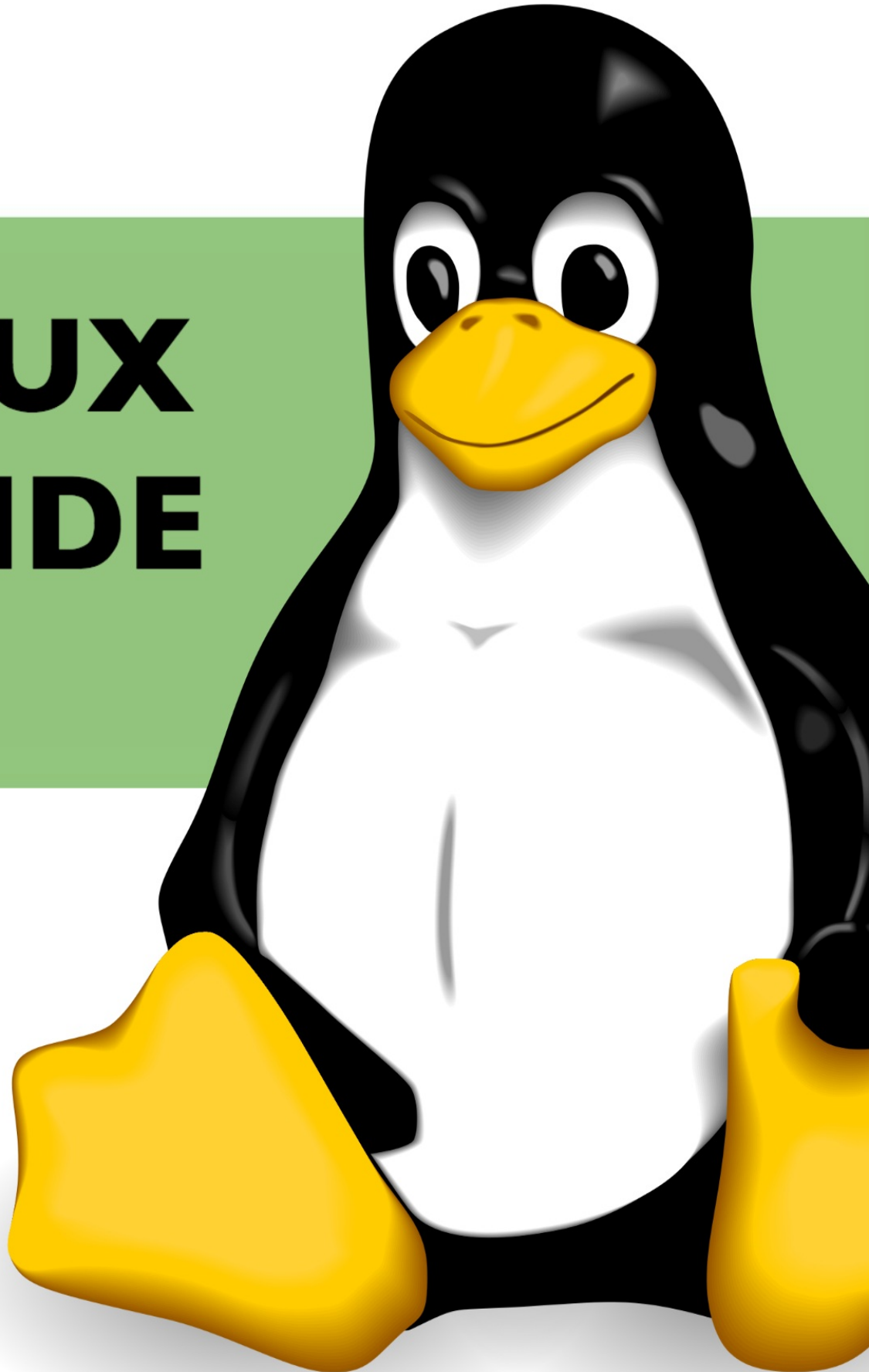


Published  
with GitBook



# LINUX INSIDE

By OxAX



# Table of Contents

Introduction	0
Booting	1
From bootloader to kernel	1.1
First steps in the kernel setup code	1.2
Video mode initialization and transition to protected mode	1.3
Transition to 64-bit mode	1.4
Kernel decompression	1.5
Initialization	2
First steps in the kernel	2.1
Early interrupts handler	2.2
Last preparations before the kernel entry point	2.3
Kernel entry point	2.4
Continue architecture-specific boot-time initializations	2.5
Architecture-specific initializations, again...	2.6
End of the architecture-specific initializations, almost...	2.7
Scheduler initialization	2.8
RCU initialization	2.9
End of initialization	2.10
Interrupts	3
Introduction	3.1
Start to dive into interrupts	3.2
Interrupt handlers	3.3
Initialization of non-early interrupt gates	3.4
Implementation of some exception handlers	3.5
Handling Non-Maskable interrupts	3.6
Dive into external hardware interrupts	3.7
Initialization of external hardware interrupts structures	3.8
Softirq, Tasklets and Workqueues	3.9
Last part	3.10
System calls	4

Introduction to system calls	4.1
How the Linux kernel handles a system call	4.2
vsyscall and vDSO	4.3
How the Linux kernel runs a program	4.4
Timers and time management	5
Introduction	5.1
Clocksource framework	5.2
The tick broadcast framework and dyntick	5.3
Introduction to timers	5.4
Clockevents framework	5.5
x86 related clock sources	5.6
Time related system calls	5.7
Synchronization primitives	6
Introduction to spinlocks	6.1
Memory management	7
Memblock	7.1
Fixmaps and ioremap	7.2
SMP	8
Concepts	9
Per-CPU variables	9.1
Cpumasks	9.2
The initcall mechanism	9.3
Data Structures in the Linux Kernel	10
Doubly linked list	10.1
Radix tree	10.2
Bit arrays	10.3
Theory	11
Paging	11.1
Elf64	11.2
CPUID	11.3
MSR	11.4
Initial ram disk	12
initrd	12.1
Misc	13

---

How the kernel is compiled	13.1
Linkers	13.2
Linux kernel development	13.3
Write and Submit your first Linux kernel Patch	13.4
Data types in the kernel	13.5
Useful links	14
Contributors	15

# linux-insides

An book-in-progress about the linux kernel and its insides.

**The goal is simple** - to share my modest knowledge about the insides of the linux kernel and help people who are interested in linux kernel insides, 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:



## On other languages

- [Chinese](#)
- [Spanish](#)

## LICENSE

Licensed [BY-NC-SA Creative Commons](#).

## Contributions

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

Please read [CONTRIBUTING.md](#) before pushing any changes.



# 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 running 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, query 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 details of transition.
- [Kernel Decompression](#) - describes preparation before kernel decompression and details of direct decompression.

# Kernel booting process. Part 1.

## From the bootloader to the 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 at a 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 start to learn some tools, I will try to explain some parts during this and the following posts. Ok, simple introduction finishes 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 the Linux kernel, we will not start from the kernel code (at least not in this paragraph). Ok, you press the magic power button on your laptop or desktop computer and it starts 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 the motherboard receives the [power good signal](#), it tries to start the CPU. The CPU resets all leftover data in its registers and sets up predefined values for each of them.



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

```
IP          0xffff0
CS selector 0xf000
CS base     0xfffff0000
```

The processor starts working in **real mode**. Let's back up a little to try and understand memory segmentation in this mode. Real mode is supported on all x86-compatible processors, from the **8086** all the way to the modern Intel 64-bit CPUs. The 8086 processor has a 20-bit address bus, which means that it could work with a 0-0x100000 address space (1 megabyte). But it only has 16-bit registers, and with 16-bit registers the maximum address is  $2^{16} - 1$  or 0xffff (64 kilobytes). **Memory segmentation** is used to make use of all the address space available. All memory is divided into small, fixed-size segments of 65536 bytes, or 64 KB. Since we cannot address memory above 64 KB with 16 bit registers, an alternate method is devised. An address consists of two parts: a segment selector which has an associated base address and an offset from this base address. In real mode, the associated base address of a segment selector is `Segment Selector * 16`. Thus, to get a physical address in memory, we need to multiply the segment selector part by 16 and add the offset part:

```
PhysicalAddress = Segment Selector * 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 largest segment selector and offset: `0xffff:0xffff`, it will be:

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

which is 65520 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 discuss about register values after reset:

The `cs` register consists of two parts: the visible segment selector and the hidden base address. While the base address is normally formed by multiplying the segment selector value by 16, during a hardware reset, the segment selector in the CS register is loaded with

0xf000 and the base address is loaded with 0xffff0000. The processor uses this special base address until CS is changed.

The starting address is formed by adding the base address to the value in the EIP register:

```
>>> 0xffff0000 + 0xfff0
'0xffffffff0'
```

We get 0xffffffff0 which is 4GB - 16 bytes. This point is called the [Reset vector](#). This is the memory location at which the 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 the [coreboot](#) source code, we see:

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

Here we can see the jmp instruction [opcode](#) - 0xe9 and its destination address - `_start - ( . + 2 )`, and we can see that the `reset` section is 16 bytes and starts at 0xffffffff0 :

```
SECTIONS {
    _ROMTOP = 0xffffffff0;
    . = _ROMTOP;
    .reset . : {
        *(.reset)
        . = 15 ;
        BYTE(0x00);
    }
}
```

Now the BIOS starts: after initializing and checking the hardware, it needs to find a bootable device. A boot order is stored in the BIOS configuration, controlling which devices the BIOS attempts to boot from. When attempting to boot from 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 (which is 512 bytes). The final two bytes of the first sector are 0x55 and 0xaa, which signals the BIOS that this 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

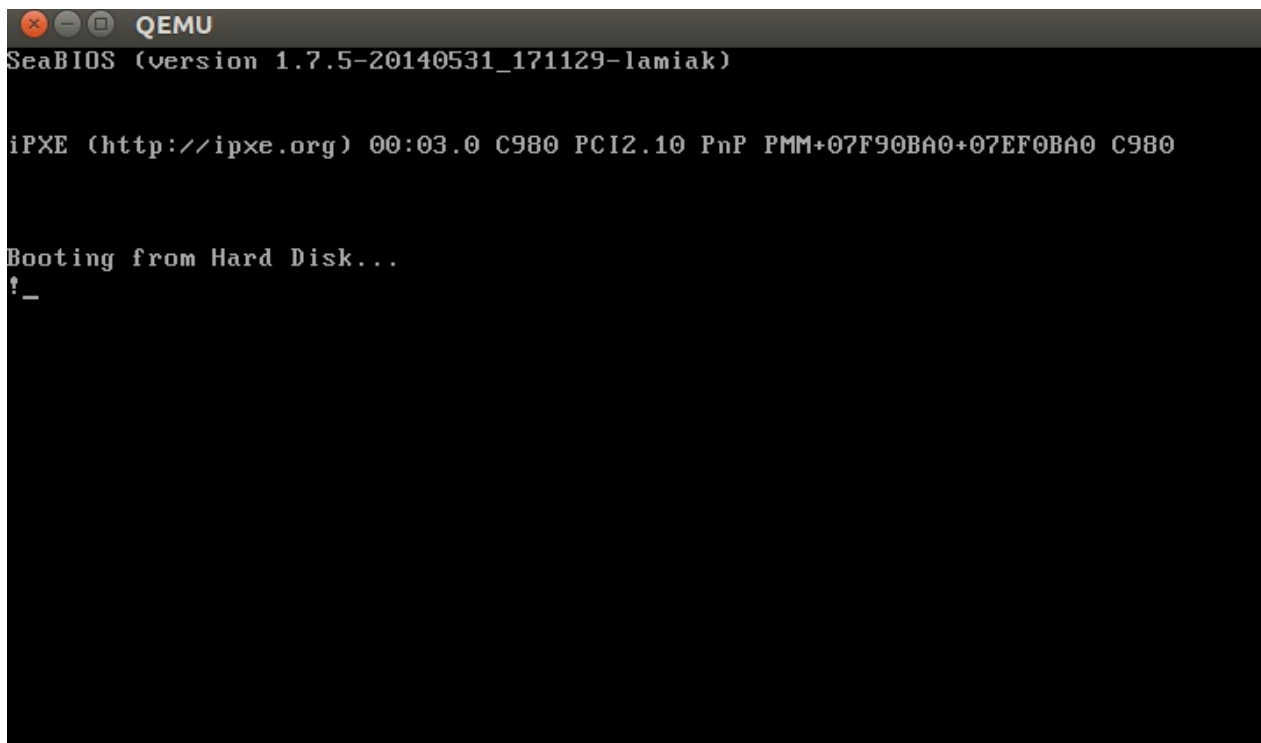
db 0x55
db 0xaa
```

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.

You will see:



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

You can see a binary dump of this with the `objdump` util:

```
nasm -f bin boot.nasm
objdump -D -b binary -mi386 -Maddr16,data16,intel boot
```

A real-world boot sector has code to continue the boot process and the partition table instead of a bunch of 0's and an exclamation mark :) From this point onwards, BIOS hands over control to the bootloader.

