Published
with GitBook

LINUX INSIDE

By 0xAX

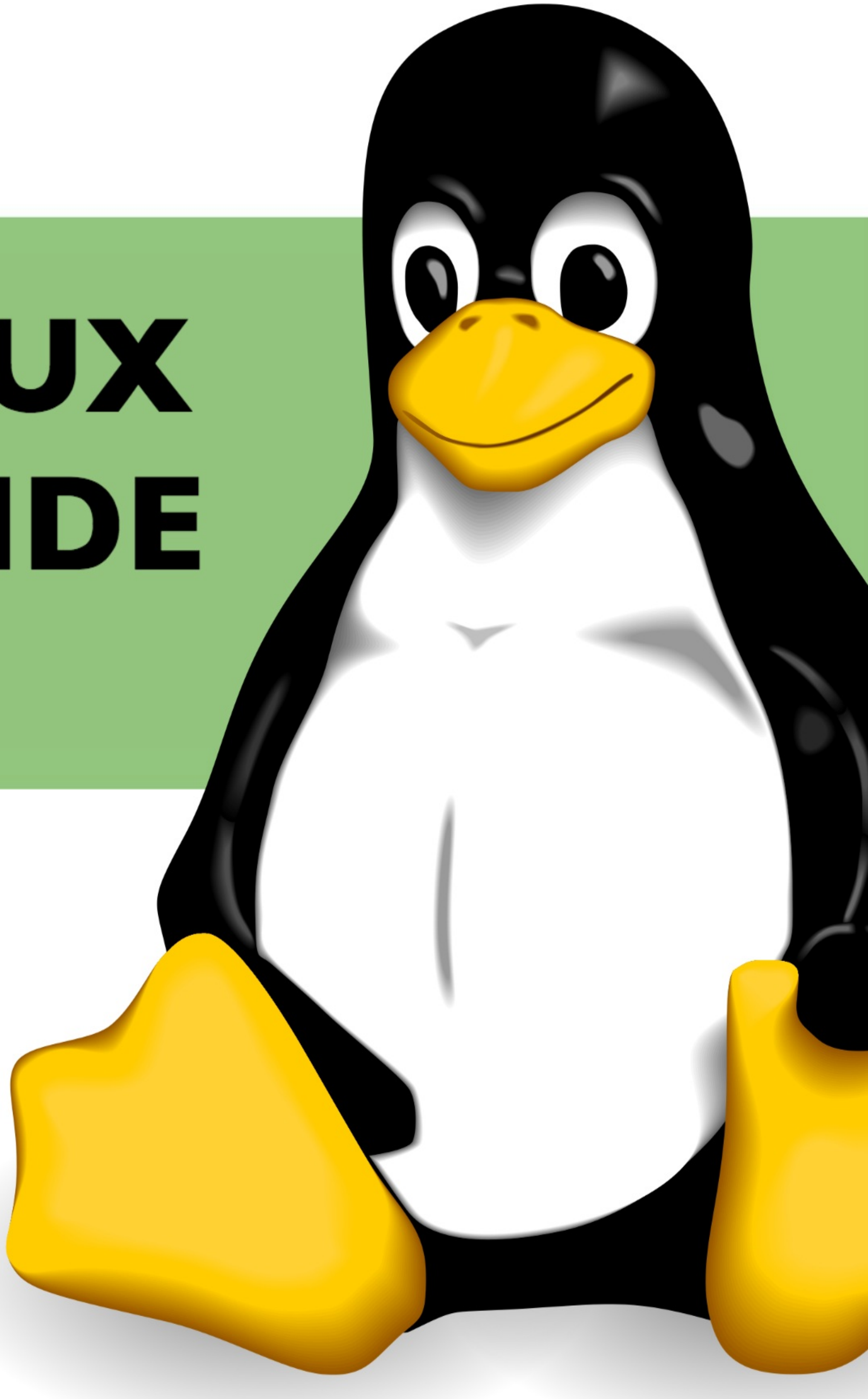


Table of Contents

linux-internals

A series of posts about the linux kernel and its insides.

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 internals, 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

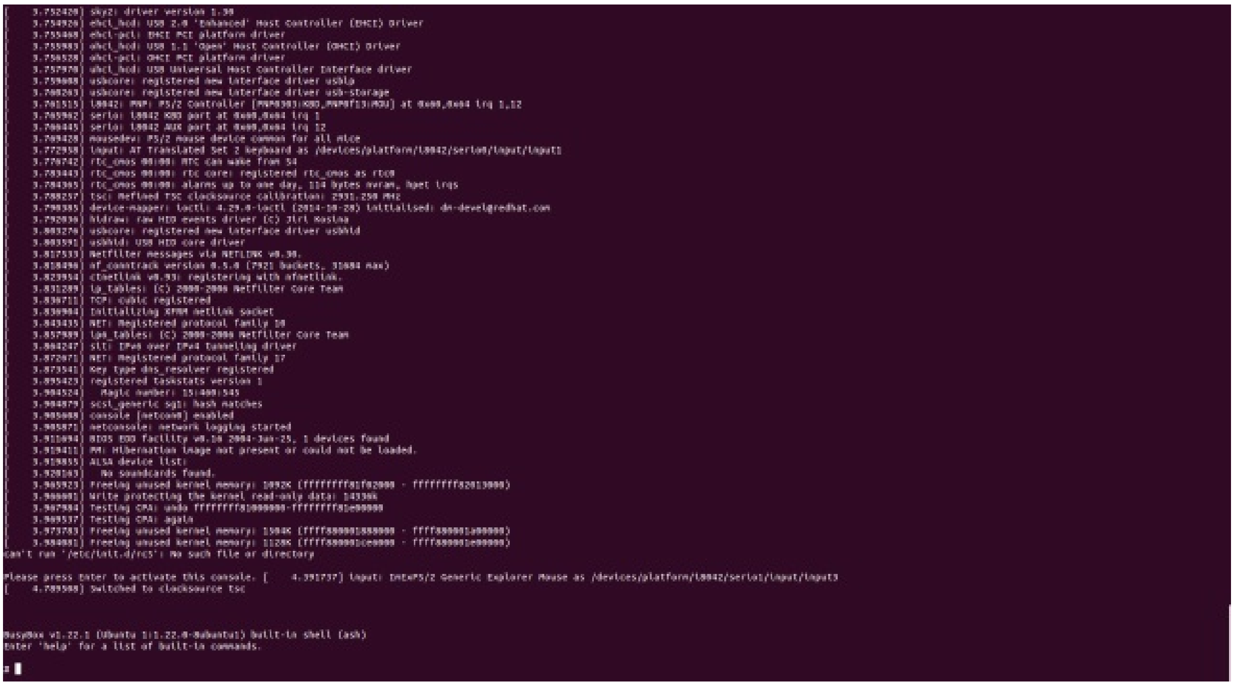
Support If you like `linux-insides` you can support me with:

donate flattr tips $2/week donate bitcoin donate gitbook

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.



Author

@0xAX

Kernel boot process

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

- From the bootloader to kernel - describes all stages from turning on the computer to 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:

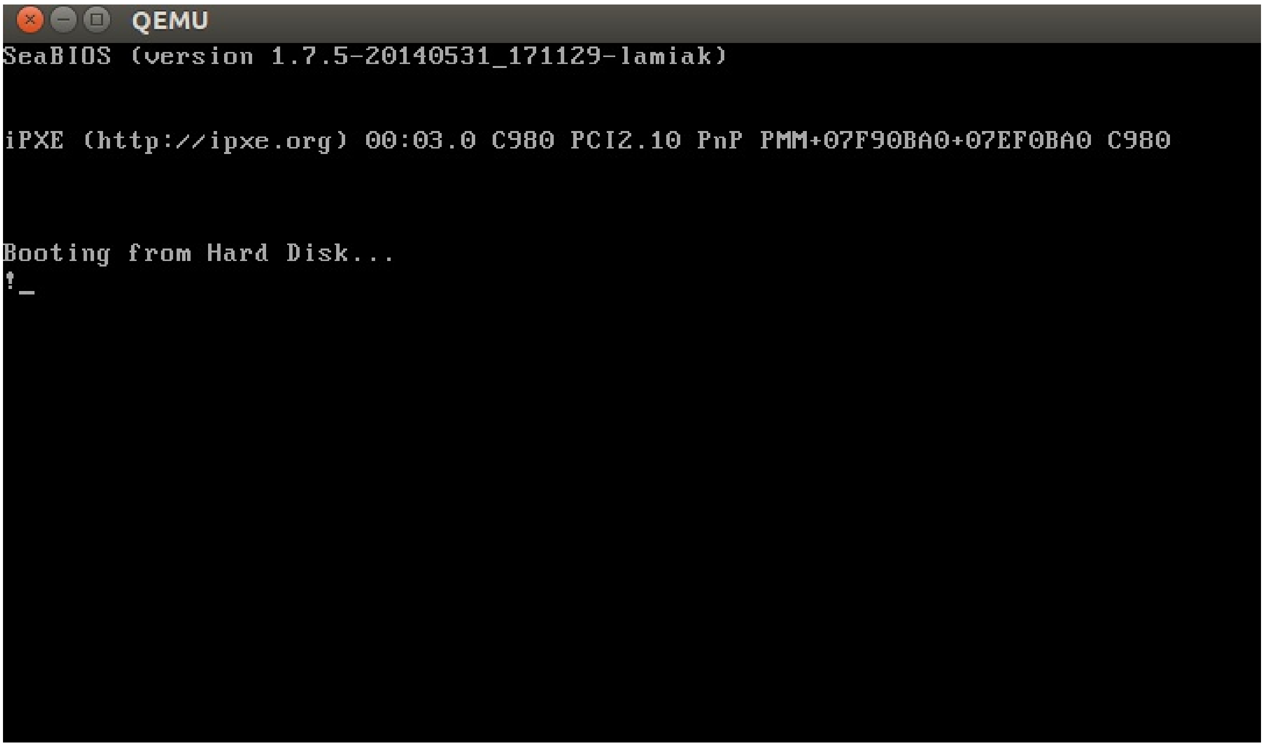
```
;
; Note: this example written with Intel syntax
;
[BITS 16]
[ORG  0x7c00]

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:



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 bin 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 `0xfffff` (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.

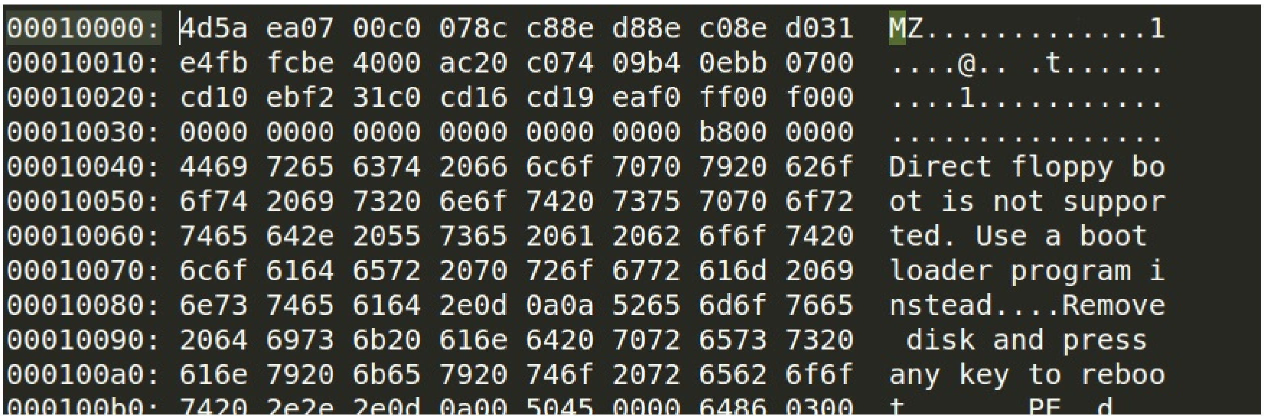
As we can see in kernel boot protocol, the memory map will be the following after kernel loading:

```
         | 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            |
```

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:



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

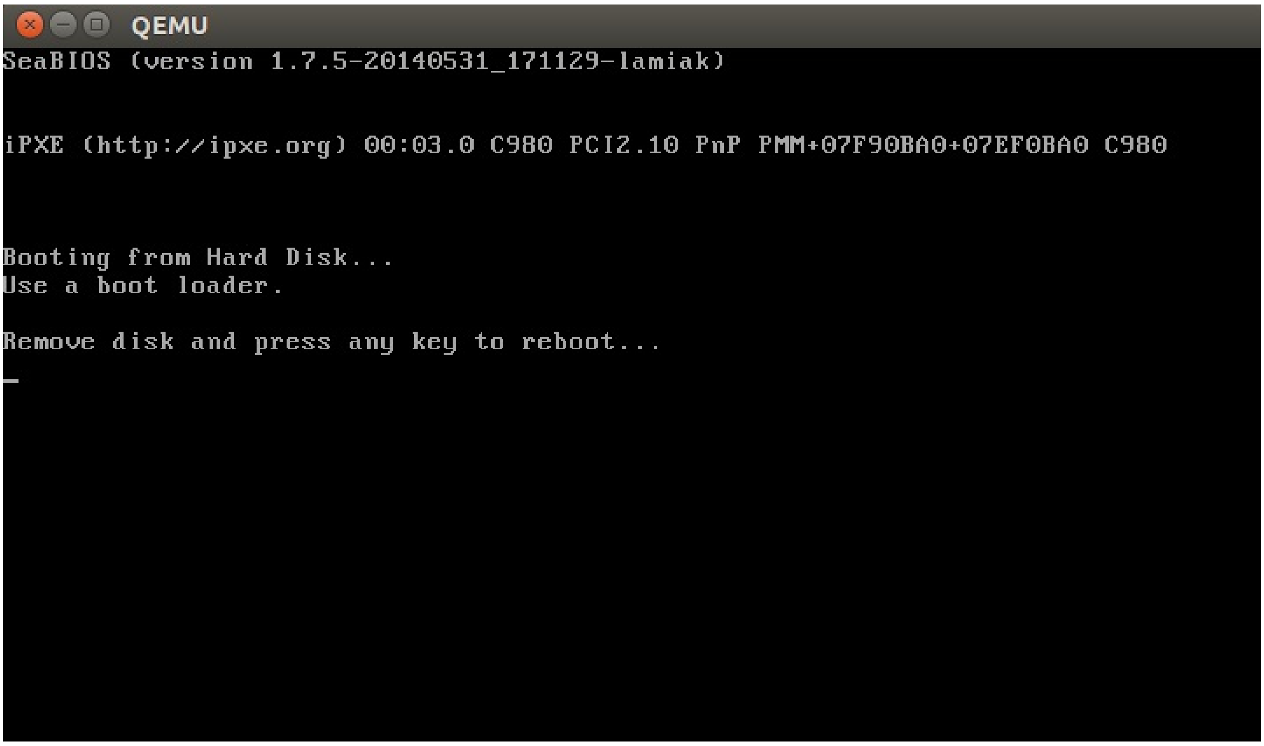
Start of kernel setup

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:



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)

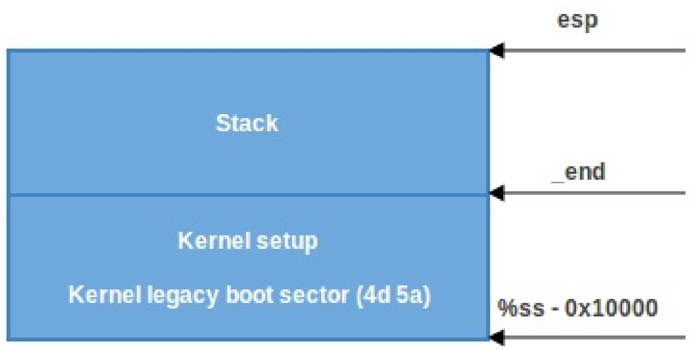
- `ss` is invalid and `CAN_USE_HEAP` flag is not set (see below)

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:



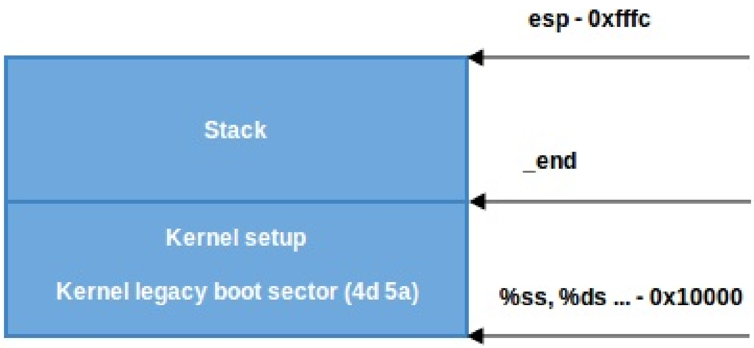
1. In the second case (`ss` != `ds`), first of all put _end (address of end of setup code) value in `dx`. And check `loadflags` header field with `testb` instruction too see if we can use heap or not. loadflags is a bitmask header which is defined as:

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

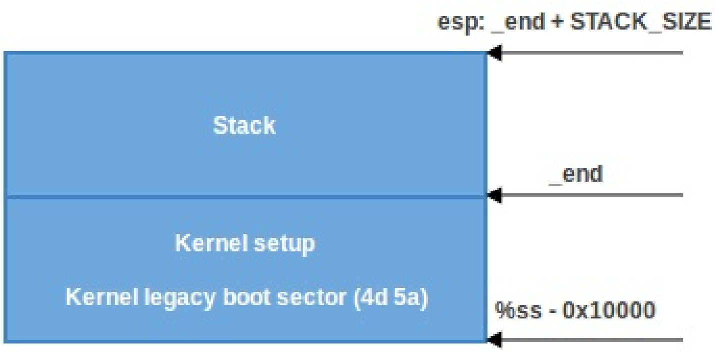
And as we can read in the boot protocol:

```
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

The last two steps that need to happen before we can jump to the main C code, are that we need to set up the bss area, and check the "magic" signature. Firstly, signature checking:

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

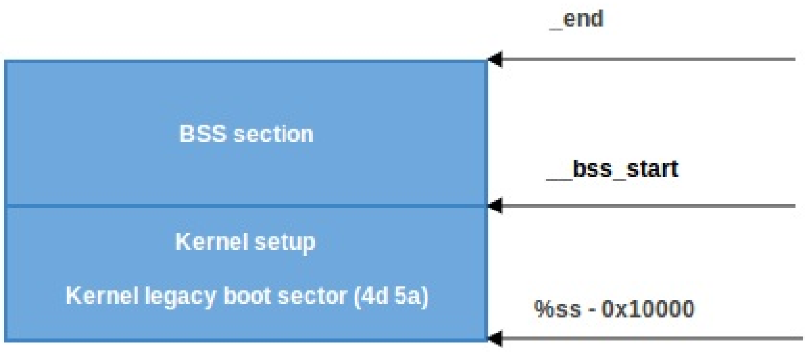
This simply consists of comparing the setup_sig against the magic number `0x5a5aaa55` ; if they are not equal, a fatal error is reported.

But if the magic number matches, knowing we have a set of correct segment registers, and a stack, we need only setup the bss section before jumping into the C code.

The bss section is used for storing statically allocated, uninitialized, data. Linux carefully ensures this area of memory is first blanked, using the following code:

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

First of all the __bss_start address is moved into `di`, and the `_end + 3` address (+3 - aligns to 4 bytes) is moved into `cx`. The `eax` register is cleared (using an `xor` instruction), and the bss section size (`cx` - `di`) is calculated and put into `cx`. Then, `cx` is divided by four (the size of a 'word'), and the `stosl` instruction is repeatedly used, storing the value of `eax` (zero) into the address pointed to by `di`, and automatically increasing `di` by four (this occurs until `cx` reaches zero). The net effect of this code, is that zeros are written through all words in memory 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 the kernel setup code and see what is `protected mode`, some preparation for the transition into it, the heap and console initialization, memory detection and much much 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 the protected mode? The 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. The Main reason to move away 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, sometimes even only 640 kilobytes.

Protected mode brought many changes, but the main is a different memory management.The 24-bit address bus was replaced with a 32-bit address bus. It allows to access to 4 gigabytes of physical address space. 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 only see segmentation. As you can read in the previous part, addresses consist of two parts in the real mode:

- Base address of segment
- Offset from the segment base

And we can get the 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 are described by the `Global Descriptor Table` (GDT) instead of segment registers.The GDT is a structure which resides in memory. There is no fixed place for it in memory, but its address is stored in the special `GDTR` register. Later we will see the GDT loading in the linux kernel code. There will be an operation for loading it into memory, something like:

```
lgdt gdt
```

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

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

The global descriptor table contains `descriptors` which describe memory segments. Every descriptor is 64-bit. General scheme of a 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 a 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 is 1 byte
    - if `G` is 1 and segment limit is 0, size of segment is 4096 bytes
    - if `G` is 0 and segment limit is 0xfffff, size of segment is 1 megabyte
    - if `G` is 1 and segment limit is 0xfffff, size of segment is 4 gigabytes
2. Base (0-15, 32-39 and 56-63 bits) defines the physical address of the segment's start address.

3. Type (40-47 bits) defines the type of segment and kinds of access to it. Next `S` flag specifies descriptor type. if `S` is 0 then this segment is a system segment, whereas if `S` is 1 then this is a code or data segment (Stack segments are data segments which must be read/write segments). If the 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 the 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, you can read more about it 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. It can be 0-3 where 0 is the most privileged.

2. P flag - indicates if segment is present 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 then the code segment executes in 64 bit mode.

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

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

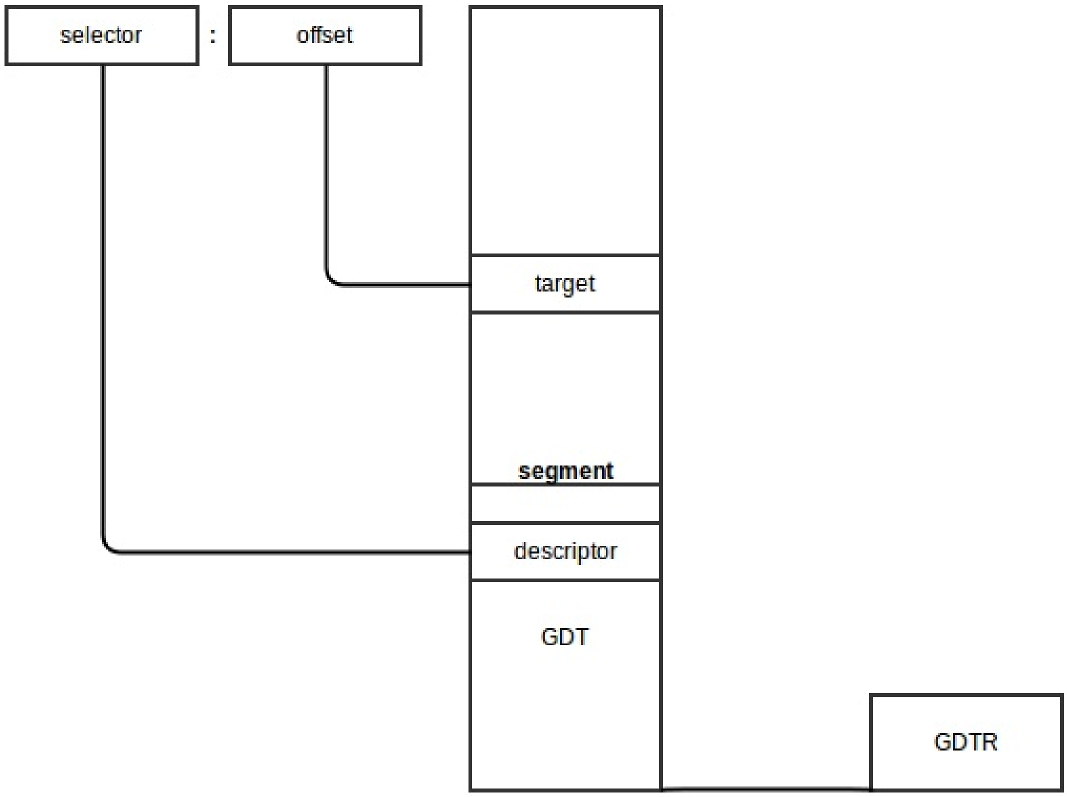
```
-----------------------------
|       Index    | TI | RPL |
-----------------------------
```

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

Every segment register has a visible and hidden part. When a selector is loaded into one of the segment registers, it will be stored into the visible part. The hidden part contains the base address, limit and access information of the descriptor which pointed to the selector. The following steps are needed to get the 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 load the descriptor into the hidden part of segment register;
- Base address (from segment descriptor) + offset will be the linear address of the segment which is the physical address (if paging is disabled).

Schematically it will look like this:



THe algorithm for the 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 the 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, etc... Let's look into it.

Copying boot parameters into the "zeropage"

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

The `boot_params` structure contains the `struct setup_header hdr` field. This structure contains the same fields as defined in linux boot protocol and is filled by the boot loader and also at kernel compile/build time. `copy_boot_params` 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.

Note that it copies `hdr` with `memcpy` function which is defined in the copy.S source file. Let's have a look inside:

```
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 now assembly again :) First of all we can see that `memcpy` and other routines which are defined here, start and end with the two macros: `GLOBAL` and `ENDPROC` . GLOBAL is described in arch/x86/include/asm/linkage.h which defines `globl` directive and the label for it. ENDPROC is described in 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 on the stack to preserve their values. `memcpy` (and other functions in copy.S) use `fastcall` calling conventions. So it gets incoming parameters from the `ax` , `dx` and `cx` registers. Calling `memcpy` looks like this:

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

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

Console initialization

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

It tries to find the `earlyprintk` option in the command line and if the search was successful, it parses the port address and baud rate of the serial port and initializes the 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 the first output:

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

`puts` definition is in tty.c. As we can see it prints character by character in the loop by calling The `putchar` function. Let's look into the `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 for the `\n` symbol and if it is found, prints `\r` before. After that it outputs the character on the VGA screen by calling the BIOS with the `0x10` interrupt 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);
}
```

Here `initregs` takes the `biosregs` structure and first fills `biosregs` with zeros using the `memset` function and then fills it with register 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 like the `memcpy` function, which means that the function gets parameters from `ax`, `dx` and `cx` registers.

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

The next instruction multiplies `eax` with `0x01010101`. It needs to because `memset` will copy 4 bytes at the same time. For example we need to fill a structure with `0x7` with memset. `eax` will contain `0x00000007` value in this case. So if we multiply `eax` with `0x01010101`, we will get `0x07070707` and now we can copy these 4 bytes into the 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 is filled with `memset`, `bios_putchar` calls the 0x10 interrupt which prints a character. Afterwards it checks if the serial port was 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 header.S (see previous part), the kernel needs to initialize the heap with the init_heap function.

First of all `init_heap` checks the `CAN_USE_HEAP` flag from the `loadflags` kernel setup header and calculates the 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`.

Then there is the `heap_end` calculation which is `heap_end_ptr` or `_end` + 512 and a check if `heap_end` is greater than `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 the API for it is implemented in next posts.

CPU validation

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

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

Memory detection

The next step is memory detection by the `detect_memory` function. It uses different programming interfaces for memory detection like `0xe820`, `0xe801` and `0x88`. We will see only the implementation of 0xE820 here. Let's look into the `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 fills registers with special values for the `0xe820` call:

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

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

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

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

Ultimately, it does iterations in the loop to collect data from the address allocation table and writes this data into the `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 this 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 the initialization of the keyboard with the call of the `keyboard_init` function. At first `keyboard_init` initializes registers using the `initregs` function and calling the 0x16 interrupt for getting the keyboard status. After this it calls 0x16 again to set repeat rate and delay.

Querying

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

The query_mca routine calls the 0x15 BIOS interrupt to get the 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 the `ah` register with `0xc0` and calls the `0x15` BIOS interruption. After the interrupt execution it checks the carry flag and if it is set to 1, BIOS doesn't support `MCA`. If carry flag is set to 0, `ES:BX` will contain a 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 the `set_fs` routine and pass the value of the `es` register to it. Implementation of `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 the value of the `seg` parameter and puts it into the `fs` register. There are many functions in boot.h like `set_fs`, for example `set_gs`, `fs`, `gs` for reading a value in it and etc...

