Published with GitBook

LINUX INSIDE

By OxAX

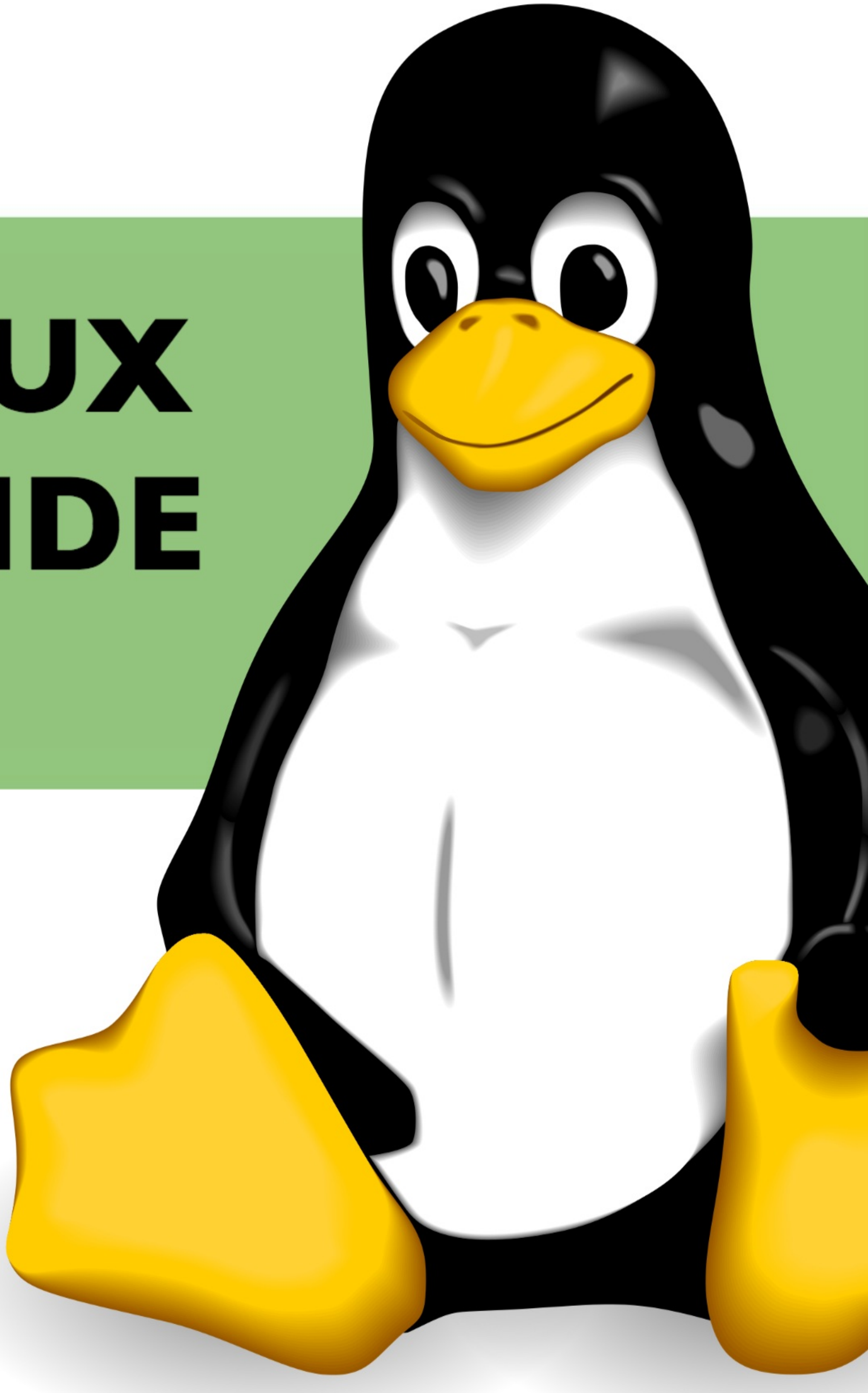


Table of Contents

linux-insides

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 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 an issue or just drop me an email.

Support

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

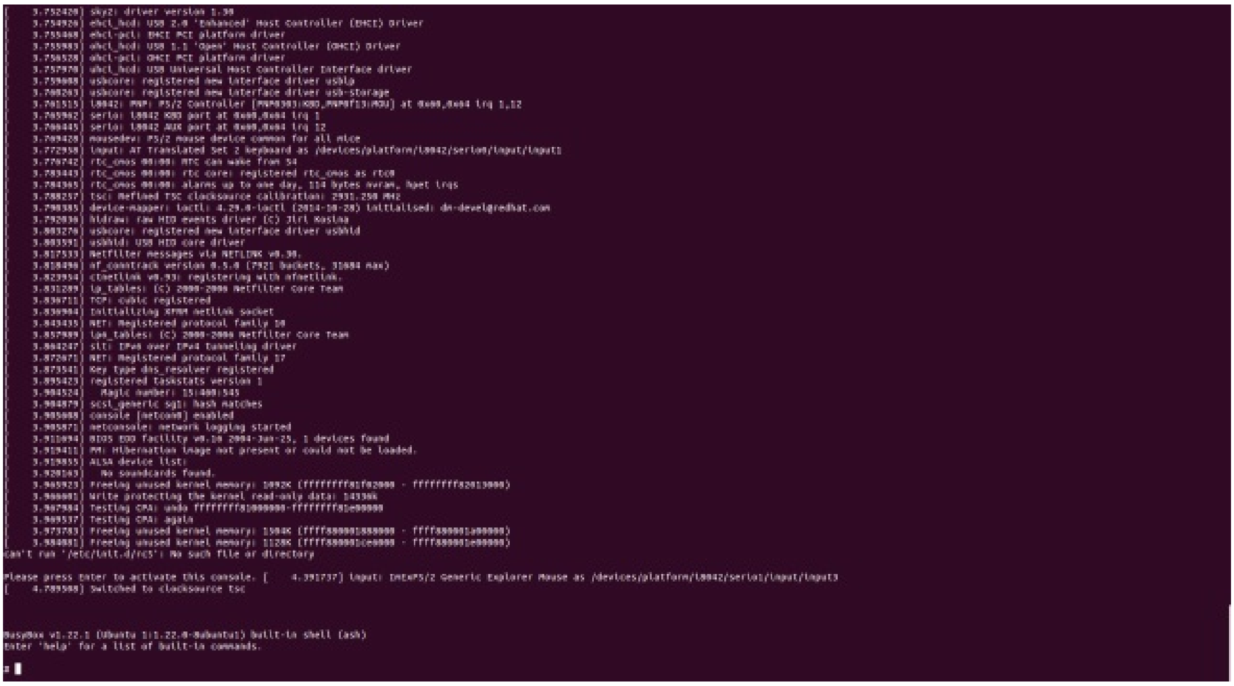
donate flattr | tips $2/week | donate bitcoin | donate gitbook

LICENSE

Contributions

Feel free to create issues or pull-requests if you have any problems.

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 sometime 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. I have a great interest in understanding 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. So, 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 the post content, feel free to send a pull request.

Note that this isn't the 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 the following posts. Ok, little introduction finished and now we can start to dive into the kernel and low-level stuff.

All code is actually for kernel - 3.18. If there are changes, I will update the posts accordingly.

The Magic Power Button, What happens 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 the 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 and we need to back up a little to understand 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 has 16-bit registers, and with 16-bit registers the maximum address is 2^16 or 0xffff (64 kilobytes). Memory segmentation is used to make use of all of the address space available. All memory is divided into small, fixed-size segments of 65535 bytes, or 64 KB. Since we cannot address memory below 64 KB with 16 bit registers, an alternate 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 if `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 the 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. The function of boot order is to control 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 is bootable. For example:

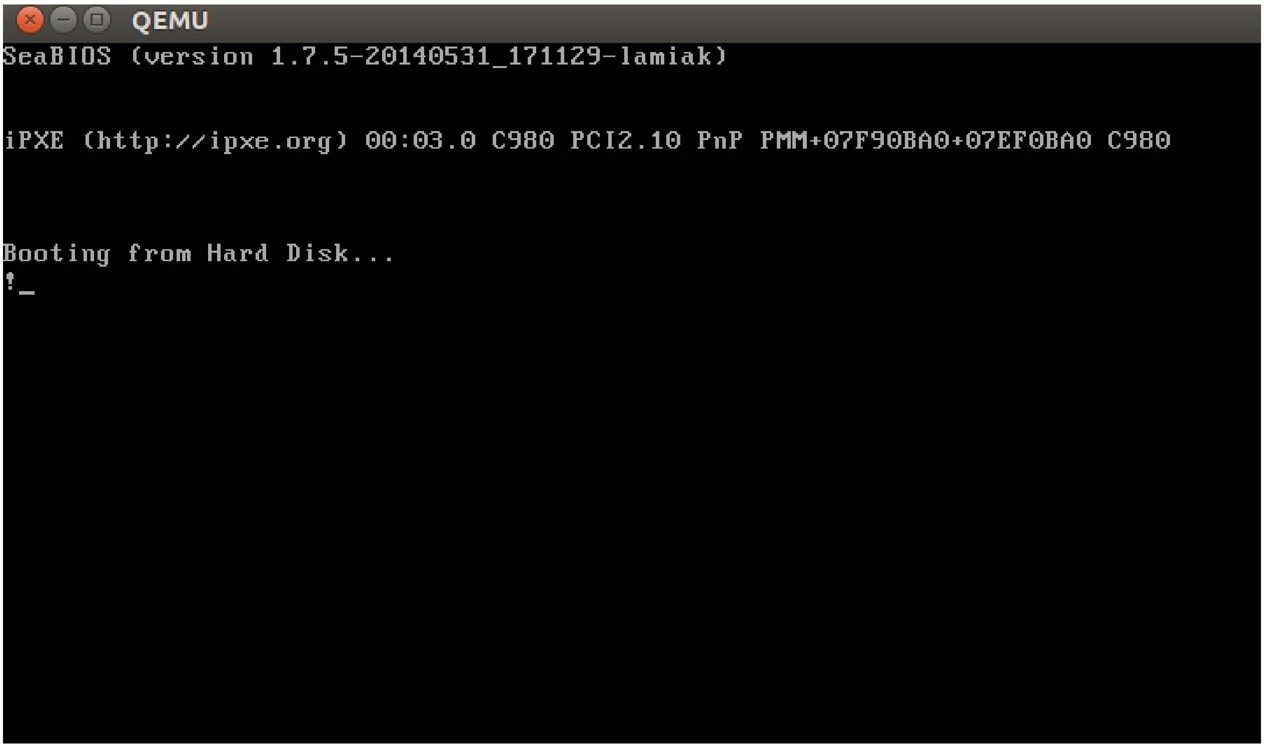
```
;
; Note: this example is written in Intel Assembly syntax
;
[BITS 16]
[ORG  0x7c00]

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(MBR) 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`.

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 point onwards BIOS hands over the 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 done as following:

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

Same as I mentioned before. 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 the result will be:

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

Where `0x10ffef` is equal to `1MB + 64KB - 16b`. But a 8086 processor, which was the first processor with real mode. It had 20 bit address line and `2^20 = 1048576.0` is 1MB. So, it means that the actual memory available 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 the address `0xFFFFFFF0`, which is much bigger than `0xFFFFF` (1MB). How can CPU access it in real mode? As I write about it 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 the 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 the selected 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 of kernel bootsector loaded. In my case `X` is `0x10000`, we can see it in memory dump:

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

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

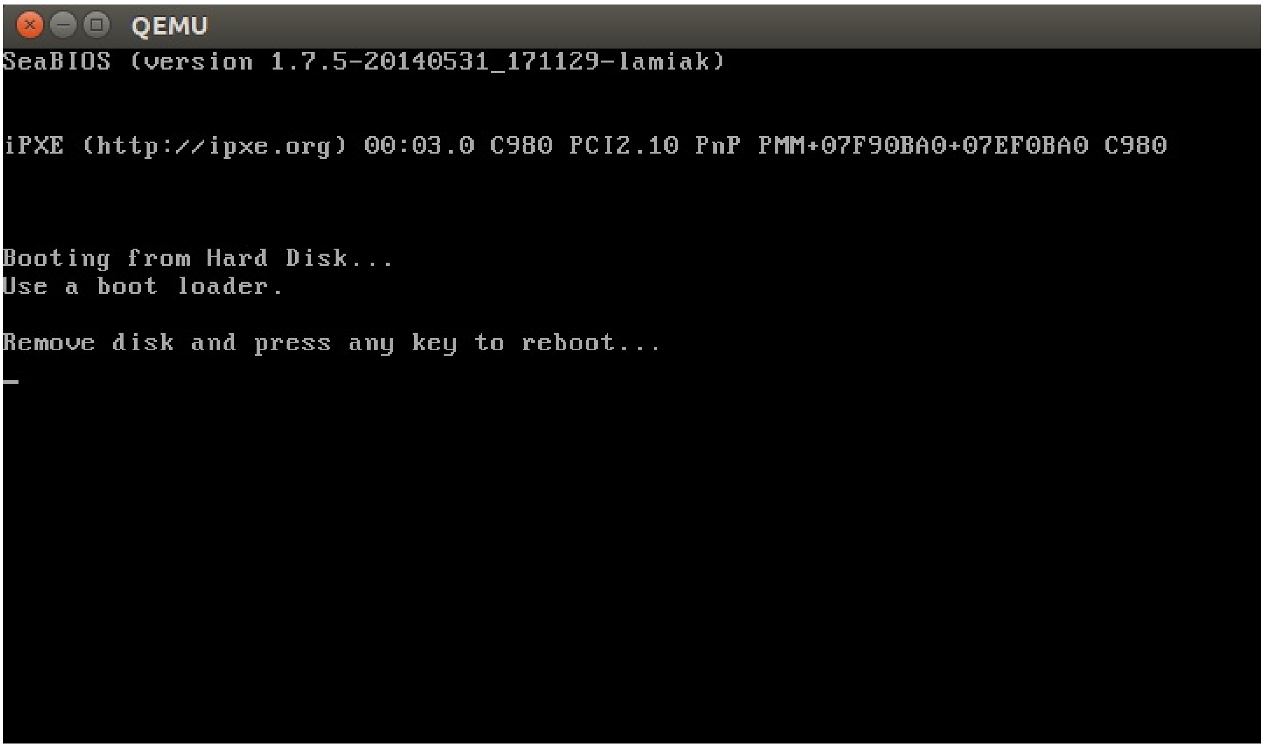
Start of Kernel Setup

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

Actually Long time ago Linux kernel 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 the operating system with UEFI. Here we will not see how it works (we will these later in the next parts).

So the 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 the 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`, it needs to do the following things:

- Be sure that all values of all segment registers are equal
- Setup correct stack if needed
- 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 disable interrupts with `cli` instruction:

```
    movw    %ds, %ax
    movw    %ax, %es
    cli
```

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 the 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 label `6` 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:

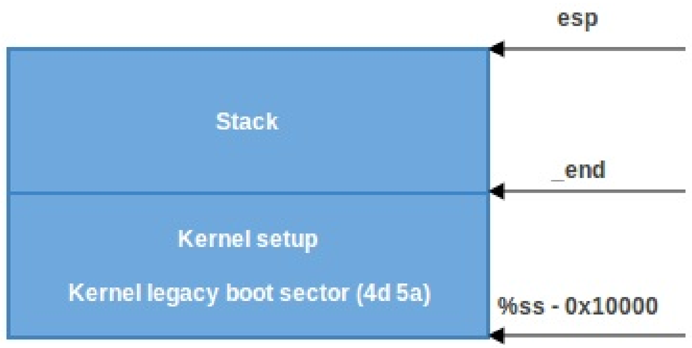
- `ss` has valid value 0x10000 (as all other segment registers beside `cs`)
- `ss` is invalid and `CAN_USE_HEAP` flag is set (see below)
- `ss` is invalid and `CAN_USE_HEAP` flag is not set (see below)

Let's look at all of these cases:

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

```
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:



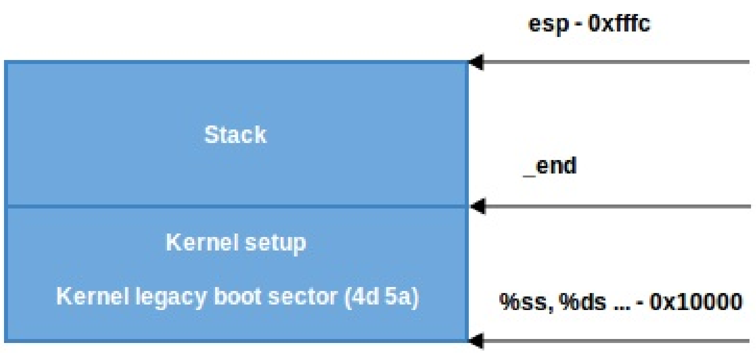
1. In the second case (`ss` != `ds`), first of all put _end (address of end of setup code) value in `dx` . And check `loadflags` header field with `testb` instruction too see if we can use heap or not. loadflags is a bitmask header which is defined as:

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

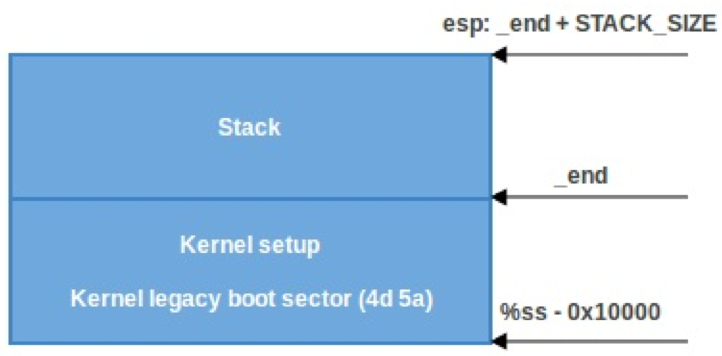
And as we can read in the boot protocol:

```
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 not be 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
```

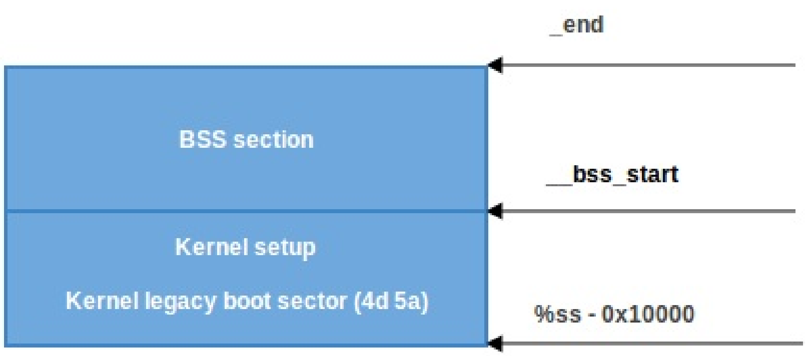
This simply consists of comparing the setup_sig against the magic number `0x5a5aaa55`. If they are not equal, a fatal error is reported.

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 the stack, BSS and now we can jump to the `main()` C function:

```
calll main
```

The `main()` function is located 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 to the `main` function (which is the first function written in C) from arch/x86/boot/main.c.

In this part we will continue to research the kernel setup code and

- see what `protected mode` is,
- some preparation for the transition into it,
- the heap and console initialization,
- memory detection, cpu validation, keyboard initialization
- 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 protected mode? Protected mode was first added to the x86 architecture in 1982 and was the main mode of Intel processors from the 80286 processor until Intel 64 and long mode came.

The main reason to move away from Real mode is that there is very limited access to the RAM. As you may remember from the previous part, there is only 2^{20} bytes or 1 Megabyte, sometimes even only 640 Kilobytes of RAM available in the Real mode.

Protected mode brought many changes, but the main one is the difference in memory management. The 20-bit address bus was replaced with a 32-bit address bus. It allowed access to 4 Gigabytes of memory vs 1 Megabyte of real mode. Also paging support was added, which you can read about in the next sections.

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

- Segmentation
- Paging

Here we will only see segmentation. Paging will be discussed in the next sections.

As you can read in the previous part, addresses consist of two parts in real mode:

- Base address of the 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 protected mode. There are no 64 Kilobyte fixed-size segments. Instead, the size and location of each segment is described by an associated data structure called *Segment Descriptor*. The segment descriptors are stored in a data structure called `Global Descriptor Table` (GDT).

The GDT is a structure which resides in memory. It has no fixed place in the memory so, 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(size) 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.

As mentioned above the GDT contains `segment descriptors` which describe memory segments. Each descriptor is 64-bits in size. The 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 it looks a little scary after real mode, but it's easy. For example LIMIT 15:0 means that bit 0-15 of the Descriptor contain the value for the limit. The rest of it is in LIMIT 16:19. So, the size of Limit is 0-19 i.e 20-bits. Let's take a closer look at it:

1. Limit[20-bits] is at 0-15,16-19 bits. It defines `length_of_segment - 1`. It depends on `G` (Granularity) bit.

   - if `G` (bit 55) is 0 and segment limit is 0, the size of the segment is 1 Byte
   - if `G` is 1 and segment limit is 0, the size of the segment is 4096 Bytes
   - if `G` is 0 and segment limit is 0xfffff, the size of the segment is 1 Megabyte
   - if `G` is 1 and segment limit is 0xfffff, the size of the segment is 4 Gigabytes

   So, it means that if

   - if G is 0, Limit is interpreted in terms of 1 Byte and the maximum size of the segment can be 1 Megabyte.
   - if G is 1, Limit is interpreted in terms of 4096 Bytes = 4 KBytes = 1 Page and the maximum size of the segment can be 4 Gigabytes. Actually when G is 1, the value of Limit is shifted to the left by 12 bits. So, 20 bits + 12 bits = 32 bits and 2^{32} = 4 Gigabytes.
2. Base[32-bits] is at (0-15, 32-39 and 56-63 bits). It defines the physical address of the segment's starting location.

3. Type/Attribute (40-47 bits) defines the type of segment and kinds of access to it.

   - `S` flag at bit 44 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).

To determine if the segment is a code or data segment we can check its Ex(bit 43) Attribute marked as 0 in the above diagram. If it is 0, then the segment is a Data segment otherwise it is a code segment.

A segment can be of one of the following 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(bit 43) is `0` for a *data* segment and `1` for a *code* segment. The next three bits(40, 41, 42, 43) are either `EWA` (*E*xpansion *W*ritable *A*ccessible) or CRA(*C*onforming *R*eadable *A*ccessible).

- if E(bit 42) is 0, expand up other wise expand down. Read more here.
- if W(bit 41)(for Data Segments) is 1, write access is allowed otherwise not. Note that read access is always allowed on data segments.
- A(bit 40) - Whether the segment is accessed by processor or not.
- C(bit 43) is conforming bit(for code selectors). If C is 1, the segment code can be executed from a lower level privilege for e.g user level. If C is 0, it can only be executed from the same privilege level.
- R(bit 41)(for code segments). If 1 read access to segment is allowed otherwise not. Write access is never allowed to code segments.

1. DPL[2-bits] (Descriptor Privilege Level) is at bits 45-46. It defines the privilege level of the segment. It can be 0-3 where 0 is the most privileged.

2. P flag(bit 47) - indicates if the segment is present in memory or not. If P is 0, the segment will be presented as *invalid* and the processor will refuse to read this segment.

3. AVL flag(bit 52) - Available and reserved bits. It is ignored in Linux.

4. L flag(bit 53) - indicates whether a code segment contains native 64-bit code. If 1 then the code segment executes in 64 bit mode.

5. D/B flag(bit 54) - Default/Big flag represents the operand size i.e 16/32 bits. If it is set then 32 bit otherwise 16.

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

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

Where,

- **Index** shows the index number of the descriptor in the GDT.
- **TI**(Table Indicator) shows where to search for the descriptor. If it is 0 then search in the Global Descriptor Table(GDT) otherwise it will look in Local Descriptor Table(LDT).

* And **RPL** is Requester's Privilege Level.

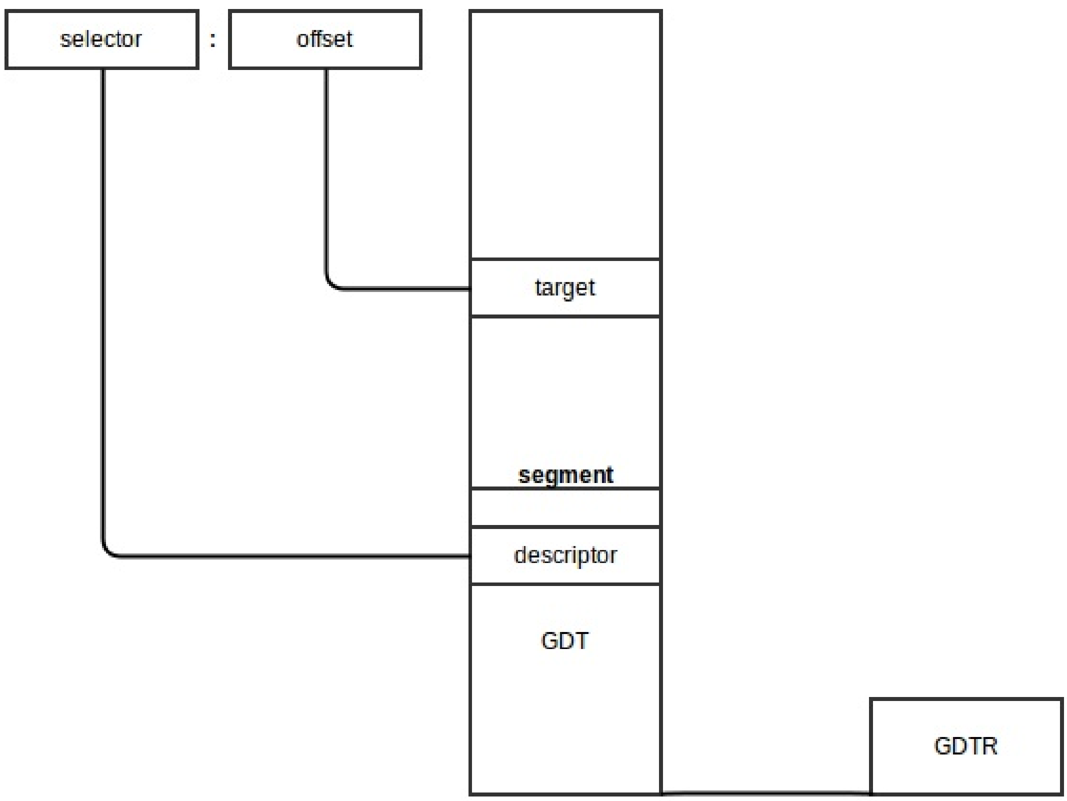
Every segment register has a visible and hidden part.

* Visible - Segment Selector is stored here
* Hidden - Segment Descriptor(base, limit, attributes, flags)

The following steps are needed to get the physical address in the protected mode:

* The segment selector must be loaded in one of the segment registers
* The CPU tries to find a segment descriptor by GDT address + Index from selector and load the descriptor into the *hidden* part of the 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 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 complete transition to protected mode in the linux kernel in the next part, but before we can move to protected mode, we need to do some more preparations.

Let's look at arch/x86/boot/main.c. We can see some routines there which perform keyboard initialization, heap initialization, etc... Let's take a look.

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(void)`. 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:

1. Copies `hdr` from header.S to the `boot_params` structure in `setup_header` field

2. Updates pointer to the kernel command line if the kernel was loaded with the 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 `memcpy` is easy. At first, it pushes values from `si` and `di` registers to the stack because their values will change during the `memcpy`, so it pushes them on the stack to preserve their values. `memcpy` (and other functions in copy.S) use `fastcall` calling conventions. So it gets its 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` in bytes
- `dx` will contain the address of `hdr` in bytes
- `cx` will contain the size of `hdr` in bytes.

`memcpy` puts the address of `boot_params.hdr` into `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 this we restore the size of `hdr` again, align it by 4 bytes and copy the rest of the bytes from `si` to `di` byte by byte (if there is more). 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");
```

The definition of `puts` is in tty.c. As we can see it prints character by character in a 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 at 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 the `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` in the 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:

```
heap_end = (char *)((size_t)boot_params.hdr.heap_end_ptr + 0x200);
```

which means `heap_end_ptr` or `_end` + `512` (`0x200h`). And at the last is checked that whether `heap_end` is greater than `stack_end` . If it is then `stack_end` is assigned to `heap_end` to make them equal.

Now the heap is initialized and we can use it using the `GET_HEAP` method. We will see how it is used, how to use it and how the it is implemented in the 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 the kernel launches on the right cpu level.

```
check_cpu(&cpu_level, &req_level, &err_flags);
    if (cpu_level < req_level) {
    ...
    return -1;
    }
```

`check_cpu` checks the cpu's flags, presence of long mode in case of x86_64(64-bit) CPU, checks the processor's vendor and makes preparation for certain vendors like turning off SSE+SSE2 for AMD if they are missing, etc.

Memory detection

The next step is memory detection by the `detect_memory` function. `detect_memory` basically provides a map of available RAM to the cpu. 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, the `detect_memory_e820` function initializes the `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;
```

- `ax` contains 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
- `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.

```
    initregs(&ireg);
    ireg.ah = 0x02;        /* Get keyboard status */
    intcall(0x16, &ireg, &oreg);
    boot_params.kbd_status = oreg.al;
```

After this it calls 0x16 again to set repeat rate and delay.

```
    ireg.ax = 0x0305;    /* Set keyboard repeat rate */
    intcall(0x16, &ireg, NULL);
```

Querying

The next couple of steps are queries for different parameters. We will not dive into details about these queries, but will get back to it in later parts. Let's take a short look at these 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, the BIOS doesn't support (**MCA**)[https://en.wikipedia.org/wiki/Micro_Channel_architecture]. 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));
}
```

This function contains 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 etc...

At 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 step is getting Intel SpeedStep information by calling the `query_ist` function. First of all it checks the 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 connecting the 32-bit 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 queries `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 `0x80` is the first hard drive and the value of `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` containing the address of the buffer where EDD information 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 a PR to linux-internals.

Links

- Protected mode
- Protected mode
- Long mode
- Nice explanation of CPU Modes with code
- How to Use Expand Down Segments on Intel 386 and Later CPUs
- earlyprintk documentation
- Kernel Parameters
- Serial console
- Intel SpeedStep
- APM
- EDD specification

* TLDP documentation for Linux Boot Process (old)
* 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. In this part, we will see:

- video mode initialization in the kernel setup code,
- preparation before switching into the protected mode,
- transition to protected mode

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 by first getting the video mode from the `boot_params.hdr` structure:

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

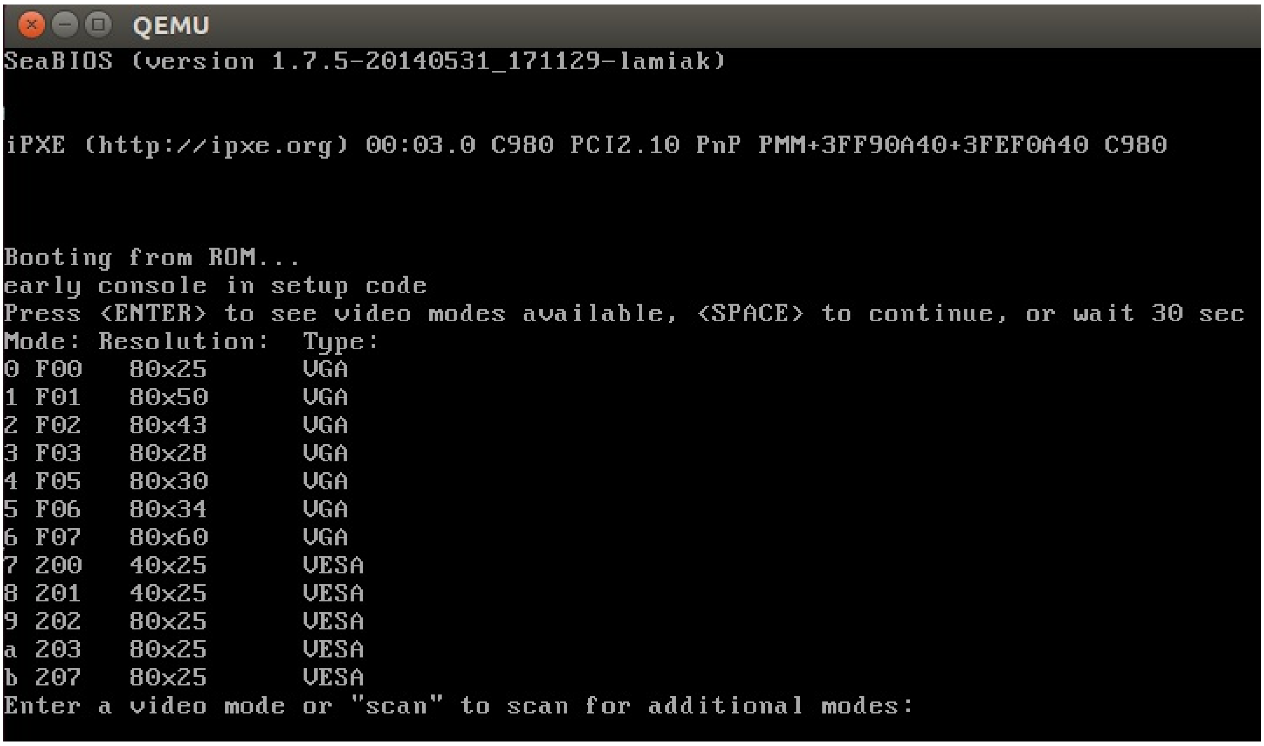
which we filled in the `copy_boot_params` function (you can read about it in the previous post). `vid_mode` is an obligatory field which is filled by the bootloader. You can find information about it in the kernel boot protocol:

```
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 mentioned in the description, for example it can be an integer number `0xFFFD` or `ask`. If you pass `ask` t `vga`, you will see a menu like this:



which will ask to select a video mode. We will look at it's implementation, but before diving into the implementation we have to look at some other things.

Kernel data types

Earlier we saw definitions of different data types like `u16` 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 read source code of the kernel, you'll see these very often and so it will be good to remember them.

Heap API

After we have `vid_mode` from the `boot_params.hdr` in the `set_video` function we can see call to `RESET_HEAP` function. `RESET_HEAP` is a macro which defined in the boot.h. It is defined as:

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

If you have read the second part, you will remember that we initialized the heap with the `init_heap` function. We have a couple of utility functions for heap which are defined in `boot.h`. They are:

```
#define RESET_HEAP()
```

As we saw just above it resets 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
- `__alignof__(type)` shows how type of variable is aligned
- `n` tells 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 its usage, something like:

```
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 is 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. Next is the call to `store_mode_params` which stores video mode parameters in the `boot_params.screen_info` structure which is defined in the include/uapi/linux/screen_info.h.

If we will look at `store_mode_params` function, we can see that it starts with the call to `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 sets the `video_segment` . After the BIOS transfers control to the boot sector, the following addresses are for 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 font size needs to be stored 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 is located at address `0x485` (this memory location is used to get the font size) and save font size in the `boot_params.screen_info.orig_video_points` .

```
 x = rdfs16(0x44a);
 y = (adapter == ADAPTER_CGA) ? 25 : rdfs8(0x484)+1;
```

Next we get amount of columns by `0x44a` and rows by address `0x484` and store them 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.

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 etc. and stores it in the `saved_screen` structure, which is defined as:

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

It then checks whether the 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 in 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 `probe_cards` execution is finished, we move to the main loop in the `set_video` function. There is infinite loop which tries to setup video mode with the `set_mode` function or prints a 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 the 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 i.e. `card->set_mode(struct mode_info*)`. We can get access to this function from the `card_info` structure, every video mode defines this structure with values filled depending upon the 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 the vga mode and calls the respective function:

```
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 calls `0x10` BIOS interrupt with certain value in the `AH` register.

After we have set video mode, we pass it to the `boot_params.hdr.vid_mode`.

Next `vesa_store_edid` is called. This function simply stores the EDID (**E**xtended **D**isplay **I**dentification **D**ata) information for kernel use. After this `store_mode_params` is called again. Lastly, if `do_restore` is set, screen is restored to an earlier state.

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 the comment says: `Do the last things and invoke protected mode`, so let's see these last things 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.

First is the call to `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. You can read more about hooks 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 interrupts. After `realmode_switch` hook (it isn't present for me) is checked, disabling of Non-Maskable Interrupts(NMI) occurs:

```
asm volatile("cli");
outb(0x80, 0x70);    /* Disable NMI */
io_delay();
```

At first there is inline assembly instruction with `cli` instruction which clears the interrupt flag (`IF`). After this, external interrupts are disabled. Next line disables NMI (non-maskable interrupt).

Interrupt is a signal to the CPU which is emitted by hardware or software. After getting signal, CPU suspends current instructions sequence, saves its state and transfers control to the interrupt handler. After interrupt handler has finished it's work, it transfers control to the interrupted instruction. Non-maskable interrupts (NMI) are interrupts which are always processed, independently of permission. It cannot be ignored and is typically used to signal for non-recoverable hardware errors. We will not dive into details of interrupts now, but will discuss it in the next posts.

Let's get back to the code. We can see that second line is writing `0x80` (disabled bit) byte to the `0x70` (CMOS Address register). After that call to the `io_delay` function occurs. `io_delay` causes a 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. So we can write any value (value from `AL` register in our case) to the `0x80` port. After this delay `realmode_switch_hook` function has finished execution and we can move to the next function.

The next function is `enable_a20`, which enables A20 line. This function is 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, then delay for 1ms, and then read the 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 enable it with a different method which you can find in the `a20.c`. For example with call of `0x15` BIOS interrupt with `AH=0x2041` etc.

If `enabled_a20` function finished with fail, print an error message and call 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 is successfully enabled, `reset_coprocessor` function is called:

```
outb(0, 0xf0);
outb(0, 0xf1);
```

This function clears the Math Coprocessor by writing `0` to `0xf0` and then resets it by writing `0` to `0xf1`.

After this `mask_all_interrupts` function is called:

```
outb(0xff, 0xa1);       /* Mask all interrupts on the secondary PIC */
outb(0xfb, 0x21);       /* Mask all but cascade on the primary PIC */
```

This masks all interrupts on the secondary PIC (Programmable Interrupt Controller) and primary PIC except for IRQ2 on the primary PIC.

And after all of these preparations, we can see actual transition into protected mode.

Setup Interrupt Descriptor Table

Now we setup the 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 the 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 as defined as:

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

where we can see - 16-bit length(`len`) 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 might remember from the previous post that it is 48-bits in size).

Setup Global Descriptor Table

Next is the setup of Global Descriptor Table (GDT). We can see `setup_gdt` function which sets up 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 note that it has `__attribute__((aligned(16)))` attribute. It means that this structure will be aligned by 16 bytes. Let's look at a 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 read 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 left-shifted by 4 bits (remember we're in the real mode now).

Lastly 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 is 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` in the `esi` register and address of code segment register `cs` (0x1000) in the `bx`. After this we shift `bx` by 4 bits 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 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
- `__BOOT_CS` is the code segment.

After this we are finally in the protected mode:

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

Let's look at 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 stop at it's entry point. What happens next, 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 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 find any mistakes, please send me a PR with corrections at 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 stopped right before jump on 64-bit entry point - `startup_64` which located in the arch/x86/boot/compressed/head_64.S source code file. We already saw the 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 beginning of the `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.

Note That there will not be description of the all kernel initialization steps. Here will be only generic kernel part, without interrupts handling, ACPI, and many other parts. All parts which I'll miss, will be described in other chapters.

- 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.
- Scheduler initialization - describes preparation before scheduler initialization and initialization of it.
- RCU initialization - describes the initialization of the RCU.
- End of the initialization - the last part about linux kernel initialization.