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

$$\text{PhysicalAddress} = \text{Segment Selector} * 16 + \text{Offset}$$

The same as mentioned before. We have only 16 bit general purpose registers, the maximum value of a 16 bit register is `0xffff`, so if we take the largest values, the result will be:

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

Where `0x10ffef` is equal to `1MB + 64KB - 16b`. But a **8086** processor, which is the first processor with real mode, has a 20 bit address line and `2^20 = 1048576` is 1MB. This means 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 - 0x00009FFF - Unused
0x0000A000 - 0x0000BFFF - Video RAM (VRAM) Memory
0x0000B000 - 0x0000B777 - Monochrome Video Memory
0x0000B800 - 0x0000BFFF - Color Video Memory
0x0000C000 - 0x0000C7FF - Video ROM BIOS
0x0000C800 - 0x0000EFFF - BIOS Shadow Area
0x0000F000 - 0x0000FFFF - System BIOS
```

In the beginning of this post I wrote that the first instruction executed by the CPU is located at address `0xFFFFFFFF`, which is much larger than `0xFFFF` (1MB). How can the CPU access this in real mode? This is in the [coreboot](#) documentation:

```
0xFFFFE_0000 - 0xFFFF_FFFF: 128 kilobyte ROM mapped into address space
```

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

## Bootloader

There are a number of bootloaders that 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 which is used 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 the console, gets the base address for modules, sets the root device, loads/parses the 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 the last preparation and shows a menu to select an operating system. When we select one of the grub menu entries, `grub_menu_execute_entry` runs, which executes the grub `boot` command, booting the selected operating system.

As we can read in the kernel boot protocol, the bootloader must read and fill some fields of the kernel setup header, which starts at `0x01f1` offset from the kernel setup code. The 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 a description and explanation of all fields of the kernel setup header, we will get back to that when the kernel uses them. You can find a description of all fields in the [boot protocol](#).

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

```

      | 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 when the bootloader transfers control to the kernel, it starts at:

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

where `x` is the address of the kernel boot sector loaded. In my case `x` is `0x10000`, as we can see in a 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 PF d
```

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

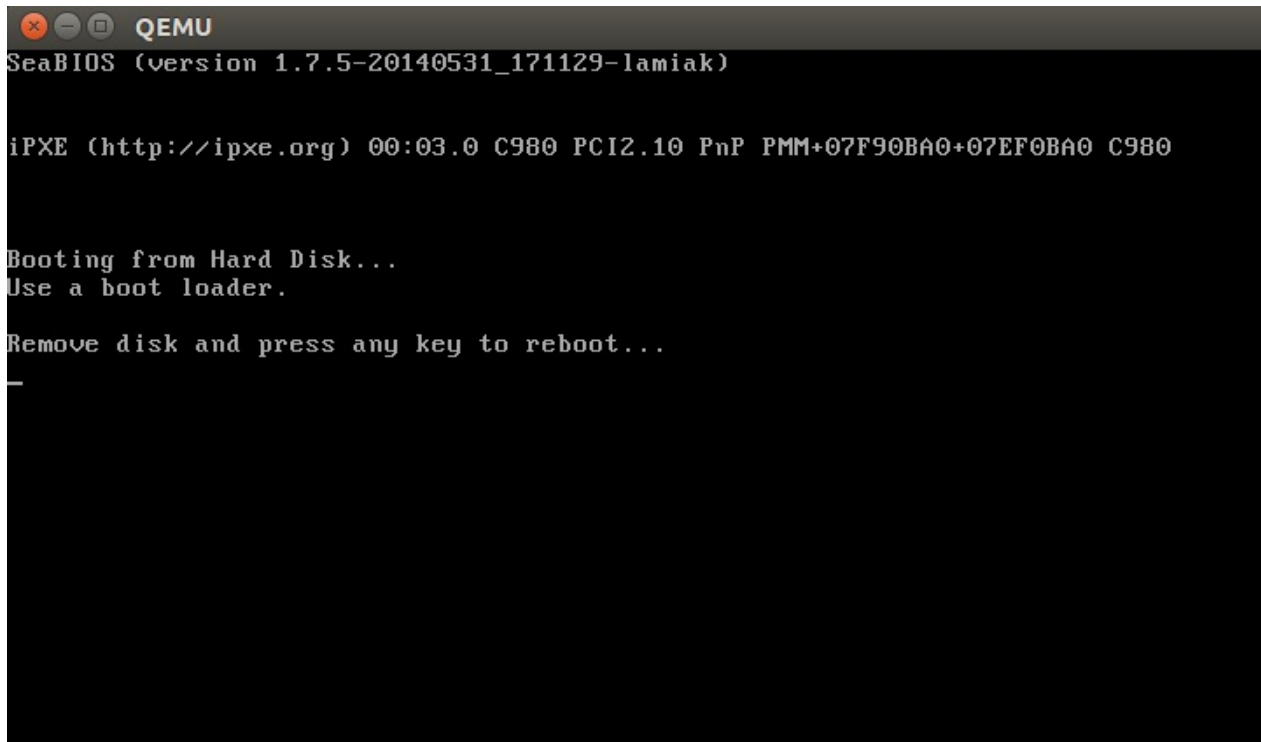
## Start of Kernel Setup

Finally we are in the kernel. Technically the kernel hasn't run yet, we need to set up the kernel, memory manager, process manager etc first. Kernel setup execution starts from [arch/x86/boot/header.S](#) at `_start`. It is a little strange at first sight, as there are several instructions before it.

A Long time ago the 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 to load an operating system with [UEFI](#). We won't see how this works right now, we'll see this in one of the next chapters.

So the actual kernel setup entry point is:

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

The 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 `.btext` section which prints an error message:



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

So the kernel setup entry point is:

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

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

Actually this is the first code that runs (aside from the previous jump instruction of course). After the kernel setup got the control from the bootloader, the first `jmp` instruction is located at `0x200` (first 512 bytes) offset from the start of the kernel real mode. This we can read in the Linux kernel boot protocol and also see in the grub2 source code:

```
segment = grub_linux_real_target >> 4;
state.gs = state.fs = state.es = state.ds = state.ss = segment;
state.cs = segment + 0x20;
```

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

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

In my case when the kernel is loaded at `0x10000`.

After the jump to `start_of_setup`, it needs to do the following:

- Be sure that all values of all segment registers are equal
- Set up correct stack if needed
- Set up [bss](#)

- Jump to C code at [main.c](#)

Let's look at the implementation.

## Segment registers align

First of all it ensures that `ds` and `es` segment registers point to the same address and clears the direction flag with the `cld` instruction:

```
movw    %ds, %ax
movw    %ax, %es
cld
```

As I wrote earlier, grub2 loads kernel setup code at address `0x10000` 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 at 512 bytes offset from the [4d 5a](#). It also needs to align `cs` from `0x10200` to `0x10000` as all other segment registers. After that we set up the stack:

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

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

## Stack Setup

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

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

This can lead to 3 different scenarios:

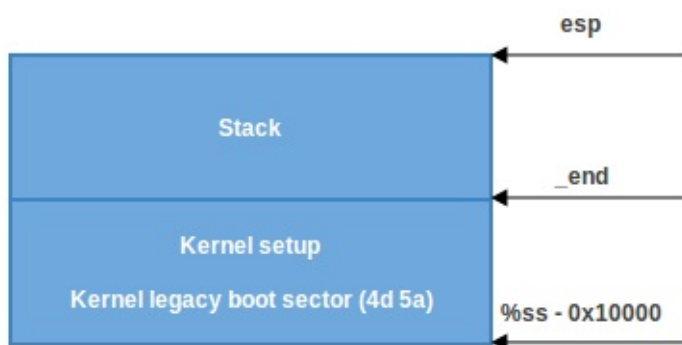
- `ss` has valid value `0x10000` (as all other segment registers beside `cs` )
- `ss` is invalid and `CAN_USE_HEAP` flag is set (see below)
- `ss` is invalid and `CAN_USE_HEAP` flag is not set (see below)

Let's look at all three of these scenarios:

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

```
2:  andw    $~3, %dx
    jnz     3f
    movw    $0xffffc, %dx
3:  movw    %ax, %ss
    movzwl  %dx, %esp
    sti
```

Here we can see the alignment of `dx` (contains `sp` given by bootloader) to 4 bytes and a check for whether or not it is zero. If it is zero, we put `0xffffc` (4 byte aligned address before maximum segment size - 64 KB) in `dx` . If it is not zero we continue to use `sp` given by the bootloader (`0xf7f4` in my case). After this we put the `ax` value to `ss` which stores the correct segment address of `0x10000` and sets up a correct `sp` . We now have a correct stack:



- In the second scenario, ( `ss`  $\neq$  `ds` ). First of all put the `_end` (address of end of setup code) value in `dx` and check the `loadflags` header field with the `testb` instruction to see whether we can use the 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:

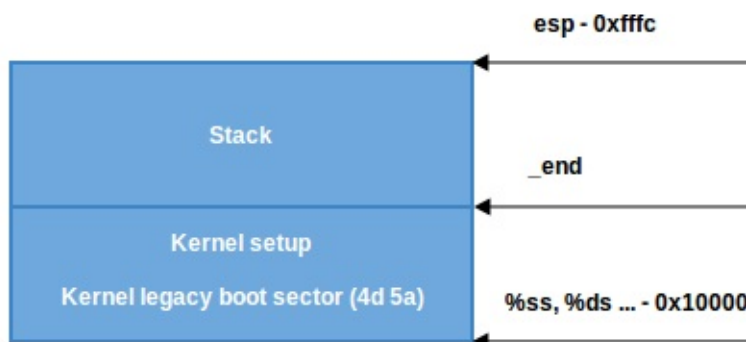
Field name: loadflags

This field is a bitmask.

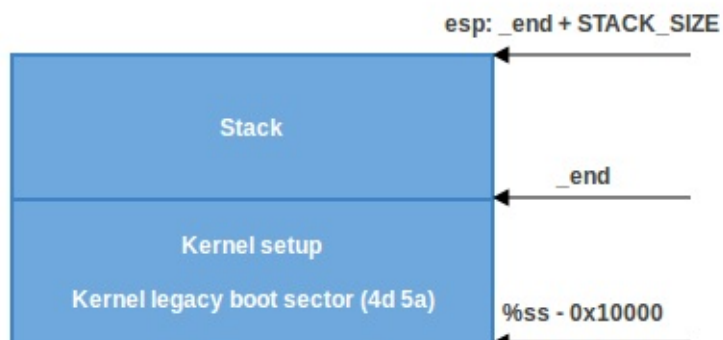
Bit 7 (write): CAN\_USE\_HEAP

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



- When `CAN_USE_HEAP` is not set, we just use a 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 setting up the **BSS** area and checking the "magic" signature. First, signature checking:

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

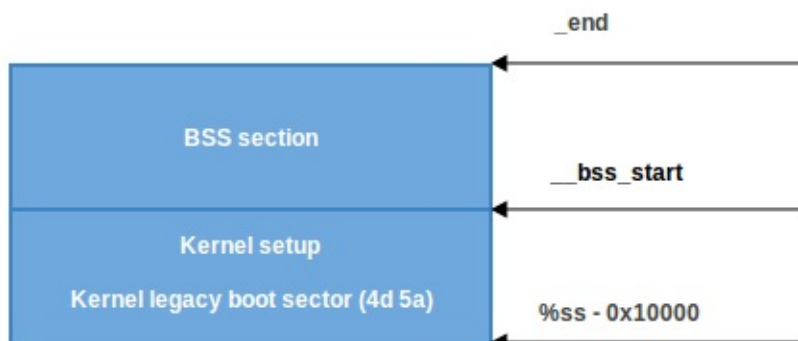
This simply compares the **setup\_sig** with the magic number `0x5a5aaa55`. If they are not equal, a fatal error is reported.

If the magic number matches, knowing we have a set of correct segment registers and a stack, we only need to set up the BSS section before jumping into the C code.

The BSS section is used to store 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 a **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**, 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 and BSS so we can jump to the `main()` C function:

```
calll main
```

The `main()` function is located in [arch/x86/boot/main.c](#). You can read about what this does in the next part.

## Conclusion

This is the end of the first part about Linux kernel insides. 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 find any mistakes please send me PR to [linux-insides](#).**

## 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 insides 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 Selector * 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).

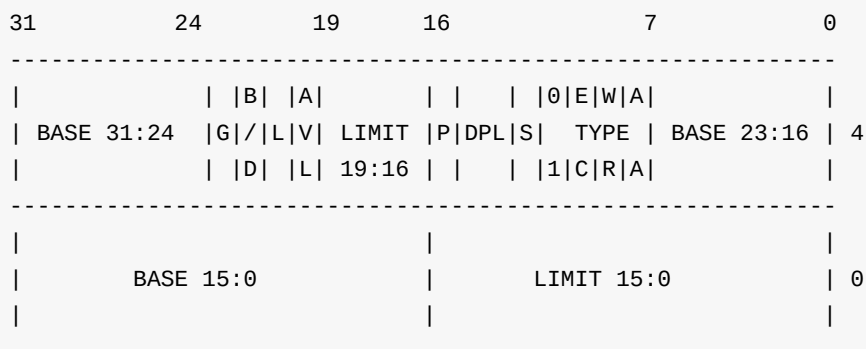
The GDT is a structure which resides in memory. It has no fixed place in the memory so, its address is stored in the special `GDTR` register. Later we will see the GDT loading in the Linux kernel code. There will be an operation for loading it into memory, something like:

```
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:



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 19:16`. 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 0xffff, the size of the segment is 1 Megabyte
- if `G` is 1 and segment limit is 0xffff, 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` (*Expansion Writable Accessible*) or `CRA`(*Conforming Readable Accessible*).

- 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 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 contain segment selectors as in real mode. However, in protected mode, a segment selector is handled differently. Each Segment Descriptor has an associated Segment Selector which is a 16-bit structure:



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.