In the end of `query_mca` it just copies the table which pointed to 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 the result to `boot_params`.

The following query_apm_bios function 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 is supported).

Next it calls the `0x15` again, but with `ax = 0x5304` for disconnecting the `APM` interface and connect the 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 the query_edd function, which asks `Enhanced Disk Drive` information from the BIOS. Let's look into the `query_edd` implementation.

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

If EDD is enabled, `query_edd` goes over BIOS-supported hard disks and queries EDD information in the following 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 into the array of edd_info structures. `get_edd_info` checks that EDD is present by invoking the `0x13` interrupt with `ah` as `0x41` and if EDD is present, `get_edd_info` again calls the `0x13` interrupt, but with `ah` as `0x48` and `si` contianing the address of the buffer where EDD informantion will be stored.

Conclusion

This is the end of the second part about linux kernel internals. In the next part we will see video mode setting and the rest of preparations before transition to protected mode and directly transitioning 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;
```

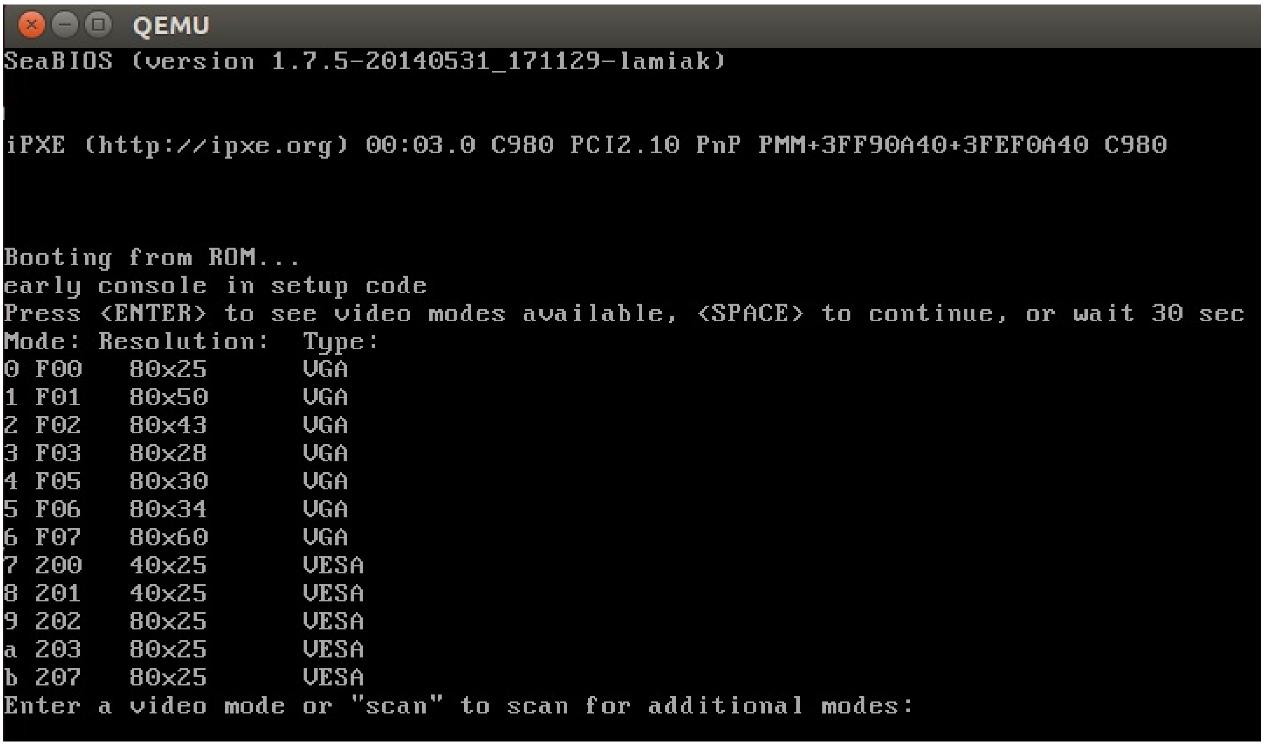
which we filled in the `copy_boot_params` function (you can read about it in the previous post). `vid_mode` is an obligatory field which filled by the bootloader. You can find information about it in the kernel boot protocol:

```
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:



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 align 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 addresses 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 chosen 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 initialization process

You will find here a couple of posts which describe the full cycle of kernel initialization from its first steps after the kernel has decompressed to the start of the first process run by the kernel itself.

- First steps after kernel decompression - describes first steps in the kernel.
- Early interrupt and exception handling - describes early interrupts initialization and early page fault handler.
- Last preparations before the kernel entry point - describes the last preparations before the call of the `start_kernel`.
- Kernel entry point - describes first steps in the kernel generic code.
- Continue of architecture-specific initializations - describes architecture-specific initialization.
- Architecture-specific initializations, again... - describes continue of the architecture-specific initialization process.
- The End of the architecture-specific initializations, almost... - describes the end of the `setup_arch` related stuff.

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

Okay, 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 definition of the `startup_64` routine and it defined in the `__HEAD` section, which is just:

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

We can see definition 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 physical addresses from the linker script. Note that address 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 subtract `$_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 contain physical address of the `text` - `0x1000000` after this calculation. We need to calculate it because kernel can be runned not on the default address, but now we know actual 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
```

Okay, 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 of the page table directories, 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 subtract `__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 subtract `__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 definition 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 megabytes will be modules memory space).

Now we know Let's back to our code which is in the beginning of the section. Remember that `rbp` contains actual physical address of the `_text` section. We just add this address to the base address 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.

Identity mapping setup

Now we can see set up the identity mapping early page tables. 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 contaitns 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` definition. 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 every time:

```
    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 address 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 constants.

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

Kernel initialization. Part 2.

Early interrupt and exception handling

In the previous part we stopped before setting of early interrupt handlers. We continue in this part and will know more about interrupt and exception handling.

Remember that we stopped before following loop:

```
for (i = 0; i < NUM_EXCEPTION_VECTORS; i++)
    set_intr_gate(i, early_idt_handlers[i]);
```

from the arch/x86/kernel/head64.c source code file. But before we started to sort out this code, we need to know about interrupts and handlers.

Some theory

Interrupt is an event caused by software or hardware to the CPU. On interrupt, CPU stops the current task and transfer control to the interrupt handler, which handles interruption and transfer control back to the previously stopped task. We can split interrupts on three types:

- Software interrupts - when a software signals CPU that it needs kernel attention. These interrupts generally used for system calls;
- Hardware interrupts - when a hardware, for example button pressed on a keyboard;
- Exceptions - interrupts generated by CPU, when the CPU detects error, for example division by zero or accessing a memory page which is not in RAM.

Every interrupt and exception is assigned an unique number which called - `vector number`. `Vector number` can be any number from `0` to `255`. There is common practice to use first `32` vector numbers for exceptions, and vector numbers from `31` to `255` are used for user-defined interrupts. We can see it in the code above - `NUM_EXCEPTION_VECTORS`, which defined as:

```
#define NUM_EXCEPTION_VECTORS 32
```

CPU uses vector number as an index in the `Interrupt Descriptor Table` (we will see description of it soon). CPU catch interrupts from the APIC or through it's pins. Following table shows `0-31` exceptions:

```
----------------------------------------------------------------------------------------------
|Vector|Mnemonic|Description         |Type |Error Code|Source                                |
----------------------------------------------------------------------------------------------
|0     | #DE    |Divide Error        |Fault|NO        |DIV and IDIV                          |
|---------------------------------------------------------------------------------------------
|1     | #DB    |Reserved            |F/T  |NO        |                                      |
|---------------------------------------------------------------------------------------------
|2     | ---    |NMI                 |INT  |NO        |external NMI                          |
|---------------------------------------------------------------------------------------------
|3     | #BP    |Breakpoint          |Trap |NO        |INT 3                                 |
|---------------------------------------------------------------------------------------------
|4     | #OF    |Overflow            |Trap |NO        |INTO  instruction                     |
|---------------------------------------------------------------------------------------------
|5     | #BR    |Bound Range Exceeded|Fault|NO        |BOUND instruction                     |
|---------------------------------------------------------------------------------------------
```

```
|6     | #UD    |Invalid Opcode      |Fault|NO        |UD2 instruction                        |
|---------------------------------------------------------------------------------------------
|7     | #NM    |Device Not Available|Fault|NO        |Floating point or [F]WAIT              |
|---------------------------------------------------------------------------------------------
|8     | #DF    |Double Fault        |Abort|YES       |Ant instrctions which can generate NMI|
|---------------------------------------------------------------------------------------------
|9     | ---    |Reserved            |Fault|NO        |                                       |
|---------------------------------------------------------------------------------------------
|10    | #TS    |Invalid TSS         |Fault|YES       |Task switch or TSS access              |
|---------------------------------------------------------------------------------------------
|11    | #NP    |Segment Not Present |Fault|NO        |Accessing segment register             |
|---------------------------------------------------------------------------------------------
|12    | #SS    |Stack-Segment Fault |Fault|YES       |Stack operations                       |
|---------------------------------------------------------------------------------------------
|13    | #GP    |General Protection  |Fault|YES       |Memory reference                       |
|---------------------------------------------------------------------------------------------
|14    | #PF    |Page fault          |Fault|YES       |Memory reference                       |
|---------------------------------------------------------------------------------------------
|15    | ---    |Reserved            |     |NO        |                                       |
|---------------------------------------------------------------------------------------------
|16    | #MF    |x87 FPU fp error    |Fault|NO        |Floating point or [F]Wait              |
|---------------------------------------------------------------------------------------------
|17    | #AC    |Alignment Check     |Fault|YES       |Data reference                         |
|---------------------------------------------------------------------------------------------
|18    | #MC    |Machine Check       |Abort|NO        |                                       |
|---------------------------------------------------------------------------------------------
|19    | #XM    |SIMD fp exception   |Fault|NO        |SSE[2,3] instructions                  |
|---------------------------------------------------------------------------------------------
|20    | #VE    |Virtualization exc. |Fault|NO        |EPT violations                         |
|---------------------------------------------------------------------------------------------
|21-31 | ---    |Reserved            |INT  |NO        |External interrupts                    |
 ---------------------------------------------------------------------------------------------
```

To react on interrupt CPU uses special structure - Interrupt Descriptor Table or IDT. IDT is an array of 8-byte descriptors like Global Descriptor Table, but IDT entries are called `gates`. CPU multiplies vector number on 8 to find index of the IDT entry. But in 64-bit mode IDT is an array of 16-byte descriptors and CPU multiplies vector number on 16 to find index of the entry in the IDT. We remember from the previous part that CPU uses special `GDTR` register to locate Global Descriptor Table, so CPU uses special register `IDTR` for Interrupt Descriptor Table and `lidt` instruuction for loading base address of the table into this register.

64-bit mode IDT entry has following structure:

```
127                                                                             96
 --------------------------------------------------------------------------------
|                                                                               |
|                                Reserved                                       |
|                                                                               |
 --------------------------------------------------------------------------------
95                                                                              64
 --------------------------------------------------------------------------------
|                                                                               |
|                               Offset 63..32                                   |
|                                                                               |
 --------------------------------------------------------------------------------
63                               48 47      46  44   42    39             34    32
 --------------------------------------------------------------------------------
|                                  |       |  D  |   |     |      |   |   |     |
|       Offset 31..16              |   P   |  P  | 0 |Type |0 0 0 | 0 | 0 | IST |
|                                  |       |  L  |   |     |      |   |   |     |
 --------------------------------------------------------------------------------
31                                   15 16                                      0
 --------------------------------------------------------------------------------
|                                      |                                        |
|          Segment Selector            |                 Offset 15..0           |
|                                      |                                        |
 --------------------------------------------------------------------------------
```

Where:

* Offset - is offset to entry point of an interrupt handler;
* DPL - Descriptor Privilege Level;
* P - Segment Present flag;
* Segment selector - a code segment selector in GDT or LDT
* IST - provides ability to switch to a new stack for interrupts handling.

And the last `Type` field describes type of the `IDT` entry. There are three different kinds of handlers for interrupts:

* Task descriptor
* Interrupt descriptor
* Trap descriptor

Interrupt and trap descriptors contain a far pointer to the entry point of the interrupt handler. Only one difference between these types is how CPU handles `IF` flag. If interrupt handler was accessed through interrupt gate, CPU clear the `IF` flag to prevent other interrupts while current interrupt handler executes. After that current interrupt handler executes, CPU sets the `IF` flag again with `iret` instruction.

Other bits reserved and must be 0.

Now let's look how CPU handles interrupts:

* CPU save flags register, `CS`, and instruction pointer on the stack.
* If interrupt causes an error code (like `#PF` for example), CPU saves an error on the stack after instruction pointer;
* After interrupt handler executed, `iret` instruction used to return from it.

Now let's back to code.

Fill and load IDT

We stopped at the following point:

```
for (i = 0; i < NUM_EXCEPTION_VECTORS; i++)
    set_intr_gate(i, early_idt_handlers[i]);
```

Here we call `set_intr_gate` in the loop, which takes two parameters:

* Number of an interrupt;
* Address of the idt handler.

and inserts an interrupt gate in the nth `IDT` entry. First of all let's look on the `early_idt_handlers`. It is an array which contains address of the first 32 interrupt handlers:

```
extern const char early_idt_handlers[NUM_EXCEPTION_VECTORS][2+2+5];
```

We're filling only first 32 IDT entries because all of the early setup runs with interrupts disabled, so there is no need to set up early exception handlers for vectors greater than 32. `early_idt_handlers` contains generic idt handlers and we can find it in the arch/x86/kernel/head_64.S, we will look it soon.

Now let's look on `set_intr_gate` implementation:

```
#define set_intr_gate(n, addr)                                  \
        do {                                                    \
```

```
                BUG_ON((unsigned)n > 0xFF);                         \
                _set_gate(n, GATE_INTERRUPT, (void *)addr, 0, 0,        \
                          __KERNEL_CS);                             \
                _trace_set_gate(n, GATE_INTERRUPT, (void *)trace_##addr,\
                                0, 0, __KERNEL_CS);                 \
        } while (0)
```

First of all it checks with that passed interrupt number is not greater than `255` with `BUG_ON` macro. We need to do this check because we can have only 256 interrupts. After this it calls `_set_gate` which writes address of an interrupt gate to the `IDT`:

```
static inline void _set_gate(int gate, unsigned type, void *addr,
                             unsigned dpl, unsigned ist, unsigned seg)
{
        gate_desc s;
        pack_gate(&s, type, (unsigned long)addr, dpl, ist, seg);
        write_idt_entry(idt_table, gate, &s);
        write_trace_idt_entry(gate, &s);
}
```

At the start of `_set_gate` function we can see call of the `pack_gate` function which fills `gate_desc` structure with the given values:

```
static inline void pack_gate(gate_desc *gate, unsigned type, unsigned long func,
                             unsigned dpl, unsigned ist, unsigned seg)
{
        gate->offset_low        = PTR_LOW(func);
        gate->segment           = __KERNEL_CS;
        gate->ist               = ist;
        gate->p                 = 1;
        gate->dpl               = dpl;
        gate->zero0             = 0;
        gate->zero1             = 0;
        gate->type              = type;
        gate->offset_middle     = PTR_MIDDLE(func);
        gate->offset_high       = PTR_HIGH(func);
}
```

As mentioned above we fill gate descriptor in this function. We fill three parts of the address of the interrupt handler with the address which we got in the main loop (address of the interrupt handler entry point). We are using three following macro to split address on three parts:

```
#define PTR_LOW(x) ((unsigned long long)(x) & 0xFFFF)
#define PTR_MIDDLE(x) (((unsigned long long)(x) >> 16) & 0xFFFF)
#define PTR_HIGH(x) ((unsigned long long)(x) >> 32)
```

With the first `PTR_LOW` macro we get the first 2 bytes of the address, with the second `PTR_MIDDLE` we get the second 2 bytes of the address and with the third `PTR_HIGH` macro we get the last 4 bytes of the address. Next we setup the segment selector for interrupt handler, it will be our kernel code segment - `__KERNEL_CS`. In the next step we fill `Interrupt Stack Table` and `Descriptor Privilege Level` (highest privilege level) with zeros. And we set `GAT_INTERRUPT` type in the end.

Now we have filled IDT entry and we can call `native_write_idt_entry` function which just copies filled `IDT` entry to the `IDT`:

```
static inline void native_write_idt_entry(gate_desc *idt, int entry, const gate_desc *gate)
{
        memcpy(&idt[entry], gate, sizeof(*gate));
}
```

After that main loop will finished, we will have filled `idt_table` array of `gate_desc` structures and we can load `IDT` with:

```
load_idt((const struct desc_ptr *)&idt_descr);
```

Where `idt_descr` is:

```
struct desc_ptr idt_descr = { NR_VECTORS * 16 - 1, (unsigned long) idt_table };
```

and `load_idt` just executes `lidt` instruction:

```
asm volatile("lidt %0"::"m" (*dtr));
```

You can note that there are calls of the `_trace_*` functions in the `_set_gate` and other functions. These functions fills `IDT` gates in the same manner that `_set_gate` but with one difference. These functions use `trace_idt_table` Interrupt Descriptor Table instead of `idt_table` for tracepoints (we will cover this theme in the another part).

Okay, now we have filled and loaded Interrupt Descriptor Table, we know how the CPU acts during interrupt. So now time to deal with interrupts handlers.

Early interrupts handlers

As you can read above, we filled `IDT` with the address of the `early_idt_handlers`. We can find it in the [arch/x86/kernel/head_64.S](https://github.com/torvalds/linux/blob/master/arch/x86/kernel/head_64.S):

```
    .globl early_idt_handlers
early_idt_handlers:
    i = 0
    .rept NUM_EXCEPTION_VECTORS
    .if (EXCEPTION_ERRCODE_MASK >> i) & 1
    ASM_NOP2
    .else
    pushq $0
    .endif
    pushq $i
    jmp early_idt_handler
    i = i + 1
    .endr
```

We can see here, interrupt handlers generation for the first 32 exceptions. We check here, if exception has error code then we do nothing, if exception does not return error code, we push zero to the stack. We do it for that would stack was uniform. After that we push exception number on the stack and jump on the `early_idt_handler` which is generic interrupt handler for now. As i wrote above, CPU pushes flag register, `CS` and `RIP` on the stack. So before `early_idt_handler` will be executed, stack will contain following data:

```
|--------------------|
| %rflags            |
| %cs                |
| %rip               |
| rsp --> error code |
|--------------------|
```

Now let's look on the `early_idt_handler` implementation. It locates in the same [arch/x86/kernel/head_64.S](https://github.com/torvalds/linux/blob/master/arch/x86/kernel/head_64.S). First of all we can see check for NMI, we no need to handle it, so just ignore they in the `early_idt_handler`:

```
    cmpl $2,(%rsp)
    je is_nmi
```

where `is_nmi`:

```
is_nmi:
    addq $16,%rsp
    INTERRUPT_RETURN
```

we drop error code and vector number from the stack and call `INTERRUPT_RETURN` which is just `iretq`. As we checked the vector number and it is not `NMI`, we check `early_recursion_flag` to prevent recursion in the `early_idt_handler` and if it's correct we save general registers on the stack:

```
    pushq %rax
    pushq %rcx
    pushq %rdx
    pushq %rsi
    pushq %rdi
    pushq %r8
    pushq %r9
    pushq %r10
    pushq %r11
```

we need to do it to prevent wrong values in it when we return from the interrupt handler. After this we check segment selector in the stack:

```
    cmpl $__KERNEL_CS,96(%rsp)
    jne 11f
```

it must be equal to the kernel code segment and if it is not we jump on label `11` which prints `PANIC` message and makes stack dump.

After code segment was checked, we check the vector number, and if it is `#PF`, we put value from the `cr2` to the `rdi` register and call `early_make_pgtable` (well see it soon):

```
    cmpl $14,72(%rsp)
    jnz 10f
    GET_CR2_INTO(%rdi)
    call early_make_pgtable
    andl %eax,%eax
    jz 20f
```

If vector number is not `#PF`, we restore general purpose registers from the stack:

```
    popq %r11
    popq %r10
    popq %r9
    popq %r8
    popq %rdi
    popq %rsi
    popq %rdx
    popq %rcx
    popq %rax
```

and exit from the handler with `iret`.

It is the end of the first interrupt handler. Note that it is very early interrupt handler, so it handles only Page Fault now. We will see handlers for the other interrupts, but now let's look on the page fault handler.

Page fault handling

In the previous paragraph we saw first early interrupt handler which checks interrupt number for page fault and calls `early_make_pgtable` for building new page tables if it is. We need to have `#PF` handler in this step because there are plans to add ability to load kernel above 4G and make access to `boot_params` structure above the 4G.

You can find implementation of the `early_make_pgtable` in the arch/x86/kernel/head64.c and takes one parameter - address from the `cr2` register, which caused Page Fault. Let's look on it:

```
int __init early_make_pgtable(unsigned long address)
{
    unsigned long physaddr = address - __PAGE_OFFSET;
    unsigned long i;
    pgdval_t pgd, *pgd_p;
    pudval_t pud, *pud_p;
    pmdval_t pmd, *pmd_p;
    ...
    ...
    ...
}
```

It starts from the definition of some variables which have `*val_t` types. All of these types are just:

```
typedef unsigned long   pgdval_t;
```

Also we will operate with the `*_t` (not val) types, for example `pgd_t` and etc... All of these types defined in the arch/x86/include/asm/pgtable_types.h and represent structures like this:

```
typedef struct { pgdval_t pgd; } pgd_t;
```

For example,

```
extern pgd_t early_level4_pgt[PTRS_PER_PGD];
```

Here `early_level4_pgt` presents early top-level page table directory which consists of an array of `pgd_t` types and `pgd` points to low-level page entries.

After we made the check that we have no invalid address, we're getting the address of the Page Global Directory entry which contains `#PF` address and put it's value to the `pgd` variable:

```
pgd_p = &early_level4_pgt[pgd_index(address)].pgd;
pgd = *pgd_p;
```

In the next step we check `pgd`, if it contains correct page global directory entry we put physical address of the page global directory entry and put it to the `pud_p` with:

```
pud_p = (pudval_t *)((pgd & PTE_PFN_MASK) + __START_KERNEL_map - phys_base);
```

where `PTE_PFN_MASK` is a macro:

```
#define PTE_PFN_MASK            ((pteval_t)PHYSICAL_PAGE_MASK)
```

which expands to:

```
(~(PAGE_SIZE-1)) & ((1 << 46) - 1)
```

or

```
0b1111111111111111111111111111111111111111111111
```

which is 46 bits to mask page frame.

If `pgd` does not contain correct address we check that `next_early_pgt` is not greater than `EARLY_DYNAMIC_PAGE_TABLES` which is `64` and present a fixed number of buffers to set up new page tables on demand. If `next_early_pgt` is greater than `EARLY_DYNAMIC_PAGE_TABLES` we reset page tables and start again. If `next_early_pgt` is less than `EARLY_DYNAMIC_PAGE_TABLES`, we create new page upper directory pointer which points to the current dynamic page table and writes it's physical address with the `_KERPG_TABLE` access rights to the page global directory:

```
if (next_early_pgt >= EARLY_DYNAMIC_PAGE_TABLES) {
    reset_early_page_tables();
    goto again;
}

pud_p = (pudval_t *)early_dynamic_pgts[next_early_pgt++];
for (i = 0; i < PTRS_PER_PUD; i++)
    pud_p[i] = 0;
*pgd_p = (pgdval_t)pud_p - __START_KERNEL_map + phys_base + _KERNPG_TABLE;
```

After this we fix up address of the page upper directory with:

```
pud_p += pud_index(address);
pud = *pud_p;
```

In the next step we do the same actions as we did before, but with the page middle directory. In the end we fix address of the page middle directory which contains maps kernel text+data virtual addresses:

```
pmd = (physaddr & PMD_MASK) + early_pmd_flags;
pmd_p[pmd_index(address)] = pmd;
```

After page fault handler finished it's work and as result our `early_level4_pgt` contains entries which point to the valid addresses.

Conclusion

This is the end of the second 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 all steps before kernel entry point - `start_kernel` function.

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

- GNU assembly .rept
- APIC
- NMI
- Previous part

Kernel initialization. Part 3.

