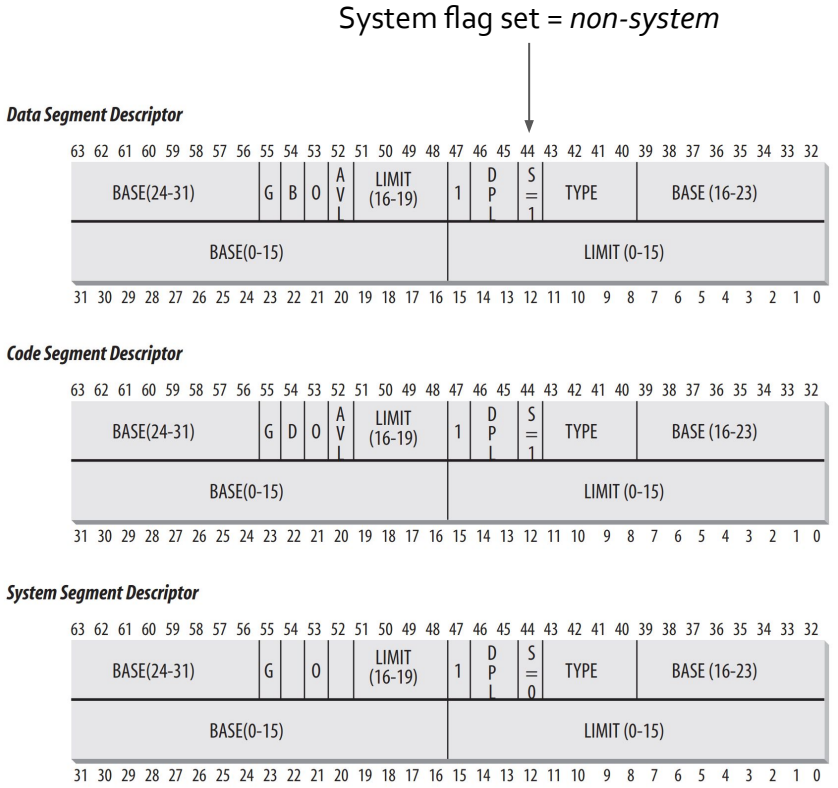
Segmentation

Segments Descriptors

In protected mode a segment in no longer a raw number but it contains an index to a table of **segment descriptors**. The table is an array containing 8-byte records of this kind:

There are three type of segments: code, data and system. The main sections are:

- **Base,** a 32-bit linear address that pointing to the beginning of the segment
- **Limit,** the size of the segment
- **G,** the granularity (if `0` size is bytes otherwise it is a multiple of 4096)
- **DPL,** the descriptor privilege a number from 0 to 3 to control the access to the segment



Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

Privileges

Privileges

Ring 3 has **restricted access** to memory management, instructions execution and I/O ports

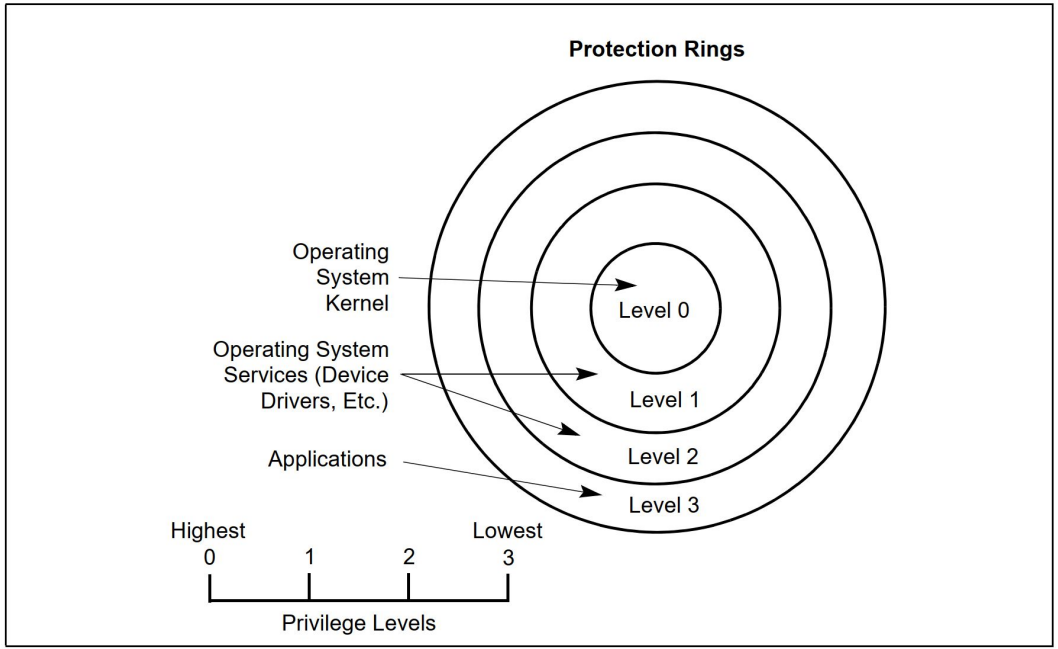


Figure 6-4. Protection Rings

Figures with blue caption are from the latest version of the Intel Manual

Segmentation

Segments Descriptors Tables and Selectors

The Segment Descriptors are stored either in:

- the **Global Descriptor Table** (GDT) that is system wide and pointed by the register GDTR (with the size)
- the **Local Descriptor Table** (LDT) that was specific for one process and it was pointed by the register LDTR (with the size), today is not used anymore

Segment Selectors

Each segment register (CS, DS, SS, FS, GS), contains a *Segment Selector* (16bit). Beside of the index to the GDT also contains TI (the table indicator 0/1 = GDT/LDT) and the **RPL** that we will see later. Remember that a logical address is a segment selector + offset.

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

Segmentation

In the Linux Kernel

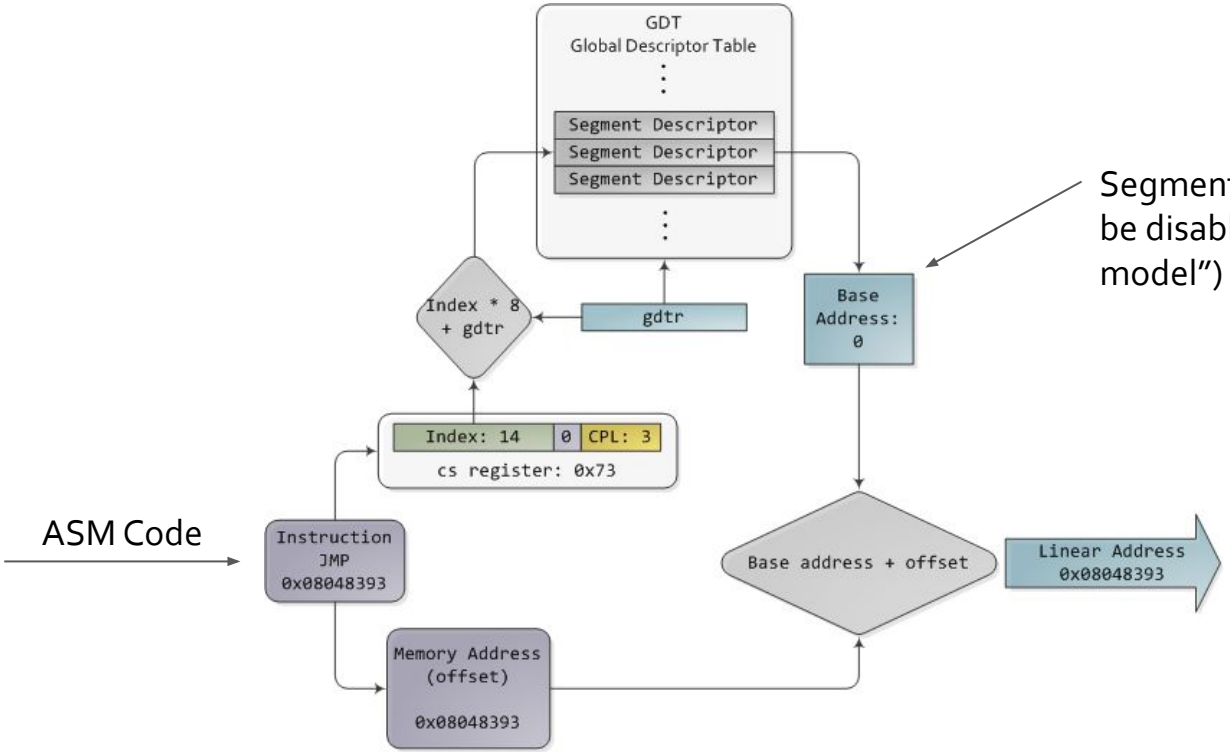
In the Linux kernel segmentation is redundant and used in very limited way, since paging is favoured. All Linux Processes running is User Mode use the user code segment (__USER_CS) and the user data segment (__USER_DS), the ones that runs in Kernel Mode uses the kernel code segment (__KERNEL_CS) and the kernel data segment (__KERNEL_DS). All of these segments have base 0 and max limit, therefore all processes **may use** the same logical addresses and **coincide** with the linear addresses.

