linux-insides

Alex

Published with GitBook

Table of Contents

linux-internals

A series of posts about the linux kernel.

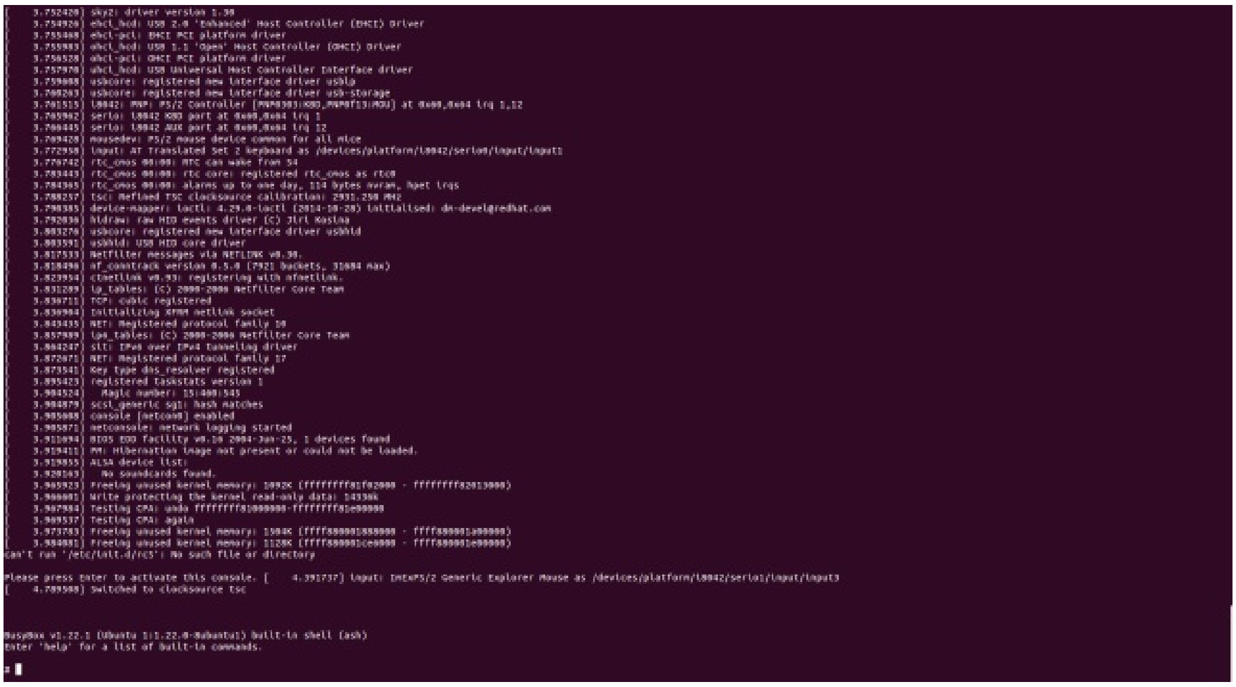
The goal is simple - to share my modest knowledge about the internals of the linux kernel and help people who are interested in the linux kernel, and other low-level subject matter.

Questions/Suggestions: Feel free about any questions or suggestions by pinging me at twitter @0xAX, adding issue or just drop me email.

Support If you like `linux-insides` you can support me with Flattr this!

Contributions: Feel free to create issues or create pull-requests if you find any issues or my English is poor.

Please read CONTRIBUTING.md before pushing any changes.



```
[    3.752428] sky2: driver version 1.30
[    3.754926] ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
[    3.755408] ehci-pci: EHCI PCI platform driver
[    3.755983] ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
[    3.756528] ohci-pci: OHCI PCI platform driver
[    3.757970] uhci_hcd: USB Universal Host Controller Interface driver
[    3.759008] usbcore: registered new interface driver usblp
[    3.760283] usbcore: registered new interface driver usb-storage
[    3.761515] i8042: PNP: PS/2 Controller [PNP0303:KBD,PNP0f13:MOU] at 0x60,0x64 irq 1,12
[    3.765962] serio: i8042 KBD port at 0x60,0x64 irq 1
[    3.766445] serio: i8042 AUX port at 0x60,0x64 irq 12
[    3.769428] mousedev: PS/2 mouse device common for all mice
[    3.772958] input: AT Translated Set 2 keyboard as /devices/platform/i8042/serio0/input/input1
[    3.776742] rtc_cmos 00:00: RTC can wake from S4
[    3.783443] rtc_cmos 00:00: rtc core: registered rtc_cmos as rtc0
[    3.784365] rtc_cmos 00:00: alarms up to one day, 114 bytes nvram, hpet irqs
[    3.788257] tsc: Refined TSC clocksource calibration: 2931.250 MHz
[    3.790385] device-mapper: ioctl: 4.29.0-ioctl (2014-10-28) initialised: dm-devel@redhat.com
[    3.792036] hidraw: raw HID events driver (C) Jiri Kosina
[    3.803270] usbcore: registered new interface driver usbhid
[    3.803591] usbhid: USB HID core driver
[    3.817533] Netfilter messages via NETLINK v0.30.
[    3.818496] nf_conntrack version 0.5.0 (7921 buckets, 31684 max)
[    3.823954] ctnetlink v0.93: registering with nfnetlink.
[    3.831209] ip_tables: (C) 2000-2006 Netfilter Core Team
[    3.836711] TCP: cubic registered
[    3.836904] Initializing XFRM netlink socket
[    3.843435] NET: Registered protocol family 10
[    3.857909] ip6_tables: (C) 2000-2006 Netfilter Core Team
[    3.864247] sit: IPv6 over IPv4 tunneling driver
[    3.872071] NET: Registered protocol family 17
[    3.873541] Key type dns_resolver registered
[    3.895423] registered taskstats version 1
[    3.904524]   Magic number: 15:400:545
[    3.904879] scsi_generic sg1: hash matches
[    3.905008] console [netcon0] enabled
[    3.905871] netconsole: network logging started
[    3.911094] BIOS EDD facility v0.16 2004-Jun-25, 1 devices found
[    3.919411] PM: Hibernation image not present or could not be loaded.
[    3.919855] ALSA device list:
[    3.920163]   No soundcards found.
[    3.965923] Freeing unused kernel memory: 1092K (ffffffff81f02000 - ffffffff82013000)
[    3.966001] Write protecting the kernel read-only data: 14336k
[    3.967984] Testing CPA: undo ffffffff81000000-ffffffff81e00000
[    3.969537] Testing CPA: again
[    3.973783] Freeing unused kernel memory: 1504K (ffff880001888000 - ffff880001a00000)
[    3.984081] Freeing unused kernel memory: 1128K (ffff880001ce6000 - ffff880001e00000)
can't run '/etc/init.d/rcS': No such file or directory

Please press Enter to activate this console. [    4.391737] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/input3
[    4.789588] Switched to clocksource tsc


BusyBox v1.22.1 (Ubuntu 1:1.22.0-8ubuntu1) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# █
```

Author

@0xAX

Kernel booting process

This chapter describes linux kernel booting process. You will see here a couple of posts which describes full cycle of the kernel loading:

- From the bootloader to kernel - describes all stages from turning on the computer before the first instruction of the kernel;
- First steps in the kernel setup code - describes first steps in the kernel setup code. You will see heap initialization, querying of different parameters like EDD, IST and etc...
- Video mode initialization and transition to protected mode - describes video mode initialization in the kernel setup code and transition to protected mode.
- Transition to 64-bit mode - describes preparation for transition into 64-bit mode and transition into it.
- Kernel Decompression - describes preparation before kernel decompression and directly decompression.

Kernel booting process. Part 1.

From the bootloader to kernel

If you have read my previous blog posts, you can see that some time ago I started to get involved with low-level programming. I wrote some posts about x86_64 assembly programming for Linux. At the same time, I started to dive into the Linux source code. It is very interesting for me to understand how low-level things work, how programs run on my computer, how they are located in memory, how the kernel manages processes and memory, how the network stack works on low-level and many many other things. I decided to write yet another series of posts about the Linux kernel for **x86_64**.

Note that I'm not a professional kernel hacker, and I don't write code for the kernel at work. It's just a hobby. I just like low-level stuff, and it is interesting for me to see how these things work. So if you notice anything confusing, or if you have any questions/remarks, ping me on twitter 0xAX, drop me an email or just create an issue. I appreciate it. All posts will also be accessible at linux-insides and if you find something wrong with my English or post content, feel free to send pull request.

Note that this isn't official documentation, just learning and sharing knowledge.

Required knowledge

- Understanding C code
- Understanding assembly code (AT&T syntax)

Anyway, if you just started to learn some tools, I will try to explain some parts during this and following posts. Ok, little introduction finished and now we can start to dive into kernel and low-level stuff.

All code is actual for kernel - 3.18, if there are changes, I will update posts.

Magic power button, what's next?

Despite that this is a series of posts about linux kernel, we will not start from kernel code (at least in this paragraph). Ok, you pressed magic power button on your laptop or desktop computer and it started to work. After the motherboard sends a signal to the power supply, the power supply provides the computer with the proper amount of electricity. Once motherboard receives the power good signal, it tries to run the CPU. The CPU resets all leftover data in its registers and sets up predefined values for every register.

80386 and later CPUs define the following predefined data in CPU registers after the computer resets:

```
IP          0xfff0
CS selector 0xf000
CS base     0xffff0000
```

The processor starts working in real mode now and we need to make a little retreat for understanding memory segmentation in this mode. Real mode is supported in all x86-compatible processors, from 8086 to modern Intel 64-bit CPUs. The 8086 processor had a 20-bit address bus, which means that it could work with 0-2^20 bytes address space (1 megabyte). But it only had 16-bit registers, and with 16-bit registers the maximum address is 2^16 or 0xffff (64 kilobytes). Memory segmentation was used to make use of all of the address space. All memory was divided into small, fixed-size segments of 65535 bytes, or 64 KB. Since we cannot address memory behind 64 KB with 16 bit registers, another method to do it was devised. An address consists of two parts: the beginning address of the segment and the offset from the beginning of this segment. To get a physical address in memory, we need to multiply the segment part by 16 and add the offset part:

```
PhysicalAddress = Segment * 16 + Offset
```

For example `CS:IP` is `0x2000:0x0010`. The corresponding physical address will be:

```
>>> hex((0x2000 << 4) + 0x0010)
'0x20010'
```

But if we take the biggest segment part and offset: `0xffff:0xffff`, it will be:

```
>>> hex((0xffff << 4) + 0xffff)
'0x10ffef'
```

which is 65519 bytes over first megabyte. Since only one megabyte is accessible in real mode, `0x10ffef` becomes `0x00ffef` with disabled A20.

Ok, now we know about real mode and memory addressing. Let's get back to register values after reset.

`CS` register consists of two parts: the visible segment selector and hidden base address. We know predefined `CS` base and `IP` value, logical address will be:

```
0xffff0000:0xfff0
```

In this way starting address formed by adding the base address to the value in the EIP register:

```
>>> 0xffff0000 + 0xfff0
'0xfffffff0'
```

We get `0xfffffff0` which is 4GB - 16 bytes. This point is the Reset vector. This is the memory location at which CPU expects to find the first instruction to execute after reset. It contains a jump instruction which usually points to the BIOS entry point. For example, if we look in coreboot source code, we will see it:

```
    .section ".reset"
    .code16
.globl    reset_vector
reset_vector:
    .byte  0xe9
    .int   _start - ( . + 2 )
    ...
```

We can see here jump instruction opcode - 0xe9 to the address `_start - ( . + 2)`. And we can see that `reset` section is 16 bytes and starts at `0xfffffff0`:

```
SECTIONS {
    _ROMTOP = 0xfffffff0;
    . = _ROMTOP;
    .reset . : {
        *(.reset)
        . = 15 ;
        BYTE(0x00);
    }
}
```

Now the BIOS has started to work. After initializing and checking the hardware, it needs to find a bootable device. A boot order is stored in the BIOS configuration, controlling which devices the kernel attempts to boot. In the case of attempting to boot a hard drive, the BIOS tries to find a boot sector. On hard drives partitioned with an MBR partition layout, the boot sector is stored in the first 446 bytes of the first sector (512 bytes). The final two bytes of the first sector are `0x55` and `0xaa` which signals the BIOS that the device as bootable. For example:

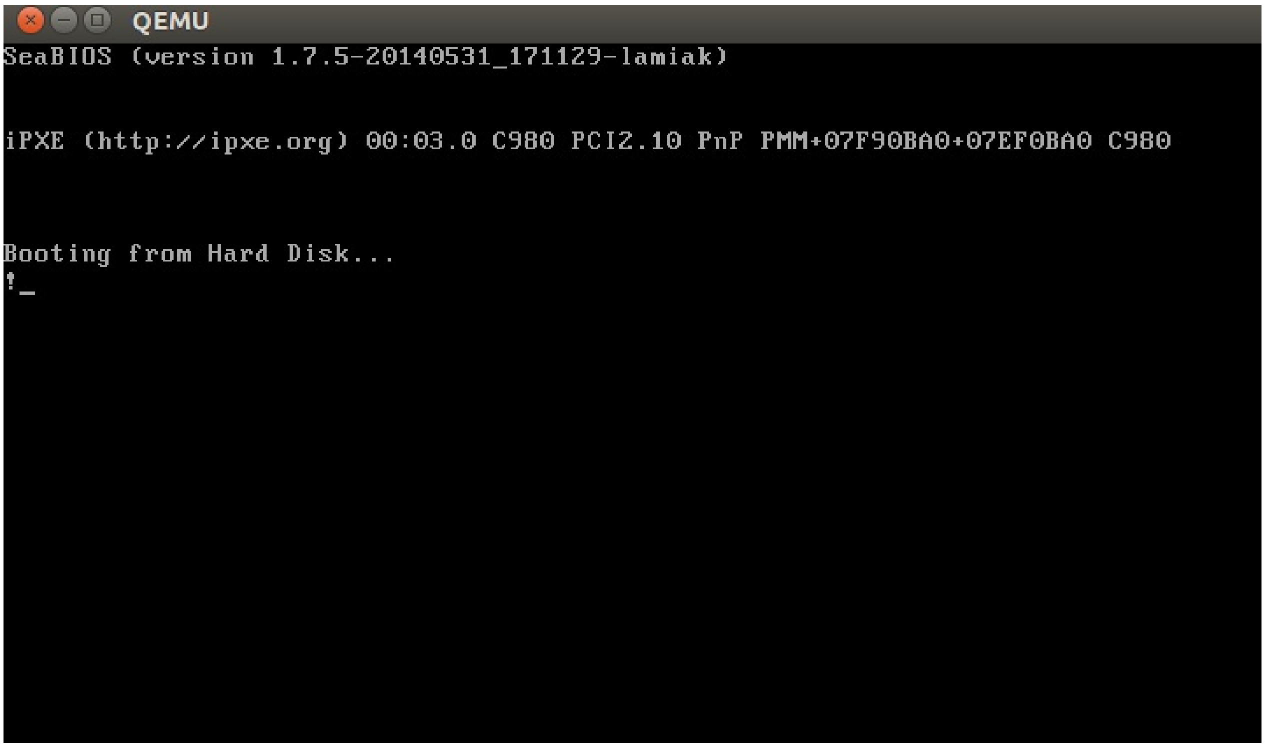
```
#
# Note: this example written with Intel syntax
#
[BITS 16]
[ORG  0x7c00]

jmp boot

boot:
    mov al, '!'
    mov ah, 0x0e
    mov bh, 0x00
    mov bl, 0x07

    int 0x10
    jmp $

times 510-($-$$) db 0

db 0x55
db 0xaa
```

Build and run it with:

```
nasm -f bin boot.nasm && qemu-system-x86_64 boot
```

This will instruct QEMU to use the `boot` binary we just built as a disk image. Since the binary generated by the assembly code above fulfills the requirements of the boot sector (the origin is set to 0x7c00, and we end with the magic sequence), QEMU will treat the binary as the master boot record of a disk image.

We will see:



```
SeaBIOS (version 1.7.5-20140531_171129-lamiak)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F90BA0+07EF0BA0 C980



Booting from Hard Disk...
!_
```

In this example we can see that this code will be executed in 16 bit real mode and will start at 0x7c00 in memory. After the start it calls the 0x10 interrupt which just prints `!` symbol. It fills rest of 510 bytes with zeros and finish with two magic bytes `0xaa` and `0x55`.

Although you can see binary dump of it with `objdump` util:

```
nasm -f binary boot.nasm
```

```
objdump -D -b binary -mi386 -Maddr16,data16,intel boot
```

A real-world boot sector has code for continuing the boot process and the partition table... instead of a bunch of 0's and an exclamation point :) Ok, so, from this moment BIOS handed control to the bootloader and we can go ahead.

NOTE: as you can read above the CPU is in real mode. In real mode, calculating the physical address in memory is as follows:

```
PhysicalAddress = Segment * 16 + Offset
```

as I wrote above. But we have only 16 bit general purpose registers. The maximum value of 16 bit register is: `0xffff`; So if we take the biggest values, it will be:

```
>>> hex((0xffff * 16) + 0xffff)
'0x10ffef'
```

Where `0x10ffef` is equal to `1mb + 64KB - 16b`. But a 8086 processor, which was first processor with real mode, had 20 bit address line, and `2^20 = 1048576.0` is 1MB, so it means that actually available memory amount is 1MB.

General real mode's memory map is:

```
0x00000000 - 0x000003FF - Real Mode Interrupt Vector Table
0x00000400 - 0x000004FF - BIOS Data Area
0x00000500 - 0x00007BFF - Unused
0x00007C00 - 0x00007DFF - Our Bootloader
0x00007E00 - 0x0009FFFF - Unused
0x000A0000 - 0x000BFFFF - Video RAM (VRAM) Memory
0x000B0000 - 0x000B7777 - Monochrome Video Memory
0x000B8000 - 0x000BFFFF - Color Video Memory
0x000C0000 - 0x000C7FFF - Video ROM BIOS
0x000C8000 - 0x000EFFFF - BIOS Shadow Area
0x000F0000 - 0x000FFFFF - System BIOS
```

But stop, at the beginning of post I wrote that first instruction executed by the CPU is located at address `0xfffffff0`, which is much bigger than `0xffff` (1MB). How can CPU access it in real mode? As I write about and you can read in coreboot documentation:

```
0xFFFE_0000 - 0xFFFF_FFFF: 128 kilobyte ROM mapped into address space
```

At the start of execution BIOS is not in RAM, it is located in ROM.

Bootloader

There are a number of bootloaders which can boot Linux, such as GRUB 2 and syslinux. The Linux kernel has a Boot protocol which specifies the requirements for bootloaders to implement Linux support. This example will describe GRUB 2.

Now that the BIOS has chosen a boot device and transferred control to the boot sector code, execution starts from boot.img. This code is very simple due to the limited amount of space available, and contains a pointer that it uses to jump to the location of GRUB 2's core image. The core image begins with diskboot.img, which is usually stored immediately after the first sector in the unused space before the first partition. The above code loads the rest of the core image into memory, which contains GRUB 2's kernel and drivers for handling filesystems. After loading the rest of the core image, it executes grub_main.

`grub_main` initializes console, gets base address for modules, sets root device, loads/parses grub configuration file, loads modules etc... At the end of execution, `grub_main` moves grub to normal mode. `grub_normal_execute` (from `grub-`

`core/normal/main.c`) completes last preparation and shows a menu for selecting an operating system. When we select one of grub menu entries, `grub_menu_execute_entry` begins to be executed, which executes grub `boot` command. It starts to boot operating system.

As we can read in the kernel boot protocol, the bootloader must read and fill some fields of kernel setup header which starts at `0x01f1` offset from the kernel setup code. Kernel header arch/x86/boot/header.S starts from:

```
    .globl hdr
hdr:
    setup_sects: .byte 0
    root_flags:  .word ROOT_RDONLY
    syssize:     .long 0
    ram_size:    .word 0
    vid_mode:    .word SVGA_MODE
    root_dev:    .word 0
    boot_flag:   .word 0xAA55
```

The bootloader must fill this and the rest of the headers (only marked as `write` in the linux boot protocol, for example this) with values which it either got from command line or calculated. We will not see description and explanation of all fields of kernel setup header, we will get back to it when kernel uses it. Anyway, you can find description of any field in the boot protocol.