Last preparations before the kernel entry point

This is the third part of the Linux kernel initialization process series. In the previous part we saw early interrupt and exception handling and will continue to dive into the linux kernel initialization process in the current part. Our next point is 'kernel entry point' - `start_kernel` function from the init/main.c source code file. Yes, technically it is not kernel's entry point but the start of the generic kernel code which does not depend on certain architecture. But before we will see call of the `start_kernel` function, we must do some preparations. So let's continue.

boot_params again

In the previous part we stopped at setting Interrupt Descriptor Table and loading it in the `IDTR` register. At the next step after this we can see a call of the `copy_bootdata` function:

```
copy_bootdata(__va(real_mode_data));
```

This function takes one argument - virtual address of the `real_mode_data`. Remember that we passed the address of the `boot_params` structure from arch/x86/include/uapi/asm/bootparam.h to the `x86_64_start_kernel` function as first argument in arch/x86/kernel/head_64.S:

```
    /* rsi is pointer to real mode structure with interesting info.
       pass it to C */
    movq    %rsi, %rdi
```

Now let's look at `__va` macro. This macro defined in init/main.c:

```
#define __va(x)                 ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

where `PAGE_OFFSET` is `__PAGE_OFFSET` which is `0xffff880000000000` and the base virtual address of the direct mapping of all physical memory. So we're getting virtual address of the `boot_params` structure and pass it to the `copy_bootdata` function, where we copy `real_mod_data` to the `boot_params` which is declared in the arch/x86/kernel/setup.h

```
extern struct boot_params boot_params;
```

Let's look at the `copy_boot_data` implementation:

```
static void __init copy_bootdata(char *real_mode_data)
{
    char * command_line;
    unsigned long cmd_line_ptr;

    memcpy(&boot_params, real_mode_data, sizeof boot_params);
    sanitize_boot_params(&boot_params);
    cmd_line_ptr = get_cmd_line_ptr();
    if (cmd_line_ptr) {
        command_line = __va(cmd_line_ptr);
        memcpy(boot_command_line, command_line, COMMAND_LINE_SIZE);
    }
```

```
}
```

First of all, note that this function is declared with `__init` prefix. It means that this function will be used only during the initialization and used memory will be freed.

We can see declaration of two variables for the kernel command line and copying `real_mode_data` to the `boot_params` with the `memcpy` function. The next call of the `sanitize_boot_params` function which fills some fields of the `boot_params` structure like `ext_ramdisk_image` and etc... if bootloaders which fail to initialize unknown fields in `boot_params` to zero. After this we're getting address of the command line with the call of the `get_cmd_line_ptr` function:

```
unsigned long cmd_line_ptr = boot_params.hdr.cmd_line_ptr;
cmd_line_ptr |= (u64)boot_params.ext_cmd_line_ptr << 32;
return cmd_line_ptr;
```

which gets the 64-bit address of the command line from the kernel boot header and returns it. In the last step we check that we got `cmd_line_pty`, getting its virtual address and copy it to the `boot_command_line` which is just an array of bytes:

```
extern char __initdata boot_command_line[];
```

After this we will have copied kernel command line and `boot_params` structure. In the next step we can see call of the `load_ucode_bsp` function which loads processor microcode, but we will not see it here.

After microcode was loaded we can see the check of the `console_loglevel` and the `early_printk` function which prints `Kernel Alive` string. But you'll never see this output because `early_printk` is not initilized yet. It is a minor bug in the kernel and i sent the patch - commit and you will see it in the mainline soon. So you can skip this code.

Move on init pages

In the next step as we have copied `boot_params` structure, we need to move from the early page tables to the page tables for initialization process. We already set early page tables for switchover, you can read about it in the previous part and dropped all it in the `reset_early_page_tables` function (you can read about it in the previous part too) and kept only kernel high mapping. After this we call:

```
    clear_page(init_level4_pgt);
```

function and pass `init_level4_pgt` which defined also in the arch/x86/kernel/head_64.S and looks:

```
NEXT_PAGE(init_level4_pgt)
    .quad   level3_ident_pgt - __START_KERNEL_map + _KERNPG_TABLE
    .org    init_level4_pgt + L4_PAGE_OFFSET*8, 0
    .quad   level3_ident_pgt - __START_KERNEL_map + _KERNPG_TABLE
    .org    init_level4_pgt + L4_START_KERNEL*8, 0
    .quad   level3_kernel_pgt - __START_KERNEL_map + _PAGE_TABLE
```

which maps first 2 gigabytes and 512 megabytes for the kernel code, data and bss. `clear_page` function defined in the arch/x86/lib/clear_page_64.S let look on this function:

```
ENTRY(clear_page)
    CFI_STARTPROC
    xorl %eax,%eax
    movl $4096/64,%ecx
```

```
    .p2align 4
    .Lloop:
    decl    %ecx
#define PUT(x) movq %rax,x*8(%rdi)
    movq %rax,(%rdi)
    PUT(1)
    PUT(2)
    PUT(3)
    PUT(4)
    PUT(5)
    PUT(6)
    PUT(7)
    leaq 64(%rdi),%rdi
    jnz    .Lloop
    nop
    ret
    CFI_ENDPROC
    .Lclear_page_end:
    ENDPROC(clear_page)
```

As you can understart from the function name it clears or fills with zeros page tables. First of all note that this function starts with the `CFI_STARTPROC` and `CFI_ENDPROC` which are expands to GNU assembly directives:

```
#define CFI_STARTPROC           .cfi_startproc
#define CFI_ENDPROC             .cfi_endproc
```

and used for debugging. After `CFI_STARTPROC` macro we zero out `eax` register and put 64 to the `ecx` (it will be counter). Next we can see loop which starts with the `.Lloop` label and it starts from the `ecx` decrement. After it we put zero from the `rax` register to the `rdi` which contains the base address of the `init_level4_pgt` now and do the same procedure seven times but every time move `rdi` offset on 8. After this we will have first 64 bytes of the `init_level4_pgt` filled with zeros. In the next step we put the address of the `init_level4_pgt` with 64-bytes offset to the `rdi` again and repeat all operations which `ecx` is not zero. In the end we will have `init_level4_pgt` filled with zeros.

As we have `init_level4_pgt` filled with zeros, we set the last `init_level4_pgt` entry to kernel high mapping with the:

```
init_level4_pgt[511] = early_level4_pgt[511];
```

Remember that we dropped all `early_level4_pgt` entries in the `reset_early_page_table` function and kept only kernel high mapping there.

The last step in the `x86_64_start_kernel` function is the call of the:

```
x86_64_start_reservations(real_mode_data);
```

function with the `real_mode_data` as argument. The `x86_64_start_reservations` function defined in the same source code file as the `x86_64_start_kernel` function and looks:

```
void __init x86_64_start_reservations(char *real_mode_data)
{
    if (!boot_params.hdr.version)
        copy_bootdata(__va(real_mode_data));

    reserve_ebda_region();

    start_kernel();
}
```

You can see that it is the last function before we are in the kernel entry point - `start_kernel` function. Let's look what it does and how it works.

Last step before kernel entry point

First of all we can see in the `x86_64_start_reservations` function check for `boot_params.hdr.version`:

```
if (!boot_params.hdr.version)
    copy_bootdata(__va(real_mode_data));
```

and if it is not we call again `copy_bootdata` function with the virtual address of the `real_mode_data` (read about about it's implementation).

In the next step we can see the call of the `reserve_ebda_region` function which defined in the arch/x86/kernel/head.c. This function reserves memory block for th `EBDA` or Extended BIOS Data Area. The Extended BIOS Data Area located in the top of conventional memory and contains data about ports, disk parameters and etc...

Let's look on the `reserve_ebda_region` function. It starts from the checking is paravirtualization enabled or not:

```
if (paravirt_enabled())
    return;
```

we exit from the `reserve_ebda_region` function if paravirtualization is enabled because if it enabled the extended bios data area is absent. In the next step we need to get the end of the low memory:

```
lowmem = *(unsigned short *)__va(BIOS_LOWMEM_KILOBYTES);
lowmem <<= 10;
```

We're getting the virtual address of the BIOS low memory in kilobytes and convert it to bytes with shifting it on 10 (multiply on 1024 in other words). After this we need to get the address of the extended BIOS data are with the:

```
ebda_addr = get_bios_ebda();
```

where `get_bios_ebda` function defined in the arch/x86/include/asm/bios_ebda.h and looks like:

```
static inline unsigned int get_bios_ebda(void)
{
    unsigned int address = *(unsigned short *)phys_to_virt(0x40E);
    address <<= 4;
    return address;
}
```

Let's try to understand how it works. Here we can see that we converting physical address `0x40E` to the virtual, where `0x0040:0x000e` is the segment which contains base address of the extended BIOS data area. Don't worry that we are using `phys_to_virt` function for converting a physical address to virtual address. You can note that previously we have used `__va` macro for the same point, but `phys_to_virt` is the same:

```
static inline void *phys_to_virt(phys_addr_t address)
{
         return __va(address);
```

```
}
```

only with one difference: we pass argument with the `phys_addr_t` which depends on `CONFIG_PHYS_ADDR_T_64BIT` :

```
#ifdef CONFIG_PHYS_ADDR_T_64BIT
    typedef u64 phys_addr_t;
#else
    typedef u32 phys_addr_t;
#endif
```

This configuration option is enabled by `CONFIG_PHYS_ADDR_T_64BIT` . After that we got virtual address of the segment which stores the base address of the extended BIOS data area, we shift it on 4 and return. After this `ebda_addr` variables contains the base address of the extended BIOS data area.

In the next step we check that address of the extended BIOS data area and low memory is not less than `INSANE_CUTOFF` macro

```
if (ebda_addr < INSANE_CUTOFF)
    ebda_addr = LOWMEM_CAP;

if (lowmem < INSANE_CUTOFF)
    lowmem = LOWMEM_CAP;
```

which is:

```
#define INSANE_CUTOFF        0x20000U
```

or 128 kilobytes. In the last step we get lower part in the low memory and extended bios data area and call `memblock_reserve` function which will reserve memory region for extended bios data between low memory and one megabyte mark:

```
lowmem = min(lowmem, ebda_addr);
lowmem = min(lowmem, LOWMEM_CAP);
memblock_reserve(lowmem, 0x100000 - lowmem);
```

`memblock_reserve` function is defined at mm/block.c and takes two parameters:

- base physical address;
- region size.

and reserves memory region for the given base address and size. `memblock_reserve` is the first function in this book from linux kernel memory manager framework. We will take a closer look on memory manager soon, but now let's look at its implementation.

First touch of the linux kernel memory manager framework

In the previous paragraph we stopped at the call of the `memblock_reserve` function and as i sad before it is the first function from the memory manager framework. Let's try to understand how it works. `memblock_reserve` function just calls:

```
memblock_reserve_region(base, size, MAX_NUMNODES, 0);
```

function and passes 4 parameters there:

- physical base address of the memory region;
- size of the memory region;
- maximum number of numa nodes;
- flags.

At the start of the `memblock_reserve_region` body we can see definition of the `memblock_type` structure:

```
struct memblock_type *_rgn = &memblock.reserved;
```

which presents the type of the memory block and looks:

```
struct memblock_type {
         unsigned long cnt;
         unsigned long max;
         phys_addr_t total_size;
         struct memblock_region *regions;
};
```

As we need to reserve memory block for extended bios data area, the type of the current memory region is reserved where `memblock` structure is:

```
struct memblock {
         bool bottom_up;
         phys_addr_t current_limit;
         struct memblock_type memory;
         struct memblock_type reserved;
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
         struct memblock_type physmem;
#endif
};
```

and describes generic memory block. You can see that we initialize `_rgn` by assigning it to the address of the `memblock.reserved`. `memblock` is the global variable which looks:

```
struct memblock memblock __initdata_memblock = {
    .memory.regions        = memblock_memory_init_regions,
    .memory.cnt        = 1,
    .memory.max        = INIT_MEMBLOCK_REGIONS,
    .reserved.regions    = memblock_reserved_init_regions,
    .reserved.cnt        = 1,
    .reserved.max        = INIT_MEMBLOCK_REGIONS,
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
    .physmem.regions    = memblock_physmem_init_regions,
    .physmem.cnt        = 1,
    .physmem.max        = INIT_PHYSMEM_REGIONS,
#endif
    .bottom_up        = false,
    .current_limit        = MEMBLOCK_ALLOC_ANYWHERE,
};
```

We will not dive into detail of this varaible, but we will see all details about it in the parts about memory manager. Just note that `memblock` variable defined with the `__initdata_memblock` which is:

```
#define __initdata_memblock __meminitdata
```

and `__meminit_data` is:

```
#define __meminitdata    __section(.meminit.data)
```

From this we can conclude that all memory blocks will be in the `.meminit.data` section. After we defined `_rgn` we print information about it with `memblock_dbg` macros. You can enable it by passing `memblock=debug` to the kernel command line.

After debugging lines were printed next is the call of the following function:

```
memblock_add_range(_rgn, base, size, nid, flags);
```

which adds new memory block region into the `.meminit.data` section. As we do not initlieze `_rgn` but it just contains `&memblock.reserved`, we just fill passed `_rgn` with the base address of the extended BIOS data area region, size of this region and flags:

```
if (type->regions[0].size == 0) {
    WARN_ON(type->cnt != 1 || type->total_size);
    type->regions[0].base = base;
    type->regions[0].size = size;
    type->regions[0].flags = flags;
    memblock_set_region_node(&type->regions[0], nid);
    type->total_size = size;
    return 0;
}
```

After we filled our region we can see the call of the `memblock_set_region_node` function with two parameters:

- address of the filled memory region;
- NUMA node id.

where our regions represented by the `memblock_region` structure:

```
struct memblock_region {
    phys_addr_t base;
    phys_addr_t size;
    unsigned long flags;
#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
    int nid;
#endif
};
```

NUMA node id depends on `MAX_NUMNODES` macro which is defined in the include/linux/numa.h:

```
#define MAX_NUMNODES    (1 << NODES_SHIFT)
```

where `NODES_SHIFT` depends on `CONFIG_NODES_SHIFT` configuration parameter and defined as:

```
#ifdef CONFIG_NODES_SHIFT
  #define NODES_SHIFT     CONFIG_NODES_SHIFT
#else
  #define NODES_SHIFT     0
#endif
```

`memblick_set_region_node` function just fills `nid` field from `memblock_region` with the given value:

```
static inline void memblock_set_region_node(struct memblock_region *r, int nid)
{
         r->nid = nid;
}
```

After this we will have first reserved `memblock` for the extended bios data area in the `.meminit.data` section. `reserve_ebda_region` function finished its work on this step and we can go back to the arch/x86/kernel/head64.c.

We finished all preparations before the kernel entry point! The last step in the `x86_64_start_reservations` function is the call of the:

```
start_kernel()
```

function from init/main.c file.

That's all for this part.

Conclusion

It is the end of the third part about linux kernel internals. In next part we will see the first initialization steps in the kernel entry point - `start_kernel` function. It will be the first step before we will see launch of the first `init` process.

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 will find any mistakes please send me PR to linux-internals.

Links

- BIOS data area
- What is in the extended BIOS data area on a PC?
- Previous part

Kernel initialization. Part 4.

Kernel entry point

If you have read the previous part - Last preparations before the kernel entry point, you can remember that we finished all pre-initialization stuff and stopped right before the call of the `start_kernel` function from the init/main.c. The `start_kernel` is the entry of the generic and architecture independent kernel code, although we will return to the `arch/` folder many times. If you will look inside of the `start_kernel` function, you will see that this function is very big. For this moment it contains about `86` calls of functions. Yes, it's very big and of course this part will not cover all processes which are occur in this function. In the current part we will only start to do it. This part and all the next which will be in the Kernel initialization process chapter will cover it.

The main purpose of the `start_kernel` to finish kernel initialization process and launch first `init` process. Before the first process will be started, the `start_kernel` must do many things as: to enable lock validator, to initialize processor id, to enable early cgroups subsystem, to setup per-cpu areas, to initialize different caches in vfs, to initialize memory manager, rcu, vmalloc, scheduler, IRQs, ACPI and many many more. Only after these steps we will see the launch of the first `init` process in the last part of this chapter. So many kernel code waits us, let's start.

NOTE: All parts from this big chapter `Linux Kernel initialization process` will not cover anything about debugging. There will be separate chapter about kernel debugging tips.

A little about function attributes

As I wrote above, the `start_kernel` funcion defined in the init/main.c. This function defined with the `__init` attribute and as you already may know from other parts, all function which are defined with this attributed are necessary during kernel initialization.

```
#define __init      __section(.init.text) __cold notrace
```

After initilization process will be finished, the kernel will release these sections with the call of the `free_initmem` function. Note also that `__init` defined with two attributes: `__cold` and `notrace`. Purpose of the first `cold` attribute is to mark the function that it is rarely used and compiler will optimize this function for size. The second `notrace` is defined as:

```
#define notrace __attribute__((no_instrument_function))
```

where `no_instrument_function` says to compiler to not generate profiling function calls.

In the definition of the `start_kernel` function, you can also see the `__visible` attribute which expands to the:

```
#define __visible __attribute__((externally_visible))
```

where `externally_visible` tells to the compiler that something uses this function or variable, to prevent marking this function/variable as `unusable`. Definition of this and other macro attributes you can find in the include/linux/init.h.

First steps in the start_kernel

At the beginning of the `start_kernel` you can see definition of the two variables:

```
char *command_line;
char *after_dashes;
```

The first presents pointer to the kernel command line and the second will contain result of the `parse_args` function which parses an input string with parameters in the form `name=value`, looking for specific keywords and invoking the right handlers. We will not go into details at this time related with these two variables, but will see it in the next parts. In the next step we can see call of:

```
lockdep_init();
```

function. `lockdep_init` initializes lock validator. It's implementation is pretty easy, it just initializes two list_head hashes and set global variable `lockdep_initialized` to `1`. Lock validator detects circular lock dependecies and called when any spinlock or mutex is acquired.

The next function is `set_task_stack_end_magic` which takes address of the `init_task` and sets `STACK_END_MAGIC` (`0x57AC6E9D`) as canary for it. `init_task` presents initial task structure:

```
struct task_struct init_task = INIT_TASK(init_task);
```

where `task_struct` structure stores all informantion about a process. I will not definition of this structure in this book, because it's very big. You can find its definition in the include/linux/sched.h. For this moment `task_struct` contains more than `100` fields! Although you will not see definition of the `task_struct` in this book, we will use it very often, since it is the fundamental structure which describes the `process` in the Linux kernel. I will describe the meaning of the fields of this structure as we will meet with them in practice.

You can see the definition of the `init_task` and it initialized by `INIT_TASK` macro. This macro is from the include/linux/init_task.h and it just fills the `init_task` with the values for the first process. For example it sets:

- init process state to zero or `runnable`. A runnable process is one which is waiting only for a CPU to run on;
- init process flags - `PF_KTHREAD` which means - kernel thread;
- a list of runnable task;
- process address space;
- init process stack to the `&init_thread_info` which is `init_thread_union.thread_info` and `initthread_union` has type - `thread_union` which contains `thread_info` and process stack:

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

Every process has own stack and it is 16 killobytes or 4 page frames. in `x86_64`. We can note that it defined as array of `unsigned long`. The next field of the `thread_union` is - `thread_info` defined as:

```
struct thread_info {
        struct task_struct      *task;
        struct exec_domain      *exec_domain;
        __u32                   flags;
        __u32                   status;
        __u32                   cpu;
        int                     saved_preempt_count;
```

```
        mm_segment_t            addr_limit;
        struct restart_block    restart_block;
        void __user             *sysenter_return;
        unsigned int            sig_on_uaccess_error:1;
        unsigned int            uaccess_err:1;
};
```

and occupies 52 bytes. `thread_info` structure contains archetecture-specific inforamtion the thread. We know that on `x86_64` stack grows down and `thread_union.thread_info` is stored at the bottom of the stack in our case. So the process stack is 16 killobytes and `thread_info` is at the bottom. Remaining thread_size will be `16 killobytes - 62 bytes = 16332 bytes`. Note that `thread_unioun` represented as the union and not structure, it means that `thread_info` and stack share the memory space.

Schematically it can be represented as follows:

```
+-----------------------+
|                       |
|                       |
|        stack          |
|                       |
|_______________________|
|          |            |
|          |            |
|          |            |
|__________↓____________|             +--------------------+
|                       |             |                    |
|      thread_info      |<----------->|     task_struct    |
|                       |             |                    |
+-----------------------+             +--------------------+
```

http://www.quora.com/In-Linux-kernel-Why-thread_info-structure-and-the-kernel-stack-of-a-process-binds-in-union-construct

So `INIT_TASK` macro fills these `task_struct's` fields and many many more. As i already wrote about, I will not describe all fields and its values in the `INIT_TASK` macro, but we will see it soon.

Now let's back to the `set_task_stack_end_magic` function. This function defined in the kernel/fork.c and sets a canary to the `init` process stack to prevent stack overflow.

```
void set_task_stack_end_magic(struct task_struct *tsk)
{
    unsigned long *stackend;
    stackend = end_of_stack(tsk);
    *stackend = STACK_END_MAGIC; /* for overflow detection */
}
```

Its implementation is easy. `set_task_stack_end_magic` gets the end of the stack for the give `task_struct` with the `end_of_stack` function. End of a process stack depends on `CONFIG_STACK_GROWSUP` configuration option. As we learning `x86_64` architecture, stack grows down. So the end of the process stack will be:

```
(unsigned long *)(task_thread_info(p) + 1);
```

where `task_thread_info` just returns the stack which we filled with the `INIT_TASK` macro:

```
#define task_thread_info(task)  ((struct thread_info *)(task)->stack)
```

As we got end of the init process stack, we write `STACK_END_MAGIC` there. After `canary` set, we can check it like this:

```
if (*end_of_stack(task) != STACK_END_MAGIC) {
        //
        // handle stack overflow here
        //
}
```

The next function after the `set_task_stack_end_magic` is `smp_setup_processor_id`. This function has empty body for `x86_64`:

```
void __init __weak smp_setup_processor_id(void)
{
}
```

as it implemented not for all architectures, but for s390, arm64 and etc...

The next function is - `debug_objects_early_init` in the `start_kernel`. Implementation of these function is almost the same as `lockdep_init`, but fills hashes for object debugging. As i wrote about, we will not see description of this and other functions which are for debugging purposes in this chapter.

After `debug_object_early_init` function we can see the call of the `boot_init_stack_canary` function which fills `task_struct->canary` with the canary value for the `-fstack-protector` gcc feature. This function depends on `CONFIG_CC_STACKPROTECTOR` configuration option and if this option is disabled `boot_init_stack_canary` does not anything, in another way it generate random number based on random pool and the TSC:

```
get_random_bytes(&canary, sizeof(canary));
tsc = __native_read_tsc();
canary += tsc + (tsc << 32UL);
```

After we got a random number, we fill `stack_canary` field of the `task_struct` with it:

```
current->stack_canary = canary;
```

and writes this value to the top of the IRQ stack with the:

```
this_cpu_write(irq_stack_union.stack_canary, canary); // read bellow about this_cpu_write
```

Again, we will not dive into details here, will cover it in the part about IRQs. As canary set, we disable local and early boot IRQs and register the bootstrap cpu in the cpu maps. We disable local irqs (interrupts for current CPU) with the `local_irq_disable` macro which expands to the call of the `arch_local_irq_disable` function from the include/linux/percpu-defs.h:

```
static inline notrace void arch_local_irq_enable(void)
{
        native_irq_enable();
}
```

Where `native_irq_enable` is `cli` instruction for `x86_64`. As interrupts are disabled we can register current cpu with the given ID in the cpu bitmap.

The first processor activation

Current function from the `start_kernel` is the - `boot_cpu_init`. This function initalizes various cpu masks for the boostrap processor. First of all it gets the bootstrap processor id with the call of:

```
int cpu = smp_processor_id();
```

For now it is just zero. If `CONFIG_DEBUG_PREEMPT` configuration option is disabled, `smp_processor_id` just expands to the call of the `raw_smp_processor_id` which expands to the:

```
#define raw_smp_processor_id() (this_cpu_read(cpu_number))
```

`this_cpu_read` as many other function like this (`this_cpu_write`, `this_cpu_add` and etc...) defined in the include/linux/percpu-defs.h and presents `this_cpu` operation. These operations provide a way of opmizing access to the per-cpu variables which are associated with the current processor. In our case it is - `this_cpu_read` expands to the of the:

```
__pcpu_size_call_return(this_cpu_read_, pcp)
```

Remember that we have passed `cpu_number` as `pcp` to the `this_cpu_read` from the `raw_smp_processor_id`. Now let's look on `__pcpu_size_call_return` implementation:

```
#define __pcpu_size_call_return(stem, variable)                         \
({                                                                      \
        typeof(variable) pscr_ret__;                                    \
        __verify_pcpu_ptr(&(variable));                                 \
        switch(sizeof(variable)) {                                      \
        case 1: pscr_ret__ = stem##1(variable); break;                  \
        case 2: pscr_ret__ = stem##2(variable); break;                  \
        case 4: pscr_ret__ = stem##4(variable); break;                  \
        case 8: pscr_ret__ = stem##8(variable); break;                  \
        default:                                                        \
                __bad_size_call_parameter(); break;                     \
        }                                                               \
        pscr_ret__;                                                     \
})
```

Yes, it look a little strange, but it's easy. First of all we can see definition of the `pscr_ret__` variable with the `int` type. Why int? Ok, `variable` is `common_cpu` and it was declared as per-cpu int variable:

```
DECLARE_PER_CPU_READ_MOSTLY(int, cpu_number);
```

In the next step we call `__verify_pcpu_ptr` with the address of `cpu_number`. `__veryf_pcpu_ptr` used to verifying that given parameter is an per-cpu pointer. After that we set `pscr_ret__` value which depends on the size of the variable. Our `common_cpu` variable is `int`, so it 4 bytes size. It means that we will get `this_cpu_read_4(common_cpu)` in `pscr_ret__`. In the end of the `__pcpu_size_call_return` we just call it. `this_cpu_read_4` is a macro:

```
#define this_cpu_read_4(pcp)       percpu_from_op("mov", pcp)
```

which calls `percpu_from_op` and pass `mov` instruction and per-cpu variable there. `percpu_from_op` will expand to the inline assembly call:

```
asm("movl %%gs:%1,%0" : "=r" (pfo_ret__) : "m" (common_cpu))
```

Let's try to understand how it works and what it does. `gs` segment register contains the base of per-cpu area. Here we just copy `common_cpu` which is in memory to the `pfo_ret__` with the `movl` instruction. Or with another words:

```
this_cpu_read(common_cpu)
```

is the same that:

```
movl %gs:$common_cpu, $pfo_ret__
```

As we didn't setup per-cpu area, we have only one - for the current running CPU, we will get `zero` as a result of the `smp_processor_id`.

As we got current processor id, `boot_cpu_init` sets the given cpu online,active,present and possible with the:

```
set_cpu_online(cpu, true);
set_cpu_active(cpu, true);
set_cpu_present(cpu, true);
set_cpu_possible(cpu, true);
```

All of these functions use the concept - `cpumask`. `cpu_possible` is a set of cpu ID's which can be plugged in anytime during the life of that system boot. `cpu_present` represents which CPUs are currently plugged in. `cpu_online` represents subset of the `cpu_present` and indicates CPUs which are available for scheduling. These masks depends on `CONFIG_HOTPLUG_CPU` configuration option and if this option is disabled `possible == present` and `active == online`. Implementation of the all of these functions are very similar. Every function checks the second parameter. If it is `true`, calls `cpumask_set_cpu` or `cpumask_clear_cpu` otherwise.

For example let's look on `set_cpu_possible`. As we passed `true` as the second parameter, the:

```
cpumask_set_cpu(cpu, to_cpumask(cpu_possible_bits));
```

will be called. First of all let's try to understand `to_cpu_mask` macro. This macro casts a bitmap to a `struct cpumask *`. Cpu masks provide a bitmap suitable for representing the set of CPU's in a system, one bit position per CPU number. CPU mask presented by the `cpu_mask` structure:

```
typedef struct cpumask { DECLARE_BITMAP(bits, NR_CPUS); } cpumask_t;
```

which is just bitmap declared with the `DECLARE_BITMAP` macro:

```
#define DECLARE_BITMAP(name, bits) unsigned long name[BITS_TO_LONGS(bits)]
```

As we can see from its definition, `DECLARE_BITMAP` macro expands to the array of `unsigned long`. Now let's look on how `to_cpumask` macro implemented:

```
#define to_cpumask(bitmap)                                              \
        ((struct cpumask *)(1 ? (bitmap)                                \
```

```
                               : (void *)sizeof(__check_is_bitmap(bitmap))))
```

I don't know how about you, but it looked really weird for me at the first time. We can see ternary operator operator here which is `true` every time, but why the `__check_is_bitmap` here? It's simple, let's look on it:

```
static inline int __check_is_bitmap(const unsigned long *bitmap)
{
        return 1;
}
```

Yeah, it just returns `1` every time. Actually we need in it here only for one purpose: In compile time it checks that given `bitmap` is a bitmap, or with another words it checks that given `bitmap` has type - `unsigned long *`. So we just pass `cpu_possible_bits` to the `to_cpumask` macro for converting array of `unsigned long` to the `struct cpumask *`. Now we can call `cpumask_set_cpu` function with the `cpu` - 0 and `struct cpumask *cpu_possible_bits`. This function makes only one call of the `set_bit` function which sets the given `cpu` in the cpumask. All of these `set_cpu_*` functions work on the same principle.

If you're not sure that this `set_cpu_*` operations and `cpumask` are not clear for you, don't worry about it. You can get more info by reading of the special part about it - cpumask or documentation.

As we activated the bootstrap processor, time to go to the next function in the `start_kernel.` Now it is `page_address_init`, but this function does nothing in our case, because it executes only when all `RAM` can't be mapped directly.

Print linux banner

The next call is `pr_notice`:

```
#define pr_notice(fmt, ...) \
    printk(KERN_NOTICE pr_fmt(fmt), ##__VA_ARGS__)
```

as you can see it just expands to the `printk` call. For this moment we use `pr_notice` for printing linux banner:

```
pr_notice("%s", linux_banner);
```

which is just kernel version with some additional parameters:

```
Linux version 4.0.0-rc6+ (alex@localhost) (gcc version 4.9.1 (Ubuntu 4.9.1-16ubuntu6) ) #319 SMP
```

Architecture-dependent parts of initialization

The next step is architecture-specific initializations. Linux kernel does it with the call of the `setup_arch` function. This is very big function as the `start_kernel` and we do not have time to consider all of its implementation in this part. Here we'll only start to do it and continue in the next part. As it is `architecture-specific`, we need to go again to the `arch/` directory. `setup_arch` function defined in the arch/x86/kernel/setup.c source code file and takes only one argument - address of the kernel command line.

This function starts from the reserving memory block for the kernel `_text` and `_data` which starts from the `_text` symbol (you can remember it from the arch/x86/kernel/head_64.S) and ends before `__bss_stop`. We are using `memblock` for the reserving of memory block:

```
memblock_reserve(__pa_symbol(_text), (unsigned long)__bss_stop - (unsigned long)_text);
```

You can read about `memblock` in the Linux kernel memory management Part 1.. As you can remember `memblock_reserve` function takes two parameters:

- base physical address of a memory block;
- size of a memor block.

Base physical address of the `_text` symbol we will get with the `__pa_symbol` macro:

```
#define __pa_symbol(x) \
    __phys_addr_symbol(__phys_reloc_hide((unsigned long)(x)))
```

First of all it calls `__phys_reloc_hide` macro on the given parameter. `__phys_reloc_hide` macro does nothing for `x86_64` and just returns the given parameter. Implementation of the `__phys_addr_symbol` macro is easy. It just subtracts the symbol address from the base address of the kernel text mapping base virtual address (you can remember that it is `__START_KERNEL_map`) and adds `phys_base` which is base address of the `_text` :