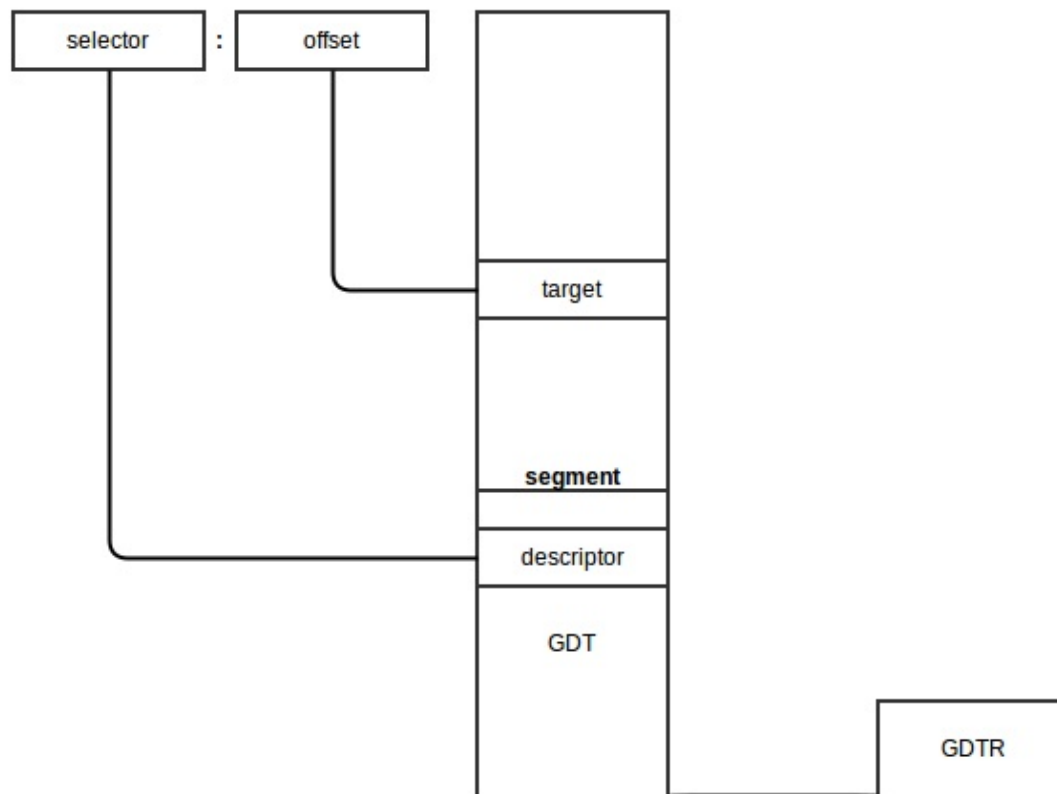
Every segment register has a visible and hidden part.

- 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 the `name` symbol as a function name and ends with the size of the `name` symbol.

Implementation of `memcpy` is easy. At first, it pushes values from the `si` and `di` registers to the stack to preserve their values because they will change during the `memcpy`. `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`
- `dx` will contain the address of `hdr`

- `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 `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 these:

- `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, `putchar` 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 `d1` 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 the `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` ). The last check is 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, &iereg, &oreg);
iereg.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 0x0000000001000000-0x0000000003ffdfbf] usable
[ 0.000000] BIOS-e820: [mem 0x0000000003ffdfbf-0x0000000003ffffff] 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(&iereg);
iereg.ah = 0x02; /* Get keyboard status */
intcall(0x16, &iereg, &oreg);
boot_params.kbd_status = oreg.al;
```

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

```
iereg.ax = 0x0305; /* Set keyboard repeat rate */
intcall(0x16, &iereg, 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](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. The 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 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 check the 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 `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 the 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 the 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 insides. 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 find any mistakes please send me a PR to [linux-insides](#).**

## 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 [main.c](#). In this part, we will see:

- video mode initialization in the kernel setup code,
- preparation before switching into 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 is 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;
```

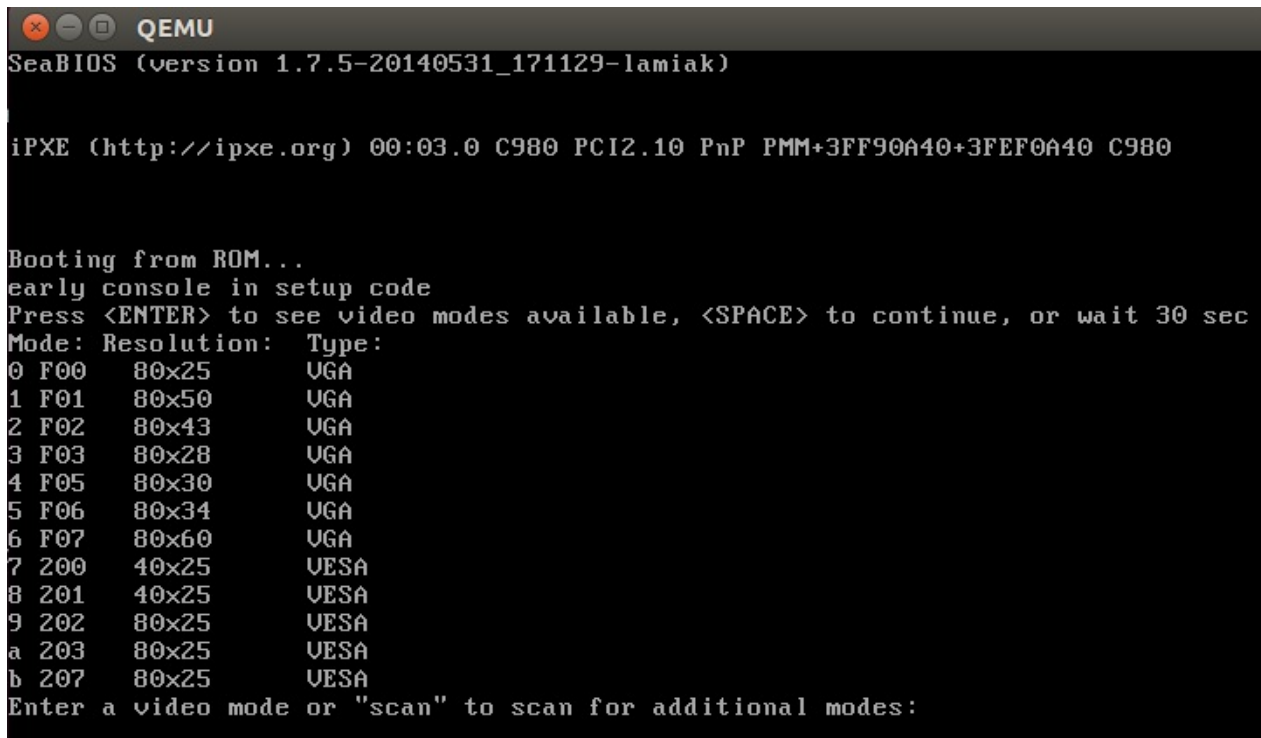
which we filled in the `copy_boot_params` function (you can read about it in the previous post). `vid_mode` is an obligatory field which is filled by the bootloader. You can find information about it in the kernel boot protocol:

Offset /Size	Proto	Name	Meaning
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 mentioned in the description. For example, it can be an integer number `0xFFFFD` or `ask`. If you pass `ask` to `vga`, you will see a menu like this:



```

QEMU
SeaBIOS (version 1.7.5-20140531_171129-lamiak)

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

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

```

which will ask to select a video mode. We will look at its 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 at 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 get `vid_mode` from `boot_params.hdr` in the `set_video` function, we can see the call to the `RESET_HEAP` function. `RESET_HEAP` is a macro which is defined in `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 the `GET_HEAP` macro:

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

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

- size of a type in bytes, which need be allocated
- `__alignof__(type)` shows how variables of this type are aligned
- `n` tells how many items 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 the `a` parameter. After this we save the memory address from `HEAP` to the `tmp` variable, move `HEAP` to the end of the allocated block and return `tmp` which is the 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 a simple API for heap and can setup video mode.

## Set up 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 [include/uapi/linux/screen\\_info.h](#).

If we look at the `store_mode_params` function, we can see that it starts with the call to the `store_cursor_position` function. As you can understand from the function name, it gets information about cursor and stores it.

First of all `store_cursor_position` initializes two variables which have type `biosregs` with `AH = 0x3`, and calls `0x10` BIOS interruption. After the interruption is 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 `boot_params.screen_info` structure.

After `store_cursor_position` is executed, the `store_video_mode` function will be called. It just gets the current video mode and stores it in `boot_params.screen_info.orig_video_mode`.

After this, it checks the 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 the current video mode is MDA, HGC, or VGA in monochrome mode and to `0xB800` if the current video mode is in color mode. After setting up the address of the video segment, font size needs to be stored in `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 in the `FS` register with the `set_fs` function. We already saw functions like `set_fs` in the previous part. They are all defined in [boot.h](#). Next we read the value which is located at address `0x485` (this memory location is used to get the font size) and save the font size in `boot_params.screen_info.orig_video_points`.

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

Next we get the amount of columns by address `0x44a` and rows by address `0x484` and store them in `boot_params.screen_info.orig_video_cols` and `boot_params.screen_info.orig_video_lines`. After this, execution of `store_mode_params` is finished.

Next we can see the `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 [arch/x86/boot/video-mode.c](#). It goes over all `video_cards` and collects the 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` is 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 a 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` executes 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 an infinite loop which tries to set up 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 `video-mode.c` and gets only one parameter, `mode`, which is the number of video modes (we got it from the menu or in the start of `setup_video`, from the kernel setup header).

The `set_mode` function checks the `mode` and calls the `raw_set_mode` function. The `raw_set_mode` calls the `set_mode` function for the 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 the `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 sets up video mode just calls the `0x10` BIOS interrupt with a certain value in the `AH` register.

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

Next `vesa_store_edid` is called. This function simply stores the **EDID** (Extended Display Identification Data) information for kernel use. After this `store_mode_params` is called again. Lastly, if `do_restore` is set, the 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 [main.c](#). As the comment says: `Do the last things and invoke protected mode` , so let's see these last things and switch into protected mode.

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

First is the call to the `realmode_switch_hook` function in `go_to_protected_mode` . This function invokes the real mode switch hook if it is present and disables **NMI**. Hooks are used if the bootloader runs in a hostile environment. You can read more about hooks in the [boot protocol](#) (see **ADVANCED BOOT LOADER HOOKS**).

The `realmode_switch` hook presents a 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 an inline assembly instruction with a `cli` instruction which clears the interrupt flag ( `IF` ). After this, external interrupts are disabled. The next line disables NMI (non-maskable interrupt).

An interrupt is a signal to the CPU which is emitted by hardware or software. After getting the signal, the CPU suspends the current instruction sequence, saves its state and transfers control to the interrupt handler. After the 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 `0x70` (CMOS Address register). After that, a 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 [arch/x86/boot/a20.c](#) and it tries to enable the A20 gate with different methods. The first is the `a20_test_short` function which checks if A20 is already enabled or not with the `a20_test` function:

```
static int a20_test(int loops)
{
    int ok = 0;
    int saved, ctr;

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

    saved = ctr = rdgs32(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` in the `FS` register and `0xffff` in the `GS` register. Next we read the value in address `A20_TEST_ADDR` (it is `0x200`) and put this value into the `saved` variable and `ctr`.

Next we write an updated `ctr` value into `fs:gs` with the `wrfs32` function, then delay for 1ms, and then read the value from the `GS` register by address `A20_TEST_ADDR+0x10`, if it's not zero we already have enabled the 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 the `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, the `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, the `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 the actual transition into protected mode.

## Set up Interrupt Descriptor Table

Now we set up 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 sets up the Interrupt Descriptor Table (describes interrupt handlers and etc.). For now the IDT is not installed (we will see it later), but now we just load IDT with the `lidtl` instruction. `null_idt` contains address and size of IDT, but now they are just zero.

`null_idt` is a `gdt_ptr` structure, it is defined as:

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

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

## Set up Global Descriptor Table

Next is the setup of the Global Descriptor Table (GDT). We can see the `setup_gdt` function which sets up GDT (you can read about it in the [Kernel booting process. Part 2.](#)). There is a definition of the `boot_gdt` array in this function, which contains the definition of the three segments:

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

For code, data and TSS (Task State Segment). We will not use the 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 interested you can find commit which describes it - [here](#)). Let's look at `boot_gdt`. First of all note that it has the `__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 a 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 at the code segment entry. `GDT_ENTRY` takes following values:

- base - 0
- limit - 0xffff
- flags - 0xc09b

What does this mean? The segment's base address is 0, and the limit (size of segment) is - 0xffff (1 MB). Let's look at the 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 the length of the GDT with:

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

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

Next we get a pointer to the GDT with:

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

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

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

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

## Actual transition into protected mode

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

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

which is defined in [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 `arch/x86/boot/pmjump.S`. The first parameter will be in the `eax` register and second is in `edx`.

First of all we put the address of `boot_params` in the `esi` register and the address of code segment register `cs` (`0x1000`) in `bx`. After this we shift `bx` by 4 bits and add the address of label `2` to it (we will have the physical address of label `2` in the `bx` after this) 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 the `PE` (Protection Enable) bit in the `CR0` control register:

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

and make a long jump to protected mode:

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

where

- `0x66` is the operand-size prefix which allows us 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 protected mode. First of all we set up the data segment with:

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

If you paid attention, you can remember that we saved `$_BOOT_DS` in the `cx` register. Now we fill it with 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:

```
jmp     *%eax
```

Remember that `eax` contains the 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. We will see what happens next in the next part.

## Conclusion

This is the end of the third part about linux kernel insides. 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-insides](#).**

## 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

This is the fourth part of the `kernel booting process` where we will see first steps in [protected mode](#), like checking that cpu supports [long mode](#) and [SSE](#), [paging](#), initializes the page tables and at the end we will discuss the transition to [long mode](#).

**NOTE: will be much assembly code in this part, so if you are unfaimilat you might want to consult a a book about it**

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

```
    jmp    *%eax
```

You will recall that `eax` register contains the address of the 32-bit entry point. We can read about this in the [linux kernel x86 boot protocol](#):

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

Let's make sure that it is true by looking at the register values at the 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 will remember from the previous part, this is the second index in the Global Descriptor Table), `eip` register is `0x100000` and base address of all segments including the code segment are zero. So we can get the physical address, it will be `0:0x100000` or just `0x100000`, as specified by the boot protocol. Now let's start with the 32-bit entry point.