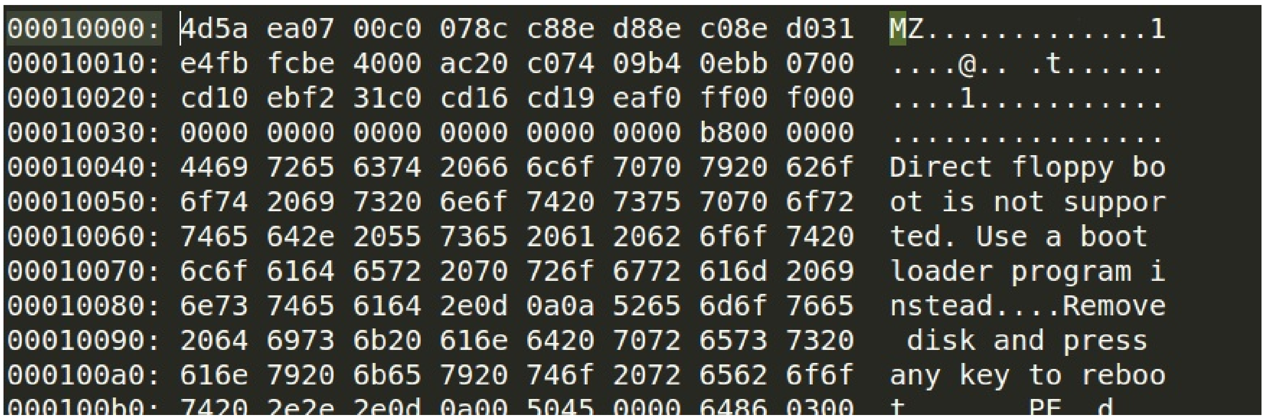
As we can see in kernel boot protocol, the memory map will be the following after kernel loading:

```
         | Protected-mode kernel  |
100000   +------------------------+
         | I/O memory hole        |
0A0000   +------------------------+
         | Reserved for BIOS      | Leave as much as possible unused
         ~                        ~
         | Command line           | (Can also be below the X+10000 mark)
X+10000  +------------------------+
         | Stack/heap             | For use by the kernel real-mode code.
X+08000  +------------------------+
         | Kernel setup           | The kernel real-mode code.
         | Kernel boot sector     | The kernel legacy boot sector.
       X +------------------------+
         | Boot loader            | <- Boot sector entry point 0x7C00
001000   +------------------------+
         | Reserved for MBR/BIOS  |
000800   +------------------------+
         | Typically used by MBR  |
000600   +------------------------+
         | BIOS use only          |
000000   +------------------------+
```

So after the bootloader transferred control to the kernel, it starts somewhere at:

```
0x1000 + X + sizeof(KernelBootSector) + 1
```

where `X` is the address kernel bootsector loaded. In my case `X` is `0x10000` (), we can see it in memory dump:



```
00010000: 4d5a ea07 00c0 078c c88e d88e c08e d031  MZ.............1
00010010: e4fb fcbe 4000 ac20 c074 09b4 0ebb 0700  ....@.. .t......
00010020: cd10 ebf2 31c0 cd16 cd19 eaf0 ff00 f000  ....1...........
00010030: 0000 0000 0000 0000 0000 0000 b800 0000  ................
00010040: 4469 7265 6374 2066 6c6f 7070 7920 626f  Direct floppy bo
00010050: 6f74 2069 7320 6e6f 7420 7375 7070 6f72  ot is not suppor
00010060: 7465 642e 2055 7365 2061 2062 6f6f 7420  ted. Use a boot
00010070: 6c6f 6164 6572 2070 726f 6772 616d 2069  loader program i
00010080: 6e73 7465 6164 2e0d 0a0a 5265 6d6f 7665  nstead....Remove
00010090: 2064 6973 6b20 616e 6420 7072 6573 7320   disk and press
000100a0: 616e 7920 6b65 7920 746f 2072 6562 6f6f  any key to reboo
000100b0: 7420 2e2e 2e0d 0a00 5045 0000 6486 0300  t .......PE..d...
```

Ok, bootloader loaded linux kernel into memory, filled header fields and jumped to it. Now we can move directly to the kernel setup code.

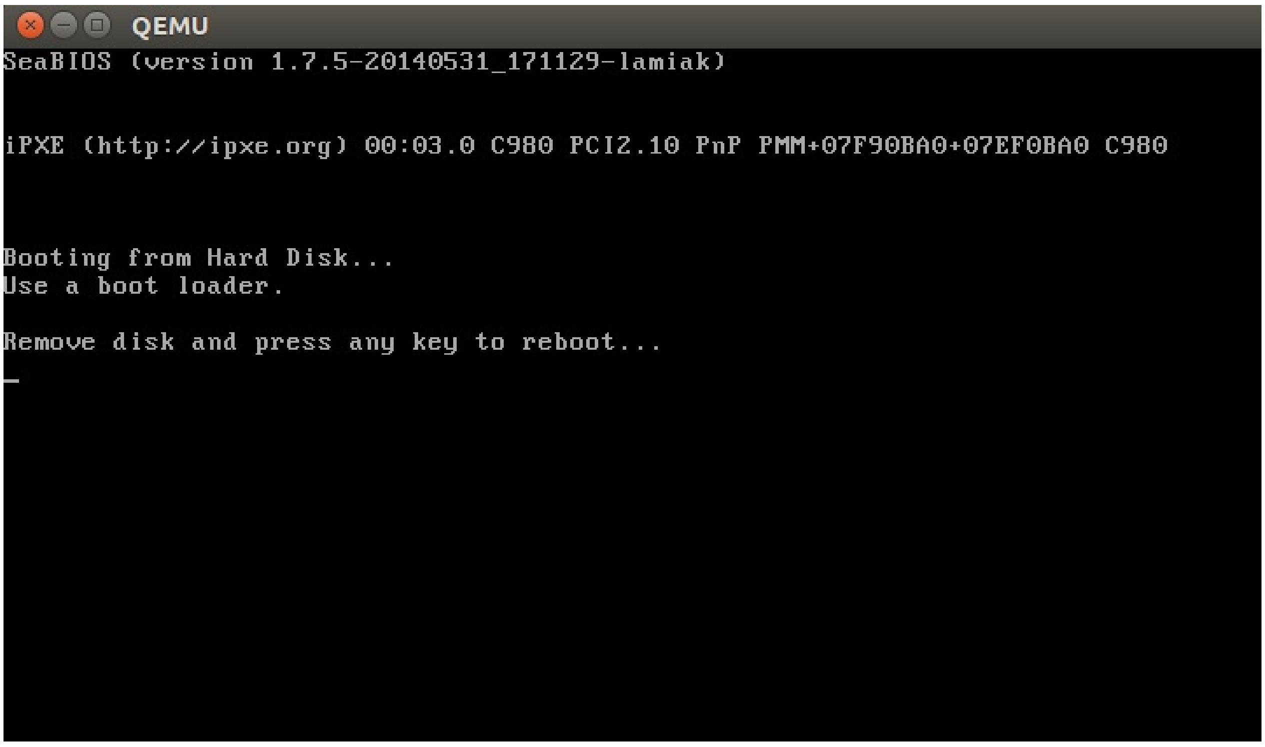
Start of kernel setup

Finally we are in the kernel. Technically kernel didn't run yet, first of all we need to setup kernel, memory manager, process manager and etc... Kernel setup execution starts from arch/x86/boot/header.S at the _start. It is little strange at the first look, there are many instructions before it. Actually....

Long time ago linux had its own bootloader, but now if you run for example:

```
qemu-system-x86_64 vmlinuz-3.18-generic
```

You will see:



```
SeaBIOS (version 1.7.5-20140531_171129-lamiak)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F90BA0+07EF0BA0 C980



Booting from Hard Disk...
Use a boot loader.

Remove disk and press any key to reboot...
_
```

Actually `header.S` starts from MZ (see image above), error message printing and following PE header:

```
#ifdef CONFIG_EFI_STUB
# "MZ", MS-DOS header
.byte 0x4d
.byte 0x5a
#endif
...
```

```
...
...
pe_header:
    .ascii "PE"
    .word 0
```

It needs this for loading operating system with UEFI. Here we will not see how it works (will look into it in the next parts).

So actual kernel setup entry point is:

```
// header.S line 292
.globl _start
_start:
```

Bootloader (grub2 and others) knows about this point (`0x200` offset from `MZ`) and makes a jump directly to this point, despite the fact that `header.S` starts from `.bstext` section which prints error message:

```
//
// arch/x86/boot/setup.ld
//
. = 0;                    // current position
.bstext : { *(.bstext) }  // put .bstext section to position 0
.bsdata : { *(.bsdata) }
```

So kernel setup entry point is:

```
    .globl _start
_start:
    .byte 0xeb
    .byte start_of_setup-1f
1:
    //
    // rest of the header
    //
```

Here we can see `jmp` instruction opcode - `0xeb` to the `start_of_setup-1f` point. `Nf` notation means following: `2f` refers to the next local `2:` label. In our case it is label `1` which goes right after jump. It contains rest of setup header and right after setup header we can see `.entrytext` section which starts at `start_of_setup` label.

Actually it's first code which starts to execute besides previous jump instruction. After kernel setup got the control from bootloader, first `jmp` instruction is located at `0x200` (first 512 bytes) offset from the start of kernel real mode. This we can read in linux kernel boot protocol and also see in grub2 source code:

```
  state.gs = state.fs = state.es = state.ds = state.ss = segment;
  state.cs = segment + 0x20;
```

It means that segment registers will have following values after kernel setup starts to work:

```
fs = es = ds = ss = 0x1000
cs = 0x1020
```

for my case when kernel loaded at `0x10000`.

After jump to `start_of_setup`, needs to do following things:

- Be sure that all values of all segment registers are equal
- Setup correct stack if need

- Setup bss
- Jump to C code at main.c

Let's look at implementation.

Segment registers align

First of all it ensures that `ds` and `es` segment registers point to the same address and enables interrupts with `sti` instruction:

```
    movw    %ds, %ax
    movw    %ax, %es
    sti
```

As i wrote above, grub2 loads kernel setup code at `0x10000` address and `cs` at `0x1020` because execution doesn't start from the start of file, but from:

```
_start:
    .byte 0xeb
    .byte start_of_setup-1f
```

jump, which is 512 bytes offset from the 4d 5a. Also need to align `cs` from 0x10200 to 0x10000 as all other segment registers. After that we setup stack:

```
    pushw    %ds
    pushw    $6f
    lretw
```

push `ds` value to stack, and address of 6 label and execute `lretw` instruction. When we call `lretw`, it loads address of `6` label to instruction pointer register and `cs` with value of `ds`. After it we will have `ds` and `cs` with the same values.

Stack setup

Actually, almost all of the setup code is preparation for C language environment in the real mode. The next step is checking of `ss` register value and making of correct stack if `ss` is wrong:

```
    movw    %ss, %dx
    cmpw    %ax, %dx
    movw    %sp, %dx
    je    2f
```

Generally, it can be 3 different cases:

- `ss` has valid value 0x10000 (as all other segment registers beside `cs`)
- `ss` is invalid and `CAN_USE_HEAP` flag is set (see below)
- `ss` is invalid and `CAN_USE_HEAP` flag is not set (see below)

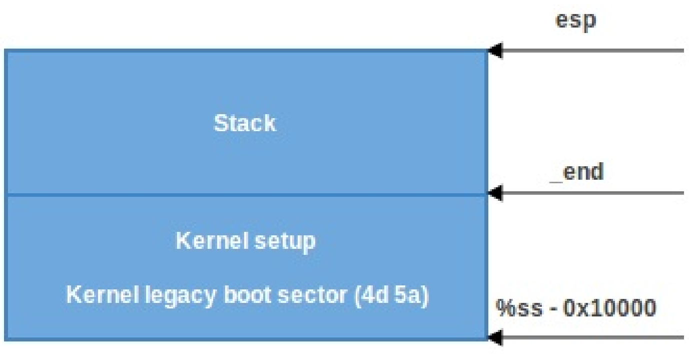
Let's look at all of these cases:

1. `ss` has a correct address (0x10000). In this case we go to 2 label:

```
2:     andw    $~3, %dx
    jnz    3f
    movw    $0xfffc, %dx
```

```
3:  movw    %ax, %ss
    movzwl %dx, %esp
    sti
```

Here we can see aligning of `dx` (contains `sp` given by bootloader) to 4 bytes and checking that it is not zero. If it is zero we put `0xfffc` (4 byte aligned address before maximum segment size - 64 KB) to `dx`. If it is not zero we continue to use `sp` given by bootloader (0xf7f4 in my case). After this we put `ax` value to `ss` which stores correct segment address `0x10000` and set up correct `sp`. After it we have correct stack:



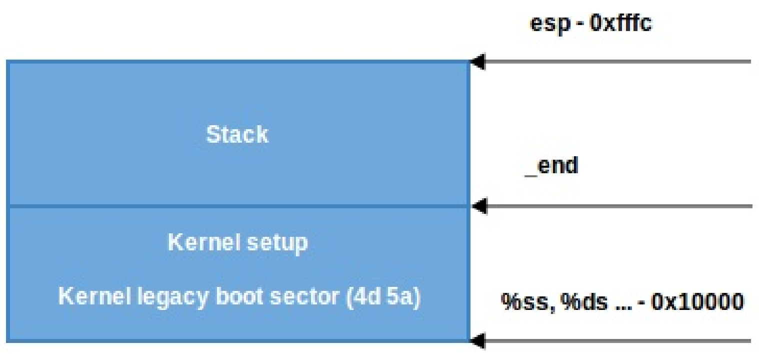
1. In the second case (`ss` != `ds`), first of all put _end (address of end of setup code) value in `dx`. And check `loadflags` header field with `testb` instruction too see if we can use heap or not. loadflags is a bitmask header which is defined as:

```
#define LOADED_HIGH        (1<<0)
#define QUIET_FLAG        (1<<5)
#define KEEP_SEGMENTS    (1<<6)
#define CAN_USE_HEAP    (1<<7)
```

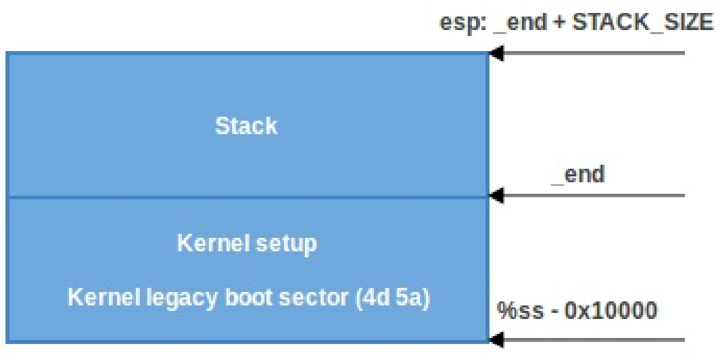
And as we can read in the boot protocol:

```
Field name:    loadflags

  This field is a bitmask.

  Bit 7 (write): CAN_USE_HEAP
    Set this bit to 1 to indicate that the value entered in the
    heap_end_ptr is valid.  If this field is clear, some setup code
    functionality will be disabled.
```

If `CAN_USE_HEAP` bit is set, put `heap_end_ptr` to `dx` which points to `_end` and add `STACK_SIZE` (minimal stack size - 512 bytes) to it. After this if `dx` is not carry, jump to `2` (it will be not carry, dx = _end + 512) label as in previous case and make correct stack.



1. The last case when `CAN_USE_HEAP` is not set, we just use minimal stack from `_end` to `_end + STACK_SIZE` :



Bss setup

Last two steps before we can jump to see code need to setup bss and check magic signature. Signature checking:

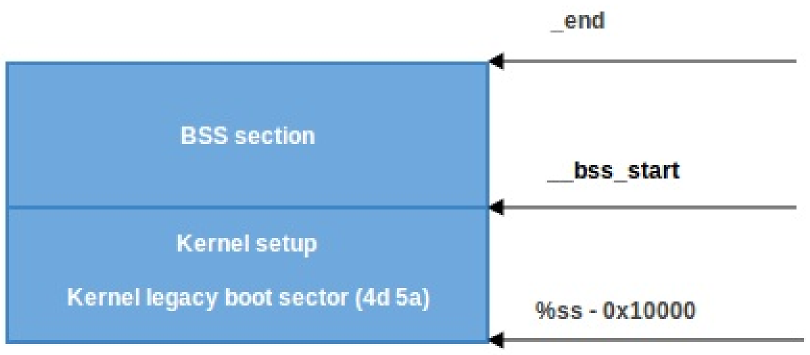
```
cmpl    $0x5a5aaa55, setup_sig
jne     setup_bad
```

just consists of comparing of setup_sig and `0x5a5aaa55` number, and if they are not equal jump to error printing.

Ok now we have correct segment registers, stack, need only setup bss and jump to C code. Bss section used for storing statically allocated uninitialized data. Here is the code:

```
    movw    $__bss_start, %di
    movw    $_end+3, %cx
    xorl    %eax, %eax
    subw    %di, %cx
    shrw    $2, %cx
    rep; stosl
```

First of all we put __bss_start address in `di` and `_end + 3` (+3 - align to 4 bytes) in `cx` . Clear `eax` register with `xor` instruction and calculate size of BSS section (put in `cx`). Divide `cx` by 4 and repeat `cx` times `stosl` instruction which stores value of `eax` (it is zero) and increase `di` by the size of `eax` . In this way, we write zeros from `__bss_start` to `_end` :



Jump to main

That's all, we have stack, bss and now we can jump to `main` C function:

```
calll main
```

which is in arch/x86/boot/main.c. What will be there? We will see it in the next part.

Conclusion

This is the end of the first part about linux kernel internals. If you have questions or suggestions, ping me in twitter 0xAX, drop me email or just create issue. In the next part we will see first C code which executes in linux kernel setup, implementation of memory routines as memset, memcpy, `earlyprintk` implementation and early console initialization and many more.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-internals.

Links

- Intel 80386 programmer's reference manual 1986
- Minimal Boot Loader for Intel® Architecture
- 8086
- 80386
- Reset vector
- Real mode
- Linux kernel boot protocol
- CoreBoot developer manual
- Ralf Brown's Interrupt List
- Power supply
- Power good signal

Kernel booting process. Part 2.

First steps in the kernel setup

We started to dive into linux kernel internals in the previous part and saw the initial part of the kernel setup code. We stopped at the first call of the `main` function (which is the first function written in C) from arch/x86/boot/main.c. Here we will continue to research of the kernel setup code and see what is `protected mode`, some preparation for transition to it, the heap and console initialization, memory detection and many many more. So... Let's go ahead.

Protected mode

Before we can move to the native Intel64 Long mode, the kernel must switch the CPU into protected mode. What is protected mode? Protected mode was first added to the x86 architecture in 1982 and was the main mode of Intel processors from 80286 processor until Intel 64 and long mode. Main reason to move from the real mode that there is very limited access to the RAM. As you can remember from the previous part, there is only 2^20 bytes or 1 megabyte, or even less 640 kilobytes.

Protected mode brought many changes, but main is another memory management. 24-bit address bus was replaced with 32-bit address bus. It gives 4 gigabytes of physical address. Also paging support was added which we will see in the next parts.

Memory management in the protected mode is divided into two, almost independent parts:

- Segmentation
- Paging

Here we can see only segmentation. As you can read in previous part, address consists from two parts in the real mode:

- Base address of segment
- Offset from the segment base

And we can get physical address if we know these two parts by:

```
PhysicalAddress = Segment * 16 + Offset
```

Memory segmentation was completely redone in the protected mode. There are no 64 kilobytes fixed-size segments. All memory segments described by `Global Descriptor Table` (GDT) instead segment registers. GDT is a structure which contains in memory. There is no fixed place for it in memory, but it's address stored in the special register - `GDTR`. Later, when we will see the GDT loading in the linux kernel code. There will be operation for loading it in memory, something like:

```
lgdt gdt
```

where the `lgdt` instruction loads base address and limit of global descriptor table to the `GDTR` register. `GDTR` is 48-bit register and consists of two parts:

- size - 16 bit of global descriptor table;
- address - 32-bit of the global descriptor table.

Global descriptor table contains `descriptors` which describes memory segment. Every descriptor is 64-bit. General scheme of descriptor is:

```
31          24        19      16              7            0
------------------------------------------------------------
|             | |B| |A|       | |   | |0|E|W|A|            |
| BASE 31..24 |G|/|L|V| LIMIT |P|DPL|S|  TYPE | BASE 23:16 | 4
|             | |D| |L| 19..16| |   | |1|C|R|A|            |
------------------------------------------------------------
|                             |                            |
|        BASE 15..0           |       LIMIT 15..0          | 0
|                             |                            |
------------------------------------------------------------
```

Don't worry, i know that it looks little scary after real mode, but it's easy. Let's look on it closer:

1. Limit (0 - 15 bits) defines a `length_of_segment - 1`. It depends on `G` bit.
    - if `G` (55-bit) is 0 and segment limit is 0 - size of segment - 1 byte
    - if `G` is 1 and segment limit is 0 - size of segment 4096 bytes
    - if `G` is 0 and segment limit is 0xfffff - size of segment 1 megabyte
    - if `G` is 1 and segment limit is 0xfffff - size of segment 4 gigabytes
2. Base (0-15, 32-39 and 56-63 bits) defines physical address of segment's start address.

3. Type (40-47 bits) defines type of segment and kinds of access to it. Next `S` flag specifies descriptor type. if `S` is 0 - this segment is system segment, if `S` is 1 - code or data segment (Stack segments are data segments which must be read/write segments). If segment is a code or data segment, it can be one of the following access types:

```
|           Type Field        | Descriptor Type | Description
|-----------------------------|-----------------|------------------
| Decimal                     |                 |
|             0    E    W   A |                 |
| 0           0    0    0   0 | Data            | Read-Only
| 1           0    0    0   1 | Data            | Read-Only, accessed
| 2           0    0    1   0 | Data            | Read/Write
| 3           0    0    1   1 | Data            | Read/Write, accessed
| 4           0    1    0   0 | Data            | Read-Only, expand-down
| 5           0    1    0   1 | Data            | Read-Only, expand-down, accessed
| 6           0    1    1   0 | Data            | Read/Write, expand-down
| 7           0    1    1   1 | Data            | Read/Write, expand-down, accessed
|                  C    R   A |                 |
| 8           1    0    0   0 | Code            | Execute-Only
| 9           1    0    0   1 | Code            | Execute-Only, accessed
| 10          1    0    1   0 | Code            | Execute/Read
| 11          1    0    1   1 | Code            | Execute/Read, accessed
| 12          1    1    0   0 | Code            | Execute-Only, conforming
| 14          1    1    0   1 | Code            | Execute-Only, conforming, accessed
| 13          1    1    1   0 | Code            | Execute/Read, conforming
| 15          1    1    1   1 | Code            | Execute/Read, conforming, accessed
```

As we can see first bit is 0 for data segment and 1 for code segment. Next three bits `EWA` are expansion direction (expand-down segment will grow down, more about it you can read here), write enable and accessed for data segments. `CRA` bits are conforming (A transfer of execution into a more-privileged conforming segment allows execution to continue at the current privilege level), read enable and accessed.