```
#define __phys_addr_symbol(x) \
 ((unsigned long)(x) - __START_KERNEL_map + phys_base)
```

After we got physical address of the `_text` symbol, `memblock_reserve` can reserve memory block from the `_text` to the `__bss_stop - _text` .

Reserve memory for initrd

In the next step after we reserved place for the kernel text and data is resering place for the initrd. We will not see details about `initrd` in this post, you just may know that it is temporary root file system stored in memory and used by the kernel during its startup. `early_reserve_initrd` function does all work. First of all this function get the base address of the ram disk, its size and the end address with:

```
u64 ramdisk_image = get_ramdisk_image();
u64 ramdisk_size  = get_ramdisk_size();
u64 ramdisk_end   = PAGE_ALIGN(ramdisk_image + ramdisk_size);
```

All of these parameters it takes from the `boot_params` . If you have read chapter abot Linux Kernel Booting Process, you must remember that we filled `boot_params` structure during boot time. Kerne setup header contains a couple of fields which describes ramdisk, for example:

```
Field name:    ramdisk_image
Type:        write (obligatory)
Offset/size:    0x218/4
Protocol:    2.00+

  The 32-bit linear address of the initial ramdisk or ramfs.  Leave at
  zero if there is no initial ramdisk/ramfs.
```

So we can get all information which interests us from the `boot_params` . For example let's look on `get_ramdisk_image` :

```
static u64 __init get_ramdisk_image(void)
{
        u64 ramdisk_image = boot_params.hdr.ramdisk_image;

        ramdisk_image |= (u64)boot_params.ext_ramdisk_image << 32;

        return ramdisk_image;
}
```

Here we get address of the ramdisk from the `boot_params` and shift left it on `32`. We need to do it because as you can read in the Documentation/x86/zero-page.txt:

```
0C0/004    ALL    ext_ramdisk_image ramdisk_image high 32bits
```

So after shifting it on 32, we're getting 64-bit address in `ramdisk_image`. After we got it just return it. `get_ramdisk_size` works on the same principle as `get_ramdisk_image`, but it used `ext_ramdisk_size` instead of `ext_ramdisk_image`. After we got ramdisk's size, base address and end address, we check that bootloader provided ramdisk with the:

```
if (!boot_params.hdr.type_of_loader ||
    !ramdisk_image || !ramdisk_size)
    return;
```

and reserve memory block with the calculated addresses for the initial ramdisk in the end:

```
memblock_reserve(ramdisk_image, ramdisk_end - ramdisk_image);
```

Conclusion

It is the end of the fourth part about linux kernel initialization process. We started to dive in the kernel generic code from the `start_kernel` function in this part and stopped on the architecture-specific initializations in the `setup_arch`. In next part we will continue with architecture-dependent initialization steps.

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

- GCC function attributes
- this_cpu operations
- cpumask
- lock validator
- cgroups
- stack buffer overflow
- IRQs
- initrd
- Previous part

Kernel initialization. Part 5.

Continue of architecture-specific initializations

In the previous part, we stopped at the initialization of an architecture-specific stuff from the setup_arch function and will continue with it. As we reserved memory for the initrd, next step is the `olpc_ofw_detect` which detects One Laptop Per Child support. We will not consider platform related stuff in this book and will miss functions related with it. So let's go ahead. The next step is the `early_trap_init` function. This function initializes debug (`#DB` - raised when the `TF` flag of rflags is set) and `int3` (`#BP`) interrupts gate. If you don't know anything about interrupts, you can read about it in the Early interrupt and exception handling. In `x86` architecture `INT` , `INTO` and `INT3` are special instructions which allow a task to explicitly call an interrupt handler. The `INT3` instruction calls the breakpoint (`#BP`) handler. You can remember, we already saw it in the part about interrupts: and exceptions:

```
----------------------------------------------------------------------------------------------
|Vector|Mnemonic|Description         |Type |Error Code|Source                                |
----------------------------------------------------------------------------------------------
|3     | #BP    |Breakpoint          |Trap |NO        |INT 3                                 |
----------------------------------------------------------------------------------------------
```

Debug interrupt `#DB` is the primary means of invoking debuggers. `early_trap_init` defined in the arch/x86/kernel/traps.c. This functions sets `#DB` and `#BP` handlers and reloads IDT:

```
void __init early_trap_init(void)
{
        set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
        set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
        load_idt(&idt_descr);
}
```

We already saw implementation of the `set_intr_gate` in the previous part about interrupts. Here are two similar functions `set_intr_gate_ist` and `set_system_intr_gate_ist` . Both of these two functions take two parameters:

- number of the interrupt;
- base address of the interrupt/exception handler;
- third parameter is - `Interrupt Stack Table` . `IST` is a new mechanism in the `x86_64` and part of the TSS. Every active thread in kernel mode has own kernel stack which is 16 killobytes. While a thread in user space, kernel stack is empty except `thread_info` (read about it previous part) at the bottom. In addition to per-thread stacks, there are a couple of specialized stacks associated with each CPU. All about these stack you can read in the linux kernel documentation - Kernel stacks. `x86_64` provides feature which allows to switch to a new `special` stack for during any events as non-maskable interrupt and etc... And the name of this feature is - `Interrupt Stack Table` . There can be up to 7 `IST` entries per CPU and every entry points to the dedicated stack. In our case this is `DEBUG_STACK` .

`set_intr_gate_ist` and `set_system_intr_gate_ist` work by the same principle as `set_intr_gate` with only one difference. Both of these functions checks interrupt number and call `_set_gate` inside:

```
BUG_ON((unsigned)n > 0xFF);
_set_gate(n, GATE_INTERRUPT, addr, 0, ist, __KERNEL_CS);
```

as `set_intr_gate` does this. But `set_intr_gate` calls `_set_gate` with dpl - 0, and ist - 0, but `set_intr_gate_ist` and `set_system_intr_gate_ist` sets `ist` as `DEBUG_STACK` and `set_system_intr_gate_ist` sets `dpl` as `0x3` which is the lowest privilege. When an interrupt occurs and the hardware loads such a descriptor, then hardware automatically sets the new stack pointer based on the IST value, then invokes the interrupt handler. All of the special kernel stacks will be setted in the `cpu_init` function (we will see it later).

As `#DB` and `#BP` gates written to the `idt_descr`, we reload `IDT` table with `load_idt` which just cals `ldtr` instruction. Now let's look on interrupt handlers and will try to understand how they works. Of course, I can't cover all interrupt handlers in this book and I do not see the point in this. It is very interesting to delve in the linux kernel source code, so we will see how `debug` handler implemented in this part, and understand how other interrupt handlers are implemented will be your task.

DB handler

As you can read above, we passed address of the `#DB` handler as `&debug` in the `set_intr_gate_ist`. lxr.free-electorns.com is a great resource for searching identificators in the linux kernel source code, but unfortunately you will not find `debug` handler with it. All of you can find, it is `debug` definition in the arch/x86/include/asm/traps.h:

```
asmlinkage void debug(void);
```

We can see `asmlinkage` attribute which tells to us that `debug` is function written with assembly. Yeah, again and again assembly :). Implementation of the `#DB` handler as other handlers is in this arch/x86/kernel/entry_64.S and defined with the `idtentry` assembly macro:

```
idtentry debug do_debug has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
```

`idtentry` is a macro which defines an interrupt/exception entry point. As you can see it takes five arguments:

- name of the interrupt entry point;
- name of the interrupt handler;
- has interrupt error code or not;
- paranoid - if this parameter = 1, switch to special stack (read above);
- shift_ist - stack to switch during interrupt.

Now let's look on `idtentry` macro implementation. This macro defined in the same assembly file and defines `debug` function with the `ENTRY` macro. For the start `idtentry` macro checks that given parameters are correct in case if need to switch to the special stack. In the next step it checks that give interrupt returns error code. If interrupt does not return error code (in our case `#DB` does not return error code), it calls `INTR_FRAME` or `XCPT_FRAME` if interrupt has error code. Both of these macros `XCPT_FRAME` and `INTR_FRAME` do nothing and need only for the building initial frame state for interrupts. They uses `CFI` directives and used for debugging. More info you can find in the CFI directives. As comment from the arch/x86/kernel/entry_64.S says: `CFI macros are used to generate dwarf2 unwind information for better backtraces. They don't change any code.` so we will ignore them.

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
    /* Sanity check */
    .if \shift_ist != -1 && \paranoid == 0
    .error "using shift_ist requires paranoid=1"
    .endif

    .if \has_error_code
    XCPT_FRAME
    .else
    INTR_FRAME
    .endif
    ...
```

```
    ...
    ...
```

You can remember from the previous part about early interrupts/exceptions handling that after interrupt occurs, current stack will have following format:

```
    +-----------------------+
    |                       |
+40 |         SS            |
+32 |         RSP           |
+24 |        RFLAGS         |
+16 |         CS            |
+8  |         RIP           |
 0  |       Error Code      | <---- rsp
    |                       |
    +-----------------------+
```

The next two macro from the `idtentry` implementation are:

```
    ASM_CLAC
    PARAVIRT_ADJUST_EXCEPTION_FRAME
```

First `ASM_CLAC` macro depends on `CONFIG_X86_SMAP` configuration option and need for security resason, more about it you can read here. The second `PARAVIRT_ADJUST_EXCEPTION_FRAME` macro is for handling handle Xen-type-exceptions (this chapter about kernel initializations and we will not consider virtualization stuff here).

The next piece of code checks is interrupt has error code or not and pushes `$-1` which is `0xffffffffffffffff` on `x86_64` on the stack if not:

```
    .ifeq \has_error_code
    pushq_cfi $-1
    .endif
```

We need to do it as `dummy` error code for stack consistency for all interrupts. In the next step we subscract from the stack pointer `$ORIG_RAX-R15` :

```
    subq $ORIG_RAX-R15, %rsp
```

where `ORIRG_RAX` , `R15` and other macros defined in the arch/x86/include/asm/calling.h and `ORIG_RAX-R15` is 120 bytes. General purpose registers will occupy these 120 bytes because we need to store all registers on the stack during interrupt handling. After we set stack for general purpose registers, the next step is checking that interrupt came from userspace with:

```
testl $3, CS(%rsp)
jnz 1f
```

Here we checks first and second bits in the `cs` . You can remember that `cs` register contains segment selector where first two bits are `RPL` . All privilege levels are integers in the range 0–3, where the lowest number corresponds to the highest privilege. So if interrupt came from the kernel mode we call `save_paranoid` or jump on label `1` if not. In the `save_paranoid` we store all general purpose registers on the stack and switch user `gs` on kernel `gs` if need:

```
    movl $1,%ebx
```

```
    movl $MSR_GS_BASE,%ecx
    rdmsr
    testl %edx,%edx
    js 1f
    SWAPGS
    xorl %ebx,%ebx
1:    ret
```

In the next steps we put `pt_regs` pointer to the `rdi`, save error code in the `rsi` if it is and call interrupt handler which is - `do_debug` in our case from the arch/x86/kernel/traps.c. `do_debug` like other handlers takes two parameters:

- pt_regs - is a structure which presents set of CPU registers which are saved in the process' memory region;
- error code - error code of interrupt.

After interrupt handler finished its work, calls `paranoid_exit` which restores stack, switch on userspace if interrupt came from there and calls `iret`. That's all. Of course it is not all :), but we will see more deeply in the separate chapter about interrupts.

This is general view of the `idtentry` macro for `#DB` interrupt. All interrupts are similar on this implementation and defined with idtentry too. After `early_trap_init` finished its work, the next function is `early_cpu_init`. This function defined in the arch/x86/kernel/cpu/common.c and collects information about a CPU and its vendor.

Early ioremap initialization

The next step is initialization of early `ioremap`. In general there are two ways to comminicate with devices:

- I/O Ports;
- Device memory.

We already saw first method (`outb/inb` instructions) in the part about linux kernel booting process. The second method is to map I/O physical addresses to virtual addresses. When a physical address is accessed by the CPU, it may refer to a portion of physical RAM which can be mapped on memory of the I/O device. So `ioremap` used to map device memory into kernel address space.

As i wrote above next function is the `early_ioremap_init` which re-maps I/O memory to kernel address space so it can access it. We need to initialize early ioremap for early initialization code which needs to temporarily map I/O or memory regions before the normal mapping functions like `ioremap` are available. Implementation of this function is in the arch/x86/mm/ioremap.c. At the start of the `early_ioremap_init` we can see definition of the `pmd` point with `pmd_t` type (which presents page middle directory entry `typedef struct { pmdval_t pmd; } pmd_t;` where `pmdval_t` is `unsigned long`) and make a check that `fixmap` aligned in a correct way:

```
pmd_t *pmd;
BUILD_BUG_ON((fix_to_virt(0) + PAGE_SIZE) & ((1 << PMD_SHIFT) - 1));
```

`fixmap` - is fixed virtual address mappings which extends from `FIXADDR_START` to `FIXADDR_TOP`. Fixed virtual addresses are needed for subsystems that need to know the virtual address at compile time. After the check `early_ioremap_init` makes a call of the `early_ioremap_setup` function from the mm/early_ioremap.c. `early_ioremap_setup` fills `slot_virt` arry of the `unsigned long` with virtual addresses with 512 temporary boot-time fix-mappings:

```
for (i = 0; i < FIX_BTMAPS_SLOTS; i++)
    slot_virt[i] = __fix_to_virt(FIX_BTMAP_BEGIN - NR_FIX_BTMAPS*i);
```

After this we get page middle directory entry for the `FIX_BTMAP_BEGIN` and put to the `pmd` variable, fills with zeros `bm_pte` which is boot time page tables and call `pmd_populate_kernel` function for setting given page table entry in the given page middle directory:

```
pmd = early_ioremap_pmd(fix_to_virt(FIX_BTMAP_BEGIN));
memset(bm_pte, 0, sizeof(bm_pte));
pmd_populate_kernel(&init_mm, pmd, bm_pte);
```

That's all for this. If you feeling missunderstanding, don't worry. There is special part about `ioremap` and `fixmaps` in the Linux Kernel Memory Management. Part 2 chapter.

Obtaining major and minor numbers for the root device

After early `ioremap` was initialized, you can see the following code:

```
ROOT_DEV = old_decode_dev(boot_params.hdr.root_dev);
```

This code obtains major and minor numbers for the root device where `initrd` will be mounted later in the `do_mount_root` function. Major number of the device identifies a driver associated with the device. Minor number referred on the device controlled by driver. Note that `old_decode_dev` takes one parameter from the `boot_params_structure`. As we can read from the x86 linux kernel boot protocol:

```
Field name:    root_dev
Type:         modify (optional)
Offset/size:    0x1fc/2
Protocol:     ALL

  The default root device device number.  The use of this field is
  deprecated, use the "root=" option on the command line instead.
```

Now let's try understand what is it `old_decode_dev`. Actually it just calls `MKDEV` inside which generates `dev_t` from the give major and minor numbers. It's implementation pretty easy:

```
static inline dev_t old_decode_dev(u16 val)
{
         return MKDEV((val >> 8) & 255, val & 255);
}
```

where `dev_t` is a kernel data type to present major/minor number pair. But what's the strange `old_` prefix? For historical reasons, there are two ways of managing the major and minor numbers of a device. In the first way major and minor numbers occupied 2 bytes. You can see it in the previous code: 8 bit for major number and 8 bit for minor number. But there is problem with this way: 256 major numbers and 256 minor numbers are possible. So 16-bit integer was replaced with 32-bit integer where 12 bits reserved for major number and 20 bits for minor. You can see this in the `new_decode_dev` implementation:

```
static inline dev_t new_decode_dev(u32 dev)
{
         unsigned major = (dev & 0xfff00) >> 8;
         unsigned minor = (dev & 0xff) | ((dev >> 12) & 0xfff00);
         return MKDEV(major, minor);
}
```

After calculation we will get `0xfff` or 12 bits for `major` if it is `0xffffffff` and `0xfffff` or 20 bits for `minor`. So in the end of execution of the `old_decode_dev` we will get major and minor numbers for the root device in `ROOT_DEV` .

Memory map setup

The next point is the setup of the memory map with the call of the `setup_memory_map` function. But before this we setup different parameters as information about a screen (current row and column, video page and etc... (you can read about it in the Video mode initialization and transition to protected mode)), Extended display identification data, video mode, bootloader_type and etc...:

```
    screen_info = boot_params.screen_info;
    edid_info = boot_params.edid_info;
    saved_video_mode = boot_params.hdr.vid_mode;
    bootloader_type = boot_params.hdr.type_of_loader;
    if ((bootloader_type >> 4) == 0xe) {
        bootloader_type &= 0xf;
        bootloader_type |= (boot_params.hdr.ext_loader_type+0x10) << 4;
    }
    bootloader_version  = bootloader_type & 0xf;
    bootloader_version |= boot_params.hdr.ext_loader_ver << 4;
```

All of these parameters we got during boot time and stored in the `boot_params` structure. After this we need to setup the end of the I/O memory. As you know the one of the main purposes of the kernel is resource management. And one of the resource is a memory. As we already know there are two ways to communicate with devices are I/O ports and device memory. All information about registered resources available through:

- /proc/ioports - provides a list of currently registered port regions used for input or output communication with a device;
- /proc/iomem - provides current map of the system's memory for each physical device.

At the moment we are interested in `/proc/iomem` :

```
cat /proc/iomem
00000000-00000fff : reserved
00001000-0009d7ff : System RAM
0009d800-0009ffff : reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000cffff : Video ROM
000d0000-000d3fff : PCI Bus 0000:00
000d4000-000d7fff : PCI Bus 0000:00
000d8000-000dbfff : PCI Bus 0000:00
000dc000-000dffff : PCI Bus 0000:00
000e0000-000fffff : reserved
  000e0000-000e3fff : PCI Bus 0000:00
  000e4000-000e7fff : PCI Bus 0000:00
  000f0000-000fffff : System ROM
```

As you can see range of addresses are shown in hexadecimal notation with its owner. Linux kernel provides API for managing any resources in a general way. Global resources (for example PICs or I/O ports) can be divided into subsets - relating to any hardware bus slot. The main structure `resource` :

```
struct resource {
        resource_size_t start;
        resource_size_t end;
        const char *name;
        unsigned long flags;
        struct resource *parent, *sibling, *child;
};
```

presents abstraction for a tree-like subset of system resources. This structure provides range of addresses from `start` to `end` (`resource_size_t` is `phys_addr_t` or `u64` for `x86_64`) which a resource covers, `name` of a resource (you see these names in the `/proc/iomem` output) and `flags` of a resource (All resources flags defined in the include/linux/ioport.h). The last are three pointers to the `resource` structure. These pointers enable a tree-like structure:

```
+-------------+      +-------------+
|             |      |             |
|    parent   |------|    sibling  |
|             |      |             |
+-------------+      +-------------+
       |
       |
+-------------+
|             |
|    child    |
|             |
+-------------+
```

Every subset of resources has root range resources. For `iomem` it is `iomem_resource` which defined as:

```
struct resource iomem_resource = {
        .name   = "PCI mem",
        .start  = 0,
        .end    = -1,
        .flags  = IORESOURCE_MEM,
};
EXPORT_SYMBOL(iomem_resource);
```

TODO EXPORT_SYMBOL

`iomem_resource` defines root addresses range for io memory with `PCI mem` name and `IORESOURCE_MEM` (`0x00000200`) as flags. As i wrote about our current point is setup the end address of the `iomem` . We will do it with:

```
iomem_resource.end = (1ULL << boot_cpu_data.x86_phys_bits) - 1;
```

Here we shift `1` on `boot_cpu_data.x86_phys_bits` . `boot_cpu_data` is `cpuinfo_x86` structure which we filled during execution of the `early_cpu_init` . As you can understand from the name of the `x86_phys_bits` field, it presents maximum bits amount of the maximum physical address in the system. Note also that `iomem_resource` passed to the `EXPORT_SYMBOL` macro. This macro exports the given symbol (`iomem_resource` in our case) for dynamic linking or in another words it makes a symbol accessible to dynamically loaded modules.

As we set the end address of the root `iomem` resource address range, as I wrote about the next step will be setup of the memory map. It will be produced with the call of the `setup_memory_map` function:

```
void __init setup_memory_map(void)
{
        char *who;

        who = x86_init.resources.memory_setup();
        memcpy(&e820_saved, &e820, sizeof(struct e820map));
        printk(KERN_INFO "e820: BIOS-provided physical RAM map:\n");
        e820_print_map(who);
}
```

First of all we call look here the call of the `x86_init.resources.memory_setup` . `x86_init` is a `x86_init_ops` structure which presents platform specific setup functions as resources initializtion, pci initialization and etc... Initiaization of the `x86_init` is in the arch/x86/kernel/x86_init.c. I will not give here the full description because it is very long, but only one part which interests us for now:

```
struct x86_init_ops x86_init __initdata = {
    .resources = {
            .probe_roms             = probe_roms,
            .reserve_resources      = reserve_standard_io_resources,
            .memory_setup           = default_machine_specific_memory_setup,
    },
    ...
    ...
    ...
}
```

As we can see here `memry_setup` field is `default_machine_specific_memory_setup` where we get the number of the e820 entries which we collected in the boot time, sanitize the BIOS e820 map and fill `e820map` structure with the memory regions. As all regions collect, print of all regions with printk. You can find this print if you execute `dmesg` command, you must see something like this:

```
[    0.000000] e820: BIOS-provided physical RAM map:
[    0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009d7ff] usable
[    0.000000] BIOS-e820: [mem 0x000000000009d800-0x000000000009ffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000000e0000-0x00000000000fffff] reserved
[    0.000000] BIOS-e820: [mem 0x0000000000100000-0x00000000be825fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000be826000-0x00000000be82cfff] ACPI NVS
[    0.000000] BIOS-e820: [mem 0x00000000be82d000-0x00000000bf744fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000bf745000-0x00000000bfff4fff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000bfff5000-0x00000000dc041fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000dc042000-0x00000000dc0d2fff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000dc0d3000-0x00000000dc138fff] usable
[    0.000000] BIOS-e820: [mem 0x00000000dc139000-0x00000000dc27dfff] ACPI NVS
[    0.000000] BIOS-e820: [mem 0x00000000dc27e000-0x00000000deffefff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000defff000-0x00000000deffffff] usable
...
...
...
```

Copying of the BIOS Enhanced Disk Device information

The next two steps is parsing of the `setup_data` with `parse_setup_data` function and copying BIOS EDD to the safe place. `setup_data` is a field from the kernel boot header and as we can read from the `x86` boot protocol:

```
Field name:    setup_data
Type:         write (special)
Offset/size:    0x250/8
Protocol:    2.09+

  The 64-bit physical pointer to NULL terminated single linked list of
  struct setup_data. This is used to define a more extensible boot
  parameters passing mechanism.
```

It used for storing setup information for different types as device tree blob, EFI setup data and etc... In the second step we copy BIOS EDD informantion from the `boot_params` structure that we collected in the arch/x86/boot/edd.c to the `edd` structure:

```
static inline void __init copy_edd(void)
{
     memcpy(edd.mbr_signature, boot_params.edd_mbr_sig_buffer,
            sizeof(edd.mbr_signature));
     memcpy(edd.edd_info, boot_params.eddbuf, sizeof(edd.edd_info));
     edd.mbr_signature_nr = boot_params.edd_mbr_sig_buf_entries;
     edd.edd_info_nr = boot_params.eddbuf_entries;
}
```

Memory descriptor initialization

The next step is initialization of the memory descriptor of the init process. As you already can know every process has own address space. This address space presented with special data structure which called `memory descriptor`. Directly in the linux kernel source code memory descriptor presented with `mm_struct` structure. `mm_struct` contains many different fields related with the process address space as start/end address of the kernel code/data, start/end of the brk, number of memory areas, list of memory areas and etc... This structure defined in the include/linux/mm_types.h. As every process has own memory descriptor, `task_struct` structure contains it in the `mm` and `active_mm` field. And our first `init` process has it too. You can remember that we saw the part of initialization of the init `task_struct` with `INIT_TASK` macro in the previous part:

```
#define INIT_TASK(tsk)  \
{
    ...
    ...
    ...
    .mm = NULL,         \
    .active_mm  = &init_mm, \
    ...
}
```

`mm` points to the process address space and `active_mm` points to the active address space if process has no own as kernel threads (more about it you can read in the documentation). Now we fill memory descriptor of the initial process:

```
    init_mm.start_code = (unsigned long) _text;
    init_mm.end_code = (unsigned long) _etext;
    init_mm.end_data = (unsigned long) _edata;
    init_mm.brk = _brk_end;
```

with the kernel's text, data and brk. `init_mm` is memory descriptor of the initial process and defined as:

```
struct mm_struct init_mm = {
    .mm_rb          = RB_ROOT,
    .pgd            = swapper_pg_dir,
    .mm_users       = ATOMIC_INIT(2),
    .mm_count       = ATOMIC_INIT(1),
    .mmap_sem       = __RWSEM_INITIALIZER(init_mm.mmap_sem),
    .page_table_lock =  __SPIN_LOCK_UNLOCKED(init_mm.page_table_lock),
    .mmlist         = LIST_HEAD_INIT(init_mm.mmlist),
    INIT_MM_CONTEXT(init_mm)
};
```

where `mm_rb` is a red-black tree of the virtual memory areas, `pgd` is a pointer to the page global directory, `mm_users` is address space users, `mm_count` is primary usage counter and `mmap_sem` is memory area semaphore. After that we setup memory descriptor of the initiali process, next step is initialization of the intel Memory Protection Extensions with `mpx_mm_init`. The next step after it is initialization of the code/data/bss resources with:

```
    code_resource.start = __pa_symbol(_text);
    code_resource.end = __pa_symbol(_etext)-1;
    data_resource.start = __pa_symbol(_etext);
    data_resource.end = __pa_symbol(_edata)-1;
    bss_resource.start = __pa_symbol(__bss_start);
    bss_resource.end = __pa_symbol(__bss_stop)-1;
```

We already know a little about `resource` structure (read above). Here we fills code/data/bss resources with the physical addresses of they. You can see it in the `/proc/iomem` output:

```
00100000-be825fff : System RAM
  01000000-015bb392 : Kernel code
  015bb393-01930c3f : Kernel data
  01a11000-01ac3fff : Kernel bss
```

All of these structures defined in the arch/x86/kernel/setup.c and look like typical resource initialization:

```
static struct resource code_resource = {
    .name    = "Kernel code",
    .start    = 0,
    .end    = 0,
    .flags    = IORESOURCE_BUSY | IORESOURCE_MEM
};
```

The last step which we will cover in this part will be `NX` configuration. `NX-bit` or no execute bit is 63-bit in the page directory entry which controls the ability to execute code from all physical pages mapped by the table entry. This bit can only be used/set when the `no-execute` page-protection mechanism is enabled by the setting `EFER.NXE` to 1. In the `x86_configure_nx` function we check that CPU has support of `NX-bit` and it does not disabled. After the check we fill `__supported_pte_mask` depend on it:

```
void x86_configure_nx(void)
{
        if (cpu_has_nx && !disable_nx)
                __supported_pte_mask |= _PAGE_NX;
        else
                __supported_pte_mask &= ~_PAGE_NX;
}
```

Conclusion

It is the end of the fifth part about linux kernel initialization process. In this part we continued to dive in the `setup_arch` function which makes initialization of architecutre-specific stuff. It was long part, but we not finished with it. As i already wrote, the `setup_arch` is big function, and I am really not sure that we will cover full of it even in the next part. There were some new interesting concepts in this part like `Fix-mapped` addresses, ioremap and etc... Don't worry if they are unclear for you. There is special part about these concepts - Linux kernel memory management Part 2.. In the next part we will continue with the initialization of the architecture-specific stuff and will see parsing of the early kernel parameteres, early dump of the pci devices, direct Media Interface scanning and many many more.

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

- mm vs active_mm
- e820
- Supervisor mode access prevention
- Kernel stacks
- TSS
- IDT
- Memory mapped I/O
- CFI directives

* PDF. dwarf4 specification
* Call stack
* Previous part

Kernel initialization. Part 6.