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 then 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't be run on the default address, but now we know the 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't be run at the default `0x1000000` address. `rbp` register contains actual address so we add to the `early_level4_pgt`, `level3_kernel_pgt` and `level2_fixmap_pgt`. Let's try to understand what these 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 ran 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 are generally used for system calls;
- Hardware interrupts - when a hardware event happens, for example button is 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 a 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 `32` 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:

```
    .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. 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 to 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 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 the processes that 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 the first `init` process. Before the first process will be started, the `start_kernel` must do many things such 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 much kernel code awaits us, let's start.

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

A little about function attributes

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

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

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

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

where `no_instrument_function` says to the compiler not to 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`. You can find the definition of this and other macro attributes in include/linux/init.h.

First steps in the start_kernel

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

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

The first represents a pointer to the kernel command line and the second will contain the 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 the details related with these two variables at this time, but will see it in the next parts. In the next step we can see a call to the:

```
lockdep_init();
```

function. `lockdep_init` initializes lock validator. Its implementation is pretty simple, it just initializes two list_head hashes and sets the `lockdep_initialized` global variable to `1`. Lock validator detects circular lock dependencies and is 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` represents the initial task structure:

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

where `task_struct` stores all the information about a process. I will not explain this structure in this book because it's very big. You can find its definition in include/linux/sched.h. At this moment `task_struct` contains more than `100` fields! Although you will not see the explanation 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 meet them in practice.

You can see the definition of the `init_task` and it initialized by the `INIT_TASK` macro. This macro is from 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 its own stack and it is 16 killobytes or 4 page frames. in `x86_64`. We can note that it is 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. The `thread_info` structure contains architecture-specific information on the thread. We know that on `x86_64` the 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. The 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 the `INIT_TASK` macro fills these `task_struct's` fields and many many more. As I already wrote about, I will not describe all the fields and values in the `INIT_TASK` macro but we will see them soon.

Now let's go 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 simple. `set_task_stack_end_magic` gets the end of the stack for the given `task_struct` with the `end_of_stack` function. The end of a process stack depends on the `CONFIG_STACK_GROWSUP` configuration option. As we learn in `x86_64` architecture, the 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 the end of the init process stack, we write `STACK_END_MAGIC` there. After `canary` is 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 an empty body for `x86_64`:

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

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

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

After the `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 the `CONFIG_CC_STACKPROTECTOR` configuration option and if this option is disabled, `boot_init_stack_canary` does nothing, otherwise it generates random numbers 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 the `stack_canary` field of `task_struct` with it:

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

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

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

Again, we will not dive into details here, we will cover it in the part about IRQs. As canary is 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 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 the current CPU with the given ID in the CPU bitmap.

The first processor activation

The current function from the `start_kernel` is `boot_cpu_init`. This function initializes various CPU masks for the bootstrap processor. First of all it gets the bootstrap processor id with a call to:

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

For now it is just zero. If the `CONFIG_DEBUG_PREEMPT` configuration option is disabled, `smp_processor_id` just expands to the call of `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 optimizing access to the per-cpu variables which are associated with the current processor. In our case it is `this_cpu_read`:

```
__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 at the `__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 looks a little strange but it's easy. First of all we can see the 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 verify that the given parameter is a 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 in 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. The `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 as:

```
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 the 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 at any time 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 depend on the `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`, it calls `cpumask_set_cpu` or `cpumask_clear_cpu` otherwise.

For example let's look at `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 the `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, the `DECLARE_BITMAP` macro expands to the array of `unsigned long`. Now let's look at how the `to_cpumask` macro is implemented:

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

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

I don't know about you, but it looked really weird for me at the first time. We can see a ternary operator here which is `true` every time, but why the `__check_is_bitmap` here? It's simple, let's look at 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: at compile time it checks that the given `bitmap` is a bitmap, or in other words it checks that the given `bitmap` has a type of `unsigned long *`. So we just pass `cpu_possible_bits` to the `to_cpumask` macro for converting the 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 the special part about it - cpumask or documentation.

As we activated the bootstrap processor, it's 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. At this moment we use `pr_notice` to print the Linux banner:

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

which is just the 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. The Linux kernel does it with the call of the `setup_arch` function. This is a very big function like `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. The `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 memory block.

We can get the base physical address of the `_text` symbol 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. The `__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 the base address of `_text`:

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

After we got the physical address of the `_text` symbol, `memblock_reserve` can reserve a 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 reserving 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. The `early_reserve_initrd` function does all work. First of all this function gets 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 are taken from `boot_params`. If you have read the chapter about Linux Kernel Booting Process, you must remember that we filled the `boot_params` structure during boot time. The kernel 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 the information that interests us from `boot_params`. For example let's look at `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 the 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 a 64-bit address in `ramdisk_image` and we 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 the 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 the 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 a 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) = .;
```

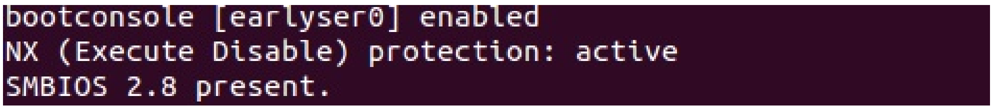
Now we know how parameters are defined, let's back to the `parse_early_param` implementation:

```
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

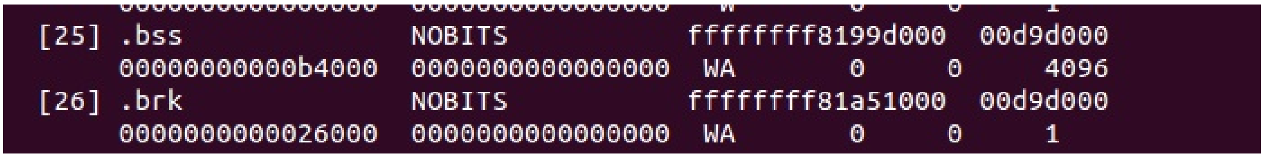
In the next step of the `setup_arch` we can see the call of the `early_alloc_pgt_buf` function which allocates the page table buffer for early stage. The page table buffer will be place in the `brk` area. Let's look on its implementation:

```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](http://en.wikipedia.org/wiki/Non-uniform_memory_access) 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`:

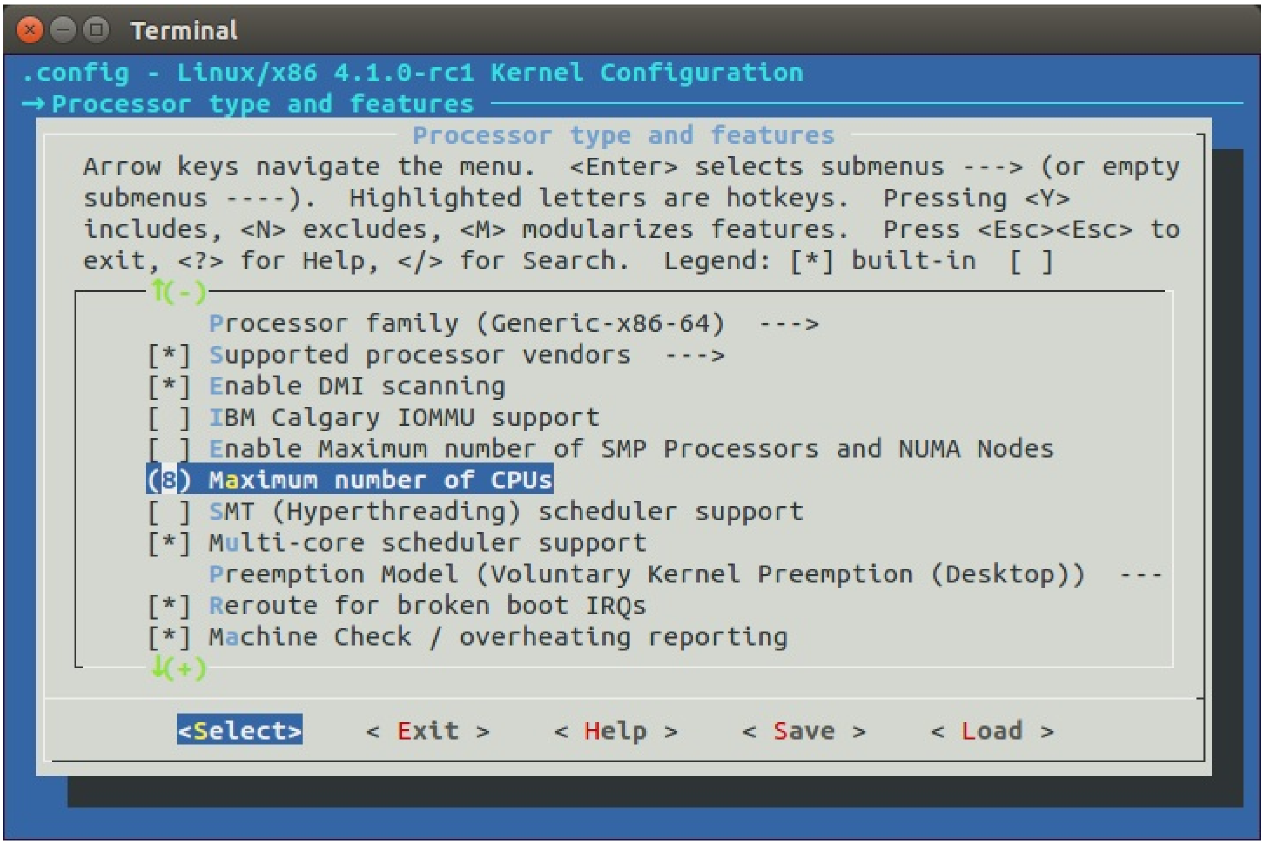
```
static void __init setup_command_line(char *command_line)
{
        saved_command_line =
                memblock_virt_alloc(strlen(boot_command_line) + 1, 0);
        initcall_command_line =
                memblock_virt_alloc(strlen(boot_command_line) + 1, 0);
        static_command_line = memblock_virt_alloc(strlen(command_line)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

Kernel initialization. Part 8.

Scheduler initialization

This is the eighth part of the Linux kernel initialization process and we stopped on the `setup_nr_cpu_ids` function in the previous part. The main point of the current part is scheduler initialization. But before we will start to learn initialization process of the scheduler, we need to do some stuff. The next step in the init/main.c is the `setup_per_cpu_areas` function. This function setups areas for the `percpu` variables, more about it you can read in the special part about the Per-CPU variables. After `percpu` areas up and running, the next step is the `smp_prepare_boot_cpu` function. This function does some preparations for the SMP:

```
static inline void smp_prepare_boot_cpu(void)
{
         smp_ops.smp_prepare_boot_cpu();
}
```

where the `smp_prepare_boot_cpu` expands to the call of the `native_smp_prepare_boot_cpu` function (more about `smp_ops` will be in the special parts about `SMP`):

```
void __init native_smp_prepare_boot_cpu(void)
{
        int me = smp_processor_id();
        switch_to_new_gdt(me);
        cpumask_set_cpu(me, cpu_callout_mask);
        per_cpu(cpu_state, me) = CPU_ONLINE;
}
```

The `native_smp_prepare_boot_cpu` function gets the number of the current CPU (which is Bootstrap processor and its `id` is zero) with the `smp_processor_id` function. I will not explain how the `smp_processor_id` works, because we alread saw it in the Kernel entry point part. As we got processor `id` number we reload Global Descriptor Table for the given CPU with the `switch_to_new_gdt` function:

```
void switch_to_new_gdt(int cpu)
{
        struct desc_ptr gdt_descr;

        gdt_descr.address = (long)get_cpu_gdt_table(cpu);
        gdt_descr.size = GDT_SIZE - 1;
        load_gdt(&gdt_descr);
        load_percpu_segment(cpu);
}
```

The `gdt_descr` variable represents pointer to the `GDT` descriptor here (we already saw `desc_ptr` in the Early interrupt and exception handling). We get the address and the size of the `GDT` descriptor where `GDT_SIZE` is `256` or:

```
#define GDT_SIZE (GDT_ENTRIES * 8)
```

and the address of the descriptor we will get with the `get_cpu_gdt_table`:

```
static inline struct desc_struct *get_cpu_gdt_table(unsigned int cpu)
{
```

```
        return per_cpu(gdt_page, cpu).gdt;
}
```

The `get_cpu_gdt_table` uses `per_cpu` macro for getting `gdt_page` percpu variable for the given CPU number (bootstrap processor with `id` - 0 in our case). You can ask the following question: so, if we can access `gdt_page` percpu variable, where it was defined? Actually we alread saw it in this book. If you have read the first part of this chapter, you can remember that we saw definition of the `gdt_page` in the arch/x86/kernel/head_64.S:

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

and if we will look on the linker file we can see that it locates after the `__per_cpu_load` symbol:

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

and filled `gdt_page` in the arch/x86/kernel/cpu/common.c:

```
DEFINE_PER_CPU_PAGE_ALIGNED(struct gdt_page, gdt_page) = { .gdt = {
#ifdef CONFIG_X86_64
    [GDT_ENTRY_KERNEL32_CS]        = GDT_ENTRY_INIT(0xc09b, 0, 0xfffff),
    [GDT_ENTRY_KERNEL_CS]        = GDT_ENTRY_INIT(0xa09b, 0, 0xfffff),
    [GDT_ENTRY_KERNEL_DS]        = GDT_ENTRY_INIT(0xc093, 0, 0xfffff),
    [GDT_ENTRY_DEFAULT_USER32_CS]    = GDT_ENTRY_INIT(0xc0fb, 0, 0xfffff),
    [GDT_ENTRY_DEFAULT_USER_DS]    = GDT_ENTRY_INIT(0xc0f3, 0, 0xfffff),
    [GDT_ENTRY_DEFAULT_USER_CS]    = GDT_ENTRY_INIT(0xa0fb, 0, 0xfffff),
    ...
    ...
    ...
```

more about `percpu` variables you can read in the Per-CPU variables part. As we got address and size of the `GDT` descriptor we case reload `GDT` with the `load_gdt` which just execute `lgdt` instruct and load `percpu_segment` with the following function:

```
void load_percpu_segment(int cpu) {
    loadsegment(gs, 0);
    wrmsrl(MSR_GS_BASE, (unsigned long)per_cpu(irq_stack_union.gs_base, cpu));
    load_stack_canary_segment();
}
```

The base address of the `percpu` area must contain `gs` register (or `fs` register for `x86`), so we are using `loadsegment` macro and pass `gs` . In the next step we writes the base address if the IRQ stack and setup stack canary (this is only for `x86_32`). After we load new `GDT` , we fill `cpu_callout_mask` bitmap with the current cpu and set cpu state as online with the setting `cpu_state` percpu variable for the current processor - `CPU_ONLINE` :

```
cpumask_set_cpu(me, cpu_callout_mask);
per_cpu(cpu_state, me) = CPU_ONLINE;
```

So, what is it `cpu_callout_mask` bitmap... As we initialized bootstrap processor (procesoor which is booted the first on `x86`) the other processors in a multiprocessor system are known as `secondary processors` . Linux kernel uses two following bitmasks:

- `cpu_callout_mask`
- `cpu_callin_mask`

After bootstrap processor initialized, it updates the `cpu_callout_mask` to indicate which secondary processor can be initialized next. All other or secondary processors can do some initialization stuff before and check the `cpu_callout_mask` on the boostrap processor bit. Only after the bootstrap processor filled the `cpu_callout_mask` this secondary processor, it will continue the rest of its initialization. After that the certain processor will finish its initialization process, the processor sets bit in the `cpu_callin_mask`. Once the bootstrap processor finds the bit in the `cpu_callin_mask` for the current secondary processor, this processor repeats the same procedure for initialization of the rest of a secondary processors. In a short words it works as i described, but more details we will see in the chapter about `SMP`.

That's all. We did all `SMP` boot preparation.

Build zonelists

In the next step we can see the call of the `build_all_zonelists` function. This function sets up the order of zones that allocations are preferred from. What are zones and what's order we will understand now. For the start let's see how linux kernel considers physical memory. Physical memory may be arranged into banks which are called - `nodes`. If you has no hardware with support for `NUMA`, you will see only one node:

```
$ cat /sys/devices/system/node/node0/numastat
numa_hit 72452442
numa_miss 0
numa_foreign 0
interleave_hit 12925
local_node 72452442
other_node 0
```

Every `node` presented by the `struct pglist data` in the linux kernel. Each node devided into a number of special blocks which are called - `zones`. Every zone presented by the `zone struct` in the linux kernel and has one of the type:

- `ZONE_DMA` - 0-16M;
- `ZONE_DMA32` - used for 32 bit devices that can only do DMA areas below 4G;
- `ZONE_NORMAL` - all RAM from the 4GB on the `x86_64`;
- `ZONE_HIGHMEM` - absent on the `x86_64`;
- `ZONE_MOVABLE` - zone which contains movable pages.

which are presented by the `zone_type` enum. Information about zones we can get with the:

```
$ cat /proc/zoneinfo
Node 0, zone      DMA
  pages free     3975
        min      3
        low      3
        ...
        ...
Node 0, zone    DMA32
  pages free     694163
        min      875
        low      1093
        ...
        ...
Node 0, zone   Normal
  pages free     2529995
        min      3146
        low      3932
        ...
        ...
```

As I wrote above all nodes are described with the `pglist_data` or `pg_data_t` structure in memory. This structure defined in the include/linux/mmzone.h. The `build_all_zonelists` function from the mm/page_alloc.c constructs an ordered `zonelist` (of different zones `DMA`, `DMA32`, `NORMAL`, `HIGH_MEMORY`, `MOVABLE`) which specifies the zones/nodes to visit when a selected `zone` or `node` cannot satisfy the allocation request. That's all. More about `NUMA` and multiprocessor systems will be in the special part.

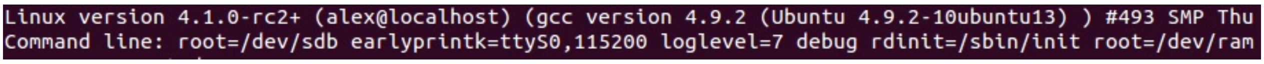
The rest of the stuff before scheduler initialization

Before we will start to dive into linux kernel scheduler initialization process we must to do a couple of things. The fisrt thing is the `page_alloc_init` function from the mm/page_alloc.c. This function looks pretty easy:

```
void __init page_alloc_init(void)
{
        hotcpu_notifier(page_alloc_cpu_notify, 0);
}
```

and initializes handler for the `CPU` hotplug. Of course the `hotcpu_notifier` depends on the `CONFIG_HOTPLUG_CPU` configuration option and if this option is set, it just calls `cpu_notifier` macro which expands to the call of the `register_cpu_notifier` which adds hotplug cpu handler (`page_alloc_cpu_notify` in our case).

After this we can see the kernel command line in the initialization output:



And a couple of functions as `parse_early_param` and `parse_args` which are handles linux kernel command line. You can remember that we already saw the call of the `parse_early_param` function in the sixth part of the kernel initialization chapter, so why we call it again? Answer is simple: we call this function in the architecture-specific code (`x86_64` in our case), but not all architecture calls this function. And we need in the call of the second function `parse_args` to parse and handle non-early command line arguments.

In the next step we can see the call of the `jump_label_init` from the kernel/jump_label.c. and initializes jump label.

After this we can see the call of the `setup_log_buf` function which setups the printk log buffer. We already saw this function in the seventh part of the linux kernel initialization process chapter.

PID hash initialization

The next is `pidhash_init` function. As you know an each process has assigned unique number which called - `process identification number` or `PID`. Each process generated with fork or clone is automatically assigned a new unique `PID` value by the kernel. The management of `PIDs` centered around the two special data structures: `struct pid` and `struct upid`. First structure represents information about a `PID` in the kernel. The second structure represents the information that is visible in a specific namespace. All `PID` instances stored in the special hash table:

```
static struct hlist_head *pid_hash;
```

This hash table is used to find the pid instance that belongs to a numeric `PID` value. So, `pidhash_init` initializes this hash. In the start of the `pidhash_init` function we can see the call of the `alloc_large_system_hash`:

```
pid_hash = alloc_large_system_hash("PID", sizeof(*pid_hash), 0, 18,
                                   HASH_EARLY | HASH_SMALL,
                                   &pidhash_shift, NULL,
```

```
                                    0, 4096);
```

The number of elements of the `pid_hash` depends on the `RAM` configuration, but it can be between `2^4` and `2^12`. The `pidhash_init` computes the size and allocates the required storage (which is `hlist` in our case - the same as doubly linked list, but contains one pointer instead on the struct hlist_head]. The `alloc_large_system_hash` function allocates a large system hash table with `memblock_virt_alloc_nopanic` if we pass `HASH_EARLY` flag (as it in our case) or with `__vmalloc` if we did no pass this flag.

The result we can see in the `dmesg` output:

```
$ dmesg | grep hash
[    0.000000] PID hash table entries: 4096 (order: 3, 32768 bytes)
...
...
...
```

That's all. The rest of the stuff before scheduler initialization is the following functions: `vfs_caches_init_early` does early initialization of the virtual file system (more about it will be in the chapter which will describe virtual file system), `sort_main_extable` sorts the kernel's built-in exception table entries which are between `__start___ex_table` and `__stop___ex_table,`, and `trap_init` initializies trap handlers (morea about last two function we will know in the separate chapter about interrupts).

The last step before the scheduler initialization is initialization of the memory manager with the `mm_init` function from the init/main.c. As we can see, the `mm_init` function initializes different part of the linux kernel memory manager:

```
page_ext_init_flatmem();
mem_init();
kmem_cache_init();
percpu_init_late();
pgtable_init();
vmalloc_init();
```

The first is `page_ext_init_flatmem` depends on the `CONFIG_SPARSEMEM` kernel configuration option and initializes extended data per page handling. The `mem_init` releases all `bootmem`, the `kmem_cache_init` initializes kernel cache, the `percpu_init_late` - replaces `percpu` chunks with those allocated by slub, the `pgtable_init` - initilizes the `vmalloc_init` - initializes `vmalloc`. Please, **NOTE** that we will not dive into details about all of these functions and concepts, but we will see all of they it in the Linux kernem memory manager chapter.

That's all. Now we can look on the `scheduler`.

Scheduler initialization

And now we came to the main purpose of this part - initialization of the task scheduler. I want to say again as I did it already many times, you will not see the full explanation of the scheduler here, there will be special chapter about this. Ok, next point is the `sched_init` function from the kernel/sched/core.c and as we can understand from the function's name, it initializes scheduler. Let's start to dive in this function and try to understand how the scheduler initialized. At the start of the `sched_init` function we can see the following code:

```
#ifdef CONFIG_FAIR_GROUP_SCHED
         alloc_size += 2 * nr_cpu_ids * sizeof(void **);
#endif
#ifdef CONFIG_RT_GROUP_SCHED
         alloc_size += 2 * nr_cpu_ids * sizeof(void **);
#endif
```

First of all we can see two configuration options here:

* `CONFIG_FAIR_GROUP_SCHED`
* `CONFIG_RT_GROUP_SCHED`

Both of this options provide two different planning models. As we can read from the documentation, the current scheduler - `CFS` or `Completely Fair Scheduler` used a simple concept. It models process scheduling as if the system had an ideal multitasking processor where each process would receive `1/n` processor time, where `n` is the number of the runnable processes. The scheduler uses the special set of rules used. These rules determine when and how to select a new process to run and they are called `scheduling policy`. The Completely Fair Scheduler supports following `normal` or `non-real-time` scheduling policies: `SCHED_NORMAL`, `SCHED_BATCH` and `SCHED_IDLE`. The `SCHED_NORMAL` is used for the most normal applications, the amount of cpu each process consumes is mostly determined by the nice value, the `SCHED_BATCH` used for the 100% non-interactive tasks and the `SCHED_IDLE` runs tasks only when the processor has not to run anything besides this task. The `real-time` policies are also supported for the time-critial applications: `SCHED_FIFO` and `SCHED_RR`. If you've read something about the Linux kernel scheduler, you can know that it is modular. It means that it supports different algorithms to schedule different types of processes. Usually this modularity is called `scheduler classes`. These modules encapsulate scheduling policy details and are handled by the scheduler core without the core code assuming too much about them.

Now let's back to the our code and look on the two configuration options `CONFIG_FAIR_GROUP_SCHED` and `CONFIG_RT_GROUP_SCHED`. The scheduler operates on an individual task. These options allows to schedule group tasks (more about it you can read in the CFS group scheduling). We can see that we assign the `alloc_size` variables which represent size based on amount of the processors to allocate for the `sched_entity` and `cfs_rq` to the `2 * nr_cpu_ids * sizeof(void **)` expression with `kzalloc`:

```
ptr = (unsigned long)kzalloc(alloc_size, GFP_NOWAIT);

#ifdef CONFIG_FAIR_GROUP_SCHED
        root_task_group.se = (struct sched_entity **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);

        root_task_group.cfs_rq = (struct cfs_rq **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);
#endif
```

The `sched_entity` is struture which defined in the include/linux/sched.h and used by the scheduler to keep track of process accounting. The `cfs_rq` presents run queue. So, you can see that we allocated space with size `alloc_size` for the run queue and scheduler entity of the `root_task_group`. The `root_task_group` is an instance of the `task_group` structure from the kernel/sched/sched.h which contains task group related information:

```
struct task_group {
    ...
    ...
    struct sched_entity **se;
    struct cfs_rq **cfs_rq;
    ...
    ...
}
```

The root task group is the task group which belongs every task in system. As we allocated space for the root task group scheduler entity and runqueue, we go over all possible CPUs (`cpu_possible_mask` bitmap) and allocate zeroed memory from a particular memory node with the `kzalloc_node` function for the `load_balance_mask` `percpu` variable:

```
DECLARE_PER_CPU(cpumask_var_t, load_balance_mask);
```

Here `cpumask_var_t` is the `cpumask_t` with one difference: `cpumask_var_t` is allocated only `nr_cpu_ids` bits when the `cpumask_t` always has `NR_CPUS` bits (more about `cpumask` you can read in the CPU masks part). As you can see:

```
#ifdef CONFIG_CPUMASK_OFFSTACK
    for_each_possible_cpu(i) {
        per_cpu(load_balance_mask, i) = (cpumask_var_t)kzalloc_node(
                cpumask_size(), GFP_KERNEL, cpu_to_node(i));
    }
#endif
```

this code depends on the `CONFIG_CPUMASK_OFFSTACK` configuration option. This configuration options says to use dynamic allocation for `cpumask`, instead of putting it on the stack. All groups have to be able to rely on the amount of CPU time. With the call of the two following functions:

```
init_rt_bandwidth(&def_rt_bandwidth,
                  global_rt_period(), global_rt_runtime());
init_dl_bandwidth(&def_dl_bandwidth,
                  global_rt_period(), global_rt_runtime());
```

we initialize bandwidth management for the `SCHED_DEADLINE` real-time tasks. These functions initializes `rt_bandwidth` and `dl_bandwidth` structures which are store information about maximum `deadline` bandwith of the system. For example, let's look on the implementation of the `init_rt_bandwidth` function:

```
void init_rt_bandwidth(struct rt_bandwidth *rt_b, u64 period, u64 runtime)
{
        rt_b->rt_period = ns_to_ktime(period);
        rt_b->rt_runtime = runtime;

        raw_spin_lock_init(&rt_b->rt_runtime_lock);

        hrtimer_init(&rt_b->rt_period_timer,
                     CLOCK_MONOTONIC, HRTIMER_MODE_REL);
        rt_b->rt_period_timer.function = sched_rt_period_timer;
}
```

It takes three parameters:

- address of the `rt_bandwidth` structure which contains information about the allocated and consumed quota within a period;
- `period` - period over which real-time task bandwidth enforcement is measured in `us`;
- `runtime` - part of the period that we allow tasks to run in `us`.

As `period` and `runtime` we pass result of the `global_rt_period` and `global_rt_runtime` functions. Which are `1s` second and and `0.95s` by default. The `rt_bandwidth` structure defined in the kernel/sched/sched.h and looks:

```
struct rt_bandwidth {
        raw_spinlock_t          rt_runtime_lock;
        ktime_t                 rt_period;
        u64                     rt_runtime;
        struct hrtimer          rt_period_timer;
};
```

As you can see, it contains `runtime` and `period` and also two following fields:

- `rt_runtime_lock` - spinlock for the `rt_time` protection;
- `rt_period_timer` - high-resolution kernel timer for unthrottled of real-time tasks.

So, in the `init_rt_bandwidth` we initialize `rt_bandwidth` period and runtime with the given parameters, initialize the spinlock and high-resolution time. In the next step, depends on the enabled SMP, we make initialization of the root domain:

```
#ifdef CONFIG_SMP
    init_defrootdomain();
#endif
```

The real-time scheduler requires global resources to make scheduling decision. But unfortenatelly scalability bottlenecks appear as the number of CPUs increase. The concept of root domains was introduced for improving scalability. The linux kernel provides special mechanism for assigning a set of CPUs and memory nodes to a set of task and it is called - `cpuset`. If a `cpuset` contains non-overlapping with other `cpuset` CPUs, it is `exclusive cpuset`. Each exclusive cpuset defines an isolated domain or `root domain` of CPUs partitioned from other cpusets or CPUs. A `root domain` presented by the `struct root_domain` from the kernel/sched/sched.h in the linux kernel and its main purpose is to narrow the scope of the global variables to per-domain variables and all real-time scheduling decisions are made only within the scope of a root domain. That's all about it, but we will see more details about it in the chapter about scheduling about real-time scheduler.

After `root domain` initialization, we make initialization of the bandwidth for the real-time tasks of the root task group as we did it above:

```
#ifdef CONFIG_RT_GROUP_SCHED
    init_rt_bandwidth(&root_task_group.rt_bandwidth,
            global_rt_period(), global_rt_runtime());
#endif
```

In the next step, depends on the `CONFIG_CGROUP_SCHED` kernel configuration option we initialze the `siblings` and `children` lists of the root task group. As we can read from the documentation, the `CONFIG_CGROUP_SCHED` is:

```
This option allows you to create arbitrary task groups using the "cgroup" pseudo
filesystem and control the cpu bandwidth allocated to each such task group.
```

As we finished with the lists initialization, we can see the call of the `autogroup_init` function:

```
#ifdef CONFIG_CGROUP_SCHED
        list_add(&root_task_group.list, &task_groups);
        INIT_LIST_HEAD(&root_task_group.children);
        INIT_LIST_HEAD(&root_task_group.siblings);
        autogroup_init(&init_task);
#endif
```

which initializes automatic process group scheduling.

After this we are going through the all `possible` cpu (you can remember that `possible` CPUs store in the `cpu_possible_mask` bitmap of possible CPUs that can ever be available in the system) and initialize a `runqueue` for each possible cpu:

```
for_each_possible_cpu(i) {
    struct rq *rq;
    ...
    ...
    ...
```

Each processor has its own locking and individual runqueue. All runnalble tasks are stored in an active array and indexed according to its priority. When a process consumes its time slice, it is moved to an expired array. All of these arras are stored in the special structure which names is `runqueu`. As there are no global lock and runqueu, we are going through the all possible CPUs and initialize runqueue for the every cpu. The `runque` is presented by the `rq` structure in the linux kernel which defined in the kernel/sched/sched.h.

```
rq = cpu_rq(i);
raw_spin_lock_init(&rq->lock);
rq->nr_running = 0;
rq->calc_load_active = 0;
rq->calc_load_update = jiffies + LOAD_FREQ;
init_cfs_rq(&rq->cfs);
init_rt_rq(&rq->rt);
init_dl_rq(&rq->dl);
rq->rt.rt_runtime = def_rt_bandwidth.rt_runtime;
```

Here we get the runque for the every CPU with the `cpu_rq` macto which returns `runqueues` percpu variable and start to initialize it with runqueu lock, number of running tasks, `calc_load` relative fields (`calc_load_active` and `calc_load_update`) which are used in the reckoning of a CPU load and initialization of the completely fair, real-time and deadline related fields in a runqueue. After this we initialize `cpu_load` array with zeros and set the last load update tick to the `jiffies` variable which determines the number of time ticks (cycles), since the system boot:

```
for (j = 0; j < CPU_LOAD_IDX_MAX; j++)
    rq->cpu_load[j] = 0;

rq->last_load_update_tick = jiffies;
```

where `cpu_load` keeps history of runqueue loads in the past, for now `CPU_LOAD_IDX_MAX` is 5. In the next step we fill `runqueue` fields which are related to the SMP, but we will not cover they in this part. And in the end of the loop we initialize high-resolution timer for the give `runqueue` and set the `iowait` (more about it in the separate part about scheduler) number:

```
init_rq_hrtick(rq);
atomic_set(&rq->nr_iowait, 0);
```

Now we came out from the `for_each_possible_cpu` loop and the next we need to set load weight for the `init` task with the `set_load_weight` function. Weight of process is calculated through its dynamic priority which is static priority + scheduling class of the process. After this we increase memory usage counter of the memory descriptor of the `init` process and set scheduler class for the current process:

```
atomic_inc(&init_mm.mm_count);
current->sched_class = &fair_sched_class;
```

And make current process (it will be the first `init` process) `idle` and update the value of the `calc_load_update` with the 5 seconds interval:

```
init_idle(current, smp_processor_id());
calc_load_update = jiffies + LOAD_FREQ;
```

So, the `init` process will be run, when there will be no other candidates (as it is the first process in the system). In the end we just set `scheduler_running` variable:

```
scheduler_running = 1;
```

That's all. Linux kernel scheduler is initialized. Of course, we missed many different details and explanations here, because we need to know and understand how different concepts (like process and process groups, runqueue, rcu and etc...) works in the linux kernel , but we took a short look on the scheduler initialization process. All other details we will look in the separate part which will be fully dedicated to the scheduler.

Conclusion

It is the end of the eighth part about the linux kernel initialization process. In this part, we looked on the initialization process of the scheduler and we will continue in the next part to dive in the linux kernel initialization process and will see initialization of the RCU and many more.

and other initialization stuff in the next part.

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

- CPU masks
- high-resolution kernel timer
- spinlock
- Run queue
- Linux kernem memory manager
- slub
- virtual file system
- Linux kernel hotplug documentation
- IRQ
- Global Descriptor Table
- Per-CPU variables
- SMP
- RCU
- CFS Scheduler documentation
- Real-Time group scheduling
- Previous part

Kernel initialization. Part 9.

RCU initialization

This is ninth part of the Linux Kernel initialization process and in the previous part we stopped at the scheduler initialization. In this part we will continue to dive to the linux kernel initialization process and the main purpose of this part will be to learn about initialization of the RCU. We can see that the next step in the init/main.c after the `sched_init` is the call of the `preempt_disablepreempt_disable`. There are two macros:

- `preempt_disable`
- `preempt_enable`

for preemption disabling and enabling. First of all let's try to understand what is it `preempt` in the context of an operating system kernel. In a simple words, preemption is ability of the operating system kernel to preempt current task to run task with higher priority. Here we need to disable preemption because we will have only one `init` process for the early boot time and we no need to stop it before we will call `cpu_idle` function. The `preempt_disable` macro defined in the include/linux/preempt.h and depends on the `CONFIG_PREEMPT_COUNT` kernel configuration option. This maco implemeted as:

```
#define preempt_disable() \
do { \
        preempt_count_inc(); \
        barrier(); \
} while (0)
```

and if `CONFIG_PREEMPT_COUNT` is not set just:

```
#define preempt_disable()                       barrier()
```

Let's look on it. First of all we can see one difference between these macro implementations. The `preempt_disable` with `CONFIG_PREEMPT_COUNT` contains the call of the `preempt_count_inc`. There is special `percpu` variable which stores the number of held locks and `preempt_disable` calls:

```
DECLARE_PER_CPU(int, __preempt_count);
```

In the first implementation of the `preempt_disable` we increment this `__preempt_count`. There is API for returning value of the `__preempt_count`, it is the `preempt_count` function. As we called `preempt_disable`, first of all we increment preemption counter with the `preempt_count_inc` macro which expands to the:

```
#define preempt_count_inc() preempt_count_add(1)
#define preempt_count_add(val)  __preempt_count_add(val)
```

where `preempt_count_add` calls the `raw_cpu_add_4` macro which adds `1` to the given `percpu` variable (`__preempt_count`) in our case (more about `precpu` variables you can read in the part about Per-CPU variables). Ok, we increased `__preempt_count` and th next step we can see the call of the `barrier` macro in the both macros. The `barrier` macro inserts an optimization barrier. In the processors with `x86_64` architecture independent memory access operations can be performed in any order. That's why we need in the oportunity to point compiler and processor on compliance of order. This mechanism is memory barrier. Let's consider simple example:

```
preempt_disable();
foo();
preempt_enable();
```

Compiler can rearrange it as:

```
preempt_disable();
preempt_enable();
foo();
```

In this case non-preemptible function `foo` can be preempted. As we put `barrier` macro in the `preempt_disable` and `preempt_enable` macros, it prevents the compiler from swapping `preempt_count_inc` with other statements. More about barriers you can read here and here.

In the next step we can see following statement:

```
if (WARN(!irqs_disabled(),
     "Interrupts were enabled *very* early, fixing it\n"))
    local_irq_disable();
```

which check IRQs state, and disabling (with `cli` instruction for `x86_64`) if they are enabled.

That's all. Preemption is disabled and we can go ahead.

Initialization of the integer ID management

In the next step we can see the call of the `idr_init_cache` function which defined in the lib/idr.c. The `idr` library used in a various places in the linux kernel to manage assigning integer `IDs` to objects and looking up objects by id.

Let's look on the implementation of the `idr_init_cache` function:

```
void __init idr_init_cache(void)
{
        idr_layer_cache = kmem_cache_create("idr_layer_cache",
                                sizeof(struct idr_layer), 0, SLAB_PANIC, NULL);
}
```

Here we can see the call of the `kmem_cache_create`. We already called the `kmem_cache_init` in the init/main.c. This function create generalized caches again using the `kmem_cache_alloc` (more about caches we will see in the Linux kernel memory management chapter). In our case, as we are using `kmem_cache_t` it will be used the slab allocator and `kmem_cache_create` creates it. As you can seee we pass five parameters to the `kmem_cache_create`:

- name of the cache;
- size of the object to store in cache;
- offset of the first object in the page;
- flags;
- constructor for the objects.

and it will create `kmem_cache` for the integer IDs. Integer `IDs` is commonly used pattern for the to map set of integer IDs to the set of pointers. We can see usage of the integer IDs for example in the i2c drivers subsystem. For example drivers/i2c/i2c-core.c which presentes the core of the `i2c` subsystem defines `ID` for the `i2c` adapter with the `DEFINE_IDR` macro:

```
static DEFINE_IDR(i2c_adapter_idr);
```

and than it uses it for the declaration of the `i2c` adapter:

```
static int __i2c_add_numbered_adapter(struct i2c_adapter *adap)
{
  int     id;
  ...
  ...
  ...
  id = idr_alloc(&i2c_adapter_idr, adap, adap->nr, adap->nr + 1, GFP_KERNEL);
  ...
  ...
  ...
}
```

and `id2_adapter_idr` presents dynamically calculated bus number.

More about integer ID management you can read here.

RCU initialization

The next step is RCU initialization with the `rcu_init` function and it's implementation depends on two kernel configuration options:

- `CONFIG_TINY_RCU`
- `CONFIG_TREE_RCU`

In the first case `rcu_init` will be in the kernel/rcu/tiny.c and in the second case it will be defined in the kernel/rcu/tree.c. We will see the implementation of the `tree rcu`, but first of all about the `RCU` in general.

`RCU` or read-copy update is a scalable high-performance synchronization mechanism implemented in the Linux kernel. On the early stage the linux kernel provided support and environment for the concurently running applications, but all execution was serialized in the kernel using a single global lock. In our days linux kernel has no single global lock, but provides different mechanisms including lock-free data structures, percpu data structures and other. One of these mechanisms is - the `read-copy update`. The `RCU` technique designed for rarely-modified data structures. The idea of the `RCU` is simple. For example we have a rarely-modified data structure. If somebody wants to change this data structure, we make a copy of this data structure and make all changes in the copy. In the same time all other users of the data structure use old version of it. Next, we need to choose safe moment when original version of the data structure will have no users and update it with the modified copy.

Of course this description of the `RCU` is very simplified. To understand some details about `RCU`, first of all we need to learn some terminology. Data readers in the `RCU` executed in the critical section. Everytime when data reader joins to the critical section, it calls the `rcu_read_lock`, and `rcu_read_unlock` on exit from the critical section. If the thread is not in the critical section, it will be in state which called - `quiescent state`. Every moment when every thread was in the `quiescent state` called - `grace period`. If a thread wants to remove element from the data structure, this occurs in two steps. First steps is `removal` - atomically removes element from the data structure, but does not release the physical memory. After this thread-writer announces and waits while it will be finsihed. From this moment, the removed element is available to the thread-readers. After the `grace perioud` will be finished, the second step of the element removal will be started, it just removes element from the physical memory.

There a couple implementations of the `RCU`. Old `RCU` called classic, the new implemetation called `tree` RCU. As you already can undrestand, the `CONFIG_TREE_RCU` kernel configuration option enables tree `RCU`. Another is the `tiny` RCU which depends on `CONFIG_TINY_RCU` and `CONFIG_SMP=n`. We will see more details about the `RCU` in general in the separate chapter about synchronization primitives, but now let's look on the `rcu_init` implementation from the kernel/rcu/tree.c:

```
void __init rcu_init(void)
{
        int cpu;

        rcu_bootup_announce();
        rcu_init_geometry();
        rcu_init_one(&rcu_bh_state, &rcu_bh_data);
        rcu_init_one(&rcu_sched_state, &rcu_sched_data);
        __rcu_init_preempt();
        open_softirq(RCU_SOFTIRQ, rcu_process_callbacks);

        /*
         * We don't need protection against CPU-hotplug here because
         * this is called early in boot, before either interrupts
         * or the scheduler are operational.
         */
        cpu_notifier(rcu_cpu_notify, 0);
        pm_notifier(rcu_pm_notify, 0);
        for_each_online_cpu(cpu)
                rcu_cpu_notify(NULL, CPU_UP_PREPARE, (void *)(long)cpu);

        rcu_early_boot_tests();
}
```

In the beginning of the `rcu_init` function we define `cpu` variable and call `rcu_bootup_announce`. The `rcu_bootup_announce` function is pretty simple:

```
static void __init rcu_bootup_announce(void)
{
        pr_info("Hierarchical RCU implementation.\n");
        rcu_bootup_announce_oddness();
}
```

It just prints information about the `RCU` with the `pr_info` function and `rcu_bootup_announce_oddness` which uses `pr_info` too, for printing different information about the current `RCU` configuration which depends on different kernel configuration options like `CONFIG_RCU_TRACE`, `CONFIG_PROVE_RCU`, `CONFIG_RCU_FANOUT_EXACT` and etc... In the next step, we can see the call of the `rcu_init_geometry` function. This function defined in the same source code file and computes the node tree geometry depends on amount of CPUs. Actually `RCU` provides scalability with extremely low internal to RCU lock contention. What if a data structure will be read from the different CPUs? `RCU` API provides the `rcu_state` structure wihch presents RCU global state including node hierarchy. Hierachy presented by the:

```
struct rcu_node node[NUM_RCU_NODES];
```

array of structures. As we can read in the comment which is above definition of this structure:

```
The root (first level) of the hierarchy is in ->node[0] (referenced by ->level[0]), the second
level in ->node[1] through ->node[m] (->node[1] referenced by ->level[1]), and the third level
in ->node[m+1] and following (->node[m+1] referenced by ->level[2]).  The number of levels is
determined by the number of CPUs and by CONFIG_RCU_FANOUT.

Small systems will have a "hierarchy" consisting of a single rcu_node.
```

The `rcu_node` structure defined in the kernel/rcu/tree.h and contains information about current grace period, is grace period completed or not, CPUs or groups that need to switch in order for current grace period to proceed and etc... Every `rcu_node` contains a lock for a couple of CPUs. These `rcu_node` structures embedded into a linear array in the `rcu_state` structure and represeted as a tree with the root in the zero element and it covers all CPUs. As you can see the number of the rcu nodes determined by the `NUM_RCU_NODES` which depends on number of available CPUs:

```
#define NUM_RCU_NODES (RCU_SUM - NR_CPUS)
#define RCU_SUM (NUM_RCU_LVL_0 + NUM_RCU_LVL_1 + NUM_RCU_LVL_2 + NUM_RCU_LVL_3 + NUM_RCU_LVL_4)
```

where levels values depend on the `CONFIG_RCU_FANOUT_LEAF` configuration option. For example for the simplest case, one `rcu_node` will cover two CPU on machine with the eight CPUs:

```
+-----------------------------------------------------------------+
|  rcu_state                                                      |
|                 +----------------------+                        |
|                 |         root         |                        |
|                 |       rcu_node       |                        |
|                 +----------------------+                        |
|                    |                |                           |
|               +----v-----+       +--v-------+                   |
|               |          |       |          |                   |
|               | rcu_node |       | rcu_node |                   |
|               |          |       |          |                   |
|         +------------------+     +----------------+             |
|         |                  |        |             |             |
|         |                  |        |             |             |
|    +----v-----+    +-------v--+   +-v--------+  +-v--------+    |
|    |          |    |          |   |          |  |          |    |
|    | rcu_node |    | rcu_node |   | rcu_node |  | rcu_node |    |
|    |          |    |          |   |          |  |          |    |
|    +----------+    +----------+   +----------+  +----------+    |
|         |                 |             |               |       |
|         |                 |             |               |       |
|         |                 |             |               |       |
|         |                 |             |               |       |
+---------|-----------------|-------------|---------------|-------+
          |                 |             |               |
+---------v-----------------v-------------v---------------v--------+
|                 |                |               |               |
|     CPU1        |      CPU3      |      CPU5     |     CPU7      |
|                 |                |               |               |
|     CPU2        |      CPU4      |      CPU6     |     CPU8      |
|                 |                |               |               |
+------------------------------------------------------------------+
```

So, in the `rcu_init_geometry` function we just need to calculate the total number of `rcu_node` structures. We start to do it with the calculation of the `jiffies` till to the first and next `fqs` which is `force-quiescent-state` (read above about it):

```
d = RCU_JIFFIES_TILL_FORCE_QS + nr_cpu_ids / RCU_JIFFIES_FQS_DIV;
if (jiffies_till_first_fqs == ULONG_MAX)
        jiffies_till_first_fqs = d;
if (jiffies_till_next_fqs == ULONG_MAX)
        jiffies_till_next_fqs = d;
```

where:

```
#define RCU_JIFFIES_TILL_FORCE_QS (1 + (HZ > 250) + (HZ > 500))
#define RCU_JIFFIES_FQS_DIV     256
```

As we calculated these jiffies, we check that previous defined `jiffies_till_first_fqs` and `jiffies_till_next_fqs` variables are equal to the ULONG_MAX (their default values) and set they equal to the calculated value. As we did not touch these variables before, they are equal to the `ULONG_MAX`:

```
static ulong jiffies_till_first_fqs = ULONG_MAX;
static ulong jiffies_till_next_fqs = ULONG_MAX;
```

In the next step of the `rcu_init_geometry`, we check that `rcu_fanout_leaf` didn't chage (it has the same value as `CONFIG_RCU_FANOUT_LEAF` in compile-time) and equal to the value of the `CONFIG_RCU_FANOUT_LEAF` configuration option, we just return:

```
if (rcu_fanout_leaf == CONFIG_RCU_FANOUT_LEAF &&
    nr_cpu_ids == NR_CPUS)
    return;
```

After this we need to compute the number of nodes that can be handled an `rcu_node` tree with the given number of levels:

```
rcu_capacity[0] = 1;
rcu_capacity[1] = rcu_fanout_leaf;
for (i = 2; i <= MAX_RCU_LVLS; i++)
    rcu_capacity[i] = rcu_capacity[i - 1] * CONFIG_RCU_FANOUT;
```

And in the last step we calcluate the number of rcu_nodes at each level of the tree in the loop.

As we calculated geometry of the `rcu_node` tree, we need to back to the `rcu_init` function and next step we need to initialize two `rcu_state` structures with the `rcu_init_one` function:

```
rcu_init_one(&rcu_bh_state, &rcu_bh_data);
rcu_init_one(&rcu_sched_state, &rcu_sched_data);
```

The `rcu_init_one` function takes two arguments:

- Global `RCU` state;
- Per-CPU data for `RCU`.

Both variables defined in the kernel/rcu/tree.h with its `percpu` data:

```
extern struct rcu_state rcu_bh_state;
DECLARE_PER_CPU(struct rcu_data, rcu_bh_data);
```

About this states you can read here. As I wrote above we need to initialize `rcu_state` structures and `rcu_init_one` function will help us with it. After the `rcu_state` initialization, we can see the call of the `__rcu_init_preempt` which depends on the `CONFIG_PREEMPT_RCU` kernel configuration option. It does the same that previous functions - initialization of the `rcu_preempt_state` structure with the `rcu_init_one` function which has `rcu_state` type. After this, in the `rcu_init`, we can see the call of the:

```
open_softirq(RCU_SOFTIRQ, rcu_process_callbacks);
```

function. This function registers a handler of the `pending interrupt`. Pending interrupt or `softirq` supposes that part of actions cab be delayed for later execution when the system will be less loaded. Pending interrupts represeted by the following structure:

```
struct softirq_action
{
        void    (*action)(struct softirq_action *);
};
```

which defined in the include/linux/interrupt.h and contains only one field - handler of an interrupt. You can know about `softirqs` in the your system with the:

```
$ cat /proc/softirqs
                    CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
          HI:          2          0          0          1          0          2          0          0
       TIMER:     137779     108110     139573     107647     107408     114972      99653      98665
      NET_TX:       1127          0          4          0          1          1          0          0
      NET_RX:        334        221     132939       3076        451        361        292        303
       BLOCK:       5253       5596          8        779       2016      37442         28       2855
BLOCK_IOPOLL:          0          0          0          0          0          0          0          0
     TASKLET:         66          0       2916        113          0         24      26708          0
       SCHED:     102350      75950      91705      75356      75323      82627      69279      69914
     HRTIMER:        510        302        368        260        219        255        248        246
         RCU:      81290      68062      82979      69015      68390      69385      63304      63473
```

The `open_softirq` function takes two parameters:

- index of the interrupt;
- interrupt handler.

and adds interrupt handler to the array of the pending interrupts:

```
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
        softirq_vec[nr].action = action;
}
```

In our case the interrupt handler is - `rcu_process_callbacks` which defined in the kernel/rcu/tree.c and does the `RCU` core processing for the current CPU. After we registered `softirq` interrupt for the `RCU`, we can see the following code:

```
cpu_notifier(rcu_cpu_notify, 0);
pm_notifier(rcu_pm_notify, 0);
for_each_online_cpu(cpu)
    rcu_cpu_notify(NULL, CPU_UP_PREPARE, (void *)(long)cpu);
```

Here we can see registration of the `cpu` notifier which needs in sysmtems which supports CPU hotplug and we will not dive into details about this theme. The last function in the `rcu_init` is the `rcu_early_boot_tests`:

```
void rcu_early_boot_tests(void)
{
        pr_info("Running RCU self tests\n");

        if (rcu_self_test)
                 early_boot_test_call_rcu();
         if (rcu_self_test_bh)
                 early_boot_test_call_rcu_bh();
         if (rcu_self_test_sched)
                early_boot_test_call_rcu_sched();
}
```

which runs self tests for the `RCU`.

That's all. We saw initialization process of the `RCU` subsystem. As I wrote above, more about the `RCU` will be in the separate chapter about synchronization primitives.

Rest of the initialization process

Ok, we already passed the main theme of this part which is `RCU` initialization, but it is not the end of the linux kernel initialization process. In the last paragraph of this theme we will see a couple of functions which work in the initialization time, but we will not dive into deep details around this function by different reasons. Some reasons not to dive into details are following:

- They are not very important for the generic kernel initialization process and can depend on the different kernel configuration;
- They have the character of debugging and not important too for now;
- We will see many of this stuff in the separate parts/chapters.

After we initilized `RCU`, the next step which you can see in the init/main.c is the - `trace_init` function. As you can understand from its name, this function initialize tracing subsystem. More about linux kernel trace system you can read - here.

After the `trace_init`, we can see the call of the `radix_tree_init`. If you are familar with the different data structures, you can understand from the name of this function that it initializes kernel implementation of the Radix tree. This function defined in the lib/radix-tree.c and more about it you can read in the part about Radix tree.

In the next step we can see the functions which are related to the `interrupts handling` subsystem, they are:

- `early_irq_init`
- `init_IRQ`
- `softirq_init`

We will see explanation about this functions and their implementation in the special part about interrupts and exceptions handling. After this many different functions (like `init_timers`, `hrtimers_init`, `time_init` and etc...) which are related to different timing and timers stuff. More about these function we will see in the chapter about timers.

The next couple of functions related with the perf events - `perf_event-init` (will be separate chapter about perf), initialization of the `profiling` with the `profile_init`. After this we enable `irq` with the call of the:

```
local_irq_enable();
```

which expands to the `sti` instruction and making post initialization of the SLAB with the call of the `kmem_cache_init_late` function (As I wrote above we will know about the `SLAB` in the Linux memory management chapter).

After the post initialization of the `SLAB`, next point is initialization of the console with the `console_init` function from the drivers/tty/tty_io.c.

After the console initialization, we can see the `lockdep_info` function which prints information about the Lock dependency validator. After this, we can see the initialization of the dynamic allocation of the `debug objects` with the `debug_objects_mem_init`, kernel memory leack detector initialization with the `kmemleak_init`, `percpu` pageset setup with the `setup_per_cpu_pageset`, setup of the NUMA policy with the `numa_policy_init`, setting time for the scheduler with the `sched_clock_init`, `pidmap` initialization with the call of the `pidmap_init` function for the initial `PID` namespace, cache creation with the `anon_vma_init` for the private virtual memory areas and early initialization of the ACPI with the `acpi_early_init`.

This is the end of the ninth part of the linux kernel initialization process and here we saw initialization of the RCU. In the last paragraph of this part (`Rest of the initialization process`) we went thorugh the many functions but did not dive into details about their implementations. Do not worry if you do not know anything about these stuff or you know and do not understand anything about this. As I wrote already many times, we will see details of implementations, but in the other parts or other chapters.

Conclusion

It is the end of the ninth part about the linux kernel initialization process. In this part, we looked on the initialization process of the `RCU` subsystem. In the next part we will continue to dive into linux kernel initialization process and I hope that we will finish with the `start_kernel` function and will go to the `rest_init` function from the same init/main.c source code file and will see that start of the first 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

- lock-free data structures
- kmemleak
- ACPI
- IRQs
- RCU
- RCU documentation
- integer ID management
- Documentation/memory-barriers.txt
- Runtime locking correctness validator
- Per-CPU variables
- Linux kernel memory management
- slab
- i2c
- Previous part

Kernel initialization. Part 10.

End of the linux kernel initialization process

This is tenth part of the chapter about linux kernel initialization process and in the previous part we saw the initialization of the RCU and stopped on the call of the `acpi_early_init` function. This part will be the last part of the Kernel initialization process chapter, so let's finish with it.

After the call of the `acpi_early_init` function from the init/main.c, we can see the following code:

```
#ifdef CONFIG_X86_ESPFIX64
    init_espfix_bsp();
#endif
```

Here we can see the call of the `init_espfix_bsp` function which depends on the `CONFIG_X86_ESPFIX64` kernel configuration option. As we can understand from the function name, it does something with the stack. This function defined in the arch/x86/kernel/espfix_64.c and prevents leaking of `31:16` bits of the `esp` register during returning to 16-bit stack. First of all we install `espfix` page upper directory into the kernel page directory in the `init_espfix_bs` :

```
pgd_p = &init_level4_pgt[pgd_index(ESPFIX_BASE_ADDR)];
pgd_populate(&init_mm, pgd_p, (pud_t *)espfix_pud_page);
```

Where `ESPFIX_BASE_ADDR` is:

```
#define PGDIR_SHIFT     39
#define ESPFIX_PGD_ENTRY _AC(-2, UL)
#define ESPFIX_BASE_ADDR (ESPFIX_PGD_ENTRY << PGDIR_SHIFT)
```

Also we can find it in the Documentation/arch/x86_64/mm:

```
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
```

After we've filled page global directory with the `espfix` pud, the next step is call of the `init_espfix_random` and `init_espfix_ap` functions. The first function returns random locations for the `espfix` page and the second enables the `espfix` the current CPU. After the `init_espfix_bsp` finished to work, we can see the call of the `thread_info_cache_init` function which defined in the kernel/fork.c and allocates cache for the `thread_info` if its size is less than `PAGE_SIZE` :

```
# if THREAD_SIZE >= PAGE_SIZE
...
...
...
void thread_info_cache_init(void)
{
        thread_info_cache = kmem_cache_create("thread_info", THREAD_SIZE,
                                              THREAD_SIZE, 0, NULL);
        BUG_ON(thread_info_cache == NULL);
}
...
...
```

```
...
#endif
```

As we already know the `PAGE_SIZE` is `(_AC(1,UL) << PAGE_SHIFT)` or `4096` bytes and `THREAD_SIZE` is `(PAGE_SIZE << THREAD_SIZE_ORDER)` or `16384` bytes for the `x86_64`. The next function after the `thread_info_cache_init` is the `cred_init` from the [kernel/cred.c](). This function just allocates space for the credentials (like `uid`, `gid` and etc...):

```
void __init cred_init(void)
{
         cred_jar = kmem_cache_create("cred_jar", sizeof(struct cred),
                                      0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
}
```

more about credentials you can read in the [Documentation/security/credentials.txt](). Next step is the `fork_init` function from the [kernel/fork.c](). The `fork_init` function allocates space for the `task_struct`. Let's look on the implementation of the `fork_init`. First of all we can see definitions of the `ARCH_MIN_TASKALIGN` macro and creation of a slab where task_structs will be allocated:

```
#ifndef CONFIG_ARCH_TASK_STRUCT_ALLOCATOR
#ifndef ARCH_MIN_TASKALIGN
#define ARCH_MIN_TASKALIGN      L1_CACHE_BYTES
#endif
        task_struct_cachep =
                kmem_cache_create("task_struct", sizeof(struct task_struct),
                        ARCH_MIN_TASKALIGN, SLAB_PANIC | SLAB_NOTRACK, NULL);
#endif
```

As we can see this code depends on the `CONFIG_ARCH_TASK_STRUCT_ACLLOCATOR` kernel configuration option. This configuration option shows the presence of the `alloc_task_struct` for the given architecture. As `x86_64` has no `alloc_task_struct` function, this code will not work and even will not be compiled on the `x86_64`.

Allocating cache for init task

After this we can see the call of the `arch_task_cache_init` function in the `fork_init`:

```
void arch_task_cache_init(void)
{
        task_xstate_cachep =
                kmem_cache_create("task_xstate", xstate_size,
                                  __alignof__(union thread_xstate),
                                  SLAB_PANIC | SLAB_NOTRACK, NULL);
        setup_xstate_comp();
}
```

The `arch_task_cache_init` does initialization of the architecture-specific caches. In our case it is `x86_64`, so as we can see, the `arch_task_cache_init` allocates space for the `task_xstate` which represents [FPU]() state and sets up offsets and sizes of all extended states in [xsave]() area with the call of the `setup_xstate_comp` function. After the `arch_task_cache_init` we calculate default maximum number of threads with the:

```
set_max_threads(MAX_THREADS);
```

where default maximum number of threads is:

```
#define FUTEX_TID_MASK  0x3fffffff
#define MAX_THREADS     FUTEX_TID_MASK
```

In the end of the `fork_init` function we initalize signal handler:

```
init_task.signal->rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
init_task.signal->rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
init_task.signal->rlim[RLIMIT_SIGPENDING] =
        init_task.signal->rlim[RLIMIT_NPROC];
```

As we know the `init_task` is an instance of the `task_struct` structure, so it contains `signal` field which represents signal handler. It has following type `struct signal_struct`. On the first two lines we can see setting of the current and maximum limit of the `resource limits`. Every process has an associated set of resource limits. These limits specify amount of resources which current process can use. Here `rlim` is resource control limit and presented by the:

```
struct rlimit {
        __kernel_ulong_t        rlim_cur;
        __kernel_ulong_t        rlim_max;
};
```

structure from the include/uapi/linux/resource.h. In our case the resource is the `RLIMIT_NPROC` which is the maximum number of process that use can own and `RLIMIT_SIGPENDING` - the maximum number of pending signals. We can see it in the:

```
cat /proc/self/limits
Limit                     Soft Limit           Hard Limit           Units
...
...
...
Max processes             63815                63815                processes
Max pending signals       63815                63815                signals
...
...
...
```

Initialization of the caches

The next function after the `fork_init` is the `proc_caches_init` from the kernel/fork.c. This function allocates caches for the memory descriptors (or `mm_struct` structure). At the beginning of the `proc_caches_init` we can see allocation of the different SLAB caches with the call of the `kmem_cache_create`:

- `sighand_cachep` - manage information about installed signal handlers;
- `signal_cachep` - manage information about process signal descriptor;
- `files_cachep` - manage information about opened files;
- `fs_cachep` - manage filesystem information.

After this we allocate `SLAB` cache for the `mm_struct` structures:

```
mm_cachep = kmem_cache_create("mm_struct",
                         sizeof(struct mm_struct), ARCH_MIN_MMSTRUCT_ALIGN,
                         SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_NOTRACK, NULL);
```

After this we allocate `SLAB` cache for the important `vm_area_struct` which used by the kernel to manage virtual memory space:

```
vm_area_cachep = KMEM_CACHE(vm_area_struct, SLAB_PANIC);
```

Note, that we use `KMEM_CACHE` macro here instead of the `kmem_cache_create`. This macro defined in the include/linux/slab.h and just expands to the `kmem_cache_create` call:

```
#define KMEM_CACHE(__struct, __flags) kmem_cache_create(#__struct,\
                sizeof(struct __struct), __alignof__(struct __struct),\
                (__flags), NULL)
```

The `KMEM_CACHE` has one difference from `kmem_cache_create`. Take a look on `__alignof__` operator. The `KMEM_CACHE` macro aligns `SLAB` to the size of the given structure, but `kmem_cache_create` uses given value to align space. After this we can see the call of the `mmap_init` and `nsproxy_cache_init` functions. The first function initalizes virtual memory area `SLAB` and the second function initializes `SLAB` for namespaces.

The next function after the `proc_caches_init` is `buffer_init`. This function defined in the fs/buffer.c source code file and allocate cache for the `buffer_head`. The `buffer_head` is a special structure which defined in the include/linux/buffer_head.h and used for managing buffers. In the start of the `bufer_init` function we allocate cache for the `struct buffer_head` structures with the call of the `kmem_cache_create` function as we did it in the previous functions. And calcuate the maximum size of the buffers in memory with:

```
nrpages = (nr_free_buffer_pages() * 10) / 100;
max_buffer_heads = nrpages * (PAGE_SIZE / sizeof(struct buffer_head));
```

which will be equal to the `10%` of the `ZONE_NORMAL` (all RAM from the 4GB on the `x86_64`). The next function after the `buffer_init` is - `vfs_caches_init`. This function allocates `SLAB` caches and hashtable for different VFS caches. We already saw the `vfs_caches_init_early` function in the eighth part of the linux kernel initialization process which initialized caches for `dcache` (or directory-cache) and inode cache. The `vfs_caches_init` function makes post-early initialization of the `dcache` and `inode` caches, private data cache, hash tables for the mount points and etc... More details about VFS will be described in the separate part. After this we can see `signals_init` function. This function defined in the kernel/signal.c and allocates a cache for the `sigqueue` structures which represents queue of the real time signals. The next function is `page_writeback_init`. This function initializes the ratio for the dirty pages. Every low-level page entry contains the `dirty` bit which indicates whether a page has been written to when set.

Creation of the root for the procfs

After all of this preparations we need to create the root for the proc filesystem. We will do it with the call of the `proc_root_init` function from the fs/proc/root.c. At the start of the `proc_root_init` function we allocate the cache for the inodes and register a new filesystem in the system with the:

```
err = register_filesystem(&proc_fs_type);
      if (err)
              return;
```

As I wrote above we will not dive into details about VFS and different filesystems in this chapter, but will see it in the chapter about the `VFS`. After we've registered a new filesystem in the our system, we call the `proc_self_init` function from the TOfs/proc/self.c and this function allocates `inode` number for the `self` (`/proc/self` directory refers to the process accessing the `/proc` filesystem). The next step after the `proc_self_init` is `proc_setup_thread_self` which setups the `/proc/thread-self` directory which contains information about current thread. After this we create `/proc/self/mounts` symllink which will contains mount points with the call of the

```
proc_symlink("mounts", NULL, "self/mounts");
```

and a couple of directories depends on the different configuration options:

```
#ifdef CONFIG_SYSVIPC
        proc_mkdir("sysvipc", NULL);
#endif
        proc_mkdir("fs", NULL);
        proc_mkdir("driver", NULL);
        proc_mkdir("fs/nfsd", NULL);
#if defined(CONFIG_SUN_OPENPROMFS) || defined(CONFIG_SUN_OPENPROMFS_MODULE)
        proc_mkdir("openprom", NULL);
#endif
        proc_mkdir("bus", NULL);
        ...
        ...
        ...
        if (!proc_mkdir("tty", NULL))
                 return;
        proc_mkdir("tty/ldisc", NULL);
        ...
        ...
        ...
```

In the end of the `proc_root_init` we call the `proc_sys_init` function which creates `/proc/sys` directory and initializes the Sysctl.

It is the end of `start_kernel` function. I did not describe all functions which are called in the `start_kernel`. I missed it, because they are not so important for the generic kernel initialization stuff and depend on only different kernel configurations. They are `taskstats_init_early` which exports per-task statistic to the user-space, `delayacct_init` - initializes per-task delay accounting, `key_init` and `security_init` initialize diferent security stuff, `check_bugs` - makes fix up of the some architecture-dependent bugs, `ftrace_init` function executes initialization of the ftrace, `cgroup_init` makes initialization of the rest of the cgroup subsystem and etc... Many of these parts and subsystems will be described in the other chapters.

That's all. Finally we passed through the long-long `start_kernel` function. But it is not the end of the linux kernel initialization process. We haven't run the first process yet. In the end of the `start_kernel` we can see the last call of the - `rest_init` function. Let's go ahead.

First steps after the start_kernel

The `rest_init` function defined in the same source code file as `start_kernel` function, and this file is init/main.c. In the beginning of the `rest_init` we can see call of the two following functions:

```
    rcu_scheduler_starting();
    smpboot_thread_init();
```

The first `rcu_scheduler_starting` makes RCU scheduler active and the second `smpboot_thread_init` registers the `smpboot_thread_notifier` CPU notifier (more about it you can read in the CPU hotplug documentation. After this we can see the following calls:

```
kernel_thread(kernel_init, NULL, CLONE_FS);
pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
```

Here the `kernel_thread` function (defined in the kernel/fork.c) creates new kernel thread.As we can see the `kernel_thread` function takes three arguments:

* Function which will be executed in a new thread;
* Parameter for the `kernel_init` function;
* Flags.

We will not dive into details about `kernel_thread` implementation (we will see it in the chapter which will describe scheduler, just need to say that `kernel_thread` invokes clone). Now we only need to know that we create new kernel thread with `kernel_thread` function, parent and child of the thread will use shared information about a filesystem and it will start to execute `kernel_init` function. A kernel thread differs from an user thread that it runs in a kernel mode. So with these two `kernel_thread` calls we create two new kernel threads with the `PID = 1` for `init` process and `PID = 2` for `kthread`. We already know what is `init` process. Let's look on the `kthread`. It is special kernel thread which allows to `init` and different parts of the kernel to create another kernel threads. We can see it in the output of the `ps` util:

```
$ ps -ef | grep kthradd
alex     12866  4767  0 18:26 pts/0    00:00:00 grep kthradd
```

Let's postpone `kernel_init` and `kthreadd` for now and will go ahead in the `rest_init`. In the next step after we have created two new kernel threads we can see the following code:

```
    rcu_read_lock();
    kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
    rcu_read_unlock();
```

The first `rcu_read_lock` function marks the beginning of an RCU read-side critical section and the `rcu_read_unlock` marks the end of an RCU read-side critical section. We call these functions because we need to protect the `find_task_by_pid_ns`. The `find_task_by_pid_ns` returns pointer to the `task_struct` by the given pid. So, here we are getting the pointer to the `task_struct` for the `PID = 2` (we got it after `kthreadd` creation with the `kernel_thread`). In the next step we call `complete` function

```
complete(&kthreadd_done);
```

and pass address of the `kthreadd_done`. The `kthreadd_done` defined as

```
static __initdata DECLARE_COMPLETION(kthreadd_done);
```

where `DECLARE_COMPLETION` macro defined as:

```
#define DECLARE_COMPLETION(work) \
         struct completion work = COMPLETION_INITIALIZER(work)
```

and expands to the definition of the `completion` structure. This structure defined in the include/linux/completion.h and presents `completions` concept. Completions are a code synchronization mechanism which is provide race-free solution for the threads that must wait for some process to have reached a point or a specific state. Using completions consists of three parts: The first is definition of the `complete` structure and we did it with the `DECLARE_COMPLETION`. The second is call of the `wait_for_completion`. After the call of this function, a thread which called it will not continue to execute and will wait while other thread did not call `complete` function. Note that we call `wait_for_completion` with the `kthreadd_done` in the beginning of the `kernel_init_freeable` :

```
wait_for_completion(&kthreadd_done);
```

And the last step is to call `complete` function as we saw it above. After this the `kernel_init_freeable` function will not be executed while `kthreadd` thread will not be set. After the `kthreadd` was set, we can see three following functions in the `rest_init` :

```
    init_idle_bootup_task(current);
    schedule_preempt_disabled();
    cpu_startup_entry(CPUHP_ONLINE);
```

The first `init_idle_bootup_task` function from the kernel/sched/core.c sets the Scheduling class for the current process (`idle` class in our case):

```
void init_idle_bootup_task(struct task_struct *idle)
{
        idle->sched_class = &idle_sched_class;
}
```

where `idle` class is a low priority tasks and tasks can be run only when the processor doesn't have to run anything besides this tasks. The second function `schedule_preempt_disabled` disables preempt in `idle` tasks. And the third function `cpu_startup_entry` defined in the kernel/sched/idle.c and calls `cpu_idle_loop` from the kernel/sched/idle.c. The `cpu_idle_loop` function works as process with `PID = 0` and works in the background. Main purpose of the `cpu_idle_loop` is usage of the idle CPU cycles. When there are no one process to run, this process starts to work. We have one process with `idle` scheduling class (we just set the `current` task to the `idle` with the call of the `init_idle_bootup_task` function), so the `idle` thread does not do useful work and checks that there is not active task to switch:

```
static void cpu_idle_loop(void)
{
        ...
        ...
        ...
        while (1) {
                while (!need_resched()) {
                ...
                ...
                ...
                }
        ...
        }
```

More about it will be in the chapter about scheduler. So for this moment the `start_kernel` calls the `rest_init` function which spawns an `init` (`kernel_init` function) process and become `idle` process itself. Now is time to look on the `kernel_init` . Execution of the `kernel_init` function starts from the call of the `kernel_init_freeable` function. The `kernel_init_freeable` function first of all waits for the completion of the `kthreadd` setup. I already wrote about it above:

```
wait_for_completion(&kthreadd_done);
```

After this we set `gfp_allowed_mask` to `__GFP_BITS_MASK` which means that already system is running, set allowed cpus/mems to all CPUs with NUMA nodes with the `set_mems_allowed` function, allow `init` process to run on any CPU with the `set_cpus_allowed_ptr` , set pid for the `cad` or `Ctrl-Alt-Delete` , do preparation for booting of the other CPUs with the call of the `smp_prepare_cpus` , call early initcalls with the `do_pre_smp_initcalls` , initialization of the `SMP` with the `smp_init` and initialization of the lockup_detector with the call of the `lockup_detector_init` and initialize scheduler with the `sched_init_smp`.

After this we can see the call of the following functions - `do_basic_setup`. Before we will call the `do_basic_setup` function, our kernel already initialized for this moment. As comment says:

```
Now we can finally start doing some real work..
```

The `do_basic_setup` will reinitialize cpuset to the active CPUs, initialization of the `khelper` - which is a kernel thread which used for making calls out to userspace from within the kernel, initialize tmpfs, initialize `drivers` subsystem, enable the user-mode helper `workqueue` and make post-early call of the `initcalls`. We can see openinng of the `dev/console` and dup twice file descriptors from `0` to `2` after the `do_basic_setup`:

```C
if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
    pr_err("Warning: unable to open an initial console.\n");

(void) sys_dup(0);
(void) sys_dup(0);
```

We are using two system calls here `sys_open` and `sys_dup`. In the next chapters we will see explanation and implementation of the different system calls. After we opened initial console, we check that `rdinit=` option was passed to the kernel command line or set default path of the ramdisk:

```C
if (!ramdisk_execute_command)
    ramdisk_execute_command = "/init";
```

Check user's permissions for the `ramdisk` and call the `prepare_namespace` function from the init/do_mounts.c which checks and mounts the initrd:

```C
if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0) {
    ramdisk_execute_command = NULL;
    prepare_namespace();
}
```

This is the end of the `kernel_init_freeable` function and we need return to the `kernel_init`. The next step after the `kernel_init_freeable` finished its execution is the `async_synchronize_full`. This function waits until all asynchronous function calls have been done and after it we will call the `free_initmem` which will release all memory occupied by the initialization stuff which located between `__init_begin` and `__init_end`. After this we protect `.rodata` with the `mark_rodata_ro` and update state of the system from the `SYSTEM_BOOTING` to the

```C
system_state = SYSTEM_RUNNING;
```

And tries to run the `init` process:

```C
if (ramdisk_execute_command) {
    ret = run_init_process(ramdisk_execute_command);
    if (!ret)
        return 0;
    pr_err("Failed to execute %s (error %d)\n",
           ramdisk_execute_command, ret);
}
```

First of all it checks the `ramdisk_execute_command` which we set in the `kernel_init_freeable` function and it will be equal to the value of the `rdinit=` kernel command line parameters or `/init` by default. The `run_init_process` function fills the first element of the `argv_init` array:

```
static const char *argv_init[MAX_INIT_ARGS+2] = { "init", NULL, };
```

which represents arguments of the `init` program and call `do_execve` function:

```
argv_init[0] = init_filename;
return do_execve(getname_kernel(init_filename),
    (const char __user *const __user *)argv_init,
    (const char __user *const __user *)envp_init);
```

The `do_execve` function defined in the include/linux/sched.h and runs program with the given file name and arguments. If we did not pass `rdinit=` option to the kernel command line, kernel starts to check the `execute_command` which is equal to value of the `init=` kernel command line parameter:

```
    if (execute_command) {
        ret = run_init_process(execute_command);
        if (!ret)
            return 0;
        panic("Requested init %s failed (error %d).",
              execute_command, ret);
    }
```

If we did not pass `init=` kernel command line parameter too, kernel tries to run one of the following executable files:

```
if (!try_to_run_init_process("/sbin/init") ||
    !try_to_run_init_process("/etc/init") ||
    !try_to_run_init_process("/bin/init") ||
    !try_to_run_init_process("/bin/sh"))
    return 0;
```

In other way we finish with panic:

```
panic("No working init found.  Try passing init= option to kernel. "
      "See Linux Documentation/init.txt for guidance.");
```

That's all! Linux kernel initialization process is finished!

Conclusion

It is the end of the tenth part about the linux kernel initialization process. And it is not only `tenth` part, but this is the last part which describes initialization of the linux kernel. As I wrote in the first part of this chapter, we will go through all steps of the kernel initialization and we did it. We started at the first architecture-independent function - `start_kernel` and finished with the launch of the first `init` process in the our system. I missed details about different subsystem of the kernel, for example I almost did not cover linux kernel scheduler or we did not see almost anything about interrupts and exceptions handling and etc... From the next part we will start to dive to the different kernel subsystems. Hope it will be interesting.

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

- SLAB
- xsave
- FPU
- Documentation/security/credentials.txt
- Documentation/x86/x86_64/mm
- RCU
- VFS
- inode
- proc
- man proc
- Sysctl
- ftrace
- cgroup
- CPU hotplug documentation
- completions - wait for completion handling
- NUMA
- cpus/mems
- initcalls
- Tmpfs
- initrd
- panic
- Previous part

Interrupts and Interrupt Handling

You will find a couple of posts which describe an interrupts and an exceptions handling in the linux kernel.

- Interrupts and Interrupt Handling. Part 1. - describes an interrupts handling theory.
- Start to dive into interrupts in the Linux kernel - this part starts to describe interrupts and exceptions handling related stuff from the early stage.
- Early interrupt handlers - third part describes early interrupt handlers.
- Interrupt handlers - fourth part describes first non-early interrupt handlers.
- Implementation of exception handlers - descripbes implementation of some exception handlers as double fault, divide by zero and etc.
- Handling Non-Maskable interrupts - describes handling of non-maskable interrupts and the rest of interrupts handlers from the architecture-specific part.
- Dive into external hardware interrupts - this part describes early initialization of code which is related to handling of external hardware interrupts.
- Non-early initialization of the IRQs - this part describes non-early initialization of code which is related to handling of external hardware interrupts.
- Softirq, Tasklets and Workqueues - this part describes softirqs, tasklets and workqueues concepts.

Interrupts and Interrupt Handling. Part 1.

Introduction

This is the first part of the new chapter of the linux insides book. We have come a long way in the previous chapter of this book. We started from the earliest steps of kernel initialization and finished with the launch of the first `init` process. Yes, we saw several initialization steps which are related to the various kernel subsystems. But we did not dig deep into the details of these subsystems. With this chapter, we will try to understand how the various kernel subsystems work and how they are implemented. As you can already understand from the chapter's title, the first subsystem will be interrupts.

What is an Interrupt?

We have already heard of the word `interrupt` in several parts of this book. We even saw a couple of examples of interrupt handlers. In the current chapter we will start from the theory i.e.

- What are `interrupts` ?
- What are `interrupt handlers` ?

We will then continue to dig deeper into the details of `interrupts` and how the Linux kernel handles them.

So..., First of all what is an interrupt? An interrupt is an `event` which is raised by software or hardware when its needs the CPU's attention. For example, we press a button on the keyboard and what do we expect next? What should the operating system and computer do after this? To simplify matters assume that each peripheral device has an interrupt line to the CPU. A device can use it to signal an interrupt to the CPU. However interrupts are not signaled directly to the CPU. In the old machines there was a PIC which is a chip responsible for sequentially processing multiple interrupt requests from multiple devices. In the new machines there is an Advanced Programmable Interrupt Controller commonly known as - `APIC` . An `APIC` consists of two separate devices:

- `Local APIC`
- `I/O APIC`

The first - `Local APIC` is located on each CPU core. The local APIC is responsible for handling the CPU-specific interrupt configuration. The local APIC is usually used to manage interrupts from the APIC-timer, thermal sensor and any other such locally connected I/O devices.

The second - `I/O APIC` provides multi-processor interrupt management. It is used to distribute external interrupts among the CPU cores. More about the local and I/O APICs will be covered later in this chapter. As you can understand, interrupts can occur at any time. When an interrupt occurs, the operating system must handle it immediately. But what does it mean `to handle an interrupt` ? When an interrupt occurs, the operating system must ensure the following steps:

- The kernel must pause execution of the current process; (preempt current task);
- The kernel must search for the handler of the interrupt and transfer control (execute interrupt handler);
- After the interrupt handler completes execution, the interrupted process can resume execution.

Of course there are numerous intricacies involved in this procedure of handling interrupts. But the above 3 steps form the basic skeleton of the procedure.

Addresses of each of the interrupt handlers are maintained in a special location referred to as the - `Interrupt Descriptor Table` or `IDT` . The processor uses a unique number for recognizing the type of interruption or exception. This number is called - `vector number` . A vector number is an index in the `IDT` . There is limited amount of the vector numbers and it can be from `0` to `255` . You can note the following range-check upon the vector number within the Linux kernel source-code:

```
BUG_ON((unsigned)n > 0xFF);
```

You can find this check within the Linux kernel source code related to interrupt setup (eg. The `set_intr_gate`, `void set_system_intr_gate` in arch/x86/include/asm/desc.h). First 32 vector numbers from `0` to `31` are reserved by the processor and used for the processing of architecture-defined exceptions and interrupts. You can find the table with the description of these vector numbers in the second part of the Linux kernel initialization process - Early interrupt and exception handling. Vector numbers from `32` to `255` are designated as user-defined interrupts and are not reserved by the processor. These interrupts are generally assigned to external I/O devices to enable those devices to send interrupts to the processor.

Now let's talk about the types of interrupts. Broadly speaking, we can split interrupts into 2 major classes:

- External or hardware generated interrupts;
- Software-generated interrupts.

The first - external interrupts are received through the `Local APIC` or pins on the processor which are connected to the `Local APIC`. The second - software-generated interrupts are caused by an exceptional condition in the processor itself (sometimes using special architecture-specific instructions). A common example for an exceptional condition is `division by zero`. Another example is exiting a program with the `syscall` instruction.

As mentioned earlier, an interrupt can occur at any time for a reason which the code and CPU have no control over. On the other hand, exceptions are `synchronous` with program execution and can be classified into 3 categories:

- `Faults`
- `Traps`
- `Aborts`

A `fault` is an exception reported before the execution of a "faulty" instruction (which can then be corrected). If corrected, it allows the interrupted program to be resume.

Next a `trap` is an exception which is reported immediately following the execution of the `trap` instruction. Traps also allow the interrupted program to be continued just as a `fault` does.

Finally an `abort` is an exception that does not always report the exact instruction which caused the exception and does not allow the interrupted program to be resumed.

Also we already know from the previous part that interrupts can be classified as `maskable` and `non-maskable`. Maskable interrupts are interrupts which can be blocked with the two following instructions for `x86_64` - `sti` and `cli`. We can find them in the Linux kernel source code:

```
static inline void native_irq_disable(void)
{
        asm volatile("cli": : :"memory");
}
```

and

```
static inline void native_irq_enable(void)
{
        asm volatile("sti": : :"memory");
}
```

These two instructions modify the `IF` flag bit within the interrupt register. The `sti` instruction sets the `IF` flag and the `cli` instruction clears this flag. Non-maskable interrupts are always reported. Usually any failure in the hardware is mapped to such non-maskable interrupts.

If multiple exceptions or interrupts occur at the same time, the processor handles them in order of their predefined priorities. We can determine the priorities from the highest to the lowest in the following table:

```
+----------------------------------------------------------------+
|              |                                                 |
|   Priority   | Description                                     |
|              |                                                 |
+--------------+-------------------------------------------------+
|              | Hardware Reset and Machine Checks               |
|     1        | - RESET                                         |
|              | - Machine Check                                 |
+--------------+-------------------------------------------------+
|              | Trap on Task Switch                             |
|     2        | - T flag in TSS is set                          |
|              |                                                 |
+--------------+-------------------------------------------------+
|              | External Hardware Interventions                 |
|              | - FLUSH                                         |
|     3        | - STOPCLK                                       |
|              | - SMI                                           |
|              | - INIT                                          |
+--------------+-------------------------------------------------+
|              | Traps on the Previous Instruction               |
|     4        | - Breakpoints                                   |
|              | - Debug Trap Exceptions                         |
+--------------+-------------------------------------------------+
|     5        | Nonmaskable Interrupts                          |
+--------------+-------------------------------------------------+
|     6        | Maskable Hardware Interrupts                    |
+--------------+-------------------------------------------------+
|     7        | Code Breakpoint Fault                           |
+--------------+-------------------------------------------------+
|     8        | Faults from Fetching Next Instruction           |
|              | Code-Segment Limit Violation                    |
|              | Code Page Fault                                 |
+--------------+-------------------------------------------------+
|              | Faults from Decoding the Next Instruction       |
|              | Instruction length > 15 bytes                   |
|     9        | Invalid Opcode                                  |
|              | Coprocessor Not Available                       |
|              |                                                 |
+--------------+-------------------------------------------------+
|     10       | Faults on Executing an Instruction              |
|              | Overflow                                        |
|              | Bound error                                     |
|              | Invalid TSS                                     |
|              | Segment Not Present                             |
|              | Stack fault                                     |
|              | General Protection                              |
|              | Data Page Fault                                 |
|              | Alignment Check                                 |
|              | x87 FPU Floating-point exception                |
|              | SIMD floating-point exception                   |
|              | Virtualization exception                        |
+--------------+-------------------------------------------------+
```

Now that we know a little about the various types of interrupts and exceptions, it is time to move on to a more practical part. We start with the description of the `Interrupt Descriptor Table`. As mentioned earlier, the `IDT` stores entry points of the interrupts and exceptions handlers. The `IDT` is similar in structure to the `Global Descriptor Table` which we saw in the second part of the Kernel booting process. But of course it has some differences. Instead of `descriptors`, the `IDT` entries are called `gates`. It can contain one of the following gates:

* Interrupt gates
* Task gates
* Trap gates.

in the `x86` architecture. Only long mode interrupt gates and trap gates can be referenced in the `x86_64`. Like the `Global Descriptor Table`, the `Interrupt Descriptor table` is an array of 8-byte gates on `x86` and an array of 16-byte gates on `x86_64`. We can remember from the second part of the Kernel booting process, that `Global Descriptor Table` must contain `NULL` descriptor as its first element. Unlike the `Global Descriptor Table`, the `Interrupt Descriptor Table` may contain a gate; it is not mandatory. For example, you may remember that we have loaded the Interrupt Descriptor table with the `NULL` gates only in the earlier part while transitioning into protected mode:

```
/*
 * Set up the IDT
 */
static void setup_idt(void)
{
    static const struct gdt_ptr null_idt = {0, 0};
    asm volatile("lidtl %0" : : "m" (null_idt));
}
```

from the arch/x86/boot/pm.c. The `Interrupt Descriptor table` can be located anywhere in the linear address space and the base address of it must be aligned on an 8-byte boundary on `x86` or 16-byte boundary on `x86_64`. Base address of the `IDT` is stored in the special register - `IDTR`. There are two instructions on `x86`-compatible processors to modify the `IDTR` register:

- `LIDT`
- `SIDT`

The first instruction `LIDT` is used to load the base-address of the `IDT` i.e. the specified operand into the `IDTR`. The second instruction `SIDT` is used to read and store the contents of the `IDTR` into the specified operand. The `IDTR` register is 48-bits on the `x86` and contains following information:

```
+-----------------------------------+----------------------+
|                                   |                      |
|     Base address of the IDT       |   Limit of the IDT   |
|                                   |                      |
+-----------------------------------+----------------------+
47                                16 15                    0
```

Looking at the implementation of `setup_idt`, we have prepared a `null_idt` and loaded it to the `IDTR` register with the `lidt` instruction. Note that `null_idt` has `gdt_ptr` type which is defined as:

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

Here we can see the definition of the structure with the two fields of 2-bytes and 4-bytes each (a total of 48-bits) as we can see in the diagram. Now let's look at the `IDT` entries structure. The `IDT` entries structure is an array of the 16-byte entries which are called gates in the `x86_64`. They have the 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                                   16 15                                        0
+-------------------------------------------------------------------------------+
|                                      |                                          |
|          Segment Selector            |                 Offset 15..0             |
|                                      |                                          |
+-------------------------------------------------------------------------------+
```

To form an index into the IDT, the processor scales the exception or interrupt vector by sixteen. The processor handles the occurrence of exceptions and interrupts just like it handles calls of a procedure when it sees the `call` instruction. A processor uses an unique number or `vector number` of the interrupt or the exception as the index to find the necessary `Interrupt Descriptor Table` entry. Now let's take a closer look at an `IDT` entry.

As we can see, `IDT` entry on the diagram consists of the following fields:

- `0-15` bits - offset from the segment selector which is used by the processor as the base address of the entry point of the interrupt handler;
- `16-31` bits - base address of the segment select which contains the entry point of the interrupt handler;
- `IST` - a new special mechanism in the `x86_64`, will see it later;
- `DPL` - Descriptor Privilege Level;
- `P` - Segment Present flag;
- `48-63` bits - second part of the handler base address;
- `64-95` bits - third part of the base address of the handler;
- `96-127` bits - and the last bits are reserved by the CPU.

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

- Interrupt gate
- Trap gate
- Task gate

The `IST` or `Interrupt Stack Table` is a new mechanism in the `x86_64`. It is used as an alternative to the the legacy stack-switch mechanism. Previously The `x86` architecture provided a mechanism to automatically switch stack frames in response to an interrupt. The `IST` is a modified version of the `x86` Stack switching mode. This mechanism unconditionally switches stacks when it is enabled and can be enabled for any interrupt in the `IDT` entry related with the certain interrupt (we will soon see it). From this we can understand that `IST` is not necessary for all interrupts. Some interrupts can continue to use the legacy stack switching mode. The `IST` mechanism provides up to seven `IST` pointers in the Task State Segment or `TSS` which is the special structure which contains information about a process. The `TSS` is used for stack switching during the execution of an interrupt or exception handler in the Linux kernel. Each pointer is referenced by an interrupt gate from the `IDT`.

The `Interrupt Descriptor Table` represented by the array of the `gate_desc` structures:

```
extern gate_desc idt_table[];
```

where `gate_desc` is:

```
#ifdef CONFIG_X86_64
...
...
...
```

```
typedef struct gate_struct64 gate_desc;
...
...
...
#endif
```

and `gate_struct64` defined as:

```
struct gate_struct64 {
        u16 offset_low;
        u16 segment;
        unsigned ist : 3, zero0 : 5, type : 5, dpl : 2, p : 1;
        u16 offset_middle;
        u32 offset_high;
        u32 zero1;
} __attribute__((packed));
```

Each active thread has a large stack in the Linux kernel for the `x86_64` architecture. The stack size is defined as `THREAD_SIZE` and is equal to:

```
#define PAGE_SHIFT      12
#define PAGE_SIZE       (_AC(1,UL) << PAGE_SHIFT)
...
...
...
#define THREAD_SIZE_ORDER       (2 + KASAN_STACK_ORDER)
#define THREAD_SIZE  (PAGE_SIZE << THREAD_SIZE_ORDER)
```

The `PAGE_SIZE` is `4096` -bytes and the `THREAD_SIZE_ORDER` depends on the `KASAN_STACK_ORDER` . As we can see, the `KASAN_STACK` depends on the `CONFIG_KASAN` kernel configuration parameter and equals to the:

```
#ifdef CONFIG_KASAN
    #define KASAN_STACK_ORDER 1
#else
    #define KASAN_STACK_ORDER 0
#endif
```

`KASan` is a runtime memory debugger. So... the `THREAD_SIZE` will be `16384` bytes if `CONFIG_KASAN` is disabled or `32768` if this kernel configuration option is enabled. These stacks contain useful data as long as a thread is alive or in a zombie state. While the thread is in user-space, the kernel stack is empty except for the `thread_info` structure (details about this structure are available in the fourth part of the Linux kernel initialization process) at the bottom of the stack. The active or zombie threads aren't the only threads with their own stack. There also exist specialized stacks that are associated with each available CPU. These stacks are active when the kernel is executing on that CPU. When the user-space is executing on the CPU, these stacks do not contain any useful information. Each CPU has a few special per-cpu stacks as well. The first is the `interrupt stack` used for the external hardware interrupts. Its size is determined as follows:

```
#define IRQ_STACK_ORDER (2 + KASAN_STACK_ORDER)
#define IRQ_STACK_SIZE (PAGE_SIZE << IRQ_STACK_ORDER)
```

or `16384` bytes. The per-cpu interrupt stack represented by the `irq_stack_union` union in the Linux kernel for `x86_64` :

```
union irq_stack_union {
    char irq_stack[IRQ_STACK_SIZE];

    struct {
        char gs_base[40];
        unsigned long stack_canary;
```

```
    };
};
```

The first `irq_stack` field is a 16 kilobytes array. Also you can see that `irq_stack_union` contains structure with the two fields:

* `gs_base` - The `gs` register always points to the bottom of the `irqstack` union. On the `x86_64`, the `gs` register is shared by per-cpu area and stack canary (more about `per-cpu` variables you can read in the special part). All per-cpu symbols are zero based and the `gs` points to the base of per-cpu area. You already know that segmented memory model is abolished in the long mode, but we can set base address for the two segment registers - `fs` and `gs` with the Model specific registers and these registers can be still be used as address registers. If you remember the first part of the Linux kernel initialization process, you can remember that we have set the `gs` register:

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

where `initial_gs` points to the `irq_stack_union`:

```
GLOBAL(initial_gs)
.quad    INIT_PER_CPU_VAR(irq_stack_union)
```

* `stack_canary` - Stack canary for the interrupt stack is a `stack protector` to verify that the stack hasn't been overwritten. Note that `gs_base` is an 40 bytes array. `GCC` requires that stack canary will be on the fixed offset from the base of the `gs` and its value must be `40` for the `x86_64` and `20` for the `x86`.

The `irq_stack_union` is the first datum in the `percpu` area, we can see it in the `System.map`:

```
0000000000000000 D __per_cpu_start
0000000000000000 D irq_stack_union
0000000000004000 d exception_stacks
0000000000009000 D gdt_page
...
...
...
```

We can see its definition in the code:

```
DECLARE_PER_CPU_FIRST(union irq_stack_union, irq_stack_union) __visible;
```

Now, its time to look at the initialization of the `irq_stack_union`. Besides the `irq_stack_union` definition, we can see the definition of the following per-cpu variables in the arch/x86/include/asm/processor.h:

```
DECLARE_PER_CPU(char *, irq_stack_ptr);
DECLARE_PER_CPU(unsigned int, irq_count);
```

The first is the `irq_stack_ptr`. From the variable's name, it is obvious that this is a pointer to the top of the stack. The second - `irq_count` is used to check if a CPU is already on an interrupt stack or not. Initialization of the `irq_stack_ptr` is located in the `setup_per_cpu_areas` function in arch/x86/kernel/setup_percpu.c:

```
void __init setup_per_cpu_areas(void)
{
...
...
#ifdef CONFIG_X86_64
for_each_possible_cpu(cpu) {
    ...
    ...
    ...
    per_cpu(irq_stack_ptr, cpu) =
            per_cpu(irq_stack_union.irq_stack, cpu) +
            IRQ_STACK_SIZE - 64;
    ...
    ...
    ...
#endif
...
...
}
```

Here we go over all the CPUs on-by-one and setup `irq_stack_ptr`. This turns out to be equal to the top of the interrupt stack minus `64`. Why `64`? If you remember, we set the stack canary in the beginning of the `start_kernel` function from the init/main.c with the call of the `boot_init_stack_canary` function:

```
static __always_inline void boot_init_stack_canary(void)
{
    u64 canary;
    ...
    ...
    ...

#ifdef CONFIG_X86_64
    BUILD_BUG_ON(offsetof(union irq_stack_union, stack_canary) != 40);
#endif
    //
    // getting canary value here
    //

    this_cpu_write(irq_stack_union.stack_canary, canary);
    ...
    ...
    ...
}
```

Note that `canary` is `64` bits value. That's why we need to subtract `64` from the size of the interrupt stack to avoid overlapping with the stack canary value. Initialization of the `irq_stack_union.gs_base` is in the `load_percpu_segment` function from the arch/x86/kernel/cpu/common.c:

TODO maybe more about the wrmsl

```
void load_percpu_segment(int cpu)
{
        ...
        ...
        ...
        loadsegment(gs, 0);
        wrmsrl(MSR_GS_BASE, (unsigned long)per_cpu(irq_stack_union.gs_base, cpu));
}
```

and as we already know `gs` register points to the bottom of the interrupt stack:

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

```
    wrmsr

    GLOBAL(initial_gs)
    .quad    INIT_PER_CPU_VAR(irq_stack_union)
```

Here we can see the `wrmsr` instruction which loads the data from `edx:eax` into the Model specific register pointed by the `ecx` register. In our case model specific register is `MSR_GS_BASE` which contains the base address of the memory segment pointed by the `gs` register. `edx:eax` points to the address of the `initial_gs` which is the base address of our `irq_stack_union`.

We already know that `x86_64` has a feature called `Interrupt Stack Table` or `IST` and this feature provides the ability to switch to a new stack for events non-maskable interrupt, double fault and etc... There can be up to seven `IST` entries per-cpu. Some of them are:

- `DOUBLEFAULT_STACK`
- `NMI_STACK`
- `DEBUG_STACK`
- `MCE_STACK`

or

```
#define DOUBLEFAULT_STACK 1
#define NMI_STACK 2
#define DEBUG_STACK 3
#define MCE_STACK 4
```

All interrupt-gate descriptors which switch to a new stack with the `IST` are initialized with the `set_intr_gate_ist` function. For example:

```
set_intr_gate_ist(X86_TRAP_NMI, &nmi, NMI_STACK);
...
...
...
set_intr_gate_ist(X86_TRAP_DF, &double_fault, DOUBLEFAULT_STACK);
```

where `&nmi` and `&double_fault` are addresses of the entries to the given interrupt handlers:

```
asmlinkage void nmi(void);
asmlinkage void double_fault(void);
```

defined in the arch/x86/kernel/entry_64.S

```
idtentry double_fault do_double_fault has_error_code=1 paranoid=2
...
...
...
ENTRY(nmi)
...
...
...
END(nmi)
```

When an interrupt or an exception occurs, the new `ss` selector is forced to `NULL` and the `ss` selector's `rpl` field is set to the new `cpl`. The old `ss`, `rsp`, register flags, `cs`, `rip` are pushed onto the new stack. In 64-bit mode, the size of interrupt stack-frame pushes is fixed at 8-bytes, so we will get the following stack:

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

If the `IST` field in the interrupt gate is not `0`, we read the `IST` pointer into `rsp`. If the interrupt vector number has an error code associated with it, we then push the error code onto the stack. If the interrupt vector number has no error code, we go ahead and push the dummy error code on to the stack. We need to do this to ensure stack consistency. Next we load the segment-selector field from the gate descriptor into the CS register and must verify that the target code-segment is a 64-bit mode code segment by the checking bit `21` i.e. the `L` bit in the `Global Descriptor Table`. Finally we load the offset field from the gate descriptor into `rip` which will be the entry-point of the interrupt handler. After this the interrupt handler begins to execute. After an interrupt handler finishes its execution, it must return control to the interrupted process with the `iret` instruction. The `iret` instruction unconditionally pops the stack pointer (`ss:rsp`) to restore the stack of the interrupted process and does not depend on the `cpl` change.

That's all.

Conclusion

It is the end of the first part about interrupts and interrupt handling in the Linux kernel. We saw some theory and the first steps of the initialization of stuff related to interrupts and exceptions. In the next part we will continue to dive into interrupts and interrupts handling - into the more practical aspects of it.

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 a PR to linux-internals.