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xfffff	1	10	3	1	1
user data	0x00000000	1	0xfffff	1	2	3	1	1
kernel code	0x00000000	1	0xfffff	1	10	0	1	1
kernel data	0x00000000	1	0xfffff	1	2	0	1	1

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

Segmentation

Logical to Linear Resolution



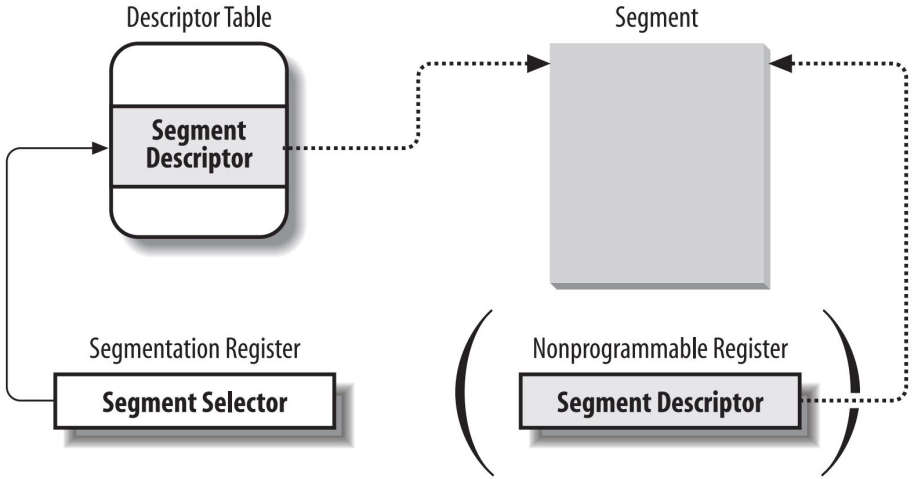
Segmentation cannot be disabled (Intel "flat model")

https://manybutfinite.com/post/memory-translation-and-segmentation/

Segmentation

Caching

Accessing the GDT every time an address has to be translated is not performance-wise. For this reason the 8086 provides an additional **non-programmable register** (for every segment register) which contains the last resolved 8byte Segment Descriptor.



Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

1.2.4

1. x86 Boot Process
2. Step 2: Stage 1 Bootloader

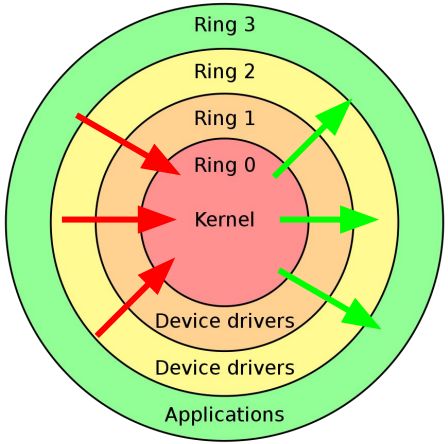
x86 Privileges and Protection

DIAG

Privileges and Protection

We have seen that each S. Descriptor has a **DPL** (Descriptor Privilege Level), each S. Selector has an **RPL** (Requestor Privilege Level). We also need a current execution privilege level (**CPL**), that describes the current privileges that the CPU has.

Now, how the memory protection is enforced by using these metadata? And how we can change our current privilege level?



Decreasing the ring level should be **denied** or **controlled**

Increasing the ring level should be **allowed**

Privileges Levels

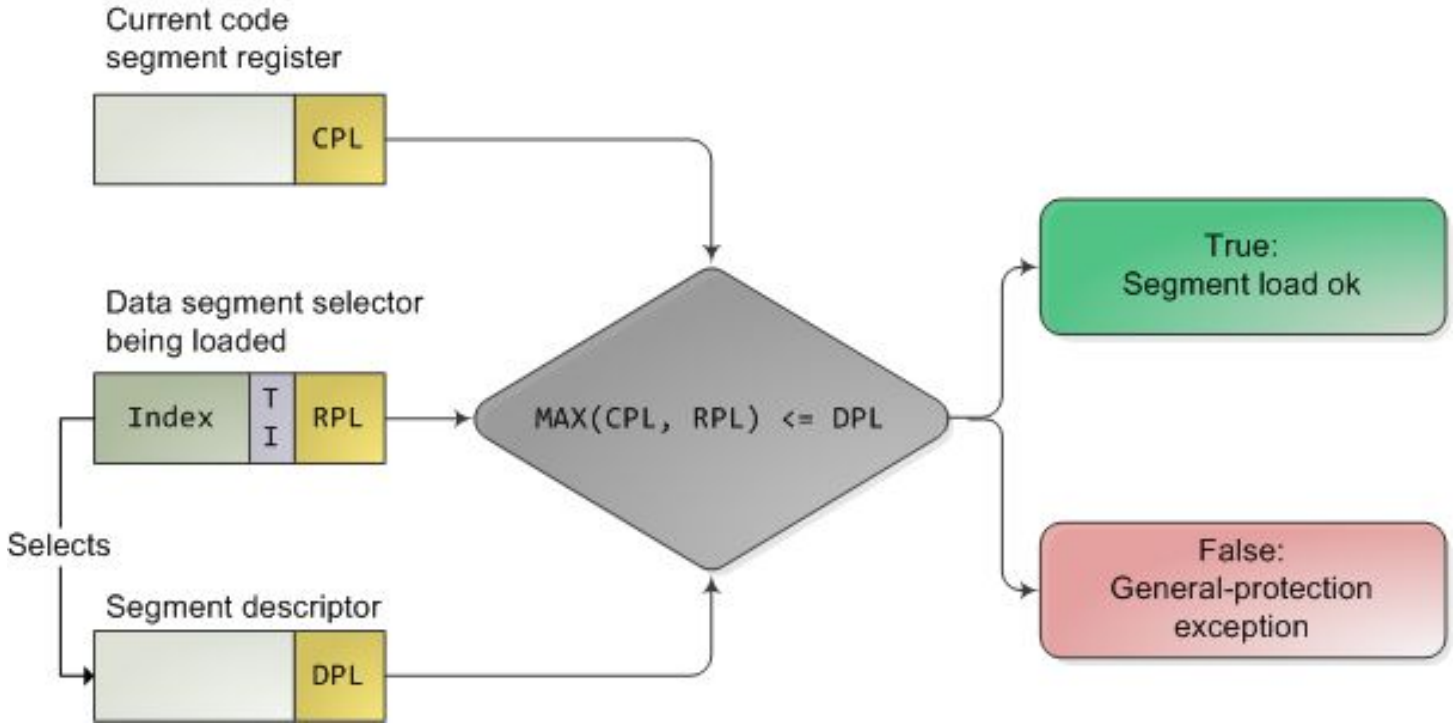
More in detail, the privilege fields are three:

1. **RPL** is the *Requestor Privilege Level* and it is present only in data segment selectors (e.g. SS, DS registers)
2. **CPL** is the *Current Privilege Level* and it is present only in code segment selectors (i.e. CS register that can be loaded only with ljmp/call); the CPL it's always equals to the current CPU privilege level
3. **DPL** is the *Descriptor Privilege Level* and it is present in segment descriptors of the GDT

When enforcing memory protection? In two cases:

- when memory is accessed through a linear address
- when a data segment is loaded from a selector

Protection upon segment load



https://manybutfinite.com/post/cpu-rings-privilege-and-protection/

Gates

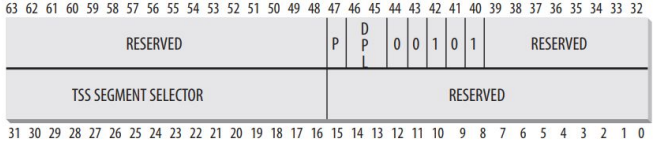
Accessing a segment with a higher privilege (lower ring) with no control might allow malicious code to subvert the kernel. To transfer control, code must pass through a controlled **gate**.

Gates are represented again by descriptors, in particular by **system descriptors** (S = `0`). There are different kinds of gates descriptors:

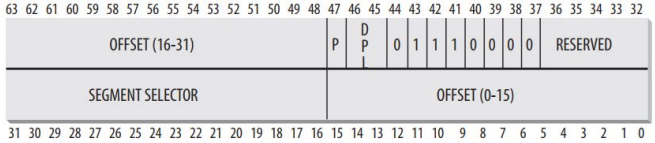
- interrupt-gate descriptors
- trap-gate descriptors
- task-gate descriptors
- (call-gate descriptors)

These descriptors are referenced by the **Interrupt Descriptor Table** (IDT), pointed by the `IDTR` register.