Architecture-specific initializations, again...

In the previous part we saw architecture-specific (`x86_64` in our case) initialization stuff from the arch/x86/kernel/setup.c and finished on `x86_configure_nx` function which sets the `_PAGE_NX` flag depends on support of NX bit. As I wrote before `setup_arch` function and `start_kernel` are very big, so in this and in the next part we will continue to learn about architecture-specific initialization process. The next function after `x86_configure_nx` is `parse_early_param`. This function defined in the init/main.c and as you can understand from its name, this function parses kernel command line and setups different some services depends on give parameters (all kernel command line parameters you can find in the Documentation/kernel-parameters.txt). You can remember how we setup `earlyprintk` in the earliest part. On the early stage we looked for kernel parameters and their value with the `cmdline_find_option` function and `__cmdline_find_option`, `__cmdline_find_option_bool` helpers from the arch/x86/boot/cmdline.c. There we're in the generic kernel part which does not depend on architecture and here we use another approach. If you are reading linux kernel source code, you already can note calls like this:

```
early_param("gbpages", parse_direct_gbpages_on);
```

`early_param` macro takes two parameters:

- command line parameter name;
- function which will be called if given parameter passed.

and defined as:

```
#define early_param(str, fn) \
        __setup_param(str, fn, fn, 1)
```

in the include/linux/init.h. As you can see `early_param` macro just makes call of the `__setup_param` macro:

```
#define __setup_param(str, unique_id, fn, early)                \
        static const char __setup_str_##unique_id[] __initconst \
                __aligned(1) = str; \
        static struct obs_kernel_param __setup_##unique_id      \
                __used __section(.init.setup)                   \
                __attribute__((aligned((sizeof(long)))))        \
                = { __setup_str_##unique_id, fn, early }
```

This macro defines `__setup_str_*_id` variable (where `*` depends on given function name) and assigns it to the given command line parameter name. In the next line we can see definition of the `__setup_*` variable which type is `obs_kernel_param` and its initialization. `obs_kernel_param` structure defined as:

```
struct obs_kernel_param {
        const char *str;
        int (*setup_func)(char *);
        int early;
};
```

and contains three fields:

- name of the kernel parameter;
- function which setups something depend on parameter;
- field determinies is parameter early (1) or not (0).

Note that `__set_param` macro defines with `__section(.init.setup)` attribute. It means that all `__setup_str_*` will be placed in the `.init.setup` section, moreover, as we can see in the include/asm-generic/vmlinux.lds.h, they will be placed between `__setup_start` and `__setup_end`:

```
#define INIT_SETUP(initsetup_align)                \
                . = ALIGN(initsetup_align);        \
                VMLINUX_SYMBOL(__setup_start) = .; \
                *(.init.setup)                     \
                VMLINUX_SYMBOL(__setup_end) = .;
```

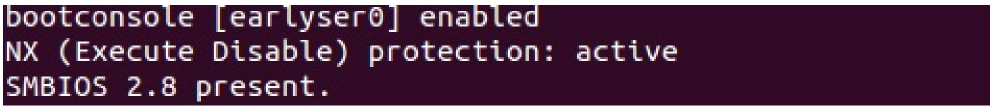
Now we know how parameters are defined, let's back to the `parse_early_param` implementation:

```
void __init parse_early_param(void)
{
        static int done __initdata;
        static char tmp_cmdline[COMMAND_LINE_SIZE] __initdata;

        if (done)
                return;

        /* All fall through to do_early_param. */
        strlcpy(tmp_cmdline, boot_command_line, COMMAND_LINE_SIZE);
        parse_early_options(tmp_cmdline);
        done = 1;
}
```

The `parse_early_param` function defines two static variables. First `done` check that `parse_early_param` already called and the second is temporary storage for kernel command line. After this we copy `boot_command_line` to the temporary commad line which we just defined and call the `parse_early_options` function from the the same source code `main.c` file. `parse_early_options` calls the `parse_args` function from the kernel/params.c where `parse_args` parses given command line and calls `do_early_param` function. This function goes from the `__setup_start` to `__setup_end`, and calls the function from the `obs_kernel_param` if a parameter is early. After this all services which are depend on early command line parameters were setup and the next call after the `parse_early_param` is `x86_report_nx`. As I wrote in the beginning of this part, we already set `NX-bit` with the `x86_configure_nx`. The next `x86_report_nx` function the arch/x86/mm/setup_nx.c just prints information about the `NX`. Note that we call `x86_report_nx` not right after the `x86_configure_nx`, but after the call of the `parse_early_param`. The answer is simple: we call it after the `parse_early_param` because the kernel support `noexec` parameter:

```
noexec          [X86]
            On X86-32 available only on PAE configured kernels.
            noexec=on: enable non-executable mappings (default)
            noexec=off: disable non-executable mappings
```

We can see it in the booting time:



After this we can see call of the:

```
    memblock_x86_reserve_range_setup_data();
```

function. This function defined in the same arch/x86/kernel/setup.c source code file and remaps memory for the `setup_data` and reserved memory block for the `setup_data` (more about `setup_data` you can read in the previous part and about `ioremap` and `memblock` you can read in the Linux kernel memory management).

In the next step we can see following conditional statement:

```
    if (acpi_mps_check()) {
#ifdef CONFIG_X86_LOCAL_APIC
        disable_apic = 1;
#endif
        setup_clear_cpu_cap(X86_FEATURE_APIC);
    }
```

The first `acpi_mps_check` function from the arch/x86/kernel/acpi/boot.c depends on `CONFIG_X86_LOCAL_APIC` and `CNOFIG_x86_MPPARSE` configuration options:

```
int __init acpi_mps_check(void)
{
#if defined(CONFIG_X86_LOCAL_APIC) && !defined(CONFIG_X86_MPPARSE)
        /* mptable code is not built-in*/
        if (acpi_disabled || acpi_noirq) {
                printk(KERN_WARNING "MPS support code is not built-in.\n"
                       "Using acpi=off or acpi=noirq or pci=noacpi "
                       "may have problem\n");
                 return 1;
        }
#endif
        return 0;
}
```

It checks the built-in `MPS` or MultiProcessor Specification table. If `CONFIG_X86_LOCAL_APIC` is set and `CONFIG_x86_MPPAARSE` is not set, `acpi_mps_check` prints warning message if the one of the command line options: `acpi=off`, `acpi=noirq` or `pci=noacpi` passed to the kernel. If `acpi_mps_check` returns `1` which means that

we disable local APIC and clears `X86_FEATURE_APIC` bit in the of the current CPU with the `setup_clear_cpu_cap` macro. (more about CPU mask you can read in the CPU masks).

Early PCI dump

In the next step we make a dump of the PCI devices with the following code:

```
#ifdef CONFIG_PCI
    if (pci_early_dump_regs)
        early_dump_pci_devices();
#endif
```

`pci_early_dump_regs` variable defined in the arch/x86/pci/common.c and its value depends on the kernel command line parameter: `pci=earlydump`. We can find defition of this parameter in the drivers/pci/pci.c:

```
early_param("pci", pci_setup);
```

`pci_setup` function gets the string after the `pci=` and analyzes it. This function calls `pcibios_setup` which defined as `__weak` in the drivers/pci/pci.c and every architecture defines the same function which overrides `__weak` analog. For example `x86_64` architecture-depened version is in the arch/x86/pci/common.c:

```
char *__init pcibios_setup(char *str) {
        ...
        ...
        ...
        } else if (!strcmp(str, "earlydump")) {
                pci_early_dump_regs = 1;
                return NULL;
        }
        ...
        ...
        ...
}
```

So, if `CONFIG_PCI` option is set and we passed `pci=earlydump` option to the kernel command line, next function which will be called - `early_dump_pci_devices` from the arch/x86/pci/early.c. This function checks `noearly` pci parameter with:

```
if (!early_pci_allowed())
        return;
```

and returns if it was passed. Each PCI domain can host up to `256` buses and each bus hosts up to 32 devices. So, we goes in a loop:

```
for (bus = 0; bus < 256; bus++) {
                for (slot = 0; slot < 32; slot++) {
                        for (func = 0; func < 8; func++) {
                        ...
                        ...
                        ...
                        }
                }
}
```

and read the `pci` config with the `read_pci_config` function.

That's all. We will no go deep in the `pci` details, but will see more details in the special `Drivers/PCI` part.

Finish with memory parsing

After the `early_dump_pci_devices`, there are a couple of function related with available memory and e820 which we collected in the First steps in the kernel setup part:

```
    /* update the e820_saved too */
    e820_reserve_setup_data();
    finish_e820_parsing();
    ...
    ...
    ...
    e820_add_kernel_range();
    trim_bios_range(void);
    max_pfn = e820_end_of_ram_pfn();
    early_reserve_e820_mpc_new();
```

Let's look on it. As you can see the first function is `e820_reserve_setup_data`. This function does almost the same as `memblock_x86_reserve_range_setup_data` which we saw above, but it also calls `e820_update_range` which adds new regions to the `e820map` with the given type which is `E820_RESERVED_KERN` in our case. The next function is `finish_e820_parsing` which sanitazes `e820map` with the `sanitize_e820_map` function. Besides this two functions we can see a couple of functions related to the e820. You can see it in the listing which is above. `e820_add_kernel_range` function takes the physical address of the kernel start and end:

```
u64 start = __pa_symbol(_text);
u64 size = __pa_symbol(_end) - start;
```

checks that `.text` `.data` and `.bss` marked as `E820RAM` in the `e820map` and prints the warning message if not. The next function `trm_bios_range` update first 4096 bytes in `e820Map` as `E820_RESERVED` and sanitizes it again with the call of the `sanitize_e820_map`. After this we get the last page frame number with the call of the `e820_end_of_ram_pfn` function. Every memory page has an unique number - `Page frame number` and `e820_end_of_ram_pfn` function returns the maximum with the call of the `e820_end_pfn`:

```
unsigned long __init e820_end_of_ram_pfn(void)
{
    return e820_end_pfn(MAX_ARCH_PFN);
}
```

where `e820_end_pfn` takes maximum page frame number on the certain architecture (`MAX_ARCH_PFN` is `0x400000000` for `x86_64`). In the `e820_end_pfn` we go through the all `e820` slots and check that `e820` entry has `E820_RAM` or `E820_PRAM` type because we calcluate page frame numbers only for these types, gets the base address and end address of the page frame number for the current `e820` entry and makes some checks for these addresses:

```
for (i = 0; i < e820.nr_map; i++) {
        struct e820entry *ei = &e820.map[i];
        unsigned long start_pfn;
        unsigned long end_pfn;

        if (ei->type != E820_RAM && ei->type != E820_PRAM)
            continue;

        start_pfn = ei->addr >> PAGE_SHIFT;
        end_pfn = (ei->addr + ei->size) >> PAGE_SHIFT;

        if (start_pfn >= limit_pfn)
            continue;
        if (end_pfn > limit_pfn) {
            last_pfn = limit_pfn;
            break;
        }
        if (end_pfn > last_pfn)
            last_pfn = end_pfn;
}
```

```
    if (last_pfn > max_arch_pfn)
        last_pfn = max_arch_pfn;

    printk(KERN_INFO "e820: last_pfn = %#lx max_arch_pfn = %#lx\n",
             last_pfn, max_arch_pfn);
    return last_pfn;
```

After this we check that `last_pfn` which we got in the loop is not greater that maximum page frame number for the certain architecture (`x86_64` in our case), print inofmration about last page frame number and return it. We can see the `last_pfn` in the `dmesg` output:

```
...
[    0.000000] e820: last_pfn = 0x41f000 max_arch_pfn = 0x400000000
...
```

After this, as we have calculated the biggest page frame number, we calculate `max_low_pfn` which is the biggest page frame number in the `low memory` or bellow first `4` gigabytes. If installed more than 4 gigabytes of RAM, `max_low_pfn` will be result of the `e820_end_of_low_ram_pfn` function which does the same `e820_end_of_ram_pfn` but with 4 gigabytes limit, in other way `max_low_pfn` will be the same as `max_pfn` :

```
if (max_pfn > (1UL<<(32 - PAGE_SHIFT)))
    max_low_pfn = e820_end_of_low_ram_pfn();
else
    max_low_pfn = max_pfn;

high_memory = (void *)__va(max_pfn * PAGE_SIZE - 1) + 1;
```

Next we calculate `high_memory` (defines the upper bound on direct map memory) with `__va` macro which returns a virtual address by the given physical.

DMI scanning

The next step after manipulations with different memory regions and `e820` slots is collecting information about computer. We will get all information with the Desktop Management Interface and following functions:

```
dmi_scan_machine();
dmi_memdev_walk();
```

First is `dmi_scan_machine` defined in the drivers/firmware/dmi_scan.c. This function goes through the System Management BIOS structures and extracts informantion. There are two ways specified to gain access to the `SMBIOS` table: get the pointer to the `SMBIOS` table from the EFI's configuration table and scanning the physycal memory between `0xF0000` and `0x10000` addresses. Let's look on the second approach. `dmi_scan_machine` function remaps memory between `0xf0000` and `0x10000` with the `dmi_early_remap` which just expands to the `early_ioremap` :

```
void __init dmi_scan_machine(void)
{
    char __iomem *p, *q;
    char buf[32];
    ...
    ...
    ...
    p = dmi_early_remap(0xF0000, 0x10000);
    if (p == NULL)
            goto error;
```

and iterates over all `DMI` header address and find search `_SM_` string:

```
memset(buf, 0, 16);
for (q = p; q < p + 0x10000; q += 16) {
        memcpy_fromio(buf + 16, q, 16);
        if (!dmi_smbios3_present(buf) || !dmi_present(buf)) {
            dmi_available = 1;
            dmi_early_unmap(p, 0x10000);
            goto out;
        }
        memcpy(buf, buf + 16, 16);
}
```

`_SM_` string must be between `000F0000h` and `0x000FFFFF`. Here we copy 16 bytes to the `buf` with `memcpy_fromio` which is the same `memcpy` and execute `dmi_smbios3_present` and `dmi_present` on the buffer. These functions check that first 4 bytes is `_SM_` string, get `SMBIOS` version and gets `_DMI_` attributes as `DMI` structure table length, table address and etc... After one of these function will finish to execute, you will see the result of it in the `dmesg` output:

```
[    0.000000] SMBIOS 2.7 present.
[    0.000000] DMI: Gigabyte Technology Co., Ltd. Z97X-UD5H-BK/Z97X-UD5H-BK, BIOS F6 06/17/2014
```

In the end of the `dmi_scan_machine`, we unmap the previously remaped memory:

```
dmi_early_unmap(p, 0x10000);
```

The second function is - `dmi_memdev_walk`. As you can understand it goes over memory devices. Let's look on it:

```
void __init dmi_memdev_walk(void)
{
    if (!dmi_available)
        return;

    if (dmi_walk_early(count_mem_devices) == 0 && dmi_memdev_nr) {
        dmi_memdev = dmi_alloc(sizeof(*dmi_memdev) * dmi_memdev_nr);
        if (dmi_memdev)
            dmi_walk_early(save_mem_devices);
    }
}
```

It checks that `DMI` available (we got it in the previous function - `dmi_scan_machine`) and collects information about memory devices with `dmi_walk_early` and `dmi_alloc` which defined as:

```
#ifdef CONFIG_DMI
RESERVE_BRK(dmi_alloc, 65536);
#endif
```

`RESERVE_BRK` defined in the arch/x86/include/asm/setup.h and reserves space with given size in the `brk` section.

```
init_hypervisor_platform();
x86_init.resources.probe_roms();
insert_resource(&iomem_resource, &code_resource);
insert_resource(&iomem_resource, &data_resource);
insert_resource(&iomem_resource, &bss_resource);
early_gart_iommu_check();
```

SMP config

The next step is parsing of the SMP configuration. We do it with the call of the `find_smp_config` function which just calls function:

```
static inline void find_smp_config(void)
{
        x86_init.mpparse.find_smp_config();
}
```

inside. `x86_init.mpparse.find_smp_config` is a `default_find_smp_config` function from the arch/x86/kernel/mpparse.c. In the `default_find_smp_config` function we are scanning a couple of memory regions for `SMP` config and return if they are not:

```
if (smp_scan_config(0x0, 0x400) ||
            smp_scan_config(639 * 0x400, 0x400) ||
            smp_scan_config(0xF0000, 0x10000))
            return;
```

First of all `smp_scan_config` function defines a couple of variables:

```
unsigned int *bp = phys_to_virt(base);
struct mpf_intel *mpf;
```

First is virtual address of the memory region where we will scan `SMP` config, second is the pointer to the `mpf_intel` structure. Let's try to understand what is it `mpf_intel`. All information stores in the multiprocessor configuration data structure. `mpf_intel` presents this structure and looks:

```
struct mpf_intel {
        char signature[4];
        unsigned int physptr;
        unsigned char length;
        unsigned char specification;
        unsigned char checksum;
        unsigned char feature1;
        unsigned char feature2;
        unsigned char feature3;
        unsigned char feature4;
        unsigned char feature5;
};
```

As we can read in the documentation - one of the main functions of the system BIOS is to construct the MP floating pointer structure and the MP configuration table. And operating system must have access to this information about the multiprocessor configuration and `mpf_intel` stores the physical address (look at second parameter) of the multiprocessor configuration table. So, `smp_scan_config` going in a loop through the given memory range and tries to find `MP floating pointer structure` there. It checks that current byte points to the `SMP` signature, checks checksum, checks that `mpf->specification` is 1 (it must be `1` or `4` by specification) in the loop:

```
while (length > 0) {
if ((*bp == SMP_MAGIC_IDENT) &&
    (mpf->length == 1) &&
    !mpf_checksum((unsigned char *)bp, 16) &&
    ((mpf->specification == 1)
    || (mpf->specification == 4))) {

        mem = virt_to_phys(mpf);
        memblock_reserve(mem, sizeof(*mpf));
        if (mpf->physptr)
            smp_reserve_memory(mpf);
    }
}
```

reserves given memory block if search is successful with `memblock_reserve` and reserves physical address of the multiprocessor configuration table. All documentation about this you can find in the - MultiProcessor Specification. More details you can read in the special part about `SMP`.

Additional early memory initialization routines

In the next step of the `setup_arch` we can see the call of the `early_alloc_pgt_buf` function which allocates the page table buffer for early stage. The page table buffer will be place in the `brk` area. Let's look on its implementation:

```C
void  __init early_alloc_pgt_buf(void)
{
        unsigned long tables = INIT_PGT_BUF_SIZE;
        phys_addr_t base;

        base = __pa(extend_brk(tables, PAGE_SIZE));

        pgt_buf_start = base >> PAGE_SHIFT;
        pgt_buf_end = pgt_buf_start;
        pgt_buf_top = pgt_buf_start + (tables >> PAGE_SHIFT);
}
```

First of all it get the size of the page table buffer, it will be `INIT_PGT_BUF_SIZE` which is `(6 * PAGE_SIZE)` in the current linux kernel 4.0. As we got the size of the page table buffer, we call `extend_brk` function with two parameters: size and align. As you can understand from its name, this function extends the `brk` area. As we can see in the linux kernel linker script `brk` in memory right after the BSS:

```
    . = ALIGN(PAGE_SIZE);
    .brk : AT(ADDR(.brk) - LOAD_OFFSET) {
        __brk_base = .;
        . += 64 * 1024;        /* 64k alignment slop space */
        *(.brk_reservation)    /* areas brk users have reserved */
        __brk_limit = .;
    }
```

Or we can find it with `readelf` util:

```
  [25] .bss              NOBITS           ffffffff8199d000  00d9d000
       00000000000b4000  0000000000000000  WA       0     0     4096
  [26] .brk              NOBITS           ffffffff81a51000  00d9d000
       0000000000026000  0000000000000000  WA       0     0     1
```

After that we got physical address of the new `brk` with the `__pa` macro, we calculate the base address and the end of the page table buffer. In the next step as we got page table buffer, we reserve memory block for the brk are with the `reserve_brk` function:

```C
static void __init reserve_brk(void)
{
    if (_brk_end > _brk_start)
        memblock_reserve(__pa_symbol(_brk_start),
                 _brk_end - _brk_start);

    _brk_start = 0;
}
```

Note that in the end of the `reserve_brk`, we set `brk_start` to zero, because after this we will not allocate it anymore. The next step after reserving memory block for the `brk`, we need to unmap out-of-range memory areas in the kernel mapping with the `cleanup_highmap` function. Remeber that kernel mapping is `__START_KERNEL_map` and `_end - _text` or `level2_kernel_pgt` maps the kernel `_text`, `data` and `bss`. In the start of the `clean_high_map` we define these parameters:

```C
unsigned long vaddr = __START_KERNEL_map;
unsigned long end = roundup((unsigned long)_end, PMD_SIZE) - 1;
pmd_t *pmd = level2_kernel_pgt;
pmd_t *last_pmd = pmd + PTRS_PER_PMD;
```

Now, as we defined start and end of the kernel mapping, we go in the loop through the all kernel page middle directory entries and clean entries which are not between `_text` and `end`:

```
for (; pmd < last_pmd; pmd++, vaddr += PMD_SIZE) {
        if (pmd_none(*pmd))
            continue;
        if (vaddr < (unsigned long) _text || vaddr > end)
            set_pmd(pmd, __pmd(0));
}
```

After this we set the limit for the `memblock` allocation with the `memblock_set_current_limit` function (read more about `memblock` you can in the Linux kernel memory management Part 2), it will be `ISA_END_ADDRESS` or `0x100000` and fill the `memblock` information according to `e820` with the call of the `memblock_x86_fill` function. You can see the result of this function in the kernel initialization time:

```
MEMBLOCK configuration:
 memory size = 0x1fff7ec00 reserved size = 0x1e30000
 memory.cnt  = 0x3
 memory[0x0]    [0x00000000001000-0x0000000009efff], 0x9e000 bytes flags: 0x0
 memory[0x1]    [0x00000000100000-0x000000bffdffff], 0xbfee0000 bytes flags: 0x0
 memory[0x2]    [0x00000100000000-0x0000023fffffff], 0x140000000 bytes flags: 0x0
 reserved.cnt  = 0x3
 reserved[0x0]    [0x0000000009f000-0x000000000fffff], 0x61000 bytes flags: 0x0
 reserved[0x1]    [0x00000001000000-0x00000001a57fff], 0xa58000 bytes flags: 0x0
 reserved[0x2]    [0x0000007ec89000-0x0000007fffffff], 0x1377000 bytes flags: 0x0
```

The rest functions after the `memblock_x86_fill` are: `early_reserve_e820_mpc_new` alocates additional slots in the `e820map` for MultiProcessor Specification table, `reserve_real_mode` - reserves low memory from `0x0` to 1 megabyte for the trampoline to the real mode (for rebootin and etc...), `trim_platform_memory_ranges` - trims certain memory regions started from `0x20050000` , `0x20110000` and etc... these regions must be excluded because Sandy Bridge has problems with these regions, `trim_low_memory_range` reserves the first 4 killobytes page in `memblock` , `init_mem_mapping` function reconstructs direct memory mapping and setups the direct mapping of the physical memory at `PAGE_OFFSET` , `early_trap_pf_init` setups `#PF` handler (we will look on it in the chapter about interrupts) and `setup_real_mode` function setups trampoline to the real mode code.

That's all. You can note that this part will not cover all functions which are in the `setup_arch` (like `early_gart_iommu_check` , mtrr initalization and etc...). As I already wrote many times, `setup_arch` is big, and linux kernel is big. That's why I can't cover every line in the linux kernel. I don't think that we missed something important,... but you can say something like: each line of code is important. Yes, it's true, but I missed they anyway, because I think that it is not real to cover full linux kernel. Anyway we will often return to the idea that we have already seen, and if something will be unfamiliar, we will cover this theme.

Conclusion

It is the end of the sixth part about linux kernel initialization process. In this part we continued to dive in the `setup_arch` function again It was long part, but we not finished with it. Yes, `setup_arch` is big, hope that next part will be last about this function.

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

- MultiProcessor Specification
- NX bit

- Documentation/kernel-parameters.txt
- APIC
- CPU masks
- Linux kernel memory management
- PCI
- e820
- System Management BIOS
- System Management BIOS
- EFI
- SMP
- MultiProcessor Specification
- BSS
- SMBIOS specification
- Previous part

Kernel initialization. Part 7.

The End of the architecture-specific initializations, almost...

This is the seventh parth of the Linux Kernel initialization process which covers internals of the `setup_arch` function from the arch/x86/kernel/setup.c. As you can know from the previous parts, the `setup_arch` function does some architecture-specific (in our case it is x86_64) initialization stuff like reserving memory for kernel code/data/bss, early scanning of the Desktop Management Interface, early dump of the PCI device and many many more. If you have read the previous part, you can remember that we've finished it at the `setup_real_mode` function. In the next step, as we set limit of the memblock to the all mapped pages, we can see the call of the `setup_log_buf` function from the kernel/printk/printk.c.

The `setup_log_buf` function setups kernel cyclic buffer which length depends on the `CONFIG_LOG_BUF_SHIFT` configuration option. As we can read from the documentation of the `CONFIG_LOG_BUF_SHIFT` it can be between `12` and `21`. In the internals, buffer defined as array of chars:

```
#define __LOG_BUF_LEN (1 << CONFIG_LOG_BUF_SHIFT)
static char __log_buf[__LOG_BUF_LEN] __aligned(LOG_ALIGN);
static char *log_buf = __log_buf;
```

Now let's look on the implementation of th `setup_log_buf` function. It starts with check that current buffer is empty (It must be empty, because we just setup it) and another check that it is early setup. If setup of the kernel log buffer is not early, we call the `log_buf_add_cpu` function which increase size of the buffer for every CPU:

```
if (log_buf != __log_buf)
    return;

if (!early && !new_log_buf_len)
    log_buf_add_cpu();
```

We will not research `log_buf_add_cpu` function, because as you can see in the `setup_arch`, we call `setup_log_buf` as:

```
setup_log_buf(1);
```

where `1` means that is is early setup. In the next step we check `new_log_buf_len` variable which is updated length of the kernel log buffer and allocate new space for the buffer with the `memblock_virt_alloc` function for it, or just return.

As kernel log buffer is ready, the next function is `reserve_initrd`. You can remember that we already called the `early_reserve_initrd` function in the fourth part of the Kernel initialization. Now, as we reconstructed direct memory mapping in the `init_mem_mapping` function, we need to move initrd to the down into directly mapped memory. The `reserve_initrd` function starts from the definition of the base address and end address of the `initrd` and check that `initrd` was provided by a bootloader. All the same as we saw it in the `early_reserve_initrd`. But instead of the reserving place in the `memblock` area with the call of the `memblock_reserve` function, we get the mapped size of the direct memory area and check that the size of the `initrd` is not greater that this area with:

```
mapped_size = memblock_mem_size(max_pfn_mapped);
if (ramdisk_size >= (mapped_size>>1))
    panic("initrd too large to handle, "
          "disabling initrd (%lld needed, %lld available)\n",
```

```
            ramdisk_size, mapped_size>>1);
```

You can see here that we call `memblock_mem_size` function and pass the `max_pfn_mapped` to it, where `max_pfn_mapped` contains the highest direct mapped page frame number. If you do not remember what is it `page frame number`, explanation is simple: First `12` bits of the virtual address represent offset in the physical page or page frame. If we will shift right virtual address on `12`, we'll discard offset part and will get `Page Frame Number`. In the `memblock_mem_size` we go through the all memblock `mem` (not reserved) regions and calculates size of the mapped pages amount and return it to the `mapped_size` variable (see code above). As we got amount of the direct mapped memory, we check that size of the `initrd` is not greater than mapped pages. If it is greater we just call `panic` which halts the system and prints popular Kernel panic message. In the next step we print information about the `initrd` size. We can see the result of this in the `dmesg` output:

```
[0.000000] RAMDISK: [mem 0x36d20000-0x37687fff]
```

and relocate `initrd` to the direct mapping area with the `relocate_initrd` function. In the start of the `relocate_initrd` function we try to find free area with the `memblock_find_in_range` function:

```
relocated_ramdisk = memblock_find_in_range(0, PFN_PHYS(max_pfn_mapped), area_size, PAGE_SIZE);

if (!relocated_ramdisk)
    panic("Cannot find place for new RAMDISK of size %lld\n",
           ramdisk_size);
```

The `memblock_find_in_range` function tries to find free area in a given range, in our case from `0` to the maximum mapped physical address and size must equal to the aligned size of the `initrd`. If we didn't find area with the given size, we call `panic` again. If all is good, we start to relocated RAM disk to the down of the directly mapped meory in the next step.