1. DPL (descriptor privilege level) defines the privilege level of the segment. I can be 0-3 where 0 is the most privileged.

2. P flag - indicates if segment is presented in memory or not.

3. AVL flag - Available and reserved bits.

4. L flag - indicates whether a code segment contains native 64-bit code. If 1 than code segment executes in 64 bit mode.

5. B/D flag - default operation size/default stack pointer size and/or upper bound.

Segment registers doesn't contain base address of the segment as in the real mode. Instead it contains special structure - `segment selector`. `Selector` is a 16-bit structure:

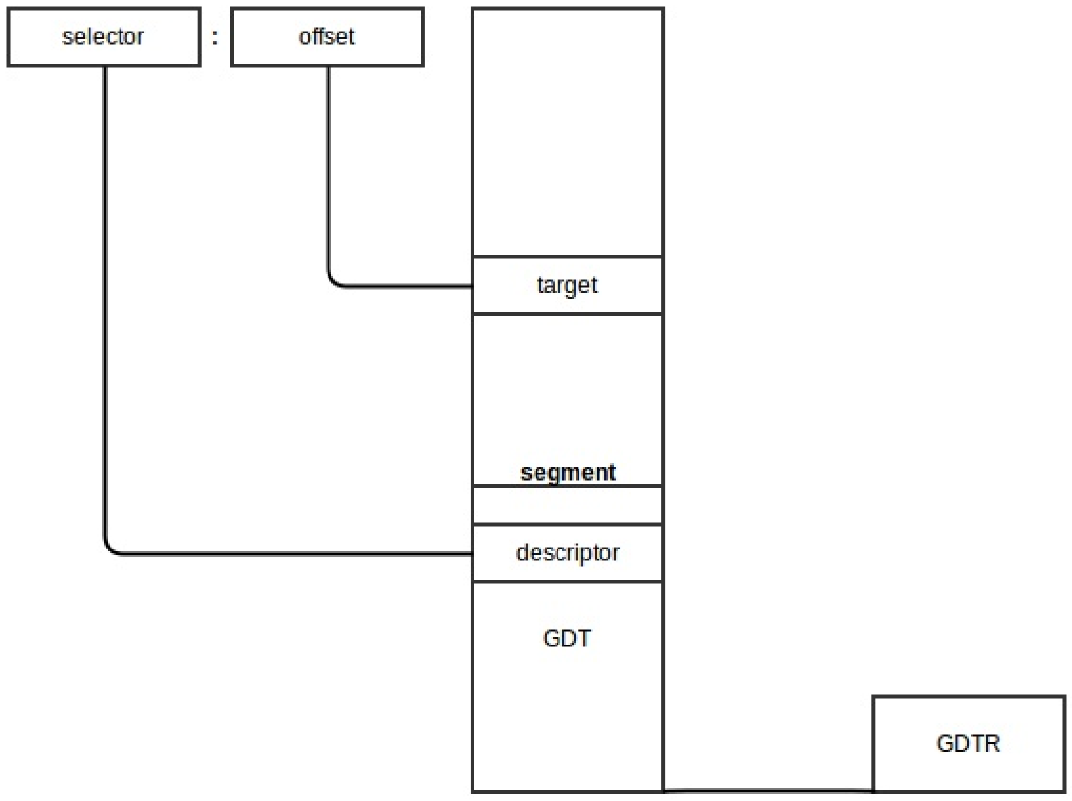
```
-----------------------------
|       Index    | TI | RPL |
-----------------------------
```

Where `Index` shows the index number of the descriptor in descriptor table. `TI` shows where to search the descriptor: in the global descriptor table or local. And `RPL` is privilege level.

Every segment register has visible and hidden part. When selector is loaded into one of the segment registers, it will be stored into visible part. Hidden part contains base address, limit and access information of descriptor which pointed by the selector. There are following steps for getting physical address in the protected mode:

- Segment selector must be loaded in one of the segment registers;
- CPU tries to find (by GDT address + Index from selector) and loads descriptor into the hidden part of segment register;
- Base address (from segment descriptor) + offset will be linear address of segment which is physical address (if paging is disabled).

Schematically it will look like this:



Algorithm for transition from the real mode into protected mode is:

- Disable interrupts;
- Describe and load GDT with `lgdt` instruction;
- Set PE (Protection Enable) bit in CR0 (Control Register 0);
- Jump to protected mode code;

We will see transition to the protected mode in the linux kernel in the next part, but before we can move to protected mode, we need to do some preparations.

Let's look on arch/x86/boot/main.c. We can see some routines there which make keyboard initialization, heap initialization and etc... Let's look on it.

Copying boot parameters into "zeropage"

We will start from the `main` routine in the main.c. First function which called in the `main` is a copy_boot_params. It copies kernel setup header into the field of `boot_params` structure which defined in the arch/x86/include/uapi/asm/bootparam.h.

`boot_params` structure contains `struct setup_header hdr` field. This structure contains the same fields as defined in linux boot protocol and filled by boot loader and also in the kernel building time. `copy_boot_params` function does two things: copies `hdr` from header.S to the `boot_params` structure in `setup_header` field and updates pointer to the kernel command line if the kernel was loaded with old command line protocol.

You can note that It copies `hdr` with `memcpy` function which is defined in the copy.S source file. Let's look on it:

```
GLOBAL(memcpy)
    pushw    %si
    pushw    %di
    movw    %ax, %di
    movw    %dx, %si
    pushw    %cx
    shrw    $2, %cx
    rep; movsl
    popw    %cx
    andw    $3, %cx
    rep; movsb
    popw    %di
    popw    %si
    retl
ENDPROC(memcpy)
```

Yeah, we just moved to C code and assembly again :) First of all we can see that `memcpy` and other routines which are defined here, starts and ends with the two macro: `GLOBAL` and `ENDPROC`. GLOBAL described in the arch/x86/include/asm/linkage.h which defines `globl` directive and label for it. ENDPROC described in the include/linux/linkage.h which marks `name` symbol as function name and ends with the size of the `name` symbol.

Implementation of the `memcpy` is easy. At first, it pushes values from `si` and `di` registers to the stack because their values will change in the `memcpy`, so push it in the stack to preserve their values. `memcpy` (and other functions in copy.S) uses `fastcall` calling conventions. So it gets incoming parameters from the `ax`, `dx` and `cx` registers. Call of `memcpy` looks as:

```
memcpy(&boot_params.hdr, &hdr, sizeof hdr);
```

So `ax` will contain address of the `boot_params.hdr`, `dx` will contain address of the `hdr` and `cx` will contain size of the `hdr` in bytes. memcpy puts address of `boot_params.hdr` to the `di` register and address of `hdr` to `si` and saves size in stack. After this it shifts to the right on 2 size (or divide on 4) and copies from `si` to `di` by 4 bytes. After it we restore size of `hdr` again, align it by 4 bytes and copy rest of bytes from `si` to `di` by one byte (if there is rest). Restore `si` and `di` values from the stack in the end and after this copying finished.

Console initialization

After the `hdr` has copied into the `boot_params.hdr`, next step is console initialization by call of `console_init` function which defined in the arch/x86/boot/early_serial_console.c.

It tries to find `earlyprintk` option in the command line and if the search was successful, it parses port address and baud rate of the serial port and initializes serial port. Value of `earlyprintk` command line option can be one of the:

```
* serial,0x3f8,115200
* serial,ttyS0,115200
* ttyS0,115200
```

After serial port initialization we can see first output:

```
if (cmdline_find_option_bool("debug"))
        puts("early console in setup code\n");
```

`puts` definition is in the tty.c. As we can see it prints character by character in the loop with calling of `putchar` function. Let's look on `putchar` implementation:

```
void __attribute__((section(".inittext"))) putchar(int ch)
{
    if (ch == '\n')
        putchar('\r');

    bios_putchar(ch);

    if (early_serial_base != 0)
        serial_putchar(ch);
}
```

`__attribute__((section(".inittext")))` means that this code will be in the .inittext section. We can find it in the linker file setup.ld.

First of all, `put_char` checks `\n` symbol and if it is, prints `\r` before. After it puts characters on VGA by bios with `0x10` interruption call:

```
static void __attribute__((section(".inittext"))) bios_putchar(int ch)
{
    struct biosregs ireg;

    initregs(&ireg);
    ireg.bx = 0x0007;
    ireg.cx = 0x0001;
    ireg.ah = 0x0e;
    ireg.al = ch;
    intcall(0x10, &ireg, NULL);
}
```

Where `initregs` takes `biosregs` structure and fills `biosregs` with zeros with `memset` function at first and than fills it with registers values.

```
    memset(reg, 0, sizeof *reg);
    reg->eflags |= X86_EFLAGS_CF;
    reg->ds = ds();
    reg->es = ds();
    reg->fs = fs();
    reg->gs = gs();
```

Let's look on the memset implementation:

```
GLOBAL(memset)
    pushw    %di
    movw    %ax, %di
    movzbl    %dl, %eax
    imull    $0x01010101,%eax
    pushw    %cx
    shrw    $2, %cx
    rep; stosl
    popw    %cx
    andw    $3, %cx
    rep; stosb
    popw    %di
    retl
ENDPROC(memset)
```

As you can read above, it uses `fastcall` calling conventions as the `memcpy` function, it means that function gets parameters from `ax`, `dx` and `cx` registers.

Generally `memset` is like a memcpy implementation. It saves value of `di` register on the stack and puts `ax` value into `di` which is the address of `biosregs` structure. Next is `movzbl` instruction, which copies `dl` value to the low 2 bytes of `eax` register. The rest 2 high bytes of `eax` will be filled by zeros.

The next instruction multiplies `eax` value on the `0x01010101`. It needs because `memset` will copy by 4 bytes at the time. For example we need to fill structure with `0x7` byte with memset. `eax` will contain `0x00000007` value in this case. So if we multiply `eax` on `0x01010101`, we will get `0x07070707` and now we can copy this 4 bytes into structure. `memset` uses `rep; stosl` instructions for copying `eax` into `es:di`.

The rest of the `memset` function does almost the same as `memcpy`.

After that `biosregs` structure filled with `memset`, `bios_putchar` calls 0x10 interruption which prints a character. After it checks was serial port initialized or not and writes a character there with serial_putchar and `inb/outb` instructions if it was set.

Heap initialization

After the stack and bss section were prepared in the header.S (see previous part), need to initialize the heap with the init_heap function.

First of all `init_heap` checks `CAN_USE_HEAP` flag from the `loadflags` kernel setup header and calculates end of the stack if this flag was set:

```
    char *stack_end;

    if (boot_params.hdr.loadflags & CAN_USE_HEAP) {
        asm("leal %P1(%%esp),%0"
            : "=r" (stack_end) : "i" (-STACK_SIZE));
```

or in other words `stack_end = esp - STACK_SIZE`.

Than there is `heap_end` calculation which is `heap_end_ptr` or `_end` + 512 and check if `heap_end` greater that `stack_end` makes it equal.

From this moment we can use the heap in the kernel setup code. We will see how to use it and how implemented API for it in next posts.

CPU validation

The next step as we can see cpu validation by `validate_cpu` from arch/x86/boot/cpu.c.

It calls `check_cpu` function and passes cpu level and required cpu level to it and checks that kernel launched at the right cpu. It checks cpu's flags, presence of long mode (which we will see more details about it in the next parts) for x86_64, checks processor's vendor and makes preparation for epy certain vendor like turning off SSE+SSE2 for amd if they are missing and etc...

Memory detection

The next step is memory detection by `detect_memory` function. It uses different programming interfaces for memory detection like `0xe820` , `0xe801` and `0x88` . We will see only implementation of 0xE820 here. Let's look on `detect_memory_e820` implementation from the arch/x86/boot/memory.c source file. First of all, `detect_memory_e820` function initializes `biosregs` structure as we saw above and filled registers with special values for `0xe820` call:

```
    initregs(&ireg);
    ireg.ax  = 0xe820;
    ireg.cx  = sizeof buf;
    ireg.edx = SMAP;
    ireg.di  = (size_t)&buf;
```

`ax` register must contain number of the function (0xe820 in our case), `cx` register contains size of the buffer which will contain data about memory, `edx` must contain `SMAP` magic number, `es:di` must contain address of the buffer which will contain memory data and `ebx` must be zero.

The next is loop, where will be collected data about the memory. It starts from the call of the 0x15 bios interruption, which writes one line from the address allocation table. For getting the next line need to call this interruption again (what we do in the loop). Before the every next call `ebx` must contain value returned by previous value:

```
    intcall(0x15, &ireg, &oreg);
    ireg.ebx = oreg.ebx;
```

Ultimately, it makes iterations in the loop, collecting data from address allocation table and writes this data into array of `e820entry` array:

- start of memory segment
- size of memory segment
- type of memory segment (which can be reserved, usable and etc...).

You can see the result of execution in the `dmesg` output, something like:

```
[    0.000000] e820: BIOS-provided physical RAM map:
[    0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable
[    0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff] reserved
[    0.000000] BIOS-e820: [mem 0x0000000000100000-0x000000003ffdffff] usable
[    0.000000] BIOS-e820: [mem 0x000000003ffe0000-0x000000003fffffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000fffc0000-0x00000000ffffffff] reserved
```

Keyboard initialization

The next step is initialization of the keyboard with the call of `keyboard_init` function. At first `keyboard_init` initializes registers with `initregs` function and call the 0x16 interruption for getting keyboard status. After this it calls the 0x16 again for setting repeat rate and delay.

Querying

The next couple of steps are queries of different parameters. We will not dive into details about these queries, but will back to the all of it in the next parts. Let's make a short look on this functions:

The query_mca routine calls 0x15 BIOS interruption for getting machine model number, sub-model number, BIOS revision level, and other hardware-specific attributes:

```
int query_mca(void)
{
    struct biosregs ireg, oreg;
    u16 len;

    initregs(&ireg);
    ireg.ah = 0xc0;
    intcall(0x15, &ireg, &oreg);

    if (oreg.eflags & X86_EFLAGS_CF)
        return -1;    /* No MCA present */

    set_fs(oreg.es);
    len = rdfs16(oreg.bx);

    if (len > sizeof(boot_params.sys_desc_table))
        len = sizeof(boot_params.sys_desc_table);

    copy_from_fs(&boot_params.sys_desc_table, oreg.bx, len);
    return 0;
}
```

It fills `ah` register with `0xc0` value and calls the `0x15` BIOS interruption. After the interruption execution it checks carry flag flag and if it set to 1, BIOS doesn't support `MCA`. If carry flag is et to 0, `ES:BX` will contain pointer to the system information table, which looks like this:

```
Offset    Size    Description    )
 00h    WORD    number of bytes following
 02h    BYTE    model (see #00515)
 03h    BYTE    submodel (see #00515)
 04h    BYTE    BIOS revision: 0 for first release, 1 for 2nd, etc.
 05h    BYTE    feature byte 1 (see #00510)
 06h    BYTE    feature byte 2 (see #00511)
 07h    BYTE    feature byte 3 (see #00512)
 08h    BYTE    feature byte 4 (see #00513)
 09h    BYTE    feature byte 5 (see #00514)
---AWARD BIOS---
 0Ah  N BYTEs    AWARD copyright notice
---Phoenix BIOS---
 0Ah    BYTE    ??? (00h)
 0Bh    BYTE    major version
 0Ch    BYTE    minor version (BCD)
 0Dh  4 BYTEs    ASCIZ string "PTL" (Phoenix Technologies Ltd)
---Quadram Quad386---
 0Ah 17 BYTEs    ASCII signature string "Quadram Quad386XT"
---Toshiba (Satellite Pro 435CDS at least)---
 0Ah  7 BYTEs    signature "TOSHIBA"
 11h    BYTE    ??? (8h)
 12h    BYTE    ??? (E7h) product ID??? (guess)
 13h  3 BYTEs    "JPN"
```

Next we call `set_fs` routine and pass the value of `es` register to it. Implementation of the `set_fs` is pretty simple:

```
static inline void set_fs(u16 seg)
{
    asm volatile("movw %0,%%fs" : : "rm" (seg));
}
```

There is inline assembly which gets value of the `seg` parameter and puts it to the `fs` register. There are many functions in the boot.h, like `set_fs`, there are `set_gs`, `fs`, `gs` for reading value in it and etc...

In the end of `query_mca` it just copies table which pointed by `es:bx` to the `boot_params.sys_desc_table`.

The next is getting Intel SpeedStep information with the call of `query_ist` function. First of all it checks cpu level and if it is correct, calls `0x15` for getting info and saves result to `boot_params`.

The following query_apm_bios function which gets Advanced Power Management information from the BIOS. `query_apm_bios` calls the `0x15` BIOS interruption too, but with `ah` - `0x53` to check `APM` installation. After the `0x15` execution, `query_apm_bios` functions checks `PM` signature (it must be `0x504d`), carry flag (it must be 0 if `APM` supported) and value of the `cx` register (if it's 0x02, protected mode interface supported).

The next it calls the `0x15` again, but with `ax = 0x5304` for disconnecting `APM` interface and connects 32bit protected mode interface. In the end it fills `boot_params.apm_bios_info` with values obtained from the BIOS.

Note that `query_apm_bios` will be executed only if `CONFIG_APM` or `CONFIG_APM_MODULE` was set in configuration file:

```
#if defined(CONFIG_APM) || defined(CONFIG_APM_MODULE)
    query_apm_bios();
#endif
```

The last is query_edd function, which asks `Enhanced Disk Drive` information from the BIOS. Let's look on `query_edd` implementation.

First of all it reads edd option from kernel's command line and if it was set to `off` than `query_edd` just returns.

If EDD is enabled, `query_edd` goes over BIOS-supported hard disks and queries EDD information in the loop:

```
    for (devno = 0x80; devno < 0x80+EDD_MBR_SIG_MAX; devno++) {
        if (!get_edd_info(devno, &ei) && boot_params.eddbuf_entries < EDDMAXNR) {
            memcpy(edp, &ei, sizeof ei);
            edp++;
            boot_params.eddbuf_entries++;
        }
        ...
        ...
        ...
```

where the `0x80` is the first hard drive and the `EDD_MBR_SIG_MAX` macro is 16. It collects data to the array of edd_info structures. `get_edd_info` checks that presence of EDD by invoking `0x13` interruption with `ah` is `0x41` and If EDD is presented, `get_edd_info` again calls `0x13` interruption, but with `ah` is `0x48` and `si` address of buffer where EDD informantion will be stored.

Conclusion

It is the end of the second part about linux kernel internals. In next part we will see setting of video mode and the rest of preparation before transition to protected mode and directly transition into it.

If you have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-internals.

Links

- Protected mode
- Long mode
- How to Use Expand Down Segments on Intel 386 and Later CPUs
- earlyprintk documentation
- Kernel Parameters
- Serial console
- Intel SpeedStep
- APM
- EDD specification
- Previous part

Kernel booting process. Part 3.

Video mode initialization and transition to protected mode

This is the third part of the `Kernel booting process` series. In the previous part, we stopped right before the call of the `set_video` routine from the main.c. We will see video mode initialization in the kernel setup code, preparation before switching into the protected mode and transition into it in this part.

NOTE If you don't know anything about protected mode, you can find some information about it in the previous part. Also there are a couple of links which can help you.