Links

- PIC
- Advanced Programmable Interrupt Controller
- protected mode
- long mode
- kernel stacks
- Task State Segement
- segmented memory model
- Model specific registers
- Stack canary
- Previous chapter

Interrupts and Interrupt Handling. Part 2.

Start to dive into interrupt and exceptions handling in the Linux kernel

We saw some theory about an interrupts and an exceptions handling in the previous part and as I already wrote in that part, we will start to dive into interrupts and exceptions in the Linux kernel source code in this part. As you already can note, the previous part mostly described theoretical aspects and since this part we will start to dive directly into the Linux kernel source code. We will start to do it as we did it in other chapters, from the very early places. We will not see the Linux kernel source code from the earliest code lines as we saw it for example in the Linux kernel booting process chapter, but we will start from the earliest code which is related to the interrupts and exceptions. Since this part we will try to go through the all interrupts and exceptions related stuff which we can find in the Linux kernel source code.

If you've read the previous parts, you can remember that the earliest place in the Linux kernel `x86_64` architecture-specifix source code which is related to the interrupt is located in the arch/x86/boot/pm.c source code file and represents the first setup of the Interrupt Descriptor Table. It occurs right before the transition into the protected mode in the `go_to_protected_mode` function by the call of the `setup_idt`:

```
void go_to_protected_mode(void)
{
    ...
    setup_idt();
    ...
}
```

The `setup_idt` function defined in the same source code file as the `go_to_protected_mode` function and just loads address of the `NULL` interrupts descriptor table:

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

where `gdt_ptr` represents special 48-bit `GTDR` register which must contain base address of the `Global Descriptor Table`:

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

Of course in our case the `gdt_ptr` does not represent `GDTR` register, but `IDTR` since we set `Interrupt Descriptor Table`. You will not find `idt_ptr` structure, because if it had been in the Linux kernel source code, it would have been the same as `gdt_ptr` but with different name. So, as you can understand there is no sense to have two similar structures which are differ only in a name. You can note here, that we do not fill the `Interrupt Descriptor Table` with entries, because it is too early to handle any interrupts or exceptions for this moment. That's why we just fill the `IDT` with the `NULL`.

And after the setup of the Interrupt descriptor table, Global Descriptor Table and other stuff we jump into protected mode in the - arch/x86/boot/pmjump.S. More about it you can read in the part which describes transition to the protected mode.

We already know from the earliest parts that entry of the protected mode located in the `boot_params.hdr.code32_start` and you can see that we pass the entry of the protected mode and `boot_params` to the `protected_mode_jump` in the end of the arch/x86/boot/pm.c:

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

The `protected_mode_jump` defined in the arch/x86/boot/pmjump.S and gets these two parameters in the `ax` and `dx` registers using one of the 8086 calling convention:

```
GLOBAL(protected_mode_jump)
    ...
    ...
    ...
    .byte    0x66, 0xea        # ljmpl opcode
2:    .long    in_pm32            # offset
    .word    __BOOT_CS        # segment
...
...
...
ENDPROC(protected_mode_jump)
```

where `in_pm32` contains jump to the 32-bit entrypoint:

```
GLOBAL(in_pm32)
    ...
    ...
    jmpl    *%eax // %eax contains address of the `startup_32`
    ...
    ...
ENDPROC(in_pm32)
```

As you can remember 32-bit entry point is in the arch/x86/boot/compressed/head_64.S assembly file, although it contains `_64` in the its name. We can see the two similar files in the `arch/x86/boot/compressed` directory:

- `arch/x86/boot/compressed/head_32.S`.
- `arch/x86/boot/compressed/head_64.S`;

But the 32-bit mode entry point the the second file in our case. The first file even not compiled for `x86_64`. Let's look on the arch/x86/boot/compressed/Makefile:

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

We can see here that `head_*` depends on the `$(BITS)` variable which depends on the architecture. You can find it in the arch/x86/Makefile:

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

Now as we jumped on the `startup_32` from the arch/x86/boot/compressed/head_64.S we will not find anything related to the interrupt handling here. The `startup_32` contains code that makes preparations before transition into the long mode and directly jumps in it. The `long mode` entry located `startup_64` and it makes preparation before the kernel decompression that occurs in the `decompress_kernel` from the arch/x86/boot/compressed/misc.c. After kernel decompressed, we jump on the `startup_64` from the arch/x86/kernel/head_64.S. In the `startup_64` we start to build identity-mapped pages. After we have built identity-mapped pages, checked NX bit, made setup of the `Extended Feature Enable Register` (see in links), updated early `Global Descriptor Table` wit the `lgdt` instruction, we need to setup `gs` register with the following code:

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

We already saw this code in the previous part and not time to know better what is going on here. First of all pay attention on the last `wrmsr` instruction. This instruction writes data from the `edx:eax` registers to the model specific register specified by the `ecx` register. We can see that `ecx` contains `$MSR_GS_BASE` which declared in the arch/x86/include/uapi/asm/msr-index.h and looks:

```
#define MSR_GS_BASE             0xc0000101
```

From this we can understand that `MSR_GS_BASE` defines number of the `model specific register`. Since registers `cs`, `ds`, `es`, and `ss` are not used in the 64-bit mode, their fields are ignored. But we can access memory over `fs` and `gs` registers. The model specific register provides `back door` to the hidden parts of these segment registers and allows to use 64-bit base address for segment register addressed by the `fs` and `gs`. So the `MSR_GS_BASE` is the hidden part and this part is mapped on the `GS.base` field. Let's look on the `initial_gs`:

```
GLOBAL(initial_gs)
    .quad    INIT_PER_CPU_VAR(irq_stack_union)
```

We pass `irq_stack_union` symbol to the `INIT_PER_CPU_VAR` macro which just concatenates `init_per_cpu__` prefix with the given symbol. In our case we will get `init_per_cpu__irq_stack_union` symbol. Let's look on the linker script. There we can see following definition:

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

It tells us that address of the `init_per_cpu__irq_stack_union` will be `irq_stack_union + __per_cpu_load`. Now we need to understand where are `init_per_cpu__irq_stack_union` and `__per_cpu_load` and what they mean. The first `irq_stack_union` defined in the arch/x86/include/asm/processor.h with the `DECLARE_INIT_PER_CPU` macro which expands to call of the `init_per_cpu_var` macro:

```
DECLARE_INIT_PER_CPU(irq_stack_union);

#define DECLARE_INIT_PER_CPU(var) \
       extern typeof(per_cpu_var(var)) init_per_cpu_var(var)

#define init_per_cpu_var(var)  init_per_cpu__##var
```

If we will expand all macro we will get the same `init_per_cpu__irq_stack_union` as we got after expanding of the `INIT_PER_CPU` macro, but you can note that it is already not just symbol, but variable. Let's look on the `typeof(percpu_var(var))` expression. Our `var` is `irq_stack_union` and `per_cpu_var` macro defined in the arch/x86/include/asm/percpu.h:

```
#define PER_CPU_VAR(var)        %__percpu_seg:var
```

where:

```
#ifdef CONFIG_X86_64
    #define __percpu_seg gs
endif
```

So, we accessing `gs:irq_stack_union` and geting its type which is `irq_union`. Ok, we defined the first variable and know its address, now let's look on the second `__per_cpu_load` symbol. There are a couple of percpu variables which are located after this symbol. The `__per_cpu_load` defined in the include/asm-generic/sections.h:

```
extern char __per_cpu_load[], __per_cpu_start[], __per_cpu_end[];
```

and presented base address of the `per-cpu` variables from the data area. So, we know address of the `irq_stack_union`, `__per_cpu_load` and we know that `init_per_cpu__irq_stack_union` must be placed right after `__per_cpu_load`. And we can see it in the System.map:

```
...
...
...
ffffffff819ed000 D __init_begin
ffffffff819ed000 D __per_cpu_load
ffffffff819ed000 A init_per_cpu__irq_stack_union
...
...
...
```

Now we know about `initia_gs`, so let's book to the our code:

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

Here we specified model specific register with `MSR_GS_BASE`, put 64-bit address of the `initial_gs` to the `edx:eax` pair and execute `wrmsr` instruction for filling the `gs` register with base address of the `init_per_cpu__irq_stack_union` which will be bottom of the interrupt stack. After this we will jump to the C code on the `x86_64_start_kernel` from the arch/x86/kernel/head64.c. In the `x86_64_start_kernel` function we do the last preparations before we jump into the generic and architecture-independent kernel code and on of these preparations is filling of the early `Interrupt Descriptor Table` with the interrupts handlres entries or `early_idt_handlers`. You can remember it, if you have read the part about the Early interrupt and exception handling and can remember following code:

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

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

but I wrote `Early interrupt and exception handling` part when Linux kernel version was - `3.18`. For this day actual version of the Linux kernel is `4.1.0-rc6+` and `Andy Lutomirski` sent the patch and soon it will be in the mainline kernel that changes behaviour for the `early_idt_handlers`. **NOTE** While I wrote this part the patch already turned in the Linux kernel source code. Let's look on it. Now the same part looks like:

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

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

AS you can see it has only one difference in the name of the array of the interrupts handlers entry points. Now it is `early_idt_handler_arry`:

```
extern const char early_idt_handler_array[NUM_EXCEPTION_VECTORS][EARLY_IDT_HANDLER_SIZE];
```

where `NUM_EXCEPTION_VECTORS` and `EARLY_IDT_HANDLER_SIZE` are defined as:

```
#define NUM_EXCEPTION_VECTORS 32
#define EARLY_IDT_HANDLER_SIZE 9
```

So, the `early_idt_handler_array` is an array of the interrupts handlers entry points and contains one entry point on every nine bytes. You can remember that previous `early_idt_handlers` was defined in the arch/x86/kernel/head_64.S. The `early_idt_handler_array` is defined in the same source code file too:

```
ENTRY(early_idt_handler_array)
...
...
...
ENDPROC(early_idt_handler_common)
```

It fills `early_idt_handler_arry` with the `.rept NUM_EXCEPTION_VECTORS` and contains entry of the `early_make_pgtable` interrupt handler (more about its implementation you can read in the part about Early interrupt and exception handling). For now we come to the end of the `x86_64` architecture-specific code and the next part is the generic kernel code. Of course you already can know that we will return to the architecture-specific code in the `setup_arch` function and other places, but this is the end of the `x86_64` early code.

Setting stack canary for the interrupt stack

The next stop after the arch/x86/kernel/head_64.S is the biggest `start_kernel` function from the init/main.c. If you've read the previous chapter about the Linux kernel initialization process, you must remember it. This function does all initialization stuff before kernel will launch first `init` process with the pid - `1`. The first thing that is related to the interrupts and exceptions handling is the call of the `boot_init_stack_canary` function.

This function sets the canary value to protect interrupt stack overflow. We already saw a little some details about implementation of the `boot_init_stack_canary` in the previous part and now let's take a closer look on it. You can find implementation of this function in the arch/x86/include/asm/stackprotector.h and its depends on the `CONFIG_CC_STACKPROTECTOR` kernel configuration option. If this option is not set this function will not do anything:

```
#ifdef CONFIG_CC_STACKPROTECTOR
...
...
...
#else
static inline void boot_init_stack_canary(void)
{
}
#endif
```

If the `CONFIG_CC_STACKPROTECTOR` kernel configuration option is set, the `boot_init_stack_canary` function starts from the check stat `irq_stack_union` that represents per-cpu interrupt stack has offset equal to forty bytes from the `stack_canary` value:

```
#ifdef CONFIG_X86_64
        BUILD_BUG_ON(offsetof(union irq_stack_union, stack_canary) != 40);
#endif
```

As we can read in the previous part the `irq_stack_union` represented by the following union:

```
union irq_stack_union {
    char irq_stack[IRQ_STACK_SIZE];

    struct {
        char gs_base[40];
        unsigned long stack_canary;
    };
};
```

which defined in the arch/x86/include/asm/processor.h. We know that unioun in the C programming language is a data structure which stores only one field in a memory. We can see here that structure has first field - `gs_base` which is 40 bytes size and represents bottom of the `irq_stack`. So, after this our check with the `BUILD_BUG_ON` macro should end successfully. (you can read the first part about Linux kernel initialization process if you're interesting about the `BUILD_BUG_ON` macro).

After this we calculate new `canary` value based on the random number and Time Stamp Counter:

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

and write `canary` value to the `irq_stack_union` with the `this_cpu_write` macro:

```
this_cpu_write(irq_stack_union.stack_canary, canary);
```

more about `this_cpu_*` operation you can read in the Linux kernel documentation.

Disabling/Enabling local interrupts

The next step in the init/main.c which is related to the interrupts and interrupts handling after we have set the `canary` value to the interrupt stack - is the call of the `local_irq_disable` macro.

This macro defined in the include/linux/irqflags.h header file and as you can understand, we can disable interrupts for the CPU with the call of this macro. Let's look on its implementation. First of all note that it depends on the `CONFIG_TRACE_IRQFLAGS_SUPPORT` kernel configuration option:

```
#ifdef CONFIG_TRACE_IRQFLAGS_SUPPORT
...
#define local_irq_disable() \
         do { raw_local_irq_disable(); trace_hardirqs_off(); } while (0)
...
#else
...
```

```
#define local_irq_disable()     do { raw_local_irq_disable(); } while (0)
...
#endif
```

They are both similar and as you can see have only one difference: the `local_irq_disable` macro contains call of the `trace_hardirqs_off` when `CONFIG_TRACE_IRQFLAGS_SUPPORT` is enabled. There is special feature in the lockdep subsystem - `irq-flags tracing` for tracing `hardirq` and `stoftirq` state. In ourcase `lockdep` subsytem can give us interesting information about hard/soft irqs on/off events which are occurs in the system. The `trace_hardirqs_off` function defined in the kernel/locking/lockdep.c:

```
void trace_hardirqs_off(void)
{
         trace_hardirqs_off_caller(CALLER_ADDR0);
}
EXPORT_SYMBOL(trace_hardirqs_off);
```

and just calls `trace_hardirqs_off_caller` function. The `trace_hardirqs_off_caller` checks the `hardirqs_enabled` filed of the current process increment the `redundant_hardirqs_off` if call of the `local_irq_disable` was redundant or the `hardirqs_off_events` if it was not. These two fields and other `lockdep` statistic related fields are defined in the kernel/locking/lockdep_internals.h and located in the `lockdep_stats` structure:

```
struct lockdep_stats {
...
...
...
int     softirqs_off_events;
int     redundant_softirqs_off;
...
...
...
}
```

If you will set `CONFIG_DEBUG_LOCKDEP` kernel configuration option, the `lockdep_stats_debug_show` function will write all tracing information to the `/proc/lockdep`:

```
static void lockdep_stats_debug_show(struct seq_file *m)
{
#ifdef CONFIG_DEBUG_LOCKDEP
    unsigned long long hi1 = debug_atomic_read(hardirqs_on_events),
                             hi2 = debug_atomic_read(hardirqs_off_events),
                             hr1 = debug_atomic_read(redundant_hardirqs_on),
    ...
    ...
    ...
    seq_printf(m, " hardirq on events:             %11llu\n", hi1);
    seq_printf(m, " hardirq off events:            %11llu\n", hi2);
    seq_printf(m, " redundant hardirq ons:         %11llu\n", hr1);
#endif
}
```

and you can see its result with the:

```
$ sudo cat /proc/lockdep
 hardirq on events:             12838248974
 hardirq off events:            12838248979
 redundant hardirq ons:               67792
 redundant hardirq offs:         3836339146
 softirq on events:                38002159
 softirq off events:               38002187
 redundant softirq ons:                   0
```

```
redundant softirq offs:                  0
```

Ok, now we know a little about tracing, but more info will be in the separate part about `lockdep` and `tracing`. You can see that the both `local_disable_irq` macros have the same part - `raw_local_irq_disable`. This macro defined in the arch/x86/include/asm/irqflags.h and expands to the call of the:

```
static inline void native_irq_disable(void)
{
        asm volatile("cli": : :"memory");
}
```

And you already must remember that `cli` instruction clears the IF flag which determines ability of a processor to handle and interrupt or an exception. Besides the `local_irq_disable`, as you already can know there is an inverse macr - `local_irq_enable`. This macro has the same tracing mechanism and very similar on the `local_irq_enable`, but as you can understand from its name, it enables interrupts with the `sti` instruction:

```
static inline void native_irq_enable(void)
{
        asm volatile("sti": : :"memory");
}
```

Now we know how `local_irq_disable` and `local_irq_enable` work. It was the first call of the `local_irq_disable` macro, but we will meet these macros many times in the Linux kernel source code. But for now we are in the `start_kernel` function from the init/main.c and we just disabled `local` interrupts. Why local and why we did it? Previously kernel provided a method to disable interrupts on all processors and it was called `cli`. This function was removed and now we have `local_irq_{enabled,disable}` to disable or enable interrupts on the current processor. After we've disabled the interrupts with the `local_irq_disable` macro, we set the:

```
early_boot_irqs_disabled = true;
```

The `early_boot_irqs_disabled` variable defined in the include/linux/kernel.h:

```
extern bool early_boot_irqs_disabled;
```

and used in the different places. For example it used in the `smp_call_function_many` function from the kernel/smp.c for the checking possible deadlock when interrupts are disabled:

```
WARN_ON_ONCE(cpu_online(this_cpu) && irqs_disabled()
                     && !oops_in_progress && !early_boot_irqs_disabled);
```

Early trap initialization during kernel initialization

The next functions after the `local_disable_irq` are `boot_cpu_init` and `page_address_init`, but they are not related to the interrupts and exceptions (more about this functions you can read in the chapter about Linux kernel initialization process). The next is the `setup_arch` function. As you can remember this function located in the arch/x86/kernel/setup.c source code file and makes initialization of many different architecture-dependent stuff. The first interrupts related function which we can see in the `setup_arch` is the - `early_trap_init` function. This function defined in the arch/x86/kernel/traps.c and fills `Interrupt Descriptor Table` with the couple of entries:

```C
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);
#ifdef CONFIG_X86_32
        set_intr_gate(X86_TRAP_PF, page_fault);
#endif
        load_idt(&idt_descr);
}
```

Here we can see calls of three different functions:

- `set_intr_gate_ist`
- `set_system_intr_gate_ist`
- `set_intr_gate`

All of these functions defined in the arch/x86/include/asm/desc.h and do the similar thing but not the same. The first `set_intr_gate_ist` function inserts new an interrupt gate in the `IDT`. Let's look on its implementation:

```C
static inline void set_intr_gate_ist(int n, void *addr, unsigned ist)
{
        BUG_ON((unsigned)n > 0xFF);
        _set_gate(n, GATE_INTERRUPT, addr, 0, ist, __KERNEL_CS);
}
```

First of all we can see the check that `n` which is vector number of the interrupt is not greater than `0xff` or 255. We need to check it because we remember from the previous part that vector number of an interrupt must be between `0` and `255`. In the next step we can see the call of the `_set_gate` function that sets a given interrupt gate to the `IDT` table:

```C
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);
}
```

Here we start from the `pack_gate` function which takes clean `IDT` entry represented by the `gate_desc` structure and fills it with the base address and limit, Interrupt Stack Table, Privilege level, type of an interrupt which can be one of the following values:

- `GATE_INTERRUPT`
- `GATE_TRAP`
- `GATE_CALL`
- `GATE_TASK`

and set the present bit for the given `IDT` entry:

```C
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);
}
```

After this we write just filled interrupt gate to the `IDT` with the `write_idt_entry` macro which expands to the `native_write_idt_entry` and just copy the interrupt gate to the `idt_table` table by the given index:

```
#define write_idt_entry(dt, entry, g)           native_write_idt_entry(dt, entry, g)

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

where `idt_table` is just array of `gate_desc` :

```
extern gate_desc idt_table[];
```

That's all. The second `set_system_intr_gate_ist` function has only one difference from the `set_intr_gate_ist` :

```
static inline void set_system_intr_gate_ist(int n, void *addr, unsigned ist)
{
        BUG_ON((unsigned)n > 0xFF);
        _set_gate(n, GATE_INTERRUPT, addr, 0x3, ist, __KERNEL_CS);
}
```

Do you see it? Look on the fourth parameter of the `_set_gate` . It is `0x3` . In the `set_intr_gate` it was `0x0` . We know that this parameter represent `DPL` or privilege level. We also know that `0` is the highest privilge level and `3` is the lowest.Now we know how `set_system_intr_gate_ist` , `set_intr_gate_ist` , `set_intr_gate` are work and we can return to the `early_trap_init` function. Let's look on it again:

```
set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
```

We set two `IDT` entries for the `#DB` interrupt and `int3` . These functions takes the same set of parameters:

- vector number of an interrupt;
- address of an interrupt handler;
- interrupt stack table index.

That's all. More about interrupts and handlers you will know in the next parts.

Conclusion

It is the end of the second part about interrupts and interrupt handling in the Linux kernel. We saw the some theory in the previous part and started to dive into interrupts and exceptions handling in the current part. We have started from the earliest parts in the Linux kernel source code which are related to the interrupts. In the next part we will continue to dive into this interesting theme and will know more about interrupt handling 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

* IDT
* Protected mode
* List of x86 calling conventions
* 8086
* Long mode
* NX
* Extended Feature Enable Register
* Model-specific register
* Process identifier
* lockdep
* irqflags tracing
* IF
* Stack canary
* Union type
* this_cpu_* operations
* vector number
* Interrupt Stack Table
* Privilege level
* Previous part

Interrupts and Interrupt Handling. Part 3.

Interrupt handlers

This is the third part of the chapter about an interrupts and an exceptions handling and in the previous part we stoped in the `setup_arch` function from the arch/x86/kernel/setup.c on the setting of the two exceptions handlers for the two following exceptions:

- `#DB` - debug exception, transfers control from the interrupted process to the debug handler;
- `#BP` - breakpoint exception, caused by the `int 3` instruction.

These exceptions allow the `x86_64` architecture to have early exception processing for the purpose of debugging via the kgdb.

As you can remember we set these exceptions handlers in the `early_trap_init` function:

```
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);
}
```

from the arch/x86/kernel/traps.c. We already saw implementation of the `set_intr_gate_ist` and `set_system_intr_gate_ist` functions in the previous part and now we will look on the implementation of these early exceptions handlers.

Debug and Breakpoint exceptions

Ok, we set the interrupts gates in the `early_trap_init` function for the `#DB` and `#BP` exceptions and now time is to look on their handlers. But first of all let's look on these exceptions. The first exceptions - `#DB` or debug exception occurs when a debug event occurs, for example attempt to change the contents of a debug register. Debug registers are special registers which present in processors starting from the Intel 80386 and as you can understand from its name they are used for debugging. These registers allow to set breakpoints on the code and read or write data to trace, thus tracking the place of errors. The debug registers are privileged resources available and the program in either real-address or protected mode at `CPL` is `0`, that's why we have used `set_intr_gate_ist` for the `#DB`, but not the `set_system_intr_gate_ist`. The verctor number of the `#DB` exceptions is `1` (we pass it as `X86_TRAP_DB`) and has no error code:

```
----------------------------------------------------------------------------------------------
|Vector|Mnemonic|Description         |Type |Error Code|Source                                |
----------------------------------------------------------------------------------------------
|1     | #DB    |Reserved            |F/T  |NO        |                                      |
----------------------------------------------------------------------------------------------
```

The second is `#BP` or breakpoint exception occurs when processor executes the INT 3 instruction. We can add it anywhere in our code, for example let's look on the simple program:

```
// breakpoint.c
#include <stdio.h>

int main() {
    int i;
    while (i < 6){
```

```
        printf("i equal to: %d\n", i);
        __asm__("int3");
        ++i;
    }
}
```

If we will compile and run this program, we will see following output:

```
$ gcc breakpoint.c -o breakpoint
i equal to: 0
Trace/breakpoint trap
```

But if will run it with gdb, we will see our breakpoint and can continue execution of our program:

```
$ gdb breakpoint
...
...
...
(gdb) run
Starting program: /home/alex/breakpoints
i equal to: 0

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000400585 in main ()
=> 0x0000000000400585 <main+31>:    83 45 fc 01    add    DWORD PTR [rbp-0x4],0x1
(gdb) c
Continuing.
i equal to: 1

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000400585 in main ()
=> 0x0000000000400585 <main+31>:    83 45 fc 01    add    DWORD PTR [rbp-0x4],0x1
(gdb) c
Continuing.
i equal to: 2

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000400585 in main ()
=> 0x0000000000400585 <main+31>:    83 45 fc 01    add    DWORD PTR [rbp-0x4],0x1
...
...
...
```

Now we know a little about these two exceptions and we can move on to consideration of their handlers.

Preparation before an interrupt handler

As you can note, the `set_intr_gate_ist` and `set_system_intr_gate_ist` functions takes an addresses of the exceptions handlers in the second parameter:

- `&debug` ;
- `&int3` .

You will not find these functions in the C code. All that can be found in in the `*.c/*.h` files only definition of this functions in the arch/x86/include/asm/traps.h:

```
asmlinkage void debug(void);
asmlinkage void int3(void);
```

But we can see `asmlinkage` descriptor here. The `asmlinkage` is the special specificator of the gcc. Actually for a `C` functions which are will be called from assembly, we need in explicit declaration of the function calling convention. In our case, if function maked with `asmlinkage` descriptor, then `gcc` will compile the function to retrieve parameters from stack. So, both handlers are defined in the arch/x86/kernel/entry_64.S assembly source code file with the `idtentry` macro:

```
idtentry debug do_debug has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
idtentry int3 do_int3 has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
```

Actually `debug` and `int3` are not interrupts handlers. Remember that before we can execute an interrupt/exception handler, we need to do some preparations as:

- When an interrupt or exception occured, the processor uses an exception or interrupt vector as an index to a descriptor in the `IDT` ;
- In legacy mode `ss:esp` registers are pushed on the stack only if privilege level changed. In 64-bit mode `ss:rsp` pushed on the stack everytime;
- During stack switching with `IST` the new `ss` selector is forced to null. Old `ss` and `rsp` are pushed on the new stack.
- The `rflags` , `cs` , `rip` and error code pushed on the stack;
- Control transfered to an interrupt handler;
- After an interrupt handler will finish its work and finishes with the `iret` instruction, old `ss` will be poped from the stack and loaded to the `ss` register.
- `ss:rsp` will be popped from the stack unconditionally in the 64-bit mode and will be popped only if there is a privilege level change in legacy mode.
- `iret` instruction will restore `rip` , `cs` and `rflags` ;
- Interrupted program will continue its execution.

```
    +--------------------+
+40 |        ss          |
+32 |       rsp          |
+24 |      rflags        |
+16 |        cs          |
 +8 |       rip          |
  0 |    error code      |
    +--------------------+
```

Now we can see on the preparations before a process will transfer control to an interrupt/exception handler from practical side. As I already wrote above the first thirteen exceptions handlers defined in the arch/x86/kernel/entry_64.S assembly file with the idtentry macro:

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
...
...
...
END(\sym)
.endm
```

This macro defines an exception entry point and as we can see it takes `five` arguments:

- `sym` - defines global symbol with the `.globl name` .
- `do_sym` - an interrupt handler.
- `has_error_code:req` - information about error code, The `:req` qualifier tells the assembler that the argument is required;
- `paranoid` - shows us how we need to check current mode;
- `shift_ist` - shows us what's stack to use;

As we can see our exceptions handlers are almost the same:

```
idtentry debug do_debug has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
idtentry int3 do_int3 has_error_code=0 paranoid=1 shift_ist=DEBUG_STACK
```

The differences are only in the global name and name of exceptions handlers. Now let's look how `idtentry` macro implemented. It starts from the two checks:

```
    .if \shift_ist != -1 && \paranoid == 0
    .error "using shift_ist requires paranoid=1"
    .endif

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

First check makes the check that an exceptions uses `Interrupt stack table` and `paranoid` is set, in other way it emits the erorr with the .error directive. The second `if` clause checks existence of an error code and calls `XCPT_FRAME` or `INTR_FRAME` macros depends on it. These macros just expand to the set of CFI directives which are used by `GNU AS` to manage call frames. The `CFI` directives are used only to generate dwarf2 unwind information for better backtraces and they don't change any code, so we will not go into detail about it and from this point I will skip all code which is related to these directives. In the next step we check error code again and push it on the stack if an exception has it with the:

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

The `pushq_cfi` macro defined in the arch/x86/include/asm/dwarf2.h and expands to the `pushq` instruction which pushes given error code:

```
    .macro pushq_cfi reg
    pushq \reg
    CFI_ADJUST_CFA_OFFSET 8
    .endm
```

Pay attention on the `$-1`. We already know that when an exception occrus, the processor pushes `ss`, `rsp`, `rflags`, `cs` and `rip` on the stack:

```
#define RIP        16*8
#define CS        17*8
#define EFLAGS    18*8
#define RSP        19*8
#define SS        20*8
```

With the `pushq \reg` we denote that place before the `RIP` will contain error code of an exception:

```
#define ORIG_RAX    15*8
```

The `ORIG_RAX` will contain error code of an exception, IRQ number on a hardware interrupt and system call number on system call entry. In the next step we can see thr `ALLOC_PT_GPREGS_ON_STACK` macro which allocates space for the 15 general purpose registers on the stack:

```
.macro ALLOC_PT_GPREGS_ON_STACK addskip=0
subq    $15*8+\addskip, %rsp
CFI_ADJUST_CFA_OFFSET 15*8+\addskip
.endm
```

After this we check `paranoid` and if it is set we check first three `CPL` bits. We compare it with the `3` and it allows us to know did we come from userspace or not:

```
.if \paranoid
  .if \paranoid == 1
    CFI_REMEMBER_STATE
    testl $3, CS(%rsp)
    jnz 1f
  .endif
  call paranoid_entry
.else
  call error_entry
.endif
```

If we came from userspace we jump on the label `1` which starts from the `call error_entry` instruction. The `error_entry` saves all registers in the `pt_regs` structure which presetens an interrupt/exception stack frame and defined in the arch/x86/include/uapi/asm/ptrace.h. It saves common and extra registers on the stack with the:

```
SAVE_C_REGS 8
SAVE_EXTRA_REGS 8
```

from `rdi` to `r15` and executes swapgs instruction. This instruction provides a method to for the Linux kernel to obtain a pointer to the kernel data structures and save the user's `gsbase`. After this we will exit from the `error_entry` with the `ret` instruction. After the `error_entry` finished to execute, since we came from userspace we need to switch on kernel interrupt stack:

```
    movq %rsp,%rdi
    call sync_regs
```

We just save all registers to the `error_entry` in the `error_entry`, we put address of the `pt_regs` to the `rdi` and call `sync_regs` function from the arch/x86/kernel/traps.c:

```
asmlinkage __visible notrace struct pt_regs *sync_regs(struct pt_regs *eregs)
{
    struct pt_regs *regs = task_pt_regs(current);
    *regs = *eregs;
    return regs;
}
```

This function switchs off the `IST` stack if we came from usermode. After this we switch on the stack which we got from the `sync_regs`:

```
movq %rax,%rsp
movq %rsp,%rdi
```

and put pointer of the `pt_regs` again in the `rdi`, and in the last step we call an exception handler:

```
call \do_sym
```

So, realy exceptions handlers are `do_debug` and `do_int3` functions. We will see these function in this part, but little later. First of all let's look on the preparations before a processor will transfer control to an interrupt handler. In another way if `paranoid` is set, but it is not 1, we call `paranoid_entry` which makes almost the same that `error_entry`, but it checks current mode with more slow but accurate way:

```
ENTRY(paranoid_entry)
    SAVE_C_REGS 8
    SAVE_EXTRA_REGS 8
    ...
    ...
    movl $MSR_GS_BASE,%ecx
    rdmsr
    testl %edx,%edx
    js 1f    /* negative -> in kernel */
    SWAPGS
    ...
    ...
    ret
END(paranoid_entry)
```

If `edx` wll be negative, we are in the kernel mode. As we store all registers on the stack, check that we are in the kernel mode, we need to setup `IST` stack if it is set for a given exception, call an exception handler and restore the exception stack:

```
    .if \shift_ist != -1
    subq $EXCEPTION_STKSZ, CPU_TSS_IST(\shift_ist)
    .endif

    call \do_sym

    .if \shift_ist != -1
    addq $EXCEPTION_STKSZ, CPU_TSS_IST(\shift_ist)
    .endif
```

The last step when an exception handler will finish it's work all registers will be restored from the stack with the `RESTORE_C_REGS` and `RESTORE_EXTRA_REGS` macros and control will be returned an interrupted task. That's all. Now we know about preparation before an interrupt/exception handler will start to execute and we can go directly to the implementation of the handlers.

Implementation of ainterrupts and exceptions handlers

Both handlers `do_debug` and `do_int3` defined in the arch/x86/kernel/traps.c source code file and have two similar things: All interrupts/exceptions handlers marked with the `dotraplinkage` prefix that expands to the:

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

which tells to compiler that something else uses this function (in our case these functions are called from the assembly interrupt preparation code). And also they takes two parameters:

- pointer to the `pt_regs` structure which contains registers of the interrupted task;
- error code.

First of all let's consider `do_debug` handler. This function starts from the getting previous state with the `ist_enter` function from the arch/x86/kernel/traps.c. We call it because we need to know, did we come to the interrupt handler from the kernel mode or user mode.

```
prev_state = ist_enter(regs);
```

The `ist_enter` function returns previous state context state and executes a couple preprartions before we continue to handle an exception. It starts from the check of the previous mode with the `user_mode_vm` macro. It takes `pt_regs` structure which contains a set of registers of the interrupted task and returns `1` if we came from userspace and `0` if we came from kernel space. According to the previous mode we execute `exception_enter` if we are from the userspace or inform RCU if we are from krenel space:

```
...
if (user_mode_vm(regs)) {
    prev_state = exception_enter();
} else {
    rcu_nmi_enter();
    prev_state = IN_KERNEL;
}
...
...
...
return prev_state;
```

After this we load the `DR6` debug registers to the `dr6` variable with the call of the `get_debugreg` macro from the arch/x86/include/asm/debugreg.h:

```
get_debugreg(dr6, 6);
dr6 &= ~DR6_RESERVED;
```

The `DR6` debug register is debug status register contains information about the reason for stopping the `#DB` or debug exception handler. After we loaded its value to the `dr6` variable we filter out all reserved bits (`4:12` bits). In the next step we check `dr6` register and previous state with the following `if` condition expression:

```
if (!dr6 && user_mode_vm(regs))
    user_icebp = 1;
```

If `dr6` does not show any reasons why we caught this trap we set `user_icebp` to one which means that user-code wants to get SIGTRAP signal. In the next step we check was it kmemcheck trap and if yes we go to exit:

```
if ((dr6 & DR_STEP) && kmemcheck_trap(regs))
    goto exit;
```

After we did all these checks, we clear the `dr6` register, clear the `DEBUGCTLMSR_BTF` flag which provides single-step on branches debugging, set `dr6` register for the current thread and increase `debug_stack_usage` per-cpu) variable with the:

```
set_debugreg(0, 6);
clear_tsk_thread_flag(tsk, TIF_BLOCKSTEP);
tsk->thread.debugreg6 = dr6;
debug_stack_usage_inc();
```

As we saved `dr6` , we can allow irqs:

```
static inline void preempt_conditional_sti(struct pt_regs *regs)
{
        preempt_count_inc();
        if (regs->flags & X86_EFLAGS_IF)
                local_irq_enable();
}
```

more about `local_irq_enabled` and related stuff you can read in the second part about interrupts handling in the Linux kernel. In the next step we check the previous mode was virtual 8086 and handle the trap:

```
if (regs->flags & X86_VM_MASK) {
    handle_vm86_trap((struct kernel_vm86_regs *) regs, error_code, X86_TRAP_DB);
      preempt_conditional_cli(regs);
      debug_stack_usage_dec();
      goto exit;
}
...
...
...
exit:
    ist_exit(regs, prev_state);
```

If we came not from the virtual 8086 mode, we need to check `dr6` register and previous mode as we did it above. Here we check if step mode debugging is enabled and we are not from the user mode, we enabled step mode debugging in the `dr6` copy in the current thread, set `TIF_SINGLE_STEP` falg and re-enable Trap flag for the user mode:

```
if ((dr6 & DR_STEP) && !user_mode(regs)) {
        tsk->thread.debugreg6 &= ~DR_STEP;
        set_tsk_thread_flag(tsk, TIF_SINGLESTEP);
        regs->flags &= ~X86_EFLAGS_TF;
}
```

Then we get `SIGTRAP` signal code:

```
si_code = get_si_code(tsk->thread.debugreg6);
```

and send it for user icebp traps:

```
if (tsk->thread.debugreg6 & (DR_STEP | DR_TRAP_BITS) || user_icebp)
    send_sigtrap(tsk, regs, error_code, si_code);
preempt_conditional_cli(regs);
debug_stack_usage_dec();
exit:
    ist_exit(regs, prev_state);
```

In the end we disabled `irqs`, decrement value of the `debug_stack_usage` and exit from the exception handler with the `ist_exit` function.

The second exception handler is `do_int3` defined in the same source code file - arch/x86/kernel/traps.c. In the `do_int3` we makes almost the same that in the `do_debug` handler. We get the previous state with the `ist_enter`, increment and decrement the `debug_stack_usage` per-cpu variable, enabled and disable local interrupts. But of course there is one difference between these two handlers. We need to lock and than sync processor cores during breakpoint patching.

That's all.

Conclusion

It is the end of the third part about interrupts and interrupt handling in the Linux kernel. We saw the initialization of the Interrupt descriptor table in the previous part with the `#DB` and `#BP` gates and started to dive into preparation before control will be transfered to an exception handler and implementation of some interrupt handlers in this part. In the next part we will continue to dive into this theme and will go next by the `setup_arch` function and will try to understand interrupts handling related stuff.

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

- Debug registers
- Intel 80385
- INT 3
- gcc
- TSS
- GNU assembly .error directive
- dwarf2
- CFI directives
- IRQ
- system call
- swapgs
- SIGTRAP
- Per-CPU variables
- kgdb
- ACPI
- Previous part

Interrupts and Interrupt Handling. Part 4.

Initialization of non-early interrupt gates

This is fourth part about an interrupts and exceptions handling in the Linux kernel and in the previous part we saw first early `#DB` and `#BP` exceptions handlers from the arch/x86/kernel/traps.c. We stopped on the right after the `early_trap_init` function that called in the `setup_arch` function which defined in the arch/x86/kernel/setup.c. In this part we will continue to dive into an interrupts and exceptions handling in the Linux kernel for `x86_64` and continue to do it from from the place where we left off in the last part. First thing which is related to the interrupts and exceptions handling is the setup of the `#PF` or page fault handler with the `early_trap_pf_init` function. Let's start from it.

Early page fault handler

The `early_trap_pf_init` function defined in the arch/x86/kernel/traps.c. It uses `set_intr_gate` macro that filles Interrupt Descriptor Table with the given entry:

```
void __init early_trap_pf_init(void)
{
#ifdef CONFIG_X86_64
         set_intr_gate(X86_TRAP_PF, page_fault);
#endif
}
```

This macro defined in the arch/x86/include/asm/desc.h. We already saw macros like this in the previous part - `set_system_intr_gate` and `set_intr_gate_ist`. This macro checks that given vector number is not greater than `255` (maximum vector number) and calls `_set_gate` function as `set_system_intr_gate` and `set_intr_gate_ist` did it:

```
#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)
```

The `set_intr_gate` macro takes two parameters:

- vector number of a interrupt;
- address of an interrupt handler;

In our case they are:

- `X86_TRAP_PF` - `14`;
- `page_fault` - the interrupt handler entry point.

The `X86_TRAP_PF` is the element of enum which defined in the arch/x86/include/asm/traprs.h:

```
enum {
    ...
    ...
    ...
```

```
    ...
    X86_TRAP_PF,            /* 14, Page Fault */
    ...
    ...
    ...
}
```

When the `early_trap_pf_init` will be called, the `set_intr_gate` will be expanded to the call of the `_set_gate` which will fill the `IDT` with the handler for the page fault. Now let's look on the implementation of the `page_fault` handler. The `page_fault` handler defined in the arch/x86/kernel/entry_64.S assembly source code file as all exceptions handlers. Let's look on it:

```
trace_idtentry page_fault do_page_fault has_error_code=1
```

We saw in the previous part how `#DB` and `#BP` handlers defined. They were defined with the `idtentry` macro, but here we can see `trace_idtentry`. This macro defined in the same source code file and depends on the `CONFIG_TRACING` kernel configuration option:

```
#ifdef CONFIG_TRACING
.macro trace_idtentry sym do_sym has_error_code:req
idtentry trace(\sym) trace(\do_sym) has_error_code=\has_error_code
idtentry \sym \do_sym has_error_code=\has_error_code
.endm
#else
.macro trace_idtentry sym do_sym has_error_code:req
idtentry \sym \do_sym has_error_code=\has_error_code
.endm
#endif
```

We will not dive into exceptions Tracing now. If `CONFIG_TRACING` is not set, we can see that `trace_idtentry` macro just expands to the normal `idtentry`. We already saw implementation of the `idtentry` macro in the previous part, so let's start from the `page_fault` exception handler.

As we can see in the `idtentry` definition, the handler of the `page_fault` is `do_page_fault` function which defined in the arch/x86/mm/fault.c and as all exceptions handlers it takes two arguments:

- `regs` - `pt_regs` structure that holds state of an interrupted process;
- `error_code` - error code of the page fault exception.

Let's look inside this function. First of all we read content of the cr2 control register:

```
dotraplinkage void notrace
do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    unsigned long address = read_cr2();
    ...
    ...
    ...
}
```

This register contains a linear address which caused `page fault`. In the next step we make a call of the `exception_enter` function from the include/linux/context_tracking.h. The `exception_enter` and `exception_exit` are functions from context tracking subsytem in the Linux kernel used by the RCU to remove its dependency on the timer tick while a processor runs in userspace. Almost in the every exception handler we will see similar code:

```
enum ctx_state prev_state;
```

```
prev_state = exception_enter();
...
... // exception handler here
...
exception_exit(prev_state);
```

The `exception_enter` function checks that `context tracking` is enabled with the `context_tracking_is_enabled` and if it is in enabled state, we get previous context with te `this_cpu_read` (more about `this_cpu_*` operations you can read in the Documentation). After this it calls `context_tracking_user_exit` function which informs that Inform the context tracking that the processor is exiting userspace mode and entering the kernel:

```
static inline enum ctx_state exception_enter(void)
{
        enum ctx_state prev_ctx;

        if (!context_tracking_is_enabled())
                return 0;

        prev_ctx = this_cpu_read(context_tracking.state);
        context_tracking_user_exit();

        return prev_ctx;
}
```

The state can be one of the:

```
enum ctx_state {
    IN_KERNEL = 0,
    IN_USER,
} state;
```

And in the end we return previous context. Between the `exception_enter` and `exception_exit` we call actual page fault handler:

```
__do_page_fault(regs, error_code, address);
```

The `__do_page_fault` is defined in the same source code file as `do_page_fault` - arch/x86/mm/fault.c. In the bingging of the `__do_page_fault` we check state of the kmemcheck checker. The `kmemcheck` detects warns about some uses of uninitialized memory. We need to check it because page fault can be caused by kmemcheck:

```
if (kmemcheck_active(regs))
        kmemcheck_hide(regs);
    prefetchw(&mm->mmap_sem);
```

After this we can see the call of the `prefetchw` which executes instruction with the same name which fetches X86_FEATURE_3DNOW to get exclusive cache line. The main purpose of prefetching is to hide the latency of a memory access. In the next step we check that we got page fault not in the kernel space with the following conditiion:

```
if (unlikely(fault_in_kernel_space(address))) {
...
...
...
}
```

where `fault_in_kernel_space` is:

```
static int fault_in_kernel_space(unsigned long address)
{
        return address >= TASK_SIZE_MAX;
}
```

The `TASK_SIZE_MAX` macro expands to the:

```
#define TASK_SIZE_MAX   ((1UL << 47) - PAGE_SIZE)
```

or `0x00007ffffffff000` . Pay attention on `unlikely` macro. There are two macros in the Linux kernel:

```
#define likely(x)      __builtin_expect(!!(x), 1)
#define unlikely(x)    __builtin_expect(!!(x), 0)
```

You can often find these macros in the code of the Linux kernel. Main purpose of these macros is optimization. Sometimes this situation is that we need to check the condition of the code and we know that it will rarely be `true` or `false` . With these macros we can tell to the compiler about this. For example

```
static int proc_root_readdir(struct file *file, struct dir_context *ctx)
{
        if (ctx->pos < FIRST_PROCESS_ENTRY) {
                int error = proc_readdir(file, ctx);
                if (unlikely(error <= 0))
                        return error;
...
...
...
}
```

Here we can see `proc_root_readdir` function which will be called when the Linux VFS needs to read the `root` directory contents. If condition marked with `unlikely` , compiler can put `false` code right after branching. Now let's back to the our address check. Comparison between the given address and the `0x00007ffffffff000` will give us to know, was page fault in the kernel mode or user mode. After this check we know it. After this `__do_page_fault` routine will try to understand the problem that provoked page fault exception and then will pass address to the approprite routine. It can be `kmemcheck` fault, spurious fault, kprobes fault and etc. Will not dive into implementation details of the page fault exception handler in this part, because we need to know many different concepts which are provided by the Linux kerne, but will see it in the chapter about the memory management in the Linux kernel.

Back to start_kernel

There are many different function calls after the `early_trap_pf_init` in the `setup_arch` function from different kernel subsystems, but there are no one interrupts and exceptions handling related. So, we have to go back where we came from - `start_kernel` function from the init/main.c. The first things after the `setup_arch` is the `trap_init` function from the arch/x86/kernel/traps.c. This function makes initialization of the remaining exceptions handlers (remember that we already setup 3 handlres for the `#DB` - debug exception, `#BP` - breakpoint exception and `#PF` - page fault exception). The `trap_init` function starts from the check of the Extended Industry Standard Architecture:

```
#ifdef CONFIG_EISA
        void __iomem *p = early_ioremap(0x0FFFD9, 4);

        if (readl(p) == 'E' + ('I'<<8) + ('S'<<16) + ('A'<<24))
                EISA_bus = 1;
        early_iounmap(p, 4);
```