In the end of the `reserve_initrd` function, we free memblock memory which occupied by the ramdisk with the call of the:

```
memblock_free(ramdisk_image, ramdisk_end - ramdisk_image);
```

After we relocated `initrd` ramdisk image, the next function is `vsmp_init` from the arch/x86/kernel/vsmp_64.c. This function initializes support of the `ScaleMP vSMP`. As I already wrote in the previous parts, this chapter will not cover non-related `x86_64` initialization parts (for example as the current or `ACPI` and etc...). So we will miss implementation of this for now and will back to it in the part which will cover techniques of parallel computing.

The next function is `io_delay_init` from the arch/x86/kernel/io_delay.c. This function allows to override default default I/O delay `0x80` port. We already saw I/O delay in the Last preparation before transition into protected mode, now let's look on the `io_delay_init` implementation:

```
void __init io_delay_init(void)
{
    if (!io_delay_override)
        dmi_check_system(io_delay_0xed_port_dmi_table);
}
```

This function check `io_delay_override` variable and overrides I/O delay port if `io_delay_override` is set. We can set `io_delay_override` variably by passing `io_delay` option to the kernel command line. As we can read from the Documentation/kernel-parameters.txt, `io_delay` option is:

```
io_delay=    [X86] I/O delay method
    0x80
```

```
        Standard port 0x80 based delay
    0xed
        Alternate port 0xed based delay (needed on some systems)
    udelay
        Simple two microseconds delay
    none
        No delay
```

We can see `io_delay` command line parameter setup with the `early_param` macro in the arch/x86/kernel/io_delay.c

```
early_param("io_delay", io_delay_param);
```

More about `early_param` you can read in the previous part. So the `io_delay_param` function which setups `io_delay_override` variable will be called in the do_early_param function. `io_delay_param` function gets the argument of the `io_delay` kernel command line parameter and sets `io_delay_type` depends on it:

```
static int __init io_delay_param(char *s)
{
        if (!s)
                return -EINVAL;

        if (!strcmp(s, "0x80"))
                io_delay_type = CONFIG_IO_DELAY_TYPE_0X80;
        else if (!strcmp(s, "0xed"))
                io_delay_type = CONFIG_IO_DELAY_TYPE_0XED;
        else if (!strcmp(s, "udelay"))
                io_delay_type = CONFIG_IO_DELAY_TYPE_UDELAY;
        else if (!strcmp(s, "none"))
                io_delay_type = CONFIG_IO_DELAY_TYPE_NONE;
        else
                return -EINVAL;

        io_delay_override = 1;
        return 0;
}
```

The next functions are `acpi_boot_table_init`, `early_acpi_boot_init` and `initmem_init` after the `io_delay_init`, but as I wrote above we will not cover ACPI related stuff in this `Linux Kernel initialization process` chapter.

Allocate area for DMA

In the next step we need to allocate area for the Direct memory access with the `dma_contiguous_reserve` function which defined in the drivers/base/dma-contiguous.c. `DMA` area is a special mode when devices comminicate with memory without CPU. Note that we pass one parameter - `max_pfn_mapped << PAGE_SHIFT`, to the `dma_contiguous_reserve` function and as you can understand from this expression, this is limit of the reserved memory. Let's look on the implementation of this function. It starts from the definition of the following variables:

```
phys_addr_t selected_size = 0;
phys_addr_t selected_base = 0;
phys_addr_t selected_limit = limit;
bool fixed = false;
```

where first represents size in bytes of the reserved area, second is base address of the reserved area, third is end address of the reserved area and the last `fixed` parameter shows where to place reserved area. If `fixed` is `1` we just reserve area with the `memblock_reserve`, if it is `0` we allocate space with the `kmemleak_alloc`. In the next step we check `size_cmdline` variable and if it is not equal to `-1` we fill all variables which you can see above with the values from the `cma` kernel command line parameter:

```
if (size_cmdline != -1) {
    ...
    ...
    ...
}
```

You can find in this source code file definition of the early parameter:

```
early_param("cma", early_cma);
```

where `cma` is:

```
cma=nn[MG]@[start[MG][-end[MG]]]
        [ARM,X86,KNL]
        Sets the size of kernel global memory area for
        contiguous memory allocations and optionally the
        placement constraint by the physical address range of
        memory allocations. A value of 0 disables CMA
        altogether. For more information, see
        include/linux/dma-contiguous.h
```

If we will not pass `cma` option to the kernel command line, `size_cmdline` will be equal to `-1`. In this way we need to calculate size of the reserved area which depends on the following kernel configuration options:

- `CONFIG_CMA_SIZE_SEL_MBYTES` - size in megabytes, default global `CMA` area, which is equal to `CMA_SIZE_MBYTES * SZ_1M` or `CONFIG_CMA_SIZE_MBYTES * 1M`;
- `CONFIG_CMA_SIZE_SEL_PERCENTAGE` - percentage of total memory;
- `CONFIG_CMA_SIZE_SEL_MIN` - use lower value;
- `CONFIG_CMA_SIZE_SEL_MAX` - use higher value.

As we calculated the size of the reserved area, we reserve area with the call of the `dma_contiguous_reserve_area` function which first of all calls:

```
ret = cma_declare_contiguous(base, size, limit, 0, 0, fixed, res_cma);
```

function. The `cma_declare_contiguous` reserves contiguous area from the given base address and with given size. After we reserved area for the `DMA`, next function is the `memblock_find_dma_reserve`. As you can understand from its name, this function counts the reserved pages in the `DMA` area. This part will not cover all details of the `CMA` and `DMA`, because they are big. We will see much more details in the special part in the Linux Kernel Memory management which covers contiguous memory allocators and areas.

Initialization of the sparse memory

The next step is the call of the function - `x86_init.paging.pagetable_init`. If you will try to find this function in the linux kernel source code, in the end of your search, you will see the following macro:

```
#define native_pagetable_init        paging_init
```

which expands as you can see to the call of the `paging_init` function from the arch/x86/mm/init_64.c. The `paging_init` function initializes sparse memory and zone sizes. First of all what's zones and what is it `Sparsemem`. The `Sparsemem` is a special foundation in the linux kernen memory manager which used to split memory area to the different memory banks in the NUMA systems. Let's look on the implementation of the `paginig_init` function:

```
void __init paging_init(void)
{
        sparse_memory_present_with_active_regions(MAX_NUMNODES);
        sparse_init();

        node_clear_state(0, N_MEMORY);
        if (N_MEMORY != N_NORMAL_MEMORY)
                node_clear_state(0, N_NORMAL_MEMORY);

        zone_sizes_init();
}
```

As you can see there is call of the `sparse_memory_present_with_active_regions` function which records a memory area for every `NUMA` node to the array of the `mem_section` structure which contains a pointer to the structure of the array of `struct page`. The next `sparse_init` function allocates non-linear `mem_section` and `mem_map`. In the next step we clear state of the movable memory nodes and initialize sizes of zones. Every `NUMA` node is devided into a number of pieces which are called - `zones`. So, `zone_sizes_init` function from the arch/x86/mm/init.c initializes size of zones.

Again, this part and next parts do not cover this theme in full details. There will be special part about `NUMA`.

vsyscall mapping

The next step after `SparseMem` initialization is setting of the `trampoline_cr4_features` which must contain content of the `cr4` Control register. First of all we need to check that current CPU has support of the `cr4` register and if it has, we save its content to the `trampoline_cr4_features` which is storage for `cr4` in the real mode:

```
if (boot_cpu_data.cpuid_level >= 0) {
    mmu_cr4_features = __read_cr4();
    if (trampoline_cr4_features)
        *trampoline_cr4_features = mmu_cr4_features;
}
```

The next function which you can see is `map_vsyscal` from the arch/x86/kernel/vsyscall_64.c. This function maps memory space for vsyscalls and depends on `CONFIG_X86_VSYSCALL_EMULATION` kernel configuration option. Actually `vsyscall` is a special segment which provides fast access to the certain system calls like `getcpu` and etc... Let's look on implementation of this function:

```
void __init map_vsyscall(void)
{
        extern char __vsyscall_page;
        unsigned long physaddr_vsyscall = __pa_symbol(&__vsyscall_page);

        if (vsyscall_mode != NONE)
                __set_fixmap(VSYSCALL_PAGE, physaddr_vsyscall,
                             vsyscall_mode == NATIVE
                             ? PAGE_KERNEL_VSYSCALL
                             : PAGE_KERNEL_VVAR);

        BUILD_BUG_ON((unsigned long)__fix_to_virt(VSYSCALL_PAGE) !=
                     (unsigned long)VSYSCALL_ADDR);
}
```

In the beginning of the `map_vsyscal` we can see definition of two variables. The first is extern valirable `__vsyscall_page`. As variable extern, it defined somewhere in other source code file. Actually we can see definition of the `__vsyscall_page` in the arch/x86/kernel/vsyscall_emu_64.S. The `__vsyscall_page` symbol points to the aligned calls of the `vsyscalls` as `gettimeofday` and etc...:

```
    .globl __vsyscall_page
    .balign PAGE_SIZE, 0xcc
    .type __vsyscall_page, @object
__vsyscall_page:

    mov $__NR_gettimeofday, %rax
    syscall
    ret

    .balign 1024, 0xcc
    mov $__NR_time, %rax
    syscall
    ret
    ...
    ...
    ...
```

The second variable is `physaddr_vsyscall` which just stores physical address of the `__vsyscall_page` symbol. In the next step we check the `vsyscall_mode` variable, and if it is not equal to `NONE` which is `EMULATE` by default:

```
static enum { EMULATE, NATIVE, NONE } vsyscall_mode = EMULATE;
```

And after this check we can see the call of the `__set_fixmap` function which calls `native_set_fixmap` with the same parameters:

```
void native_set_fixmap(enum fixed_addresses idx, unsigned long phys, pgprot_t flags)
{
        __native_set_fixmap(idx, pfn_pte(phys >> PAGE_SHIFT, flags));
}

void __native_set_fixmap(enum fixed_addresses idx, pte_t pte)
{
        unsigned long address = __fix_to_virt(idx);

        if (idx >= __end_of_fixed_addresses) {
                BUG();
                return;
        }
        set_pte_vaddr(address, pte);
        fixmaps_set++;
}
```

Here we can see that `native_set_fixmap` makes value of `Page Table Entry` from the given physical address (physical address of the `__vsyscall_page` symbol in our case) and calls internal function - `__native_set_fixmap`. Internal function gets the virtual address of the given `fixed_addresses` index (`VSYSCALL_PAGE` in our case) and checks that given index is not greated than end of the fix-mapped addresses. After this we set page table entry with the call of the `set_pte_vaddr` function and increase count of the fix-mapped addresses. And in the end of the `map_vsyscall` we check that virtual address of the `VSYSCALL_PAGE` (which is first index in the `fixed_addresses`) is not greater than `VSYSCALL_ADDR` which is `-10UL << 20` or `ffffffffff600000` with the `BUILD_BUG_ON` macro:

```
BUILD_BUG_ON((unsigned long)__fix_to_virt(VSYSCALL_PAGE) !=
                     (unsigned long)VSYSCALL_ADDR);
```

Now `vsyscall` area is in the `fix-mapped` area. That's all about `map_vsyscall`, if you do not know anything about fix-mapped addresses, you can read Fix-Mapped Addresses and ioremap. More about `vsyscalls` we will see in the `vsyscalls and vdso` part.

Getting the SMP configuration

You can remember how we made a search of the SMP configuration in the previous part. Now we need to get the `SMP` configurtaion if we found it. For this we check `smp_found_config` variable which we set in the `smp_scan_config` function (read about it the previous part) and call the `get_smp_config` function:

```
if (smp_found_config)
    get_smp_config();
```

The `get_smp_config` expands to the `x86_init.mpparse.default_get_smp_config` function which defined in the arch/x86/kernel/mpparse.c. This function defines pointer to the multiprocessor floating pointer structure - `mpf_intel` (you can read about it in the previous part) and does some checks:

```
struct mpf_intel *mpf = mpf_found;

if (!mpf)
    return;

if (acpi_lapic && early)
   return;
```

Here we can see that multiprocessor configuration was found in the `smp_scan_config` function or just return from the function if not. The next check check that it is early. And as we did this checks, we start to read the `SMP` configuration. As we finished to read it, the next step is - `prefill_possible_map` function which makes preliminary filling of the possible CPUs `cpumask` (more about it you can read in the Introduction to the cpumasks).

The rest of the setup_arch

Here we are getting to the end of the `setup_arch` function. The rest function of course make important stuff, but details about these stuff will not will not be included in this part. We will just take a short look on these functions, because although they are important as I wrote above, but they cover non-generic kernel features related with the `NUMA`, `SMP`, `ACPI` and `APICs` and etc... First of all, the next call of the `init_apic_mappings` function. As we can understand this function sets the address of the local APIC. The next is `x86_io_apic_ops.init` and this function initializes I/O APIC. Please note that all details related with `APIC`, we will see in the chapter about interrupts and exceptions handling. In the next step we reserve standard I/O resources like `DMA`, `TIMER`, `FPU` and etc..., with the call of the `x86_init.resources.reserve_resources` function. Following is `mcheck_init` function initializes `Machine check Exception` and the last is `register_refined_jiffies` which registers jiffy (There will be separate chapter about timers in the kernel).

So that's all. Finally we have finished with the big `setup_arch` function in this part. Of course as I already wrote many times, we did not see full details about this function, but do not worry about it. We will be back more than once to this function from different chapters for understanding how different platform-dependent parts are initialized.

That's all, and now we can back to the `start_kernel` from the `setup_arch`.

Back to the main.c

As I wrote above, we have finished with the `setup_arch` function and now we can back to the `start_kernel` function from the init/main.c. As you can remember or even you saw yourself, `start_kernel` function is very big too as the `setup_arch`. So the couple of the next part will be dedicated to the learning of this function. So, let's continue with it. After the `setup_arch` we can see the call of the `mm_init_cpumask` function. This function sets the cpumask) pointer to the memory descriptor `cpumask`. We can look on its implementation:

```
static inline void mm_init_cpumask(struct mm_struct *mm)
{
#ifdef CONFIG_CPUMASK_OFFSTACK
        mm->cpu_vm_mask_var = &mm->cpumask_allocation;
#endif
        cpumask_clear(mm->cpu_vm_mask_var);
}
```

As you can see in the init/main.c, we passed memory descriptor of the init process to the `mm_init_cpumask` and here depend on `CONFIG_CPUMASK_OFFSTACK` configuration option we set or clear TLB switch `cpumask`.

In the next step we can see the call of the following function:

```
setup_command_line(command_line);
```

This function takes pointer to the kernel command line allocates a couple of buffers to store command line. We need a couple of buffers, because one buffer used for future reference and accessing to command line and one for parameter parsing. We will allocate space for the following buffers:

- `saved_command_line` - will contain boot command line;
- `initcall_command_line` - will contain boot command line. will be used in the `do_initcall_level`;
- `static_command_line` - will contain command line for parameters parsing.

We will allocate space with the `memblock_virt_alloc` function. This function calls `memblock_virt_alloc_try_nid` which allocates boot memory block with `memblock_reserve` if slab is not available or uses `kzalloc_node` (more about it will be in the linux memory management chapter). The `memblock_virt_alloc` uses `BOOTMEM_LOW_LIMIT` (physicall address of the `(PAGE_OFFSET + 0x1000000)` value) and `BOOTMEM_ALLOC_ACCESSIBLE` (equal to the current value of the `memblock.current_limit`) as minimum address of the memory egion and maximum address of the memory region.

Let's look on the implementation of the `setup_command_line`:

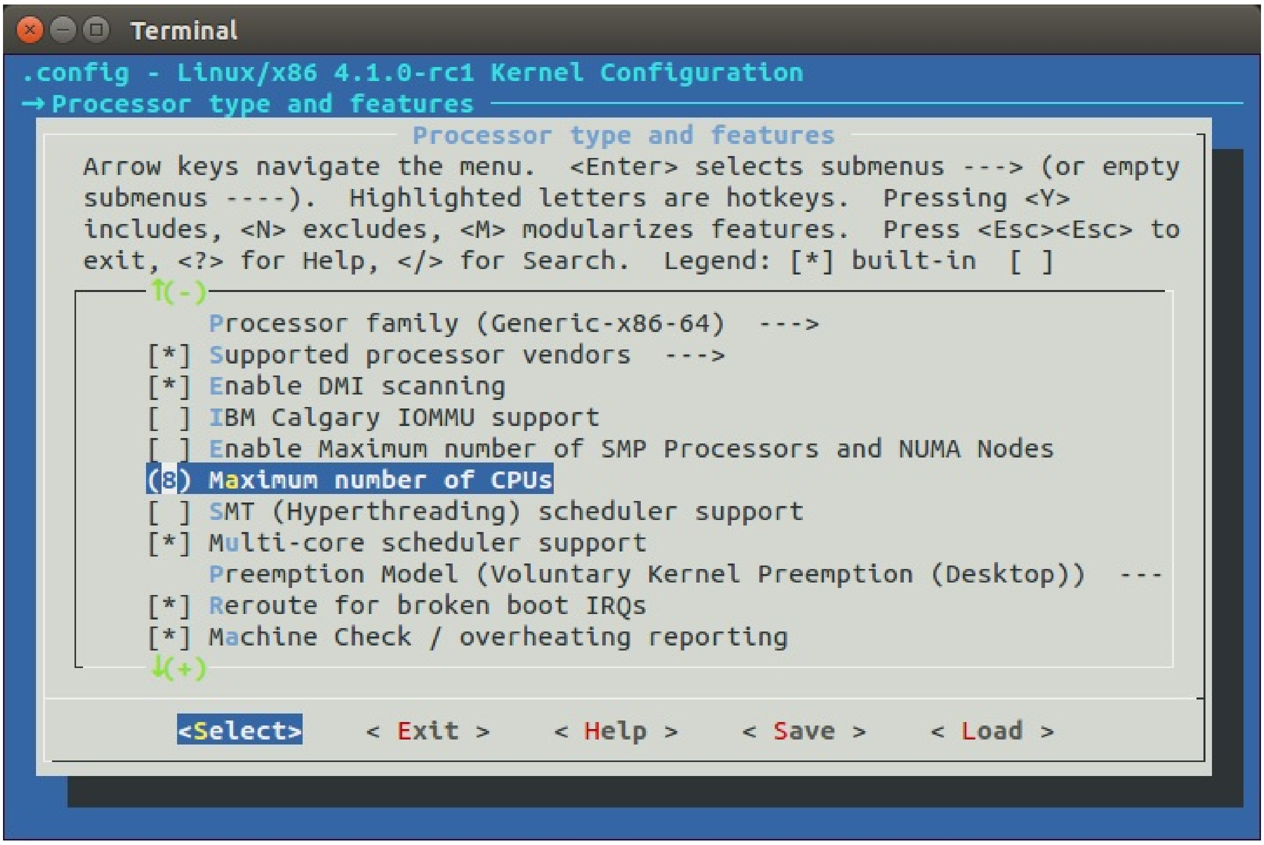
```
static void __init setup_command_line(char *command_line)
{
        saved_command_line =
                memblock_virt_alloc(strlen(boot_command_line) + 1, 0);
        initcall_command_line =
                memblock_virt_alloc(strlen(boot_command_line) + 1, 0);
        static_command_line = memblock_virt_alloc(strlen(command_line) + 1, 0);
        strcpy(saved_command_line, boot_command_line);
        strcpy(static_command_line, command_line);
 }
```

Here we can see that we allocate space for the three buffers which will contain kernel command line for the different purposes (read above). And as we allocated space, we storing `boot_comand_line` in the `saved_command_line` and `command_line` (kernel command line from the `setup_arch` to the `static_command_line`).

The next function after the `setup_command_line` is the `setup_nr_cpu_ids`. This function setting `nr_cpu_ids` (number of CPUs) according to the last bit in the `cpu_possible_mask` (more about it you can read in the chapter describes cpumasks concept). Let's look on its implementation:

```
void __init setup_nr_cpu_ids(void)
{
        nr_cpu_ids = find_last_bit(cpumask_bits(cpu_possible_mask),NR_CPUS) + 1;
}
```

Here `nr_cpu_ids` represents number of CPUs, `NR_CPUS` represents the maximum number of CPUs which we can set in configuration time:



Actually we need to call this function, because `NR_CPUS` can be greater than actual amount of the CPUs in the your computer. Here we can see that we call `find_last_bit` function and pass two parameters to it:

- `cpu_possible_mask` bits;
- maximim number of CPUS.

In the `setup_arch` we can find the call of the `prefill_possible_map` function which calculates and writes to the `cpu_possible_mask` actual number of the CPUs. We call the `find_last_bit` function which takes the address and maximum size to search and returns bit number of the first set bit. We passed `cpu_possible_mask` bits and maximum number of the CPUs. First of all the `find_last_bit` function splits given `unsigned long` address to the words:

```
words = size / BITS_PER_LONG;
```

where `BITS_PER_LONG` is `64` on the `x86_64`. As we got amount of words in the given size of the search data, we need to check is given size does not contain partial words with the following check:

```
if (size & (BITS_PER_LONG-1)) {
        tmp = (addr[words] & (~0UL >> (BITS_PER_LONG
                                 - (size & (BITS_PER_LONG-1)))));
        if (tmp)
                goto found;
}
```

if it contains partial word, we mask the last word and check it. If the last word is not zero, it means that current word contains at least one set bit. We go to the `found` label:

```
found:
    return words * BITS_PER_LONG + __fls(tmp);
```

Here you can see `__fls` function which returns last set bit in a given word with help of the `bsr` instruction:

```
static inline unsigned long __fls(unsigned long word)
{
        asm("bsr %1,%0"
            : "=r" (word)
            : "rm" (word));
        return word;
}
```

The `bsr` instruction which scans the given operand for first bit set. If the last word is not partial we going through the all words in the given address and trying to find first set bit:

```
while (words) {
    tmp = addr[--words];
    if (tmp) {
found:
        return words * BITS_PER_LONG + __fls(tmp);
    }
}
```

Here we put the last word to the `tmp` variable and check that `tmp` contains at least one set bit. If a set bit found, we return the number of this bit. If no one words do not contains set bit we just return given size:

```
return size;
```

After this `nr_cpu_ids` will contain the correct amount of the avaliable CPUs.

That's all.

Conclusion

It is the end of the seventh part about the linux kernel initialization process. In this part, finally we have finsihed with the `setup_arch` function and returned to the `start_kernel` function. In the next part we will continue to learn generic kernel code from the `start_kernel` and will continue our way to the first `init` process.

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

- Desktop Management Interface
- x86_64
- initrd
- Kernel panic
- Documentation/kernel-parameters.txt
- ACPI

* Direct memory access
* NUMA
* Control register
* vsyscalls
* SMP
* jiffy
* Previous part

Linux kernel memory management

This chapter describes memory management in the linux kernel. You will see here a couple of posts which describe different parts of the linux memory management framework:

* Memblock - describes early `memblock` allocator.
* Fix-Mapped Addresses and ioremap - describes `fix-mapped` addresses and early `ioremap`.

Linux kernel memory management Part 1.

Introduction

Memory management is a one of the most complex (and I think that it is the most complex) parts of the operating system kernel. In the last preparations before the kernel entry point part we stopped right before call of the `start_kernel` function. This function initializes all the kernel features (including architecture-dependent features) before the kernel runs the first `init` process. You may remember as we built early page tables, identity page tables and fixmap page tables in the boot time. No compilcated memory management is working yet. When the `start_kernel` function is called we will see the transition to more complex data structures and techniques for memory management. For a good understanding of the initialization process in the linux kernel we need to have clear understanding of the techniques. This chapter will provide an overview of the different parts of the linux kernel memory management framework and its API, starting from the `memblock`.

Memblock

Memblock is one of methods of managing memory regions during the early bootstrap period while the usual kernel memory allocators are not up and running yet. Previously it was called - `Logical Memory Block`, but from the patch by Yinghai Lu, it was renamed to the `memblock`. As Linux kernel for `x86_64` architecture uses this method. We already met `memblock` in the Last preparations before the kernel entry point part. And now time to get acquainted with it closer. We will see how it is implemented.

We will start to learn `memblock` from the data structures. Definitions of the all data structures can be found in the include/linux/memblock.h header file.

The first structure has the same name as this part and it is:

```
struct memblock {
         bool bottom_up;
         phys_addr_t current_limit;
         struct memblock_type memory;   --> array of memblock_region
         struct memblock_type reserved; --> array of memblock_region
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
         struct memblock_type physmem;
#endif
};
```

This structure contains five fields. First is `bottom_up` which allows to allocate memory in bottom-up mode when it is `true`. Next field is `current_limit`. This field describes the limit size of the memory block. The next three fields describes the type of the memory block. It can be: reserved, memory and physical memory if `CONFIG_HAVE_MEMBLOCK_PHYS_MAP` configuration option is enabled. Now we met yet another data structure - `memblock_type`. Let's look on its definition:

```
struct memblock_type {
    unsigned long cnt;
    unsigned long max;
    phys_addr_t total_size;
    struct memblock_region *regions;
};
```

This structure provides information about memory type. It contains fields which describe number of memory regions which are inside current memory block, size of the all memory regions, size of the allocated array of the memory regions and pointer to the array of the `memblock_region` structures. `memblock_region` is a structure which describes memory region. Its definition looks:

```
struct memblock_region {
        phys_addr_t base;
        phys_addr_t size;
        unsigned long flags;
#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
        int nid;
#endif
};
```

`memblock_region` provides base address and size of the memory region, flags which can be:

```
#define MEMBLOCK_ALLOC_ANYWHERE    (~(phys_addr_t)0)
#define MEMBLOCK_ALLOC_ACCESSIBLE    0
#define MEMBLOCK_HOTPLUG    0x1
```

Also `memblock_region` provides integer field - numa node selector, if `CONFIG_HAVE_MEMBLOCK_NODE_MAP` configuration option is enabled.

Schematically we can imagine it as:

```
+---------------------------+   +---------------------------+
|         memblock          |   |                           |
|  _______________________  |   |                           |
| |        memory         | |   |       Array of the        |
| |      memblock_type    |-|-->|      membock_region       |
| |_______________________| |   |                           |
|                           |   +---------------------------+
|  _______________________  |   +---------------------------+
| |       reserved        | |   |                           |
| |      memblock_type    |-|-->|       Array of the        |
| |_______________________| |   |      memblock_region      |
|                           |   |                           |
+---------------------------+   +---------------------------+
```

These three structures: `memblock`, `memblock_type` and `memblock_region` are main in the `Memblock`. Now we know about it and can look at Memblock initialization process.

Memblock initialization

As all API of the `memblock` described in the include/linux/memblock.h header file, all implementation of these function is in the mm/memblock.c source code file. Let's look on the top of source code file and we will look there initialization of the `memblock` structure:

```
struct memblock memblock __initdata_memblock = {
    .memory.regions         = memblock_memory_init_regions,
    .memory.cnt             = 1,
    .memory.max             = INIT_MEMBLOCK_REGIONS,

    .reserved.regions     = memblock_reserved_init_regions,
    .reserved.cnt         = 1,
    .reserved.max         = INIT_MEMBLOCK_REGIONS,

#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
    .physmem.regions     = memblock_physmem_init_regions,
    .physmem.cnt         = 1,
    .physmem.max         = INIT_PHYSMEM_REGIONS,
#endif
    .bottom_up             = false,
    .current_limit         = MEMBLOCK_ALLOC_ANYWHERE,
};
```

Here we can see initialization of the `memblock` structure which has the same name as structure - `memblock`. First of all note on `__initdata_memblock`. Defenition of this macro looks like:

```
#ifdef CONFIG_ARCH_DISCARD_MEMBLOCK
    #define __init_memblock __meminit
    #define __initdata_memblock __meminitdata
#else
    #define __init_memblock
    #define __initdata_memblock
#endif
```

You can note that it depends on `CONFIG_ARCH_DISCARD_MEMBLOCK`. If this configuration option is enabled, memblock code will be put to the `.init` section and it will be released after the kernel is booted up.