As i wrote above, we will start from the `set_video` function which defined in the arch/x86/boot/video.c source code file. We can see that it starts with getting of video mode from the `boot_params.hdr` structure:

```
u16 mode = boot_params.hdr.vid_mode;
```

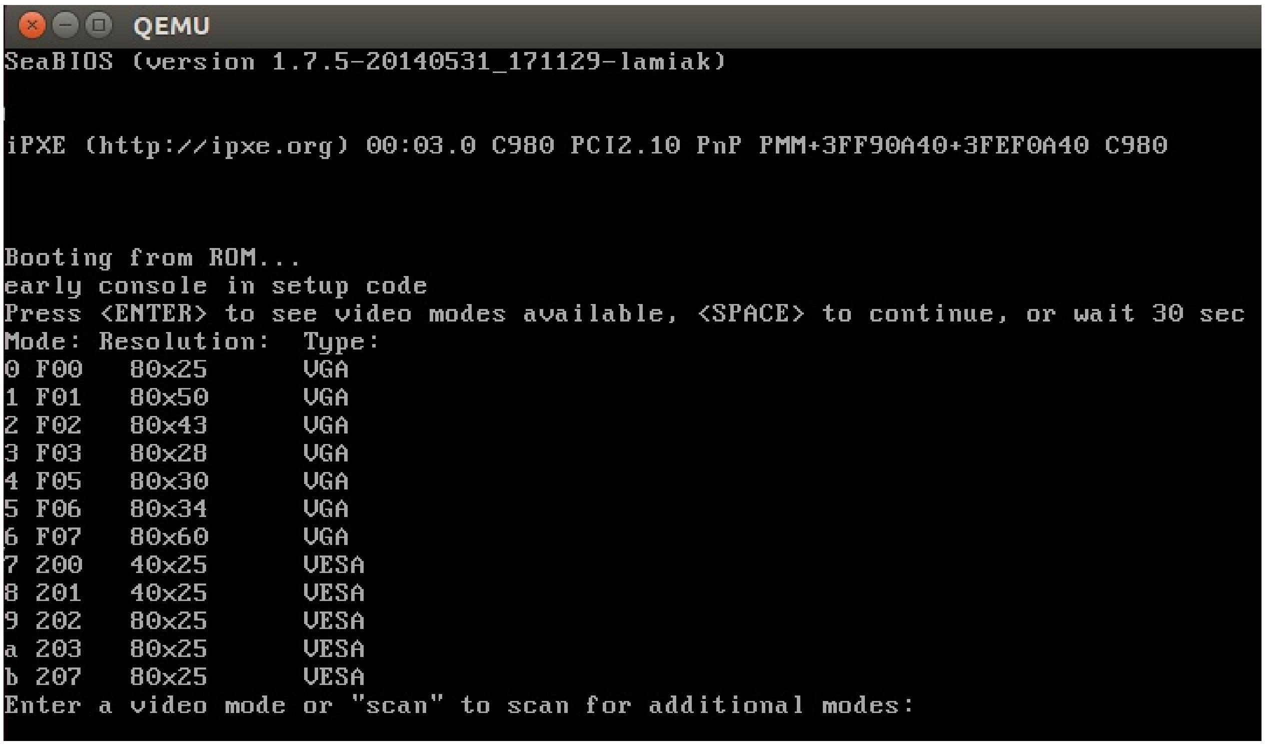
which we filled in the `copy_boot_params` function (you can read about it in the previous post). `vid_mode` is an obligatory field which filled by the bootloader. You can find information about it in the kernel boot protocol:

```
Offset    Proto    Name        Meaning
/Size
01FA/2    ALL        vid_mode    Video mode control
```

As we can read from the linux kernel boot protocol:

```
vga=<mode>
    <mode> here is either an integer (in C notation, either
    decimal, octal, or hexadecimal) or one of the strings
    "normal" (meaning 0xFFFF), "ext" (meaning 0xFFFE) or "ask"
    (meaning 0xFFFD).  This value should be entered into the
    vid_mode field, as it is used by the kernel before the command
    line is parsed.
```

So we can add `vga` option to the grub or another bootloader configuration file and it will pass this option to the kernel command line. This option can have different values as we can read from the description, for example it can be integer number or `ask`. If you will pass `ask`, you see menu like this:



```
SeaBIOS (version 1.7.5-20140531_171129-lamiak)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+3FF90A40+3FEF0A40 C980



Booting from ROM...
early console in setup code
Press <ENTER> to see video modes available, <SPACE> to continue, or wait 30 sec
Mode: Resolution:  Type:
0 F00   80x25      VGA
1 F01   80x50      VGA
2 F02   80x43      VGA
3 F03   80x28      VGA
4 F05   80x30      VGA
5 F06   80x34      VGA
6 F07   80x60      VGA
7 200   40x25      VESA
8 201   40x25      VESA
9 202   80x25      VESA
a 203   80x25      VESA
b 207   80x25      VESA
Enter a video mode or "scan" to scan for additional modes:
```

which will suggest to select video mode. We will look on it's implementation, but before we must to know about another things.

Kernel data types

Earlier we saw definitions of different data types like `u16` and etc... in the kernel setup code. Let's look on a couple of data types provided by the kernel:

```
| Type | char | short | int | long | u8 | u16 | u32 | u64 |
|------|------|-------|-----|------|----|-----|-----|-----|
| Size |  1   |   2   |  4  |   8  |  1 |  2  |  4  |  8  |
```

If you will read source code of the kernel, you'll see it very often, so it will be good to remember about it.

Heap API

As we got `vid_mode` from the `boot_params.hdr`, we can see call of the `RESET_HEAP` in the `set_video` function. `RESET_HEAP` is a macro which defined in the boot.h and looks as:

```
#define RESET_HEAP() ((void *)( HEAP = _end ))
```

If you read second part, you can remember that we initialized the heap with the init_heap function. Since we can use heap, we have a couple functions for it which defined in the `boot.h`. They are:

```
#define RESET_HEAP()...
```

As we saw just now. It uses for resetting the heap by setting the `HEAP` variable equal to `_end`, where `_end` is just:

```
extern char _end[];
```

Next is `GET_HEAP` macro:

```
#define GET_HEAP(type, n) \
    ((type *)__get_heap(sizeof(type),__alignof__(type),(n)))
```

for heap allocation. It calls internal function `__get_heap` with 3 parameters:

- size of a type in bytes, which need be allocated
- next parameter shows how type of variable is aligned
- how many bytes to allocate

Implementation of `__get_heap` is:

```
static inline char *__get_heap(size_t s, size_t a, size_t n)
{
    char *tmp;

    HEAP = (char *)(((size_t)HEAP+(a-1)) & ~(a-1));
    tmp = HEAP;
    HEAP += s*n;
    return tmp;
}
```

and further we will see usage of it, something like this:

```
saved.data = GET_HEAP(u16, saved.x*saved.y);
```

Let's try to understand how `GET_HEAP` works. We can see here that `HEAP` (which equal to `_end` after `RESET_HEAP()`) is the address of aligned memory according to `a` parameter. After it we save memory address from `HEAP` to the `tmp` variable, move `HEAP` to the end of allocated block and return `tmp` which is start address of allocated memory.

And the last function is:

```
static inline bool heap_free(size_t n)
{
    return (int)(heap_end-HEAP) >= (int)n;
}
```

which subtracts value of the `HEAP` from the `heap_end` (we calculated it in the previous part) and returns 1 if there is enough memory for `n`.

That's all. Now we have simple API for heap and can setup video mode.

Setup video mode

Now we can move directly to video mode initialization. We stopped at the `RESET_HEAP()` call in the `set_video` function. The next call of `store_mode_params` which stores video mode parameters in the `boot_params.screen_info` structure which defined in the include/uapi/linux/screen_info.h. If we will look at `store_mode_params` function, we can see that it starts from the call of the `store_cursor_position` function. As you can understand from the function name, it gets information about cursor and stores it. First of all `store_cursor_position` initializes two variables which has type - `biosregs`, with `AH = 0x3` and calls `0x10` BIOS interruption. After interruption successfully executed, it returns row and column in the `DL` and `DH` registers. Row and column will be stored in the `orig_x` and `orig_y` fields from the the `boot_params.screen_info` structure. After `store_cursor_position` executed, `store_video_mode` function will be called. It just gets current video mode and stores it in the `boot_params.screen_info.orig_video_mode`.

After this, it checks current video mode and set the `video_segment`. After the BIOS transfers control to the boot sector, the following addresses are video memory:

```
0xB000:0x0000     32 Kb     Monochrome Text Video Memory
0xB800:0x0000     32 Kb     Color Text Video Memory
```

So we set the `video_segment` variable to `0xb000` if current video mode is MDA, HGC, VGA in monochrome mode or `0xb800` in color mode. After setup of the address of the video segment need to store font size in the `boot_params.screen_info.orig_video_points` with:

```
set_fs(0);
font_size = rdfs16(0x485);
boot_params.screen_info.orig_video_points = font_size;
```

First of all we put 0 to the `FS` register with `set_fs` function. We already saw functions like `set_fs` in the previous part. They are all defined in the boot.h. Next we read value which located at address `0x485` (this memory location used to get the font size) and save font size in the `boot_params.screen_info.orig_video_points`.

The next we get amount of columns and rows by address `0x44a` and stores they in the `boot_params.screen_info.orig_video_cols` and `boot_params.screen_info.orig_video_lines`. After this, execution of the `store_mode_params` is finished.

The next we can see `save_screen` function which just saves screen content to the heap. This function collects all data which we got in the previous functions like rows and columns amount and etc... to the `saved_screen` structure, which defined as:

```
static struct saved_screen {
    int x, y;
    int curx, cury;
    u16 *data;
} saved;
```

It checks that heap has free space for it with:

```
if (!heap_free(saved.x*saved.y*sizeof(u16)+512))
        return;
```

and allocates space in the heap if it is enough and stores `saved_screen` in it.

The next call is `probe_cards(0)` from the arch/x86/boot/video-mode.c. It goes over all video_cards and collects number of modes provided by the cards. Here is the interesting moment, we can see the loop:

```
for (card = video_cards; card < video_cards_end; card++) {
  /* collecting number of modes here */
}
```

but `video_cards` not declared anywhere. Answer is simple: Every video mode presented in the x86 kernel setup code has definition like this:

```
static __videocard video_vga = {
    .card_name    = "VGA",
    .probe        = vga_probe,
    .set_mode    = vga_set_mode,
};
```

where `__videocard` is a macro:

```
#define __videocard struct card_info __attribute__((used,section(".videocards")))
```

which means that `card_info` structure:

```
struct card_info {
    const char *card_name;
    int (*set_mode)(struct mode_info *mode);
    int (*probe)(void);
    struct mode_info *modes;
    int nmodes;
    int unsafe;
    u16 xmode_first;
    u16 xmode_n;
};
```

is in the `.videocards` segment. Let's look on the arch/x86/boot/setup.ld linker file, we can see there:

```
    .videocards    : {
        video_cards = .;
        *(.videocards)
        video_cards_end = .;
    }
```

It means that `video_cards` is just memory address and all `card_info` structures are placed in this segment. It means that all `card_info` structures are placed between `video_cards` and `video_cards_end`, so we can use it in a loop to go over all of it. After `probe_cards` executed we have all structures like `static __videocard video_vga` with filled `nmodes` (number of video modes).

After that `probe_cards` executed, we move to the main loop in the `setup_video` function. There is infinite loop which tries to setup video mode with the `set_mode` function or prints menu if we passed `vid_mode=ask` to the kernel command line or video mode is undefined.

The `set_mode` function is defined in the video-mode.c and gets only one parameter - `mode` which is number of video mode (we got it or from the menu or in the start of the `setup_video`, from kernel setup header).

`set_mode` function checks the `mode` and calls `raw_set_mode` function. The `raw_set_mode` calls `set_mode` function for selected card. We can get access to this function from the `card_info` structure, every video mode defines this structure with filled value which depends on video mode (for example for `vga` it is `video_vga.set_mode` function, see above example of `card_info` structure for `vga`). `video_vga.set_mode` is `vga_set_mode`, which checks vga mode and call function depend on mode:

```
static int vga_set_mode(struct mode_info *mode)
{
    vga_set_basic_mode();

    force_x = mode->x;
    force_y = mode->y;

    switch (mode->mode) {
    case VIDEO_80x25:
        break;
    case VIDEO_8POINT:
        vga_set_8font();
        break;
    case VIDEO_80x43:
        vga_set_80x43();
        break;
    case VIDEO_80x28:
        vga_set_14font();
        break;
    case VIDEO_80x30:
        vga_set_80x30();
        break;
    case VIDEO_80x34:
        vga_set_80x34();
        break;
    case VIDEO_80x60:
        vga_set_80x60();
        break;
    }
    return 0;
}
```

Every function which setups video mode, just call `0x10` BIOS interruption with certain value in the `AH` register. After this we have set video mode and now we can switch to the protected mode.

Last preparation before transition into protected mode

We can see the last function call - `go_to_protected_mode` in the main.c. As comment says: `Do the last things and invoke protected mode`, so let's see last preparation and switch into the protected mode.

`go_to_protected_mode` defined in the arch/x86/boot/pm.c. It contains some functions which make last preparations before we can jump into protected mode, so let's look on it and try to understand what they do and how it works.

At first we see call of `realmode_switch_hook` function in the `go_to_protected_mode`. This function invokes real mode switch hook if it is present and disables NMI. Hooks are used if bootloader runs in a hostile environment. More about hooks you can read in the boot protocol (see **ADVANCED BOOT LOADER HOOKS**). `readlmode_swtich` hook presents pointer to the 16-bit real mode far subroutine which disables non-maskable interruptions. After we checked `realmode_switch` hook (it doesn't present for me), there is disabling of non-maskable interruptions:

```
asm volatile("cli");
outb(0x80, 0x70);
io_delay();
```

At first there is inline assembly instruction with `cli` instruction which clears the interrupt flag (`IF`), after this external interrupts disabled. Next line disables NMI (non-maskable interruption). Interruption is a signal to the CPU which emitted by hardware or software. After getting signal, CPU stops to execute current instructions sequence and transfers control to the interruption handler. After interruption handler finished it's work, it transfers control to the interrupted instruction. Non-maskable interruptions (NMI) - interruptions which are always processed, independently of permission. We will not dive into details interruptions now, but will back to it in the next posts.

Let's back to the code. We can see that second line is writing `0x80` (disabled bit) byte to the `0x70` (CMOS Address register). And call the `io_delay` function after it. `io_delay` which initiates small delay and looks like:

```
static inline void io_delay(void)
{
    const u16 DELAY_PORT = 0x80;
    asm volatile("outb %%al,%0" : : "dN" (DELAY_PORT));
}
```

Outputting any byte to the port `0x80` should delay exactly 1 microsecond. Sow we can write any value (value from `AL` register in our case) to the `0x80` port. After this delay `realmode_switch_hook` function finished execution and we can move to the next function.

The next function is `enable_a20`, which enables A20 line. This function defined in the arch/x86/boot/a20.c and it tries to enable A20 gate with different methods. The first is `a20_test_short` function which checks is A20 already enabled or not with `a20_test` function:

```
static int a20_test(int loops)
```

```
{
    int ok = 0;
    int saved, ctr;

    set_fs(0x0000);
    set_gs(0xffff);

    saved = ctr = rdfs32(A20_TEST_ADDR);

    while (loops--) {
        wrfs32(++ctr, A20_TEST_ADDR);
        io_delay();    /* Serialize and make delay constant */
        ok = rdgs32(A20_TEST_ADDR+0x10) ^ ctr;
        if (ok)
            break;
    }

    wrfs32(saved, A20_TEST_ADDR);
    return ok;
}
```

First of all we put `0x0000` to the `FS` register and `0xffff` to the `GS` register. Next we read value by address `A20_TEST_ADDR` (it is `0x200`) and put this value into `saved` variable and `ctr` . Next we write updated `ctr` value into `fs:gs` with `wrfs32` function, make little delay, and read value into the `GS` register by address `A20_TEST_ADDR+0x10` , if it's not zero we already have enabled a20 line. If A20 is disabled, we try to enabled it with different method which you can find in the `a20.c` . For example with call of `0x15` BIOS interruption with `AH=0x2041` and etc... If `enabled_a20` function finished with fail, printed error message and called function `die` . You can remember it from the first source code file where we started - arch/x86/boot/header.S:

```
die:
    hlt
    jmp    die
    .size    die, .-die
```

After the a20 gate successfully enabled, there are reset coprocessor and mask all interrupts. And after all of this preparations, we can see actual transition into protected mode.

Setup Interrupt Descriptor Table

Then next ist setup of Interrupt Descriptor table (IDT). `setup_idt` :

```
static void setup_idt(void)
{
    static const struct gdt_ptr null_idt = {0, 0};
    asm volatile("lidtl %0" : : "m" (null_idt));
}
```

which setups `Interrupt descriptor table` (describes interrupt handlers and etc...). For now IDT is not installed (we will see it later), but now we just load IDT with `lidtl` instruction. `null_idt` contains address and size of IDT, but now they are just zero. `null_idt` is a `gdt_ptr` structure, it looks:

```
struct gdt_ptr {
    u16 len;
    u32 ptr;
} __attribute__((packed));
```

where we can see - 16-bit length of IDT and 32-bit pointer to it (More details about IDT and interruptions we will see in the next posts). `__attribute__((packed))` means here that size of `gdt_ptr` minimum as required. So size of the `gdt_ptr` will be 6 bytes here or 48 bits. (Next we will load pointer to the `gdt_ptr` to the `GDTR` register and you can remember from the previous post that it is 48-bits size).

Setup Global Descriptor Table

The next point is setup of the Global Descriptor Table (GDT). We can see `setup_gdt` function which setups GDT (you can read about it in the Kernel booting process. Part 2.). There is definition of the `boot_gdt` array in this function, which contains definition of the three segments:

```
    static const u64 boot_gdt[] __attribute__((aligned(16))) = {
        [GDT_ENTRY_BOOT_CS] = GDT_ENTRY(0xc09b, 0, 0xfffff),
        [GDT_ENTRY_BOOT_DS] = GDT_ENTRY(0xc093, 0, 0xfffff),
        [GDT_ENTRY_BOOT_TSS] = GDT_ENTRY(0x0089, 4096, 103),
    };
```

For code, data and TSS (Task state segment). We will not use task state segment for now, it was added there to make Intel VT happy as we can see in the comment line (if you're interesting you can find commit which describes it - here). Let's look on `boot_gdt`. First of all we can note that it has `__attribute__((aligned(16)))` attribute. It means that this structure will be aligned by 16 bytes. Let's look on simple example:

```
#include <stdio.h>

struct aligned {
    int a;
}__attribute__((aligned(16)));

struct nonaligned {
    int b;
};

int main(void)
{
    struct aligned    a;
    struct nonaligned na;

    printf("Not aligned - %zu \n", sizeof(na));
    printf("Aligned - %zu \n", sizeof(a));

    return 0;
}
```

Technically structure which contains one `int` field, must be 4 bytes, but here `aligned` structure will be 16 bytes:

```
$ gcc test.c -o test && test
Not aligned - 4
Aligned - 16
```

`GDT_ENTRY_BOOT_CS` has index - 2 here, `GDT_ENTRY_BOOT_DS` is `GDT_ENTRY_BOOT_CS + 1` and etc... It starts from 2, because first is a mandatory null descriptor (index - 0) and the second is not used (index - 1).

`GDT_ENTRY` is a macro which takes flags, base and limit and builds GDT entry. For example let's look on the code segment entry. `GDT_ENTRY` takes following values:

- base - 0
- limit - 0xfffff
- flags - 0xc09b

What does it mean? Segment's base address is 0, limit (size of segment) is - `0xffff` (1 MB). Let's look on flags. It is `0xc09b` and it will be:

```
1100 0000 1001 1011
```

in binary. Let's try to understand what every bit means. We will go through all bits from left to right:

- 1 - (G) granularity bit
- 1 - (D) if 0 16-bit segment; 1 = 32-bit segment
- 0 - (L) executed in 64 bit mode if 1
- 0 - (AVL) available for use by system software
- 0000 - 4 bit length 19:16 bits in the descriptor
- 1 - (P) segment presence in memory
- 00 - (DPL) - privilege level, 0 is the highest privilege
- 1 - (S) code or data segment, not a system segment
- 101 - segment type execute/read/
- 1 - accessed bit

You can know more about every bit in the previous post or in the Intel® 64 and IA-32 Architectures Software Developer's Manuals 3A.

After this we get length of GDT with:

```
gdt.len = sizeof(boot_gdt)-1;
```

We get size of `boot_gdt` and subtract 1 (the last valid address in the GDT).

Next we get pointer to the GDT with:

```
gdt.ptr = (u32)&boot_gdt + (ds() << 4);
```

Here we just get address of `boot_gdt` and add it to address of data segment shifted on 4 (remember we're in the real mode now).

In the last we execute `lgdtl` instruction to load GDT into GDTR register:

```
asm volatile("lgdtl %0" : : "m" (gdt));
```

Actual transition into protected mode

It is the end of `go_to_protected_mode` function. We loaded IDT, GDT, disable interruptions and now can switch CPU into protected mode. The last step we call `protected_mode_jump` function with two parameters:

```
protected_mode_jump(boot_params.hdr.code32_start, (u32)&boot_params + (ds() << 4));
```

which defined in the arch/x86/boot/pmjump.S. It takes two parameters:

- address of protected mode entry point
- address of `boot_params`

Let's look inside `protected_mode_jump`. As i wrote above, you can find it in the `arch/x86/boot/pmjump.S`. First parameter will be in `eax` register and second is in `edx`. First of all we put address of `boot_params` to the `esi` register and address of code segment register `cs` (0x1000) to the `bx`. After this we shift `bx` on 4 and add address of label `2` to it (we will have physical address of label `2` in the `bx` after it) and jump to label `1`. Next we put data segment and task state segment in the `cs` and `di` registers with:

```
movw    $__BOOT_DS, %cx
movw    $__BOOT_TSS, %di
```

As you can read above `GDT_ENTRY_BOOT_CS` has index 2 and every GDT entry is 8 byte, so `CS` will be `2 * 8 = 16`, `__BOOT_DS` is 24 and etc... Next we set `PE` (Protection Enable) bit in the `CR0` control register:

```
movl    %cr0, %edx
orb    $X86_CR0_PE, %dl
movl    %edx, %cr0
```

and make long jump to the protected mode:

```
    .byte    0x66, 0xea
2:    .long    in_pm32
    .word    __BOOT_CS
```

where `0x66` is the operand-size prefix, which allows to mix 16-bit and 32-bit code, `0xea` - is the jump opcode, `in_pm32` is the segment offset and `__BOOT_CS` is the segment.

After this we are in the protected mode:

```
.code32
.section ".text32","ax"
```

Let's look on the first steps in the protected mode. First of all we setup data segment with:

```
movl    %ecx, %ds
movl    %ecx, %es
movl    %ecx, %fs
movl    %ecx, %gs
movl    %ecx, %ss
```

if you read with attention, you can remember that we saved `$__BOOT_DS` in the `cx` register. Now we fill with it all segment registers besides `cs` (`cs` is already `__BOOT_CS`). Next we zero out all general purpose registers besides `eax` with:

```
xorl    %ecx, %ecx
xorl    %edx, %edx
xorl    %ebx, %ebx
xorl    %ebp, %ebp
xorl    %edi, %edi
```

And jump to the 32-bit entry point in the end:

```
jmpl    *%eax
```

remember that `eax` contains address of the 32-bit entry (we passed it as first parameter into `protected_mode_jump`). That's all we're in the protected mode and stops before it's entry point. What is happening after we joined in the 32-bit entry point we will see in the next part.