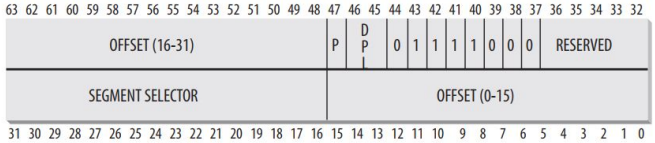
Task Gate Descriptor



Interrupt Gate Descriptor

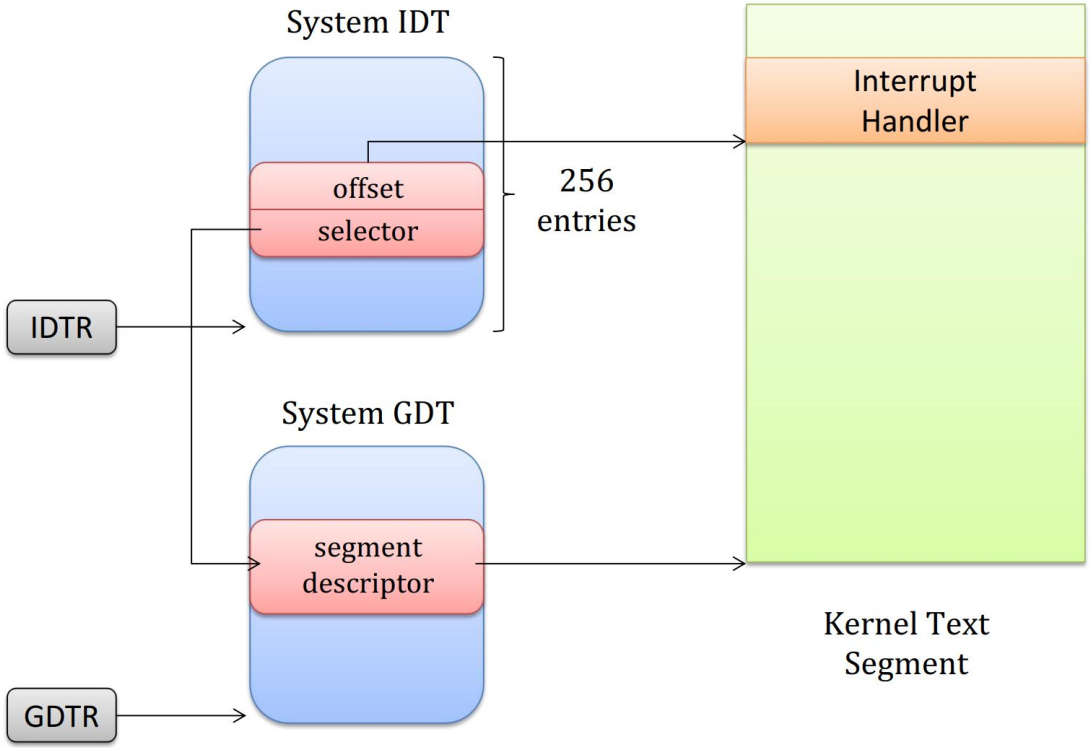


Trap Gate Descriptor



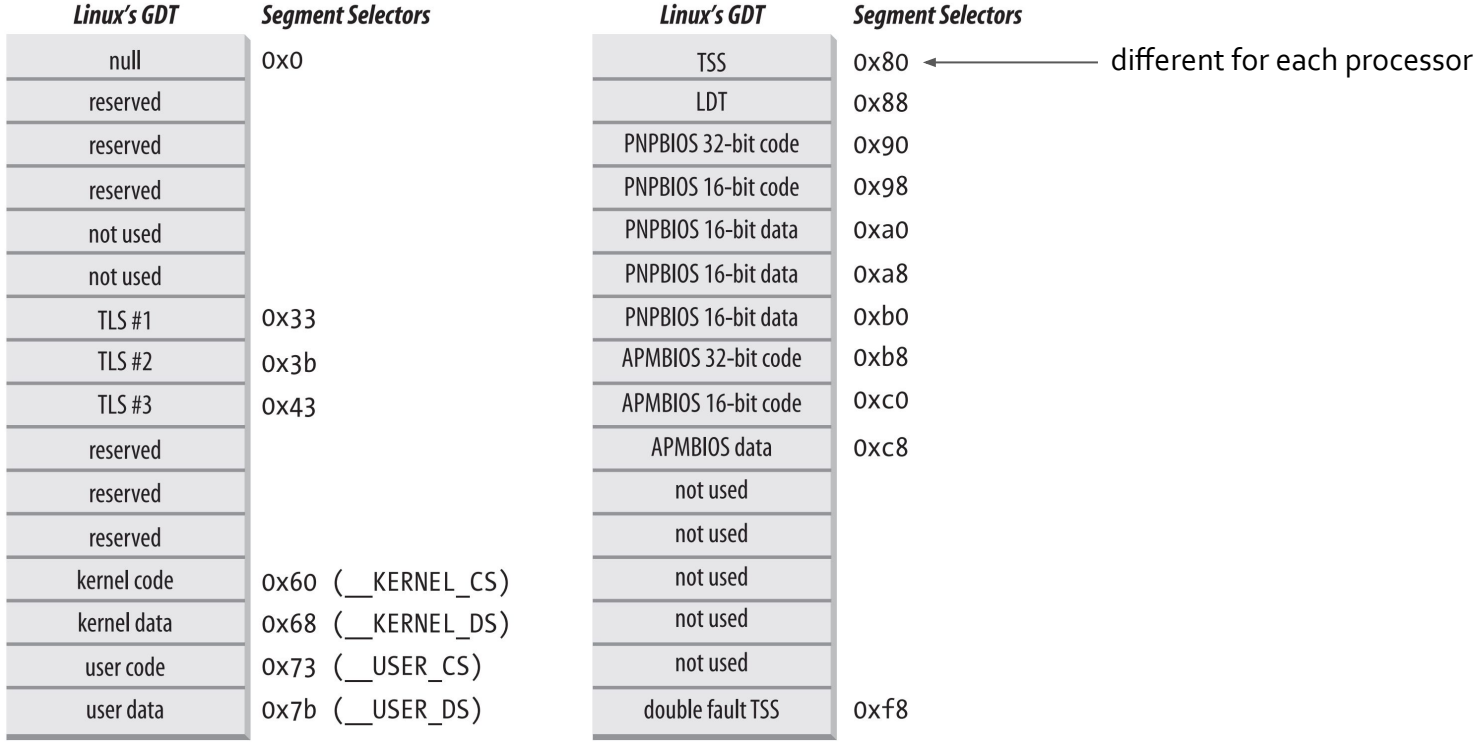
Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

IDT and GDT


System IDT
Interrupt Handler
offset
selector
256 entries
IDTR
System GDT
segment descriptor
GDTR
Kernel Text Segment

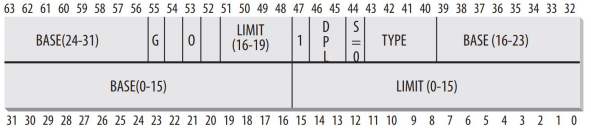
The GDT in Linux (2.6)

There is one GDT in the picture per CPU core.



Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

The TSS (Task State Segment)



The `Base` field of the TSS entry in the GDT (the TSSD) for the *n*-th CPU stored a pointer to the *n*-th entry of the `init_tss` array (Kernel 2.6 - L1 L2).

According to the Intel Manual, the role of the structure is to contain all the necessary information about the current "task" (i.d. process/thread). It stores:

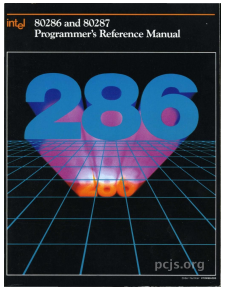
- processor registers (as in the figure)
- I/O ports permissions
- Inner-level stack pointers
- a link to the previous TSS (after a context switch)

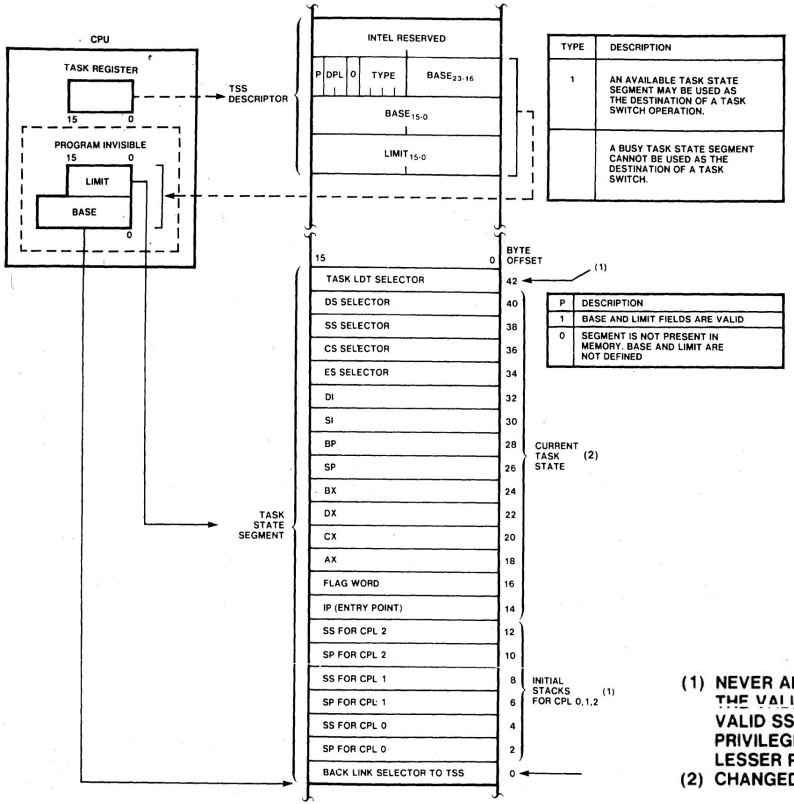
Linux does not use hardware context switches but it is obliged to maintain a TSS for each CPU. A TSS is maintained by the Linux kernel only for active processes.