## 32-bit entry point

We can find the definition of the 32-bit entry point in the [arch/x86/boot/compressed/head\\_64.S](#) assembly source code file:

```
__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 all of the previous parts. So, the main goal of the `head_64.S` is to prepare for entering long mode, enter into it and then decompress the kernel. We will see all of the steps up to kernel decompression in this part.

There were two files in the `arch/x86/boot/compressed` directory:

- [head\\_32.S](#)
- [head\\_64.S](#)

but we will see only `head_64.S` because as you may remember this book is only `x86_64` related; `head_32.S` was not used in our case. Let's look at [arch/x86/boot/compressed/Makefile](#). There we can see the 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** `$(obj)/head_${BITS}.o`. This means that we will select which file to link based on what `$(BITS)` is set to, either `head_32.o` or `head_64.o`. `$(BITS)` is defined elsewhere in [arch/x86/kernel/Makefile](#) based on the `.config` file:



```
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 needed

As indicated above, we start in the [arch/x86/boot/compressed/head\\_64.S](#) assembly source code file. First we see the definition of the special section attribute before the `startup_32` definition:

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

The `__HEAD` is macro which is defined in [include/linux/init.h](#) header file and expands to the definition of the following section:

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

with `.head.text` name and `ax` flags. In our case, these flags show us that this section is [executable](#) or in other words contains code. We can find definition of this section in the [arch/x86/boot/compressed/vmlinux.lds.S](#) linker script:

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

If you are not familiar with syntax of `GNU LD` linker scripting language, you can find more information in the [documentation](#). In short, the `.` symbol is a special variable of linker - location counter. The value assigned to it is an offset relative to the offset of the segment. In our case we assign zero to location counter. This means that that our code is linked to run from the `0` offset in memory. Moreover, we can find this information in comments:

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

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

In the beginning of the `startup_32` function, we can see the `cld` instruction which clears the `DF` bit in the `flags` register. When direction flag is clear, all string operations like `stos`, `scas` and others will increment the index registers `esi` or `edi`. We need to clear direction flag because later we will use strings operations for clearing space for page tables, etc.

After we have cleared the `DF` bit, next step is the check of the `KEEP_SEGMENTS` flag from `loadflags` kernel setup header field. If you remember we already saw `loadflags` in the very first [part](#) of this book. There we checked `CAN_USE_HEAP` flag to get ability to use heap. Now we need to check the `KEEP_SEGMENTS` flag. This flag is described in the linux [boot protocol](#) documentation:

```
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).
```

So, if the `KEEP_SEGMENTS` bit is not set in the `loadflags`, we need to reset `ds`, `ss` and `es` segment registers to a 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 the `__BOOT_DS` is `0x18` (index of data segment in the [Global Descriptor Table](#)). If `KEEP_SEGMENTS` is set, we jump to the nearest `1f` label or update segment registers with `__BOOT_DS` if it is not set. It is pretty easy, but here is one interesting moment.

If you've read the previous [part](#), you may remember that we already updated these segment registers right after we switched to [protected mode](#) in [arch/x86/boot/pmjump.S](#). So why do we need to care about values of segment registers again? The answer is easy. The Linux kernel also has a 32-bit boot protocol and if a bootloader uses it to load the Linux kernel all code before the `startup_32` will be missed. In this case, the `startup_32` will be first entry point of the Linux kernel right after bootloader and there are no guarantees that segment registers will be in known state.

After we have checked the `KEEP_SEGMENTS` flag and put the correct value to the segment registers, the next step is to calculate difference between where we loaded and compiled to run. Remember that `setup.ld.S` contains following definition: `. = 0` at the start of the `.head.text` section. This means that the code in this section is compiled to run from `0` address. We can see this in `objdump` output:

```
arch/x86/boot/compressed/vmlinux:      file format elf64-x86-64

Disassembly of section .head.text:

0000000000000000 <startup_32>:
   0:   fc                      cld
   1:   f6 86 11 02 00 00 40     testb $0x40,0x211(%rsi)
```

The `objdump` util tells us that the address of the `startup_32` is `0`. But actually it is not so. Our current goal is to know where actually we are. It is pretty simple to do in [long mode](#), because it support `rip` relative addressing, but currently we are in [protected mode](#). We will use common pattern to know the address of the `startup_32`. We need to define a label and make a call to this label and pop the top of the stack to a register:

```
call label
label: pop %reg
```

After this a register will contain the address of a label. Let's look to the similar code which search address of the `startup_32` in the Linux kernel:

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

As you remember from the previous part, the `esi` register contains the address of the [boot\\_params](#) structure which was filled before we moved to the protected mode. The `boot_params` structure contains a special field `scratch` with offset `0x1e4`. These four bytes

field will be temporary stack for `call` instruction. We are getting the address of the `scratch` field + 4 bytes and putting it in the `esp` register. We add 4 bytes to the base of the `BP_scratch` field because, as just described, it will be a temporary stack and the stack grows from top to down in `x86_64` architecture. So our stack pointer will point to the top of the stack. Next we can see the pattern that I've described above. We make a call to the `1f` label and put the address of this label to the `ebp` register, because we have return address on the top of stack after the `call` instruction will be executed. So, for now we have an address of the `1f` label and now it is easy to get address of the `startup_32`. We need just to subtract address of label from the address which we got from the stack:

```
startup_32 (0x0) +-----+ ||||| 1f (0x0 + 1f offset) +-----+
---+ %ebp - real physical address ||| +-----+
```

The `startup_32` is linked to run at `0x0` address and this means that `1f` has `0x0 + offset` to `1f` address. Actually it is something about `0x22` bytes. The `ebp` register contains the real physical address of the `1f` label. So, if we will subtract `1f` from the `ebp` we will get the real physical address of the `startup_32`. The Linux kernel [boot protocol](#) describes that the base of the protected mode kernel is `0x100000`. We can verify this with [gdb](#). Let's start debugger and put breakpoint to the `1f` address which is `0x100022`. If this is correct we will see `0x100022` in the `ebp` register:

```

$ gdb
(gdb)$ target remote :1234
Remote debugging using :1234
0x0000ffff in ?? ()
(gdb)$ br *0x100022
Breakpoint 1 at 0x100022
(gdb)$ c
Continuing.

Breakpoint 1, 0x00100022 in ?? ()
(gdb)$ i r
eax            0x18      0x18
ecx            0x0       0x0
edx            0x0       0x0
ebx            0x0       0x0
esp            0x144a8    0x144a8
ebp            0x100021    0x100021
esi            0x142c0    0x142c0
edi            0x0       0x0
eip            0x100022    0x100022
eflags         0x46      [ PF ZF ]
cs             0x10      0x10
ss             0x18      0x18
ds             0x18      0x18
es             0x18      0x18
fs             0x18      0x18
gs             0x18      0x18

```

If we will execute next instruction which is `subl $1b, %ebp`, we will see:

```

nexti
...
ebp            0x100000    0x100000
...

```

Ok, that's true. The address of the `startup_32` is `0x100000`. After we know the address of the `startup_32` label, we can start to prepare for the transition to [long mode](#). Our next goal is to setup the stack and verify that the CPU supports long mode and [SSE](#).

## Stack setup and CPU verification

We could not setup the stack while we did not know the address of the `startup_32` label. We can imagine the stack as an array and the stack pointer register `esp` must point to the end of this array. Of course we can define an array in our code, but we need to know its actual address to configure stack pointer in a correct way. Let's look at the code:

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

The `boot_stack_end` defined in the same [arch/x86/boot/compressed/head\\_64.S](#) assembly source code file and located in the `.bss` section:

```
.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 the address of `boot_stack_end` into the `eax` register. From now the `eax` register will contain address of the `boot_stack_end` where it was linked or in other words `0x0 + boot_stack_end`. To get the real address of the `boot_stack_end` we need to add the real address of the `startup_32`. As you remember, we have found this address above and put it to the `ebp` register. In the end, the register `eax` will contain real address of the `boot_stack_end` and we just need to put to the stack pointer.

After we have set up the stack, next step is CPU verification. As we are going to execute transition to the `long mode`, we need to check that the CPU supports `long mode` and `SSE`. We will do it by the call of the `verify_cpu` function:

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

This function defined in the [arch/x86/kernel/verify\\_cpu.S](#) assembly file and just contains a couple of calls to the `cpuid` instruction. This instruction is used for getting information about the 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 the value of the `eax` is not zero, we jump to the `no_longmode` label which just stops the CPU by the call of the `hlt` instruction while no hardware interrupt will not happen:

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

If the value of the `eax` register is zero, everything is ok and we are able to continue.

## Calculate relocation address

The next step is calculating relocation address for decompression if needed. First we need to know what it means for a kernel to be `relocatable`. We already know that the base address of the 32-bit entry point of the Linux kernel is `0x100000`. But that is a 32-bit entry point. Default base address of the Linux kernel is determined by the value of the `CONFIG_PHYSICAL_START` kernel configuration option and its default value is - `0x1000000` or `1 MB`. The main problem here is that if the Linux kernel crashes, a kernel developer must have a `rescue kernel` for `kdump` which is configured to load from a different address. The Linux kernel provides special configuration option to solve this problem - `CONFIG_RELOCATABLE`. As we can read in the documentation of the Linux kernel:

```
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 simple terms this means that the Linux kernel with the same configuration can be booted from different addresses. Technically, this is done by the compiling decompressor as [position independent code](#). If we look at [/arch/x86/boot/compressed/Makefile](#), we will see that the decompressor is indeed compiled with the `-fPIC` flag:

```
KBUILD_CFLAGS += -fno-strict-aliasing -fPIC
```

When we are using position-independent code an address obtained by adding the address field of the command and the value of the program counter. We can load a code which is uses such addressing from any address. That's why we had to get the real physical address of `startup_32`. Now let's get back to the Linux kernel code. Our current goal is to calculate an address where we can relocate the kernel for decompression. Calculation of this address depends on `CONFIG_RELOCATABLE` kernel configuration option. Let's look at the 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

```

Remember that value of the `ebp` register is the physical address of the `startup_32` label. If the `CONFIG_RELOCATABLE` kernel configuration option is enabled during kernel configuration, we put this address to the `ebx` register, align it to the `2M` and compare it with the `LOAD_PHYSICAL_ADDR` value. The `LOAD_PHYSICAL_ADDR` macro defined in the [arch/x86/include/asm/boot.h](#) header file and it looks like this:

```

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

```

As we can see it just expands to the aligned `CONFIG_PHYSICAL_ALIGN` value which represents physical address of where to load kernel. After comparison of the `LOAD_PHYSICAL_ADDR` and value of the `ebx` register, we add offset from the `startup_32` where to decompress the compressed kernel image. If the `CONFIG_RELOCATABLE` option is not enabled during kernel configuration, we just put default address where to load kernel and add `z_extract_offset` to it.

After all of these calculations we will have `ebp` which contains the address where we loaded it and `ebx` set to the address of where kernel will be moved after decompression.

## Preparation before entering long mode

When we have the base address where we will relocate compressed kernel image we need to do the last preparation before we can transition to 64-bit mode. First we need to update the [Global Descriptor Table](#) for this:

```

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

```



Here we put the base address from `ebp` register with `gdt` offset into the `eax` register. Next we put this address into `ebp` register with offset `gdt+2` and load the `Global Descriptor Table` with the `lgdt` instruction. To understand the magic with `gdt` offsets we need to look at the definition of the `Global Descriptor Table`. We can find its definition in the same source code [file](#):

```
.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 */
gdt_end:
```

We can see that it is located in the `.data` section and contains five descriptors: `null` descriptor, kernel code segment, kernel data segment and two task descriptors. We already loaded the `Global Descriptor Table` in the previous [part](#), and now we're doing almost the same here, but descriptors with `CS.L = 1` and `CS.D = 0` for execution in `64` bit mode. As we can see, the definition of the `gdt` starts from two bytes: `gdt_end - gdt` which represents last byte in the `gdt` table or table limit. The next four bytes contains base address of the `gdt`. Remember that the `Global Descriptor Table` is stored in the `48-bits GDTR` which consists of two parts:

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

So, we put address of the `gdt` to the `eax` register and then we put it to the `.long gdt` or `gdt+2` in our assembly code. From now we have formed structure for the `GDTR` register and can load the `Global Descriptor Table` with the `lgdt` instruction.

After we have loaded the `Global Descriptor Table`, we must enable [PAE](#) mode by putting the value of the `cr4` register into `eax`, setting 5 bit in it and loading it again into `cr4`:

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

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

# Long mode

[Long mode](#) is the native mode for [x86\\_64](#) processors. First let's look at some differences between `x86_64` and the `x86`.

The `64-bit` mode provides features such 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 an example if it in the next parts).

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

- 64-bit mode;
- compatibility mode.

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

- To enable [PAE](#);
- To build page tables and load the address of the top level page table into the `cr3` register;
- To enable `EFER.LME` ;
- To enable paging.

We already enabled `PAE` by setting the `PAE` bit in the `cr4` control register. Our next goal is to build structure for [paging](#). We will see this in next paragraph.

## Early page tables initialization

So, we already know that before we can move into `64-bit` mode, we need to build page tables, so, let's look at the 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 of this part**

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

- One `PML4` or `Page Map Level 4` table with one entry;
- One `PDP` or `Page Directory Pointer` table with four entries;
- Four `Page Directory` tables with `2048` entries.

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

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

We put the address of the `pgtable` relative to `ebx` (remember that `ebx` contains the address to relocate the kernel for decompression) to the `edi` register, clear `eax` register and 6144 to the `ecx` register. The `rep stosl` instruction will write value of the `eax` to the `edi`, increase value of the `edi` register on 4 and decrease value of the `ecx` register on 4. This operation will be repeated while value of the `ecx` register will be greater than zero. That's why we put magic 6144 to the `ecx`.

The `pgtable` is defined in the end of [arch/x86/boot/compressed/head\\_64.S](#) assembly file and looks:

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

As we can see, it is located in the `.pgtable` section and its size is 24 kilobytes.

After we have got buffer for the `pgtable` structure, we can start to build the top level page table - `PML4` - with:

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

Here again, we put the address of the `pgtable` relative to `ebx` or in other words relative to address of the `startup_32` to the `edi` register. Next we put this address with offset 0x1007 in the `eax` register. The 0x1007 is 4096 bytes which is the size of the `PML4` plus 7. The 7 here represents flags of the `PML4` entry. In our case, these flags are `PRESENT+RW+USER`. In the end we just write first the address of the first `PDP` entry to the `PML4`.

In the next step we will build four `Page Directory` entries in the `Page Directory Pointer` table with the same `PRESENT+RW+USE` flags:

```

    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 the base address of the page directory pointer which is `4096` or `0x1000` offset from the `pgtable` table in `edi` and the address of the first page directory pointer entry in `eax` register. Put `4` in the `ecx` register, it will be a 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 the address of the first page directory pointer entry with flags `0x7`. Next we just calculate the address of following page directory pointer entries where each entry is `8` bytes, and write their addresses to `eax`. The last step of building paging structure is the building of the `2048` page table entries with `2-MByte` pages:

```

    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 as in the previous example, all entries will be with flags -

`$0x00000183` - `PRESENT + WRITE + MBZ`. In the end we will have `2048` pages with `2-MByte` page or:

```

>>> 2048 * 0x00200000
4294967296

```

`4G` page table. We just finished to build our early page table structure which maps `4` gigabytes of memory and now we can put the address of the high-level page table - `PML4` - in `cr3` control register:

```

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

```

That's all. All preparation are finished and now we can see transition to the long mode.

# Transition to the 64-bit mode

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

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

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

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

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

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

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

and execute:

```
lret
```

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

After all of these steps we're finally in 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 find any mistakes please send me PR to [linux-insides](#).**

## 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 the jump on the 64-bit entry point - `startup_64` which is 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 the 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` now point 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 the 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 the decompressed kernel start address and after this code executes `rbx` register will contain address 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 the 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 pointer and resetting of the flags register:

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

    pushq    $0
    popfq
```

As you can see above, the `rbx` register contains the start address of the kernel decompressor code and we just put this address with `boot_stack_end` offset to the `rsp` register which represents pointer to the top of the stack. After this step, the stack will be correct. You can find definition of the `boot_stack_end` in the end of [arch/x86/boot/compressed/head\\_64.S](#) assembly source code 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 end of the `.bss` section, right before the `.pgtable`. If you will look into [arch/x86/boot/compressed/vmlinux.lds.S](#) linker script, you will find Definition of the `.bss` and `.pgtable` there.