Conclusion

It is the end of the third part about linux kernel internals. In next part we will see first steps in the protected mode and transition into the long mode.

If you will have any questions or suggestions write me a comment or ping me at twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you will find any mistakes please send me PR to linux-internals.

Links

- VGA
- VESA BIOS Extensions
- Data structure alignment
- Non-maskable interrupt
- A20
- GCC designated inits
- GCC type attributes
- Previous part

Kernel booting process. Part 4.

Transition to 64-bit mode

It is the fourth part of the `Kernel booting process` and we will see first steps in the protected mode, like checking that cpu supports the long mode and SSE, paging and initialization of the page tables and transition to the long mode in in the end of this part.

NOTE: will be much assembly code in this part, so if you have poor knowledge, read a book about it

In the previous part we stopped at the jump to the 32-bit entry point in the arch/x86/boot/pmjump.S:

```
jmpl    *%eax
```

Remind that `eax` register contains the address of the 32-bit entry point. We can read about this point from the linux kernel x86 boot protocol:

```
When using bzImage, the protected-mode kernel was relocated to 0x100000
```

And now we can make sure that it is true. Let's look on registers value in 32-bit entry point:

```
eax            0x100000    1048576
ecx            0x0        0
edx            0x0        0
ebx            0x0        0
esp            0x1ff5c    0x1ff5c
ebp            0x0        0x0
esi            0x14470    83056
edi            0x0        0
eip            0x100000    0x100000
eflags         0x46        [ PF ZF ]
cs             0x10    16
ss             0x18    24
ds             0x18    24
es             0x18    24
fs             0x18    24
gs             0x18    24
```

We can see here that `cs` register contains - `0x10` (as you can remember from the previous part, it is the second index in the Global Descriptor Table), `eip` register is `0x100000` and base address of the all segments include code segment is zero. So we can get physical address, it will be `0:0x100000` or just `0x100000`, as in boot protocol. Now let's start with 32-bit entry point.

32-bit entry point

We can find definition of the 32-bit entry point in the arch/x86/boot/compressed/head_64.S:

```
    __HEAD
    .code32
ENTRY(startup_32)
....
....
....
ENDPROC(startup_32)
```

First of all why `compressed` directory? Actually `bzimage` is a gzipped `vmlinux + header + kernel setup code`. We saw the kernel setup code in the all of previous parts. So, the main goal of the `head_64.S` is to prepare for entering long mode, enter into it and decompress the kernel. We will see all of these steps besides kernel decompression in this part.

Also you can note that there are two files in the `arch/x86/boot/compressed` directory:

- head_32.S
- head_64.S

We will see only `head_64.S` because we are learning linux kernel for `x86_64`. `head_32.S` even not compiled in our case. Let's look on the arch/x86/boot/compressed/Makefile, we can see there following target:

```
vmlinux-objs-y := $(obj)/vmlinux.lds $(obj)/head_$(BITS).o $(obj)/misc.o \
    $(obj)/string.o $(obj)/cmdline.o \
    $(obj)/piggy.o $(obj)/cpuflags.o
```

Note on `$(obj)/head_$(BITS).o`. It means that compilation of the head_{32,64}.o depends on value of the `$(BITS)`. We can find it in the other Makefile - arch/x86/kernel/Makefile:

```
ifeq ($(CONFIG_X86_32),y)
        BITS := 32
        ...
        ...
else
        ...
        ...
        BITS := 64
endif
```

Now we know where to start, so let's do it.

Reload the segments if need

As i wrote above, we start in the arch/x86/boot/compressed/head_64.S. First of all we can see before `startup_32` definition:

```
    __HEAD
    .code32
ENTRY(startup_32)
```

`__HEAD` defined in the include/linux/init.h and looks as:

```
#define __HEAD        .section    ".head.text","ax"
```

We can find this section in the arch/x86/boot/compressed/vmlinux.lds.S linker script:

```
SECTIONS
{
    . = 0;
    .head.text : {
        _head = . ;
        HEAD_TEXT
        _ehead = . ;
    }
```

Note on `. = 0;`. `.` is a special variable of linker - location counter. Assigning a value to it, is an offset relative to the offset of the segment. As we assign zero to it, we can read from comments:

```
Be careful parts of head_64.S assume startup_32 is at address 0.
```

Ok, now we know where we are, and now the best time to look inside the `startup_32` function.

In the start of the `startup_32` we can see the `cld` instruction which clears `DF` flag. After this, string operations like `stosb` and other will increment the index registers `esi` or `edi`.

The Next we can see the check of `KEEP_SEGMENTS` flag from `loadflags`. If you remember we already saw `loadflags` in the `arch/x86/boot/head.S` (there we checked flag `CAN_USE_HEAP`). Now we need to check `KEEP_SEGMENTS` flag. We can find description of this flag in the linux boot protocol:

```
Bit 6 (write): KEEP_SEGMENTS
  Protocol: 2.07+
  - If 0, reload the segment registers in the 32bit entry point.
  - If 1, do not reload the segment registers in the 32bit entry point.
    Assume that %cs %ds %ss %es are all set to flat segments with
    a base of 0 (or the equivalent for their environment).
```

and if `KEEP_SEGMENTS` is not set, we need to set `ds`, `ss` and `es` registers to flat segment with base 0. That we do:

```
    testb $(1 << 6), BP_loadflags(%esi)
    jnz 1f

    cli
    movl    $(__BOOT_DS), %eax
    movl    %eax, %ds
    movl    %eax, %es
    movl    %eax, %ss
```

remember that `__BOOT_DS` is `0x18` (index of data segment in the Global Descriptor Table). If `KEEP_SEGMENTS` is not set, we jump to the label `1f` or update segment registers with `__BOOT_DS` if this flag is set.

If you read previous the part, you can remember that we already updated segment registers in the arch/x86/boot/pmjump.S, so why we need to set up it again? Actually linux kernel has also 32-bit boot protocol, so `startup_32` can be first function which will be executed right after a bootloader transfers control to the kernel.

As we checked `KEEP_SEGMENTS` flag and put the correct value to the segment registers, next step is calculate difference between where we loaded and compiled to run (remember that `setup.ld.S` contains `. = 0` at the start of the section):

```
    leal    (BP_scratch+4)(%esi), %esp
    call    1f
1:  popl    %ebp
    subl    $1b, %ebp
```

Here `esi` register contains address of the boot_params structure. `boot_params` contains special field `scratch` with offset `0x1e4`. We are getting address of the `scratch` field + 4 bytes and put it to the `esp` register (we will use it as stack for these calculations). After this we can see call instruction and `1f` label as operand of it. What does it mean `call`? It means that it pushes `ebp` value in the stack, next `esp` value, next function arguments and return address in the end. After this we pop return address from the stack into `ebp` register (`ebp` will contain return address) and subtract address of the previous label `1`.

After this we have address where we loaded in the `ebp` - `0x100000`.

Now we can setup the stack and verify CPU that it has support of the long mode and SSE.

stack setup and CPU verification

The next we can see assembly code which setups new stack for kernel decompression:

```
    movl    $boot_stack_end, %eax
    addl    %ebp, %eax
    movl    %eax, %esp
```

`boots_stack_end` is in the `.bss` section, we can see definition of it in the end of `head_64.S` :

```
    .bss
    .balign 4
boot_heap:
    .fill BOOT_HEAP_SIZE, 1, 0
boot_stack:
    .fill BOOT_STACK_SIZE, 1, 0
boot_stack_end:
```

First of all we put address of the `boot_stack_end` into `eax` register and add to it value of the `ebp` (remember that `ebp` now contains address where we loaded - `0x100000`). In the end we just put `eax` value into `esp` and that's all, we have correct stack pointer.

The next step is CPU verification. Need to check that CPU has support of `long mode` and `SSE` :

```
    call    verify_cpu
    testl    %eax, %eax
    jnz    no_longmode
```

It just calls `verify_cpu` function from the arch/x86/kernel/verify_cpu.S which contains a couple of calls of the `cpuid` instruction. `cpuid` is instruction which is used for getting information about processor. In our case it checks long mode and SSE support and returns `0` on success or `1` on fail in the `eax` register.

If `eax` is not zero, we jump to the `no_longmode` label which just stops the CPU with `hlt` instruction while any hardware interrupt will not happen.

```
no_longmode:
1:
    hlt
    jmp     1b
```

We set stack, cheked CPU and now can move on the next step.

Calculate relocation address

The next step is calculating relocation address for decompression if need. We can see following assembly code:

```
#ifdef CONFIG_RELOCATABLE
    movl    %ebp, %ebx
    movl    BP_kernel_alignment(%esi), %eax
    decl    %eax
    addl    %eax, %ebx
    notl    %eax
    andl    %eax, %ebx
    cmpl    $LOAD_PHYSICAL_ADDR, %ebx
    jge    1f
#endif
    movl    $LOAD_PHYSICAL_ADDR, %ebx
1:
    addl    $z_extract_offset, %ebx
```

First of all note on `CONFIG_RELOCATABLE` macro. This configuration option defined in the arch/x86/Kconfig and as we can read from it's description:

```
This builds a kernel image that retains relocation information
so it can be loaded someplace besides the default 1MB.

Note: If CONFIG_RELOCATABLE=y, then the kernel runs from the address
it has been loaded at and the compile time physical address
(CONFIG_PHYSICAL_START) is used as the minimum location.
```

In short words, this code calculates address where to move kernel for decompression put it to `ebx` register if the kernel is relocatable or bzimage will decompress itself above `LOAD_PHYSICAL_ADDR`.

Let's look on the code. If we have `CONFIG_RELOCATABLE=n` in our kernel configuration file, it just puts `LOAD_PHYSICAL_ADDR` to the `ebx` register and adds `z_extract_offset` to `ebx`. As `ebx` is zero for now, it will contain `z_extract_offset`. Now let's try to understand these two values.

`LOAD_PHYSICAL_ADDR` is the macro which defined in the arch/x86/include/asm/boot.h and it looks like this:

```
#define LOAD_PHYSICAL_ADDR ((CONFIG_PHYSICAL_START \
                + (CONFIG_PHYSICAL_ALIGN - 1)) \
                & ~(CONFIG_PHYSICAL_ALIGN - 1))
```

Here we calculates aligned address where kernel is loaded (`0x100000` or 1 megabyte in our case). `PHYSICAL_ALIGN` is an alignment value to which kernel should be aligned, it ranges from `0x200000` to `0x1000000` for x86_64. With the default values we will get 2 megabytes in the `LOAD_PHYSICAL_ADDR`:

```
>>> 0x100000 + (0x200000 - 1) & ~(0x200000 - 1)
2097152
```

After that we got alignment unit, we adds `z_extract_offset` (which is `0xe5c000` in my case) to the 2 megabytes. In the end we will get 17154048 byte offset. You can find `z_extract_offset` in the `arch/x86/boot/compressed/piggy.S`. This file generated in compile time by mkpiggy program.

Now let's try to understand the code if `CONFIG_RELOCATABLE` is `y`.

First of all we put `ebp` value to the `ebx` (remember that `ebp` contains address where we loaded) and `kernel_alignment` field from kernel setup header to the `eax` register. `kernel_alignment` is a physical address of alignment required for the kernel. Next we do the same as in the previous case (when kernel is not relocatable), but we just use value of the `kernel_alignment` field as align unit and `ebx` (address where we loaded) as base address instead of `CONFIG_PHYSICAL_ALIGN` and `LOAD_PHYSICAL_ADDR`.

After that we calculated address, we compare it with `LOAD_PHYSICAL_ADDR` and add `z_extract_offset` to it again or put `LOAD_PHYSICAL_ADDR` in the `ebx` if calculated address is less than we need.

After all of this calculation we will have `ebp` which contains address where we loaded and `ebx` with address where to move kernel for decompression.

Preparation before entering long mode

Now we need to do the last preparations before we can see transition to the 64-bit mode. At first we need to update Global Descriptor Table for this:

```
	leal	gdt(%ebp), %eax
	movl	%eax, gdt+2(%ebp)
	lgdt	gdt(%ebp)
```

Here we put the address from `ebp` with `gdt` offset to `eax` register, next we put this address into `ebp` with offset `gdt+2` and load Global Descriptor Table with the `lgdt` instruction.

Let's look on Global Descriptor Table definition:

```
	.data
gdt:
	.word	gdt_end - gdt
	.long	gdt
	.word	0
	.quad	0x0000000000000000	/* NULL descriptor */
	.quad	0x00af9a000000ffff	/* __KERNEL_CS */
	.quad	0x00cf92000000ffff	/* __KERNEL_DS */
	.quad	0x0080890000000000	/* TS descriptor */
	.quad   0x0000000000000000	/* TS continued */
```

It defined in the same file in the `.data` section. It contains 5 descriptors: null descriptor, for kernel code segment, kernel data segment and two task descriptors. We already loaded GDT in the previous part, we're doing almost the same here, but descriptors with `CS.L = 1` and `CS.D = 0` for execution in the 64 bit mode.

After we have loaded Global Descriptor Table, we must enable PAE mode with putting value of `cr4` register into `eax`, setting 5 bit in it and load it again in the `cr4`:

```
	movl	%cr4, %eax
	orl	$X86_CR4_PAE, %eax
	movl	%eax, %cr4
```

Now we finished almost with all preparations before we can move into 64-bit mode. The last step is to build page tables, but before some information about long mode.

Long mode

Long mode is the native mode for x86_64 processors. First of all let's look on some difference between `x86_64` and `x86`.

It provides some features as:

- New 8 general purpose registers from `r8` to `r15` + all general purpose registers are 64-bit now
- 64-bit instruction pointer - `RIP`
- New operating mode - Long mode
- 64-Bit Addresses and Operands
- RIP Relative Addressing (we will see example if it in the next parts)

Long mode is an extension of legacy protected mode. It consists from two sub-modes:

- 64-bit mode
- compatibility mode

To switch into 64-bit mode we need to do following things:

- enable PAE (we already did it, see above)
- build page tables and load the address of top level page table into `cr3` register
- enable `EFER.LME`
- enable paging

We already enabled `PAE` with setting the PAE bit in the `cr4` register. Now let's look on paging.

Early page tables initialization

Before we can move in the 64-bit mode, we need to build page tables, so, let's look on building of early 4G boot page tables.

NOTE: I will not describe theory of virtual memory here, if you need to know more about it, see links in the end

Linux kernel uses 4-level paging, and generally we build 6 page tables:

- One PML4 table
- One PDP table
- Four Page Directory tables

Let's look on the implementation of it. First of all we clear buffer for the page tables in the memory. Every table is 4096 bytes, so we need 24 kilobytes buffer:

```
    leal    pgtable(%ebx), %edi
    xorl    %eax, %eax
    movl    $((4096*6)/4), %ecx
    rep    stosl
```

We put address which stored in `ebx` (remember that `ebx` contains the address where to relocate kernel for decompression) with `pgtable` offset to the `edi` register. `pgtable` defined in the end of `head_64.S` and looks:

```
    .section ".pgtable","a",@nobits
    .balign 4096
pgtable:
    .fill 6*4096, 1, 0
```

It is in the `.pgtable` section and it size is 24 kilobytes. After we put address to the `edi`, we zero out `eax` register and writes zeros to the buffer with `rep stosl` instruction.

Now we can build top level page table - `PML4` with:

```
    leal    pgtable + 0(%ebx), %edi
    leal    0x1007 (%edi), %eax
    movl    %eax, 0(%edi)
```

Here we get address which stored in the `ebx` with `pgtable` offset and put it to the `edi`. Next we put this address with offset `0x1007` to the `eax` register. `0x1007` is 4096 bytes (size of the PML4) + 7 (PML4 entry flags - `PRESENT+RW+USER`) and puts `eax` to the `edi`. After this manipulations `edi` will contain the address of the first Page Directory Pointer Entry with flags - `PRESENT+RW+USER`.

In the next step we build 4 Page Directory entry in the Page Directory Pointer table, where first entry will be with `0x7` flags and other with `0x8`:

```
    leal    pgtable + 0x1000(%ebx), %edi
    leal    0x1007(%edi), %eax
    movl    $4, %ecx
1:  movl    %eax, 0x00(%edi)
    addl    $0x00001000, %eax
    addl    $8, %edi
    decl    %ecx
    jnz    1b
```

We put base address of the page directory pointer table to the `edi` and address of the first page directory pointer entry to the `eax`. Put `4` to the `ecx` register, it will be counter in the following loop and write the address of the first page directory pointer table entry to the `edi` register.

After this `edi` will contain address of the first page directory pointer entry with flags `0x7`. Next we just calculates address of following page directory pointer entries with flags `0x8` and writes their addresses to the `edi`.

The next step is building of `2048` page table entries by 2 megabytes:

```
    leal    pgtable + 0x2000(%ebx), %edi
    movl    $0x00000183, %eax
    movl    $2048, %ecx
1:  movl    %eax, 0(%edi)
    addl    $0x00200000, %eax
    addl    $8, %edi
    decl    %ecx
    jnz    1b
```

Here we do almost the same that in the previous example, just first entry will be with flags - `$0x00000183` - `PRESENT + WRITE + MBZ` and all another with `0x8`. In the end we will have 2048 pages by 2 megabytes.

Our early page table structure are done, it maps 4 gigabytes of memory and now we can put address of the high-level page table - `PML4` to the `cr3` control register:

```
    leal    pgtable(%ebx), %eax
    movl    %eax, %cr3
```

That's all now we can see transition to the long mode.

Transition to the long mode

First of all we need to set `EFER.LME` flag in the MSR to `0xC0000080`:

```
    movl    $MSR_EFER, %ecx
    rdmsr
    btsl    $_EFER_LME, %eax
    wrmsr
```

Here we put `MSR_EFER` flag (which defined in the arch/x86/include/uapi/asm/msr-index.h) to the `ecx` register and call `rdmsr` instruction which reads MSR register. After `rdmsr` executed, we will have result data in the `edx:eax` which depends on `ecx` value. We check `EFER_LME` bit with `btsl` instruction and write data from `eax` to the `MSR` register with `wrmsr` instruction.

In next step we push address of the kernel segment code to the stack (we defined it in the GDT) and put address of the `startup_64` routine to the `eax`.

```
    pushl    $__KERNEL_CS
    leal    startup_64(%ebp), %eax
```

After this we push this address to the stack and enable paging with setting `PG` and `PE` bits in the `cr0` register:

```
    movl    $(X86_CR0_PG | X86_CR0_PE), %eax
    movl    %eax, %cr0
```

and call:

```
lret
```

Remember that we pushed address of the `startup_64` function to the stack in the previous step, and after `lret` instruction, CPU extracts address of it and jumps there.

After all of these steps we're finally in the 64-bit mode:

```
    .code64
    .org 0x200
ENTRY(startup_64)
....
....
....
```

That's all!

Conclusion

This is the end of the fourth part linux kernel booting process. If you have questions or suggestions, ping me in twitter 0xAX, drop me email or just create an issue.

In the next part we will see kernel decompression and many more.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-internals.

Links

- Protected mode
- Intel® 64 and IA-32 Architectures Software Developer’s Manual 3A
- GNU linker
- SSE
- Paging
- Model specific register
- .fill instruction
- Previous part
- Paging on osdev.org
- Paging Systems
- x86 Paging Tutorial

Kernel booting process. Part 5.

Kernel decompression

This is the fifth part of the `Kernel booting process` series. We saw transition to the 64-bit mode in the previous part and we will continue from this point in this part. We will see the last steps before we jump to the kernel code as preparation for kernel decompression, relocation and directly kernel decompression. So... let's start to dive in the kernel code again.

Preparation before kernel decompression

We stoped right before jump on 64-bit entry point - `startup_64` which located in the arch/x86/boot/compressed/head_64.S source code file. As we saw a jump to the `startup_64` in the `startup_32` :

```
    pushl    $__KERNEL_CS
    leal    startup_64(%ebp), %eax
    ...
    ...
    ...
    pushl    %eax
    ...
    ...
    ...
    lret
```

in the previous part, `startup_64` starts to work. Since we loaded the new Global Descriptor Table and there was CPU transition in other mode (64-bit mode in our case), we can see setup of the data segments:

```
    .code64
    .org 0x200
ENTRY(startup_64)
    xorl    %eax, %eax
    movl    %eax, %ds
    movl    %eax, %es
    movl    %eax, %ss
    movl    %eax, %fs
    movl    %eax, %gs
```

in the start of `startup_64` . All segment registers besides `cs` points now to the `ds` which is `0x18` (if you don't understand why it is `0x18` , read the previous part).

The next step is computation of difference between where kernel was compiled and where it was loaded:

```
#ifdef CONFIG_RELOCATABLE
    leaq    startup_32(%rip), %rbp
    movl    BP_kernel_alignment(%rsi), %eax
    decl    %eax
    addq    %rax, %rbp
    notq    %rax
    andq    %rax, %rbp
    cmpq    $LOAD_PHYSICAL_ADDR, %rbp
    jge    1f
#endif
    movq    $LOAD_PHYSICAL_ADDR, %rbp
1:
    leaq    z_extract_offset(%rbp), %rbx
```

`rbp` contains decompressed kernel start address and after this code executed `rbx` register will contain address where to relocate the kernel code for decompression. We already saw code like this in the `startup_32` (you can read about it in the previous part - Calculate relocation address), but we need to do this calculation again because bootloader can use 64-bit boot protocol and `startup_32` just will not be executed in this case.