The TR register of each CPU contains the TSSD of the corresponding TSS (`Base` and `Limit` are cached and non programmable)

```
#ifdef CONFIG_X86_32
/* This is the TSS defined by the hardware. */
struct x86_hw_tss {
        unsigned short          back_link, __blh;
        unsigned long           sp0;
        unsigned short          ss0, __ss0h;
        unsigned long           sp1;
        /* ss1 caches MSR_IA32_SYSENTER_CS: */
        unsigned short          ss1, __ss1h;
        unsigned long           sp2;
        unsigned short          ss2, __ss2h;
        unsigned long           __cr3;
        unsigned long           ip;
        unsigned long           flags;
        unsigned long           ax;
        unsigned long           cx;
        unsigned long           dx;
        unsigned long           bx;
        unsigned long           sp;
        unsigned long           bp;
        unsigned long           si;
        unsigned long           di;
        unsigned short          es, __esh;
        unsigned short          cs, __csh;
        unsigned short          ss, __ssh;
        unsigned short          ds, __dsh;
        unsigned short          fs, __fsh;
        unsigned short          gs, __gsh;
        unsigned short          ldt, __ldth;
        unsigned short          trace;
        unsigned short          io_bitmap_base;

} __attribute__((packed));
#else
```

The TSS (Task State Segment)





(1) NEVER ALTERED (STATIC) AFTER INITIALIZATION BY O.S. THE VALUES AS INITIALIZED FOR THIS TASK ARE ALWAYS VALID SS:SP VALUES TO USE UPON ENTRY TO THAT PRIVILEGE LEVEL (0, 1, OR 2) FROM A LEVEL OF LESSER PRIVILEGE.
(2) CHANGED DURING TASK SWITCH

The TSS on x86_64 (amd64)

On x86_64 hardware context switch is no more supported, indeed as we can see the registers are disappeared from the TSS.

From the Intel Manual:

*Although hardware task-switching is not supported in 64-bit mode, a 64-bit **task state segment (TSS) must exist**. Figure 7-11 shows the format of a 64-bit TSS. The TSS holds information important to 64-bit mode and that is not directly related to the task-switch mechanism. This information includes:*

- ***RSPn** — The full 64-bit canonical forms of the stack pointers (RSP) for privilege levels 0-2.*
- ***ISTn** — The full 64-bit canonical forms of the interrupt stack table (IST) pointers.*
- *I/O map base address — The 16-bit offset to the I/O permission bit map from the 64-bit TSS base.*

The operating system must create at least one 64-bit TSS after activating IA-32e mode. It must execute the LTR instruction (in 64-bit mode) to load the TR register with a pointer to the 64-bit TSS responsible for both 64-bit mode programs and compatibility-mode programs.

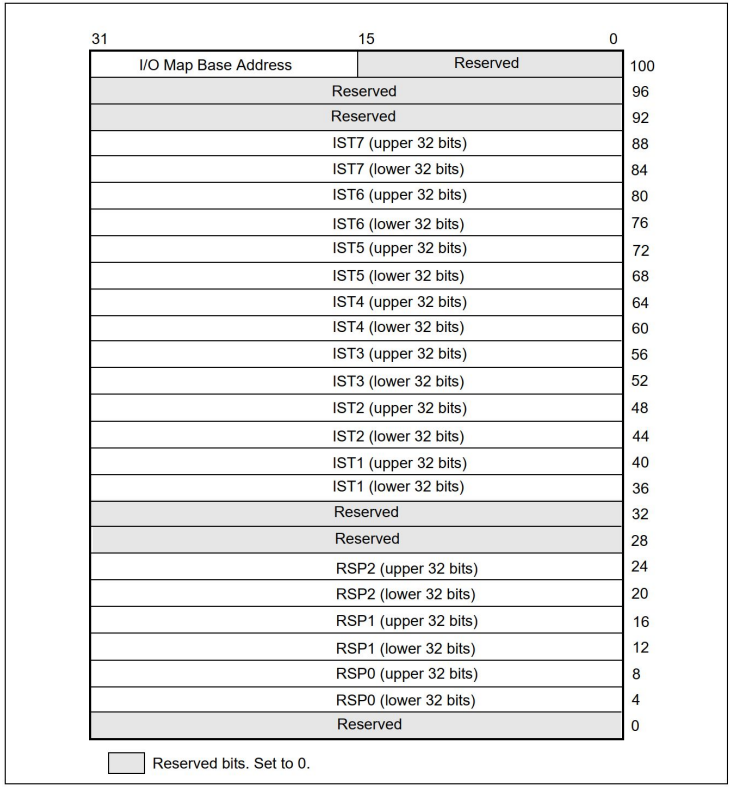
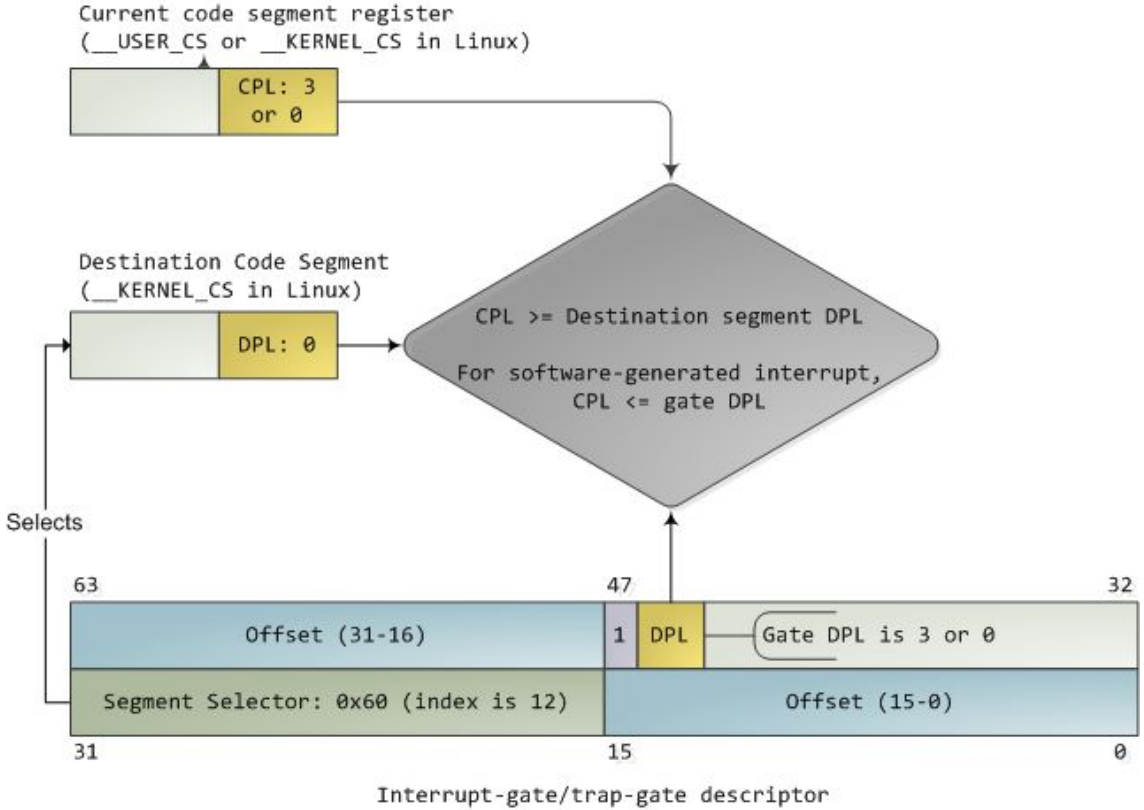


Figure 7-11. 64-Bit TSS Format

From Ring 0 to 3



https://manybutfinite.com/post/cpu-rings-privilege-and-protection/

1.2.5

1. x86 Boot Process
1. Step 1: BIOS/UEFI

Paging

DIAG

Paging

The last step for the x86 Protected Mode, is to enable memory paging. This is not done automatically when enabling x86 protected mode.

The **Paging** unit translates linear addresses to physical ones, the advantages wrt the segmentation (that we remind is not used by the kernel) is that it offers a **smaller granularity memory protection**. The paging unit also checks the request type again the access rights of the linear address, and if access is not granted it generates a Page Fault exception.

To enable paging we need to set up some data structures before. As the term itself, when the paging is enabled, the memory is represented as a set of **pages** of fixed size (4Kb). A page is a set of contiguous linear addresses. With paging, RAM is thinked as partitioned into fixed-length *page frames*, each page frame can contain a page, they have the same size.