```
#endif
```

Note that it depends on the `CONFIG_EISA` kernel configuration parameter which represetns `EISA` support. Here we use `early_ioremap` function to map `I/O` memory on the page tables. We use `readl` function to read first `4` bytes from the mapped region and if they are equal to `EISA` string we set `EISA_bus` to one. In the end we just unmap previously mapped region. More about `early_ioremap` you can read in the part which describes Fix-Mapped Addresses and ioremap.

After this we start to fill the `Interrupt Descriptor Table` with the different interrupt gates. First of all we set `#DE` or `Divide Error` and `#NMI` or `Non-maskable Interrupt`:

```
set_intr_gate(X86_TRAP_DE, divide_error);
set_intr_gate_ist(X86_TRAP_NMI, &nmi, NMI_STACK);
```

We use `set_intr_gate` macro to set the interrupt gate for the `#DE` exception and `set_intr_gate_ist` for the `#NMI`. You can remember that we already used these macros when we have set the interrupts gates for the page fault handler, debug handler and etc, you can find explanation of it in the previous part. After this we setup exception gates for the following exceptions:

```
set_system_intr_gate(X86_TRAP_OF, &overflow);
set_intr_gate(X86_TRAP_BR, bounds);
set_intr_gate(X86_TRAP_UD, invalid_op);
set_intr_gate(X86_TRAP_NM, device_not_available);
```

Here we can see:

- `#OF` or `Overflow` exception. This exception indicates that an overflow trap occurred when an special INTO instruction was executed;
- `#BR` or `BOUND Range exceeded` exception. This exception indeicates that a `BOUND-range-exceed` fault occured when a BOUND instruction was executed;
- `#UD` or `Invalid Opcode` exception. Occurs when a processor attempted to execute invalid or reserved opcode, processor attempted to execute instruction with invalid operand(s) and etc;
- `#NM` or `Device Not Available` exception. Occurs when the processor tries to execute `x87 FPU` floating point instruction while `EM` flag in the control register `cr0` was set.

In the next step we set the interrupt gate for the `#DF` or `Double fault` exception:

```
set_intr_gate_ist(X86_TRAP_DF, &double_fault, DOUBLEFAULT_STACK);
```

This exception occurs when processor detected a second exception while calling an exception handler for a prior exception. In usual way when the processor detects another exception while trying to call an exception handler, the two exceptions can be handled serially. If the processor cannot handle them serially, it signals the double-fault or `#DF` exception.

The following set of the interrupt gates is:

```
set_intr_gate(X86_TRAP_OLD_MF, &coprocessor_segment_overrun);
set_intr_gate(X86_TRAP_TS, &invalid_TSS);
set_intr_gate(X86_TRAP_NP, &segment_not_present);
set_intr_gate_ist(X86_TRAP_SS, &stack_segment, STACKFAULT_STACK);
set_intr_gate(X86_TRAP_GP, &general_protection);
set_intr_gate(X86_TRAP_SPURIOUS, &spurious_interrupt_bug);
set_intr_gate(X86_TRAP_MF, &coprocessor_error);
set_intr_gate(X86_TRAP_AC, &alignment_check);
```

Here we can see setup for the following exception handlers:

* `#CSO` or `Coprocessor Segment Overrun` - this exception indicates that math coprocessor of an old processor detected a page or segment violation. Modern processors do not generate this exception
* `#TS` or `Invalid TSS` exception - indicates that there was an error related to the Task State Segment.
* `#NP` or `Segement Not Present` exception indicates that the `present flag` of a segment or gate descriptor is clear during attempt to load one of `cs`, `ds`, `es`, `fs`, or `gs` register.
* `#SS` or `Stack Fault` exception indicates one of the stack related conditions was detected, for example a not-present stack segment is detected when attempting to load the `ss` register.
* `#GP` or `General Protection` exception indicates that the processor detected one of a class of protection violations called general-protection violations. There are many different conditions that can cause general-procetion exception. For example loading the `ss`, `ds`, `es`, `fs`, or `gs` register with a segment selector for a system segment, writing to a code segment or a read-only data segment, referencing an entry in the `Interrupt Descriptor Table` (following an interrupt or exception) that is not an interrupt, trap, or task gate and many many more.
* `Spurious Interrupt` - a hardware interrupt that is unwanted.
* `#MF` or `x87 FPU Floating-Point Error` exception caused when the x87 FPU has detected a floating point error.
* `#AC` or `Alignment Check` exception Indicates that the processor detected an unaligned memory operand when alignment checking was enabled.

After that we setup this exception gates, we can see setup of the `Machine-Check` exception:

```
#ifdef CONFIG_X86_MCE
    set_intr_gate_ist(X86_TRAP_MC, &machine_check, MCE_STACK);
#endif
```

Note that it depends on the `CONFIG_X86_MCE` kernel configuration option and indicates that the processor detected an internal machine error or a bus error, or that an external agent detected a bus error. The next exception gate is for the SIMD Floating-Point exception:

```
set_intr_gate(X86_TRAP_XF, &simd_coprocessor_error);
```

which indicates the processor has detected an `SSE` or `SSE2` or `SSE3` SIMD floating-point exception. There are six classes of numeric exception conditions that can occur while executing an SIMD floating-point instruction:

* Invalid operation
* Divide-by-zero
* Denormal operand
* Numeric overflow
* Numeric underflow
* Inexact result (Precision)

In the next step we fill the `used_vectors` array which defined in the arch/x86/include/asm/desc.h header file and represents `bitmap`:

```
DECLARE_BITMAP(used_vectors, NR_VECTORS);
```

of the first `32` interrupts (more about bitmaps in the Linux kernel you can read in the part which describes cpumasks and bitmaps)

```
for (i = 0; i < FIRST_EXTERNAL_VECTOR; i++)
```

```
    set_bit(i, used_vectors)
```

where `FIRST_EXTERNAL_VECTOR` is:

```
#define FIRST_EXTERNAL_VECTOR           0x20
```

After this we setup the interrupt gate for the `ia32_syscall` and add `0x80` to the `used_vectors` bitmap:

```
#ifdef CONFIG_IA32_EMULATION
        set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
        set_bit(IA32_SYSCALL_VECTOR, used_vectors);
#endif
```

There is `CONFIG_IA32_EMULATION` kernel configuration option on `x86_64` Linux kernels. This option provides ability to execute 32-bit processes in compatibility-mode. In the next parts we will see how it works, in the meantime we need only to know that there is yet another interrupt gate in the `IDT` with the vector number `0x80`. In the next step we maps `IDT` to the fixmap area:

```
__set_fixmap(FIX_RO_IDT, __pa_symbol(idt_table), PAGE_KERNEL_RO);
idt_descr.address = fix_to_virt(FIX_RO_IDT);
```

and write its address to the `idt_descr.address` (more about fix-mapped addresses you can read in the second part of the Linux kernel memory management chapter). After this we can see the call of the `cpu_init` function that defined in the arch/x86/kernel/cpu/common.c. This function makes initialization of the all `per-cpu` state. In the beginnig of the `cpu_init` we do the following things: First of all we wait while current cpu is initialized and than we call the `cr4_init_shadow` function which stores shadow copy of the `cr4` control register for the current cpu and load CPU microcode if need with the following function calls:

```
wait_for_master_cpu(cpu);
cr4_init_shadow();
load_ucode_ap();
```

Next we get the `Task State Segement` for the current cpu and `orig_ist` structure which represents origin `Interrupt Stack Table` values with the:

```
t = &per_cpu(cpu_tss, cpu);
oist = &per_cpu(orig_ist, cpu);
```

As we got values of the `Task State Segement` and `Interrupt Stack Table` for the current processor, we clear following bits in the `cr4` control register:

```
cr4_clear_bits(X86_CR4_VME|X86_CR4_PVI|X86_CR4_TSD|X86_CR4_DE);
```

with this we disable `vm86` extension, virtual interrupts, timestamp (RDTSC can only be executed with the highest privilege) and debug extension. After this we reload the `Glolbal Descripto Table` and `Interrupt Descriptor table` with the:

```
    switch_to_new_gdt(cpu);
    loadsegment(fs, 0);
    load_current_idt();
```

After this we setup array of the Thread-Local Storage Descriptors, configure NX and load CPU microcode. Now is time to setup and load `per-cpu` Task State Segements. We are going in a loop through the all exception stack which is `N_EXCEPTION_STACKS` or `4` and fill it with `Interrupt Stack Tables`:

```
    if (!oist->ist[0]) {
        char *estacks = per_cpu(exception_stacks, cpu);

        for (v = 0; v < N_EXCEPTION_STACKS; v++) {
            estacks += exception_stack_sizes[v];
            oist->ist[v] = t->x86_tss.ist[v] =
                    (unsigned long)estacks;
            if (v == DEBUG_STACK-1)
                per_cpu(debug_stack_addr, cpu) = (unsigned long)estacks;
        }
    }
```

As we have filled `Task State Segements` with the `Interrupt Stack Tables` we can set `TSS` descriptor for the current processor and load it with the:

```
set_tss_desc(cpu, t);
load_TR_desc();
```

where `set_tss_desc` macro from the arch/x86/include/asm/desc.h writes given descriptor to the `Global Descriptor Table` of the given processor:

```
#define set_tss_desc(cpu, addr) __set_tss_desc(cpu, GDT_ENTRY_TSS, addr)
static inline void __set_tss_desc(unsigned cpu, unsigned int entry, void *addr)
{
        struct desc_struct *d = get_cpu_gdt_table(cpu);
        tss_desc tss;
        set_tssldt_descriptor(&tss, (unsigned long)addr, DESC_TSS,
                              IO_BITMAP_OFFSET + IO_BITMAP_BYTES +
                              sizeof(unsigned long) - 1);
        write_gdt_entry(d, entry, &tss, DESC_TSS);
}
```

and `load_TR_desc` macro expands to the `ltr` or `Load Task Register` instruction:

```
#define load_TR_desc()                          native_load_tr_desc()
static inline void native_load_tr_desc(void)
{
        asm volatile("ltr %w0"::"q" (GDT_ENTRY_TSS*8));
}
```

In the end of the `trap_init` function we can see the following code:

```
set_intr_gate_ist(X86_TRAP_DB, &debug, DEBUG_STACK);
set_system_intr_gate_ist(X86_TRAP_BP, &int3, DEBUG_STACK);
...
...
...
#ifdef CONFIG_X86_64
        memcpy(&nmi_idt_table, &idt_table, IDT_ENTRIES * 16);
        set_nmi_gate(X86_TRAP_DB, &debug);
        set_nmi_gate(X86_TRAP_BP, &int3);
#endif
```

Here we copy `idt_table` to the `nmi_dit_table` and setup exception handlers for the `#DB` or `Debug exception` and `#BR` or `Breakpoint exception`. You can remember that we already set these interrupt gates in the previous part, so why do we need to setup it again? We setup it again because when we initialized it before in the `early_trap_init` function, the `Task State Segement` was not ready yet, but now it is ready after the call of the `cpu_init` function.

That's all. Soon we will consider all handlers of these interrupts/exceptions.

Conclusion

It is the end of the fourth part about interrupts and interrupt handling in the Linux kernel. We saw the initialization of the Task State Segment in this part and initialization of the different interrupt handlers as `Divide Error`, `Page Fault` excetpion and etc. You can noted that we saw just initialization stuf, and will dive into details about handlers for these exceptions. In the next part we will start to do it.

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

- page fault
- Interrupt Descriptor Table
- Tracing
- cr2
- RCU
- this*cpu** operations
- kmemcheck
- prefetchw
- 3DNow
- CPU caches
- VFS
- Linux kernel memory management
- Fix-Mapped Addresses and ioremap
- Extended Industry Standard Architecture
- INT isntruction
- INTO
- BOUND
- opcode
- control register
- x87 FPU
- MCE exception
- SIMD
- cpumasks and bitmaps
- NX
- Task State Segment
- Previous part

Interrupts and Interrupt Handling. Part 5.

Implementation of exception handlers

This is the fifth part about an interrupts and exceptions handling in the Linux kernel and in the previous part we stopped on the setting of interrupt gates to the Interrupt descriptor Table. We did it in the `trap_init` function from the arch/x86/kernel/traps.c source code file. We saw only setting of these interrupt gates in the previous part and in the current part we will see implementation of the exception handlers for these gates. The preparation before an exception handler will be executed is in the arch/x86/entry/entry_64.S assembly file and occurs in the idtentry macro that defines exceptions entry points:

```
idtentry divide_error                    do_divide_error                  has_error_code=0
idtentry overflow                        do_overflow                      has_error_code=0
idtentry invalid_op                        do_invalid_op                  has_error_code=0
idtentry bounds                            do_bounds                      has_error_code=0
idtentry device_not_available            do_device_not_available          has_error_code=0
idtentry coprocessor_segment_overrun    do_coprocessor_segment_overrun has_error_code=0
idtentry invalid_TSS                    do_invalid_TSS                  has_error_code=1
idtentry segment_not_present            do_segment_not_present          has_error_code=1
idtentry spurious_interrupt_bug            do_spurious_interrupt_bug      has_error_code=0
idtentry coprocessor_error                do_coprocessor_error          has_error_code=0
idtentry alignment_check                do_alignment_check              has_error_code=1
idtentry simd_coprocessor_error            do_simd_coprocessor_error      has_error_code=0
```

The `idtentry` macro does following preparation before an actual exception handler (`do_divide_error` for the `divide_error` , `do_overflow` for the `overflow` and etc.) will get control. In another words the `idtentry` macro allocates place for the registers (pt_regs structure) on the stack, pushes dummy error code for the stack consistency if an interrupt/exception has no error code, checks the segment selector in the `cs` segment register and switches depends on the previous state(userspace or kernelspace). After all of these preparations it makes a call of an actual interrupt/exception handler:

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
    ...
    ...
    ...
    call    \do_sym
    ...
    ...
    ...
END(\sym)
.endm
```

After an exception handler will finish its work, the `idtentry` macro restores stack and general purpose registers of an interrupted task and executes iret instruction:

```
ENTRY(paranoid_exit)
    ...
    ...
    ...
    RESTORE_EXTRA_REGS
    RESTORE_C_REGS
    REMOVE_PT_GPREGS_FROM_STACK 8
    INTERRUPT_RETURN
END(paranoid_exit)
```

where `INTERRUPT_RETURN` is:

```
#define INTERRUPT_RETURN    jmp native_iret
...
ENTRY(native_iret)
.global native_irq_return_iret
native_irq_return_iret:
iretq
```

More about the `idtentry` macro you can read in the thirt part of the http://0xax.gitbooks.io/linux-insides/content/interrupts/interrupts-3.html chapter. Ok, now we saw the preparation before an exception handler will be executed and now time to look on the handlers. First of all let's look on the following handlers:

- divide_error
- overflow
- invalid_op
- coprocessor_segment_overrun
- invalid_TSS
- segment_not_present
- stack_segment
- alignment_check

All these handlers defined in the arch/x86/kernel/traps.c source code file with the `DO_ERROR` macro:

```
DO_ERROR(X86_TRAP_DE,     SIGFPE,  "divide error",                divide_error)
DO_ERROR(X86_TRAP_OF,     SIGSEGV, "overflow",                    overflow)
DO_ERROR(X86_TRAP_UD,     SIGILL,  "invalid opcode",              invalid_op)
DO_ERROR(X86_TRAP_OLD_MF, SIGFPE,  "coprocessor segment overrun", coprocessor_segment_overrun)
DO_ERROR(X86_TRAP_TS,     SIGSEGV, "invalid TSS",                 invalid_TSS)
DO_ERROR(X86_TRAP_NP,     SIGBUS,  "segment not present",         segment_not_present)
DO_ERROR(X86_TRAP_SS,     SIGBUS,  "stack segment",               stack_segment)
DO_ERROR(X86_TRAP_AC,     SIGBUS,  "alignment check",             alignment_check)
```

As we can see the `DO_ERROR` macro takes 4 parameters:

- Vector number of an interrupt;
- Signal number which will be sent to the interrupted process;
- String which describes an exception;
- Exception handler entry point.

This macro defined in the same souce code file and expands to the function with the `do_handler` name:

```
#define DO_ERROR(trapnr, signr, str, name)                              \
dotraplinkage void do_##name(struct pt_regs *regs, long error_code)     \
{                                                                       \
        do_error_trap(regs, error_code, str, trapnr, signr);            \
}
```

Note on the `##` tokens. This is special feature - GCC macro Concatenation which concatenates two given strings. For example, first `DO_ERROR` in our example will expands to the:

```
dotraplinkage void do_divide_error(struct pt_regs *regs, long error_code)     \
{
    ...
}
```

We can see that all functions which are generated by the `DO_ERROR` macro just make a call of the `do_error_trap` function from the arch/x86/kernel/traps.c. Let's look on implementation of the `do_error_trap` function.

Trap handlers

The `do_error_trap` function starts and ends from the two following functions:

```
enum ctx_state prev_state = exception_enter();
...
...
...
exception_exit(prev_state);
```

from the include/linux/context_tracking.h. The context tracking in the Linux kernel subsystem which provide kernel boundaries probes to keep track of the transitions between level contexts with two basic initial contexts: `user` or `kernel`. The `exception_enter` function checks that context tracking is enabled. After this if it is enabled, the `exception_enter` reads previous context and compares it with the `CONTEXT_KERNEL`. If the previous context is `user`, we call `context_tracking_exit` function from the kernel/context_tracking.c which inform the context tracking subsystem that a processor is exiting user mode and entering the kernel mode:

```
if (!context_tracking_is_enabled())
    return 0;

prev_ctx = this_cpu_read(context_tracking.state);
if (prev_ctx != CONTEXT_KERNEL)
    context_tracking_exit(prev_ctx);

return prev_ctx;
```

If previous context is non `user`, we just return it. The `pre_ctx` has `enum ctx_state` type which defined in the include/linux/context_tracking_state.h and looks as:

```
enum ctx_state {
    CONTEXT_KERNEL = 0,
    CONTEXT_USER,
    CONTEXT_GUEST,
} state;
```

The second function is `exception_exit` defined in the same include/linux/context_tracking.h file and checks that context tracking is enabled and call the `contert_tracking_enter` function if the previous context was `user`:

```
static inline void exception_exit(enum ctx_state prev_ctx)
{
    if (context_tracking_is_enabled()) {
        if (prev_ctx != CONTEXT_KERNEL)
            context_tracking_enter(prev_ctx);
    }
}
```

The `context_tracking_enter` function informs the context tracking subsystem that a processor is going to enter to the user mode from the kernel mode. We can see the following code between the `exception_enter` and `exception_exit`:

```
if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) !=
        NOTIFY_STOP) {
    conditional_sti(regs);
    do_trap(trapnr, signr, str, regs, error_code,
```

```
        fill_trap_info(regs, signr, trapnr, &info));
}
```

First of all it calls the `notify_die` function which defined in the kernel/notifier.c. To get notified for kernel panic, kernel oops, Non-Maskable Interrupt or other events the caller needs to insert itself in the `notify_die` chain and the `notify_die` function does it. The Linux kernel has special mechanism that allows kernel to ask when something happens and this mechanism called `notifiers` or `notifier chains`. This mechanism used for example for the `USB` hotplug events (look on the drivers/usb/core/notify.c), for the memory hotplug (look on the include/linux/memory.h, the `hotplug_memory_notifier` macro and etc...), system reboots and etc. A notifier chain is thus a simple, singly-linked list. When a Linux kernel subsystem wants to be notified of specific events, it fills out a special `notifier_block` structure and passes it to the `notifier_chain_register` function. An event can be sent with the call of the `notifier_call_chain` function. First of all the `notify_die` function fills `die_args` structure with the trap number, trap string, registers and other values:

```
struct die_args args = {
       .regs   = regs,
       .str    = str,
       .err    = err,
       .trapnr = trap,
       .signr  = sig,
}
```

and returns the result of the `atomic_notifier_call_chain` function with the `die_chain`:

```
static ATOMIC_NOTIFIER_HEAD(die_chain);
return atomic_notifier_call_chain(&die_chain, val, &args);
```

which just expands to the `atomit_notifier_head` structure that contains lock and `notifier_block`:

```
struct atomic_notifier_head {
        spinlock_t lock;
        struct notifier_block __rcu *head;
};
```

The `atomic_notifier_call_chain` function calls each function in a notifier chain in turn and returns the value of the last notifier function called. If the `notify_die` in the `do_error_trap` does not return `NOTIFY_STOP` we execute `conditional_sti` function from the arch/x86/kernel/traps.c that checks the value of the interrupt flag and enables interrupt depends on it:

```
static inline void conditional_sti(struct pt_regs *regs)
{
        if (regs->flags & X86_EFLAGS_IF)
                local_irq_enable();
}
```

more about `local_irq_enable` macro you can read in the second part of this chapter. The next and last call in the `do_error_trap` is the `do_trap` function. First of all the `do_trap` function defined the `tsk` variable which has `trak_struct` type and represents the current interrupted process. After the definition of the `tsk`, we can see the call of the `do_trap_no_signal` function:

```
struct task_struct *tsk = current;

if (!do_trap_no_signal(tsk, trapnr, str, regs, error_code))
    return;
```

The `do_trap_no_signal` function makes two checks:

- Did we come from the Virtual 8086 mode;
- Did we come from the kernelspace.

```
if (v8086_mode(regs)) {
    ...
}

if (!user_mode(regs)) {
    ...
}

return -1;
```

We will not consider first case because the long mode does not support the Virtual 8086 mode. In the second case we invoke `fixup_exception` function which will try to recover a fault and `die` if we can't:

```
if (!fixup_exception(regs)) {
    tsk->thread.error_code = error_code;
    tsk->thread.trap_nr = trapnr;
    die(str, regs, error_code);
}
```

The `die` function defined in the arch/x86/kernel/dumpstack.c source code file, prints useful information about stack, registers, kernel modules and caused kernel oops. If we came from the userspace the `do_trap_no_signal` function will return `-1` and the execution of the `do_trap` function will continue. If we passed through the `do_trap_no_signal` function and did not exit from the `do_trap` after this, it means that previous context was - `user`. Most exceptions caused by the processor are interpreted by Linux as error conditions, for example division by zero, invalid opcode and etc. When an exception occurs the Linux kernel sends a signal to the interrupted process that caused the exception to notify it of an incorrect condition. So, in the `do_trap` function we need to send a signal with the given number (`SIGFPE` for the divide error, `SIGILL` for the overflow exception and etc...). First of all we save error code and vector number in the current interrupts process with the filling `thread.error_code` and `thread_trap_nr` :

```
tsk->thread.error_code = error_code;
tsk->thread.trap_nr = trapnr;
```

After this we make a check do we need to print information about unhandled signals for the interrupted process. We check that `show_unhandled_signals` variable is set, that `unhandled_signal` function from the kernel/signal.c will return unhandled signal(s) and printk rate limit:

```
#ifdef CONFIG_X86_64
    if (show_unhandled_signals && unhandled_signal(tsk, signr) &&
        printk_ratelimit()) {
        pr_info("%s[%d] trap %s ip:%lx sp:%lx error:%lx",
            tsk->comm, tsk->pid, str,
            regs->ip, regs->sp, error_code);
        print_vma_addr(" in ", regs->ip);
        pr_cont("\n");
    }
#endif
```

And send a given signal to interrupted process:

```
force_sig_info(signr, info ?: SEND_SIG_PRIV, tsk);
```

This is the end of the `do_trap`. We just saw generic implementation for eight different exceptions which are defined with the `DO_ERROR` macro. Now let's look on another exception handlers.

Double fault

The next exception is `#DF` or `Double fault`. This exception occurrs when the processor detected a second exception while calling an exception handler for a prior exception. We set the trap gate for this exception in the previous part:

```
set_intr_gate_ist(X86_TRAP_DF, &double_fault, DOUBLEFAULT_STACK);
```

Note that this exception runs on the `DOUBLEFAULT_STACK` Interrupt Stack Table which has index - `1`:

```
#define DOUBLEFAULT_STACK 1
```

The `double_fault` is handler for this exception and defined in the arch/x86/kernel/traps.c. The `double_fault` handler starts from the definition of two variables: string that describes excetpion and interrupted process, as other exception handlers:

```
static const char str[] = "double fault";
struct task_struct *tsk = current;
```

The handler of the double fault exception splitted on two parts. The first part is the check which checks that a fault is a `non-IST` fault on the `espfix64` stack. Actually the `iret` instruction restores only the bottom `16` bits when returning to a `16` bit segment. The `espfix` feature solves this problem. So if the `non-IST` fault on the espfix64 stack we modify the stack to make it look like `General Protection Fault`:

```
struct pt_regs *normal_regs = task_pt_regs(current);

memmove(&normal_regs->ip, (void *)regs->sp, 5*8);
ormal_regs->orig_ax = 0;
regs->ip = (unsigned long)general_protection;
regs->sp = (unsigned long)&normal_regs->orig_ax;
return;
```

In the second case we do almost the same that we did in the previous excetpion handlers. The first is the call of the `ist_enter` function that discards previous context, `user` in our case:

```
ist_enter(regs);
```

And after this we fill the interrupted process with the vector number of the `Double fault` excetpion and error code as we did it in the previous handlers:

```
tsk->thread.error_code = error_code;
tsk->thread.trap_nr = X86_TRAP_DF;
```

Next we print useful information about the double fault (PID number, registers content):

```
#ifdef CONFIG_DOUBLEFAULT
    df_debug(regs, error_code);
#endif
```

And die:

```
    for (;;)
        die(str, regs, error_code);
```

That's all.

Device not available exception handler

The next exception is the `#NM` or `Device not available`. The `Device not available` exception can occur depending on these things:

- The processor executed an x87 FPU floating-point instruction while the EM flag in control register `cr0` was set;
- The processor executed a `wait` or `fwait` instruction while the `MP` and `TS` flags of register `cr0` were set;
- The processor executed an x87 FPU, MMX or SSE instruction while the `TS` falg in control register `cr0` was set and the `EM` flag is clear.

The handler of the `Device not available` exception is the `do_device_not_available` function and it defined in the arch/x86/kernel/traps.c source code file too. It starts and ends from the getting of the previous context, as other traps which we saw in the beginning of this part:

```
enum ctx_state prev_state;
prev_state = exception_enter();
...
...
...
exception_exit(prev_state);
```

In the next step we check that `FPU` is not eager:

```
BUG_ON(use_eager_fpu());
```

When we switch into a task or interrupt we may avoid loading the `FPU` state. If a task will use it, we catch `Device not Available exception` exception. If we loading the `FPU` state during task switching, the `FPU` is eager. In the next step we check `cr0` control register on the `EM` flag which can show us is `x87` floating point unit present (flag clear) or not (flag set):

```
#ifdef CONFIG_MATH_EMULATION
    if (read_cr0() & X86_CR0_EM) {
        struct math_emu_info info = { };

        conditional_sti(regs);

        info.regs = regs;
        math_emulate(&info);
        exception_exit(prev_state);
        return;
    }
#endif
```

If the `x87` floating point unit not presented, we enable interrupts with the `conditional_sti`, fill the `math_emu_info` (defined in the arch/x86/include/asm/math_emu.h) structure with the registers of an interrupt task and call `math_emulate` function from the arch/x86/math-emu/fpu_entry.c. As you can understand from function's name, it emulates `X87 FPU` unit (more about the `x87` we will know in the special chapter). In other way, if `X86_CR0_EM` flag is clear which means that `x87 FPU` unit is presented, we call the `fpu__restore` function from the arch/x86/kernel/fpu/core.c which copies the `FPU` registers from the `fpustate` to the live hardware registers. After this `FPU` instructions can be used:

```
fpu__restore(&current->thread.fpu);
```

General protection fault exception handler

The next exception is the `#GP` or `General protection fault`. This exception occurs when the processor detected one of a class of protection violations called `general-protection violations`. It can be:

- Exceeding the segment limit when accessing the `cs`, `ds`, `es`, `fs` or `gs` segments;
- Loading the `ss`, `ds`, `es`, `fs` or `gs` register with a segment selector for a system segment.;
- Violating any of the privilege rules;
- and other...

The exception handler for this exception is the `do_general_protection` from the arch/x86/kernel/traps.c. The `do_general_protection` function starts and ends as other exception handlers from the getting of the previous context:

```
prev_state = exception_enter();
...
exception_exit(prev_state);
```

After this we enable interrupts if they were disabled and check that we came from the Virtual 8086 mode:

```
conditional_sti(regs);

if (v8086_mode(regs)) {
    local_irq_enable();
    handle_vm86_fault((struct kernel_vm86_regs *) regs, error_code);
    goto exit;
}
```

As long mode does not support this mode, we will not consider exception handling for this case. In the next step check that previous mode was kernel mode and try to fix the trap. If we can't fix the current general protection fault exception we fill the interrupted process with the vector number and error code of the exception and add it to the `notify_die` chain:

```
if (!user_mode(regs)) {
    if (fixup_exception(regs))
        goto exit;

    tsk->thread.error_code = error_code;
    tsk->thread.trap_nr = X86_TRAP_GP;
    if (notify_die(DIE_GPF, "general protection fault", regs, error_code,
               X86_TRAP_GP, SIGSEGV) != NOTIFY_STOP)
        die("general protection fault", regs, error_code);
    goto exit;
}
```

If we can fix exception we go to the `exit` label which exits from exception state:

```
exit:
    exception_exit(prev_state);
```

If we came from user mode we send `SIGSEGV` signal to the interrupted process from user mode as we did it in the `do_trap` function:

```
if (show_unhandled_signals && unhandled_signal(tsk, SIGSEGV) &&
        printk_ratelimit()) {
    pr_info("%s[%d] general protection ip:%lx sp:%lx error:%lx",
        tsk->comm, task_pid_nr(tsk),
        regs->ip, regs->sp, error_code);
    print_vma_addr(" in ", regs->ip);
    pr_cont("\n");
}

force_sig_info(SIGSEGV, SEND_SIG_PRIV, tsk);
```

That's all.

Conclusion

It is the end of the fifth part of the Interrupts and Interrupt Handling chapter and we saw implementation of some interrupt handlers in this part. In the next part we will continue to dive into interrupt and exception handlers and will see handler for the Non-Maskable Interrupts, handling of the math coprocessor and SIMD coprocessor exceptions 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

- Interrupt descriptor Table
- iret instruction
- GCC macro Concatenation
- kernel panic
- kernel oops
- Non-Maskable Interrupt
- hotplug
- interrupt flag
- long mode
- signal
- printk
- coprocessor
- SIMD
- Interrupt Stack Table
- PID
- x87 FPU
- control register
- MMX
- Previous part

Interrupts and Interrupt Handling. Part 6.

Non-maskable interrupt handler

It is sixth part of the Interrupts and Interrupt Handling in the Linux kernel chapter and in the previous part we saw implementation of some exception handlers for the General Protection Fault exception, divide exception, invalid opcode exceptions and etc. As I wrote in the previous part we will see implementations of the rest exceptions in this part. We will see implementation of the following handlers:

- Non-Maskable interrupt;
- BOUND Range Exceeded Exception;
- Coprocessor exception;
- SIMD coprocessor exception.

in this part. So, let's start.

Non-Maskable interrupt handling

A Non-Maskable interrupt is a hardware interrupt that cannot be ignore by standard masking techniques. In a general way, a non-maskable interrupt can be generated in either of two ways:

- External hardware asserts the non-maskable interrupt pin on the CPU.
- The processor receives a message on the system bus or the APIC serial bus with a delivery mode `NMI`.

When the processor receives a `NMI` from one of these sources, the processor handles it immediately by calling the `NMI` handler pointed to by interrupt vector which has number `2` (see table in the first part). We already filled the Interrupt Descriptor Table with the vector number, address of the `nmi` interrupt handler and `NMI_STACK` Interrupt Stack Table entry:

```
set_intr_gate_ist(X86_TRAP_NMI, &nmi, NMI_STACK);
```

in the `trap_init` function which defined in the arch/x86/kernel/traps.c source code file. In the previous parts we saw that entry points of the all interrupt handlers are defined with the:

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
...
...
...
END(\sym)
.endm
```

macro from the arch/x86/entry/entry_64.S assembly source code file. But the handler of the `Non-Maskable` interrupts is not defined with this macro. It has own entry point:

```
ENTRY(nmi)
...
...
...
END(nmi)
```

in the same arch/x86/entry/entry_64.S assembly file. Lets dive into it and will try to understand how `Non-Maskable` interrupt handler works. The `nmi` handlers starts from the call of the:

```
PARAVIRT_ADJUST_EXCEPTION_FRAME
```

macro but we will not dive into details about it in this part, because this macro related to the Paravirtualization stuff which we will see in another chapter. After this save the content of the `rdx` register on the stack:

```
pushq    %rdx
```

And allocated check that `cs` was not the kernel segment when an non-maskable interrupt occurs:

```
cmpl    $__KERNEL_CS, 16(%rsp)
jne    first_nmi
```

The `__KERNEL_CS` macro defined in the arch/x86/include/asm/segment.h and represented second descriptor in the Global Descriptor Table:

```
#define GDT_ENTRY_KERNEL_CS    2
#define __KERNEL_CS    (GDT_ENTRY_KERNEL_CS*8)
```

more about `GDT` you can read in the second part of the Linux kernel booting process chapter. If `cs` is not kernel segment, it means that it is not nested `NMI` and we jump on the `first_nmi` label. Let's consider this case. First of all we put address of the current stack pointer to the `rdx` and pushes `1` to the stack in the `first_nmi` label:

```
first_nmi:
    movq    (%rsp), %rdx
    pushq    $1
```

Why do we push `1` on the stack? As the comment says: `We allow breakpoints in NMIs`. On the x86_64, like other architectures, the CPU will not execute another `NMI` until the first `NMI` is complete. A `NMI` interrupt finished with the iret instruction like other interrupts and exceptions do it. If the `NMI` handler triggers either a page fault or breakpoint or another exception which are use `iret` instruction too. If this happens while in `NMI` context, the CPU will leave `NMI` context and a new `NMI` may come in. The `iret` used to return from those exceptions will re-enable `NMIs` and we will get nested non-maskable interrupts. The problem the `NMI` handler will not return to the state that it was, when the exception triggered, but instead it will return to a state that will allow new `NMIs` to preempt the running `NMI` handler. If another `NMI` comes in before the first NMI handler is complete, the new NMI will write all over the preempted `NMIs` stack. We can have nested `NMIs` where the next `NMI` is using the top of the stack of the previous `NMI`. It means that we cannot execute it because a nested non-maskable interrupt will corrupt stack of a previous non-maskable interrupt. That's why we have allocated space on the stack for temporary variable. We will check this variable that it was set when a previous `NMI` is executing and clear if it is not nested `NMI`. We push `1` here to the previously allocated space on the stack to denote that a `non-maskable` interrupt executed currently. Remember that when and `NMI` or another exception occurs we have the following stack frame:

```
+------------------------+
|         SS             |
|         RSP            |
|        RFLAGS          |
|         CS             |
|         RIP            |
+------------------------+
```

and also an error code if an exception has it. So, after all of these manipulations our stack frame will look like this:

```
+------------------------+
|         SS             |
|         RSP            |
|        RFLAGS          |
|         CS             |
|         RIP            |
|         RDX            |
|          1             |
+------------------------+
```

In the next step we allocate yet another `40` bytes on the stack:

```
subq    $(5*8), %rsp
```

and pushes the copy of the original stack frame after the allocated space:

```
.rept 5
pushq    11*8(%rsp)
.endr
```

with the .rept assembly directive. We need in the copy of the original stack frame. Generally we need in two copies of the interrupt stack. First is `copied` interrupts stack: `saved` stack frame and `copied` stack frame. Now we pushes original stack frame to the `saved` stack frame which locates after the just allocated `40` bytes (`copied` stack frame). This stack frame is used to fixup the `copied` stack frame that a nested NMI may change. The second - `copied` stack frame modified by any nested `NMIs` to let the first `NMI` know that we triggered a second `NMI` and we should repeat the first `NMI` handler. Ok, we have made first copy of the original stack frame, now time to make second copy:

```
addq    $(10*8), %rsp

.rept 5
pushq    -6*8(%rsp)
.endr
subq    $(5*8), %rsp
```

After all of these manipulations our stack frame will be like this:

```
+-------------------------+
| original SS             |
| original Return RSP     |
| original RFLAGS         |
| original CS             |
| original RIP            |
+-------------------------+
| temp storage for rdx    |
+-------------------------+
| NMI executing variable  |
+-------------------------+
| copied SS               |
| copied Return RSP       |
| copied RFLAGS           |
| copied CS               |
| copied RIP              |
+-------------------------+
| Saved SS                |
| Saved Return RSP        |
| Saved RFLAGS            |
| Saved CS                |
| Saved RIP               |
```

```
+------------------------+
```

After this we push dummy error code on the stack as we did it already in the previous exception handlers and allocate space for the general purpose registers on the stack:

```
pushq    $-1
ALLOC_PT_GPREGS_ON_STACK
```

We already saw implementation of the `ALLOC_PT_GREGS_ON_STACK` macro in the third part of the interrupts chapter. This macro defined in the arch/x86/entry/calling.h and yet another allocates `120` bytes on stack for the general purpose registers, from the `rdi` to the `r15` :

```
.macro ALLOC_PT_GPREGS_ON_STACK addskip=0
addq    $-(15*8+\addskip), %rsp
.endm
```

After space allocation for the general registers we can see call of the `paranoid_entry` :

```
call    paranoid_entry
```

We can remember from the previous parts this label. It pushes general purpose registers on the stack, reads `MSR_GS_BASE` Model Specific register and checks its value. If the value of the `MSR_GS_BASE` is negative, we came from the kernel mode and just return from the `paranoid_entry` , in other way it means that we came from the usermode and need to execute `swapgs` instruction which will change user `gs` with the kernel `gs` :

```
ENTRY(paranoid_entry)
    cld
    SAVE_C_REGS 8
    SAVE_EXTRA_REGS 8
    movl    $1, %ebx
    movl    $MSR_GS_BASE, %ecx
    rdmsr
    testl    %edx, %edx
    js    1f
    SWAPGS
    xorl    %ebx, %ebx
1:    ret
END(paranoid_entry)
```

Note that after the `swapgs` instruction we zeroed the `ebx` register. Next time we will check content of this register and if we executed `swapgs` than `ebx` must contain `0` and `1` in other way. In the next step we store value of the `cr2` control register to the `r12` register, because the `NMI` handler can cause `page fault` and corrupt the value of this control register:

```
movq    %cr2, %r12
```

Now time to call actual `NMI` handler. We push the address of the `pt_regs` to the `rdi` , error code to the `rsi` and call the `do_nmi` handler:

```
movq    %rsp, %rdi
movq    $-1, %rsi
call    do_nmi
```

We will back to the `do_nmi` little later in this part, but now let's look what occurs after the `do_nmi` will finish its execution. After the `do_nmi` handler will be finished we check the `cr2` register, because we can got page fault during `do_nmi` performed and if we got it we restore original `cr2`, in other way we jump on the label `1`. After this we test content of the `ebx` register (remember it must contain `0` if we have used `swapgs` instruction and `1` if we didn't use it) and execute `SWAPGS_UNSAFE_STACK` if it contains `1` or jump to the `nmi_restore` label. The `SWAPGS_UNSAFE_STACK` macro just expands to the `swapgs` instruction. In the `nmi_restore` label we restore general purpose registers, clear allocated space on the stack for this registers clear our temporary variable and exit from the interrupt handler with the `INTERRUPT_RETURN` macro:

```
	movq	%cr2, %rcx
	cmpq	%rcx, %r12
	je	1f
	movq	%r12, %cr2
1:
	testl	%ebx, %ebx
	jnz	nmi_restore
nmi_swapgs:
	SWAPGS_UNSAFE_STACK
nmi_restore:
	RESTORE_EXTRA_REGS
	RESTORE_C_REGS
	/* Pop the extra iret frame at once */
	REMOVE_PT_GPREGS_FROM_STACK 6*8
	/* Clear the NMI executing stack variable */
	movq	$0, 5*8(%rsp)
	INTERRUPT_RETURN
```

where `INTERRUPT_RETURN` is defined in the arch/x86/include/irqflags.h and just expands to the `iret` instruction. That's all.

Now let's consider case when another `NMI` interrupt occurred when previous `NMI` interrupt didn't finish its execution. You can remember from the beginning of this part that we've made a check that we came from userspace and jump on the `first_nmi` in this case:

```
cmpl	$__KERNEL_CS, 16(%rsp)
jne	first_nmi
```

Note that in this case it is first `NMI` every time, because if the first `NMI` catched page fault, breakpoint or another exception it will be executed in the kernel mode. If we didn't come from userspace, first of all we test our temporary variable:

```
cmpl	$1, -8(%rsp)
je	nested_nmi
```

and if it is set to `1` we jump to the `nested_nmi` label. If it is not `1`, we test the `IST` stack. In the case of nested `NMIs` we check that we are above the `repeat_nmi`. In this case we ignore it, in other way we check that we above than `end_repeat_nmi` and jump on the `nested_nmi_out` label.

Now let's look on the `do_nmi` exception handler. This function defined in the arch/x86/kernel/nmi.c source code file and takes two parameters:

- address of the `pt_regs`;
- error code.

as all exception handlers. The `do_nmi` starts from the call of the `nmi_nesting_preprocess` function and ends with the call of the `nmi_nesting_postprocess`. The `nmi_nesting_preprocess` function checks that we likely do not work with the debug stack and if we on the debug stack set the `update_debug_stack` per-cpu variable to `1` and call the `debug_stack_set_zero` function from the arch/x86/kernel/cpu/common.c. This function increases the `debug_stack_use_ctr` per-cpu variable and loads new `Interrupt Descriptor Table`:

```
static inline void nmi_nesting_preprocess(struct pt_regs *regs)
{
        if (unlikely(is_debug_stack(regs->sp))) {
                debug_stack_set_zero();
                this_cpu_write(update_debug_stack, 1);
        }
}
```

The `nmi_nesting_postprocess` function checks the `update_debug_stack` per-cpu variable which we set in the `nmi_nesting_preprocess` and resets debug stack or in another words it loads origin `Interrupt Descriptor Table`. After the call of the `nmi_nesting_preprocess` function, we can see the call of the `nmi_enter` in the `do_nmi`. The `nmi_enter` increases `lockdep_recursion` field of the interrupted process, update preempt counter and informs the RCU subsystem about `NMI`. There is also `nmi_exit` function that does the same stuff as `nmi_enter`, but vice-versa. After the `nmi_enter` we increase `__nmi_count` in the `irq_stat` structure and call the `default_do_nmi` function. First of all in the `default_do_nmi` we check the address of the previous nmi and update address of the last nmi to the actual:

```
if (regs->ip == __this_cpu_read(last_nmi_rip))
    b2b = true;
else
    __this_cpu_write(swallow_nmi, false);

__this_cpu_write(last_nmi_rip, regs->ip);
```

After this first of all we need to handle CPU-specific `NMIs`:

```
handled = nmi_handle(NMI_LOCAL, regs, b2b);
__this_cpu_add(nmi_stats.normal, handled);
```

And than non-specific `NMIs` depends on its reason:

```
reason = x86_platform.get_nmi_reason();
if (reason & NMI_REASON_MASK) {
    if (reason & NMI_REASON_SERR)
        pci_serr_error(reason, regs);
    else if (reason & NMI_REASON_IOCHK)
        io_check_error(reason, regs);

    __this_cpu_add(nmi_stats.external, 1);
    return;
}
```

That's all.

Range Exceeded Exception

The next exception is the `BOUND` range exceeded exception. The `BOUND` instruction determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). If the index is not within bounds, a `BOUND` range exceeded exception or `#BR` is occurred. The handler of the `#BR` exception is the `do_bounds` function that defined in the arch/x86/kernel/traps.c. The `do_bounds` handler starts with the call of the `exception_enter` function and ends with the call of the `exception_exit`:

```
prev_state = exception_enter();

if (notify_die(DIE_TRAP, "bounds", regs, error_code,
               X86_TRAP_BR, SIGSEGV) == NOTIFY_STOP)
```

```C
    goto exit;
...
...
...
exception_exit(prev_state);
return;
```

After we have got the state of the previous context, we add the exception to the `notify_die` chain and if it will return `NOTIFY_STOP` we return from the exception. More about notify chains and the `context tracking` functions you can read in the previous part. In the next step we enable interrupts if they were disabled with the `contidional_sti` function that checks `IF` flag and call the `local_irq_enable` depends on its value:

```C
conditional_sti(regs);

if (!user_mode(regs))
    die("bounds", regs, error_code);
```

and check that if we didn't came from user mode we send `SIGSEGV` signal with the `die` function. After this we check is MPX enabled or not, and if this feature is disabled we jump on the `exit_trap` label:

```C
if (!cpu_feature_enabled(X86_FEATURE_MPX)) {
    goto exit_trap;
}

where we execute `do_trap` function (more about it you can find in the previous part):

```C
exit_trap:
    do_trap(X86_TRAP_BR, SIGSEGV, "bounds", regs, error_code, NULL);
    exception_exit(prev_state);
```

If `MPX` feature is enabled we check the `BNDSTATUS` with the `get_xsave_field_ptr` function and if it is zero, it means that the `MPX` was not responsible for this exception:

```C
bndcsr = get_xsave_field_ptr(XSTATE_BNDCSR);
if (!bndcsr)
        goto exit_trap;
```

After all of this, there is still only one way when `MPX` is responsible for this exception. We will not dive into the details about Intel Memory Protection Extensions in this part, but will see it in another chapter.