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. Before details, let's look at this assembly 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 preserve the value of `rsi`, because this register now stores a pointer to the `boot_params` which is real mode structure that contains booting related data (you must remember this structure, we filled it in the start of kernel setup). In the end of this code we'll restore the pointer to the `boot_params` into `rsi` again.

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

```
. = 0;
.head.text : {
    _head = . ;
    HEAD_TEXT
    _ehhead = . ;
}
.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)
...
...
...
```

The `.text` section contains decompression code:

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

And `.rodata.compressed` contains the compressed kernel image. So the `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 registers, we put the address of `_bss` to the `rcx` register. As you can see in the `vmlinux.lds.S` linker script, it located in the end of all sections with the setup/kernel code. Now we can start to copy data from the `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 decremented or in other words, we will copy bytes in backwards. In the end we clear `DF` flag with `cld` instruction and restore `boot_params` structure to the `rsi`.

Now we have the address of the `.text` section address after relocation and we can jump to it:

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

## Last preparation before kernel decompression

In the previous paragraph we saw that the `.text` section 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
```

We need to initialize the `.bss` section, because soon we will jump to the `C` code. 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 a pointer to the `boot_params` structure and call `decompress_kernel` from the [arch/x86/boot/compressed/misc.c](https://elixir.bootlin.com/linux/v4.14.0/source/arch/x86/boot/compressed/misc.c) with seven arguments:

- `boot_param` - pointer to the `boot_params` structure which is filled by bootloader or during early kernel initialization;
- `heap` - pointer to the `boot_heap` which represents start address of the early boot heap;
- `input_data` - pointer to the start of the compressed kernel or in other words pointer to the `arch/x86/boot/compressed/vmlinux.bin.bz2` ;
- `input_len` - size of the compressed kernel;
- `output` - start address of the future decompressed kernel;
- `output_len` - size of decompressed kernel;
- `run_size` - amount of space needed to run the kernel including `.bss` and `.brk` sections.

All arguments will be passed through the registers according to [System V Application Binary Interface](https://elixir.bootlin.com/linux/v4.14.0/source/arch/x86/boot/compressed/misc.c). We finished all preparation and now can look on the kernel decompression.

# Kernel decompression

As we saw in previous paragraph, the `decompress_kernel` function is defined in the [arch/x86/boot/compressed/misc.c](#) source code file and takes seven arguments. This function starts with the video/console initialization that we already saw in the previous parts. Again, we need to do this because we don't know, do we started in the [real mode](#) or a bootloader used 32 or 64-bit boot protocols.

After the first initialization steps, 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 the `heap` is the second parameter of the `decompress_kernel` function which we got in the [arch/x86/boot/compressed/head\\_64.S](#):

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

As you saw above, the `boot_heap` is defined as:

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

where the `BOOT_HEAP_SIZE` is macro which expands to the `0x400000` (in a case of `bzip2` kernel and `0x8000` in other cases) value and represents size of the heap.

After heap pointers initialization, the next step is the call of the `choose_kernel_location` function from [arch/x86/boot/compressed/aslr.c](#) source code file. As we can understand from the function name it chooses the memory location where the kernel image will be decompressed. I know, it may look weird, that we need to find or even `choose` location where to decompress the compressed kernel image. But actually the Linux kernel supports [kASLR](#) feature which in simple words allows to decompress the kernel into random address for security reasons. Let's open the [arch/x86/boot/compressed/aslr.c](#) source code file and will look at the `choose_kernel_location` implementation.

At the start `choose_kernel_location` tries to find `kaslr` option in the Linux kernel command line if the `CONFIG_HIBERNATION` is set and `nokaslr` option if this configuration option otherwise:

```

#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 the `CONFIG_HIBERNATION` kernel configuration option is enabled during kernel configuration and if there is no `kaslr` option in the Linux kernel command line, we will see `KASLR disabled by default...` output and will jump to the `out` label:

```

out:
    return (unsigned char *)choice;

```

which just returns the `output` parameter which we passed to the `choose_kernel_location` without any changes. In other case, if the `CONFIG_HIBERNATION` kernel configuration option is disabled and the `nokaslr` option is in the kernel command line we do the same that in previous condition.

For now, let's suppose that kernel was configured with enabled randomization and try to understand what `kaslr` is. 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 the `kaslr` option to the kernel's command line and get a random address for the decompressed kernel (you can read more about aslr [here](#)). So, our current goal is to find random address where we can `safely` to decompress the Linux kernel. I'm not in vain wrote - `safely`. What does it mean in this context? You may remember that besides the code of decompressor and directly the kernel image, there are some unsafe places in memory. For example `initrd` image is in memory too and we must not overlap it by the decompressed kernel.

The next function will help us to find safe place where we can decompress kernel. This function is the - `mem_avoid_init` . It defined in the same source code [file](#) and takes four arguments that we already saw in the `decompress_kernel` function:

- `input_data` - pointer to the start of the compressed kernel or in other words pointer to the `arch/x86/boot/compressed/vmlinux.bin.bz2` ;
- `input_len` - size of the compressed kernel;
- `output` - start address of the future decompressed kernel;
- `output_len` - size of decompressed kernel.

The main point of this function is to fill array of the `mem_vector` structures:

```
#define MEM_AVOID_MAX 5

static struct mem_vector mem_avoid[MEM_AVOID_MAX];
```

where the `mem_vector` structure contains information about unsafe memory regions:

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

The implementation of the `mem_avoid_init` is pretty simple. Let's look on the part of this function:

```
...
...
...
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. The `ext_ramdisk_image` is high 32-bits of the `ramdisk_image` field from the setup header and `ext_ramdisk_size` is high 32-bits of the `ramdisk_size` field from [boot protocol](#):

Offset /Size	Proto	Name	Meaning
...			
...			
...			
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 /Size	Proto	Name	Meaning
...			
...			
...			
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 them 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.

The next step after we collected all unsafe memory regions in the `mem_avoid` array will be searching for 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. Once we have the aligned output address, we go through the memory regions which we got with the help of the BIOS [e820](#) service 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);
}
```

Recall that we collected `e820_entries` in the second part of the [Kernel booting process part 2](#). The `process_e820_entry` function does some checks that an `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 an `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 an address that is bigger than original memory region:

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

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

In the next step we need to get the 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 so that the end of the kernel image will be less than the 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 all unsafe memory regions and check that each region does not overlap unsafe areas 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 the memory region does not overlap unsafe regions we call the `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 is 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 an 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 retrieving the random address execution of the `choose_kernel_location` is finished.

Now let's back to the [misc.c](#). After getting the address for the kernel image, there need to be some checks to be sure that the retrieved random address is correctly aligned and address is not wrong.

After all these checks will see the familiar message:

```
Decompressing Linux...
```

and call the `__decompress` function which will decompress the kernel. The `__decompress` function depends on what decompression algorithm was chosen during kernel compilation:

```
#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 two functions are the `parse_elf` and the `handle_relocations`. The main point of these function is to move the uncompressed kernel image to the correct memory place. The fact is that the decompression will decompress compressed part **in-place** and we still need to move kernel to the correct address. As we already know, the kernel image is **ELF** executable, so the main goal of the `parse_elf` function is to move loadable segments to the correct address. We can see loadable segments in the output of the `readelf` util:

```
readelf -l vmlinux
```

Elf file type is EXEC (Executable file)

Entry point 0x10000000

There are 5 program headers, starting at offset 64

Program Headers:

Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags Align	
LOAD	0x0000000000200000	0xfffffffff81000000	0x0000000001000000	
	0x0000000000893000	0x0000000000893000	R E	200000
LOAD	0x0000000000a93000	0xfffffffff81893000	0x0000000001893000	
	0x000000000016d000	0x000000000016d000	RW	200000
LOAD	0x0000000000c00000	0x0000000000000000	0x0000000001a00000	
	0x0000000000152d8	0x0000000000152d8	RW	200000
LOAD	0x0000000000c16000	0xfffffffff81a16000	0x0000000001a16000	
	0x0000000000138000	0x000000000029b000	RWE	200000

The goal of the `parse_elf` function is to load these segments to the `output` address that we got from the `choose_kernel_location` function. This function starts from the checking of the **ELF** signature:

```
Elf64_Ehdr ehdr;
Elf64_Phdr *phdrs, *phdr;

memcpy(&ehdr, output, sizeof(ehdr));

if (ehdr.e_ident[EI_MAG0] != ELFMAG0 ||
    ehdr.e_ident[EI_MAG1] != ELFMAG1 ||
    ehdr.e_ident[EI_MAG2] != ELFMAG2 ||
    ehdr.e_ident[EI_MAG3] != ELFMAG3) {
    error("Kernel is not a valid ELF file");
    return;
}
```

and if it does not valid it prints error message and halt. If we got a valid **ELF** file, copy go through all program headers from the given **ELF** file and copies all loadable segments with correct address to the output buffer:

```
for (i = 0; i < ehdr.e_phnum; i++) {
    phdr = &phdrs[i];

    switch (phdr->p_type) {
    case PT_LOAD:
#ifdef CONFIG_RELOCATABLE
        dest = output;
        dest += (phdr->p_paddr - LOAD_PHYSICAL_ADDR);
    #else
        dest = (void *) (phdr->p_paddr);
    #endif
        memcpy(dest,
               output + phdr->p_offset,
               phdr->p_filesz);
        break;
    default: /* Ignore other PT_* */ break;
    }
}
```

That's all. From now all loadable segments are in the correct place. The last `handle_relocations` function adjusts addresses in the kernel image and called only if the `KASLR` was enabled during kernel configuration.

After the kernel is relocated we return back from the `decompress_kernel` to the [arch/x86/boot/compressed/head\\_64.S](#). The address of the kernel will be in the `rax` register and we jump to 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 insides.

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

If you 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 find any mistakes please send me PR to [linux-insides](#).**

## 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 step after the kernel has been 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 have missed, 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

The previous [post](#) was a last part of the Linux kernel [booting process](#) chapter and now we are starting to dive into initialization process of the Linux kernel. After the image of the Linux kernel is decompressed and placed in a correct place in memory, it starts to work. All previous parts describe work of the Linux kernel setup code which does preparation before first bytes of the Linux kernel code will be executed. From now we are in the kernel and all parts of this chapter will be devoted to the initialization process of the kernel before it will launch process with `pid 1`. 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 located in the [arch/x86/kernel/head\\_64.S](#) and and will move further and further. 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.

In the last [part](#) of the previous [chapter](#) we stopped at the `jmp` instruction from the [arch/x86/boot/compressed/head\\_64.S](#) assembly source code file:

```
jmp    *%rax
```

At this moment the `rax` register contains address of the Linux kernel entry point which that was obtained as a result of the call of the `decompress_kernel` function from the [arch/x86/boot/compressed/misc.c](#) source code file. So, our last instruction in the kernel setup code is a jump on the kernel entry point. We already know where is defined the entry point of the linux kernel, so we are able to start to learn what does the Linux kernel does after the start.

## First steps in the kernel

Okay, we got address of the decompressed kernel image from the `decompress_kernel` function into `rax` register and just jumped there. As we already know the entry point of the decompressed kernel image starts in the [arch/x86/kernel/head\\_64.S](#) assembly source code file and at the beginning of it, we can see following definitions:

```

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

```

We can see definition of the `startup_64` routine that is defined in the `__HEAD` section, which is just a macro which expands to the definition of executable `.head.text` section:

```
#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

```

Besides the definition of the `.text` section, 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 the `x86_64`. The definition of the `__START_KERNEL` macro is located in the [arch/x86/include/asm/page\\_types.h](#) header file and represented by the sum of the base virtual address of the kernel mapping and physical start:

```

#define __START_KERNEL    (__START_KERNEL_map + __PHYSICAL_START)

#define __PHYSICAL_START  ALIGN(CONFIG_PHYSICAL_START, CONFIG_PHYSICAL_ALIGN)

```

Or in other words:

- Base physical address of the Linux kernel - `0x10000000` ;
- Base virtual address of the Linux kernel - `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
```

Yes, it defined as `0x1000000`, but it may be different, for example if [kASLR](#) is enabled. So our current goal is to calculate delta between `0x1000000` and where we actually loaded. 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 virtual address of the `_text` is `0xffffffff81000000` and the physical address of it is `0x1000000`. The `__START_KERNEL_map` macro expands to the `0xffffffff80000000` address, so at the second line of the assembly code, we will get following expression:

```
rbp = 0x1000000 - (0xffffffff81000000 - 0xffffffff80000000)
```

So, after the calculation, the `rbp` will contain `0` which represents difference between addresses where we actually loaded and where the code was compiled. In our case `zero` means that the Linux kernel was loaded by default address and the [kASLR](#) was disabled.