Paging

In the x86 architecture

The data structures that maps linear addresses to physical addresses are called **page tables**. The linear address, in the x86 architecture is divided as in the following figure (2 levels of indirection):

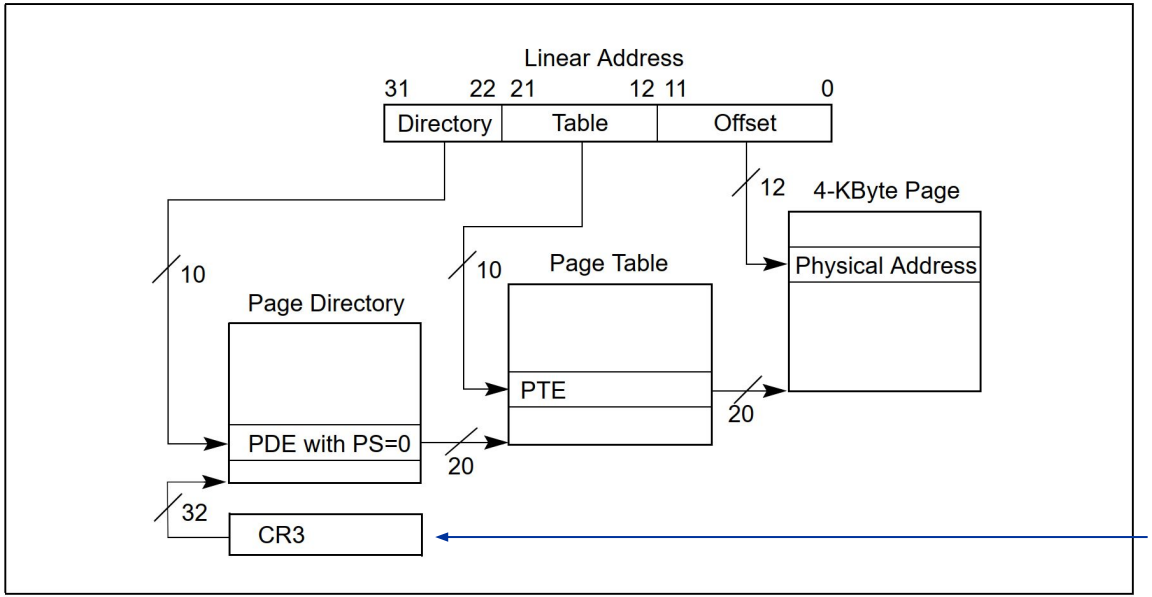
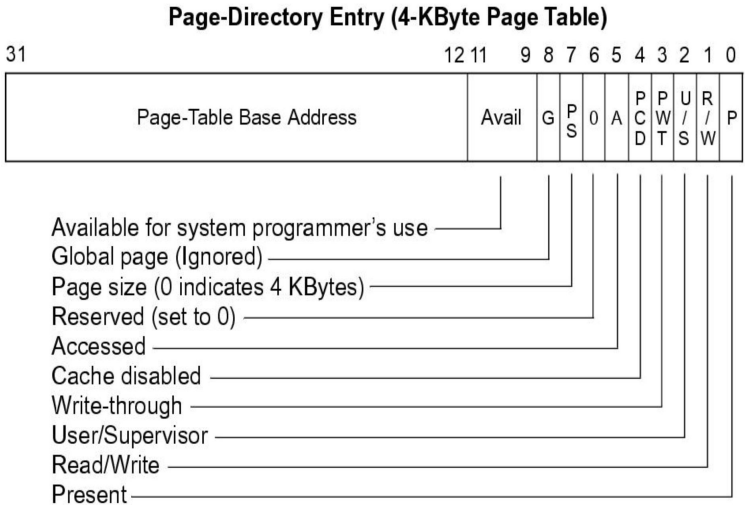


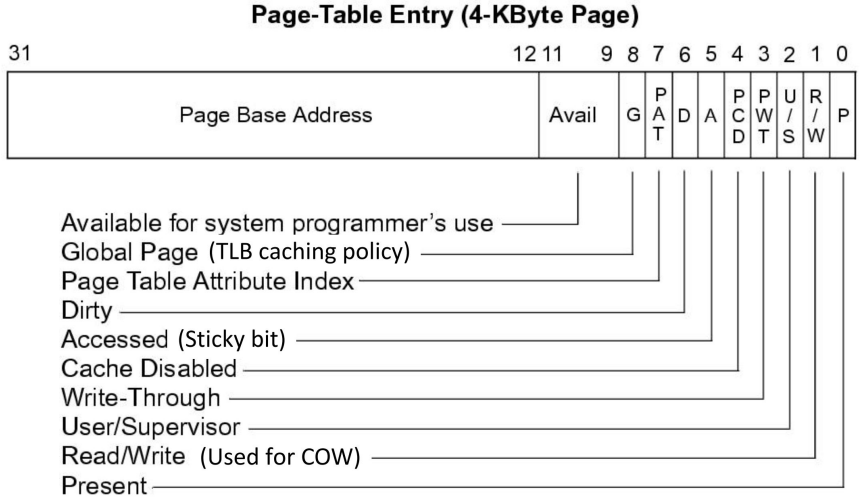
Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Paging

Every active process must have a Page Directory, but there's no need to allocate all the Page Tables. In the x86 paging mechanism:

- each block (PDE and PTE) is an array of 4-bytes
- we can map 1K x 1K pages
- every page is 4KB so we can address a total of 4GB





Paging

Complete path

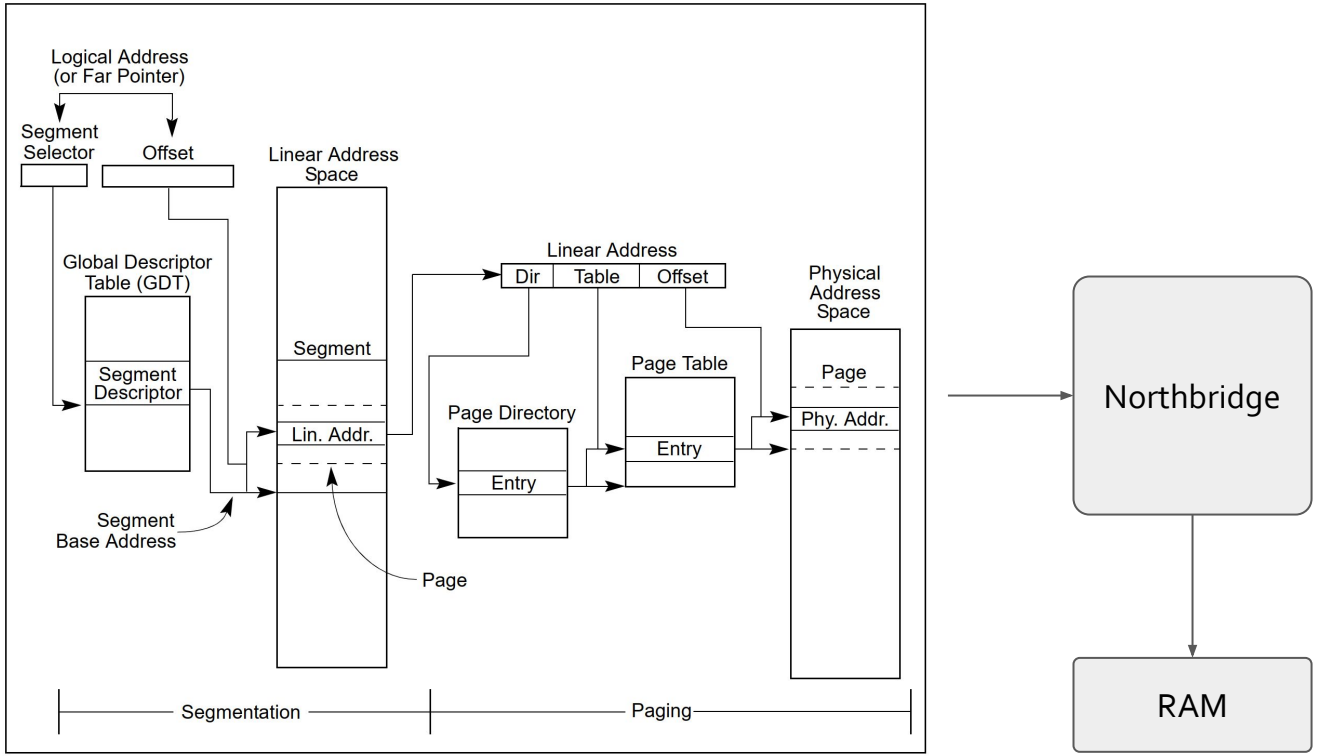


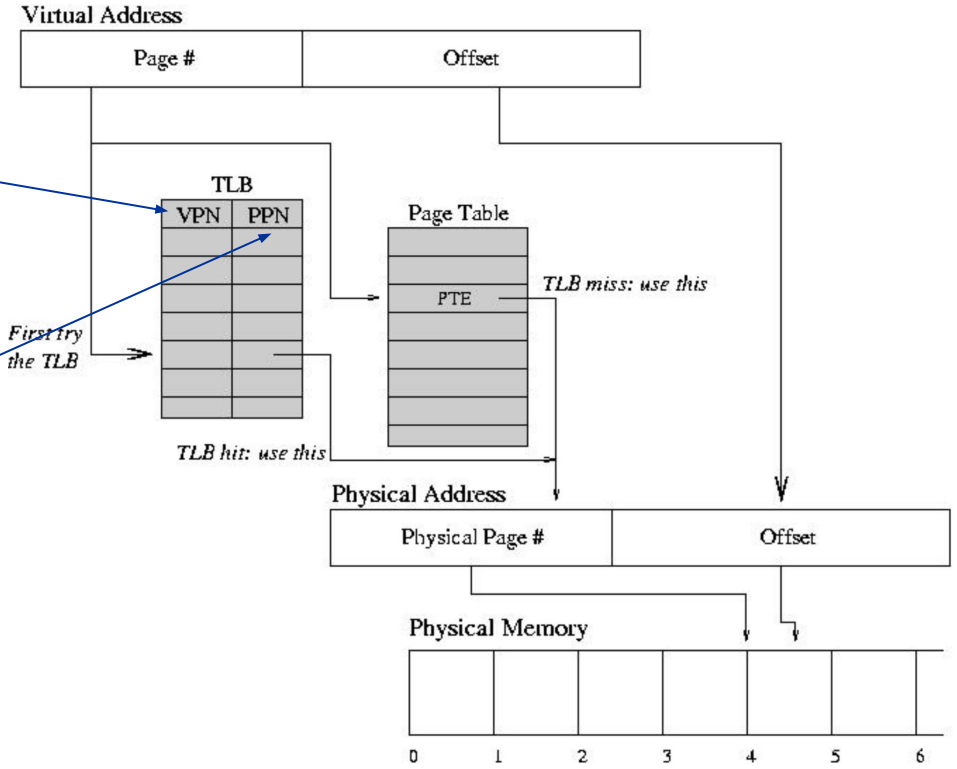
Figure 3-1. Segmentation and Paging

TLB

Translation Lookaside Buffer

Virtual Page Number (the page number described in the logical address)