In the next step we can see setup of the stack and reset of flags register:

```
    leaq    boot_stack_end(%rbx), %rsp

     pushq    $0
    popfq
```

As you can see above `rbx` register contains the start address of the decompressing kernel code and we just put this address with `boot_stack_end` offset to the `rsp` register. After this stack will be correct. You can find definition of the `boot_stack_end` in the end of `compressed/head_64.S` file:

```
    .bss
    .balign 4
boot_heap:
    .fill BOOT_HEAP_SIZE, 1, 0
boot_stack:
    .fill BOOT_STACK_SIZE, 1, 0
boot_stack_end:
```

It located in the `.bss` section right before `.pgtable`. You can look at arch/x86/boot/compressed/vmlinux.lds.S to find it.

As we set the stack, now we can copy the compressed kernel to the address that we got above, when we calculated the relocation address of the decompressed kernel. Let's look on this code:

```
    pushq    %rsi
    leaq    (_bss-8)(%rip), %rsi
    leaq    (_bss-8)(%rbx), %rdi
    movq    $_bss, %rcx
    shrq    $3, %rcx
    std
    rep    movsq
    cld
    popq    %rsi
```

First of all we push `rsi` to the stack. We need save value of `rsi`, because this register now stores pointer to the `boot_params` real mode structure (you must remember this structure, we filled it in the start of kernel setup). In the end of this code we'll restore pointer to the `boot_params` into `rsi` again.

The next two `leaq` instructions calculates effective address of the `rip` and `rbx` with `_bss - 8` offset and put it to the `rsi` and `rdi`. Why we calculate this addresses? Actually compressed kernel image located between this copying code (from `startup_32` to the current code) and the decompression code. You can verify this by looking on the linker script - arch/x86/boot/compressed/vmlinux.lds.S:

```
    . = 0;
    .head.text : {
        _head = . ;
        HEAD_TEXT
        _ehead = . ;
    }
    .rodata..compressed : {
        *(.rodata..compressed)
    }
    .text :    {
        _text = .;     /* Text */
        *(.text)
        *(.text.*)
        _etext = . ;
    }
```

Note that `.head.text` section contains `startup_32`. You can remember it from the previous part:

```
    __HEAD
    .code32
ENTRY(startup_32)
...
...
...
```

`.text` section contains decompression code:

assembly

```
    .text
relocated:
...
...
...
/*
 * Do the decompression, and jump to the new kernel..
 */
...
```

And `.rodata..compressed` contains compressed kernel image.

So `rsi` will contain `rip` relative address of the `_bss - 8` and `rdi` will contain relocation relative address of the `` `_bss - 8 ``. As we store these addresses in register, we put the address of `_bss` to the `rcx` register. As you can see in the `vmlinux.lds.S`, it located in the end of all sections with the setup/kernel code. Now we can start to copy data from `rsi` to `rdi` by 8 bytes with `movsq` instruction.

Note that there is `std` instruction before data copying, it sets `DF` flag and it means that `rsi` and `rdi` will be decremeted or in other words, we will crbxopy bytes in backwards.

In the end we clear `DF` flag with `cld` instruction and restore `boot_params` structure to the `rsi`.

After it we get `.text` section address address and jump to it:

```
    leaq    relocated(%rbx), %rax
    jmp    *%rax
```

Last preparation before kernel decompression

`.text` sections starts with the `relocated` label. For the start there is clearing of the `bss` section with:

```
    xorl    %eax, %eax
    leaq    _bss(%rip), %rdi
    leaq    _ebss(%rip), %rcx
    subq    %rdi, %rcx
    shrq    $3, %rcx
    rep    stosq
```

Here we just clear `eax`, put RIP relative address of the `_bss` to the `rdi` and `_ebss` to `rcx` and fill it with zeros with `rep stosq` instructions.

In the end we can see the call of the `decompress_kernel` routine:

```
    pushq    %rsi
    movq    $z_run_size, %r9
    pushq    %r9
    movq    %rsi, %rdi
    leaq    boot_heap(%rip), %rsi
    leaq    input_data(%rip), %rdx
    movl    $z_input_len, %ecx
    movq    %rbp, %r8
    movq    $z_output_len, %r9
    call    decompress_kernel
    popq    %r9
    popq    %rsi
```

Again we save `rsi` with pointer to `boot_params` structure and call `decompress_kernel` from the arch/x86/boot/compressed/misc.c with seven arguments. All arguments will be passed through the registers. We finished all preparation and now can look on the kernel decompression.

Kernel decompression

As i wrote above, `decompress_kernel` function is in the arch/x86/boot/compressed/misc.c source code file. This function starts with the video/console initialization that we saw in the previous parts. This calls need if bootloaded used 32 or 64-bit protocols. After this we store pointers to the start of the free memory and to the end of it:

```
    free_mem_ptr     = heap;
    free_mem_end_ptr = heap + BOOT_HEAP_SIZE;
```

where `heap` is the second parameter of the `decompress_kernel` function which we got with:

```
leaq    boot_heap(%rip), %rsi
```

As you saw about `boot_heap` defined as:

```
boot_heap:
    .fill BOOT_HEAP_SIZE, 1, 0
```

where `BOOT_HEAP_SIZE` is `0x400000` if the kernel compressed with `bzip2` or `0x8000` if not.

In the next step we call `choose_kernel_location` function from the arch/x86/boot/compressed/aslr.c. As we can understand from the function name it chooses memory location where to decompress the kernel image. Let's look on this function.

At the start `choose_kernel_location` tries to find `kaslr` option in the command line if `CONFIG_HIBERNATION` is set and `nokaslr` option if this configuration option `CONFIG_HIBERNATION` is not set:

```
#ifdef CONFIG_HIBERNATION
    if (!cmdline_find_option_bool("kaslr")) {
        debug_putstr("KASLR disabled by default...\n");
        goto out;
    }
#else
    if (cmdline_find_option_bool("nokaslr")) {
        debug_putstr("KASLR disabled by cmdline...\n");
        goto out;
    }
#endif
```

If there is no `kaslr` or `nokaslr` in the command line it jumps to `out` label:

```
out:
    return (unsigned char *)choice;
```

which just returns the `output` parameter which we passed to the `choose_kernel_location` without any changes. Let's try to understand what is it `kaslr`. We can find information about it in the documentation:

```
kaslr/nokaslr [X86]

Enable/disable kernel and module base offset ASLR
(Address Space Layout Randomization) if built into
the kernel. When CONFIG_HIBERNATION is selected,
kASLR is disabled by default. When kASLR is enabled,
hibernation will be disabled.
```

It means that we can pass `kaslr` option to the kernel's command line and get random address for the decompressed kernel (more about aslr you can read here).

Let's consider the case when kernel's command line contains `kaslr` option.

There is the call of the `mem_avoid_init` function from the same `aslr.c` source code file. This function gets the unsafe memory regions (initrd, kernel command line and etc...). We need to know about this memory regions to not overlap them with the kernel after decompression. For example:

```
    initrd_start  = (u64)real_mode->ext_ramdisk_image << 32;
    initrd_start |= real_mode->hdr.ramdisk_image;
    initrd_size  = (u64)real_mode->ext_ramdisk_size << 32;
    initrd_size |= real_mode->hdr.ramdisk_size;
    mem_avoid[1].start = initrd_start;
    mem_avoid[1].size = initrd_size;
```

Here we can see calculation of the initrd start address and size. `ext_ramdisk_image` is high 32-bits of the `ramdisk_image` field from boot header and `ext_ramdisk_size` is high 32-bits of the `ramdisk_size` field from boot protocol:

```
Offset    Proto    Name        Meaning
/Size
...
...
...
0218/4    2.00+    ramdisk_image    initrd load address (set by boot loader)
021C/4    2.00+    ramdisk_size    initrd size (set by boot loader)
...
```

And `ext_ramdisk_image` and `ext_ramdisk_size` you can find in the Documentation/x86/zero-page.txt:

```
Offset    Proto    Name        Meaning
/Size
...
...
...
0C0/004    ALL    ext_ramdisk_image ramdisk_image high 32bits
0C4/004    ALL    ext_ramdisk_size  ramdisk_size high 32bits
...
```

So we're taking `ext_ramdisk_image` and `ext_ramdisk_size`, shifting they left on 32 (now they will contain low 32-bits in the high 32-bit bits) and getting start address of the `initrd` and size of it. After this we store these values in the `mem_avoid` array which defined as:

```
#define MEM_AVOID_MAX 5
static struct mem_vector mem_avoid[MEM_AVOID_MAX];
```

where `mem_vector` structure is:

```
struct mem_vector {
    unsigned long start;
    unsigned long size;
};
```

The next step after we collected all unsafe memory regions in the `mem_avoid` array will be search of the random address which does not overlap with the unsafe regions with the `find_random_addr` function.

First of all we can see allign of the output address in the `find_random_addr` function:

```
minimum = ALIGN(minimum, CONFIG_PHYSICAL_ALIGN);
```

you can remember `CONFIG_PHYSICAL_ALIGN` configuration option from the previous part. This option provides the value to which kernel should be aligned and it is `0x200000` by default. After that we got aligned output address, we go through the memory and collect regions which are good for decompressed kernel image:

```
for (i = 0; i < real_mode->e820_entries; i++) {
    process_e820_entry(&real_mode->e820_map[i], minimum, size);
}
```

You can remember that we collected `e820_entries` in the second part of the Kernel booting process part 2.

First of all `process_e820_entry` function does some checks that e820 memory region is not non-RAM, that the start address of the memory region is not bigger than Maximum allowed `aslr` offset and that memory region is not less than value of kernel alignment:

```
struct mem_vector region, img;

if (entry->type != E820_RAM)
    return;

if (entry->addr >= CONFIG_RANDOMIZE_BASE_MAX_OFFSET)
    return;

if (entry->addr + entry->size < minimum)
    return;
```

After this, we store e820 memory region start address and the size in the `mem_vector` structure (we saw definition of this structure above):

```
region.start = entry->addr;
region.size = entry->size;
```

As we store these values, we align the `region.start` as we did it in the `find_random_addr` function and check that we didn't get address that bigger than original memory region:

```
region.start = ALIGN(region.start, CONFIG_PHYSICAL_ALIGN);

if (region.start > entry->addr + entry->size)
    return;
```

Next we get difference between the original address and aligned and check that if the last address in the memory region is bigger than `CONFIG_RANDOMIZE_BASE_MAX_OFFSET`, we reduce the memory region size that end of kernel image will be less than maximum `aslr` offset:

```
region.size -= region.start - entry->addr;
```

```
if (region.start + region.size > CONFIG_RANDOMIZE_BASE_MAX_OFFSET)
        region.size = CONFIG_RANDOMIZE_BASE_MAX_OFFSET - region.start;
```

In the end we go through the all unsafe memory regions and check that this region does not overlap unsafe ares with kernel command line, initrd and etc...:

```
for (img.start = region.start, img.size = image_size ;
         mem_contains(&region, &img) ;
         img.start += CONFIG_PHYSICAL_ALIGN) {
        if (mem_avoid_overlap(&img))
            continue;
        slots_append(img.start);
    }
```

If memory region does not overlap unsafe regions we call `slots_append` function with the start address of the region. `slots_append` function just collects start addresses of memory regions to the `slots` array:

```
    slots[slot_max++] = addr;
```

which defined as:

```
static unsigned long slots[CONFIG_RANDOMIZE_BASE_MAX_OFFSET /
               CONFIG_PHYSICAL_ALIGN];
static unsigned long slot_max;
```

After `process_e820_entry` will be executed, we will have array of the addressess which are safe for the decompressed kernel. Next we call `slots_fetch_random` function for getting random item from this array:

```
if (slot_max == 0)
    return 0;

return slots[get_random_long() % slot_max];
```

where `get_random_long` function checks different CPU flags as `X86_FEATURE_RDRAND` or `X86_FEATURE_TSC` and chooses method for getting random number (it can be obtain with RDRAND instruction, Time stamp counter, programmable interval timer and etc...). After that we got random address execution of the `choose_kernel_location` is finished.

Now let's back to the misc.c. After we got address for the kernel image, there need to do some checks to be sure that gotten random address is correctly aligned and address is not wrong.

After all these checks will see the familiar message:

```
Decompressing Linux...
```

and call `decompress` function which will decompress the kernel. `decompress` function depends on what decompression algorithm was choosen during kernel compilartion:

```
#ifdef CONFIG_KERNEL_GZIP
#include "../../../../lib/decompress_inflate.c"
#endif

#ifdef CONFIG_KERNEL_BZIP2
#include "../../../../lib/decompress_bunzip2.c"
#endif

#ifdef CONFIG_KERNEL_LZMA
```

```
#include "../../../../lib/decompress_unlzma.c"
#endif

#ifdef CONFIG_KERNEL_XZ
#include "../../../../lib/decompress_unxz.c"
#endif

#ifdef CONFIG_KERNEL_LZO
#include "../../../../lib/decompress_unlzo.c"
#endif

#ifdef CONFIG_KERNEL_LZ4
#include "../../../../lib/decompress_unlz4.c"
#endif
```

After kernel will be decompressed, the last function `handle_relocations` will relocate the kernel to the address that we got from `choose_kernel_location`. After that kernel relocated we return from the `decompress_kernel` to the `head_64.S`. The address of the kernel will be in the `rax` register and we jump on it:

```
jmp    *%rax
```

That's all. Now we are in the kernel!

Conclusion

This is the end of the fifth and the last part about linux kernel booting process. We will not see posts about kernel booting anymore (maybe only updates in this and previous posts), but there will be many posts about other kernel internals.

Next chapter will be about kernel initialization and we will see the first steps in the linux kernel initialization code.

If you will have any questions or suggestions write me a comment or ping me in twitter.

Please note that English is not my first language, And I am really sorry for any inconvenience. If you will find any mistakes please send me PR to linux-internals.

Links

- address space layout randomization
- initrd
- long mode
- bzip2
- RDdRand instruction
- Time Stamp Counter
- Programmable Interval Timers
- Previous part

Kernel booting process

You will see here a couple of posts which describes full cycle of the kernel initialization from the first steps after kernel decompressed to starting of the first process runned by kernel.

- Frist steps after kernel decompressed - describes first steps in the kernel.

Kernel initialization. Part 1.

First steps in the kernel code

In the previous post (`Kernel booting process. Part 5.`) - Kernel decompression we stopped at the jump on the decompressed kernel:

```
jmp    *%rax
```

and now we are in the kernel. There are many things to do before the kernel will start first `init` process. Hope we will see all of the preparations before kernel will start in this big chapter. We will start from the kernel entry point, which is in the arch/x86/kernel/head_64.S. We will see first preparations like early page tables initialization, switch to a new descriptor in kernel space and many many more, before we will see the `start_kernel` function from the init/main.c will be called.

So let's start.

First steps in the kernel

Ok, we got address of the kernel from the `decompress_kernel` function into `rax` register and just jumped there. Decompressed kernel code starts in the arch/x86/kernel/head_64.S:

```
    __HEAD
    .code64
    .globl startup_64
startup_64:
    ...
    ...
    ...
```

We can see defintion of the `startup_64` routine and it defined in the `__HEAD` section, which is just:

```
#define __HEAD        .section    ".head.text","ax"
```

We can see defintion of this section in the arch/x86/kernel/vmlinux.lds.S linker script:

```
.text : AT(ADDR(.text) - LOAD_OFFSET) {
    _text = .;
    ...
    ...
    ...
} :text = 0x9090
```

We can understand default virtual and physicall addresses from the linker script. Note that adddress of the `_text` is location counter which is defined as:

```
. = __START_KERNEL;
```

for `x86_64` . We can find definition of the `__START_KERNEL` macro in the arch/x86/include/asm/page_types.h:

```
#define __START_KERNEL    (__START_KERNEL_map + __PHYSICAL_START)
```

```
#define __PHYSICAL_START  ALIGN(CONFIG_PHYSICAL_START, CONFIG_PHYSICAL_ALIGN)
```

Here we can see that `__START_KERNEL` is the sum of the `__START_KERNEL_map` (which is `0xffffffff80000000` , see post about paging) and `__PHYSICAL_START` . Where `__PHYSICAL_START` is aligned value of the `CONFIG_PHYSICAL_START` . So if you will not use kASLR and will not change `CONFIG_PHYSICAL_START` in the configuration addresses will be following:

- Physical address - `0x1000000` ;
- Virtual address - `0xffffffff81000000` .

Now we know default physical and virtual addresses of the `startup_64` routine, but to know actual addresses we must to calculate it with the following code:

```
    leaq    _text(%rip), %rbp
    subq    $_text - __START_KERNEL_map, %rbp
```

Here we just put the `rip-relative` address to the `rbp` register and than substract `$_text - __START_KERNEL_map` from it. We know that compiled address of the `_text` is `0xffffffff81000000` and `__START_KERNEL_map` contains `0xffffffff81000000` , so `rbp` will contatin physical address of the `text` - `0x1000000` after this calcuation. We need to calcuate it because kernel can be runned not on the default address, but now we know actuall physical address.

In the next step we checks that this address is aligned with:

```
    movq    %rbp, %rax
    andl    $~PMD_PAGE_MASK, %eax
    testl    %eax, %eax
    jnz    bad_address
```

Here we just put address to the `%rax` and test first bit. `PMD_PAGE_MASK` indicates the mask for `Page middle directory` (read paging about it) and defined as:

```
#define PMD_PAGE_MASK           (~(PMD_PAGE_SIZE-1))

#define PMD_PAGE_SIZE           (_AC(1, UL) << PMD_SHIFT)
#define PMD_SHIFT       21
```

As we can easily calculate, `PMD_PAGE_SIZE` is 2 megabytes. Here we use standard formula for checking alignment and if `text` address is not aligned for 2 megabytes, we jump to `bad_address` label.

After this we check address that it is not too large:

```
    leaq    _text(%rip), %rax
    shrq    $MAX_PHYSMEM_BITS, %rax
    jnz    bad_address
```

Address most not be greater than 46-bits:

```
#define MAX_PHYSMEM_BITS       46
```

Ok, we did some early checks and now we can move on.

Fix base addresses of page tables

The first step before we started to setup identity paging, need to correct following addresses:

```
    addq    %rbp, early_level4_pgt + (L4_START_KERNEL*8)(%rip)
    addq    %rbp, level3_kernel_pgt + (510*8)(%rip)
    addq    %rbp, level3_kernel_pgt + (511*8)(%rip)
    addq    %rbp, level2_fixmap_pgt + (506*8)(%rip)
```

Here we need to correct `early_level4_pgt` and other addresses because as I wrote above kernel can be runned not at the default `0x1000000` address. `rbp` register contains actuall address so we add to the `early_level4_pgt`, `level3_kernel_pgt` and `level2_fixmap_pgt`. Let's try to understand what this labels means. First of all let's look on their definition:

```
NEXT_PAGE(early_level4_pgt)
    .fill    511,8,0
    .quad    level3_kernel_pgt - __START_KERNEL_map + _PAGE_TABLE

NEXT_PAGE(level3_kernel_pgt)
    .fill    L3_START_KERNEL,8,0
    .quad    level2_kernel_pgt - __START_KERNEL_map + _KERNPG_TABLE
    .quad    level2_fixmap_pgt - __START_KERNEL_map + _PAGE_TABLE

NEXT_PAGE(level2_kernel_pgt)
    PMDS(0, __PAGE_KERNEL_LARGE_EXEC,
        KERNEL_IMAGE_SIZE/PMD_SIZE)

NEXT_PAGE(level2_fixmap_pgt)
    .fill    506,8,0
    .quad    level1_fixmap_pgt - __START_KERNEL_map + _PAGE_TABLE
    .fill    5,8,0

NEXT_PAGE(level1_fixmap_pgt)
    .fill    512,8,0
```

Looks hard, but it is not true.

First of all let's look on the `early_level4_pgt`. It starts with the (4096 - 8) bytes of zeros, it means that we don't use first 511 `early_level4_pgt` entries. And after this we can see `level3_kernel_pgt` entry. Note that we substact `__START_KERNEL_map + _PAGE_TABLE` from it. As we know `__START_KERNEL_map` is a base virtual address of the kernel text, so if we substract `__START_KERNEL_map`, we will get physical address of the `level3_kernel_pgt`. Now let's look on `_PAGE_TABLE`, it is just page entry access rights:

```
#define _PAGE_TABLE     (_PAGE_PRESENT | _PAGE_RW | _PAGE_USER | \
                         _PAGE_ACCESSED | _PAGE_DIRTY)
```

more about it, you can read in the paging post.

`level3_kernel_pgt` - stores entries which map kernel space. At the start of it's definition, we can see that it filled with zeros `L3_START_KERNEL` times. Here `L3_START_KERNEL` is the index in the page upper directory which contains `__START_KERNEL_map` address and it equals `510`. After it we can see defintion of two `level3_kernel_pgt` entries: `level2_kernel_pgt` and `level2_fixmap_pgt`. First is simple, it is page table entry which contains pointer to the page middle directory which maps kernel space and it has:

```
#define _KERNPG_TABLE   (_PAGE_PRESENT | _PAGE_RW | _PAGE_ACCESSED | \
                         _PAGE_DIRTY)
```

access rights. The second - `level2_fixmap_pgt` is a virtual addresses which can refer to any physical addresses even under kernel space.