After we got the address of the `startup_64`, we need to do a check that this address is correctly aligned. We will do it with the following code:

```
testl    $~PMD_PAGE_MASK, %ebp
jnz      bad_address
```

Here we just compare low part of the `rbp` register with the complemented value of the `PMD_PAGE_MASK`. The `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 by the checking of highest `18` bits:

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

The address must 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 start to setup identity paging is to fixup 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)
```

All of `early_level4_pgt` , `level3_kernel_pgt` and other address may be wrong if the `startup_64` is not equal to default `0x1000000` address. The `rbp` register contains the delta address so we add to the certain entries of the `early_level4_pgt` , the `level3_kernel_pgt` and the `level2_fixmap_pgt` . Let's try to understand what these labels mean. First of all let's look at 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 at the `early_level4_pgt`. It starts with the (4096 - 8) bytes of zeros, it means that we don't use first 511 entries. And after this we can see one `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 at `_PAGE_TABLE`, it is just page entry access rights:

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

You can read more about it in the [paging](#) part.

The `level3_kernel_pgt` - stores two entries which map kernel space. At the start of it's definition, we can see that it is filled with zeros `L3_START_KERNEL` or 510 times. Here the `L3_START_KERNEL` is the index in the page upper directory which contains `__START_KERNEL_map` address and it equals 510. After this, we can see the definition of the 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. They represented by the one `level2_fixmap_pgt` entry and 10 megabytes hole for the [vsyscalls](#) mapping. The next `level2_kernel_pgt` calls the `PDMS` macro which creates 512 megabytes from the `__START_KERNEL_map` for kernel `.text` (after these 512 megabytes will be modules memory space).

Now, after we saw definitins of these symbols, let's get back to the code which is described at the beginning of the section. Remember that the `rbp` register contains delta between the address of the `startup_64` symbol which was got during kernel [linking](#) and the actual address. So, for this moment, we just need to add add this delta to the base address of some page table entries, that they'll have correct addresses. In our case these entries are:

```
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)
```

or the last entry of the `early_level4_pgt` which is the `level3_kernel_pgt`, last two entries of the `level3_kernel_pgt` which are the `level2_kernel_pgt` and the `level2_fixmap_pgt` and five hundred and seventh entry of the `level2_fixmap_pgt` which is `level1_fixmap_pgt` page directory.

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[507] -> level1_fixmap_pgt
```

Note that we didn't fixup base address of the `early_level4_pgt` and some of other page table directories, because we will see this during of building/filling of structures for these page tables. As we corrected base addresses of the page tables, we can start to build it.

## Identity mapping setup

Now we can see the set up of identity mapping of early page tables. In Identity Mapped Paging, virtual addresses are mapped to physical addresses that have the same value, 1 : 1. Let's look at it in detail. First of all we get the `rip-relative` address of the `_text` and `early_level4_pgt` and put them into `rdi` and `rbx` registers:

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

After this we store 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` in 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 contains 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` matches the first entry of the `level2_kernel_pgt` which is 512 MB kernel mapping.

## Last preparation before jump at the kernel entry point

After that we jump 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 supports `NX` bit with:

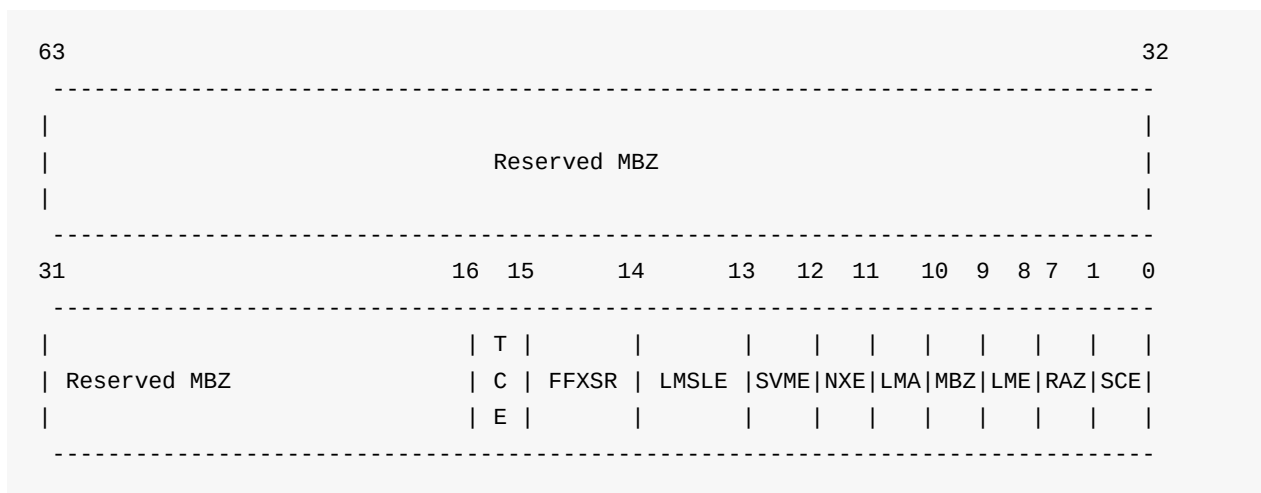
```
movl    $0x80000001, %eax
cpuid
movl    %edx, %edi
```

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:



We will not see all fields in details here, but we will learn about this and other `MSRs` in a special part about it. As we read `EFER` to the `edx:eax`, we check `_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 the `NX` bit is supported we enable `_EFER_NX` and write it too, with the `wrmsr` instruction. After the `NX` bit is set, we set some bits in the `cr0` [control register](#), namely:

- `X86_CR0_PE` - system is in protected mode;
- `X86_CR0_MP` - controls interaction of WAIT/FWAIT instructions with TS flag in CR0;
- `X86_CR0_ET` - on the 386, it allowed to specify whether the external math coprocessor was an 80287 or 80387;
- `X86_CR0_NE` - enable internal x87 floating point error reporting when set, else enables PC style x87 error detection;
- `X86_CR0_WP` - when set, the CPU can't write to read-only pages when privilege level is 0;
- `X86_CR0_AM` - alignment check enabled if AM set, AC flag (in EFLAGS register) set, and privilege level is 3;
- `X86_CR0_PG` - enable paging.

by the execution following assembly code:

```

#define CR0_STATE    (X86_CR0_PE | X86_CR0_MP | X86_CR0_ET | \
                      X86_CR0_NE | X86_CR0_WP | X86_CR0_AM | \
                      X86_CR0_PG)
movl    $CR0_STATE, %eax
movq    %rax, %cr0

```

We already know that to run any code, and even more C code from assembly, we need to setup a stack. As always, we are doing it by the setting of [stack pointer](#) to a correct place in memory and resetting [flags](#) register after this:

```

movq stack_start(%rip), %rsp
pushq $0
popfq

```

The most interesting thing here is the `stack_start`. It defined in the same [source](#) code file and looks like:

```

GLOBAL(stack_start)
.quad  init_thread_union+THREAD_SIZE-8

```

The `GLOBAL` is already familiar to us from. It defined in the [arch/x86/include/asm/linkage.h](#) header file expands to the `global` symbol definition:

```

#define GLOBAL(name) \
    .globl name; \
    name:

```

The `THREAD_SIZE` macro is defined in the [arch/x86/include/asm/page\\_64\\_types.h](#) header file and depends on value of the `KASAN_STACK_ORDER` macro:

```

#define THREAD_SIZE_ORDER    (2 + KASAN_STACK_ORDER)
#define THREAD_SIZE    (PAGE_SIZE << THREAD_SIZE_ORDER)

```

We consider when the [kasan](#) is disabled and the `PAGE_SIZE` is 4096 bytes. So the `THREAD_SIZE` will expands to 16 kilobytes and represents size of the stack of a thread. Why is `thread`? You may already know that each [process](#) may have parent [processes](#) and [child](#) processes. Actually, a parent process and child process differ in stack. A new kernel stack is allocated for a new process. In the Linux kernel this stack is represented by the [union](#) with the `thread_info` structure.

And as we can see the `init_thread_union` is represented by the `thread_union`, which defined as:



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

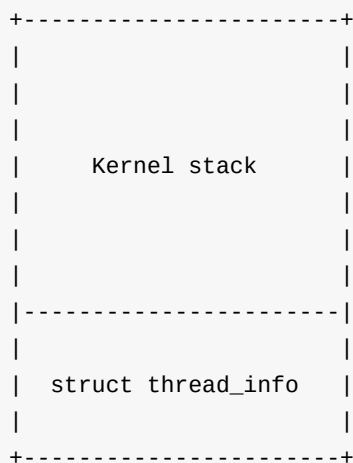
and `init_thread_union` looks like:

```
union thread_union init_thread_union __init_task_data =
    { INIT_THREAD_INFO(init_task) };
```

Where the `INIT_THREAD_INFO` macro takes `task_struct` structure which represents process descriptor in the Linux kernel and does some basic initialization of the given `task_struct` structure:

```
#define INIT_THREAD_INFO(tsk) \
{ \
    .task      = &tsk, \
    .flags     = 0, \
    .cpu       = 0, \
    .addr_limit = KERNEL_DS, \
}
```

So, the `thread_union` contains low-level information about a process and process's stack and placed in the bottom of stack:



Note that we reserve 8 bytes at the top of stack. This is necessary to guarantee illegal access of the next page memory.

After the early boot stack is set, to update the [Global Descriptor Table](#) with `lgdt` instruction:

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

where the `early_gdt_descr` is 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 low userspace addresses, but soon kernel will work in it's own space. Now let's look at the definition of `early_gdt_descr`. `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 at 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` structure which is 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 `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 the `INIT_PER_CPU_VAR` macro will be expanded, we will 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` which represents special stack where [interrupts](#) will be handled on:

```
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. When a [system call](#) or [interrupt](#) occurred, there is no kernel stack at the entry point, so the value of the `MSR_GS_BASE` will store address of the interrupt stack.

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` is defined in the same source code file and looks:

```
.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 is defined in the [arch/x86/kernel/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 - `__START_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 how 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 a lot 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-insides](#).**

## 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. At this moment we are in the decompressed Linux kernel, we have basic [paging](#) structure for early boot and our current goal is to finish early preparation before the main kernel code will start to work.

We already started to do this preparation in the previous [first](#) part of this [chapter](#). 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_handler_array[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

An interrupt is an event caused by software or hardware to the CPU. For example a user have pressed a key on keyboard. On interrupt, CPU stops the current task and transfer control to the special routine which is called - [interrupt handler](#). An interrupt handler handles and interrupt 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
```

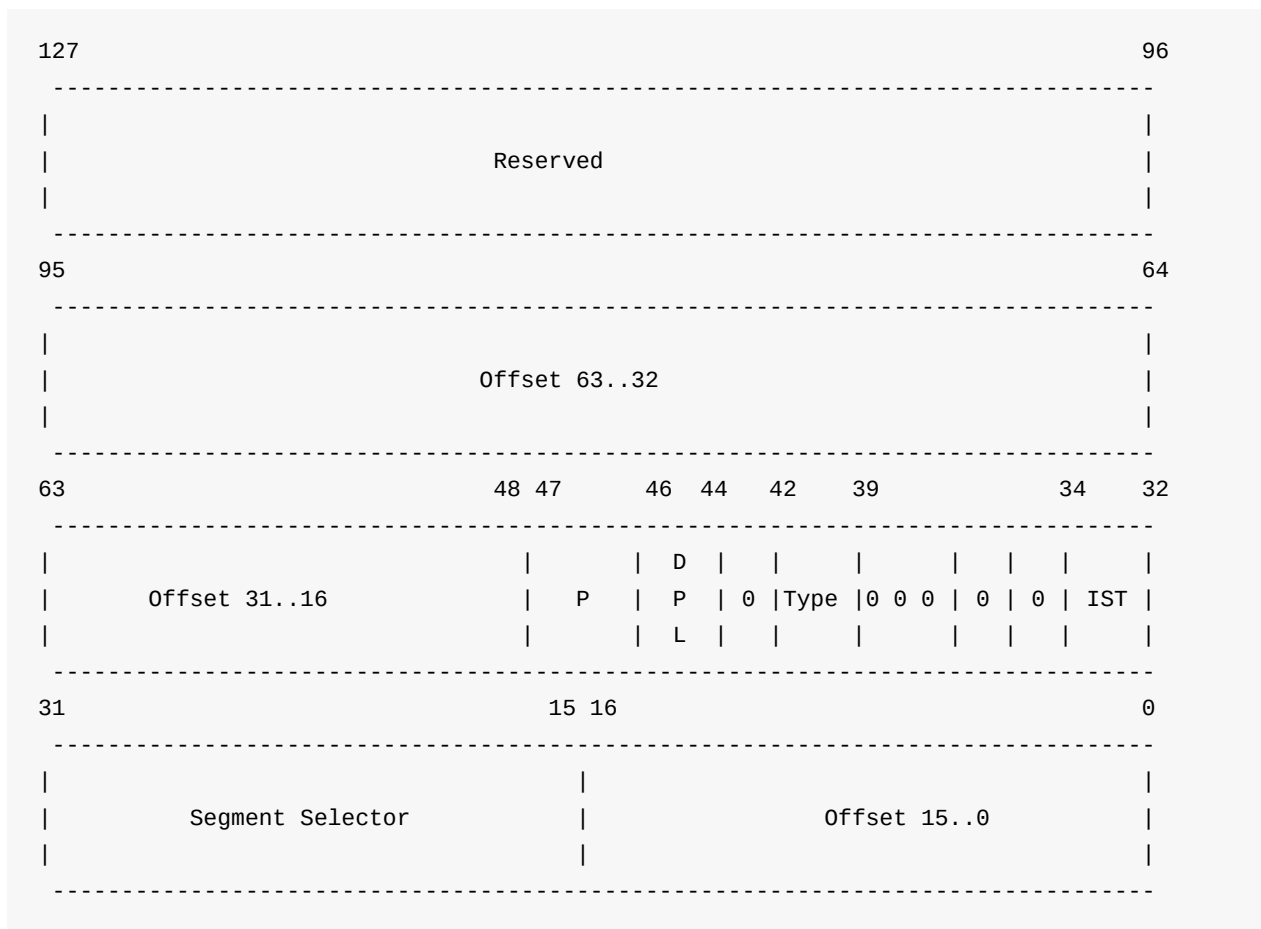
CPU uses vector number as an index in the `Interrupt Descriptor Table` (we will see description of it soon). CPU catch interrupts from the [APIC](#) or through it's pins. Following table shows `0-31` exceptions:



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 instructions which can generate
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` instruction for loading base address of the table into this register.

64-bit mode IDT entry has following structure:



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 in the interrupt gate 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_handler_array[i]);
```