Physical Page Number (the page frame number in RAM)



Paging

Operations

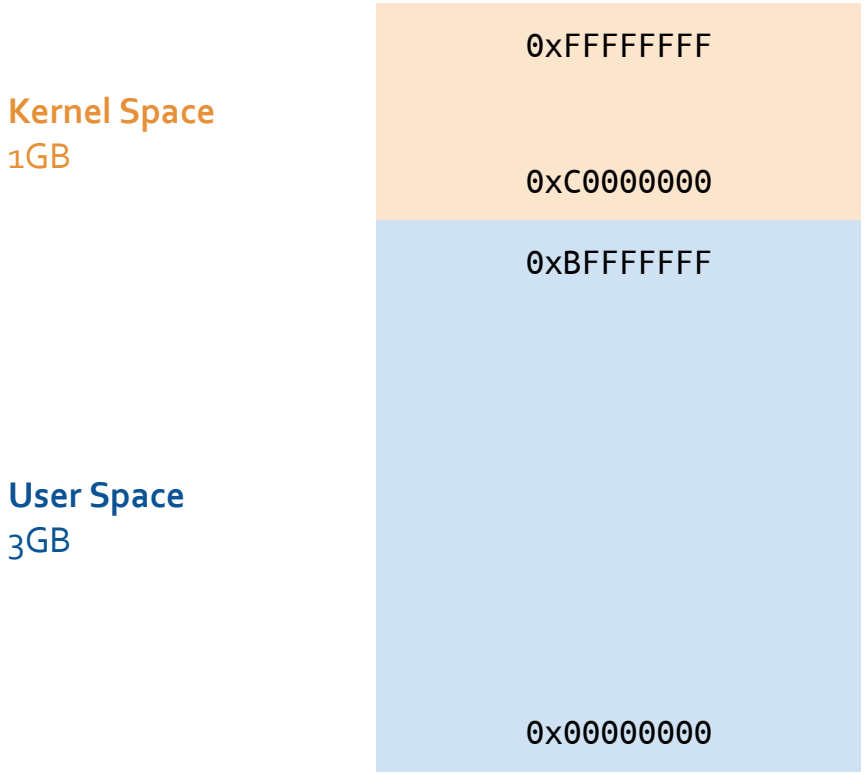
The Paging circuitry performs the following operations:

1. Upon a TLB miss the firmware access the page table
2. It checks the bit P (present) of the table:
    a. If it is **`0`** we have a page fault and a trap is risen
        i. CPU registers (incl. EIP and CS) are saved on the system stack and they will be restored returning from the trap
        ii. The trap instruction is re-executed
        iii. The re-execution can give rise to another trap and so on
    b. If it is **`1`** the page is loaded

As for example, writing to a read-only page will give rise to a trap, which is handled by the Segmentation Fault Handler.

Process Address Space

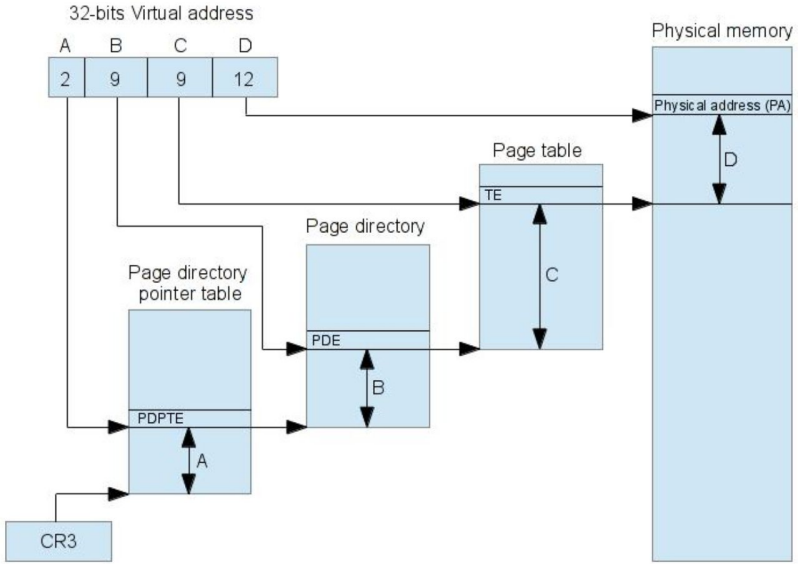
In Linux i386 (32bit)



Physical Address Extension (PAE)

The Physical Address Extension (PAE) has been introduced by Intel, starting with the Pentium Pro (1995) for increasing the RAM size support over 4GB. In practice, the address pins where increased to 36bit (max 64GB) but this required a new page indirection scheme that was increased to 3 levels. Linear addresses obviously remained of 32bit!

The support to PAE is enabled by setting the PAE bit (5th bit) in the CR4 register.

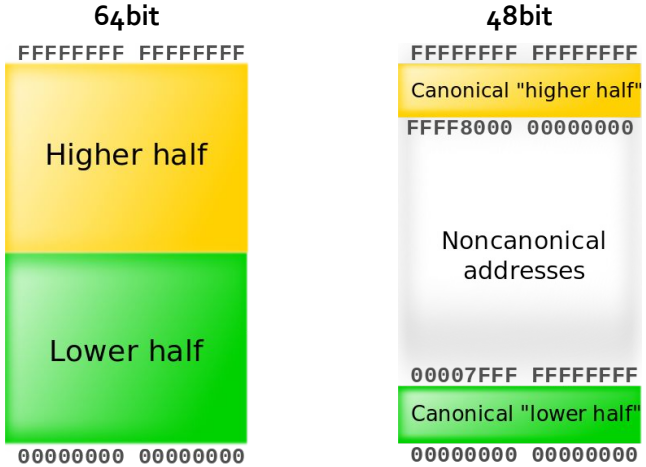


Long Mode in x86_64 (amd64)

Increasing the memory pins to 64bit, again required to extend the page indirection scheme. The PAE scheme is further extended with the **long mode** addressing.

Canonical Addresses

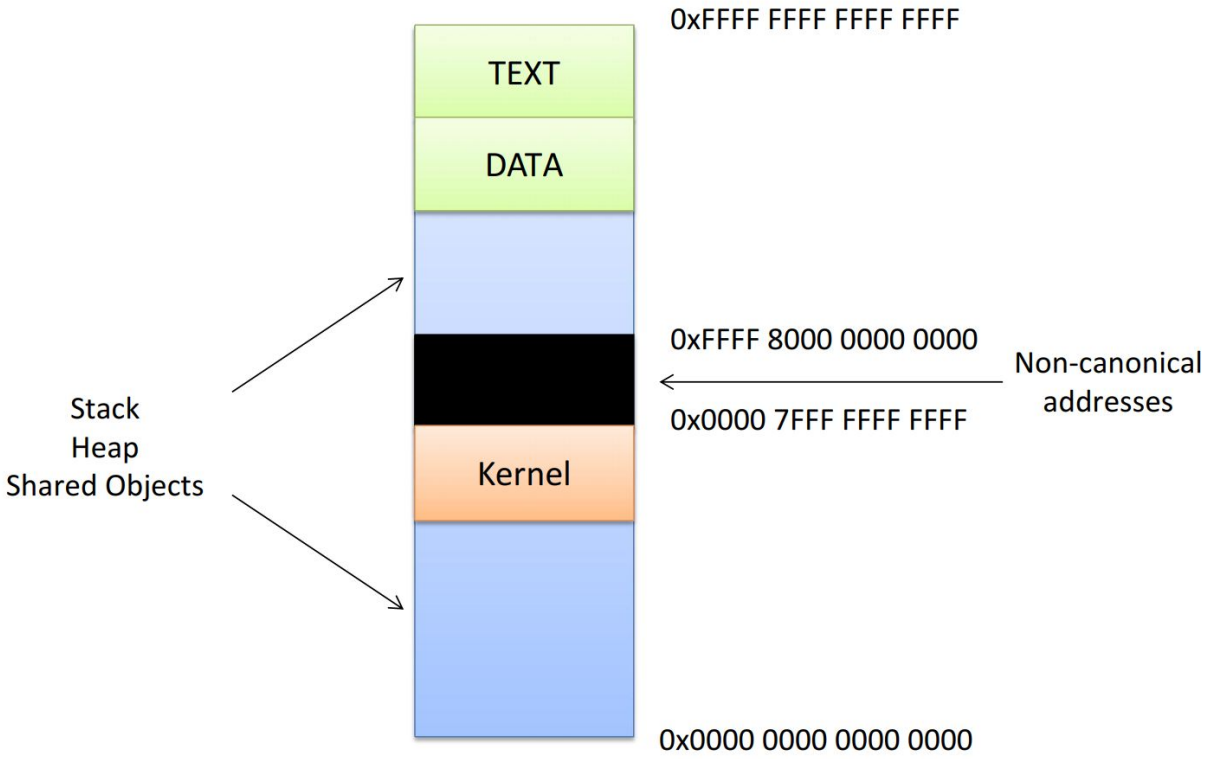
With 64bits of logical memory we have 2^{64} possible addresses, but bits **49-64 are short circuited**. This allows up to 2^{48} **canonical** form addresses, for a total of 256TB of addressable RAM.



https://en.wikipedia.org/wiki/X86-64

Linux currently allows for 128TB of logical addressing and 64TB for physical addressing.