Next we can see initialization of the `memblock_type memory`, `memblock_type reserved` and `memblock_type physmem` fields of the `memblock` structure. Here we interesting only in the `memblock_type.regions` initialization process. Note that every `memblock_type` field initialized by the arrays of the `memblock_region`:

```
static struct memblock_region memblock_memory_init_regions[INIT_MEMBLOCK_REGIONS] __initdata_memblock;
static struct memblock_region memblock_reserved_init_regions[INIT_MEMBLOCK_REGIONS] __initdata_memblock;
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
static struct memblock_region memblock_physmem_init_regions[INIT_PHYSMEM_REGIONS] __initdata_memblock;
#endif
```

Every array contains 128 memory regions. We can see it in the `INIT_MEMBLOCK_REGIONS` macro definition:

```
#define INIT_MEMBLOCK_REGIONS   128
```

Note that all arrays are also defined with the `__initdata_memblock` macro which we already saw in the `memblock` strucutre initialization (read above if you've forgot).

The last two fields describe that `bottom_up` allocation is disabled and the limit of the current Memblock is:

```
#define MEMBLOCK_ALLOC_ANYWHERE (~(phys_addr_t)0)
```

which is `0xffffffffffffffff`.

On this step initialization of the `memblock` structure finished and we can look on the Memblock API.

Memblock API

Ok we have finished with initilization of the `memblock` structure and now we can look on the Memblock API and its implementation. As i said about, all implementation of the `memblock` presented in the mm/memblock.c. To understand how `memblock` works and implemented, let's look on it's usage first of all. There are a couple of places in the linux kernel where memblock is used. For example let's take `memblock_x86_fill` function from the arch/x86/kernel/e820.c. This function goes through the memory map provided by the e820 and adds memory regions reserved by the kernel to the `memblock` with the `memblock_add` function. As we met `memblock_add` function first, let's start from it.

This function takes physical base address and size of the memory region and adds it to the `memblock`. `memblock_add` function does not anything special in its body, but just calls:

```
memblock_add_range(&memblock.memory, base, size, MAX_NUMNODES, 0);
```

function. We pass memory block type - `memory`, physical base address and size of the memory region, maximum number of nodes which are zero if `CONFIG_NODES_SHIFT` is not set in the configuration file or `CONFIG_NODES_SHIFT` if it is set, and flags. `memblock_add_range` function adds new memory region to the memory block. It starts from check the size of the given region and if it is zero just return. After this, `memblock_add_range` check existence of the memory regions in the `memblock` structure with the given `memblock_type`. If there are no memory regions, we just fill new `memory_region` with the given values and return (we already saw implementation of this in the First touch of the linux kernel memory manager framework). If `memblock_type` is no empty, we start to add new memory region to the `memblock` with the given `memblock_type`.

First of all we get the end of the memory region with the:

```
phys_addr_t end = base + memblock_cap_size(base, &size);
```

`memblock_cap_size` adjusts `size` that `base + size` will not overflow. Its implementation pretty easy:

```
static inline phys_addr_t memblock_cap_size(phys_addr_t base, phys_addr_t *size)
{
    return *size = min(*size, (phys_addr_t)ULLONG_MAX - base);
}
```

`memblock_cap_size` returns new size which is the smallest value between the given `size` and base.

After that we got end address of the new memory region, `memblock_add_region` checks overlap and merge condititions with already added memory regions. Insertion of the new memory region to the `memblcok` consists from two steps:

- Adding of non-overlapping parts of the new memory area as separate regions;
- Merging of all neighbouring regions.

We are going throuth the all already stored memory regions and check overlapping:

```
    for (i = 0; i < type->cnt; i++) {
        struct memblock_region *rgn = &type->regions[i];
        phys_addr_t rbase = rgn->base;
        phys_addr_t rend = rbase + rgn->size;

        if (rbase >= end)
            break;
        if (rend <= base)
            continue;
        ...
        ...
        ...
    }
```

if new memory region does not overlap regions which are already stored in the `memblock`, insert this region into the memblock with and this is first step, we check that new region can fit into the memory block and call `memblock_double_array` in other way:

```
while (type->cnt + nr_new > type->max)
    if (memblock_double_array(type, obase, size) < 0)
        return -ENOMEM;
    insert = true;
    goto repeat;
```

`memblock_double_array` doubles the size of the given regions array. Than we set insert to the `true` and go to the `repeat` label. In the second step, starting from the `repeat` label we go through the same loop and insert current memory region into the memory block with the `memblock_insert_region` function:

```
    if (base < end) {
        nr_new++;
        if (insert)
            memblock_insert_region(type, i, base, end - base,
                                   nid, flags);
    }
```

As we set `insert` to `true` in the first step, now `memblock_insert_region` will be called. `memblock_insert_region` has almost the same implemetation that we saw when we insert new region to the empty `memblock_type` (see above). This function get the last memory region:

```
struct memblock_region *rgn = &type->regions[idx];
```

and copies memory area with `memmove`:

```
memmove(rgn + 1, rgn, (type->cnt - idx) * sizeof(*rgn));
```

After this fills `memblock_region` fields of the new memory region base, size and etc... and increase size of the `memblock_type`. In the end of the exution, `memblock_add_range` calls `memblock_merge_regions` which merges neighboring compatible regions in the second step.

In the second case new memory region can overlap already stored regions. For example we already have `region1` in the `memblock`:

```
0                    0x1000
+-----------------------+
|                       |
|                       |
|        region1        |
|                       |
|                       |
+-----------------------+
```

And now we want to add `region2` to the `memblock` with the following base address and size:

```
0x100                 0x2000
+-----------------------+
|                       |
|                       |
|        region2        |
|                       |
|                       |
+-----------------------+
```

In this case set the base address of the new memory region as the end address of the overlapped region with:

```
base = min(rend, end);
```

So it will be `0x1000` in our case. And insert it as we did it already in the second step with:

```
if (base < end) {
    nr_new++;
    if (insert)
        memblock_insert_region(type, i, base, end - base, nid, flags);
}
```

In this case we insert `overlapping portion` (we insert only higher portion, because lower already in the overlapped memory region), than remaining portion and merge these portions with `memblock_merge_regions`. As i said above `memblock_merge_regions` function merges neighboring compatible regions. It goes through the all memory regions from the given `memblock_type`, takes two neighboring memory regions - `type->regions[i]` and `type->regions[i + 1]` and checks that these regions have the same flags, belong to the same node and that end address of the first regions is not equal to the base address of the second region:

```
while (i < type->cnt - 1) {
    struct memblock_region *this = &type->regions[i];
    struct memblock_region *next = &type->regions[i + 1];
    if (this->base + this->size != next->base ||
        memblock_get_region_node(this) !=
        memblock_get_region_node(next) ||
        this->flags != next->flags) {
        BUG_ON(this->base + this->size > next->base);
        i++;
        continue;
    }
```

If none of these conditions are not true, we update the size of the first region with the size of the next region:

```
this->size += next->size;
```

As we update the size of the first memory region with the size of the next memory region, we copy every (in the loop) memory region which is after the current (`this`) memory region on the one index ago with the `memmove` function:

```
memmove(next, next + 1, (type->cnt - (i + 2)) * sizeof(*next));
```

And decrease the count of the memory regions which are belongs to the `memblock_type`:

```
type->cnt--;
```

After this we will get two memory regions merged into one:

```
0                                              0x2000
+------------------------------------------------+
|                                                |
|                                                |
|                   region1                      |
|                                                |
|                                                |
+------------------------------------------------+
```

That's all. This is the whole principle of the work of the `memblock_add_range` function.

There is also `memblock_reserve` function which does the same as `memblock_add`, but only with one difference. It stores `memblock_type.reserved` in the memblock instead of `memblock_type.memory`.

Of course it is not full API. Memblock provides API for not only adding `memory` and `reserved` memory regions, but also:

- memblock_remove - removes memory region from memblock;
- memblock_find_in_range - finds free area in given range;
- memblock_free - releases memory region in memblock;
- for_each_mem_range - iterates through memblock areas.

and many more....

Getting info about memory regions

Memblock also provides API for the getting information about allocated memorey regions in the `memblcok` . It splitted on two parts:

- get_allocated_memblock_memory_regions_info - getting info about memory regions;
- get_allocated_memblock_reserved_regions_info - getting info about reserved regions.

Implementation of these function is easy. Let's look on `get_allocated_memblock_reserved_regions_info` for example:

```
phys_addr_t __init_memblock get_allocated_memblock_reserved_regions_info(
                    phys_addr_t *addr)
{
    if (memblock.reserved.regions == memblock_reserved_init_regions)
        return 0;

    *addr = __pa(memblock.reserved.regions);

    return PAGE_ALIGN(sizeof(struct memblock_region) *
              memblock.reserved.max);
}
```

First of all this function checks that `memblock` contains reserved memory regions. If `memblock` does not contain reserved memory regions we just return zero. In other way we write physical address of the reserved memory regions array to the given address and return aligned size of the allicated aray. Note that there is `PAGE_ALIGN` macro used for align. Actually it depends on size of page:

```
#define PAGE_ALIGN(addr) ALIGN(addr, PAGE_SIZE)
```

Implementation of the `get_allocated_memblock_memory_regions_info` function is the same. It has only one difference, `memblock_type.memory` used instead of `memblock_type.memory` .

Memblock debugging

There are many calls of the `memblock_dbg` in the memblock implementation. If you will pass `memblock=debug` option to the kernel command line, this function will be called. Actually `memblock_dbg` is just a macro which expands to the `printk` :

```
#define memblock_dbg(fmt, ...) \
         if (memblock_debug) printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
```

For example you can see call of this macro in the `memblock_reserve` function:

```
memblock_dbg("memblock_reserve: [%#016llx-%#016llx] flags %#02lx %pF\n",
```

```
                (unsigned long long)base,
                (unsigned long long)base + size - 1,
                flags, (void *)_RET_IP_);
```

And you must see something like this:

```
Kernel command line: root=/dev/sdb earlyprintk=ttyS0 loglevel=7 debug rdinit=/sbin/init root=/dev/ram memblock=debug
memblock_virt_alloc_try_nid_nopanic: 32768 bytes align=0x0 nid=-1 from=0x0 max_addr=0x0 alloc_large_system_hash+0x144/0x228
memblock_reserve: [0x0000023ff38e00-0x0000023ff40dff] flags 0x0 memblock_virt_alloc_internal+0xfd/0x13f
PID hash table entries: 4096 (order: 3, 32768 bytes)
memblock_virt_alloc_try_nid_nopanic: 67108864 bytes align=0x1000 nid=-1 from=0x0 max_addr=0xffffffff swiotlb_init+0x4c/0xad
memblock_reserve: [0x000000bbfe0000-0x000000bffdffff] flags 0x0 memblock_virt_alloc_internal+0xfd/0x13f
memblock_virt_alloc_try_nid_nopanic: 32768 bytes align=0x1000 nid=-1 from=0x0 max_addr=0xffffffff swiotlb_init_with_tbl+0x69/0x147
memblock_reserve: [0x000000bbfd8000-0x000000bbfdffff] flags 0x0 memblock_virt_alloc_internal+0xfd/0x13f
memblock_virt_alloc_try_nid: 131072 bytes align=0x1000 nid=-1 from=0x0 max_addr=0x0 swiotlb_init_with_tbl+0xb9/0x147
memblock_reserve: [0x0000023ff18000-0x0000023ff37fff] flags 0x0 memblock_virt_alloc_internal+0xfd/0x13f
memblock_virt_alloc_try_nid: 262144 bytes align=0x1000 nid=-1 from=0x0 max_addr=0x0 swiotlb_init_with_tbl+0xe8/0x147
memblock_reserve: [0x0000023fed8000-0x0000023ff17fff] flags 0x0 memblock_virt_alloc_internal+0xfd/0x13f
```

Memblock has also support in debugfs. If you run kernel not in `X86` architecture you can access:

- /sys/kernel/debug/memblock/memory
- /sys/kernel/debug/memblock/reserved
- /sys/kernel/debug/memblock/physmem

for getting dump of the `memblock` contents.

Conclusion

This is the end of the first part about linux kernel memory management. If you have questions or suggestions, ping me in twitter 0xAX, drop me email or just create issue.

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

- e820
- numa
- debugfs
- First touch of the linux kernel memory manager framework

Linux kernel memory management Part 2.

Fix-Mapped Addresses and ioremap

`Fix-Mapped` addresses is a set of the special compile-time addresses whose corresponding physical address do not have to be linear address minus `__START_KERNEL_map`. Each fix-mapped address maps one page frame and kernel uses they as pointers that never change their addresses. It is the main point of these addresses. As comment says: `to have a constant address at compile time, but to set the physical address only in the boot process`. You can remember that in the earliest part, we already set the `level2_fixmap_pgt`:

```
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
```

As you can see `level2_fixmap_pgt` is right after the `level2_kernel_pgt` which is kernel code+data+bss. Every fix-mapped address is presented by a integer index which is defined in the `fixed_addresses` enum from the arch/x86/include/asm/fixmap.h. For example it contains entries for `VSYSCALL_PAGE` - if emulation of legacy vsyscall page is enabled, `FIX_APIC_BASE` for local apic and etc... In a virtual memory fix-mapped area is placed in the modules area:

```
          +-----------+-----------------+---------------+------------------+
          |           |                 |               |                  |
          |kernel text|      kernel     |               |    vsyscalls     |
          | mapping   |       text      |    Modules    |    fix-mapped    |
          |from phys 0|       data      |               |    addresses     |
          |           |                 |               |                  |
          +-----------+-----------------+---------------+------------------+
   __START_KERNEL_map   __START_KERNEL    MODULES_VADDR            0xffffffffffffffff
```

Base virtual address and size of the `fix-mapped` area are presented by the two following macro:

```
#define FIXADDR_SIZE    (__end_of_permanent_fixed_addresses << PAGE_SHIFT)
#define FIXADDR_START        (FIXADDR_TOP - FIXADDR_SIZE)
```

Here `__end_of_permanent_fixed_addresses` is an element of the `fixed_addresses` enum and as I wrote above: Every fix-mapped address is presented by a integer index which is defined in the `fixed_addresses`. `PAGE_SHIFT` determines size of a page. For example size of the one page we can get with the `1 << PAGE_SHIFT`. In our case we need to get the size of the fix-mapped area, but not only of one page, that's why we are using `__end_of_permanent_fixed_addresses` for getting the size of the fix-mapped area. In my case it's a little more than `536` killobytes. In your case it can be different number, because the size depends on amount of the fix-mapped addresses which are depends on your kernel's configuration.

The second `FIXADDR_START` macro just extracts from the last address of the fix-mapped area its size for getting base virtual address of the fix-mapped area. `FIXADDR_TOP` is rounded up address from the base address of the vsyscall space:

```
#define FIXADDR_TOP     (round_up(VSYSCALL_ADDR + PAGE_SIZE, 1<<PMD_SHIFT) - PAGE_SIZE)
```

The `fixed_addresses` enums are used as an index to get the virtual address using the `fix_to_virt` function.

Implementation of this function is easy:

```
static __always_inline unsigned long fix_to_virt(const unsigned int idx)
{
        BUILD_BUG_ON(idx >= __end_of_fixed_addresses);
        return __fix_to_virt(idx);
}
```

first of all it check that given index of `fixed_addresses` enum is not greater or equal than `__end_of_fixed_addresses` with the `BUILD_BUG_ON` macro and than returns the result of the `__fix_to_virt` macro:

```
#define __fix_to_virt(x)        (FIXADDR_TOP - ((x) << PAGE_SHIFT))
```

Here we shift left the given `fix-mapped` address index on the `PAGE_SHIFT` which determines size of a page as I wrote above and subtract it from the `FIXADDR_TOP` which is the highest address of the `fix-mapped` area. There is inverse function for getting `fix-mapped` address from a virtual address:

```
static inline unsigned long virt_to_fix(const unsigned long vaddr)
{
        BUG_ON(vaddr >= FIXADDR_TOP || vaddr < FIXADDR_START);
        return __virt_to_fix(vaddr);
}
```

`virt_to_fix` takes virtual address, checks that this address is between `FIXADDR_START` and `FIXADDR_TOP` and calls `__virt_to_fix` macro which implemented as:

```
#define __virt_to_fix(x)        ((FIXADDR_TOP - ((x)&PAGE_MASK)) >> PAGE_SHIFT)
```

A PFN is simply in index within physical memory that is counted in page-sized units. PFN for a physical address could be trivially defined as (page_phys_addr >> PAGE_SHIFT);

`__virt_to_fix` clears first 12 bits in the given address, subtracts it from the last address the of `fix-mapped` area (`FIXADDR_TOP`) and shifts right result on `PAGE_SHIFT` which is `12` . Let I explain how it works. As i already wrote we will crear first 12 bits in the given address with `x & PAGE_MASK` . As we subtract this from the `FIXADDR_TOP` , we will get last 12 bits of the `FIXADDR_TOP` which are represent. We know that first 12 bits of the virtual address present offset in the page frame. With the shiting it on `PAGE_SHIFT` we will get `Page frame number` which is just all bits in a virtual address besides first 12 offset bits. `Fix-mapped` addresses are used in different places of the linux kernel. `IDT` descriptor stored there, Intel Trusted Execution Technology UUID stored in the `fix-mapped` area started from `FIX_TBOOT_BASE` index, Xen bootmap and many more... We already saw a little about `fix-mapped` addresses in the fifth part about linux kernel initialization. We used `fix-mapped` area in the early `ioremap` initialization. Let's look on it and try to understand what is it `ioremap` , how it implemented in the kernel and how it releated with the `fix-mapped` addresses.

ioremap

Linux kernel provides many different primitives to manage memory. For this moment we will touch `I/O memory` . Every device controlled with reading/writing from/to its registers. For example driver can turn off/on a device by writing to the its registers or get state of a device by reading from its registers. Besides registers, many devices have buffer and where driver can write something or read from there. As we know for this moment there are two ways to access device's registers and data buffers:

- through the I/O ports;

- mapping of the all registers to the memory address space;

In the first case every control register of a device has a number of input and output port. And driver of a device can read from a port and write to it with two `in` and `out` instructions which we already saw. If you want to know about currently registered port regions, you can know they by accessing of `/proc/ioports` :

```
$ cat /proc/ioports
0000-0cf7 : PCI Bus 0000:00
  0000-001f : dma1
  0020-0021 : pic1
  0040-0043 : timer0
  0050-0053 : timer1
  0060-0060 : keyboard
  0064-0064 : keyboard
  0070-0077 : rtc0
  0080-008f : dma page reg
  00a0-00a1 : pic2
  00c0-00df : dma2
  00f0-00ff : fpu
    00f0-00f0 : PNP0C04:00
  03c0-03df : vesafb
  03f8-03ff : serial
  04d0-04d1 : pnp 00:06
  0800-087f : pnp 00:01
  0a00-0a0f : pnp 00:04
  0a20-0a2f : pnp 00:04
  0a30-0a3f : pnp 00:04
0cf8-0cff : PCI conf1
0d00-ffff : PCI Bus 0000:00
...
...
...
```

`/proc/ioporst` provides information about what driver used address of a `I/O` ports region. All of these memory regions, for example `0000-0cf7` , were claimed with the `request_region` function from the include/linux/ioport.h. Actuall `request_region` is a macro which defied as:

```
#define request_region(start,n,name)   __request_region(&ioport_resource, (start), (n), (name), 0)
```

As we can see it takes three parameters:

- `start` - begin of region;
- `n` - length of region;
- `name` - name of requester.

`request_region` allocates `I/O` port region. Very often `check_region` function called before the `request_region` to check that the given address range is available and `release_region` to release memory region. `request_region` returns pointer to the `resource` structure. `resource` structure presents abstraction for a tree-like subset of system resources. We already saw `resource` structure in the firth part about kernel initialization process and it looks as:

```
struct resource {
        resource_size_t start;
        resource_size_t end;
        const char *name;
        unsigned long flags;
        struct resource *parent, *sibling, *child;
};
```

and contains start and end addresses of the resource, name and etc... Every `resource` structure contains pointers to the `parent` , `slibling` and `child` resources. As it has parent and childs, it means that every subset of resuorces has root `resource` structure. For example, for `I/O` ports it is `ioport_resource` structure:

struct resource ioport_resource = { .name = "PCI IO", .start = 0, .end = IO_SPACE_LIMIT, .flags = IORESOURCE_IO, }; EXPORT_SYMBOL(ioport_resource);

Or for `iomem`, it is `iomem_resource` structure:

```
struct resource iomem_resource = {
        .name   = "PCI mem",
        .start  = 0,
        .end    = -1,
        .flags  = IORESOURCE_MEM,
};

As I wrote about `request_regions` is used for registering of I/O port region and this macro used in many [places](http

```C
module_init(rtc_init);
```

where `rtc_init` is `rtc` initialization function. This function defined in the same `rtc.c` source code file. In the `rtc_init` function we can see a couple calls of the `rtc_request_region` functions, which wrap `request_region` for example:

```
r = rtc_request_region(RTC_IO_EXTENT);
```

where `rtc_request_region` calls:

```
r = request_region(RTC_PORT(0), size, "rtc");
```

Here `RTC_IO_EXTENT` is a size of memory region and it is `0x8`, `"rtc"` is a name of region and `RTC_PORT` is:

```
#define RTC_PORT(x)     (0x70 + (x))
```

So with the `request_region(RTC_PORT(0), size, "rtc")` we register memory region, started at `0x70` and with size `0x8`. Let's look on the `/proc/ioports`:

```
~$ sudo cat /proc/ioports | grep rtc
0070-0077 : rtc0
```

So, we got it! Ok, it was ports. The second way is use of `I/O` memory. As I wrote above this was is mapping of control registers and memory of a device to the memory address space. `I/O` memory is a set of contiguous addresses which are provides by a device to CPU through a bus. All memory-mapped I/O addresses are not used by the kernel directly. There is special `ioremap` function which allows to covert physical address on a bus to the kernel virtual address or in another words `ioremap` maps I/O physical memory region to access it from the kernel. `ioremap` function takes two parameters:

- start of the memory region;
- size of the memory region;

I/O memory mapping API provides function for the checking, requesting and release of a memory region as this does I/O ports API. There are three functions for it:

- `request_mem_region`
```

* `release_mem_region`
* `check_mem_region`

```
~$ sudo cat /proc/iomem
...
...
...
be826000-be82cfff : ACPI Non-volatile Storage
be82d000-bf744fff : System RAM
bf745000-bfff4fff : reserved
bfff5000-dc041fff : System RAM
dc042000-dc0d2fff : reserved
dc0d3000-dc138fff : System RAM
dc139000-dc27dfff : ACPI Non-volatile Storage
dc27e000-deffefff : reserved
defff000-deffffff : System RAM
df000000-dfffffff : RAM buffer
e0000000-feafffff : PCI Bus 0000:00
  e0000000-efffffff : PCI Bus 0000:01
    e0000000-efffffff : 0000:01:00.0
  f7c00000-f7cfffff : PCI Bus 0000:06
    f7c00000-f7c0ffff : 0000:06:00.0
    f7c10000-f7c101ff : 0000:06:00.0
      f7c10000-f7c101ff : ahci
  f7d00000-f7dfffff : PCI Bus 0000:03
    f7d00000-f7d3ffff : 0000:03:00.0
      f7d00000-f7d3ffff : alx
...
...
...
```

Part of these addresses is from the call of the `e820_reserve_resources` function. We can find call of this function in the arch/x86/kernel/setup.c and the function itself defined in the arch/x86/kernel/e820.c. `e820_reserve_resources` goes through the e820 map and inserts memory regions to the root `iomem` resource structure. All `e820` memory regions which are will be inserted to the `iomem` resource will have following types:

```
static inline const char *e820_type_to_string(int e820_type)
{
    switch (e820_type) {
    case E820_RESERVED_KERN:
    case E820_RAM:    return "System RAM";
    case E820_ACPI:    return "ACPI Tables";
    case E820_NVS:    return "ACPI Non-volatile Storage";
    case E820_UNUSABLE:    return "Unusable memory";
    default:    return "reserved";
    }
}
```

and we can see it in the `/proc/iomem` (read above).

Now let's try to understand how `ioremap` works. We already know little about `ioremap`, we saw it in the fifth part about linux kernel initialization. If you have read this part, you can remember call of the `early_ioremap_init` function from the arch/x86/mm/ioremap.c. Initialization of the `ioremap` splitten on two parts: there is early part which we can use before normal `ioremap` is available and normal `ioremap` which is available after `vmalloc` initialization and call of the `paging_init`. We do not know anything about `vmalloc` for now, so let's consider early initialization of the `ioremap`. First of all `early_ioremap_init` checks that `fixmap` is aligned on page middle directory boundary:

```
BUILD_BUG_ON((fix_to_virt(0) + PAGE_SIZE) & ((1 << PMD_SHIFT) - 1));
```

more about `BUILD_BUG_ON` you can read in the first part about Linux Kernel initialization. So `BUILD_BUG_ON` macro raises compilation error if the given expression is true. In the next step after this check, we can see call of the `early_ioremap_setup` function from the mm/early_ioremap.c. This function presents generic initialization of the `ioremap`. `early_ioremap_setup` function fills the `slot_virt` array with the virtual addresses of the early fixmaps. All early fixmaps are after `__end_of_permanent_fixed_addresses` in memory. They are stats from the `FIX_BITMAP_BEGIN` (top) and ends with `FIX_BITMAP_END` (down). Actually there are `512` temporary boot-time mappings, used by early `ioremap`:

```
#define NR_FIX_BTMAPS        64
#define FIX_BTMAPS_SLOTS    8
#define TOTAL_FIX_BTMAPS    (NR_FIX_BTMAPS * FIX_BTMAPS_SLOTS)
```

and `early_ioremap_setup`:

```
void __init early_ioremap_setup(void)
{
        int i;

        for (i = 0; i < FIX_BTMAPS_SLOTS; i++)
                if (WARN_ON(prev_map[i]))
                        break;

        for (i = 0; i < FIX_BTMAPS_SLOTS; i++)
                slot_virt[i] = __fix_to_virt(FIX_BTMAP_BEGIN - NR_FIX_BTMAPS*i);
}
```

the `slot_virt` and other arrays are defined in the same source code file:

```
static void __iomem *prev_map[FIX_BTMAPS_SLOTS] __initdata;
static unsigned long prev_size[FIX_BTMAPS_SLOTS] __initdata;
static unsigned long slot_virt[FIX_BTMAPS_SLOTS] __initdata;
```

`slot_virt` contains virtual addresses of the `fix-mapped` areas, `prev_map` array contains addresses of the early ioremap areas. Note that I wrote above: `Actually there are 512 temporary boot-time mappings, used by early ioremap` and you can see that all arrays defined with the `__initdata` attribute which means that this memory will be released after kernel initialization process. After `early_ioremap_setup` finished to work, we're getting page middle directory where early ioremap beginning with the `early_ioremap_pmd` function which just gets the base address of the page global directory and calculates the page middle directory for the given address:

```
static inline pmd_t * __init early_ioremap_pmd(unsigned long addr)
{
    pgd_t *base = __va(read_cr3());
    pgd_t *pgd = &base[pgd_index(addr)];
    pud_t *pud = pud_offset(pgd, addr);
    pmd_t *pmd = pmd_offset(pud, addr);
    return pmd;
}
```

After this we fills `bm_pte` (early ioremap page table entries) with zeros and call the `pmd_populate_kernel` function:

```
pmd = early_ioremap_pmd(fix_to_virt(FIX_BTMAP_BEGIN));
memset(bm_pte, 0, sizeof(bm_pte));
pmd_populate_kernel(&init_mm, pmd, bm_pte);
```

`pmd_populate_kernel` takes three parameters:

- `init_mm` - memory descriptor of the `init` process (you can read about it in the previous part);
- `pmd` - page middle directory of the beginning of the `ioremap` fixmaps;

- `bm_pte` - early `ioremap` page table entries array which defined as:

```
static pte_t bm_pte[PAGE_SIZE/sizeof(pte_t)] __page_aligned_bss;
```

The `pmd_popularte_kernel` function defined in the arch/x86/include/asm/pgalloc.h and populates given page middle directory (`pmd`) with the given page table entries (`bm_pte`):

```
static inline void pmd_populate_kernel(struct mm_struct *mm,
                                       pmd_t *pmd, pte_t *pte)
{
        paravirt_alloc_pte(mm, __pa(pte) >> PAGE_SHIFT);
        set_pmd(pmd, __pmd(__pa(pte) | _PAGE_TABLE));
}
```

where `set_pmd` is:

```
#define set_pmd(pmdp, pmd)              native_set_pmd(pmdp, pmd)
```

and `native_set_pmd` is:

```
static inline void native_set_pmd(pmd_t *pmdp, pmd_t pmd)
{
        *pmdp = pmd;
}
```

That's all. Early `ioremap` is ready to use. There are a couple of checks in the `early_ioremap_init` function, but they are not so important, anyway initialization of the `ioremap` is finished.