The next `level2_kernel_pgt` calls `PDMS` macro which creates 512 megabytes from the `__START_KERNEL_map` for kernel text (after these 512 megbytes will be modules memory space).

Now we know Let's back to our code which is in the beginnig of the section. Remember that `rbp` contains actual physical address of the `_text` section. We just add this address to the base addressess of the page tables, that they'll have correct addresses:

```
    addq    %rbp, early_level4_pgt + (L4_START_KERNEL*8)(%rip)
    addq    %rbp, level3_kernel_pgt + (510*8)(%rip)
    addq    %rbp, level3_kernel_pgt + (511*8)(%rip)
    addq    %rbp, level2_fixmap_pgt + (506*8)(%rip)
```

At the first line we add `rbp` to the `early_level4_pgt`, at the second line we add `rbp` to the `level2_kernel_pgt`, at the third line we add `rbp` to the `level2_fixmap_pgt` and add `rbp` to the `level1_fixmap_pgt`.

After all of this we will have:

```
early_level4_pgt[511] -> level3_kernel_pgt[0]
level3_kernel_pgt[510] -> level2_kernel_pgt[0]
level3_kernel_pgt[511] -> level2_fixmap_pgt[0]
level2_kernel_pgt[0]   -> 512 MB kernel mapping
level2_fixmap_pgt[506] -> level1_fixmap_pgt
```

As we corrected base addresses of the page tables, we can start to build it.

Identy mapping setup

Now we can see set up the identity mapping early page pables. Identity Mapped Paging is a virtual addresses which are mapped to physical addresses that have the same value, `1 : 1`. Let's look on it in details. First of all we get the `rip-relative` address of the `_text` and `_early_level4_pgt` and put they into `rdi` and `rbx` registers:

```
    leaq    _text(%rip), %rdi
    leaq    early_level4_pgt(%rip), %rbx
```

After this we store physical address of the `_text` in the `rax` and get the index of the page global directory entry which stores `_text` address, by shifting `_text` address on the `PGDIR_SHIFT`:

```
    movq    %rdi, %rax
    shrq    $PGDIR_SHIFT, %rax

    leaq    (4096 + _KERNPG_TABLE)(%rbx), %rdx
    movq    %rdx, 0(%rbx,%rax,8)
    movq    %rdx, 8(%rbx,%rax,8)
```

where `PGDIR_SHIFT` is `39`. `PGDIR_SHFT` indicates the mask for page global directory bits in a virtual address. There are macro for all types of page directories:

```
#define PGDIR_SHIFT     39
#define PUD_SHIFT       30
#define PMD_SHIFT       21
```

After this we put the address of the first `level3_kernel_pgt` to the `rdx` with the `_KERNPG_TABLE` access rights (see above) and fill the `early_level4_pgt` with the 2 `level3_kernel_pgt` entries.

After this we add `4096` (size of the `early_level4_pgt`) to the `rdx` (it now contains the address of the first entry of the `level3_kernel_pgt`) and put `rdi` (it now contains physical address of the `_text`) to the `rax`. And after this we write addresses of the two page upper directory entries to the `level3_kernel_pgt`:

```
    addq    $4096, %rdx
    movq    %rdi, %rax
    shrq    $PUD_SHIFT, %rax
    andl    $(PTRS_PER_PUD-1), %eax
    movq    %rdx, 4096(%rbx,%rax,8)
    incl    %eax
    andl    $(PTRS_PER_PUD-1), %eax
    movq    %rdx, 4096(%rbx,%rax,8)
```

In the next step we write addresses of the page middle directory entries to the `level2_kernel_pgt` and the last step is correcting of the kernel text+data virtual addresses:

```
    leaq    level2_kernel_pgt(%rip), %rdi
    leaq    4096(%rdi), %r8
1:    testq    $1, 0(%rdi)
    jz    2f
    addq    %rbp, 0(%rdi)
2:    addq    $8, %rdi
    cmp    %r8, %rdi
    jne    1b
```

Here we put the address of the `level2_kernel_pgt` to the `rdi` and address of the page table entry to the `r8` register. Next we check the present bit in the `level2_kernel_pgt` and if it is zero we're moving to the next page by adding 8 bytes to `rdi` which contatins address of the `level2_kernel_pgt`. After this we compare it with `r8` (contains address of the page table entry) and go back to label `1` or move forward.

In the next step we correct `phys_base` physical address with `rbp` (contains physical address of the `_text`), put physical address of the `early_level4_pgt` and jump to label `1`:

```
    addq    %rbp, phys_base(%rip)
    movq    $(early_level4_pgt - __START_KERNEL_map), %rax
    jmp 1f
```

where `phys_base` mathes the first entry of the `level2_kernel_pgt` which is 512 MB kernel mapping.

Last preparations

After that we jumped to the label `1` we enable `PAE`, `PGE` (Paging Global Extension) and put the physical address of the `phys_base` (see above) to the `rax` register and fill `cr3` register with it:

```
1:
    movl    $(X86_CR4_PAE | X86_CR4_PGE), %ecx
    movq    %rcx, %cr4

    addq    phys_base(%rip), %rax
    movq    %rax, %cr3
```

In the next step we check that CPU support NX bit with:

```
    movl    $0x80000001, %eax
    cpuid
    movl    %edx,%edi
```

We put `0x80000001` value to the `eax` and execute `cpuid` instruction for getting extended processor info and feature bits. The result will be in the `edx` register which we put to the `edi`.

Now we put `0xc0000080` or `MSR_EFER` to the `ecx` and call `rdmsr` instruction for the reading model specific register.

```
    movl    $MSR_EFER, %ecx
    rdmsr
```

The result will be in the `edx:eax`. General view of the `EFER` is following:

```
63                                                                              32
 -------------------------------------------------------------------------------
|                                                                               |
|                                Reserved MBZ                                   |
|                                                                               |
 -------------------------------------------------------------------------------
31                            16  15      14      13   12  11   10  9  8 7  1   0
 -------------------------------------------------------------------------------
|                              | T |       |       |    |   |   |   |   |   |   |
| Reserved MBZ                 | C | FFXSR | LMSLE |SVME|NXE|LMA|MBZ|LME|RAZ|SCE|
|                              | E |       |       |    |   |   |   |   |   |   |
 -------------------------------------------------------------------------------
```

We will not see all fields in details here, but we will learn about this and other `MSRs` in the special part about. As we read `EFER` to the `edx:eax`, we checks `_EFER_SCE` or zero bit which is `System Call Extensions` with `btsl` instruction and set it to one. By the setting `SCE` bit we enable `SYSCALL` and `SYSRET` instructions. In the next step we check 20th bit in the `edi`, remember that this register stores result of the `cpuid` (see above). If `20` bit is set (`NX` bit) we just write `EFER_SCE` to the model specific register.

```
    btsl    $_EFER_SCE, %eax
    btl        $20,%edi
    jnc     1f
    btsl    $_EFER_NX, %eax
    btsq    $_PAGE_BIT_NX,early_pmd_flags(%rip)
1:    wrmsr
```

If `NX` bit is supported we enable `_EFER_NX` and write it too, with the `wrmsr` instruction.

In the next step we need to update Global Descriptor table with `lgdt` instruction:

```
lgdt    early_gdt_descr(%rip)
```

where Global Descriptor table defined as:

```
early_gdt_descr:
    .word    GDT_ENTRIES*8-1
early_gdt_descr_base:
    .quad    INIT_PER_CPU_VAR(gdt_page)
```

We need to reload Global Descriptor Table because now kernel works in the userspace addresses, but soon kernel will work in it's own space. Now let's look on `early_gdt_descr` defintion. Global Descriptor Table contains 32 entries:

```
#define GDT_ENTRIES 32
```

for kernel code, data, thread local storage segments and etc... it's simple. Now let's look on the `early_gdt_descr_base`. First of `gdt_page` defined as:

```
struct gdt_page {
    struct desc_struct gdt[GDT_ENTRIES];
} __attribute__((aligned(PAGE_SIZE)));
```

in the arch/x86/include/asm/desc.h. It contains one field `gdt` which is array of the `desc_struct` structures which defined as:

```
struct desc_struct {
         union {
                 struct {
                         unsigned int a;
                         unsigned int b;
                 };
                 struct {
                         u16 limit0;
                         u16 base0;
                         unsigned base1: 8, type: 4, s: 1, dpl: 2, p: 1;
                         unsigned limit: 4, avl: 1, l: 1, d: 1, g: 1, base2: 8;
                 };
         };
 } __attribute__((packed));
```

and presents familiar to us GDT descriptor. Also we can note that `gdt_page` structure aligned to `PAGE_SIZE` which is 4096 bytes. It means that `gdt` will occupy one page. Now let's try to understand what is it `INIT_PER_CPU_VAR`. `INIT_PER_CPU_VAR` is a macro which defined in the arch/x86/include/asm/percpu.h and just concats `init_per_cpu__` with the given parameter:

```
#define INIT_PER_CPU_VAR(var) init_per_cpu__##var
```

After this we have `init_per_cpu__gdt_page`. We can see in the linker script:

```
#define INIT_PER_CPU(x) init_per_cpu__##x = x + __per_cpu_load
INIT_PER_CPU(gdt_page);
```

As we got `init_per_cpu__gdt_page` in `INIT_PER_CPU_VAR` and `INIT_PER_CPU` macro from linker script will be expanded we will get offset from the `__per_cpu_load`. After this calculations, we will have correct base address of the new GDT.

Generally per-CPU variables is a 2.6 kernel feature. You can understand what is it from it's name. When we create `per-CPU` variable, each CPU will have will have it's own copy of this variable. Here we creating `gdt_page` per-CPU variable. There are many advantages for variables of this type, like there are no locks, because each CPU works with it's own copy of variable and etc... So every core on multiprocessor will have it's own `GDT` table and every entry in the table will represent a memory segment which can be accessed from the thread which runned on the core. You can read in details about `per-CPU` variables in the Theory/per-cpu post.

As we loaded new Global Descriptor Table, we reload segments as we did it everytime:

```
    xorl %eax,%eax
    movl %eax,%ds
    movl %eax,%ss
    movl %eax,%es
    movl %eax,%fs
    movl %eax,%gs
```

After all of these steps we set up `gs` register that it post to the `irqstack` (we will see information about it in the next parts):

```
    movl    $MSR_GS_BASE,%ecx
    movl    initial_gs(%rip),%eax
    movl    initial_gs+4(%rip),%edx
    wrmsr
```

where `MSR_GS_BASE` is:

```
#define MSR_GS_BASE             0xc0000101
```

We need to put `MSR_GS_BASE` to the `ecx` register and load data from the `eax` and `edx` (which are point to the `initial_gs`) with `wrmsr` instruction. We don't use `cs`, `fs`, `ds` and `ss` segment registers for addressation in the 64-bit mode, but `fs` and `gs` registers can be used. `fs` and `gs` have a hidden part (as we saw it in the real mode for `cs`) and this part contains descriptor which mapped to Model specific registers. So we can see above `0xc0000101` is a `gs.base` MSR address.

In the next step we put the address of the real mode bootparam structure to the `rdi` (remember `rsi` holds pointer to this structure from the start) and jump to the C code with:

```
    movq    initial_code(%rip),%rax
    pushq    $0
    pushq    $__KERNEL_CS
    pushq    %rax
    lretq
```

Here we put the address of the `initial_code` to the `rax` and push fake address, `__KERNEL_CS` and the address of the `initial_code` to the stack. After this we can see `lretq` instruction which means that after it return address will be extracted from stack (now there is address of the `initial_code`) and jump there. `initial_code` defined in the same source code file and looks:

```
    __REFDATA
    .balign    8
    GLOBAL(initial_code)
    .quad    x86_64_start_kernel
    ...
    ...
    ...
```

As we can see `initial_code` contains addresss of the `x86_64_start_kernel`, which defined in the arch/x86/kerne/head64.c and looks like this:

```
asmlinkage __visible void __init x86_64_start_kernel(char * real_mode_data) {
    ...
    ...
    ...
}
```

It has one argument is a `real_mode_data` (remember that we passed address of the real mode data to the `rdi` register previously).

This is first C code in the kernel!

Next to start_kernel

We need to see last preparations before we can see "kernel entry point" - start_kernel function from the init/main.c.

First of all we can see some checks in the `x86_64_start_kernel` function:

```
BUILD_BUG_ON(MODULES_VADDR < __START_KERNEL_map);
BUILD_BUG_ON(MODULES_VADDR - __START_KERNEL_map < KERNEL_IMAGE_SIZE);
BUILD_BUG_ON(MODULES_LEN + KERNEL_IMAGE_SIZE > 2*PUD_SIZE);
BUILD_BUG_ON((__START_KERNEL_map & ~PMD_MASK) != 0);
BUILD_BUG_ON((MODULES_VADDR & ~PMD_MASK) != 0);
BUILD_BUG_ON(!(MODULES_VADDR > __START_KERNEL));
BUILD_BUG_ON(!(((MODULES_END - 1) & PGDIR_MASK) == (__START_KERNEL & PGDIR_MASK)));
BUILD_BUG_ON(__fix_to_virt(__end_of_fixed_addresses) <= MODULES_END);
```

There are checks for different things like virtual addresses of modules space is not fewer than base address of the kernel text - `__STAT_KERNEL_map`, that kernel text with modules is not less than image of the kernel and etc... `BUILD_BUG_ON` is a macro which looks as:

```
#define BUILD_BUG_ON(condition) ((void)sizeof(char[1 - 2*!!(condition)]))
```

Let's try to understand this trick works. Let's take for example first condition: `MODULES_VADDR < __START_KERNEL_map`. `!!conditions` is the same that `condition != 0`. So it means if `MODULES_VADDR < __START_KERNEL_map` is true, we will get `1` in the `!!(condition)` or zero if not. After `2*!!(condition)` we will get or `2` or `0`. In the end of calculations we can get two different behaviors:

- We will have compilation error, because try to get size of the char array with negative index (as can be in our case, because `MODULES_VADDR` can't be less than `__START_KERNEL_map` will be in our case);
- No compilation errors.

That's all. So interesting C trick for getting compile error which depends on some contants.

In the next step we can see call of the `cr4_init_shadow` function which stores shadow copy of the `cr4` per cpu. Context switches can change bits in the `cr4` so we need to store `cr4` for each CPU. And after this we can see call of the `reset_early_page_tables` function where we resets all page global directory entries and write new pointer to the PGT in `cr3`:

```
for (i = 0; i < PTRS_PER_PGD-1; i++)
    early_level4_pgt[i].pgd = 0;

next_early_pgt = 0;

write_cr3(__pa_nodebug(early_level4_pgt));
```

soon we will build new page tables. Here we can see that we go through all Page Global Directory Entries (`PTRS_PER_PGD` is `512`) in the loop and make it zero. After this we set `next_early_pgt` to zero (we will see details about it in the next post) and write physical address of the `early_level4_pgt` to the `cr3`. `__pa_nodebug` is a macro which will be expanded to:

```
((unsigned long)(x) - __START_KERNEL_map + phys_base)
```

After this we clear `_bss` from the `__bss_stop` to `__bss_start` and the next step will be setup of the early `IDT` handlers, but it's big theme so we will see it in the next part.

Conclusion

This is the end of the first part about linux kernel initialization.

If you have questions or suggestions, feel free to ping me in twitter 0xAX, drop me email or just create issue.

In the next part we will see initialization of the early interruption handlers, kernel space memory mapping and many many more.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-internals.

Links

- Model Specific Register
- Paging

* Previous part - Kernel decompression
* NX
* ASLR

Paging

Introduction

In the fifth part of the series `Linux kernel booting process` we finished to learn what and how kernel does on the earliest stage. In the next step kernel will initialize different things like `initrd` mounting, lockdep initialization, and many many different things, before we can see how the kernel will run the first init process.

Yeah, there will be many different things, but many many and once again many work with **memory**.

In my view, memory management is one of the most complex part of the linux kernel and in system programming generally. So before we will proceed with the kernel initialization stuff, we will get acquainted with the paging.

`Paging` is a process of translaion a linear memory address to a physical address. If you have read previous parts, you can remember that we saw segmentation in the real mode when physical address calculated by shifting a segment register on four and adding offset. Or also we saw segmentation in the protected mode, where we used the tables of descriptors and base addresses from descriptors with offsets to calculate physical addresses. Now we are in 64-bit mode and that we will see paging.

As intel manual says:

```
Paging provides a mech-anism for implementing a conventional demand-paged, virtual-memory system where sections of a pr
```

So... I will try to explain how paging works in theory in this post. Of course it will be closely related with the linux kernel for `x86_64`, but we will not go into deep details (at least in this post).

Enabling paging

There are three paging modes:

- 32-bit paging;
- PAE paging;
- IA-32e paging.

We will see explanation only last mode here. To enable `IA-32e paging` paging mode need to do following things:

- set `CR0.PG` bit;
- set `CR4.PAE` bit;
- set `IA32_EFER.LME` bit.

We already saw setting of this bits in the arch/x86/boot/compressed/head_64.S:

```
movl    $(X86_CR0_PG | X86_CR0_PE), %eax
movl    %eax, %cr0
```

and

```
movl    $MSR_EFER, %ecx
rdmsr
btsl    $_EFER_LME, %eax
wrmsr
```

Paging structures

Paging divides the linear address space into fixed-size pages. Pages can be mapped into the physical address space or even external storage. This fixed size is `4096` bytes for the `x86_64` linux kernel. For a linear address translation to a physical address used special structures. Every structure is `4096` bytes size and contains `512` entries (this only for `PAE` and `IA32_EFER.LME` modes). Paging structures are hierarchial and linux kernel for `x86_64` uses 4 level paging. CPU uses a part of the linear address to identify entry of the another paging structure which is at the lower level or physical memory region (`page frame`) or physical address in this region (`page offset`). The address of the top level paging structure located in the `cr3` register. We already saw this in the arch/x86/boot/compressed/head_64.S:

```
leal    pgtable(%ebx), %eax
movl    %eax, %cr3
```

We built page table structures and put the address of the top-level structure to the `cr3` register. Here `cr3` is used to store the address of the top-level `PML4` structure or `Page Global Directory` as it calls in linux kernel. `cr3` is 64-bit register and has the following structure:

```
63                  52 51                                                        32
 --------------------------------------------------------------------------------
|                     |                                                          |
|    Reserved MBZ     |            Address of the top level structure            |
|                     |                                                          |
 --------------------------------------------------------------------------------
31                                  12 11            5     4     3 2             0
 --------------------------------------------------------------------------------
|                                     |               |  P  |  P  |              |
|  Address of the top level structure |   Reserved    |  C  |  W  |    Reserved  |
|                                     |               |  D  |  T  |              |
 --------------------------------------------------------------------------------
```

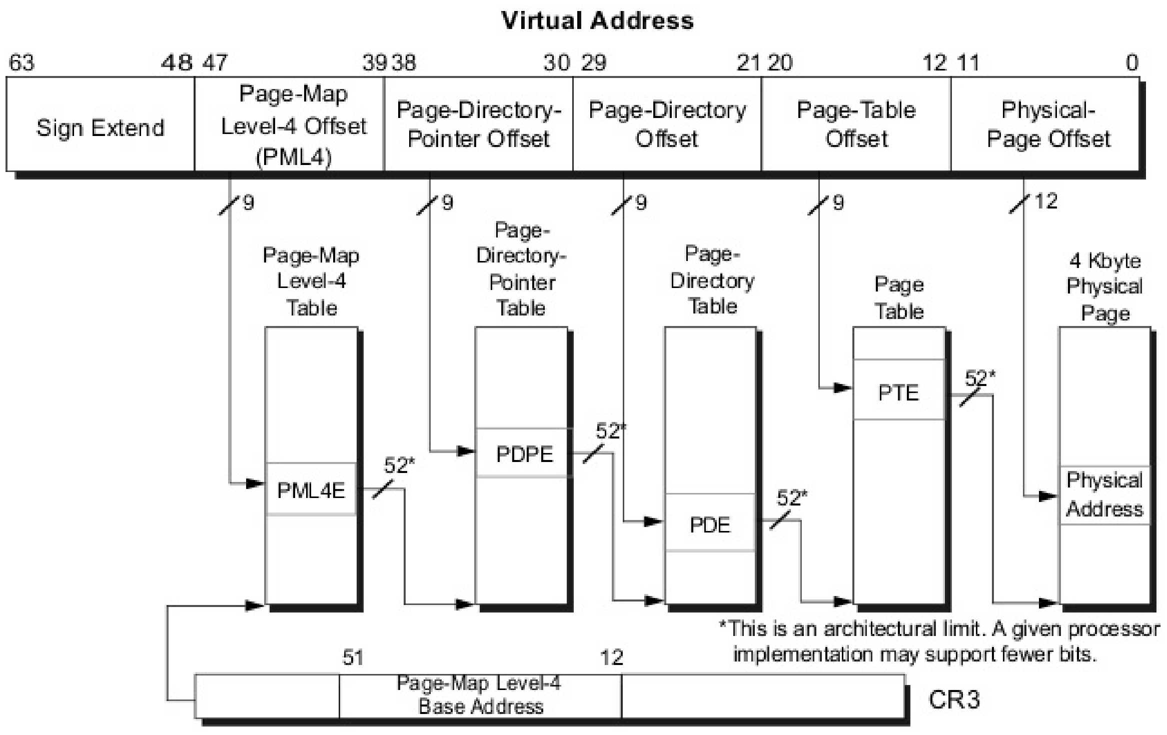
These fiealds have the following meanings:

- Bits 2:0 - ignored;
- Bits 51:12 - stores the address of the top level paging structure;
- Bit 3 and 4 - PWT or Page-Level Writethrough and PCD or Page-level cache disable indicate. These bits control the way the page or Page Table is handled by the hardware cache;
- Reserved - reserved must be 0;
- Bits 63:52 - reserved must be 0.