Linux Memory Layout on x64



Long Mode in x86_64 (amd64)

The long mode adds another level of indirection, for a total of 4.

This is also called the **GDT** = General Directory Table

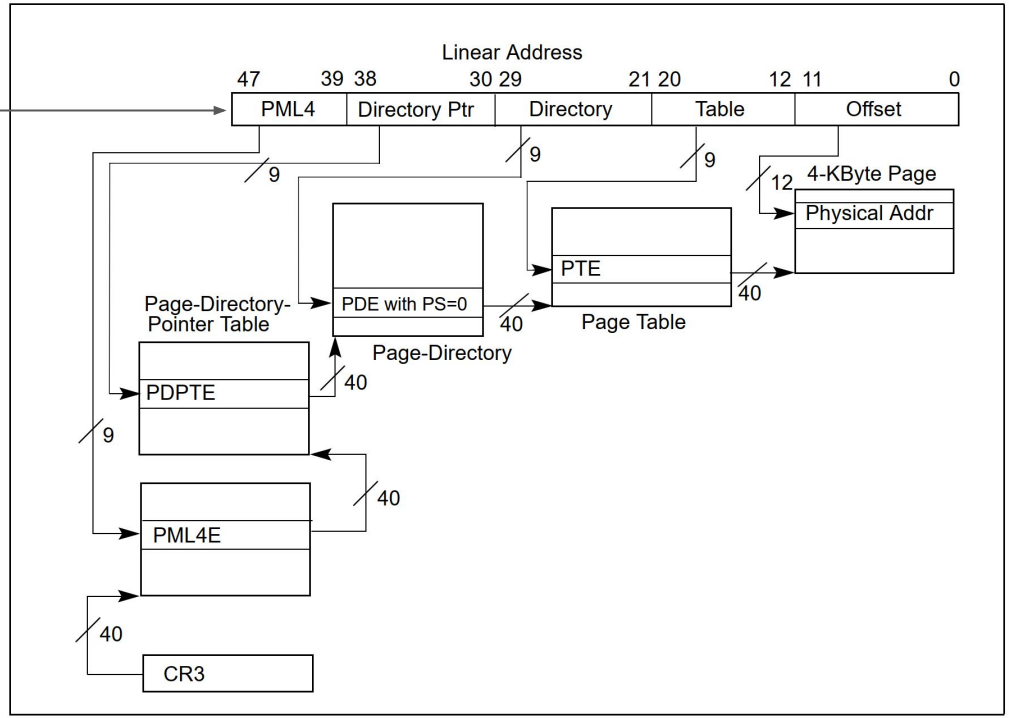


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

Long Mode in x86_64 (amd64)

CR3 and Page Structure Entries

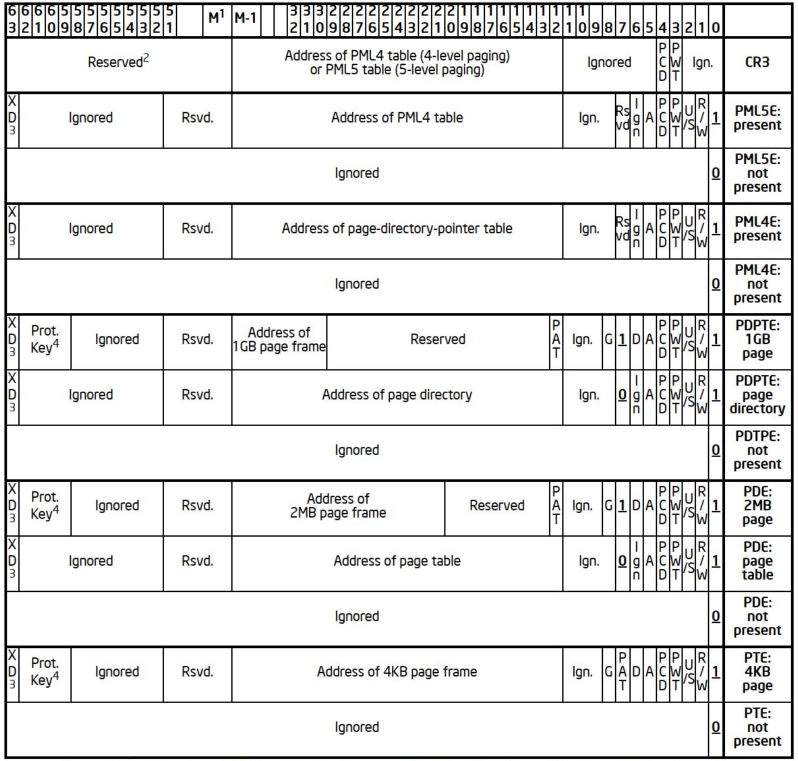


Figure 4-11. Formats of CR3 and Paging-Structure Entries with 4-Level Paging and 5-Level Paging

Huge Pages

The Linux kernel allows the usage of certain pages with bigger size than 4KB, up to 1GB (e.g. useful for DBMS).

They are listed in `/proc/meminfo` and `/proc/sys/vm/nr_hugepages`. Once enabled, huge pages can be mapped with mmap by using the flag `MAP_HUGETLB` or they can be directly requested with the instruction, by using (with `MADV_HUGEPAGE` flag):

```
int madvise(void *addr, size_t length, int advice);
```

Long Mode in x86_64 (amd64)

How can be enabled?

Once we set up the proper data structures, we tell the CPU to enable the Long Mode in the following way.

```
264         pushl   $__KERNEL_CS
265         pushl   %eax
266
267         /* Enter paged protected Mode, activating Long Mode */
268         movl    $(X86_CR0_PG | X86_CR0_PE), %eax /* Enable Paging and Protected mode */
269         movl    %eax, %cr0
270
271         /* Jump from 32bit compatibility mode into 64bit mode. */
272         lret
273 SYM_FUNC_END(startup_32)
274
```

https://elixir.bootlin.com/linux/latest/source/arch/x86/boot/compressed/head_64.S#L268

1.3

1. x86 Boot Process

Step 3: Stage 2 Bootloader

DIAG

Outline

1. **BIOS/UEFI**
   Actual hardware setup
2. **Bootloader Stage 1**
   Executes the stage 2 bootloader (skipped for UEFI)
3. **Bootloader Stage 2**
   Loads and starts the kernel
4. **Kernel**
   Takes control and initializes the machine (machine-dependent operations)
5. **Init (or systemd)**
   First process: basic environment initialization
6. **Runlevels/Targets**
   Initializes the user environment

1.3.1

1. x86 Boot Process
3. Step 3: Stage 2 Bootloader

GRUB & UEFI

DIAG

Stage 2 Bootloader

The stage 1 bootloader (MBR) leaves the control to stage 2 bootloader which has the role of starting the kernel.

- In Linux Distributions we usually have GRUB (formerly LILO), it uses `/boot/grub/grub.conf` for loading the startup entries
- In Windows NT is `ntldr` which uses `boot.ini` as configuration file

The kernel image is loaded in RAM by using BIOS I/O services

- In Linux Distributions the kernel is located in `/boot/vmlinuz-<version>`
- In Windows NT the kernel is located at `C:\Windows\System32\ntoskrnl.exe`

Historical Linux Boot Code

The initial versions of the Linux kernel did not use any Stage 2 Bootloader (GRUB). The file `arch/i386/bootsect.S` contains the historical boot sector that left the control to `arch/i386/bootsetup.S` code which loaded the kernel image in memory. Today these files are **no more used**.

The code in `arch/i386/bootsetup.S` initialized the architecture (e.g. CPU state for the kernel boot) and in the end gave control to the initial kernel image.

UEFI

/ˈuːɪfaɪ/

The **Unified Extensible Firmware Interface** is a set of specifications of software interfaces between an operating system and the platform firmware. Initially developed by Intel (as EFI), today UEFI is the actual replacement of the BIOS. The standard development is today lead by the Unified EFI Forum, a non-profit alliance between major companies like AMD, Intel and others.