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

- Number of an interrupt or `vector number`;
- Address of the idt handler.

and inserts an interrupt gate to the `IDT` table which is represented by the `&idt_descr` array. First of all let's look on the `early_idt_handler_array` array. It is an array which is defined in the [arch/x86/include/asm/segment.h](#) header file contains addresses of the first 32 exception handlers:

```
#define EARLY_IDT_HANDLER_SIZE    9
#define NUM_EXCEPTION_VECTORS    32

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

The `early_idt_handler_array` is 288 bytes array which contains address of exception entry points every nine bytes. Every nine bytes of this array consist of two bytes optional instruction for pushing dummy error code if an exception does not provide it, two bytes instruction for pushing vector number to the stack and five bytes of `jump` to the common exception handler code.

As we can see, We're filling only first 32 `IDT` entries in the loop, because all of the early setup runs with interrupts disabled, so there is no need to set up interrupt handlers for vectors greater than 32. The `early_idt_handler_array` array contains generic idt handlers and we can find its definition in the [arch/x86/kernel/head\\_64.S](#) assembly file. For now we will skip it, but will look it soon. Before this we will look on the implementation of the `set_intr_gate` macro.

The `set_intr_gate` macro is defined in the [arch/x86/include/asm/desc.h](#) header file and looks:

```
#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 make a call of the `_set_gate` function 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 I 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 macros 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 `Interrupt Descriptor table` with the call of the:

```
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 fill `IDT` gates in the same manner that `_set_gate` but with one difference. These functions use `trace_idt_table` the 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 an 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_handler_array`. We can find it in the [arch/x86/kernel/head\\_64.S](#) assembly file:

```
.globl early_idt_handler_array
early_idt_handlers:
    i = 0
    .rept NUM_EXCEPTION_VECTORS
    .if (EXCEPTION_ERRCODE_MASK >> i) & 1
    pushq $0
    .endif
    pushq $i
    jmp early_idt_handler_common
    i = i + 1
    .fill early_idt_handler_array + i*EARLY_IDT_HANDLER_SIZE - ., 1, 0xcc
    .endr
```

We can see here, interrupt handlers generation for the first 32 exceptions. We check here, if exception has an 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_array` which is generic interrupt handler for now. As we may see above, every nine bytes of the `early_idt_handler_array` array consists from optional push of an error code, push of vector number and jump instruction. We can see it in the output of the `objdump` util:

```
$ objdump -D vmlinux
...
...
...
ffffffff81fe5000 <early_idt_handler_array>:
ffffffff81fe5000:      6a 00                pushq $0x0
ffffffff81fe5002:      6a 00                pushq $0x0
ffffffff81fe5004:      e9 17 01 00 00      jmpq ffffffff81fe5120 <early_idt_handle
ffffffff81fe5009:      6a 00                pushq $0x0
ffffffff81fe500b:      6a 01                pushq $0x1
ffffffff81fe500d:      e9 0e 01 00 00      jmpq ffffffff81fe5120 <early_idt_handle
ffffffff81fe5012:      6a 00                pushq $0x0
ffffffff81fe5014:      6a 02                pushq $0x2
...
...
...
```

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_common` implementation. It locates in the same [arch/x86/kernel/head\\_64.S](#) assembly file and first of all we can see check for `NMI`. We don't need to handle it, so just ignore it in the `early_idt_handler_common` :

```
    cmpl $2, (%rsp)
    je  .Lis_nmi
```

where `is_nmi` :

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

drops an error code and vector number from the stack and call `INTERRUPT_RETURN` which is just expands to the `iretq` instruction. As we checked the vector number and it is not `NMI` , we check `early_recursion_flag` to prevent recursion in the `early_idt_handler_common` 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 of registers when we return from the interrupt handler. After this we check segment selector in the stack:

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

which 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 the code segment was checked, we check the vector number, and if it is `#PF` or [Page Fault](#), 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_INT0(%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];
```



```

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 insides. 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-insides](#).**

## Links

- [GNU assembly .rept](#)
- [APIC](#)
- [NMI](#)
- [Page table](#)
- [Interrupt handler](#)

- [Page Fault,](#)
- [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 call 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 `cmd_line_ptr`, 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 initialized 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 also defined 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's 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 understand 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 a 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 until `ecx` reaches 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 the check for

```
boot_params.hdr.version :
```

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

and if it is zero we call `copy_bootdata` function again 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 the `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` variable 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 variable, 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 initialize `_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

```

`memblock_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 insides. 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 find any mistakes please send me PR to [linux-insides](#).**

## 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` is 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 will we 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 is 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 has 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:

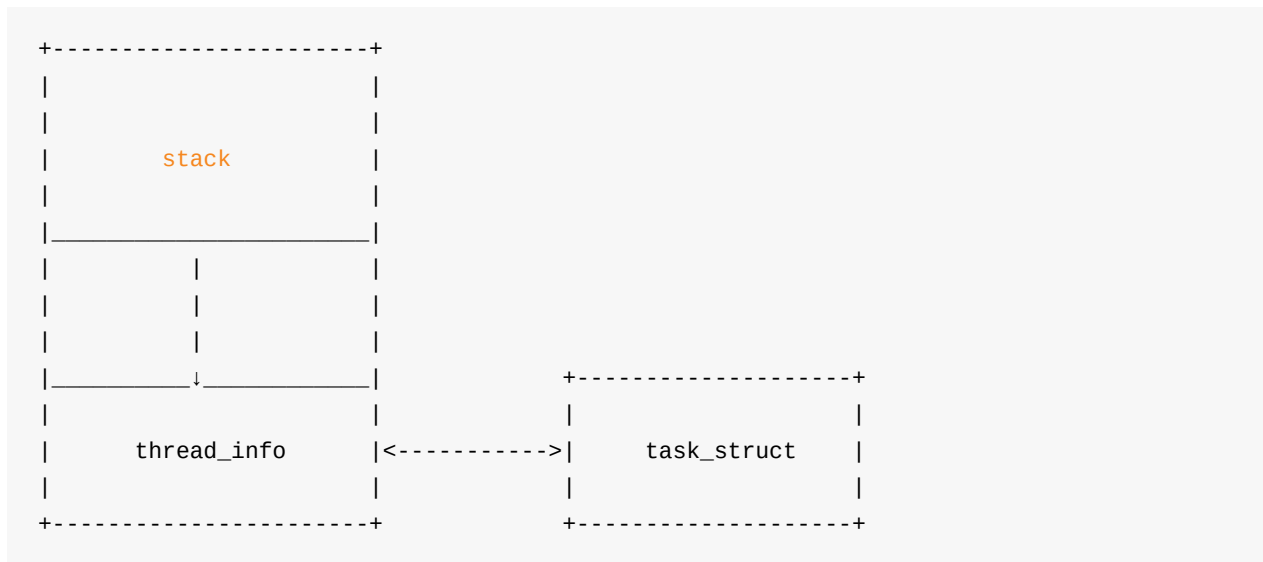
```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

Every process has its own stack and it is 16 kilobytes 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  $16 \text{ killobytes} - 62 \text{ bytes} = 16332 \text{ bytes}$ . 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:



[http://www.quora.com/In-Linux-kernel-Why-thread\\_info-structure-and-the-kernel-stack-of-a-process-binds-in-union-construct](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 above, 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 above, 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`. `__verify_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_cpumask` 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) ) #
```

## 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 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 to [linux-insides](#).**

## 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 now we 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 skip 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`, `INT0` and `INT3` are special instructions which allow a task to explicitly call an interrupt handler. The `INT3` instruction calls the breakpoint ( `#BP` ) handler. You may 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 method 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 three 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 kilobytes. 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 calls `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-electrons.com](http://lxr.free-electrons.com) is a great resource for searching identifiers 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](http://arch/x86/include/asm/traps.h):

```
asm linkage void debug(void);
```

We can see `asm linkage` 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 reason, 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 if interrupt has error code or not and pushes `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 subtract from the stack pointer `$ORIG_RAX-R15` :

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

where `ORIG_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 has 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 `identry` macro for `#DB` interrupt. All interrupts are similar to this implementation and defined with `identry` 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 CPU and its vendor.

## Early ioremap initialization

The next step is initialization of early `ioremap`. In general there are two ways to communicate 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` array 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 `bm_pte` with zeros 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 puzzled, 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 to understand what `old_decode_dev` does. Actually it just calls `MKDEV` inside which generates `dev_t` from the give major and minor numbers. It's implementation is pretty simple:

```
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 a problem: only 256 major numbers and 256 minor numbers are possible. So 16-bit integer was replaced by 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 `0xffff` 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 one of the main purposes of the kernel is resource management. And one of the resource is 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 are 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` :

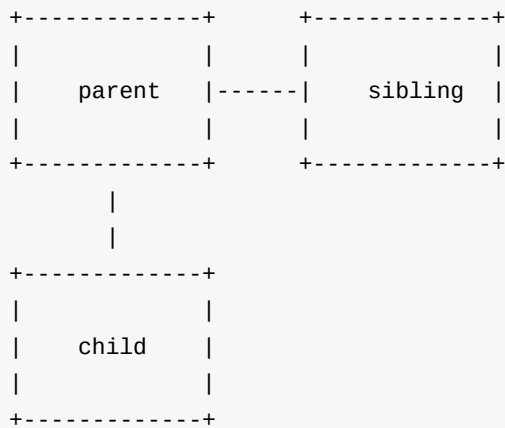
```
cat /proc/iomem
00000000-00000fff : reserved
00001000-0009d7ff : System RAM
0009d800-0009ffff : reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000cffff : Video ROM
000d0000-000d3fff : PCI Bus 0000:00
000d4000-000d7fff : PCI Bus 0000:00
000d8000-000dbfff : PCI Bus 0000:00
000dc000-000dffff : 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](https://elixir.bootlin.com/linux/v4.18.202/source/include/linux/ioport.h)). The last are three pointers to the `resource` structure. These pointers enable a tree-like structure:



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 above 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` is passed to the `EXPORT_SYMBOL` macro. This macro exports the given symbol ( `iomem_resource` in our case) for dynamic linking or in other words it makes a symbol accessible to dynamically loaded modules.

After we set the end address of the root `iomem` resource address range, as I wrote above 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 initialization, pci initialization and etc... initialization 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 `memory_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 are collected, print of all regions with `printk`. You can find this print if you execute `dmesg` command and you can 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 0x0000000001000000-0x000000000be825fff] usable
[ 0.000000] BIOS-e820: [mem 0x000000000be826000-0x000000000be82cfff] ACPI NVS
[ 0.000000] BIOS-e820: [mem 0x000000000be82d000-0x000000000bf744fff] usable
[ 0.000000] BIOS-e820: [mem 0x000000000bf745000-0x000000000bfff4fff] reserved
[ 0.000000] BIOS-e820: [mem 0x000000000bfff5000-0x000000000dc041fff] usable
[ 0.000000] BIOS-e820: [mem 0x000000000dc042000-0x000000000dc0d2fff] reserved
[ 0.000000] BIOS-e820: [mem 0x000000000dc0d3000-0x000000000dc138fff] usable
[ 0.000000] BIOS-e820: [mem 0x000000000dc139000-0x000000000dc27dfff] ACPI NVS
[ 0.000000] BIOS-e820: [mem 0x000000000dc27e000-0x000000000defffff] reserved
[ 0.000000] BIOS-e820: [mem 0x000000000defff000-0x000000000defffff] 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 information 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 its 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 its 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 address space such 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 the 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 we setup memory descriptor of the initial process, next step is initialization of the intel Memory Protection Extensions with `mpx_mm_init`. The next step 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 fill code/data/bss resources with their physical addresses. You can see it in the `/proc/iomem`:

```

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

```

All of these structures are 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 architecture-specific stuff. It was long part, but we have not finished with it. As i already wrote, the `setup_arch` is big function, and I am really not sure that we will cover all 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 a 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 parameters, early dump of the pci devices, direct Media Interface scanning and many many more.

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-insides](#).**

## 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 is defined in the [init/main.c](#) and as you can understand from its name, this function parses kernel command line and setups different services depends on the given parameters (all kernel command line parameters you can find are in the [Documentation/kernel-parameters.txt](#)). You may 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 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 is 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 command line which we just defined and call the `parse_early_options` function from 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 from 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:

```

bootconsole [earlyser0] enabled
NX (Execute Disable) protection: active
SMBIOS 2.8 present.