Coprocessor exception and SIMD exception

The next two exceptions are x87 FPU Floating-Point Error exception or `#MF` and SIMD Floating-Point Exception or `#XF`. The first exception occurs when the `x87 FPU` has detected floating point error. For example divide by zero, numeric overflow and etc. The second exception occurs when the processor has detected SSE/SSE2/SSE3 `SIMD` floating-point exception. It can be the same as for the `x87 FPU`. The handlers for these exceptions are `do_coprocessor_error` and `do_simd_coprocessor_error` are defined in the arch/x86/kernel/traps.c and very similar on each other. They both make a call of the `math_error` function from the same source code file but pass different vector number. The `do_coprocessor_error` passes `X86_TRAP_MF` vector number to the `math_error`:

```C
dotraplinkage void do_coprocessor_error(struct pt_regs *regs, long error_code)
{
    enum ctx_state prev_state;

    prev_state = exception_enter();
```
```

```
    math_error(regs, error_code, X86_TRAP_MF);
    exception_exit(prev_state);
}
```

and `do_simd_coprocessor_error` passes `X86_TRAP_XF` to the `math_error` function:

```
dotraplinkage void
do_simd_coprocessor_error(struct pt_regs *regs, long error_code)
{
    enum ctx_state prev_state;

    prev_state = exception_enter();
    math_error(regs, error_code, X86_TRAP_XF);
    exception_exit(prev_state);
}
```

First of all the `math_error` function defines current interrupted task, address of its fpu, string which describes an exception, add it to the `notify_die` chain and return from the exception handler if it will return `NOTIFY_STOP` :

```
    struct task_struct *task = current;
    struct fpu *fpu = &task->thread.fpu;
    siginfo_t info;
    char *str = (trapnr == X86_TRAP_MF) ? "fpu exception" :
                        "simd exception";

    if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, SIGFPE) == NOTIFY_STOP)
        return;
```

After this we check that we are from the kernel mode and if yes we will try to fix an excetpion with the `fixup_exception` function. If we cannot we fill the task with the exception's error code and vector number and die:

```
if (!user_mode(regs)) {
    if (!fixup_exception(regs)) {
        task->thread.error_code = error_code;
        task->thread.trap_nr = trapnr;
        die(str, regs, error_code);
    }
    return;
}
```

If we came from the user mode, we save the `fpu` state, fill the task structure with the vector number of an exception and `siginfo_t` with the number of signal, `errno` , the address where exception occurred and signal code:

```
fpu__save(fpu);

task->thread.trap_nr    = trapnr;
task->thread.error_code = error_code;
info.si_signo        = SIGFPE;
info.si_errno        = 0;
info.si_addr        = (void __user *)uprobe_get_trap_addr(regs);
info.si_code = fpu__exception_code(fpu, trapnr);
```

After this we check the signal code and if it is non-zero we return:

```
if (!info.si_code)
    return;
```

Or send the `SIGFPE` signal in the end:

```
force_sig_info(SIGFPE, &info, task);
```

That's all.

Conclusion

It is the end of the sixth part of the Interrupts and Interrupt Handling chapter and we saw implementation of some exception handlers in this part, like `non-maskable` interrupt, SIMD and x87 FPU floating point exception. Finally we have finsihed with the `trap_init` function in this part and will go ahead in the next part. The next our point is the external interrupts and the `early_irq_init` function from the init/main.c.

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

- General Protection Fault
- opcode
- Non-Maskable
- BOUND instruction
- CPU socket
- Interrupt Descriptor Table
- Interrupt Stack Table
- Paravirtualization
- .rept
- SIMD
- Coprocessor
- x86_64
- iret
- page fault
- breakpoint
- Global Descriptor Table
- stack frame
- Model Specific regiser
- percpu
- RCU
- MPX
- x87 FPU
- Previous part

Interrupts and Interrupt Handling. Part 7.

Introduction to external interrupts

This is the seventh part of the Interrupts and Interrupt Handling in the Linux kernel chapter and in the previous part we have finished with the exceptions which are generated by the processor. In this part we will continue to dive to the interrupt handling and will start with the external hardware interrupt handling. As you can remember, in the previous part we have finsihed with the `trap_init` function from the arch/x86/kernel/trap.c and the next step is the call of the `early_irq_init` function from the init/main.c.

Interrupts are signal that are sent across IRQ or `Interrupt Request Line` by a hardware or software. External hardware interrupts allow devices like keyboard, mouse and etc, to indicate that it needs attention of the processor. Once the processor receives the `Interrupt Request`, it will temporary stop execution of the running program and invoke special routine which depends on an interrupt. We already know that this routine is called interrupt handler (or how we will call it `ISR` or `Interrupt Service Routine` from this part). The `ISR` or `Interrupt Handler Routine` can be found in Interrupt Vector table that is located at fixed address in the memory. After the interrupt is handled processor resumes the interrupted process. At the boot/initialization time, the Linux kernel identifies all devices in the machine, and appropriate interrupt handlers are loaded into the interrupt table. As we saw in the previous parts, most exceptions are handled simply by the sending a Unix signal to the interrupted process. That's why kernel is can handle an exception quickly. Unfortunatelly we can not use this approach for the external hardware interrupts, because often they arrive after (and sometimes long after) the process to which they are related has been suspended. So it would make no sense to send a Unix signal to the current process. External interrupt handling depends on the type of an interrupt:

- `I/O` interrupts;
- Timer interrupts;
- Interprocessor interrupts.

I will try to describe all types of interrupts in this book.

Generally, a handler of an `I/O` interrupt must be flexible enough to service several devices at the same time. For exmaple in the PCI bus architecture several devices may share the same `IRQ` line. In the simplest way the Linux kernel must do following thing when an `I/O` interrupt occured:

- Save the value of an `IRQ` and the register's contents on the kernel stack;
- Send an acknowledgment to the hardware controller which is servicing the `IRQ` line;
- Execute the interrupt service routine (next we will call it `ISR`) which is associated with the device;
- Restore registers and return from an interrupt;

Ok, we know a little theory and now let's start with the `early_irq_init` function. The implementation of the `early_irq_init` function is in the kernel/irq/irqdesc.c. This function make early initialziation of the `irq_desc` structure. The `irq_desc` structure is the foundation of interrupt management code in the Linux kernel. An array of this structure, which has the same name - `irq_desc`, keeps track of every interrupt request source in the Linux kernel. This structure defined in the include/linux/irqdesc.h and as you can note it depends on the `CONFIG_SPARSE_IRQ` kernel configuration option. This kernel configuration option enables support for sparse irqs. The `irq_desc` structure contains many different fiels:

- `irq_common_data` - per irq and chip data passed down to chip functions;
- `status_use_accessors` - contains status of the interrupt source which is can be combination of of the values from the `enum` from the include/linux/irq.h and different macros which are defined in the same source code file;
- `kstat_irqs` - irq stats per-cpu;
- `handle_irq` - highlevel irq-events handler;
- `action` - identifies the interrupt service routines to be invoked when the IRQ occurs;

- `irq_count` - counter of interrupt occurrences on the IRQ line;
- `depth` - `0` if the IRQ line is enabled and a positive value if it has been disabled at least once;
- `last_unhandled` - aging timer for unhandled count;
- `irqs_unhandled` - count of the unhandled interrupts;
- `lock` - a spin lock used to serialize the accesses to the `IRQ` descriptor;
- `pending_mask` - pending rebalanced interrupts;
- `owner` - an owner of interrupt descriptor. Interrupt descriptors can be allocated from modules. This field is need to proved refcount on the module which provides the interrupts;
- and etc.

Of course it is not all fields of the `irq_desc` structure, because it is too long to describe each field of this structure, but we will see it all soon. Now let's start to dive into the implementation of the `early_irq_init` function.

Early external interrupts initialization

Now, let's look on the implementation of the `early_irq_init` function. Note that implementation of the `early_irq_init` function depends on the `CONFIG_SPARSE_IRQ` kernel configuration option. Now we consider implementation of the `early_irq_init` function when the `CONFIG_SPARSE_IRQ` kernel configuration option is not set. This function starts from the declaration of the following variables: `irq` descriptors counter, loop counter, memory node and the `irq_desc` descriptor:

```
int __init early_irq_init(void)
{
        int count, i, node = first_online_node;
        struct irq_desc *desc;
        ...
        ...
        ...
}
```

The `node` is an online NUMA node which depends on the `MAX_NUMNODES` value which depends on the `CONFIG_NODES_SHIFT` kernel configuration parameter:

```
#define MAX_NUMNODES    (1 << NODES_SHIFT)
...
...
...
#ifdef CONFIG_NODES_SHIFT
    #define NODES_SHIFT     CONFIG_NODES_SHIFT
#else
    #define NODES_SHIFT     0
#endif
```

As I already wrote, implementation of the `first_online_node` macro depends on the `MAX_NUMNODES` value:

```
#if MAX_NUMNODES > 1
  #define first_online_node       first_node(node_states[N_ONLINE])
#else
  #define first_online_node       0
```

The `node_states` is the enum which defined in the include/linux/nodemask.h and represent the set of the states of a node. In our case we are searching an online node and it will be `0` if `MAX_NUMNODES` is one or zero. If the `MAX_NUMNODES` is greater than one, the `node_states[N_ONLINE]` will return `1` and the `first_node` macro will be expands to the call of the `__first_node` function which will return `minimal` or the first online node:

```
#define first_node(src) __first_node(&(src))

static inline int __first_node(const nodemask_t *srcp)
{
        return min_t(int, MAX_NUMNODES, find_first_bit(srcp->bits, MAX_NUMNODES));
}
```

More about this will be in the another chapter about the `NUMA` . The next step after the declaration of these local variables is the call of the:

```
init_irq_default_affinity();
```

function. The `init_irq_default_affinity` function defined in the same source code file and depends on the `CONFIG_SMP` kernel configuration option allocates a given cpumask structure (in our case it is the `irq_default_affinity`):

```
#if defined(CONFIG_SMP)
cpumask_var_t irq_default_affinity;

static void __init init_irq_default_affinity(void)
{
        alloc_cpumask_var(&irq_default_affinity, GFP_NOWAIT);
        cpumask_setall(irq_default_affinity);
}
#else
static void __init init_irq_default_affinity(void)
{
}
#endif
```

We know that when a hardware, such as disk controller or keyboard, needs attention from the processor, it throws an interrupt. The interrupt tells to the processor that something has happened and that the processor should interrupt current process and handle an incoming event. In order to prevent mutliple devices from sending the same interrupts, the IRQ system was established where each device in a computer system is assigned its own special IRQ so that its interrupts are unique. Linux kernel can assign certain `IRQs` to specific processors. This is known as `SMP IRQ affinity` , and it allows you control how your system will respond to various hardware events (that's why it has certain implementation only if the `CONFIG_SMP` kernel configuration option is set). After we allocated `irq_default_affinity` cpumask, we can see `printk` output:

```
printk(KERN_INFO "NR_IRQS:%d\n", NR_IRQS);
```

which prints `NR_IRQS` :

```
~$ dmesg | grep NR_IRQS
[    0.000000] NR_IRQS:4352
```

The `NR_IRQS` is the maximum number of the `irq` descriptors or in another words maximum number of interrupts. Its value depends on the state of the `COFNIG_X86_IO_APIC` kernel configuration option. If the `CONFIG_X86_IO_APIC` is not set and the Linux kernel uses an old PIC chip, the `NR_IRQS` is:

```
#define NR_IRQS_LEGACY                    16

#ifdef CONFIG_X86_IO_APIC
...
...
```

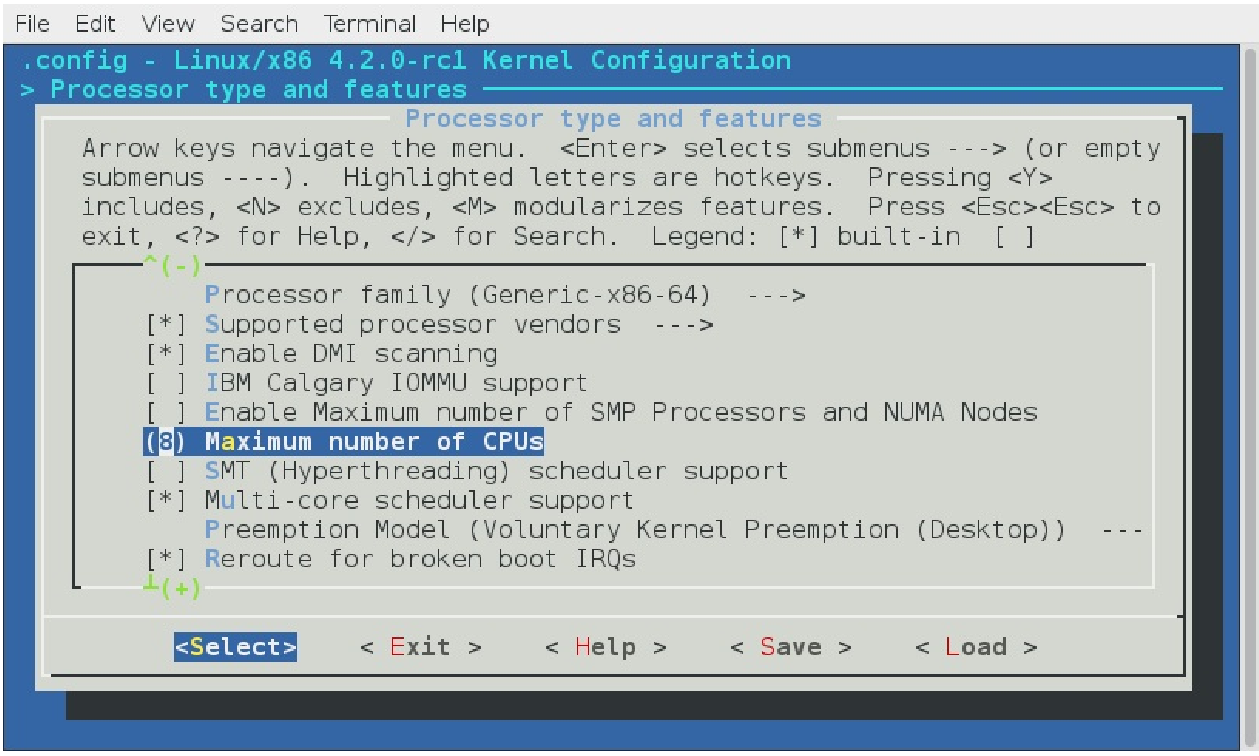
```
...
#else
# define NR_IRQS                        NR_IRQS_LEGACY
#endif
```

In other way, when the `CONFIG_X86_IO_APIC` kernel configuration option is set, the `NR_IRQS` depends on the amount of the processors and amount of the interrupt vectors:

```
#define CPU_VECTOR_LIMIT               (64 * NR_CPUS)
#define NR_VECTORS                     256
#define IO_APIC_VECTOR_LIMIT           ( 32 * MAX_IO_APICS )
#define MAX_IO_APICS                   128

# define NR_IRQS                                        \
        (CPU_VECTOR_LIMIT > IO_APIC_VECTOR_LIMIT ?      \
                (NR_VECTORS + CPU_VECTOR_LIMIT)  :      \
                (NR_VECTORS + IO_APIC_VECTOR_LIMIT))
...
...
...
```

We remember from the previous parts, that the amount of processors we can set during Linux kernel configuration process with the `CONFIG_NR_CPUS` configuration option:



In the first case (`CPU_VECTOR_LIMIT > IO_APIC_VECTOR_LIMIT`), the `NR_IRQS` will be `4352` , in the second case (`CPU_VECTOR_LIMIT < IO_APIC_VECTOR_LIMIT`), the `NR_IRQS` will be `768` . In my case the `NR_CPUS` is `8` as you can see in the my configuration, the `CPU_VECTOR_LIMIT` is `512` and the `IO_APIC_VECTOR_LIMIT` is `4096` . So `NR_IRQS` for my configuration is `4352` :

```
~$ dmesg | grep NR_IRQS
[    0.000000] NR_IRQS:4352
```

In the next step we assign array of the IRQ descriptors to the `irq_desc` variable which we defined in the start of the `early_irq_init` function and cacluate count of the `irq_desc` array with the `ARRAY_SIZE` macro:

```
desc = irq_desc;
count = ARRAY_SIZE(irq_desc);
```

The `irq_desc` array defined in the same source code file and looks like:

```
struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
        [0 ... NR_IRQS-1] = {
                .handle_irq     = handle_bad_irq,
                .depth          = 1,
                .lock           = __RAW_SPIN_LOCK_UNLOCKED(irq_desc->lock),
        }
};
```

The `irq_desc` is array of the `irq` descriptors. It has three already initialized fields:

- `handle_irq` - as I already wrote above, this field is the highlevel irq-event handler. In our case it initialized with the `handle_bad_irq` function that defined in the kernel/irq/handle.c source code file and handles spurious and unhandled irqs;
- `depth` - `0` if the IRQ line is enabled and a positive value if it has been disabled at least once;
- `lock` - A spin lock used to serialize the accesses to the `IRQ` descriptor.

As we calculated count of the interrupts and initialized our `irq_desc` array, we start to fill descriptors in the loop:

```
for (i = 0; i < count; i++) {
    desc[i].kstat_irqs = alloc_percpu(unsigned int);
    alloc_masks(&desc[i], GFP_KERNEL, node);
    raw_spin_lock_init(&desc[i].lock);
    lockdep_set_class(&desc[i].lock, &irq_desc_lock_class);
    desc_set_defaults(i, &desc[i], node, NULL);
}
```

We are going through the all interrupt descriptors and do the following things:

First of all we allocate percpu variable for the `irq` kernel statistic with the `alloc_percpu` macro. This macro allocates one instance of an object of the given type for every processor on the system. You can access kernel statistic from the userspace via `/proc/stat`:

```
~$ cat /proc/stat
cpu  207907 68 53904 5427850 14394 0 394 0 0 0
cpu0 25881 11 6684 679131 1351 0 18 0 0 0
cpu1 24791 16 5894 679994 2285 0 24 0 0 0
cpu2 26321 4 7154 678924 664 0 71 0 0 0
cpu3 26648 8 6931 678891 414 0 244 0 0 0
...
...
...
```

Where the sixth column is the servicing interrupts. After this we allocate cpumask for the given irq descriptor affinity and initialize the spinlock for the given interrupt descriptor. After this before the critical section, the lock will be aqcuired with a call of the `raw_spin_lock` and unlocked with the call of the `raw_spin_unlock`. In the next step we call the `lockdep_set_class` macro which set the Lock validator `irq_desc_lock_class` class for the lock of the given interrupt descriptor. More about `lockdep`, `spinlock` and other synchronization primitives will be described in the separate chapter.

In the end of the loop we call the `desc_set_defaults` function from the kernel/irq/irqdesc.c. This function takes four parameters:

- number of a irq;

- interrupt descriptor;
- online `NUMA` node;
- owner of interrupt descriptor. Interrupt descriptors can be allocated from modules. This field is need to proved refcount on the module which provides the interrupts;

and fills the rest of the `irq_desc` fields. The `desc_set_defaults` function fills interrupt number, `irq` chip, platform-specific per-chip private data for the chip methods, per-IRQ data for the `irq_chip` methods and MSI descriptor for the per `irq` and `irq` chip data:

```
desc->irq_data.irq = irq;
desc->irq_data.chip = &no_irq_chip;
desc->irq_data.chip_data = NULL;
desc->irq_data.handler_data = NULL;
desc->irq_data.msi_desc = NULL;
...
...
...
```

The `irq_data.chip` structure provides general `API` like the `irq_set_chip`, `irq_set_irq_type` and etc, for the irq controller drivers. You can find it in the kernel/irq/chip.c source code file.

After this we set the status of the accessor for the given descriptor and set disabled state of the interrupts:

```
...
...
...
irq_settings_clr_and_set(desc, ~0, _IRQ_DEFAULT_INIT_FLAGS);
irqd_set(&desc->irq_data, IRQD_IRQ_DISABLED);
...
...
...
```

In the next step we set the high level interrupt handlers to the `handle_bad_irq` which handles spurious and unhandled irqs (as the hardware stuff is not initialized yet, we set this handler), set `irq_desc.desc` to `1` which means that an `IRQ` is disabled, reset count of the unhandled interrupts and interrupts in general:

```
...
...
...
desc->handle_irq = handle_bad_irq;
desc->depth = 1;
desc->irq_count = 0;
desc->irqs_unhandled = 0;
desc->name = NULL;
desc->owner = owner;
...
...
...
```

After this we go through the all possible processor with the for_each_possible_cpu helper and set the `kstat_irqs` to zero for the given interrupt descriptor:

```
    for_each_possible_cpu(cpu)
        *per_cpu_ptr(desc->kstat_irqs, cpu) = 0;
```

and call the `desc_smp_init` function from the kernel/irq/irqdesc.c that initializes `NUMA` node of the given interrupt descriptor, sets default `SMP` affinity and clears the `pending_mask` of the given interrupt descriptor depends on the value of the

`CONFIG_GENERIC_PENDING_IRQ` kernel configuration option:

```
static void desc_smp_init(struct irq_desc *desc, int node)
{
        desc->irq_data.node = node;
        cpumask_copy(desc->irq_data.affinity, irq_default_affinity);
#ifdef CONFIG_GENERIC_PENDING_IRQ
        cpumask_clear(desc->pending_mask);
#endif
}
```

In the end of the `early_irq_init` function we return the return value of the `arch_early_irq_init` function:

```
return arch_early_irq_init();
```

This function defined in the kernel/apic/vector.c and contains only one call of the `arch_early_ioapic_init` function from the kernel/apic/io_apic.c. As we can understand from the `arch_early_ioapic_init` function's name, this function makes early initialization of the I/O APIC. First of all it make a check of the number of the legacy interrupts wit the call of the `nr_legacy_irqs` function. If we have no lagacy interrupts with the Intel 8259 programmable interrupt controller we set `io_apic_irqs` to the `0xffffffffffffffff`:

```
if (!nr_legacy_irqs())
    io_apic_irqs = ~0UL;
```

After this we are going through the all `I/O APICs` and allocate space for the registers with the call of the `alloc_ioapic_saved_registers`:

```
for_each_ioapic(i)
    alloc_ioapic_saved_registers(i);
```

And in the end of the `arch_early_ioapic_init` function we are going through the all legacy irqs (from `IRQ0` to `IRQ15`) in the loop and allocate space for the `irq_cfg` which represents configuration of an irq on the given `NUMA` node:

```
for (i = 0; i < nr_legacy_irqs(); i++) {
    cfg = alloc_irq_and_cfg_at(i, node);
    cfg->vector = IRQ0_VECTOR + i;
    cpumask_setall(cfg->domain);
}
```

That's all.

Sparse IRQs

We already saw in the beginning of this part that implementation of the `early_irq_init` function depends on the `CONFIG_SPARSE_IRQ` kernel configuration option. Previously we saw implementation of the `early_irq_init` function when the `CONFIG_SPARSE_IRQ` configuration option is not set, not let's look on the its implementation when this option is set. Implementation of this function very similar, but little differ. We can see the same definition of variables and call of the `init_irq_default_affinity` in the beginning of the `early_irq_init` function:

```
#ifdef CONFIG_SPARSE_IRQ
int __init early_irq_init(void)
```

```
{
    int i, initcnt, node = first_online_node;
    struct irq_desc *desc;

    init_irq_default_affinity();
    ...
    ...
    ...
}
#else
...
...
...
```

But after this we can see the following call:

```
initcnt = arch_probe_nr_irqs();
```

The `arch_probe_nr_irqs` function defined in the arch/x86/kernel/apic/vector.c and calculates count of the pre-allocated irqs and update `nr_irqs` with its number. But stop. Why there are pre-allocated irqs? There is alternative form of interrupts called - Message Signaled Interrupts available in the PCI. Instead of assigning a fixed number of the interrupt request, the device is allowed to record a message at a particular address of RAM, in fact, the display on the Local APIC. `MSI` permits a device to allocate `1`, `2`, `4`, `8`, `16` or `32` interrupts and `MSI-X` permits a device to allocate up to `2048` interrupts. Now we know that irqs can be pre-allocated. More about `MSI` will be in a next part, but now let's look on the `arch_probe_nr_irqs` function. We can see the check which assign amount of the interrupt vectors for the each processor in the system to the `nr_irqs` if it is greater and calculate the `nr` which represents number of `MSI` interrupts:

```
int nr_irqs = NR_IRQS;

if (nr_irqs > (NR_VECTORS * nr_cpu_ids))
    nr_irqs = NR_VECTORS * nr_cpu_ids;

nr = (gsi_top + nr_legacy_irqs()) + 8 * nr_cpu_ids;
```

Take a look on the `gsi_top` variable. Each `APIC` is identified with its own `ID` and with the offset where its `IRQ` starts. It is called `GSI` base or `Global System Interrupt` base. So the `gsi_top` represnters it. We get the `Global System Interrupt` base from the MultiProcessor Configuration Table table (you can remember that we have parsed this table in the sixth part of the Linux Kernel initialization process chapter).

After this we update the `nr` depends on the value of the `gsi_top`:

```
#if defined(CONFIG_PCI_MSI) || defined(CONFIG_HT_IRQ)
        if (gsi_top <= NR_IRQS_LEGACY)
                nr +=  8 * nr_cpu_ids;
        else
                nr += gsi_top * 16;
#endif
```

Update the `nr_irqs` if it less than `nr` and return the number of the legacy irqs:

```
if (nr < nr_irqs)
    nr_irqs = nr;

return nr_legacy_irqs();
}
```

The next after the `arch_probe_nr_irqs` is printing information about number of `IRQs`:

```
printk(KERN_INFO "NR_IRQS:%d nr_irqs:%d %d\n", NR_IRQS, nr_irqs, initcnt);
```

We can find it in the dmesg output:

```
$ dmesg | grep NR_IRQS
[    0.000000] NR_IRQS:4352 nr_irqs:488 16
```

After this we do some checks that `nr_irqs` and `initcnt` values is not greater than maximum allowable number of `irqs` :

```
if (WARN_ON(nr_irqs > IRQ_BITMAP_BITS))
    nr_irqs = IRQ_BITMAP_BITS;

if (WARN_ON(initcnt > IRQ_BITMAP_BITS))
    initcnt = IRQ_BITMAP_BITS;
```

where `IRQ_BITMAP_BITS` is equal to the `NR_IRQS` if the `CONFIG_SPARSE_IRQ` is not set and `NR_IRQS + 8196` in other way. In the next step we are going over all interrupt descript which need to be allocated in the loop and allocate space for the descriptor and insert to the `irq_desc_tree` radix tree:

```
for (i = 0; i < initcnt; i++) {
    desc = alloc_desc(i, node, NULL);
    set_bit(i, allocated_irqs);
    irq_insert_desc(i, desc);
}
```

In the end of the `early_irq_init` function we return the value of the call of the `arch_early_irq_init` function as we did it already in the previous variant when the `CONFIG_SPARSE_IRQ` option was not set:

```
return arch_early_irq_init();
```

That's all.

Conclusion

It is the end of the seventh part of the Interrupts and Interrupt Handling chapter and we started to dive into external hardware interrupts in this part. We saw early initialization of the `irq_desc` structure which represents description of an external interrupt and contains information about it like list of irq actions, information about interrupt handler, interrupts's owner, count of the unhandled interrupt and etc. In the next part we will continue to research external interrupts.

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

- IRQ
- numa
- Enum type

- cpumask
- percpu
- spinlock
- critical section
- Lock validator
- MSI
- I/O APIC
- Local APIC
- Intel 8259
- PIC
- MultiProcessor Configuration Table
- radix tree
- dmesg

Interrupts and Interrupt Handling. Part 8.

Non-early initialization of the IRQs

This is the eighth part of the Interrupts and Interrupt Handling in the Linux kernel chapter and in the previous part we started to dive into the external hardware interrupts. We looked on the implementation of the `early_irq_init` function from the kernel/irq/irqdesc.c source code file and saw the initialization of the `irq_desc` structure in this function. Remind that `irq_desc` structure (defined in the include/linux/irqdesc.h is the foundation of interrupt management code in the Linux kernel and represents an interrupt descriptor. In this part we will continue to dive into the initialization stuff which is related to the external hardware interrupts.

Right after the call of the `early_irq_init` function in the init/main.c we can see the call of the `init_IRQ` function. This function is architecture-specfic and defined in the arch/x86/kernel/irqinit.c. The `init_IRQ` function makes initialization of the `vector_irq` percpu variable that defined in the same arch/x86/kernel/irqinit.c source code file:

```
...
DEFINE_PER_CPU(vector_irq_t, vector_irq) = {
         [0 ... NR_VECTORS - 1] = -1,
};
...
```

and represents `percpu` array of the interrupt vector numbers. The `vector_irq_t` defined in the arch/x86/include/asm/hw_irq.h and expands to the:

```
typedef int vector_irq_t[NR_VECTORS];
```

where `NR_VECTORS` is count of the vector number and as you can remember from the first part of this chapter it is `256` for the x86_64:

```
#define NR_VECTORS                       256
```

So, in the start of the `init_IRQ` function we fill the `vecto_irq` percpu array with the vector number of the `legacy` interrupts:

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

    for (i = 0; i < nr_legacy_irqs(); i++)
        per_cpu(vector_irq, 0)[IRQ0_VECTOR + i] = i;
...
...
...
}
```

This `vector_irq` will be used during the first steps of an external hardware interrupt handling in the `do_IRQ` function from the arch/x86/kernel/irq.c:

```
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
    ...
    ...
```

```
    ...
    irq = __this_cpu_read(vector_irq[vector]);

    if (!handle_irq(irq, regs)) {
        ...
        ...
        ...
    }

    exiting_irq();
    ...
    ...
    return 1;
}
```

Why is `legacy` here? Actuall all interrupts handled by the modern IO-APIC controller. But these interrupts (from `0x30` to `0x3f`) by legacy interrupt-controllers like Programmable Interrupt Controller. If these interrupts are handled by the `I/O APIC` then this vector space will be freed and re-used. Let's look on this code closer. First of all the `nr_legacy_irqs` defined in the arch/x86/include/asm/i8259.h and just returns the `nr_legacy_irqs` field from the `legacy_pic` strucutre:

```
static inline int nr_legacy_irqs(void)
{
        return legacy_pic->nr_legacy_irqs;
}
```

This structure defined in the same header file and represents non-modern programmable interrupts controller:

```
struct legacy_pic {
        int nr_legacy_irqs;
        struct irq_chip *chip;
        void (*mask)(unsigned int irq);
        void (*unmask)(unsigned int irq);
        void (*mask_all)(void);
        void (*restore_mask)(void);
        void (*init)(int auto_eoi);
        int (*irq_pending)(unsigned int irq);
        void (*make_irq)(unsigned int irq);
};
```

Actuall default maximum number of the legacy interrupts representted by the `NR_IRQ_LEGACY` macro from the arch/x86/include/asm/irq_vectors.h:

```
#define NR_IRQS_LEGACY                    16
```

In the loop we are accessing the `vecto_irq` per-cpu array with the `per_cpu` macro by the `IRQ0_VECTOR + i` index and write the legacy vector number there. The `IRQ0_VECTOR` macro defined in the arch/x86/include/asm/irq_vectors.h header file and expands to the `0x30` :

```
#define FIRST_EXTERNAL_VECTOR           0x20

#define IRQ0_VECTOR                     ((FIRST_EXTERNAL_VECTOR + 16) & ~15)
```

Why is `0x30` here? You can remember from the first part of this chapter that first 32 vector numbers from `0` to `31` are reserved by the processor and used for the processing of architecture-defined exceptions and interrupts. Vector numbers from `0x30` to `0x3f` are reserved for the ISA. So, it means that we fill the `vector_irq` from the `IRQ0_VECTOR` which is equal to the `32` to the `IRQ0_VECTOR + 16` (before the `0x30`).

In the end of the `init_IRQ` functio we can see the call of the following function:

```
x86_init.irqs.intr_init();
```

from the arch/x86/kernel/x86_init.c source code file. If you have read chapter about the Linux kernel initialization process, you can remember the `x86_init` structure. This structure contains a couple of files which are points to the function related to the platform setup (`x86_64` in our case), for example `resources` - related with the memory resources, `mpparse` - related with the parsing of the MultiProcessor Configuration Table table and etc.). As we can see the `x86_init` also contains the `irqs` field which contains three following fields:

```
struct x86_init_ops x86_init __initdata
{
    ...
    ...
    ...
    .irqs = {
            .pre_vector_init        = init_ISA_irqs,
            .intr_init              = native_init_IRQ,
            .trap_init              = x86_init_noop,
    },
    ...
    ...
    ...
}
```

Now, we are interesting in the `native_init_IRQ` . As we can note, the name of the `native_init_IRQ` function contains the `native_` prefix which means that this function is architecture-specific. It defined in the arch/x86/kernel/irqinit.c and executes general initialization of the Local APIC and initialization of the ISA irqs. Let's look on the implementation of the `native_init_IRQ` function and will try to understand what occurs there. The `native_init_IRQ` function starts from the execution of the following function:

```
x86_init.irqs.pre_vector_init();
```

As we can see above, the `pre_vector_init` points to the `init_ISA_irqs` function that defined in the same source code file and as we can understand from the function's name, it makes initialization of the `ISA` related interrupts. The `init_ISA_irqs` function starts from the definition of the `chip` variable which has a `irq_chip` type:

```
void __init init_ISA_irqs(void)
{
    struct irq_chip *chip = legacy_pic->chip;
    ...
    ...
    ...
```

The `irq_chip` structure defined in the include/linux/irq.h header file and represents hardware interrupt chip descriptor. It contains:

- `name` - name of a device. Used in the `/proc/interrupts` :

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
  0:         16          0          0          0          0          0          0          0   IO-APIC    2-edge      ti
  1:          2          0          0          0          0          0          0          0   IO-APIC    1-edge      i8
  8:          1          0          0          0          0          0          0          0   IO-APIC    8-edge      rt
```

look on the last columnt;

- `(*irq_mask)(struct irq_data *data)` - mask an interrupt source;
- `(*irq_ack)(struct irq_data *data)` - start of a new interrupt;
- `(*irq_startup)(struct irq_data *data)` - start up the interrupt;
- `(*irq_shutdown)(struct irq_data *data)` - shutdown the interrupt
- and etc.

fields. Note that the `irq_data` structure represents set of the per irq chip data passed down to chip functions. It contains `mask` - precomputed bitmask for accessing the chip registers, `irq` - interrupt number, `hwirq` - hardware interrupt number, local to the interrupt domain chip low level interrupt hardware access and etc.

After this depends on the `CONFIG_X86_64` and `CONFIG_X86_LOCAL_APIC` kernel configuration option call the `init_bsp_APIC` function from the arch/x86/kernel/apic/apic.c:

```
#if defined(CONFIG_X86_64) || defined(CONFIG_X86_LOCAL_APIC)
    init_bsp_APIC();
#endif
```

This function makes initialization of the APIC of `bootstrap processor` (or processor which starts first). It starts from the check that we found SMP config (read more about it in the sixth part of the Linux kernel initialization process chapter) and the processor has `APIC` :

```
if (smp_found_config || !cpu_has_apic)
    return;
```

In other way we return from this function. In the next step we call the `clear_local_APIC` function from the same source code file that shutdowns the local `APIC` (more about it will be in the chapter about the `Advanced Programmable Interrupt Controller`) and enable `APIC` of the first processor by the setting `unsigned int value` to the `APIC_SPIV_APIC_ENABLED` :

```
value = apic_read(APIC_SPIV);
value &= ~APIC_VECTOR_MASK;
value |= APIC_SPIV_APIC_ENABLED;
```

and writing it with the help of the `apic_write` function:

```
apic_write(APIC_SPIV, value);
```

After we have enabled `APIC` for the bootstrap processor, we return to the `init_ISA_irqs` function and in the next step we initalize legacy `Programmable Interrupt Controller` and set the legacy chip and handler for the each legacy irq:

```
legacy_pic->init(0);

for (i = 0; i < nr_legacy_irqs(); i++)
    irq_set_chip_and_handler(i, chip, handle_level_irq);
```

Where can we find `init` function? The `legacy_pic` defined in the arch/x86/kernel/i8259.c and it is:

```
struct legacy_pic *legacy_pic = &default_legacy_pic;
```

Where the `default_legacy_pic` is:

```
struct legacy_pic default_legacy_pic = {
    ...
    ...
    ...
    .init = init_8259A,
    ...
    ...
    ...
}
```

The `init_8259A` function defined in the same source code file and executes initialization of the Intel 8259 `` `Programmable Interrupt Controller `` (more about it will be in the separate chapter abot `Programmable Interrupt Controllers` and `APIC`).

Now we can return to the `native_init_IRQ` function, after the `init_ISA_irqs` function finished its work. The next step is the call of the `apic_intr_init` function that allocates special interrupt gates which are used by the SMP architecture for the Inter-processor interrupt. The `alloc_intr_gate` macro from the arch/x86/include/asm/desc.h used for the interrupt descriptor allocation allocation:

```
#define alloc_intr_gate(n, addr)                        \
do {                                                    \
        alloc_system_vector(n);                         \
        set_intr_gate(n, addr);                         \
} while (0)
```

As we can see, first of all it expands to the call of the `alloc_system_vector` function that checks the given vector number in the `user_vectors` bitmap (read previous part about it) and if it is not set in the `user_vectors` bitmap we set it. After this we test that the `first_system_vector` is greater than given interrupt vector number and if it is greater we assign it:

```
if (!test_bit(vector, used_vectors)) {
    set_bit(vector, used_vectors);
    if (first_system_vector > vector)
        first_system_vector = vector;
} else {
    BUG();
}
```

We already saw the `set_bit` macro, now let's look on the `test_bit` and the `first_system_vector` . The first `test_bit` macro defined in the arch/x86/include/asm/bitops.h and looks like this:

```
#define test_bit(nr, addr)                      \
        (__builtin_constant_p((nr))             \
         ? constant_test_bit((nr), (addr))      \
         : variable_test_bit((nr), (addr)))
```

We can see the ternary operator here make a test with the gcc built-in function `__builtin_constant_p` tests that given vector number (`nr`) is known at compile time. If you're feeling misunderstanding of the `__builtin_constant_p` , we can make simple test:

```
#include <stdio.h>

#define PREDEFINED_VAL 1

int main() {
    int i = 5;
    printf("__builtin_constant_p(i) is %d\n", __builtin_constant_p(i));
```

```
    printf("__builtin_constant_p(PREDEFINED_VAL) is %d\n", __builtin_constant_p(PREDEFINED_VAL));
    printf("__builtin_constant_p(100) is %d\n", __builtin_constant_p(100));

    return 0;
}
```

and look on the result:

```
$ gcc test.c -o test
$ ./test
__builtin_constant_p(i) is 0
__builtin_constant_p(PREDEFINED_VAL) is 1
__builtin_constant_p(100) is 1
```

Now I think it must be clear for you. Let's get back to the `test_bit` macro. If the `__builtin_constant_p` will return non-zero, we call `constant_test_bit` function:

```
static inline int constant_test_bit(int nr, const void *addr)
{
    const u32 *p = (const u32 *)addr;

    return ((1UL << (nr & 31)) & (p[nr >> 5])) != 0;
}
```

and the `variable_test_bit` in other way:

```
static inline int variable_test_bit(int nr, const void *addr)
{
        u8 v;
        const u32 *p = (const u32 *)addr;

        asm("btl %2,%1; setc %0" : "=qm" (v) : "m" (*p), "Ir" (nr));
        return v;
}
```

What's the difference between two these functions and why do we need in two different functions for the same purpose? As you already can guess main purpose is optimization. If we will write simple example with these functions:

```
#define CONST 25

int main() {
    int nr = 24;
    variable_test_bit(nr, (int*)0x10000000);
    constant_test_bit(CONST, (int*)0x10000000)
    return 0;
}
```

and will look on the assembly output of our example we will see followig assembly code:

```
pushq    %rbp
movq    %rsp, %rbp

movl    $268435456, %esi
movl    $25, %edi
call    constant_test_bit
```

for the `constant_test_bit`, and:

```
pushq    %rbp
movq    %rsp, %rbp

subq    $16, %rsp
movl    $24, -4(%rbp)
movl    -4(%rbp), %eax
movl    $268435456, %esi
movl    %eax, %edi
call    variable_test_bit
```

for the `variable_test_bit`. These two code listings starts with the same part, first of all we save base of the current stack frame in the `%rbp` register. But after this code for both examples is different. In the first example we put `$268435456` (here the `$268435456` is our second parameter - `0x10000000`) to the `esi` and `$25` (our first parameter) to the `edi` register and call `constant_test_bit`. We put functuin parameters to the `esi` and `edi` registers because as we are learning Linux kernel for the `x86_64` architecture we use `System V AMD64 ABI` calling convention. All is pretty simple. When we are using predifined constant, the compiler can just substitute its value. Now let's look on the second part. As you can see here, the compiler can not substitute value from the `nr` variable. In this case compiler must calcuate its offset on the programm's stack frame. We substract `16` from the `rsp` register to allocate stack for the local variables data and put the `$24` (value of the `nr` variable) to the `rbp` with offset `-4`. Our stack frame will be like this:

```
         <- stack grows

              %[rbp]
                 |
+----------+ +---------+ +---------+ +--------+
|          | |         | | return  | |        |
|    nr    |-|         |-|         |-|  argc  |
|          | |         | | address | |        |
+----------+ +---------+ +---------+ +--------+
                 |
              %[rsp]
```

After this we put this value to the `eax`, so `eax` register now contains value of the `nr`. In the end we do the same that in the first example, we put the `$268435456` (the first parameter of the `variable_test_bit` function) and the value of the `eax` (value of `nr`) to the `edi` register (the second parameter of the `variable_test_bit function`).

The next step after the `apic_intr_init` function will finish its work is the setting interrup gates from the `FIRST_EXTERNAL_VECTOR` or `0x20` to the `0x256`:

```
i = FIRST_EXTERNAL_VECTOR;

#ifndef CONFIG_X86_LOCAL_APIC
#define first_system_vector NR_VECTORS
#endif

for_each_clear_bit_from(i, used_vectors, first_system_vector) {
    set_intr_gate(i, irq_entries_start + 8 * (i - FIRST_EXTERNAL_VECTOR));
}
```

But as we are using the `for_each_clear_bit_from` helper, we set only non-initialized interrupt gates. After this we use the same `for_each_clear_bit_from` helper to fill the non-filled interrupt gates in the interrupt table with the `spurious_interrupt`:

```
#ifdef CONFIG_X86_LOCAL_APIC
for_each_clear_bit_from(i, used_vectors, NR_VECTORS)
    set_intr_gate(i, spurious_interrupt);
#endif
```

Where the `spurious_interrupt` function represent interrupt handler fro the `spurious` interrupt. Here the `used_vectors` is the `unsigned long` that contains already initialized interrupt gates. We already filled first `32` interrupt vectors in the `trap_init` function from the arch/x86/kernel/setup.c source code file:

```
for (i = 0; i < FIRST_EXTERNAL_VECTOR; i++)
    set_bit(i, used_vectors);
```

You can remember how we did it in the sixth part of this chapter.

In the end of the `native_init_IRQ` function we can see the following check:

```
if (!acpi_ioapic && !of_ioapic && nr_legacy_irqs())
    setup_irq(2, &irq2);
```

First of all let's deal with the condition. The `acpi_ioapic` variable represents existence of I/O APIC. It defined in the arch/x86/kernel/acpi/boot.c. This variable set in the `acpi_set_irq_model_ioapic` function that called during the processing `Multiple APIC Description Table`. This occurs during initialization of the architecture-specific stuff in the arch/x86/kernel/setup.c (more about it we will know in the other chapter about APIC). Note that the value of the `acpi_ioapic` variable depends on the `CONFIG_ACPI` and `CONFIG_X86_LOCAL_APIC` Linux kernel configuration options. If these options did not set, this variable will be just zero:

```
#define acpi_ioapic 0
```

The second condition - `!of_ioapic && nr_legacy_irqs()` checks that we do not use Open Firmware `I/O APIC` and legacy interrupt controller. We already know about the `nr_legacy_irqs`. The second is `of_ioapic` variable defined in the arch/x86/kernel/devicetree.c and initialized in the `dtb_ioapic_setup` function that build information about `APICs` in the devicetree. Note that `of_ioapic` variable depends on the `CONFIG_OF` Linux kernel configuration opiotn. If this option is not set, the value of the `of_ioapic` will be zero too:

```
#ifdef CONFIG_OF
extern int of_ioapic;
...
...
...
#else
#define of_ioapic 0
...
...
...
#endif
```

If the condition will return non-zero vaule we call the:

```
setup_irq(2, &irq2);
```

function. First of all about the `irq2`. The `irq2` is the `irqaction` structure that defined in the arch/x86/kernel/irqinit.c source code file and represents `IRQ 2` line that is used to query devices connected cascade:

```
static struct irqaction irq2 = {
    .handler = no_action,
    .name = "cascade",
    .flags = IRQF_NO_THREAD,
};
```

Some time ago interrupt controller consisted of two chips and one was connected to second. The second chip that was connected to the first chip via this `IRQ 2` line. This chip serviced lines from `8` to `15` and after after this lines of the first chip. So, for example Intel 8259A has following lines:

- `IRQ 0` - system time;
- `IRQ 1` - keyboard;
- `IRQ 2` - used for devices which are cascade connected;
- `IRQ 8` - RTC;
- `IRQ 9` - reserved;
- `IRQ 10` - reserved;
- `IRQ 11` - reserved;
- `IRQ 12` - `ps/2` mouse;
- `IRQ 13` - coprocessor;
- `IRQ 14` - hard drive controller;
- `IRQ 1` - reserved;
- `IRQ 3` - `COM2` and `COM4` ;
- `IRQ 4` - `COM1` and `COM3` ;
- `IRQ 5` - `LPT2` ;
- `IRQ 6` - drive controller;
- `IRQ 7` - `LPT1` .

The `setup_irq` function defined in the kernel/irq/manage.c and takes two parameters:

- vector number of an interrupt;
- `irqaction` structure related with an interrupt.

This function initializes interrupt descriptor from the given vector number at the beginning:

```
struct irq_desc *desc = irq_to_desc(irq);
```

And call the `__setup_irq` function that setups given interrupt:

```
chip_bus_lock(desc);
retval = __setup_irq(irq, desc, act);
chip_bus_sync_unlock(desc);
return retval;
```

Note that the interrupt descriptor is locked during `__setup_irq` function will work. The `__setup_irq` function makes many different things: It creates a handler thread when a thread function is supplied and the interrupt does not nest into another interrupt thread, sets the flags of the chip, fills the `irqaction` structure and many many more.

All of the above it creates `/prov/vector_number` directory and fills it, but if you are using modern computer all values will be zero there:

```
$ cat /proc/irq/2/node
0

$cat /proc/irq/2/affinity_hint
00

cat /proc/irq/2/spurious
count 0
unhandled 0
last_unhandled 0 ms
```

because probably `APIC` handles interrupts on the our machine.

That's all.

Conclusion

It is the end of the eighth part of the Interrupts and Interrupt Handling chapter and we continued to dive into external hardware interrupts in this part. In the previous part we started to do it and saw early initialization of the `IRQs`. In this part we already saw non-early interrupts initialization in the `init_IRQ` function. We saw initialization of the `vector_irq` per-cpu array which is store vector numbers of the interrupts and will be used during interrupt handling and initialization of other stuff which is related to the external hardware interrupts.

In the next part we will continue to learn interrupts handling related stuff and will see initialization of the `softirqs`.

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

- IRQ
- percpu
- x86_64
- Intel 8259
- Programmable Interrupt Controller
- ISA
- MultiProcessor Configuration Table
- Local APIC
- I/O APIC
- SMP
- Inter-processor interrupt
- ternary operator
- gcc
- calling convention
- PDF. System V Application Binary Interface AMD64
- Call stack
- Open Firmware
- devicetree
- RTC
- Previous part

Interrupts and Interrupt Handling. Part 9.

Introduction to deferred interrupts (Softirq, Tasklets and Workqueues)

It is the ninth part of the linux-insides book and in the previous Previous part we saw implementation of the `init_IRQ` from that defined in the arch/x86/kernel/irqinit.c source code file. So, we will continue to dive into the initialization stuff which is related to the external hardware interrupts in this part.

After the `init_IRQ` function we can see the call of the `softirq_init` function in the init/main.c. This function defined in the kernel/softirq.c source code file and as we can understand from its name, this function makes initialization of the `softirq` or in other words initialization of the `deferred interrupts`. What is it deferreed intrrupt? We already saw a little bit about it in the ninth part of the chapter that describes initialization process of the Linux kernel. There are three types of `deffered interrupts` in the Linux kernel:

- `softirqs` ;
- `tasklets` ;
- `workqueues` ;

And we will see description of all of these types in this part. As I said, we saw only a little bit about this theme, so, now is time to dive deep into details about this theme.

Deferred interrupts

Interrupts may have different important characteristics and there are two among them:

- Handler of an interrupt must execute quickly;
- Sometime an interrupt handler must do a large amount of work.

As you can understand, it is almost impossible to make so that both characteristics were valid. Because of these, previously the handling of interrupts was splitted into two parts:

- Top half;
- Bottom half;

Once the Linux kernel was one of the ways the organization postprocessing, and which was called: `the bottom half` of the processor, but now it is already not actual. Now this term has remained as a common noun referring to all the different ways of organizing deffered processing of an interrupt. With the advent of parallelisms in the Linux kernel, all new schemes of implementation of the bottom half handlers are built on the performance of the processor specific kernel thread that called `ksoftirqd` (will be discussed below). The `softirq` mechanism represents handling of interrupts that are `almost` as important as the handling of the hardware interrupts. The deferred processing of an interrupt suggests that some of the actions for an interrupt may be postponed to a later execution when the system will be less loaded. As you can suggests, an interrupt handler can do large amount of work that is impermissible as it executes in the context where interrupts are disabled. That's why processing of an interrupt can be splitted on two different parts. In the first part, the main handler of an interrupt does only minimal and the most important job. After this it schedules the second part and finishes its work. When the system is less busy and context of the processor allows to handle interrupts, the second part starts its work and finishes to process remaing part of a deferred interrupt. That is main explanation of the deferred interrupt handling.

As I already wrote above, handling of deferred interrupts (or `softirq` in other words) and accordingly `tasklets` is performed by a set of the special kernel threads (one thread per processor). Each processor has its own thread that is called `ksoftirqd/n` where the `n` is the number of the processor. We can see it in the output of the `systemd-cgls` util:

```
$ systemd-cgls -k | grep ksoft
├─   3 [ksoftirqd/0]
├─  13 [ksoftirqd/1]
├─  18 [ksoftirqd/2]
├─  23 [ksoftirqd/3]
├─  28 [ksoftirqd/4]
├─  33 [ksoftirqd/5]
├─  38 [ksoftirqd/6]
├─  43 [ksoftirqd/7]
```

The `spawn_ksoftirqd` function starts this these threads. As we can see this function called as early initcall:

```
early_initcall(spawn_ksoftirqd);
```

Deferred interrupts are determined statically at compile-time of the Linux kernel and the `open_softirq` function takes care of `softirq` initialization. The `open_softirq` function defined in the kernel/softirq.c:

```
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;
}
```

and as we can see this function uses two parameters:

- the index of the `softirq_vec` array;
- a pointer to the softirq function to be executed;

First of all let's look on the `softirq_vec` array:

```
static struct softirq_action softirq_vec[NR_SOFTIRQS] __cacheline_aligned_in_smp;
```

it defined in the same source code file. As we can see, the `softirq_vec` array may contain `NR_SOFTIRQS` or `10` types of `softirqs` that has type `softirq_action`. First of all about its elements. In the current version of the Linux kernel there are ten softirq vectors defined; two for tasklet processing, two for networking, two for the block layer, two for timers, and one each for the scheduler and read-copy-update processing. All of these kinds are represented by the following enum:

```
enum
{
        HI_SOFTIRQ=0,
        TIMER_SOFTIRQ,
        NET_TX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        BLOCK_SOFTIRQ,
        BLOCK_IOPOLL_SOFTIRQ,
        TASKLET_SOFTIRQ,
        SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ,
        RCU_SOFTIRQ,
        NR_SOFTIRQS
};
```

All names of these kinds of softirqs are represented by the following array:

```
const char * const softirq_to_name[NR_SOFTIRQS] = {
        "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "BLOCK_IOPOLL",
        "TASKLET", "SCHED", "HRTIMER", "RCU"
};
```

Or we can see it in the output of the `/proc/softirqs`:

```
~$ cat /proc/softirqs
                    CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
          HI:          5          0          0          0          0          0          0          0
       TIMER:     332519     310498     289555     272913     282535     279467     282895     270979
      NET_TX:       2320          0          0          2          1          1          0          0
      NET_RX:     270221        225        338        281        311        262        430        265
       BLOCK:     134282         32         40         10         12          7          8          8
BLOCK_IOPOLL:          0          0          0          0          0          0          0          0
     TASKLET:     196835          2          3          0          0          0          0          0
       SCHED:     161852     146745     129539     126064     127998     128014     120243     117391
     HRTIMER:          0          0          0          0          0          0          0          0
         RCU:     337707     289397     251874     239796     254377     254898     267497     256624
```

As we can see the `softirq_vec` array has `softirq_action` types. This is the main data structure related to the `softirq` mechanism, so all `softirqs` represented by the `softirq_action` structure. The `softirq_action` structure consists a single field only: an action pointer to the softirq function:

```
struct softirq_action
{
         void    (*action)(struct softirq_action *);
};
```

So, after this we can understand that the `open_softirq` function fills the `softirq_vec` array with the given `softirq_action`. The registered deferred interrupt (with the call of the `open_softirq` function) for it to be queued for execution, it should be activated by the call of the `raise_softirq` function. This function takes only one parameter -- a softirq index `nr`. Let's look on its implementation:

```
void raise_softirq(unsigned int nr)
{
        unsigned long flags;

        local_irq_save(flags);
        raise_softirq_irqoff(nr);
        local_irq_restore(flags);
}
```

Here we can see the call of the `raise_softirq_irqoff` function between the `local_irq_save` and the `local_irq_restore` macros. The `local_irq_save` defined in the include/linux/irqflags.h header file and saves the state of the IF flag of the eflags register and disables interrupts on the local processor. The `local_irq_restore` macro defined in the same header file and does the opposite thing: restores the `interrupt flag` and enables interrupts. We disable interrupts here because a `softirq` interrupt runs in the interrupt context and that one softirq (and no others) will be run.

The `raise_softirq_irqoff` function marks the softirq as deffered by setting the bit corresponding to the given index `nr` in the `softirq` bit mask (`__softirq_pending`) of the local processor. It does it with the help of the:

```
__raise_softirq_irqoff(nr);
```

macro. After this, it checks the result of the `in_interrupt` that returns `irq_count` value. We already saw the `irq_count` in the first part of this chapter and it is used to check if a CPU is already on an interrupt stack or not. We just exit from the `raise_softirq_irqoff`, restore `IF` flang and enable interrupts on the local processor, if we are in the interrupt context, otherwise we call the `wakeup_softirqd`:

```
if (!in_interrupt())
    wakeup_softirqd();
```

Where the `wakeup_softirqd` function activates the `ksoftirqd` kernel thread of the local processor:

```
static void wakeup_softirqd(void)
{
    struct task_struct *tsk = __this_cpu_read(ksoftirqd);

    if (tsk && tsk->state != TASK_RUNNING)
        wake_up_process(tsk);
}
```

Each `ksoftirqd` kernel thread runs the `run_ksoftirqd` function that checks existence of deferred interrupts and calls the `__do_softirq` function depends on result. This function reads the `__softirq_pending` softirq bit mask of the local processor and executes the deferrable functions corresponding to every bit set. During execution of a deferred function, new pending `softirqs` might occur. The main problem here that execution of the userspace code can be delayed for a long time while the `__do_softirq` function will handle deferred interrupts. For this purpose, it has the limit of the time when it must be finsihed:

```
unsigned long end = jiffies + MAX_SOFTIRQ_TIME;
...
...
...
restart:
while ((softirq_bit = ffs(pending))) {
    ...
    h->action(h);
    ...
}
...
...
...
pending = local_softirq_pending();
if (pending) {
    if (time_before(jiffies, end) && !need_resched() &&
        --max_restart)
            goto restart;
}
...
```

Checks of the existence of the deferred interrupts performed periodically and there are some points where this check occurs. The main point where this situation occurs is the call of the `do_IRQ` function that defined in the arch/x86/kernel/irq.c and provides main possibilities for actual interrupt processing in the Linux kernel. When this function will finish to handle an interrupt, it calls the `exiting_irq` function from the arch/x86/include/asm/apic.h that expands to the call of the `irq_exit` function. The `irq_exit` checks deferred interrupts, current context and calls the `invoke_softirq` function:

```
if (!in_interrupt() && local_softirq_pending())
    invoke_softirq();
```

that executes the `__do_softirq` too. So what do we have in summary. Each `softirq` goes through the following stages: Registration of a `softirq` with the `open_softirq` function. Activation of a `softirq` by marking it as deferred with the `raise_softirq` function. After this, all marked `softirqs` will be runned in the next time the Linux kernel schedules a round of executions of deferrable functions. And execution of the deferred functions that have the same type.

As I already wrote, the `softirqs` are statically allocated and it is a problem for a kernel module that can be loaded. The second concept that built on top of `softirq` -- the `tasklets` solves this problem.

Tasklets

If you read the source code of the Linux kernel that is related to the `softirq`, you notice that it is used very rarely. The preferable way to implement deferrable functions are `tasklets`. As I already wrote above the `tasklets` are built on top of the `softirq` concept and generally on top of two `softirqs`:

- `TASKLET_SOFTIRQ`;
- `HI_SOFTIRQ`.

In short words, `tasklets` are `softirqs` that can be allocated and initialized at runtime and unlike `softirqs`, tasklets that have the same type cannot be run on multiple processors at a time. Ok, now we know a little bit about the `softirqs`, of course previous text does not cover all aspects about this, but now we can directly look on the code and to know more about the `softirqs` step by step on practice and to know about `tasklets`. Let's return back to the implementation of the `softirq_init` function that we talked about in the beginning of this part. This function is defined in the kernel/softirq.c source code file, let's look on its implementation:

```
void __init softirq_init(void)
{
        int cpu;

        for_each_possible_cpu(cpu) {
                per_cpu(tasklet_vec, cpu).tail =
                        &per_cpu(tasklet_vec, cpu).head;
                per_cpu(tasklet_hi_vec, cpu).tail =
                        &per_cpu(tasklet_hi_vec, cpu).head;
        }

        open_softirq(TASKLET_SOFTIRQ, tasklet_action);
        open_softirq(HI_SOFTIRQ, tasklet_hi_action);
}
```

We can see definition of the integer `cpu` variable at the beginning of the `softirq_init` function. Next we will use it as parameter for the `for_each_possible_cpu` macro that goes through the all possible processors in the system. If the `possible processor` is the new terminology for you, you can read more about it the CPU masks chapter. In short words, `possible cpus` is the set of processors that can be plugged in anytime during the life of that system boot. All `possible processors` stored in the `cpu_possible_bits` bitmap, you can find its definition in the kernel/cpu.c:

```
static DECLARE_BITMAP(cpu_possible_bits, CONFIG_NR_CPUS) __read_mostly;
...
...
...
const struct cpumask *const cpu_possible_mask = to_cpumask(cpu_possible_bits);
```

Ok, we defined the integer `cpu` variable and go through the all possible processors with the `for_each_possible_cpu` macro and makes initialization of the two following per-cpu variables:

- `tasklet_vec`;
- `tasklet_hi_vec`;

These two `per-cpu` variables defined in the same source code file as the `softirq_init` function and represent two `tasklet_head` structures:

```
static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec);
static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec);
```

Where `tasklet_head` structure represents a list of `Tasklets` and contains two fields, head and tail:

```
struct tasklet_head {
        struct tasklet_struct *head;
        struct tasklet_struct **tail;
};
```

The `tasklet_struct` structure is defined in the include/linux/interrupt.h and represents the `Tasklet`. Previously we did not see this word in this book. Let's try to understand what the `tasklet` is. Actually, the tasklet is one of mechanisms to handle deferred interrupt. Let's look on the implementation of the `tasklet_struct` structure:

```
struct tasklet_struct
{
        struct tasklet_struct *next;
        unsigned long state;
        atomic_t count;
        void (*func)(unsigned long);
        unsigned long data;
};
```

As we can see this structure contains five fields, they are:

- Next tasklet in the scheduling queue;
- State of the tasklet;
- Represent current state of the tasklet, active or not;
- Main callback of the tasklet;
- Parameter of the callback.

In our case, we set only for initialize only two arrays of tasklets in the `softirq_init` function: the `tasklet_vec` and the `tasklet_hi_vec`. Tasklets and high-priority tasklets are stored in the `tasklet_vec` and `tasklet_hi_vec` arrays, respectively. So, we have initialized these arrays and now we can see two calls of the `open_softirq` function that is defined in the kernel/softirq.c source code file:

```
open_softirq(TASKLET_SOFTIRQ, tasklet_action);
open_softirq(HI_SOFTIRQ, tasklet_hi_action);
```

at the end of the `softirq_init` function. The main purpose of the `open_softirq` function is the initalization of `softirq`. Let's look on the implementation of the `open_softirq` function.

, in our case they are: `tasklet_action` and the `tasklet_hi_action` or the `softirq` function associated with the `HI_SOFTIRQ` softirq is named `tasklet_hi_action` and `softirq` function associated with the `TASKLET_SOFTIRQ` is named `tasklet_action`. The Linux kernel provides API for the manipulating of `tasklets`. First of all it is the `tasklet_init` function that takes `tasklet_struct`, function and parameter for it and initializes the given `tasklet_struct` with the given data:

```
void tasklet_init(struct tasklet_struct *t,
                  void (*func)(unsigned long), unsigned long data)
{
    t->next = NULL;
    t->state = 0;
    atomic_set(&t->count, 0);
    t->func = func;
    t->data = data;
```

```
}
```

There are additional methods to initialize a tasklet statically with the two following macros:

```
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
```

The Linux kernel provides three following functions to mark a tasklet as ready to run:

```
void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule_first(struct tasklet_struct *t);
```

The first function schedules a tasklet with the normal priority, the second with the high priority and the third out of turn. Implementation of the all of these three functions is similar, so we will consider only the first -- `tasklet_schedule`. Let's look on its implementation:

```
static inline void tasklet_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
        __tasklet_schedule(t);
}

void __tasklet_schedule(struct tasklet_struct *t)
{
        unsigned long flags;

        local_irq_save(flags);
        t->next = NULL;
        *__this_cpu_read(tasklet_vec.tail) = t;
        __this_cpu_write(tasklet_vec.tail, &(t->next));
        raise_softirq_irqoff(TASKLET_SOFTIRQ);
        local_irq_restore(flags);
}
```

As we can see it checks and sets the state of the given tasklet to the `TASKLET_STATE_SCHED` and executes the `__tasklet_schedule` with the given tasklet. The `__tasklet_schedule` looks very similar to the `raise_softirq` function that we saw above. It saves the `interrupt flag` and disables interrupts at the beginning. After this, it updates `tasklet_vec` with the new tasklet and calls the `raise_softirq_irqoff` function that we saw above. When the Linux kernel scheduler will decide to run deferred functions, the `tasklet_action` function will be called for deferred functions which are associated with the `TASKLET_SOFTIRQ` and `tasklet_hi_action` for deferred functions which are associated with the `HI_SOFTIRQ`. These functions are very similar and there is only one difference between them -- `tasklet_action` uses `tasklet_vec` and `tasklet_hi_action` uses `tasklet_hi_vec`.

Let's look on the implementation of the `tasklet_action` function:

```
static void tasklet_action(struct softirq_action *a)
{
    local_irq_disable();
    list = __this_cpu_read(tasklet_vec.head);
    __this_cpu_write(tasklet_vec.head, NULL);
    __this_cpu_write(tasklet_vec.tail, this_cpu_ptr(&tasklet_vec.head));
    local_irq_enable();

    while (list) {
        if (tasklet_trylock(t)) {
            t->func(t->data);
            tasklet_unlock(t);
        }
```

```
        ...
        ...
        ...
    }
}
```

In the beginning of the `tasketl_action` function, we disable interrupts for the local processor with the help of the `local_irq_disable` macro (you can read about this macro in the second part of this chapter). In the next step, we take a head of the list that contains tasklets with normal priority and set this per-cpu list to `NULL` because all tasklets must be executed in a generaly way. After this we enable interrupts for the local processor and go through the list of taklets in the loop. In every iteration of the loop we call the `tasklet_trylock` function for the given tasklet that updates state of the given tasklet on `TASKLET_STATE_RUN` :

```
static inline int tasklet_trylock(struct tasklet_struct *t)
{
    return !test_and_set_bit(TASKLET_STATE_RUN, &(t)->state);
}
```

If this operation was successful we execute tasklet's action (it was set in the `tasklet_init`) and call the `tasklet_unlock` function that clears tasklet's `TASKLET_STATE_RUN` state.

In general, that's all about `tasklets` concept. Of course this does not cover full `tasklets` , but I think that it is a good point from where you can continue to learn this concept.

The `tasklets` are widely used concept in the Linux kernel, but as I wrote in the beginning of this part there is third mechanism for deferred functions -- `workqueue` . In the next paragraph we will see what it is.

Workqueues

The `workqueue` is another concept for handling deferred functions. It is similar to `tasklets` with some differences. Workqueue functions run in the context of a kernel process, but `tasklet` functions run in the software interrupt context. This means that `workqueue` functions must not be atomic as `tasklet` functions. Tasklets always run on the processor from which they were originally submitted. Workqueues work in the same way, but only by default. The `workqueue` concept represented by the:

```
struct worker_pool {
    spinlock_t              lock;
    int                     cpu;
    int                     node;
    int                     id;
    unsigned int            flags;

    struct list_head        worklist;
    int                     nr_workers;
...
...
...
```

structure that is defined in the kernel/workqueue.c source code file in the Linux kernel. I will not write the source code of this structure here, because it has quite a lot of fields, but we will consider some of those fields.

In its most basic form, the work queue subsystem is an interface for creating kernel threads to handle work that is queued from elsewhere. All of these kernel threads are called -- `worker threads` . The work queue are maintained by the `work_struct` that defined in the include/linux/workqueue.h. Let's look on this structure:

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
#ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
#endif
};
```

Here are two things that we are interested: `func` -- the function that will be scheduled by the `workqueue` and the `data` - parameter of this function. The Linux kernel provides special per-cpu threads that are called `kworker` :

```
systemd-cgls -k | grep kworker
├─    5 [kworker/0:0H]
├─   15 [kworker/1:0H]
├─   20 [kworker/2:0H]
├─   25 [kworker/3:0H]
├─   30 [kworker/4:0H]
...
...
...
```

This process can be used to schedule the deferred functions of the workqueues (as `ksoftirqd` for `softirqs`). Besides this we can create new separate worker thread for a `workqueue` . The Linux kernel provides following macros for the creation of workqueue:

```
#define DECLARE_WORK(n, f) \
    struct work_struct n = __WORK_INITIALIZER(n, f)
```

for static creation. It takes two parameters: name of the workqueue and the workqueue function. For creation of workqueue in runtime, we can use the:

```
#define INIT_WORK(_work, _func)       \
    __INIT_WORK((_work), (_func), 0)

#define __INIT_WORK(_work, _func, _onstack)                     \
    do {                                                        \
            __init_work((_work), _onstack);                     \
            (_work)->data = (atomic_long_t) WORK_DATA_INIT();   \
            INIT_LIST_HEAD(&(_work)->entry);                    \
             (_work)->func = (_func);                           \
    } while (0)
```

macro that takes `work_struct` structure that has to be created and the function to be scheduled in this workqueue. After a `work` was created with the one of these macros, we need to put it to the `workqueue` . We can do it with the help of the `queue_work` or the `queue_delayed_work` functions:

```
static inline bool queue_work(struct workqueue_struct *wq,
                              struct work_struct *work)
{
    return queue_work_on(WORK_CPU_UNBOUND, wq, work);
}
```

The `queue_work` function just calls the `queue_work_on` function that queue work on specific processor. Note that in our case we pass the `WORK_STRUCT_PENDING_BIT` to the `queue_work_on` function. It is a part of the `enum` that is defined in the include/linux/workqueue.h and represents workqueue which are not bound to any specific processor. The `queue_work_on` function tests and set the `WORK_STRUCT_PENDING_BIT` bit of the given `work` and executes the `__queue_work` function with the `workqueue` for the given processor and given `work` :

```
__queue_work(cpu, wq, work);
```

The `__queue_work` function gets the `work pool` . Yes, the `work pool` not `workqueue` . Actually, all `works` are not placed in the `workqueue` , but to the `work pool` that is represented by the `worker_pool` structure in the Linux kernel. As you can see above, the `workqueue_struct` structure has the `pwqs` field which is list of `worker_pools` . When we create a `workqueue` , it stands out for each processor the `pool_workqueue` . Each `pool_workqueue` associated with `worker_pool` , which is allocated on the same processor and corresponds to the type of priority queue. Through them `workqueue` interacts with `worker_pool` . So in the `__queue_work` function we set the cpu to the current processor with the `raw_smp_processor_id` (you can find information about this marco in the fouth part of the Linux kernel initialization process chapter), getting the `pool_workqueue` for the given `workqueue_struct` and insert the given `work` to the given `workqueue` :

```
static void __queue_work(int cpu, struct workqueue_struct *wq,
                         struct work_struct *work)
{
...
...
...
if (req_cpu == WORK_CPU_UNBOUND)
    cpu = raw_smp_processor_id();

if (!(wq->flags & WQ_UNBOUND))
    pwq = per_cpu_ptr(wq->cpu_pwqs, cpu);
else
    pwq = unbound_pwq_by_node(wq, cpu_to_node(cpu));
...
...
...
insert_work(pwq, work, worklist, work_flags);
```

As we can create `works` and `workqueue` , we need to know when they are executed. As I already wrote, all `works` are executed by the kernel thread. When this kernel thread is scheduled, it starts to execute `works` from the given `workqueue` . Each worker thread executes a loop inside the `worker_thread` function. This thread makes many different things and part of these things are similar to what we saw before in this part. As it starts executing, it removes all `work_struct` or `works` from its `workqueue` .

That's all.

Conclusion

It is the end of the ninth part of the Interrupts and Interrupt Handling chapter and we continued to dive into external hardware interrupts in this part. In the previous part we saw initialization of the `IRQs` and main `irq_desc` structure. In this part we saw three concepts: the `softirq` , `tasklet` and `workqueue` that are used for the deferred functions.

The next part will be last part of the `Interrupts and Interrupt Handling` chapter and we will look on the real hardware driver and will try to learn how it works with the interrupts subsystem.

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

Links

- initcall
- IF
- eflags
- CPU masks
- per-cpu
- Workqueue
- 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 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 a clear understanding of these 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 the 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 with 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 allocating 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 describe the type of the memory block. It can be: reserved, memory and physical memory if the `CONFIG_HAVE_MEMBLOCK_PHYS_MAP` configuration option is enabled. Now we see yet another data structure - `memblock_type`. Let's look at 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 the number of memory regions which are inside the current memory block, the size of all memory regions, the 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 a memory region. Its definition is:

```
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 the `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 at the top of the source code file and we will see the 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 are interested 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 above, all implementation of the `memblock` presented in the mm/memblock.c. To understand how `memblock` works and is implemented, let's look at its 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 do 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. The `memblock_add_range` function adds new memory region to the memory block. It starts by checking the size of the given region and if it is zero it just returns. After this, `memblock_add_range` checks for 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 the implementation of this in the First touch of the linux kernel memory manager framework). If `memblock_type` is not 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 is 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 `ULLONG_MAX - base` .

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

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

We are going through all the already stored memory regions and checking for overlap with the new region:

```
    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 the 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. Then we set `insert` to `true` and go to the `repeat` label. In the second step, starting from the `repeat` label we go through the same loop and insert the 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 implementation that we saw when we insert new region to the empty `memblock_type` (see above). This function gets 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 execution, `memblock_add_range` calls `memblock_merge_regions` which merges neighboring compatible regions in the second step.

In the second case the 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 the higher portion, because the lower portion is already in the overlapped memory region), then the 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 to 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 this is not the full API. Memblock provides an 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 an API for getting information about allocated memory regions in the `memblcok`. It is split in 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 functions is easy. Let's look at `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. Otherwise we write the physical address of the reserved memory regions array to the given address and return aligned size of the allocated array. 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.reserved`.

Memblock debugging

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

```
#define memblock_dbg(fmt, ...) \
```

```
        if (memblock_debug) printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
```

For example you can see a 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 will 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 on twitter 0xAX, drop me an email or just create an 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 a 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 are a set of special compile-time addresses whose corresponding physical address do not have to be a linear address minus `__START_KERNEL_map`. Each fix-mapped address maps one page frame and the kernel uses them as pointers that never change their address. That is the main point of these addresses. As the 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 represented by an 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 represented by an 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 might be a 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 checks that the index given for the `fixed_addresses` enum is not greater or equal than `__end_of_fixed_addresses` with the `BUILD_BUG_ON` macro and then 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 an 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 an 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 the 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 me explain how it works. As I already wrote we will clear the first 12 bits in the given address with `x & PAGE_MASK` . As we subtract this from the `FIXADDR_TOP` , we will get the last 12 bits of the `FIXADDR_TOP` which are present. We know that the first 12 bits of the virtual address represent the 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 the first 12 offset bits. `Fix-mapped` addresses are used in different places in 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 is implemented in the kernel and how it is releated to 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 is controlled by reading/writing from/to its registers. For example a driver can turn off/on a device by writing to its registers or get the state of a device by reading from its registers. Besides registers, many devices have buffers where a 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. Actually `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 in the kernel. For example let's look at drivers/char/rtc.c. This source code file provides Real Time Clock interface in the linux kernel. As every kernel module, `rtc` module contains `module_init` definition:

```
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 way 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 provided by a device to CPU through a bus. All memory-mapped I/O addresses are not used by the kernel directly. There is a special `ioremap` function which allows us to covert the 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. The `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 a little about `ioremap`, we saw it in the fifth part about linux kernel initialization. If you have read this part, you can remember the call of the `early_ioremap_init` function from the arch/x86/mm/ioremap.c. Initialization of the `ioremap` is split inn two parts: there is the early part which we can use before the normal `ioremap` is available and the 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 on twitter 0xAX, drop me an email or just create an 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 a 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

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

The 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 is implemented.

Take a look at the `DECLARE_PER_CPU` definition. We see that it takes 2 parameters: `type` and `name`, so we can use it to create per-cpu variable, for example like this:

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

We pass the type and the name of our variable. `DEFI_PER_CPU` calls `DEFINE_PER_CPU_SECTION` macro and passes the same two paramaters and empty string to it. Let's look at 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 are expanded 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 the kernel initilizes it calls `setup_per_cpu_areas` function which loads `.data..percpu` section multiply times, one section per CPU.

Let's look on the per-CPU areas initialization process. It start in the init/main.c from the call of the `setup_per_cpu_areas` function which defined in the arch/x86/kernel/setup_percpu.c.

```
pr_info("NR_CPUS:%d nr_cpumask_bits:%d nr_cpu_ids:%d nr_node_ids:%d\n",
        NR_CPUS, nr_cpumask_bits, nr_cpu_ids, nr_node_ids);
```

The `setup_per_cpu_areas` starts from the output information about the Maximum number of CPUs set during kernel configuration with `CONFIG_NR_CPUS` configuration option, actual number of CPUs, `nr_cpumask_bits` is the same that `NR_CPUS` bit for the new `cpumask` operators and number of `NUMA` nodes.

We can see this output in the dmesg:

```
$ dmesg | grep percpu
[    0.000000] setup_percpu: NR_CPUS:8 nr_cpumask_bits:8 nr_cpu_ids:8 nr_node_ids:1
```

In the next step we check `percpu` first chunk allocator. All percpu areas are allocated in chunks. First chunk is used for the static percpu variables. Linux kernel has `percpu_alloc` command line parameters which provides type of the first chunk allocator. We can read about it in the kernel documentation:

```
percpu_alloc=    Select which percpu first chunk allocator to use.
        Currently supported values are "embed" and "page".
        Archs may support subset or none of the    selections.
        See comments in mm/percpu.c for details on each
        allocator.  This parameter is primarily    for debugging
        and performance comparison.
```

The mm/percpu.c contains handler of this command line option:

```
early_param("percpu_alloc", percpu_alloc_setup);
```

Where `percpu_alloc_setup` function sets the `pcpu_chosen_fc` variable depends on the `percpu_alloc` parameter value. By default first chunk allocator is `auto` :

```
enum pcpu_fc pcpu_chosen_fc __initdata = PCPU_FC_AUTO;
```

If `percpu_alooc` parameter not given to the kernel command line, the `embed` allocator will be used wich as you can understand embed the first percpu chunk into bootmem with the memblock. The last allocator is first chunk `page` allocator which maps first chunk with `PAGE_SIZE` pages.

As I wrote about first of all we make a check of the first chunk allocator type in the `setup_per_cpu_areas` . First of all we check that first chunk allocator is not page:

```
if (pcpu_chosen_fc != PCPU_FC_PAGE) {
    ...
    ...
    ...
}
```

If it is not `PCPU_FC_PAGE` , we will use `embed` allocator and allocate space for the first chunk with the `pcpu_embed_first_chunk` function:

```
rc = pcpu_embed_first_chunk(PERCPU_FIRST_CHUNK_RESERVE,
                        dyn_size, atom_size,
                        pcpu_cpu_distance,
                        pcpu_fc_alloc, pcpu_fc_free);
```

As I wrote above, the `pcpu_embed_first_chunk` function embeds the first percpu chunk into bootmem. As you can see we pass a couple of parameters to the `pcup_embed_first_chunk`, they are

* `PERCPU_FIRST_CHUNK_RESERVE` - the size of the reserved space for the static `percpu` variables;
* `dyn_size` - minimum free size for dynamic allocation in byte;
* `atom_size` - all allocations are whole multiples of this and aligned to this parameter;
* `pcpu_cpu_distance` - callback to determine distance between cpus;
* `pcpu_fc_alloc` - function to allocate `percpu` page;
* `pcpu_fc_free` - function to release `percpu` page.

All of this parameters we calculat before the call of the `pcpu_embed_first_chunk`:

```
const size_t dyn_size = PERCPU_MODULE_RESERVE + PERCPU_DYNAMIC_RESERVE - PERCPU_FIRST_CHUNK_RESERVE;
size_t atom_size;
#ifdef CONFIG_X86_64
        atom_size = PMD_SIZE;
#else
        atom_size = PAGE_SIZE;
#endif
```

If first chunk allocator is `PCPU_FC_PAGE`, we will use the `pcpu_page_first_chunk` instead of the `pcpu_embed_first_chunk`. After that `percpu` areas up, we setup `percpu` offset and its segment for the every CPU with the `setup_percpu_segment` function (only for `x86` systems) and move some early data from the arrays to the `percpu` variables (`x86_cpu_to_apicid`, `irq_stack_ptr` and etc...). After the kernel finished the initialization process, we have loaded N `.data..percpu` sections, where N is the number of CPU, and section used by bootstrap processor will contain uninitialized variable created with `DEFINE_PER_CPU` macro.

The kernel provides API for per-cpu variables manipulating:

* get_cpu_var(var)
* put_cpu_var(var)

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

- The 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
- Radix tree

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.

Data Structures in the Linux Kernel

Radix tree

As you already know linux kernel provides many different libraries and functions which implement different data structures and algorithm. In this part we will consider one of these data structures - Radix tree. There are two files which are related to `radix tree` implementation and API in the linux kernel:

- include/linux/radix-tree.h
- lib/radix-tree.c

Lets talk about what is `radix tree`. Radix tree is a `compressed trie` where trie is a data structure which implements interface of an associative array and allows to store values as `key-value`. The keys are usually strings, but any other data type can be used as well. Trie is different from any `n-tree` in its nodes. Nodes of a trie do not store keys, instead, a node of a trie stores single character labels. The key which is related to a given node is derived by traversing from the root of the tree to this node. For example:

```
               +-----------+
               |           |
               |    " "    |
               |           |
        +------+-----------+------+
        |                         |
        |                         |
   +----v------+            +-----v-----+
   |           |            |           |
   |    g      |            |     c     |
   |           |            |           |
   +-----------+            +-----------+
        |                         |
        |                         |
   +----v------+            +-----v-----+
   |           |            |           |
   |    o      |            |     a     |
   |           |            |           |
   +-----------+            +-----------+
                                  |
                                  |
                            +-----v-----+
                            |           |
                            |     t     |
                            |           |
                            +-----------+
```

So in this example, we can see the `trie` with keys, `go` and `cat`. The compressed trie or `radix tree` differs from `trie`, such that all intermediates nodes which have only one child are removed.

Radix tree in linux kernel is the datastructure which maps values to the integer key. It is represented by the following structures from the file include/linux/radix-tree.h:

```
struct radix_tree_root {
         unsigned int            height;
         gfp_t                   gfp_mask;
         struct radix_tree_node  __rcu *rnode;
};
```

This structure presents the root of a radix tree and contains three fields:

* `height` - height of the tree;
* `gfp_mask` - tells how memory allocations are to be performed;
* `rnode` - pointer to the child node.

The first structure we will discuss is `gfp_mask` :

Low-level kernel memory allocation functions take a set of flags as - `gfp_mask` , which describes how that allocation is to be performed. These `GFP_` flags which control the allocation process can have following values, (`GF_NOIO` flag) be sleep and wait for memory, (`__GFP_HIGHMEM` flag) is high memory can be used, (`GFP_ATOMIC` flag) is allocation process high-priority and can't sleep etc.

The next structure is `rnode` :

```
struct radix_tree_node {
        unsigned int    path;
        unsigned int    count;
        union {
                struct {
                        struct radix_tree_node *parent;
                        void *private_data;
                };
                struct rcu_head rcu_head;
        };
        /* For tree user */
        struct list_head private_list;
        void __rcu      *slots[RADIX_TREE_MAP_SIZE];
        unsigned long   tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];
};
```

This structure contains information about the offset in a parent and height from the bottom, count of the child nodes and fields for accessing and freeing a node. The fields are described below:

* `path` - offset in parent & height from the bottom;
* `count` - count of the child nodes;
* `parent` - pointer to the parent node;
* `private_data` - used by the user of a tree;
* `rcu_head` - used for freeing a node;
* `private_list` - used by the user of a tree;

The two last fields of the `radix_tree_node` - `tags` and `slots` are important and interesting. Every node can contains a set of slots which are store pointers to the data. Empty slots in the linux kernel radix tree implementation store `NULL` . Radix tree in the linux kernel also supports tags which are associated with the `tags` fields in the `radix_tree_node` structure. Tags allow to set individual bits on records which are stored in the radix tree.

Now we know about radix tree structure, time to look on its API.

Linux kernel radix tree API

We start from the datastructure intialization. There are two ways to initialize new radix tree. The first is to use `RADIX_TREE` macro:

```
RADIX_TREE(name, gfp_mask);
`
```

As you can see we pass the `name` parameter, so with the `RADIX_TREE` macro we can define and initialize radix tree with the given name. Implementation of the `RADIX_TREE` is easy:

```
#define RADIX_TREE(name, mask) \
         struct radix_tree_root name = RADIX_TREE_INIT(mask)

#define RADIX_TREE_INIT(mask)   { \
        .height = 0,              \
        .gfp_mask = (mask),       \
        .rnode = NULL,            \
}
```

At the beginning of the `RADIX_TREE` macro we define instance of the `radix_tree_root` structure with the given name and call `RADIX_TREE_INIT` macro with the given mask. The `RADIX_TREE_INIT` macro just initializes `radix_tree_root` structure with the default values and the given mask.

The second way is to define `radix_tree_root` structure by hand and pass it with mask to the `INIT_RADIX_TREE` macro:

```
struct radix_tree_root my_radix_tree;
INIT_RADIX_TREE(my_tree, gfp_mask_for_my_radix_tree);
```

where:

```
#define INIT_RADIX_TREE(root, mask)  \
do {                                 \
        (root)->height = 0;          \
        (root)->gfp_mask = (mask);   \
        (root)->rnode = NULL;        \
} while (0)
```

makes the same initialziation with default values as it does `RADIX_TREE_INIT` macro.

The next are two functions for the inserting and deleting records to/from a radix tree:

- `radix_tree_insert` ;
- `radix_tree_delete` .

The first `radix_tree_insert` function takes three parameters:

- root of a radix tree;
- index key;
- data to insert;

The `radix_tree_delete` function takes the same set of parameters as the `radix_tree_insert` , but without data.

The search in a radix tree implemented in two ways:

- `radix_tree_lookup` ;
- `radix_tree_gang_lookup` ;
- `radix_tree_lookup_slot` .

The first `radix_tree_lookup` function takes two parameters:

- root of a radix tree;
- index key;

This function tries to find the given key in the tree and returns associated record with this key. The second `radix_tree_gang_lookup` function have the following signature

```
unsigned int radix_tree_gang_lookup(struct radix_tree_root *root,
                                    void **results,
                                    unsigned long first_index,
                                    unsigned int max_items);
```

and returns number of records, sorted by the keys, starting from the first index. Number of the returned records will be not greater than `max_items` value.

And the last `radix_tree_lookup_slot` function will return the slot which will contain the data.

Links

- Radix tree
- Trie

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:

```
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  |              |
 --------------------------------------------------------------------------------
```

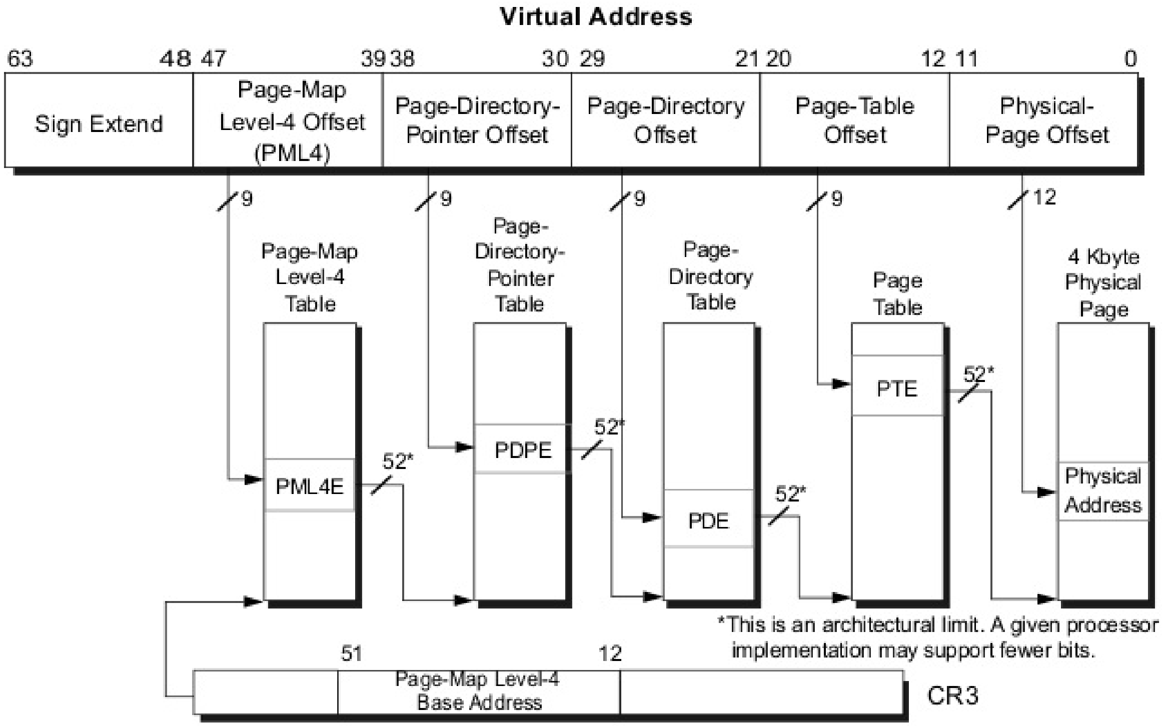
These fields have the following meanings:

- 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 well 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 a 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 is stored in 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. Now let's look on the `vmlinux`.

vmlinux

`vmlinux` is relocatable ELF object file too. So we can look at 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(Executable and Linkable Format), but if you are interested in it, you can find documentation - here

Misc

Thich chapter contains parts that are not directly related to the Linux kernel code and implementation of different subsystems.

Process of the Linux kernel building

Introduction

I won't tell you how to build and install a custom Linux kernel on your machine. If you need help with this, you can find many resources that will help you do it. Instead, we will learn what occurs when you type `make` in the directory of the Linux kernel source code.

When I started to study the source code of the Linux kernel, the makefile was the first file that I opened. And it was scary :). The makefile contained `1591` lines of code when I wrote this and this was the 4.2.0-rc3 release.

This makefile is the the top makefile in the Linux kernel source code and kernel build starts here. Yes, it is big, but moreover, if you've read the source code of the Linux kernel you can noted that all directories with a source code has an own makefile. Of course it is not real to describe how each source files compiled and linked. So, we will see compilation only for the standard case. You will not find here building of the kernel's documentation, cleaning of the kernel source code, tags generation, cross-compilation related stuff and etc. We will start from the `make` execution with the standard kernel configuration file and will finish with the building of the bzImage.

It would be good if you're already familiar with the make util, but I will anyway try to describe all code that will be in this part.

So let's start.

Preparation before the kernel compilation

There are many things to prepare before the kernel compilation will be started. The main point here is to find and configure The type of compilation, to parse command line arguments that are passed to the `make` util and etc. So let's dive into the top `Makefile` of the Linux kernel.

The Linux kernel top `Makefile` is responsible for building two major products: vmlinux (the resident kernel image) and the modules (any module files). The Makefile of the Linux kernel starts from the definition of the following variables:

```
VERSION = 4
PATCHLEVEL = 2
SUBLEVEL = 0
EXTRAVERSION = -rc3
NAME = Hurr durr I'ma sheep
```

These variables determine the current version of the Linux kernel and are used in the different places, for example in the forming of the `KERNELVERSION` variable:

```
KERNELVERSION = $(VERSION)$(if $(PATCHLEVEL),.$(PATCHLEVEL)$(if $(SUBLEVEL),.$(SUBLEVEL)))$(EXTRAVERSION)
```

After this we can see a couple of the `ifeq` condition that check some of the parameters passed to `make`. The Linux kernel `makefiles` provides a special `make help` target that prints all available targets and some of the command line arguments that can be passed to `make`. For example: `make V=1` - provides verbose builds. The first `ifeq` condition checks if the `V=n` option is passed to make:

```
ifeq ("$(origin V)", "command line")
  KBUILD_VERBOSE = $(V)
```

```
endif
ifndef KBUILD_VERBOSE
  KBUILD_VERBOSE = 0
endif

ifeq ($(KBUILD_VERBOSE),1)
  quiet =
  Q =
else
  quiet=quiet_
  Q = @
endif

export quiet Q KBUILD_VERBOSE
```

If this option is passed to `make` we set the `KBUILD_VERBOSE` variable to the value of the `V` option. Otherwise we set the `KBUILD_VERBOSE` variable to zero. After this we check value of the `KBUILD_VERBOSE` variable and set values of the `quiet` and `Q` variables depends on the `KBUILD_VERBOSE` value. The `@` symbols suppress the output of the command and if it will be set before a command we will see something like this: `CC scripts/mod/empty.o` instead of the `Compiling .... scripts/mod/empty.o`. In the end we just export all of these variables. The next `ifeq` statement checks that `O=/dir` option was passed to the `make`. This option allows to locate all output files in the given `dir`:

```
ifeq ($(KBUILD_SRC),)

ifeq ("$(origin O)", "command line")
  KBUILD_OUTPUT := $(O)
endif

ifneq ($(KBUILD_OUTPUT),)
saved-output := $(KBUILD_OUTPUT)
KBUILD_OUTPUT := $(shell mkdir -p $(KBUILD_OUTPUT) && cd $(KBUILD_OUTPUT) \
                                && /bin/pwd)
$(if $(KBUILD_OUTPUT),, \
     $(error failed to create output directory "$(saved-output)"))

sub-make: FORCE
    $(Q)$(MAKE) -C $(KBUILD_OUTPUT) KBUILD_SRC=$(CURDIR) \
    -f $(CURDIR)/Makefile $(filter-out _all sub-make,$(MAKECMDGOALS))

skip-makefile := 1
endif # ifneq ($(KBUILD_OUTPUT),)
endif # ifeq ($(KBUILD_SRC),)
```

We check the `KBUILD_SRC` that represent top directory of the source code of the linux kernel and if it is empty (it is empty every time while makefile executes first time) and the set the `KBUILD_OUTPUT` variable to the value that passed with the `O` option (if this option was passed). In the next step we check this `KBUILD_OUTPUT` variable and if we set it, we do following things:

- Store value of the `KBUILD_OUTPUT` in the temp `saved-output` variable;
- Try to create given output directory;
- Check that directory created, in other way print error;
- If custom output directory created successfully, execute `make` again with the new directory (see `-C` option).

The next `ifeq` statements checks that `C` or `M` options was passed to the make:

```
ifeq ("$(origin C)", "command line")
  KBUILD_CHECKSRC = $(C)
endif
ifndef KBUILD_CHECKSRC
  KBUILD_CHECKSRC = 0
endif

ifeq ("$(origin M)", "command line")
  KBUILD_EXTMOD := $(M)
```

```
endif
```

The first `C` option tells to the `makefile` that need to check all `c` source code with a tool provided by the `$CHECK` environment variable, by default it is sparse. The second `M` option provides build for the external modules (will not see this case in this part). As we set this variables we make a check of the `KBUILD_SRC` variable and if it is not set we set `srctree` variable to `.`:

```
ifeq ($(KBUILD_SRC),)
        srctree := .
endif

objtree    := .
src        := $(srctree)
obj        := $(objtree)

export srctree objtree VPATH
```

That tells to `Makefile` that source tree of the Linux kernel will be in the current directory where `make` command was executed. After this we set `objtree` and other variables to this directory and export these variables. The next step is the getting value for the `SUBARCH` variable that will represent what the underlying architecture is:

```
SUBARCH := $(shell uname -m | sed -e s/i.86/x86/ -e s/x86_64/x86/ \
                  -e s/sun4u/sparc64/ \
                  -e s/arm.*/arm/ -e s/sa110/arm/ \
                  -e s/s390x/s390/ -e s/parisc64/parisc/ \
                  -e s/ppc.*/powerpc/ -e s/mips.*/mips/ \
                  -e s/sh[234].*/sh/ -e s/aarch64.*/arm64/ )
```

As you can see it executes uname utils that prints information about machine, operating system and architecture. As it will get output of the `uname` util, it will parse it and assign to the `SUBARCH` variable. As we got `SUBARCH`, we set the `SRCARCH` variable that provides directory of the certain architecture and `hfr-arch` that provides directory for the header files:

```
ifeq ($(ARCH),i386)
        SRCARCH := x86
endif
ifeq ($(ARCH),x86_64)
        SRCARCH := x86
endif

hdr-arch  := $(SRCARCH)
```

Note that `ARCH` is the alias for the `SUBARCH`. In the next step we set the `KCONFIG_CONFIG` variable that represents path to the kernel configuration file and if it was not set before, it will be `.config` by default:

```
KCONFIG_CONFIG    ?= .config
export KCONFIG_CONFIG
```

and the shell that will be used during kernel compilation:

```
CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \
      else if [ -x /bin/bash ]; then echo /bin/bash; \
      else echo sh; fi ; fi)
```

The next set of variables related to the compiler that will be used during Linux kernel compilation. We set the host compilers for the `c` and `c++` and flags for it:

```
HOSTCC       = gcc
HOSTCXX      = g++
HOSTCFLAGS   = -Wall -Wmissing-prototypes -Wstrict-prototypes -O2 -fomit-frame-pointer -std=gnu89
HOSTCXXFLAGS = -O2
```

Next we will meet the `CC` variable that represent compiler too, so why do we need in the `HOST*` variables? The `CC` is the target compiler that will be used during kernel compilation, but `HOSTCC` will be used during compilation of the set of the `host` programs (we will see it soon). After this we can see definition of the `KBUILD_MODULES` and `KBUILD_BUILTIN` variables that are used for the determination of the what to compile (kernel, modules or both):

```
KBUILD_MODULES :=
KBUILD_BUILTIN := 1

ifeq ($(MAKECMDGOALS),modules)
  KBUILD_BUILTIN := $(if $(CONFIG_MODVERSIONS),1)
endif
```

Here we can see definition of these variables and the value of the `KBUILD_BUILTIN` will depens on the `CONFIG_MODVERSIONS` kernel configuration parameter if we pass only `modules` to the `make`. The next step is including of the:

```
include scripts/Kbuild.include
```

`kbuild` file. The Kbuild or `Kernel Build System` is the special infrastructure to manage building of the kernel and its modules. The `kbuild` files has the same syntax that makefiles. The scripts/Kbuild.include file provides some generic definitions for the `kbuild` system. As we included this `kbuild` files we can see definition of the variables that are related to the different tools that will be used during kernel and modules compilation (like linker, compilers, utils from the binutils and etc...):

```
AS        = $(CROSS_COMPILE)as
LD        = $(CROSS_COMPILE)ld
CC        = $(CROSS_COMPILE)gcc
CPP        = $(CC) -E
AR        = $(CROSS_COMPILE)ar
NM        = $(CROSS_COMPILE)nm
STRIP        = $(CROSS_COMPILE)strip
OBJCOPY        = $(CROSS_COMPILE)objcopy
OBJDUMP        = $(CROSS_COMPILE)objdump
AWK        = awk
...
...
...
```

After definition of these variables we define two variables: `USERINCLUDE` and `LINUXINCLUDE`. They will contain paths of the directories with headers (public for users in the first case and for kernel in the second case):

```
USERINCLUDE    := \
        -I$(srctree)/arch/$(hdr-arch)/include/uapi \
        -Iarch/$(hdr-arch)/include/generated/uapi \
        -I$(srctree)/include/uapi \
        -Iinclude/generated/uapi \
        -include $(srctree)/include/linux/kconfig.h

LINUXINCLUDE    := \
        -I$(srctree)/arch/$(hdr-arch)/include \
        ...
```

And the standard flags for the C compiler:

```
KBUILD_CFLAGS   := -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
           -fno-strict-aliasing -fno-common \
           -Werror-implicit-function-declaration \
           -Wno-format-security \
           -std=gnu89
```

It is the not last compiler flags, they can be updated by the other makefiles (for example kbuilds from `arch/`). After all of these, all variables will be exported to be available in the other makefiles. The following two the `RCS_FIND_IGNORE` and the `RCS_TAR_IGNORE` variables will contain files that will be ignored in the version control system:

```
export RCS_FIND_IGNORE := \( -name SCCS -o -name BitKeeper -o -name .svn -o    \
              -name CVS -o -name .pc -o -name .hg -o -name .git \) \
              -prune -o
export RCS_TAR_IGNORE := --exclude SCCS --exclude BitKeeper --exclude .svn \
             --exclude CVS --exclude .pc --exclude .hg --exclude .git
```

That's all. We have finished with the all preparations, next point is the building of `vmlinux` .

Directly to the kernel build

As we have finished all preparations, next step in the root makefile is related to the kernel build. Before this moment we will not see in the our terminal after the execution of the `make` command. But now first steps of the compilation are started. In this moment we need to go on the 598 line of the Linux kernel top makefile and we will see `vmlinux` target there:

```
all: vmlinux
    include arch/$(SRCARCH)/Makefile
```

Don't worry that we have missed many lines in Makefile that are placed after `export RCS_FIND_IGNORE.....` and before `all: vmlinux.....` . This part of the makefile is responsible for the `make *.config` targets and as I wrote in the beginning of this part we will see only building of the kernel in a general way.

The `all:` target is the default when no target is given on the command line. You can see here that we include architecture specific makefile there (in our case it will be arch/x86/Makefile). From this moment we will continue from this makefile. As we can see `all` target depends on the `vmlinux` target that defined a little lower in the top makefile:

```
vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
```

The `vmlinux` is the Linux kernel in a statically linked executable file format. The scripts/link-vmlinux.sh script links and combines different compiled subsystems into vmlinux. The second target is the `vmlinux-deps` that defined as:

```
vmlinux-deps := $(KBUILD_LDS) $(KBUILD_VMLINUX_INIT) $(KBUILD_VMLINUX_MAIN)
```

and consists from the set of the `built-in.o` from the each top directory of the Linux kernel. Later, when we will go through all directories in the Linux kernel, the `Kbuild` will compile all the `$(obj-y)` files. It then calls `$(LD) -r` to merge these files into one `built-in.o` file. For this moment we have no `vmlinux-deps` , so the `vmlinux` target will not be executed now. For me `vmlinux-deps` contains following files:

```
arch/x86/kernel/vmlinux.lds arch/x86/kernel/head_64.o
arch/x86/kernel/head64.o    arch/x86/kernel/head.o
init/built-in.o             usr/built-in.o
```

```
arch/x86/built-in.o        kernel/built-in.o
mm/built-in.o              fs/built-in.o
ipc/built-in.o             security/built-in.o
crypto/built-in.o          block/built-in.o
lib/lib.a                  arch/x86/lib/lib.a
lib/built-in.o             arch/x86/lib/built-in.o
drivers/built-in.o         sound/built-in.o
firmware/built-in.o        arch/x86/pci/built-in.o
arch/x86/power/built-in.o  arch/x86/video/built-in.o
net/built-in.o
```

The next target that can be executed is following:

```
$(sort $(vmlinux-deps)): $(vmlinux-dirs) ;
$(vmlinux-dirs): prepare scripts
    $(Q)$(MAKE) $(build)=$@
```

As we can see the `vmlinux-dirs` depends on the two targets: `prepare` and `scripts` . The first `prepare` defined in the top `Makefile` of the Linux kernel and executes three stages of preparations:

```
prepare: prepare0
prepare0: archprepare FORCE
    $(Q)$(MAKE) $(build)=.
archprepare: archheaders archscripts prepare1 scripts_basic

prepare1: prepare2 $(version_h) include/generated/utsrelease.h \
                   include/config/auto.conf
    $(cmd_crmodverdir)
prepare2: prepare3 outputmakefile asm-generic
```

The first `prepare0` expands to the `archprepare` that expands to the `archheaders` and `archscripts` that defined in the `x86_64` specific Makefile. Let's look on it. The `x86_64` specific makefile starts from the definition of the variables that are related to the architecture-specific configs (defconfig and etc.). After this it defines flags for the compiling of the 16-bit code, calculating of the `BITS` variable that can be `32` for `i386` or `64` for the `x86_64` flags for the assembly source code, flags for the linker and many many more (all definitions you can find in the arch/x86/Makefile). The first target is `archheaders` in the makefile generates syscall table:

```
archheaders:
    $(Q)$(MAKE) $(build)=arch/x86/entry/syscalls all
```

And the second target is `archscripts` in this makefile is:

```
archscripts: scripts_basic
    $(Q)$(MAKE) $(build)=arch/x86/tools relocs
```

We can see that it depends on the `scripts_basic` target from the top Makefile. At the first we can see the `scripts_basic` target that executes make for the scripts/basic makefile:

```
scripts_basic:
    $(Q)$(MAKE) $(build)=scripts/basic
```

The `scripts/basic/Makefile` contains targets for compilation of the two host programs: `fixdep` and `bin2` :

```
hostprogs-y    := fixdep
```

```
hostprogs-$(CONFIG_BUILD_BIN2C)     += bin2c
always        := $(hostprogs-y)

$(addprefix $(obj)/,$(filter-out fixdep,$(always))): $(obj)/fixdep
```

First program is `fixdep` - optimizes list of dependencies generated by the gcc that tells make when to remake a source code file. The second program is `bin2c` depends on the value of the `CONFIG_BUILD_BIN2C` kernel configuration option and very little C program that allows to convert a binary on stdin to a C include on stdout. You can note here strange notation: `hostprogs-y` and etc. This notation is used in the all `kbuild` files and more about it you can read in the documentation. In our case the `hostprogs-y` tells to the `kbuild` that there is one host program named `fixdep` that will be built from the will be built from `fixdep.c` that located in the same directory that `Makefile`. The first output after we will execute `make` command in our terminal will be result of this `kbuild` file:

```
$ make
  HOSTCC  scripts/basic/fixdep
```

As `script_basic` target was executed, the `archscripts` target will execute `make` for the arch/x86/tools makefile with the `relocs` target:

```
$(Q)$(MAKE) $(build)=arch/x86/tools relocs
```

The `relocs_32.c` and the `relocs_64.c` will be compiled that will contain relocation information and we will see it in the `make` output:

```
  HOSTCC  arch/x86/tools/relocs_32.o
  HOSTCC  arch/x86/tools/relocs_64.o
  HOSTCC  arch/x86/tools/relocs_common.o
  HOSTLD  arch/x86/tools/relocs
```

There is checking of the `version.h` after compiling of the `relocs.c`:

```
$(version_h): $(srctree)/Makefile FORCE
    $(call filechk,version.h)
    $(Q)rm -f $(old_version_h)
```

We can see it in the output:

```
CHK     include/config/kernel.release
```

and the building of the `generic` assembly headers with the `asm-generic` target from the `arch/x86/include/generated/asm` that generated in the top Makefile of the Linux kernel. After the `asm-generic` target the `archprepare` will be done, so the `prepare0` target will be executed. As I wrote above:

```
prepare0: archprepare FORCE
    $(Q)$(MAKE) $(build)=.
```

Note on the `build`. It defined in the scripts/Kbuild.include and looks like this:

```
build := -f $(srctree)/scripts/Makefile.build obj
```

Or in our case it is current source directory - `.` :

```
$(Q)$(MAKE) -f $(srctree)/scripts/Makefile.build obj=.
```

The scripts/Makefile.build tries to find the `Kbuild` file by the given directory via the `obj` parameter, include this `Kbuild` files:

```
include $(kbuild-file)
```

and build targets from it. In our case `.` contains the Kbuild file that generates the `kernel/bounds.s` and the `arch/x86/kernel/asm-offsets.s`. After this the `prepare` target finished to work. The `vmlinux-dirs` also depends on the second target - `scripts` that compiles following programs: `file2alias`, `mk_elfconfig`, `modpost` and etc... After scripts/host-programs compilation our `vmlinux-dirs` target can be executed. First of all let's try to understand what does `vmlinux-dirs` contain. For my case it contains paths of the following kernel directories:

```
init usr arch/x86 kernel mm fs ipc security crypto block
drivers sound firmware arch/x86/pci arch/x86/power
arch/x86/video net lib arch/x86/lib
```

We can find definition of the `vmlinux-dirs` in the top Makefile of the Linux kernel:

```
vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
             $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
             $(net-y) $(net-m) $(libs-y) $(libs-m)))

init-y        := init/
drivers-y    := drivers/ sound/ firmware/
net-y        := net/
libs-y        := lib/
...
...
...
```

Here we remove the `/` symbol from the each directory with the help of the `patsubst` and `filter` functions and put it to the `vmlinux-dirs`. So we have list of directories in the `vmlinux-dirs` and the following code:

```
$(vmlinux-dirs): prepare scripts
    $(Q)$(MAKE) $(build)=$@
```

The `$@` represents `vmlinux-dirs` here that means that it will go recursively over all directories from the `vmlinux-dirs` and its internal directories (depens on configuration) and will execute `make` in there. We can see it in the output:

```
  CC      init/main.o
  CHK     include/generated/compile.h
  CC      init/version.o
  CC      init/do_mounts.o
  ...
  CC      arch/x86/crypto/glue_helper.o
  AS      arch/x86/crypto/aes-x86_64-asm_64.o
  CC      arch/x86/crypto/aes_glue.o
  ...
  AS      arch/x86/entry/entry_64.o
  AS      arch/x86/entry/thunk_64.o
  CC      arch/x86/entry/syscall_64.o
```

Source code in each directory will be compiled and linked to the `built-in.o`:

```
$ find . -name built-in.o
./arch/x86/crypto/built-in.o
./arch/x86/crypto/sha-mb/built-in.o
./arch/x86/net/built-in.o
./init/built-in.o
./usr/built-in.o
...
...
```

Ok, all buint-in.o(s) built, now we can back to the `vmlinux` target. As you remember, the `vmlinux` target is in the top Makefile of the Linux kernel. Before the linking of the `vmlinux` it builds samples, Documentation and etc., but I will not describe it in this part as I wrote in the beginning of this part.

```
vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
    ...
    ...
    +$(call if_changed,link-vmlinux)
```

As you can see main purpose of it is a call of the scripts/link-vmlinux.sh script is linking of the all `built-in.o`(s) to the one statically linked executable and creation of the System.map. In the end we will see following output:

```
  LINK    vmlinux
  LD      vmlinux.o
  MODPOST vmlinux.o
  GEN     .version
  CHK     include/generated/compile.h
  UPD     include/generated/compile.h
  CC      init/version.o
  LD      init/built-in.o
  KSYM    .tmp_kallsyms1.o
  KSYM    .tmp_kallsyms2.o
  LD      vmlinux
  SORTEX  vmlinux
  SYSMAP  System.map
```

and `vmlinux` and `System.map` in the root of the Linux kernel source tree:

```
$ ls vmlinux System.map
System.map  vmlinux
```

That's all, `vmlinux` is ready. The next step is creation of the bzImage.

Building bzImage

The `bzImage` is the compressed Linux kernel image. We can get it with the execution of the `make bzImage` after the `vmlinux` built. In other way we can just execute `make` without arguments and will get `bzImage` anyway because it is default image:

```
all: bzImage
```

in the arch/x86/kernel/Makefile. Let's look on this target, it will help us to understand how this image builds. As I already said the `bzImage` target defined in the arch/x86/kernel/Makefile and looks like this:

```
bzImage: vmlinux
    $(Q)$(MAKE) $(build)=$(boot) $(KBUILD_IMAGE)
    $(Q)mkdir -p $(objtree)/arch/$(UTS_MACHINE)/boot
    $(Q)ln -fsn ../../x86/boot/bzImage $(objtree)/arch/$(UTS_MACHINE)/boot/$@
```

We can see here, that first of all called `make` for the boot directory, in our case it is:

```
boot := arch/x86/boot
```

The main goal now to build source code in the `arch/x86/boot` and `arch/x86/boot/compressed` directories, build `setup.bin` and `vmlinux.bin`, and build the `bzImage` from they in the end. First target in the arch/x86/boot/Makefile is the `$(obj)/setup.elf`:

```
$(obj)/setup.elf: $(src)/setup.ld $(SETUP_OBJS) FORCE
    $(call if_changed,ld)
```

We already have the `setup.ld` linker script in the `arch/x86/boot` directory and the `SETUP_OBJS` expands to the all source files from the `boot` directory. We can see first output:

```
  AS      arch/x86/boot/bioscall.o
  CC      arch/x86/boot/cmdline.o
  AS      arch/x86/boot/copy.o
  HOSTCC  arch/x86/boot/mkcpustr
  CPUSTR  arch/x86/boot/cpustr.h
  CC      arch/x86/boot/cpu.o
  CC      arch/x86/boot/cpuflags.o
  CC      arch/x86/boot/cpucheck.o
  CC      arch/x86/boot/early_serial_console.o
  CC      arch/x86/boot/edd.o
```

The next source code file is the arch/x86/boot/header.S, but we can't build it now because this target depends on the following two header files:

```
$(obj)/header.o: $(obj)/voffset.h $(obj)/zoffset.h
```

The first is `voffset.h` generated by the `sed` script that gets two addresses from the `vmlinux` with the `nm` util:

```
#define VO__end 0xffffffff82ab0000
#define VO__text 0xffffffff81000000
```

They are start and end of the kernel. The second is `zoffset.h` depens on the `vmlinux` target from the arch/x86/boot/compressed/Makefile:

```
$(obj)/zoffset.h: $(obj)/compressed/vmlinux FORCE
    $(call if_changed,zoffset)
```

The `$(obj)/compressed/vmlinux` target depends on the `vmlinux-objs-y` that compiles source code files from the arch/x86/boot/compressed directory and generates `vmlinux.bin`, `vmlinux.bin.bz2`, and compiles programm - `mkpiggy`. We can see this in the output:

```
LDS     arch/x86/boot/compressed/vmlinux.lds
AS      arch/x86/boot/compressed/head_64.o
CC      arch/x86/boot/compressed/misc.o
CC      arch/x86/boot/compressed/string.o
CC      arch/x86/boot/compressed/cmdline.o
OBJCOPY arch/x86/boot/compressed/vmlinux.bin
BZIP2   arch/x86/boot/compressed/vmlinux.bin.bz2
HOSTCC  arch/x86/boot/compressed/mkpiggy
```

Where the `vmlinux.bin` is the `vmlinux` with striped debuging information and comments and the `vmlinux.bin.bz2` compressed `vmlinux.bin.all` + `u32` size of `vmlinux.bin.all`. The `vmlinux.bin.all` is `vmlinux.bin + vmlinux.relocs`, where `vmlinux.relocs` is the `vmlinux` that was handled by the `relocs` program (see above). As we got these files, the `piggy.S` assembly files will be generated with the `mkpiggy` program and compiled:

```
MKPIGGY arch/x86/boot/compressed/piggy.S
AS      arch/x86/boot/compressed/piggy.o
```

This assembly files will contain computed offset from a compressed kernel. After this we can see that `zoffset` generated:

```
ZOFFSET arch/x86/boot/zoffset.h
```

As the `zoffset.h` and the `voffset.h` are generated, compilation of the source code files from the arch/x86/boot can be continued:

```
AS      arch/x86/boot/header.o
CC      arch/x86/boot/main.o
CC      arch/x86/boot/mca.o
CC      arch/x86/boot/memory.o
CC      arch/x86/boot/pm.o
AS      arch/x86/boot/pmjump.o
CC      arch/x86/boot/printf.o
CC      arch/x86/boot/regs.o
CC      arch/x86/boot/string.o
CC      arch/x86/boot/tty.o
CC      arch/x86/boot/video.o
CC      arch/x86/boot/video-mode.o
CC      arch/x86/boot/video-vga.o
CC      arch/x86/boot/video-vesa.o
CC      arch/x86/boot/video-bios.o
```

As all source code files will be compiled, they will be linked to the `setup.elf`:

```
LD      arch/x86/boot/setup.elf
```

or:

```
ld -m elf_x86_64   -T arch/x86/boot/setup.ld arch/x86/boot/a20.o arch/x86/boot/bioscall.o arch/x86/boot/cmdline.o arch/
```

The last two things is the creation of the `setup.bin` that will contain compiled code from the `arch/x86/boot/*` directory:

```
objcopy  -O binary arch/x86/boot/setup.elf arch/x86/boot/setup.bin
```

and the creation of the `vmlinux.bin` from the `vmlinux` :

```
objcopy  -O binary -R .note -R .comment -S arch/x86/boot/compressed/vmlinux arch/x86/boot/vmlinux.bin
```

In the end we compile host program: arch/x86/boot/tools/build.c that will create our `bzImage` from the `setup.bin` and the `vmlinux.bin` :

```
arch/x86/boot/tools/build arch/x86/boot/setup.bin arch/x86/boot/vmlinux.bin arch/x86/boot/zoffset.h arch/x86/boot/bzIma
```

Actually the `bzImage` is the concatenated `setup.bin` and the `vmlinux.bin` . In the end we will see the output which familiar to all who once build the Linux kernel from source:

```
Setup is 16268 bytes (padded to 16384 bytes).
System is 4704 kB
CRC 94a88f9a
Kernel: arch/x86/boot/bzImage is ready  (#5)
```

That's all.

Conclusion

It is the end of this part and here we saw all steps from the execution of the `make` command to the generation of the `bzImage` . I know, the Linux kernel makefiles and process of the Linux kernel building may seem confusing at first glance, but it is not so hard. Hope this part will help you to understand process of the Linux kernel building.

Links

- GNU make util
- Linux kernel top Makefile
- cross-compilation
- Ctags
- sparse
- bzImage
- uname
- shell
- Kbuild
- binutils
- gcc
- Documentation
- System.map
- Relocation

Introduction

During the writing of the linux-insides book I have received many emails with questions related to the linker script and linker-related subjects. So I've decided to write this to cover some aspects of the linker and the linking of object files.

If we open page the `Linker` page on wikipidia, we can see the following definition:

> In computer science, a linker or link editor is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file.

If you've written at least one program on C in your life, you will have seen files with the `*.o` extension. These files are object files. Object files are blocks of machine code and data with placeholder addresses that reference data and functions in other object files or libraries, as well as a list of its own functions and data. The main purpose of the linker is collect/handle the code and data of each object file, turning it into the the final executable file or library. In this post we will try to go through all aspects of this process. Let's start.

Linking process

Let's create simple project with the following structure:

```
*-linkers
*--main.c
*--lib.c
*--lib.h
```

And write there our example factorial program. Our `main.c` source code file contains:

```
#include <stdio.h>

#include "lib.h"

int main(int argc, char **argv) {
    printf("factorial of 5 is: %d\n", factorial(5));
    return 0;
}
```

The `lib.c` file contains:

```
int factorial(int base) {
    int res = 1, i = 1;

    if (base == 0) {
        return 1;
    }

    while (i <= base) {
        res *= i;
        i++;
    }

    return res;
}
```

And the `lib.h` file contains:

```
#ifndef LIB_H
#define LIB_H

int factorial(int base);

#endif
```

Now let's compile only the `main.c` source code file with:

```
$ gcc -c main.c
```

If we look inside the outputted object file with the `nm` util, we will see the following output:

```
$ nm -A main.o
main.o:                 U factorial
main.o:0000000000000000 T main
main.o:                 U printf
```

The `nm` util allows us to see the list of symbols from the given object file. It consists of three columns: the first is the name of the given object file and the address of any resolved symbols. The second column contains a character that represents the status of the given symbol. In this case the `U` means `undefined` and the `T` denotes that the symbols are placed in the `.text` section of the object. The `nm` utility shows us here that we have three symbols in the `main.c` source code file:

* `factorial` - the factorial function defined in the `lib.c` source code file. It is marked as `undefined` here because we compiled only the `main.c` source code file, and it does not know anything about code from the `lib.c` file for now;
* `main` - the main function;
* `printf` - the function from the glibc library. `main.c` does not know anything about it for now either.

What can we understand from the output of `nm` so far? The `main.o` object file contains the local symbol `main` at address `0000000000000000` (it will be filled with correct address after is is linked), and two unresolved symbols. We can see all of this information in the disassembly output of the `main.o` object file:

```
$ objdump -S main.o

main.o:     file format elf64-x86-64
Disassembly of section .text:

0000000000000000 <main>:
   0:	55                   	push   %rbp
   1:	48 89 e5             	mov    %rsp,%rbp
   4:	48 83 ec 10          	sub    $0x10,%rsp
   8:	89 7d fc             	mov    %edi,-0x4(%rbp)
   b:	48 89 75 f0          	mov    %rsi,-0x10(%rbp)
   f:	bf 05 00 00 00       	mov    $0x5,%edi
  14:	e8 00 00 00 00       	callq  19 <main+0x19>
  19:	89 c6                	mov    %eax,%esi
  1b:	bf 00 00 00 00       	mov    $0x0,%edi
  20:	b8 00 00 00 00       	mov    $0x0,%eax
  25:	e8 00 00 00 00       	callq  2a <main+0x2a>
  2a:	b8 00 00 00 00       	mov    $0x0,%eax
  2f:	c9                   	leaveq
  30:	c3                   	retq
```

Here we are interested only in the two `callq` operations. The two `callq` operations contain `linker stubs`, or the function name and offset from it to the next instruction. These stubs will be updated to the real addresses of the functions. We can see these functions' names with in the following `objdump` output:

```
$ objdump -S -r main.o
```

```
...
  14:    e8 00 00 00 00          callq  19 <main+0x19>
            15: R_X86_64_PC32    factorial-0x4
  19:    89 c6                   mov    %eax,%esi
...
  25:    e8 00 00 00 00          callq  2a <main+0x2a>
            26: R_X86_64_PC32    printf-0x4
  2a:    b8 00 00 00 00          mov    $0x0,%eax
...
```

The `-r` or `--reloc` flags of the `objdump` util print the `relocation` entries of the file. Now let's look in more detail at the relocation process.

Relocation

Relocation is the process of connecting symbolic references with symbolic definitions. Let's look at the previous snippet from the `objdump` output:

```
  14:    e8 00 00 00 00          callq  19 <main+0x19>
            15: R_X86_64_PC32    factorial-0x4
  19:    89 c6                   mov    %eax,%esi
```

Note `e8 00 00 00 00` on the first line. The `e8` is the opcode of the `call` instruction with a relative offset. So the `e8 00 00 00 00` contains a one-byte operation code followed by a four-byte address. Note that the `00 00 00 00` is 4-bytes, but why only 4-bytes if an address can be 8-bytes in the `x86_64`? Actually we compiled the `main.c` source code file with the `-mcmodel=small`. From the `gcc` man:

```
-mcmodel=small
    Generate code for the small code model: the program and its symbols must be linked in the lower 2 GB of the address
```

Of course we didn't pass this option to the `gcc` when we compiled the `main.c`, but it is default. We know that our program will be linked in the lower 2 GB of the address space from the quote from the `gcc` manual. With this code model, 4-bytes is enough to represent the address. So we have opcode of the `call` instruction and unknown address. When we compile `main.c` with all dependencies to the executable file and will look on the call of the factorial we will see:

```
$ gcc main.c lib.c -o factorial | objdump -S factorial | grep factorial

factorial:     file format elf64-x86-64
...
...
0000000000400506 <main>:
    40051a:    e8 18 00 00 00           callq  400537 <factorial>
...
...
0000000000400537 <factorial>:
    400550:    75 07                    jne    400559 <factorial+0x22>
    400557:    eb 1b                    jmp    400574 <factorial+0x3d>
    400559:    eb 0e                    jmp    400569 <factorial+0x32>
    40056f:    7e ea                    jle    40055b <factorial+0x24>
...
...
```

As we can see in the previous output, the address of the `main` function is `0x0000000000400506`. Why it does not starts from the `0x0`? You may already know that standard C programs are linked with the `glibc` C standard library unless `-nostdlib` is passed to `gcc`. The compiled code for a program includes constructors functions to initialize data in the program when the program is started. These functions need to be called before the program is started or in another words before the `main` function is called. To make the initialization and termination functions work, the compiler must output something in the assembler code to cause those functions to be called at the appropriate time. Execution of this program will starts from the code that is placed in the special section which is called `.init`. We can see it in the beginning of the objdump output:

```
objdump -S factorial | less

factorial:     file format elf64-x86-64

Disassembly of section .init:

00000000004003a8 <_init>:
  4003a8:       48 83 ec 08             sub    $0x8,%rsp
  4003ac:       48 8b 05 a5 05 20 00    mov    0x2005a5(%rip),%rax        # 600958 <_DYNAMIC+0x1d0>
```

Note that it starts at the `0x00000000004003a8` address relative to the `glibc` code. We can check it also in the resulted ELF:

```
$ readelf -d factorial | grep \(INIT\)
 0x000000000000000c (INIT)               0x4003a8
```

So, the address of the `main` function is the `0000000000400506` and it is offset from the `.init` section. As we can see from the output, the address of the `factorial` function is `0x0000000000400537` and binary code for the call of the `factorial` function now is `e8 18 00 00 00`. We already know that `e8` is opcode for the `call` instruction, the next `18 00 00 00` (note that address represented as little endian for the `x86_64`, in other words it is `00 00 00 18`) is the offset from the `callq` to the `factorial` function:

```
>>> hex(0x40051a + 0x18 + 0x5) == hex(0x400537)
True
```

So we add `0x18` and `0x5` to the address of the `call` instruction. The offset is measured from the address of the following instruction. Our call instruction is 5-bytes size - `e8 18 00 00 00` and the `0x18` is the offset from the next after call instruction to the `factorial` function. A compiler generally creates each object file with the program addresses starting at zero. But if a program is created from multiple object files, all of them will be overlapped. Just now we saw a process which is called `relocation`. This process assigns load addresses to the various parts of the program, adjusting the code and data in the program to reflect the assigned addresses.

Ok, now we know a little about linkers and relocation. Time to link our object files and to know more about linkers.

GNU linker

As you can understand from the title, I will use GNU linker or just `ld` in this post. Of course we can use `gcc` to link our `factorial` project:

```
$ gcc main.c lib.o -o factorial
```

and after it we will get executable file - `factorial` as a result:

```
./factorial
factorial of 5 is: 120
```

But `gcc` does not link object files. Instead it uses `collect2` which is just wrapper for the `GNU ld` linker:

```
~$ /usr/lib/gcc/x86_64-linux-gnu/4.9/collect2 --version
collect2 version 4.9.3
/usr/bin/ld --version
GNU ld (GNU Binutils for Debian) 2.25
...
...
...
```

Ok, we can use gcc and it will produce executable file of our program for us. But let's look how to use `GNU ld` linker for the same purpose. First of all let's try to link these object files with the following example:

```
ld main.o lib.o -o factorial
```

Try to do it and you will get following error:

```
$ ld main.o lib.o -o factorial
ld: warning: cannot find entry symbol _start; defaulting to 00000000004000b0
main.o: In function `main':
main.c:(.text+0x26): undefined reference to `printf'
```

Here we can see two problems:

- Linker can't find `_start` symbol;
- Linker does not know anything about `printf` function.

First of all let's try to understand what is this `_start` entry symbol that appears to be required for our program to run? When I started to learn programming I learned that the `main` function is the entry point of the program. I think you learned this too :) But it actually isn't the entry point, it's `_start` instead. The `_start` symbol is defined in the `crt1.o` object file. We can find it with the following command:

```
$ objdump -S /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o

/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:    31 ed                   xor    %ebp,%ebp
   2:    49 89 d1                mov    %rdx,%r9
   ...
   ...
   ...
```

We pass this object file to the `ld` command as its first argument (see above). Now let's try to link it and will look on result:

```
ld /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o \
main.o lib.o -o factorial

/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o: In function `_start':
/tmp/buildd/glibc-2.19/csu/../sysdeps/x86_64/start.S:115: undefined reference to `__libc_csu_fini'
/tmp/buildd/glibc-2.19/csu/../sysdeps/x86_64/start.S:116: undefined reference to `__libc_csu_init'
/tmp/buildd/glibc-2.19/csu/../sysdeps/x86_64/start.S:122: undefined reference to `__libc_start_main'
main.o: In function `main':
main.c:(.text+0x26): undefined reference to `printf'
```

Unfortunately we will see even more errors. We can see here old error about undefined `printf` and yet another three undefined references:

- `__libc_csu_fini`
- `__libc_csu_init`
- `__libc_start_main`

The `_start` symbol is defined in the sysdeps/x86_64/start.S assembly file in the `glibc` source code. We can find following assembly code lines there:

```
mov $__libc_csu_fini, %R8_LP
mov $__libc_csu_init, %RCX_LP
...
call __libc_start_main
```

Here we pass address of the entry point to the `.init` and `.fini` section that contain code that starts to execute when the program is ran and the code that executes when program terminates. And in the end we see the call of the `main` function from our program. These three symbols are defined in the csu/elf-init.c source code file. The following two object files:

- `crtn.o`;
- `crtn.i`.

define the function prologs/epilogs for the .init and .fini sections (with the `_init` and `_fini` symbols respectively).

The `crtn.o` object file contains these `.init` and `.fini` sections:

```
$ objdump -S /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o

0000000000000000 <.init>:
   0:	48 83 c4 08          	add    $0x8,%rsp
   4:	c3                   	retq

Disassembly of section .fini:

0000000000000000 <.fini>:
   0:	48 83 c4 08          	add    $0x8,%rsp
   4:	c3                   	retq
```

And the `crti.o` object file contains the `_init` and `_fini` symbols. Let's try to link again with these two object files:

```
$ ld \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crti.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o main.o lib.o \
-o factorial
```

And anyway we will get the same errors. Now we need to pass `-lc` option to the `ld`. This option will search for the standard library in the paths present in the `$LD_LIBRARY_PATH` enviroment variable. Let's try to link again wit the `-lc` option:

```
$ ld \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crti.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o main.o lib.o -lc \
-o factorial
```

Finally we get an executable file, but if we try to run it, we will get strange results:

```
$ ./factorial
bash: ./factorial: No such file or directory
```

What's the problem here? Let's look on the executable file with the readelf util:

```
$ readelf -l factorial

Elf file type is EXEC (Executable file)
Entry point 0x4003c0
There are 7 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x0000000000000188 0x0000000000000188  R E    8
  INTERP         0x00000000000001c8 0x00000000004001c8 0x00000000004001c8
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000610 0x0000000000000610  R E    200000
  LOAD           0x0000000000000610 0x0000000000600610 0x0000000000600610
                 0x00000000000001cc 0x00000000000001cc  RW     200000
  DYNAMIC        0x0000000000000610 0x0000000000600610 0x0000000000600610
                 0x0000000000000190 0x0000000000000190  RW     8
  NOTE           0x00000000000001e4 0x00000000004001e4 0x00000000004001e4
                 0x0000000000000020 0x0000000000000020  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .text
   03     .dynamic .got .got.plt .data
   04     .dynamic
   05     .note.ABI-tag
   06
```

Note on the strange line:

```
  INTERP         0x00000000000001c8 0x00000000004001c8 0x00000000004001c8
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

The `.interp` section in the `elf` file holds the path name of a program interpreter or in another words the `.interp` section simply contains an `ascii` string that is the name of the dynamic linker. The dynamic linker is the part of Linux that loads and links shared libraries needed by an executable when it is executed, by copying the content of libraries from disk to RAM. As we can see in the output of the `readelf` command it is placed in the `/lib64/ld-linux-x86-64.so.2` file for the `x86_64` architecture. Now let's add the `-dynamic-linker` option with the path of `ld-linux-x86-64.so.2` to the `ld` call and will see the following results:

```
$ gcc -c main.c lib.c

$ ld \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crti.o \
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crtn.o main.o lib.o \
-dynamic-linker /lib64/ld-linux-x86-64.so.2 \
-lc -o factorial
```

Now we can run it as normal executable file:

```
$ ./factorial

factorial of 5 is: 120
```

It works! With the first line we compile the `main.c` and the `lib.c` source code files to object files. We will get the `main.o` and the `lib.o` after execution of the `gcc` :

```
$ file lib.o main.o
lib.o:  ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
main.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

and after this we link object files of our program with the needed system object files and libraries. We just saw a simple example of how to compile and link a C program with the `gcc` compiler and `GNU ld` linker. In this example we have used a couple command line options of the `GNU linker` , but it supports much more command line options than `-o` , `-dynamic-linker` , etc... Moreover `GNU ld` has its own language that allows to control the linking process. In the next two paragraphs we will look into it.

Useful command line options of the GNU linker

As I already wrote and as you can see in the manual of the `GNU linker` , it has big set of the command line options. We've seen a couple of options in this post: `-o <output>` - that tells `ld` to produce an output file called `output` as the result of linking, `-l<name>` that adds the archive or object file specified by the name, `-dynamic-linker` that specifies the name of the dynamic linker. Of course `ld` supports much more command line options, let's look at some of them.

The first useful command line option is `@file` . In this case the `file` specifies filename where command line options will be read. For example we can create file with the name `linker.ld` , put there our command line arguments from the previous example and execute it with:

```
$ ld @linker.ld
```

The next command line option is `-b` or `--format` . This command line option specifies format of the input object files `ELF` , `DJGPP/COFF` and etc. There is a command line option for the same purpose but for the output file: `--oformat=output-format` .

The next command line option is `--defsym` . Full format of this command line option is the `--defsym=symbol=expression` . It allows to create global symbol in the output file containing the absolute address given by expression. We can find following case where this command line option can be useful: in the Linux kernel source code and more precisely in the Makefile that is related to the kernel decompression for the ARM architecture - arch/arm/boot/compressed/Makefile, we can find following definition:

```
LDFLAGS_vmlinux = --defsym _kernel_bss_size=$(KBSS_SZ)
```

As we already know, it defines the `_kernel_bss_size` symbol with the size of the `.bss` section in the output file. This symbol will be used in the first assembly file that will be executed during kernel decompressing:

```
ldr r5, =_kernel_bss_size
```

The next command line options is the `-shared` that allows us to create shared library. The `-M` or `-map <filename>` command line option prints the linking map with the information about symbols. In our case:

```
$ ld -M @linker.ld
...
...
...
.text           0x00000000004003c0      0x112
 *(.text.unlikely .text.*_unlikely .text.unlikely.*)
 *(.text.exit .text.exit.*)
 *(.text.startup .text.startup.*)
 *(.text.hot .text.hot.*)
 *(.text .stub .text.* .gnu.linkonce.t.*)
 .text          0x00000000004003c0       0x2a /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/crt1.o
...
...
...
 .text          0x00000000004003ea       0x31 main.o
                0x00000000004003ea                main
 .text          0x000000000040041b       0x3f lib.o
                0x000000000040041b                factorial
```

Of course the `GNU linker` support standard command line options: `--help` and `--version` that print common help of the usage of the `ld` and its version. That's all about command line options of the `GNU linker`. Of course it is not the full set of command line options supported by the `ld` util. You can find the complete documentation of the `ld` util in the manual.

Control Language linker

As I wrote previously, `ld` has support for its own language. It accepts Linker Command Language files written in a superset of AT&T's Link Editor Command Language syntax, to provide explicit and total control over the linking process. Let's look on its details.

With the linker language we can control:

- input files;
- output files;
- file formats
- addresses of sections;
- etc...

Commands written in the linker control language are usually placed in a file called linker script. We can pass it to `ld` with the `-T` command line option. The main command in a linker script is the `SECTIONS` command. Each linker script must contain this command and it determines the `map` of the output file. The special variable `.` contains current position of the output. Let's write simple assembly program andi we will look at how we can use a linker script to control linking of this program. We will take a hello world program for this example:

```
section .data
    msg    db "hello, world!",`\n`
section .text
    global    _start
_start:
    mov    rax, 1
    mov    rdi, 1
    mov    rsi, msg
    mov    rdx, 14
    syscall
    mov    rax, 60
    mov    rdi, 0
    syscall
```

We can compile and link it with the following commands:

```
$ nasm -f elf64 -o hello.o hello.asm
$ ld -o hello hello.o
```

Our program consists from two sections: `.text` contains code of the program and `.data` contains initialized variables. Let's write simple linker script and try to link our `hello.asm` assembly file with it. Our script is:

```
/*
 * Linker script for the factorial
 */
OUTPUT(hello)
OUTPUT_FORMAT("elf64-x86-64")
INPUT(hello.o)

SECTIONS
{
    . = 0x200000;
    .text : {
          *(.text)
    }

    . = 0x400000;
    .data : {
          *(.data)
    }
}
```

On the first three lines you can see a comment written in `C` style. After it the `OUTPUT` and the `OUTPUT_FORMAT` commands specifiy the name of our executable file and its format. The next command, `INPUT`, specfies the input file to the `ld` linker. Then, we can see the main `SECTIONS` command, which, as I already wrote, must be present in every linker script. The `SECTIONS` command represents the set and order of the sections which will be in the output file. At the beginning of the `SECTIONS` command we can see following line `. = 0x200000`. I already wrote above that `.` command points to the current position of the output. This line says that the code should be loaded at address `0x200000` and the line `. = 0x400000` says that data section should be loaded at address `0x400000`. The second line after the `. = 0x200000` defines `.text` as an output section. We can see `*(.text)` expression inside it. The `*` symbol is wildcard that matches any file name. In other words, the `*(.text)` expression says all `.text` input sections in all input files. We can rewrite it as `hello.o(.text)` for our example. After the following location counter `. = 0x400000`, we can see definition of the data section.

We can compile and link it with the:

```
$ nasm  -f elf64 -o hello.o hello.S && ld -T linker.script && ./hello
hello, world!
```

If we will look insidei it with the `objdump` util, we can see that `.text` section starts from the address `0x200000` and the `.data` sections starts from the address `0x400000`:

```
$ objdump -D hello

Disassembly of section .text:

0000000000200000 <_start>:
  200000:    b8 01 00 00 00           mov    $0x1,%eax
  ...

Disassembly of section .data:

0000000000400000 <msg>:
  400000:    68 65 6c 6c 6f           pushq  $0x6f6c6c65
  ...
```

Apart from the commands we have already seen, there are a few others. The first is the `ASSERT(exp, message)` that ensures that given expression is not zero. If it is zero, then exit the linker with an error code and print the given error message. If you've read about Linux kernel booting process in the linux-insides book, you may know that the setup header of the Linux kernel has offset `0x1f1`. In the linker script of the Linux kernel we can find a check for this:

```
. = ASSERT(hdr == 0x1f1, "The setup header has the wrong offset!");
```

The `INCLUDE filename` command allows to include external linker script symbols in the current one. In a linker script we can assign a value to a symbol. `ld` supports a couple of assignment operators:

- symbol = expression ;
- symbol += expression ;
- symbol -= expression ;
- symbol *= expression ;
- symbol /= expression ;
- symbol <<= expression ;
- symbol >>= expression ;
- symbol &= expression ;
- symbol |= expression ;

As you can note all operators are C assignment operators. For example we can use it in our linker script as:

```
START_ADDRESS = 0x200000;
DATA_OFFSET   = 0x200000;

SECTIONS
{
    . = START_ADDRESS;
    .text : {
          *(.text)
    }

    . = START_ADDRESS + DATA_OFFSET;
    .data : {
          *(.data)
    }
}
```

As you already may noted the syntax for expressions in the linker script language is identical to that of C expressions. Besides this the control language of the linking supports following builtin functions:

- `ABSOLUTE` - returns absolute value of the given expression;
- `ADDR` - takes the section and returns its address;
- `ALIGN` - returns the value of the location counter (`.` operator) that aligned by the boundary of the next expression after the given expression;
- `DEFINED` - returns `1` if the given symbol placed in the global symbol table and `0` in other way;
- `MAX` and `MIN` - return maximum and minimum of the two given expressions;
- `NEXT` - returns the next unallocated address that is a multiple of the give expression;
- `SIZEOF` - returns the size in bytes of the given named section.

That's all.

Conclusion

This is the end of the post about linkers. We learned many things about linkers in this post, such as what is a linker and why it is needed, how to use it, etc..

If you have any questions or suggestions, write me an email or ping me on twitter.

Please note that English is not my first language, and I am really sorry for any inconvenience. If you find any mistakes please let me know via email or send a PR.

Links

- Book about Linux kernel internals
- linker
- object files
- glibc
- opcode
- ELF
- GNU linker
- My posts about assembly programming for x86_64
- readelf

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
* Vaibhav Tulsyan
* Brandon Wamboldt
* Maxime Leboeuf
* Maximilien Richer
* marmeladema
* Anisse Astier
* TheCodeArtist
* Ehsun N
* Adam Shannon
* Donny Nadolny
* Ehsun N
* Waqar Ahmed
* Ian Miell