Features

- Ability to use large disks partitions (over 2 TB) with a GUID Partition Table (GPT)
- Flexible pre-OS environment, including network capability, GUI, multi language
- 32-bit (for example IA-32, ARM32) or 64-bit (for example x64, AArch64) pre-OS environment
- C language programming
- Modular design
- Backward and forward compatibility

UEFI

Features

The UEFI boot manager takes control right after powering on the machine. It looks at the boot configuration, loads the firmware settings from the nvRAM and then uses startup files located in a specific FAT32 partition that must be created ad hoc (ESP - EFI System Partition). The partition has a folder for every boot entry (OS) and a .efi files that follows a standard path name:

- `/efi/boot/boot_x64.efi`
- `/efi/boot/bootaa64.efi`

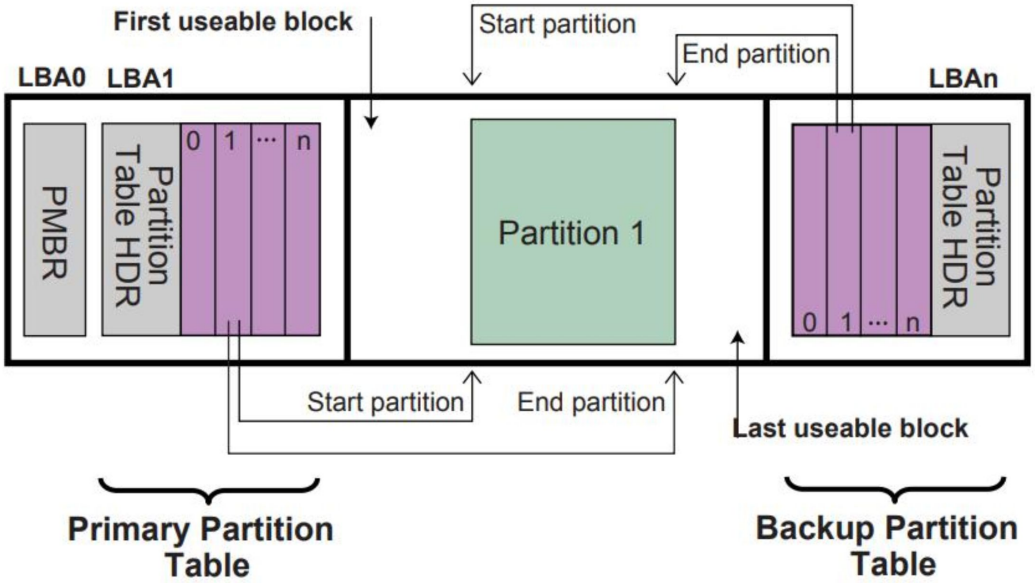
EFI Program Example

```c
#include <efi.h>
#include <efilib.h>

EFI_STATUS
EFIAPI
efi_main (EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
   InitializeLib(ImageHandle, SystemTable);
   Print(L"Hello, world!\n");

   return EFI_SUCCESS;
}
```

UEFI

GUID Partition Table (GPT)

The GUID Partition Table is a partition table standard defined within UEFI. GPT makes use of GUIDs (Globally Unique Identifiers) for identifying partitions.



UEFI

Secure Boot

A certain kind of malware can take control of the system before the OS starts (e.g. MBR Rootkits).

These Rootkits can hijack the IDT for I/O operations in order to execute their own wrapper. Once the kernel is loaded, the rootkit notices that and patches the binary code while loading it into RAM.

UEFI overcomes this issue by allowing only signed executables by using 3 kinds of keys:

- Platform Keys (PK): tell who owns and controls the hardware platform
- Key-Exchange Keys (KEK): shows who is allowed to update the hardware platform
- Signature Database Keys (DB): show who is allowed to boot the platform in secure mode

1.3.2

1. x86 Boot Process
3. Step 3: Stage 2 Bootloader

Multi-core support

DIAG

Multi-core support

Who shall execute the startup code? For legacy reasons, startup code is always sequential and it is executed by a single core (the master). For this reason, upon startup only one core is active and the others are in idle state. We need a way to "wake" all of the other cores.

Interrupts on multicore architectures

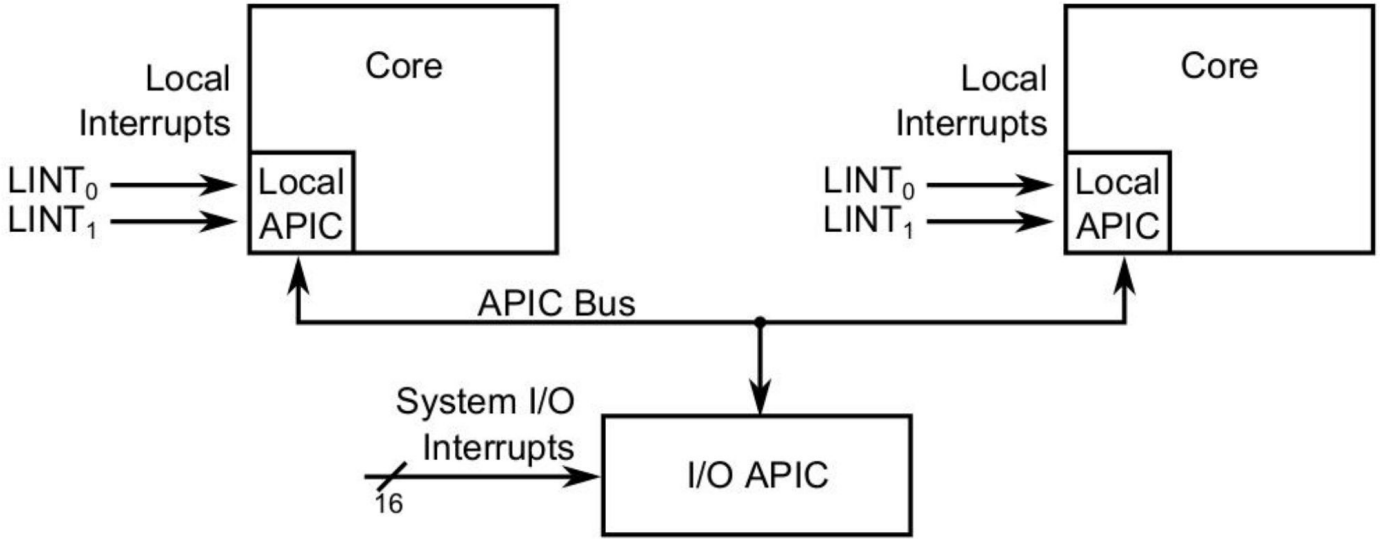
The **Advanced Programmable Interrupt Controller** (APIC) is an interrupt controller. Every processor has a Local-APIC which is used for sending inter-processor interrupts requests (**IPI**s).

LAPICs are connected through a logical bus called APIC Bus and interrupts are of two types:

- LINT 0: normal interrupts
- LINT 1: non-maskable interrupts

The I/O APIC contains a **redirection table** which is used to route the interrupts it receives from peripheral buses to one or moder LAPICs.

LAPICs and APIC Bus



Interrupt Command Register

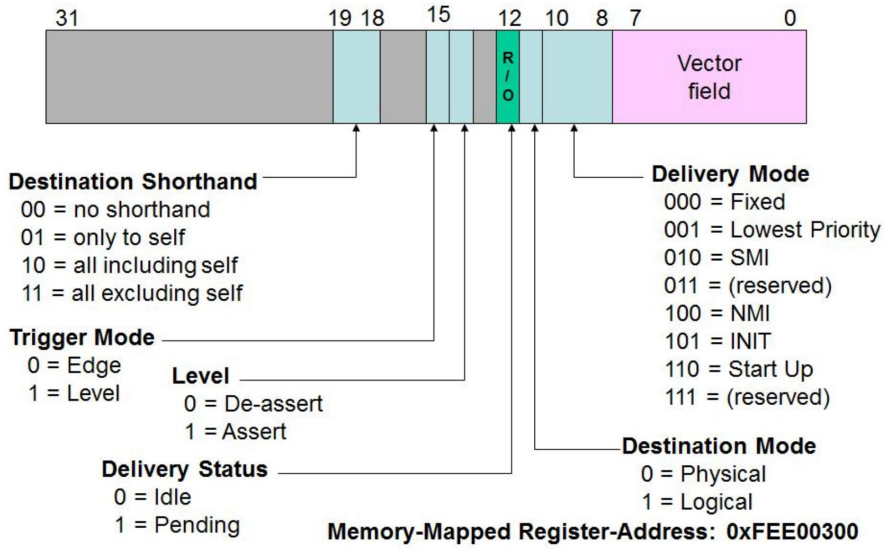
The ICR register is used for initiating an IPI. In that register is specified the kind of the interrupt and the target core.

ICR Upper Part

The Destination Field (8-bits) can be used to specify which processor (or group of processors) will receive the message

Memory-Mapped Register-Address: 0xFEE00310

ICR Lower Part



INIT-SIPI-SIPI Sequence

```
# address Local-APIC via register FS
mov $sel_fs, %ax
mov %ax, %fs

# broadcast 'INIT' IPI to 'all-except-self'
mov $0x000C4500, %eax ; 11 00 0 1 0 0 0 101 00000000
mov %eax, %fs:(0xFEE00300)
# wait until command is received
.B0: btl $12, %fs:(0xFEE00300)
jc .B0

# broadcast 'Startup' IPI to 'all-except-self'
# using vector 0x11 to specify entry-point
# at real memory-address 0x00011000
mov $0x000C4611, %eax ; 11 00 0 0 1 0 0 0 110 00010001
mov %eax, %fs:(0xFEE00300)
.B1: btl $12, %fs:(0xFEE00300)
jc .B1
```

Advanced Operating Systems and Virtualization

[1] The x86 Boot Process

LECTURER

Gabriele **Proietti Mattia**

BASED ON WORK BY

http://www.ce.uniroma2.it/~pellegrini/

gpm.name · proiettimattia@diag.uniroma1.it

DIAG