The linear address tranlation address is following:

- Given linear address arrives to the MMU instead of memory bus.
- 64-bit linear address splits on some parts. Only low 48 bits are significant, it means that `2^48` or 256 TBytes of linear-address space may be accessed at any given time.
- `cr3` register stores the address of the 4 top-level paging structure.
- `47:39` bits of the given linear address stores an index into the paging structure level-4, `38:30` bits stores index into the paging structure level-3, `29:21` bits stores an index into the paging structure level-2, `20:12` bits stores an index into the paging structure level-1 and `11:0` bits provide the byte offset into the physical page.

schematically, we can imagine it like this:



Every access to a linear address is either a supervisor-mode access or a user-mode access. This access determined by the `CPL` (current privilege level). If `CPL < 3` it is a supervisor mode access level and user mode access level in other ways. For example top level page table entry contains access bits and has the following structure:

```
63  62                  52 51                                                    32
 --------------------------------------------------------------------------------
| N |                     |                                                     |
|   |     Available       |     Address of the paging structure on lower level  |
| X |                     |                                                     |
 --------------------------------------------------------------------------------
31                                              12 11  9 8 7 6 5   4   3 2 1     0
 --------------------------------------------------------------------------------
|                                                |     | M |I| | P | P |U|W|    |
| Address of the paging structure on lower level | AVL | B |G|A| C | W | | |  P |
|                                                |     | Z |N| | D | T |S|R|    |
 --------------------------------------------------------------------------------
```

Where:

- 63 bit - N/X bit (No Execute Bit) - presents ability to execute the code from physical pages mapped by the table entry;
- 62:52 bits - ignored by CPU, used by system software;
- 51:12 bits - stores physical address of the lower level paging structure;
- 12:9 bits - ignored by CPU;
- MBZ - must be zero bits;
- Ignored bits;
- A - accessed bit indicates was physical page or page structure accessed;
- PWT and PCD used for cache;
- U/S - user/superviso bit contols user access to the all physical pages mapped by this table entry;
- R/W - read/write bit controls read/write access to the all physical pages mapped by this table entry;
- P - present bit. Current bit indicates was page table or physical page loaded into primary memory or not.

Ok, now we know about paging structures and it's entries. Let's see some details about 4-level paging in linux kernel.

Paging structures in linux kernel

As i wrote about linux kernel for `x86_64` uses 4-level page tables. Their names are:

- Page Global Directory
- Page Upper Directory
- Page Middle Directory
- Page Table Entry

After that you compiled and installed linux kernel, you can note `System.map` file which stores address of the functions that are used by the kernel. Note that addresses are virtual. For example:

```
$ grep "start_kernel" System.map
ffffffff81efe497 T x86_64_start_kernel
ffffffff81efeaa2 T start_kernel
```

We can see `0xffffffff81efe497` here. I'm not sure that you have so big RAM. But anyway `start_kernel` and `x86_64_start_kernel` will be executed. The address space in `x86_64` is `2^64` size, but it's too large, that's why used smaller address space, only 48-bits wide. So we have situation when physical address limited with 48 bits, but addressing still performed with 64 bit pointers. How to solve this problem? Ok, look on the diagram:

```
0xffffffffffffffff  +-----------+
                    |           |
                    |           | Kernelspace
                    |           |
 0xffff800000000000 +-----------+
                    |           |
                    |           |
                    |   hole    |
                    |           |
                    |           |
0x00007fffffffffff  +-----------+
                    |           |
                    |           |  Userspace
                    |           |
0x0000000000000000  +-----------+
```

This solution is `sign extension`. Here we can see that low 48 bits of a virtual address can be used for addressing. Bits `63:48` can be or 0 or 1. Note that all virtual address space is spliten on 2 parts:

- Kernel space
- Userspace

Userspace occupies the lower part of the virtual address space, from `0x000000000000000` to `0x00007fffffffffff` and kernel space occupies the highest part from the `0xffff8000000000` to `0xffffffffffffffff`. Note that bits `63:48` is 0 for userspace and 1 for kernel space. All addresses which are in kernel space and in userspace or in another words which higher `63:48` bits zero or one calls `canonical` addresses. There is `non-canonical` area between these memory regions. Together this two memory regions (kernel space and user space) are exactly `2^48` bits. We can find virtual memory map with 4 level page tables in the Documentation/x86/x86_64/mm.txt:

```
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
hole caused by [48:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffc0000000000 (=44 bits) kasan shadow memory (16TB)
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffff80000000 - ffffffffa0000000 (=512 MB)  kernel text mapping, from phys 0
ffffffffa0000000 - ffffffffff5fffff (=1525 MB) module mapping space
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
```

```
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
```

We can see here memory map for user space, kernel space and non-canonical area between. User space memory map is simple. Let's take a closer look on the kernel space. We can see that it starts from the guard hole which reserved for hypervisor. We can find definiton of this guad hole in the arch/x86/include/asm/page_64_types.h:

```
#define __PAGE_OFFSET _AC(0xffff880000000000, UL)
```

Previously this guard hole and `__PAGE_OFFSET` was from `0xffff800000000000` to `0xffff80ffffffffff` for preventing of access to non-canonical area, but later was added 3 bits for hypervisor.

Next is the lowest usable address in kernel space - `ffff880000000000`. This virtual memory region is for direct mapping of the all physical memory. After the memory space which mapped all physical address - guard hole, it needs to be between direct mapping of the all physical memory and vmalloc area. After the virtual memory map for the first terabyte and unused hole after it, we can see `kasan` shadow memory. It was added by the commit and provides kernel address sanitizer. After next unused hole we can se `esp` fixup stacks (we will talk about it in the other parts) and the start of the kernel text mapping from the physical address - `0`. We can find definition of this address in the same file as the `__PAGE_OFFSET`:

```
#define __START_KERNEL_map      _AC(0xffffffff80000000, UL)
```

Usually kernel's `.text` start here with the `CONFIG_PHYSICAL_START` offset. We saw it in the post about ELF64:

```
readelf -s vmlinux | grep ffffffff81000000
     1: ffffffff81000000     0 SECTION LOCAL  DEFAULT    1
 65099: ffffffff81000000     0 NOTYPE  GLOBAL DEFAULT    1 _text
 90766: ffffffff81000000     0 NOTYPE  GLOBAL DEFAULT    1 startup_64
```

Here i checked `vmlinux` with the `CONFIG_PHYSICAL_START` is `0x1000000`. So we have the start point of the kernel `.text` - `0xffffffff80000000` and offset - `0x1000000`, the resulted virtual address will be `0xffffffff80000000 + 1000000 = 0xffffffff81000000`.

After the kernel `.text` region, we can see virtual memory region for kernel modules, `vsyscalls` and 2 megabytes unused hole.

We know how looks kernel's virtual memory map and now we can see how a virtual address translates into physical. Let's take for example following address:

```
0xffffffff81000000
```

In binary it will be:

```
0b1111111111111111111111111111111110000001000000000000000000000000
```

The given virtual address will be splitten on some parts as i wrote above:

- `63:48` - bits not used;
- `47:39` - bits of the given linear address stores an index into the paging structure level-4;
- `38:30` - bits stores index into the paging structure level-3;
- `29:21` - bits stores an index into the paging structure level-2;
- `20:12` - bits stores an index into the paging structure level-1;
- `11:0` - bits provide the byte offset into the physical page.

That is all. Now you know a little about `paging` theory and we can go ahed in the kernel source code and see first initialization steps.

Conclusion

It's the end of this short part about paging theory. Of cousre this post doesn't cover all details about paging, but soon we will see it on practice how linux kernel builds paging structures and work with it.

Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to linux-internals.

Links

- Paging on Wikipedia
- Intel 64 and IA-32 architectures software developer's manual volume 3A
- MMU
- ELF64
- Documentation/x86/x86_64/mm.txt
- Last part - Kernel booting process

Executable and Linkable Format

ELF (Executable and Linkable Format) is a standard file format for executable files and shared libraries. Linux as many UNIX-like operating systems uses this format. Let's look on structure of the ELF-64 Object File Format and some defintions in the linux kernel source code related with it.

An ELF object file consists of the following parts:

- ELF header - describes the main characteristics of the object file: type, CPU architecture, the virtual address of the entry point, the size and offset the remaining parts and etc...;
- Program header table - listing the available segments and their attributes. Program header table need loaders for placing sections of the file as virtual memory segments;
- Section header table - contains description of the sections.

Now let's look closer on these components.

ELF header

It located in the beginning of the object file. It's main point is to locate all other parts of the object file. File header contains following fields:

- ELF identification - array of bytes which helps to identify the file as an ELF object file and also provides information about general object file characteristic;
- Object file type - identifies the object file type. This field can describe that ELF file is relocatable object file, executable file and etc...;
- Target architecture;
- Version of the object file format;
- Virtual address of the program entry point;
- File offset of the program header table;
- File offset of the section header table;
- Size of an ELF header;
- Size of a program header table entry;
- and other fields...

You can find `elf64_hdr` structure which presents ELF64 header in the linux kernel source code:

```
typedef struct elf64_hdr {
    unsigned char    e_ident[EI_NIDENT];
    Elf64_Half e_type;
    Elf64_Half e_machine;
    Elf64_Word e_version;
    Elf64_Addr e_entry;
    Elf64_Off e_phoff;
    Elf64_Off e_shoff;
    Elf64_Word e_flags;
    Elf64_Half e_ehsize;
    Elf64_Half e_phentsize;
    Elf64_Half e_phnum;
    Elf64_Half e_shentsize;
    Elf64_Half e_shnum;
    Elf64_Half e_shstrndx;
} Elf64_Ehdr;
```

This structure defined in the elf.h

Sections

All data stores in a sections in an Elf object file. Sections identified by index in the section header table. Section header contains following fields:

- Section name;
- Section type;
- Section attributes;
- Virtual address in memory;
- Offset in file;
- Size of section;
- Link to other section;
- Miscellaneous information;
- Address alignment boundary;
- Size of entries, if section has table;

And presented with the following `elf64_shdr` structure in the linux kernel:

```
typedef struct elf64_shdr {
    Elf64_Word sh_name;
    Elf64_Word sh_type;
    Elf64_Xword sh_flags;
    Elf64_Addr sh_addr;
    Elf64_Off sh_offset;
    Elf64_Xword sh_size;
    Elf64_Word sh_link;
    Elf64_Word sh_info;
    Elf64_Xword sh_addralign;
    Elf64_Xword sh_entsize;
} Elf64_Shdr;
```

Program header table

All sections are grouped into segments in an executable or shared object file. Program header is an array of structures which describe every segment. It looks like:

```
typedef struct elf64_phdr {
    Elf64_Word p_type;
    Elf64_Word p_flags;
    Elf64_Off p_offset;
    Elf64_Addr p_vaddr;
    Elf64_Addr p_paddr;
    Elf64_Xword p_filesz;
    Elf64_Xword p_memsz;
    Elf64_Xword p_align;
} Elf64_Phdr;
```

in the linux kernel source code.

`elf64_phdr` defined in the same elf.h.

And ELF object file also contains other fields/structures which you can find in the Documentation. Better let's look on the `vmlinux`.

vmlinux

`vmlinux` is relocatable ELF object file too. So we can look on it with the `readelf` util. First of all let's look on a header:

```
$ readelf -h  vmlinux
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1000000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          381608416 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         5
  Size of section headers:           64 (bytes)
  Number of section headers:         73
  Section header string table index: 70
```

Here we can see that `vmlinux` is 64-bit executable file.

We can read from the Documentation/x86/x86_64/mm.txt:

```
ffffffff80000000 - ffffffffa0000000 (=512 MB)  kernel text mapping, from phys 0
```

So we can find it in the `vmlinux` with:

```
readelf -s vmlinux | grep ffffffff81000000
     1: ffffffff81000000     0 SECTION LOCAL  DEFAULT    1
 65099: ffffffff81000000     0 NOTYPE  GLOBAL DEFAULT    1 _text
 90766: ffffffff81000000     0 NOTYPE  GLOBAL DEFAULT    1 startup_64
```

Note that here is address of the `startup_64` routine is not `ffffffff80000000`, but `ffffffff81000000` and now i'll explain why.

We can see following definition in the arch/x86/kernel/vmlinux.lds.S:

```
    . = __START_KERNEL;
    ...
    ...
    ..
    /* Text and read-only data */
    .text :  AT(ADDR(.text) - LOAD_OFFSET) {
        _text = .;
        ...
        ...
        ...
    }
```

Where `__START_KERNEL` is:

```
#define __START_KERNEL        (__START_KERNEL_map + __PHYSICAL_START)
```

`__START_KERNEL_map` is the value from documentation - `ffffffff80000000` and `__PHYSICAL_START` is `0x1000000`. That's why address of the `startup_64` is `ffffffff81000000`.

And the last we can get program headers from `vmlinux` with the following command:

```
readelf -l vmlinux

Elf file type is EXEC (Executable file)
Entry point 0x1000000
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000200000 0xffffffff81000000 0x0000000001000000
```

```
                 0x0000000000cfd000 0x0000000000cfd000  R E    200000
  LOAD           0x0000000001000000 0xffffffff81e00000 0x0000000001e00000
                 0x0000000000100000 0x0000000000100000  RW     200000
  LOAD           0x0000000001200000 0x0000000000000000 0x0000000001f00000
                 0x0000000000014d98 0x0000000000014d98  RW     200000
  LOAD           0x0000000001315000 0xffffffff81f15000 0x0000000001f15000
                 0x000000000011d000 0x0000000000279000  RWE    200000
  NOTE           0x0000000000b17284 0xffffffff81917284 0x0000000001917284
                 0x0000000000000024 0x0000000000000024         4

 Section to Segment mapping:
  Segment Sections...
   00     .text .notes __ex_table .rodata __bug_table .pci_fixup .builtin_fw
          .tracedata __ksymtab __ksymtab_gpl __kcrctab __kcrctab_gpl
          __ksymtab_strings __param __modver
   01     .data .vvar
   02     .data..percpu
   03     .init.text .init.data .x86_cpu_dev.init .altinstructions
          .altinstr_replacement .iommu_table .apicdrivers .exit.text
          .smp_locks .data_nosave .bss .brk
```

Here we can see five segments with sections list. All of these sections you can find in the generated linker script at - `arch/x86/kernel/vmlinux.lds`.

That's all. Of course it's not a full description of ELF object format, but if you are interesting in it, you can find documentation - here

Per-CPU variables

In Progress

Per-CPU variables are one of kernel features. You can understand what this feature mean by it's name. We can create variable and each processor core will have own copy of this variable. We take a closer look on this feature and try to understand how it implemented and how it work in this part.

Kernel provides API for creating per-cpu variables - `DEFINE_PER_CPU` macro:

```
#define DEFINE_PER_CPU(type, name) \
        DEFINE_PER_CPU_SECTION(type, name, "")
```

This macro defined in the include/linux/percpu-defs.h as many other macros for work with per-cpu variables. Now we will see how this feature implemented.

Take a look on `DECLARE_PER_CPU` definition. We see that it takes 2 paramters: `type` and `name`. So we can use it for creation per-cpu variable, for example like this:

```
DEFINE_PER_CPU(int, per_cpu_n)
```

We pass type of our variable and name. `DEFI_PER_CPU` calls `DEFINE_PER_CPU_SECTION` macro and passes the same two paramaters and empty string to it. Let's look on the defintion of the `DEFINE_PER_CPU_SECTION`:

```
#define DEFINE_PER_CPU_SECTION(type, name, sec)    \
         __PCPU_ATTRS(sec) PER_CPU_DEF_ATTRIBUTES  \
         __typeof__(type) name
```

```
#define __PCPU_ATTRS(sec)                                                \
         __percpu __attribute__((section(PER_CPU_BASE_SECTION sec)))     \
         PER_CPU_ATTRIBUTES
```

where section is:

```
#define PER_CPU_BASE_SECTION ".data..percpu"
```

After all macros will be exapanded we will get global per-cpu variable:

```
__attribute__((section(".data..percpu"))) int per_cpu_n
```

It means that we will have `per_cpu_n` variable in the `.data..percpu` section. We can find this section in the `vmlinux`:

```
.data..percpu 00013a58  0000000000000000  0000000001a5c000  00e00000  2**12
              CONTENTS, ALLOC, LOAD, DATA
```

Ok, now we know that when we use `DEFINE_PER_CPU` macro, per-cpu variable in the `.data..percpu` section will be created. When kernel initilizes it calls `setup_per_cpu_areas` function which loads `.data..percpu` section multiply times, one section per CPU. After kernel finished initialization process we have loaded N `.data..percpu` sections, where N is a number of CPU, and section used by bootstrap processor will contain uninitializaed variable created with `DEFINE_PER_CPU` macro.

Kernel provides API for per-cpu variables manipulating:

- get_cpu_var(var)
- put_cpu_var(var)

Let's look on `get_cpu_var` implementation:

```
#define get_cpu_var(var)     \
(*({                         \
         preempt_disable();  \
         this_cpu_ptr(&var); \
}))
```

Linux kernel is preemptible and accessing a per-cpu variable requires to know which processor kernel running on. So, current code must not be preempted and moved to the another CPU while accessing a per-cpu variable. That's why first of all we can see call of the `preempt_disable` function. After this we can see call of the `this_cpu_ptr` macro, which looks as:

```
#define this_cpu_ptr(ptr) raw_cpu_ptr(ptr)
```

and

```
#define raw_cpu_ptr(ptr)        per_cpu_ptr(ptr, 0)
```

where `per_cpu_ptr` returns a pointer to the per-cpu variable for the given cpu (second paramter). After that we got per-cpu variables and made any manipulations on it, we must call `put_cpu_var` macro which enables preemption with call of `preempt_enable` function. So the typical usage of a per-cpu variable is following:

```
get_cpu_var(var);
...
//Do something with the 'var'
...
put_cpu_var(var);
```

Let's look on `per_cpu_ptr` macro:

```
#define per_cpu_ptr(ptr, cpu)                             \
({                                                        \
        __verify_pcpu_ptr(ptr);                           \
         SHIFT_PERCPU_PTR((ptr), per_cpu_offset((cpu)));  \
})
```

As i wrote above, this macro returns per-cpu variable for the given cpu. First of all it calls `__verify_pcpu_ptr`:

```
#define __verify_pcpu_ptr(ptr)
do {
    const void __percpu *__vpp_verify = (typeof((ptr) + 0))NULL;
    (void)__vpp_verify;
} while (0)
```

which makes given `ptr` type of `const void __percpu *`,

After this we can see the call of the `SHIFT_PERCPU_PTR` macro with two paramters. At first paramter we pass our ptr and sencond we pass cpu number to the `per_cpu_offset` macro which:

```
#define per_cpu_offset(x) (__per_cpu_offset[x])
```

expands to getting `x` element from the `__per_cpu_offset` array:

```
extern unsigned long __per_cpu_offset[NR_CPUS];
```

where `NR_CPUS` is the number of CPUs. `__per_cpu_offset` array filled with the distances between cpu-variables copies. For example all per-cpu data is `x` bytes size, so if we access `__per_cpu_offset[Y]`, so `x*Y` will be accessed. Let's look on the `SHIFT_PERCPU_PTR` implementation:

```
#define SHIFT_PERCPU_PTR(__p, __offset)                                 \
        RELOC_HIDE((typeof(*(__p)) __kernel __force *)(__p), (__offset))
```

`RELOC_HIDE` just returns offset `(typeof(ptr)) (__ptr + (off))` and it will be pointer of the variable.

That's all! Of course it is not full API, but the general part. It can be hard for the start, but to understand per-cpu variables feature need to understand mainly include/linux/percpu-defs.h magic.

Let's again look on the algorithm of getting pointer on per-cpu variable:

- Kernel creates multiply `.data..percpu` sections (ones perc-pu) during intialization process;
- All variables created with the `DEFINE_PER_CPU` macro will be reloacated to the first section or for CPU0;
- `__per_cpu_offset` array filled with the distance (`BOOT_PERCPU_OFFSET`) between `.data..percpu` sections;
- When `per_cpu_ptr` called for example for getting pointer on the certain per-cpu variable for the third CPU, `__per_cpu_offset` array will be accessed, where every index points to the certain CPU.

That's all.

Useful links

Linux boot

- Linux/x86 boot protocol

Protected mode

- 64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

Serial programming

- 8250 UART Programming
- Serial ports on OSDEV

VGA

- Video Graphics Array (VGA)

IO

- IO port programming

GCC and GAS

- GCC type attributes
- Assembler Directives

Thank you to all contributors:

- Akash Shende
- Jakub Kramarz
- ckrooss
- ecksun
- Maciek Makowski
- Thomas Marcelis
- Chris Costes
- nathansoz
- RubanDeventhiran
- fuzhli
- andars
- Alexandru Pana
- Bogdan Rădulescu
- zil
- codelitt
- gulyasm
- alx741
- Haddayn
- Daniel Campoverde Carrión
- Guillaume Gomez
- Leandro Moreira
- Jonatan Pålsson
- George Horrell
- Ciro Santilli
- Kevin Soules