Use of early ioremap

As early `ioremap` is setup, we can use it. It provides two functions:

- early_ioremap
- early_iounmap

for mapping/unmapping of IO physical address to virtual address. Both functions depends on `CONFIG_MMU` configuration option. Memory management unit is a special block of memory management. Main purpose of this block is translation physical addresses to the virtual. Techinically memory management unit knows about high-level page table address (`pgd`) from the `cr3` control register. If `CONFIG_MMU` options is set to `n` , `early_ioremap` just returns the given physical address and `early_iounmap` does not nothing. In other way, if `CONFIG_MMU` option is set to `y` , `early_ioremap` calls `__early_ioremap` which takes three parameters:

- `phys_addr` - base physicall address of the `I/O` memory region to map on virtual addresses;
- `size` - size of the `I/O` memroy region;
- `prot` - page table entry bits.

First of all in the `__early_ioremap` , we goes through the all early ioremap fixmap slots and check first free are in the `prev_map` array and remember it's number in the `slot` variable and set up size as we found it:

```
slot = -1;
for (i = 0; i < FIX_BTMAPS_SLOTS; i++) {
```

```
    if (!prev_map[i]) {
        slot = i;
        break;
    }
}
...
...
...
prev_size[slot] = size;
last_addr = phys_addr + size - 1;
```

In the next spte we can see the following code:

```
offset = phys_addr & ~PAGE_MASK;
phys_addr &= PAGE_MASK;
size = PAGE_ALIGN(last_addr + 1) - phys_addr;
```

Here we are using `PAGE_MASK` for clearing all bits in the `phys_addr` besides first 12 bits. `PAGE_MASK` macro defined as:

```
#define PAGE_MASK       (~(PAGE_SIZE-1))
```

We know that size of a page is 4096 bytes or `1000000000000` in binary. `PAGE_SIZE - 1` will be `111111111111`, but with `~`, we will get `000000000000`, but as we use `~PAGE_MASK` we will get `111111111111` again. On the second line we do the same but clear first 12 bits and getting page-aligned size of the area on the third line. We getting aligned area and now we need to get the number of pages which are occupied by the new `ioremap` are and calculate the fix-mapped index from `fixed_addresses` in the next steps:

```
nrpages = size >> PAGE_SHIFT;
idx = FIX_BTMAP_BEGIN - NR_FIX_BTMAPS*slot;
```

Now we can fill `fix-mapped` area with the given physical addresses. Every iteration in the loop, we call `__early_set_fixmap` function from the arch/x86/mm/ioremap.c, increase given physical address on page size which is `4096` bytes and update `addresses` index and number of pages:

```
while (nrpages > 0) {
    __early_set_fixmap(idx, phys_addr, prot);
    phys_addr += PAGE_SIZE;
    --idx;
    --nrpages;
}
```

The `__early_set_fixmap` function gets the page table entry (stored in the `bm_pte`, see above) for the given physical address with:

```
pte = early_ioremap_pte(addr);
```

In the next step of the `early_ioremap_pte` we check the given page flags with the `pgprot_val` macro and calls `set_pte` or `pte_clear` depends on it:

```
if (pgprot_val(flags))
        set_pte(pte, pfn_pte(phys >> PAGE_SHIFT, flags));
    else
        pte_clear(&init_mm, addr, pte);
```

As you can see above, we passed `FIXMAP_PAGE_IO` as flags to the `__early_ioremap`. `FIXMPA_PAGE_IO` expands to the:

```
(__PAGE_KERNEL_EXEC | _PAGE_NX)
```

flags, so we call `set_pte` function for setting page table entry which works in the same manner as `set_pmd` but for PTEs (read above about it). As we set all `PTEs` in the loop, we can see the call of the `__flush_tlb_one` function:

```
__flush_tlb_one(addr);
```

This function defined in the arch/x86/include/asm/tlbflush.h and calls `__flush_tlb_single` or `__flush_tlb` depends on value of the `cpu_has_invlpg`:

```
static inline void __flush_tlb_one(unsigned long addr)
{
        if (cpu_has_invlpg)
                __flush_tlb_single(addr);
        else
                __flush_tlb();
}
```

`__flush_tlb_one` function invalidates given address in the TLB. As you just saw we updated paging structure, but `TLB` not informed of changes, that's why we need to do it manually. There are two ways how to do it. First is update `cr3` control register and `__flush_tlb` function does this:

```
native_write_cr3(native_read_cr3());
```

The second method is to use `invlpg` instruction invalidates `TLB` entry. Let's look on `__flush_tlb_one` implementation. As you can see first of all it checks `cpu_has_invlpg` which defined as:

```
#if defined(CONFIG_X86_INVLPG) || defined(CONFIG_X86_64)
# define cpu_has_invlpg         1
#else
# define cpu_has_invlpg         (boot_cpu_data.x86 > 3)
#endif
```

If a CPU support `invlpg` instruction, we call the `__flush_tlb_single` macro which expands to the call of the `__native_flush_tlb_single`:

```
static inline void __native_flush_tlb_single(unsigned long addr)
{
        asm volatile("invlpg (%0)" ::"r" (addr) : "memory");
}
```

or call `__flush_tlb` which just updates `cr3` register as we saw it above. After this step execution of the `__early_set_fixmap` function is finsihed and we can back to the `__early_ioremap` implementation. As we set fixmap area for the given address, need to save the base virtual address of the I/O Re-mapped area in the `prev_map` with the `slot` index:

```
prev_map[slot] = (void __iomem *)(offset + slot_virt[slot]);
```

and return it.

The second function is - `early_iounmap` - unmaps an `I/O` memory region. This function takes two parameters: base address and size of a `I/O` region and generally looks very similar on `early_ioremap`. It also goes through fixmap slots and looks for slot with the given address. After this it gets the index of the fixmap slot and calls `__late_clear_fixmap` or `__early_set_fixmap` depends on `after_paging_init` value. It calls `__early_set_fixmap` with on difference then it does `early_ioremap`: it passes `zero` as physicall address. And in the end it sets address of the I/O memory region to `NULL`:

```
prev_map[slot] = NULL;
```

That's all about `fixmaps` and `ioremap`. Of course this part does not cover full features of the `ioremap`, it was only early ioremap, but there is also normal ioremap. But we need to know more things than now before it.

So, this is the end!

Conclusion

This is the end of the second part about linux kernel memory management. If you have questions or suggestions, ping me in twitter 0xAX, drop me email or just create issue.

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

- apic
- vsyscall
- Intel Trusted Execution Technology
- Xen
- Real Time Clock
- e820
- Memory management unit
- TLB
- Paging
- Linux kernel memory management Part 1.

Linux kernel concepts

This chapter describes various concepts which are used in the Linux kernel.

- Per-CPU variables
- CPU masks

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 parameters: `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 definition 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 parameter). 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 parameters. At first parameter 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 initialization 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.

CPU masks

Introduction

`Cpumasks` is a special way provided by the Linux kernel to store information about CPUs in the system. The relevant source code and header files which are contains API for `Cpumasks` manipulating:

- include/linux/cpumask.h
- lib/cpumask.c
- kernel/cpu.c

As comment says from the include/linux/cpumask.h: Cpumasks provide a bitmap suitable for representing the set of CPU's in a system, one bit position per CPU number. We already saw a bit about cpumask in the `boot_cpu_init` function from the Kernel entry point part. This function makes first boot cpu online, active and etc...:

```
set_cpu_online(cpu, true);
set_cpu_active(cpu, true);
set_cpu_present(cpu, true);
set_cpu_possible(cpu, true);
```

`set_cpu_possible` is a set of cpu ID's which can be plugged in anytime during the life of that system boot. `cpu_present` represents which CPUs are currently plugged in. `cpu_online` represents subset of the `cpu_present` and indicates CPUs which are available for scheduling. These masks depends on `CONFIG_HOTPLUG_CPU` configuration option and if this option is disabled `possible == present` and `active == online`. Implementation of the all of these functions are very similar. Every function checks the second parameter. If it is `true`, calls `cpumask_set_cpu` or `cpumask_clear_cpu` otherwise.

There are two ways for a `cpumask` creation. First is to use `cpumask_t`. It defined as:

```
typedef struct cpumask { DECLARE_BITMAP(bits, NR_CPUS); } cpumask_t;
```

It wraps `cpumask` structure which contains one bitmak `bits` field. `DECLARE_BITMAP` macro gets two parameters:

- bitmap name;
- number of bits.

and creates an array of `unsigned long` with the give name. It's implementation is pretty easy:

```
#define DECLARE_BITMAP(name,bits) \
        unsigned long name[BITS_TO_LONGS(bits)]
```

where `BITS_TO_LONG`:

```
#define BITS_TO_LONGS(nr)       DIV_ROUND_UP(nr, BITS_PER_BYTE * sizeof(long))
#define DIV_ROUND_UP(n,d) (((n) + (d) - 1) / (d))
```

As we learning `x86_64` architecture, `unsigned long` is 8-bytes size and our array will contain only one element:

```
(((8) + (8) - 1) / (8)) = 1
```

```
```

`NR_CPUS` macro presents the number of the CPUs in the system and depends on the `CONFIG_NR_CPUS` macro which defined in the include/linux/threads.h and looks like this:

```
#ifndef CONFIG_NR_CPUS
        #define CONFIG_NR_CPUS  1
#endif

#define NR_CPUS         CONFIG_NR_CPUS
```

The second way to define cpumask is to use `DECLARE_BITMAP` macro directly and `to_cpumask` macro which convertes given bitmap to the `struct cpumask *` :

```
#define to_cpumask(bitmap)                                              \
        ((struct cpumask *)(1 ? (bitmap)                                \
                            : (void *)sizeof(__check_is_bitmap(bitmap))))
```

We can see ternary operator operator here which is `true` every time. `__check_is_bitmap` inline function defined as:

```
static inline int __check_is_bitmap(const unsigned long *bitmap)
{
        return 1;
}
```

And returns `1` every time. We need in it here only for one purpose: In compile time it checks that given `bitmap` is a bitmap, or with another words it checks that given `bitmap` has type - `unsigned long *` . So we just pass `cpu_possible_bits` to the `to_cpumask` macro for converting array of `unsigned long` to the `struct cpumask *` .

cpumask API

As we can define cpumask with one of the method, Linux kernel provides API for manipulating a cpumask. Let's consider one of the function which presented above. For example `set_cpu_online` . This function takes two parameters:

- Number of CPU;
- CPU status;

Implementation of this function looks as:

```
void set_cpu_online(unsigned int cpu, bool online)
{
    if (online) {
        cpumask_set_cpu(cpu, to_cpumask(cpu_online_bits));
        cpumask_set_cpu(cpu, to_cpumask(cpu_active_bits));
    } else {
        cpumask_clear_cpu(cpu, to_cpumask(cpu_online_bits));
    }
}
```

First of all it checks the second `state` parameter and calls `cpumask_set_cpu` or `cpumask_clear_cpu` depends on it. Here we can see casting to the `struct cpumask *` of the second parameter in the `cpumask_set_cpu` . In our case it is `cpu_online_bits` which is bitmap and defined as:

```
static DECLARE_BITMAP(cpu_online_bits, CONFIG_NR_CPUS) __read_mostly;
```

`cpumask_set_cpu` function makes only one call of the `set_bit` function inside:

```
static inline void cpumask_set_cpu(unsigned int cpu, struct cpumask *dstp)
{
        set_bit(cpumask_check(cpu), cpumask_bits(dstp));
}
```

`set_bit` function takes two parameter too, and sets a given bit (first parameter) in the memory (second parameter or `cpu_online_bits` bitmap). We can see here that before `set_bit` will be called, its two parameter will be passed to the

- cpumask_check;
- cpumask_bits.

Let's consider these two macro. First if `cpumask_check` does nothing in our case and just returns given parameter. The second `cpumask_bits` just returns `bits` field from the given `struct cpumask *` structure:

```
#define cpumask_bits(maskp) ((maskp)->bits)
```

Now let's look on the `set_bit` implementation:

```
 static __always_inline void
 set_bit(long nr, volatile unsigned long *addr)
 {
         if (IS_IMMEDIATE(nr)) {
                asm volatile(LOCK_PREFIX "orb %1,%0"
                        : CONST_MASK_ADDR(nr, addr)
                        : "iq" ((u8)CONST_MASK(nr))
                        : "memory");
        } else {
                asm volatile(LOCK_PREFIX "bts %1,%0"
                        : BITOP_ADDR(addr) : "Ir" (nr) : "memory");
        }
 }
```

This function looks scarry, but it is not so hard as it seems. First of all it passes `nr` or number of the bit to the `IS_IMMEDIATE` macro which just makes call of the GCC internal `__builtin_constant_p` function:

```
#define IS_IMMEDIATE(nr)    (__builtin_constant_p(nr))
```

`__builtin_constant_p` checks that given parameter is known constant at compile-time. As our `cpu` is not compile-time constant, `else` clause will be executed:

```
asm volatile(LOCK_PREFIX "bts %1,%0" : BITOP_ADDR(addr) : "Ir" (nr) : "memory");
```

Let's try to understand how it works step by step:

`LOCK_PREFIX` is a x86 `lock` instruction. This instruction tells to the cpu to occupy the system bus while instruction will be executed. This allows to synchronize memory access, preventing simultaneous access of multiple processors (or devices - DMA controller for example) to one memory cell.

`BITOP_ADDR` casts given parameter to the `(*(volatile long *)` and adds `+m` constraints. `+` means that this operand is bot read and written by the instruction. `m` shows that this is memory operand. `BITOP_ADDR` is defined as:

```
#define BITOP_ADDR(x) "+m" (*(volatile long *) (x))
```

Next is the `memory` clobber. It tells the compiler that the assembly code performs memory reads or writes to items other than those listed in the input and output operands (for example, accessing the memory pointed to by one of the input parameters).

`Ir` - immideate register operand.

`bts` instruction sets given bit in a bit string and stores the value of a given bit in the `CF` flag. So we passed cpu number which is zero in our case and after `set_bit` will be executed, it sets zero bit in the `cpu_online_bits` cpumask. It would mean that the first cpu is online at this moment.

Besides the `set_cpu_*` API, cpumask ofcourse provides another API for cpumasks manipulation. Let's consider it in shoft.

Additional cpumask API

cpumask provides the set of macro for getting amount of the CPUs with different state. For example:

```
#define num_online_cpus()    cpumask_weight(cpu_online_mask)
```

This macro returns amount of the `online` CPUs. It calls `cpumask_weight` function with the `cpu_online_mask` bitmap (read about about it). `cpumask_wieght` function makes an one call of the `bitmap_wiegt` function with two parameters:

- cpumask bitmap;
- `nr_cpumask_bits` - which is `NR_CPUS` in our case.

```
static inline unsigned int cpumask_weight(const struct cpumask *srcp)
{
    return bitmap_weight(cpumask_bits(srcp), nr_cpumask_bits);
}
```

and calculates amount of the bits in the given bitmap. Besides the `num_online_cpus`, cpumask provides macros for the all CPU states:

- num_possible_cpus;
- num_active_cpus;
- cpu_online;
- cpu_possible.

and many more.

Besides that Linux kernel provides following API for the manipulating of `cpumask`:

- for_each_cpu - iterates over every cpu in a mask;
- for_each_cpu_not - iterates over every cpu in a complemented mask;
- cpumask_clear_cpu - clears a cpu in a cpumask;
- cpumask_test_cpu - tests a cpu in a mask;
- cpumask_setall - set all cpus in a mask;
- cpumask_size - returns size to allocate for a 'struct cpumask' in bytes;

and many many more...

Links

* cpumask documentation

Data Structures in the Linux Kernel

Linux kernel provides implementations of a different data structures like linked list, B+ tree, prinority heap and many many more.

This part considers these data structures and algorithms.

- Doubly linked list

Data Structures in the Linux Kernel

Doubly linked list

Linux kernel provides its own doubly linked list implementation which you can find in the include/linux/list.h. We will start `Data Structures in the Linux kernel` from the doubly linked list data structure. Why? Because it is very popular in the kernel, just try to search

First of all let's look on the main structure:

```
struct list_head {
    struct list_head *next, *prev;
};
```

You can note that it is different from many lists implementations which you could see. For example this doubly linked list structure from the glib:

```
struct GList {
  gpointer data;
  GList *next;
  GList *prev;
};
```

Usually a linked list structure contains a pointer to the item. Linux kernel implementation of the list does not. So the main question is - `where does the list store the data?`. The actual implementation of lists in the kernel is - `Intrusive list`. An intrusive linked list does not contain data in its nodes - A node just contains pointers to the next and previous node and list nodes part of the data that are added to the list. This makes the data structure generic, so it does not care about entry data type anymore.

For example:

```
struct nmi_desc {
    spinlock_t lock;
    struct list_head head;
};
```

Let's look at some examples, how `list_head` is used in the kernel. As I already wrote about, there are many, really many different places where lists are used in the kernel. Let's look for example in miscellaneous character drivers. Misc character drivers API from the drivers/char/misc.c for writing small drivers for handling simple hardware or virtual devices. This drivers share major number:

```
#define MISC_MAJOR              10
```

but has own minor number. For example you can see it with:

```
ls -l /dev |  grep 10
crw-------   1 root root     10, 235 Mar 21 12:01 autofs
drwxr-xr-x  10 root root         200 Mar 21 12:01 cpu
crw-------   1 root root     10,  62 Mar 21 12:01 cpu_dma_latency
crw-------   1 root root     10, 203 Mar 21 12:01 cuse
```

```
drwxr-xr-x   2 root root         100 Mar 21 12:01 dri
crw-rw-rw-   1 root root     10, 229 Mar 21 12:01 fuse
crw-------   1 root root     10, 228 Mar 21 12:01 hpet
crw-------   1 root root     10, 183 Mar 21 12:01 hwrng
crw-rw----+  1 root kvm      10, 232 Mar 21 12:01 kvm
crw-rw----   1 root disk     10, 237 Mar 21 12:01 loop-control
crw-------   1 root root     10, 227 Mar 21 12:01 mcelog
crw-------   1 root root     10,  59 Mar 21 12:01 memory_bandwidth
crw-------   1 root root     10,  61 Mar 21 12:01 network_latency
crw-------   1 root root     10,  60 Mar 21 12:01 network_throughput
crw-r-----   1 root kmem     10, 144 Mar 21 12:01 nvram
brw-rw----   1 root disk      1,  10 Mar 21 12:01 ram10
crw--w----   1 root tty       4,  10 Mar 21 12:01 tty10
crw-rw----   1 root dialout   4,  74 Mar 21 12:01 ttyS10
crw-------   1 root root     10,  63 Mar 21 12:01 vga_arbiter
crw-------   1 root root     10, 137 Mar 21 12:01 vhci
```

Now let's look how lists are used in the misc device drivers. First of all let's look on `miscdevice` structure:

```
struct miscdevice
{
      int minor;
      const char *name;
      const struct file_operations *fops;
      struct list_head list;
      struct device *parent;
      struct device *this_device;
      const char *nodename;
      mode_t mode;
};
```

We can see the fourth field in the `miscdevice` structure - `list` which is list of registered devices. In the beginning of the source code file we can see definition of the:

```
static LIST_HEAD(misc_list);
```

which expands to definition of the variables with `list_head` type:

```
#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)
```

and initializes it with the `LIST_HEAD_INIT` macro which set previous and next entries:

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }
```

Now let's look on the `misc_register` function which registers a miscellaneous device. At the start it initializes `miscdevice->list` with the `INIT_LIST_HEAD` function:

```
INIT_LIST_HEAD(&misc->list);
```

which does the same that `LIST_HEAD_INIT` macro:

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
```

```
}
```

In the next step after device created with the `device_create` function we add it to the miscellaneous devices list with:

```
list_add(&misc->list, &misc_list);
```

Kernel `list.h` provides this API for the addition of new entry to the list. Let's look on it's implementation:

```
static inline void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}
```

It just calls internal function `__list_add` with the 3 given parameters:

- new - new entry;
- head - list head after which will be inserted new item;
- head->next - next item after list head.

Implementation of the `__list_add` is pretty simple:

```
static inline void __list_add(struct list_head *new,
                  struct list_head *prev,
                  struct list_head *next)
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}
```

Here we set new item between `prev` and `next`. So `misc` list which we defined at the start with the `LIST_HEAD_INIT` macro will contain previous and next pointers to the `miscdevice->list`.

There is still only one question how to get list's entry. There is special special macro for this point:

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

which gets three parameters:

- ptr - the structure list_head pointer;
- type - structure type;
- member - the name of the list_head within the struct;

For example:

```
const struct miscdevice *p = list_entry(v, struct miscdevice, list)
```

After this we can access to the any `miscdevice` field with `p->minor` or `p->name` and etc... Let's look on the `list_entry` implementation:

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

As we can see it just calls `container_of` macro with the same arguments. For the first look `container_of` looks strange:

```
#define container_of(ptr, type, member) ({                      \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

First of all you can note that it consists from two expressions in curly brackets. Compiler will evaluate the whole block in the curly braces and use the value of the last expression.

For example:

```
#include <stdio.h>

int main() {
    int i = 0;
    printf("i = %d\n", ({++i; ++i;}));
    return 0;
}
```

will print `2`.

The next point is `typeof`, it's simple. As you can understand from its name, it just returns the type of the given variable. When I first saw the implementation of the `container_of` macro, the strangest thing for me was the zero in the `((type *)0)` expression. Actually this pointer magic calculates the offset of the given field from the address of the structure, but as we have `0` here, it will be just a zero offset alongwith the field width. Let's look at a simple example:

```
#include <stdio.h>

struct s {
        int field1;
        char field2;
        char field3;
};

int main() {
    printf("%p\n", &((struct s*)0)->field3);
    return 0;
}
```

will print `0x5`.

The next offsetof macro calculates offset from the beginning of the structure to the given structure's field. Its implementation is very similar to the previous code:

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

Let's summarize all about `container_of` macro. `container_of` macro returns address of the structure by the given address of the structure's field with `list_head` type, the name of the structure field with `list_head` type and type of the container structure. At the first line this macro declares the `__mptr` pointer which points to the field of the structure that `ptr` points to and assigns it to the `ptr`. Now `ptr` and `__mptr` point to the same address. Technically we don't need this line but its useful for type checking. First line ensures that that given structure (`type` parameter) has a member called `member`. In the second line it calculates offset of the field from the structure with the `offsetof` macro and subtracts it from the structure address. That's all.

Of course `list_add` and `list_entry` is not only functions which provides `<linux/list.h>`. Implementation of the doubly linked list provides the following API:

- list_add
- list_add_tail
- list_del
- list_replace
- list_move
- list_is_last
- list_empty
- list_cut_position
- list_splice

and many more.

Theory

This chapter describes various theoretical concepts and concepts which are not directly related to practice but useful to know.

- Paging
- Elf64 format

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 translation 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 mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed.

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 hierarchical and linux kernel uses 4 level paging for `x86_64`. 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](https://github.com/torvalds/linux/blob/master/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  |              |
 --------------------------------------------------------------------------------
```

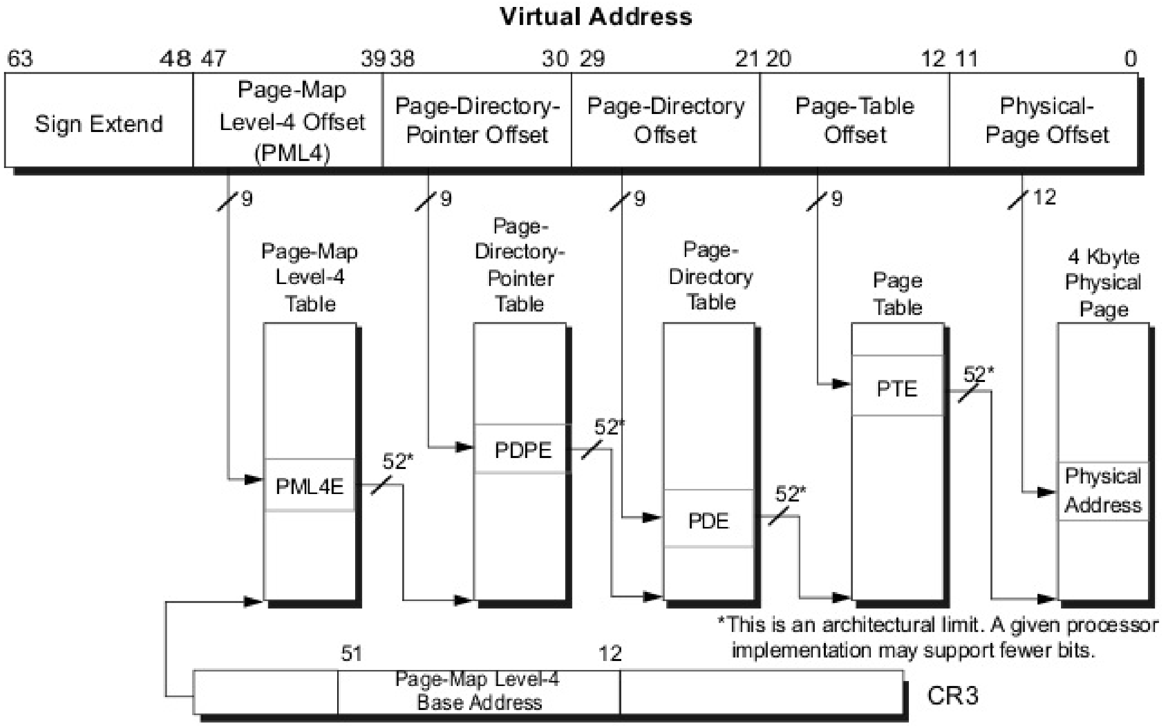
These fields have the following meanings:

- 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 translation 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/supervisor bit controls 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 definition of this guard 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:

```
1111111111111111 111111111 111111110 000001000 000000000 000000000000
      63:48        47:39     38:30     29:21     20:12      11:0
```

The given virtual address split 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 ahead in the kernel source code and see first initialization steps.

Conclusion

It's the end of this short part about paging theory. Of course 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, 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's 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, 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

Useful links

Linux boot

- Linux/x86 boot protocol
- Linux kernel parameters

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

Important data structures

- task_struct definition

Other architectures

- PowerPC and Linux Kernel Inside

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
- Fabio Pozzi
- Kevin Swinton
- Leandro Moreira
- LYF610400210
- Cam Cope
- Miquel Sabaté Solà
- Michael Aquilina
- Gabriel Sullice
- Michael Drüing
- Alexander Polakov
- Anton Davydov
- Arpan Kapoor
- Brandon Fosdick
- Ashleigh Newman-Jones
- Terrell Russell
- Mario
- Ewoud Kohl van Wijngaarden
- Jochen Maes
- Brother-Lal
- Brian McKenna
- Josh Triplett
- James Flowers
- Alexander Harding
- Dzmitry Plashchynski

- Simarpreet Singh
- umatomba