```

After this we can see call of the:

```

mmblock_x86_reserve_range_setup_data();

```

function. This function is 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 `CONFIG_X86_MPPARSE` configuration options:

```
int __init acpi_mps_check(void)
{
#ifdef 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_MPPARSE` 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` it means that we disable local [APIC](#) and clear `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 definition 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-depended 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 not 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 sanitizes `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 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 calculate 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 than maximum page frame number for the certain architecture ( `x86_64` in our case), print information 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 below 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 memory.

## 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 information. 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 physical 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` :

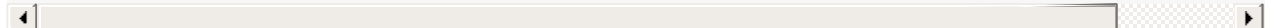
```
void __init dmi_scan_machine(void)
{
    char __iomem *p, *q;
    char buf[32];
    ...
    ...
    ...
    p = dmi_early_remap(0xF0000, 0x10000);
    if (p == NULL)
        goto error;
```

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 functions finish, 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 06/1
```



In the end of the `dmi_scan_machine` , we unmap the previously remapped 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 the `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 found:

```
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 if `mpf->specification` is 1 or 4(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. You can find documentation about this in the - [MultiProcessor Specification](#). You can read More details 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 placed in the `brk` area. Let's look on its implementation:

```
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` is 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
      000000000000b4000 0000000000000000  WA          0      0      4096
[26] .brk          NOBITS          ffffffff81a51000 00d9d000
      00000000000026000 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` area with the `reserve_brk` function:

```
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. Remember 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:

```
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]    [0x0000000000001000-0x000000000009efff], 0x9e000 bytes flags: 0x0
memory[0x1]    [0x00000000100000-0x0000000bffdffff], 0xbfee0000 bytes flags: 0x0
memory[0x2]    [0x0000001000000000-0x00000023fffffff], 0x140000000 bytes flags: 0x0
reserved.cnt = 0x3
reserved[0x0]  [0x0000000009f000-0x000000000ffffff], 0x61000 bytes flags: 0x0
reserved[0x1]  [0x000000001000000-0x00000001a57fff], 0xa58000 bytes flags: 0x0
reserved[0x2]  [0x00000007ec89000-0x00000007fffffff], 0x1377000 bytes flags: 0x0
```

The rest functions after the `memblock_x86_fill` are: `early_reserve_e820_mpc_new` allocates 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, etc.), `trim_platform_memory_ranges` - trims certain memory regions started from `0x20050000` , `0x20110000` , etc. these regions must be excluded because [Sandy Bridge](#) has problems with these regions, `trim_low_memory_range` reserves the first 4 kilobytes 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` initialization, 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 them anyway, because I think that it is not realistic to cover full linux kernel. Anyway we will often return to the idea that we have already seen, and if something is 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 and it was long part, but we are not finished with it. Yes, `setup_arch` is big, hope that next part will be the last part about this function.

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-insides](#).**

## 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 part of the Linux Kernel initialization process which covers insides 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 and its 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 insides, 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 the `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 it 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` 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` is provided by a bootloader. All the same as what we saw 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 than 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 `page frame number`, explanation is simple: First `12` bits of the virtual address represent offset in the physical page or page frame. If we right-shift out `12` bits of the virtual address, 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 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 famous [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 a free area with the `memblock_find_in_range` function:

```

relocated_ramdisk = memblock_find_in_range(0, PFN_PHYS(max_pfn_mapped), area_size, PAGE_S

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 a 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 a 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`, etc.). So we will skip implementation of this for now and will back to it in the part which 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 is defined in the [drivers/base/dma-contiguous.c](#). `DMA` is a special mode when devices communicate 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 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 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 kernel memory manager which used to split memory area into different memory banks in the [NUMA](#) systems. Let's look on the implementation of the `paginig_init` function:



```

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

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

    zone_sizes_init();
}

```

As you can see there is call of the `sparse_memory_present_with_active_regions` function which records a memory area for every `NUMA` node to the array of the `mem_section` structure which contains a pointer to the structure of the array of `struct page`. The next `sparse_init` function allocates non-linear `mem_section` and `mem_map`. In the next step we clear state of the movable memory nodes and initialize sizes of zones. Every `NUMA` node is divided 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_vsyscall` 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`, 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_vsyscall` we can see definition of two variables. The first is extern variable `__vsyscall_page`. As an extern variable, it is 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`, etc.:

```

.global __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`, it 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 greater 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 `ffffffffffff600000` 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](#). We will see more about `vsyscalls` in the `vsyscalls` and `vdso` part.

## Getting the SMP configuration

You may remember how we made a search of the `SMP` configuration in the previous [part](#).

Now we need to get the `SMP` configuration 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 is defined in the [arch/x86/kernel/mpparse.c](#). This function defines a 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 is `acpi_lapic` and `early`. And as we did this checks, we start to read the `SMP` configuration. As we finished reading it, the next step is - `prefill_possible_map` function which makes preliminary filling of the possible CPU's `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 of function of course is important, 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`, 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 we will see all details related with `APIC` in the chapter about interrupts and exceptions handling. In the next step we reserve standard I/O resources like `DMA`, `TIMER`, `FPU`, 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 may remember or saw yourself, `start_kernel` function as big as the `setup_arch`. So the couple of the next part will be dedicated to 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 pass memory descriptor of the init process to the `mm_init_cpumask` and depends on `CONFIG_CPUMASK_OFFSTACK` configuration option we 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` (physical

address of the `(PAGE_OFFSET + 0x10000000)` value) and `BOOTMEM_ALLOC_ACCESSIBLE` (equal to the current value of the `memblock.current_limit`) as minimum address of the memory region and maximum address of the memory region.

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

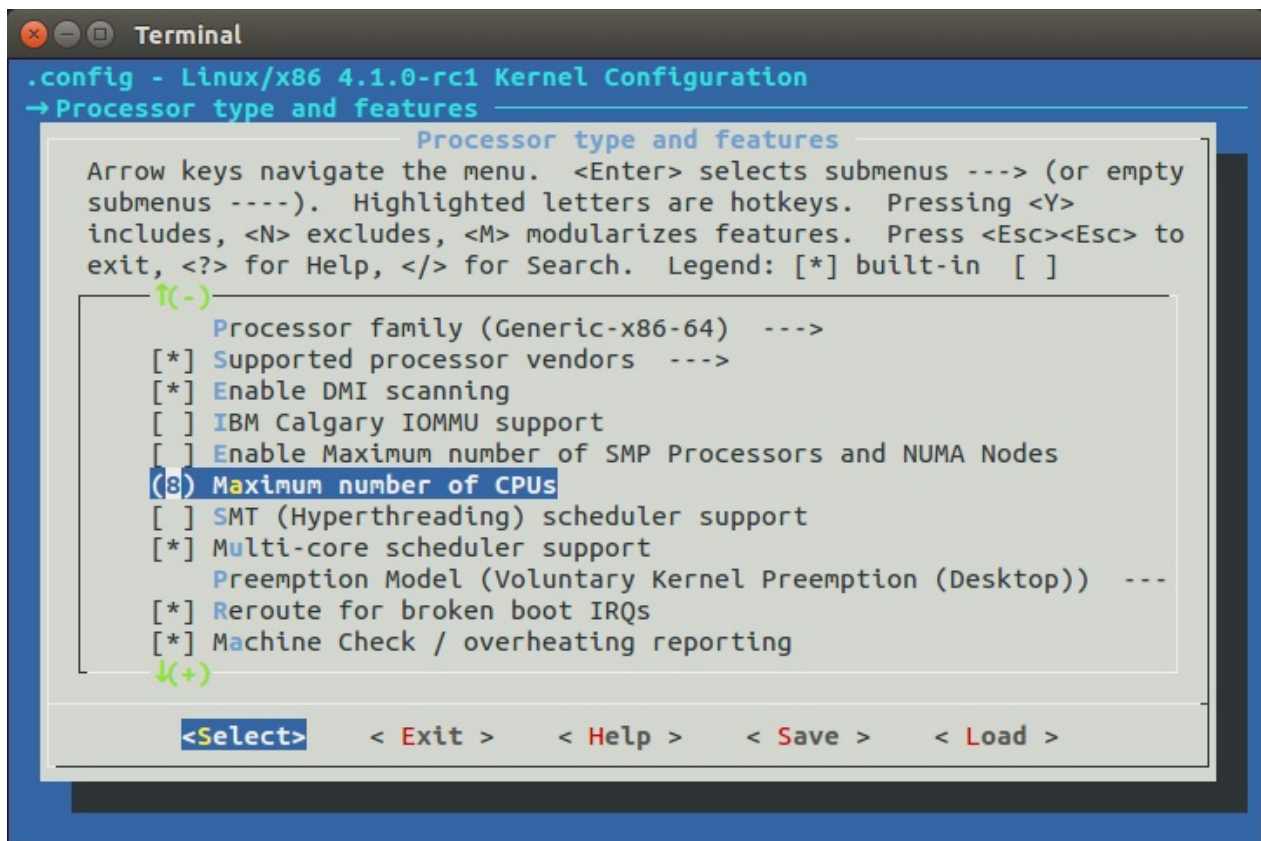
```
static void __init setup_command_line(char *command_line)
{
    saved_command_line =
        memblock_virt_alloc(strlen(boot_command_line) + 1, 0);
    initcall_command_line =
        memblock_virt_alloc(strlen(boot_command_line) + 1, 0);
    static_command_line = memblock_virt_alloc(strlen(command_line) + 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 store `boot_command_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 available 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 finished 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 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-insides](#).**

## 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 is 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 id 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 already 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 may ask the following question: so, if we can access `gdt_page` percpu variable, where it was defined? Actually we already 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, 0xffffffff),
    [GDT_ENTRY_KERNEL_CS]      = GDT_ENTRY_INIT(0xa09b, 0, 0xffffffff),
    [GDT_ENTRY_KERNEL_DS]      = GDT_ENTRY_INIT(0xc093, 0, 0xffffffff),
    [GDT_ENTRY_DEFAULT_USER32_CS] = GDT_ENTRY_INIT(0xc0fb, 0, 0xffffffff),
    [GDT_ENTRY_DEFAULT_USER_DS] = GDT_ENTRY_INIT(0xc0f3, 0, 0xffffffff),
    [GDT_ENTRY_DEFAULT_USER_CS] = GDT_ENTRY_INIT(0xa0fb, 0, 0xffffffff),
    ...
    ...
    ...

```

more about `percpu` variables you can read in the [Per-CPU variables](#) part. As we got address and size of the `GDT` descriptor we 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 `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 following two 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 bootstrap processor bit. Only after the bootstrap processor filled the `cpu_callout_mask` with this secondary processor, it will continue the rest of its initialization. After that the certain processor 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 one of the remaining secondary processors. In a short words it works as i described, but we will see more details 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 soon. For the start let's see how linux kernel considers physical memory. Physical memory is split into banks which are called - `nodes`. If you has no hardware 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` is presented by the `struct pglist_data` in the linux kernel. Each node is divided into a number of special blocks which are called - `zones`. Every zone is 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. We can get information about zones 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 is 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 do a couple of things. The first 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:

```
Linux version 4.1.0-rc2+ (alex@localhost) (gcc version 4.9.2 (Ubuntu 4.9.2-10ubuntu13) ) #493 SMP Thu
Command line: root=/dev/sdb earlyprintk=ttyS0,115200 loglevel=7 debug rdinit=/sbin/init root=/dev/ram
```

And a couple of functions such as `parse_early_param` and `parse_args` which handles linux kernel command line. You may 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 to call 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 each process has assigned a 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 table. 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 not 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` initializes trap handlers (more about last two functions 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 parts 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` which 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` - initializes the `page->ptl` kernel cache, 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 them in the [Linux kernel memory manager](#) chapter.

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

## Scheduler initialization



And now we come to the main purpose of this part - initialization of the task scheduler. I want to say again as I already did it 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 into this function and try to understand how the scheduler is 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 use a simple concept. It models process scheduling as if the system has 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. 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 no task to run besides this task. The `real-time` policies are also supported for the time-critical 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 knowing 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 a structure which is 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 to 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 store information about maximum `deadline` bandwidth 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 `0.95s` by default. The `rt_bandwidth` structure is 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 enable of [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 unfortunately scalability bottlenecks appear as the number of CPUs increase. The concept of root domains was introduced for improving scalability. The linux kernel provides a special mechanism for assigning a set of CPUs and memory nodes to a set of tasks 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` is 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 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 initialize 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 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 runnable 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 arrays are stored in the special structure which name is `runqueue`. As there are no global lock and runqueue, we are going through the all possible CPUs and initialize runqueue for the every cpu. The `runqueue` is presented by the `rq` structure in the linux kernel which is 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 runqueue for the every CPU with the `cpu_rq` macro which returns `runqueues` percpu variable and start to initialize it with runqueue 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 them 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 come 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 have skipped many different details and explanations here, because we need to know and understand how different concepts (like process and process groups, runqueue, rcu, etc.) works in the linux kernel , but we took a short look on the scheduler initialization process. We will look all other details 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 other initialization stuff in the next part.

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-insides](#).**

## 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_disable`. There are two macros:

- `preempt_disable`
- `preempt_enable`

for preemption disabling and enabling. First of all let's try to understand what is `preempt` in the context of an operating system kernel. In 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 don't need to stop it before we call `cpu_idle` function. The `preempt_disable` macro is defined in the [include/linux/preempt.h](#) and depends on the `CONFIG_PREEMPT_COUNT` kernel configuration option. This macro is implemented 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` set 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 `percpu` variables you can read in the part about [Per-CPU variables](#)). Ok, we increased `__preempt_count` and the 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 the opportunity to point compiler and processor on compliance of order. This mechanism is memory barrier. Let's consider a 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 is 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` which will be used by the [slab](#) allocator and `kmem_cache_create` creates it. As you can see 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 to map set of integer IDs to the set of pointers. We can see usage of the integer IDs in the [i2c](#) drivers subsystem. For example [drivers/i2c/i2c-core.c](#) which presents the core of the `i2c` subsystem defines `ID` for the `i2c` adapter with the `DEFINE_IDR` macro:

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

and then 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 its 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 concurrently 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 is 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 get 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` . The moment when every thread is in the `quiescent state` called - `grace period` . If a thread wants to remove an element from the data structure, this occurs in two steps. First step is `removal` - atomically removes element from the data structure, but does not release the physical memory. After this thread-writer announces and waits until it is finished. From this moment, the removed element is available to the thread-readers. After the `grace period` finished, the second step of the element removal will be started, it just removes the element from the physical memory.

There a couple of implementations of the `RCU` . Old `RCU` called classic, the new implemetation called `tree RCU`. As you may already 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`, etc. In the next step, we can see the call of the `rcu_init_geometry` function. This function is defined in the same source code file and computes the node tree geometry depends on the amount of CPUs. Actually RCU provides scalability with extremely low internal RCU lock contention. What if a data structure will be read from the different CPUs? RCU API provides the `rcu_state` structure which presents RCU global state including node hierarchy. Hierarchy is presented by the:

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

array of structures. As we can read in the comment of above definition:

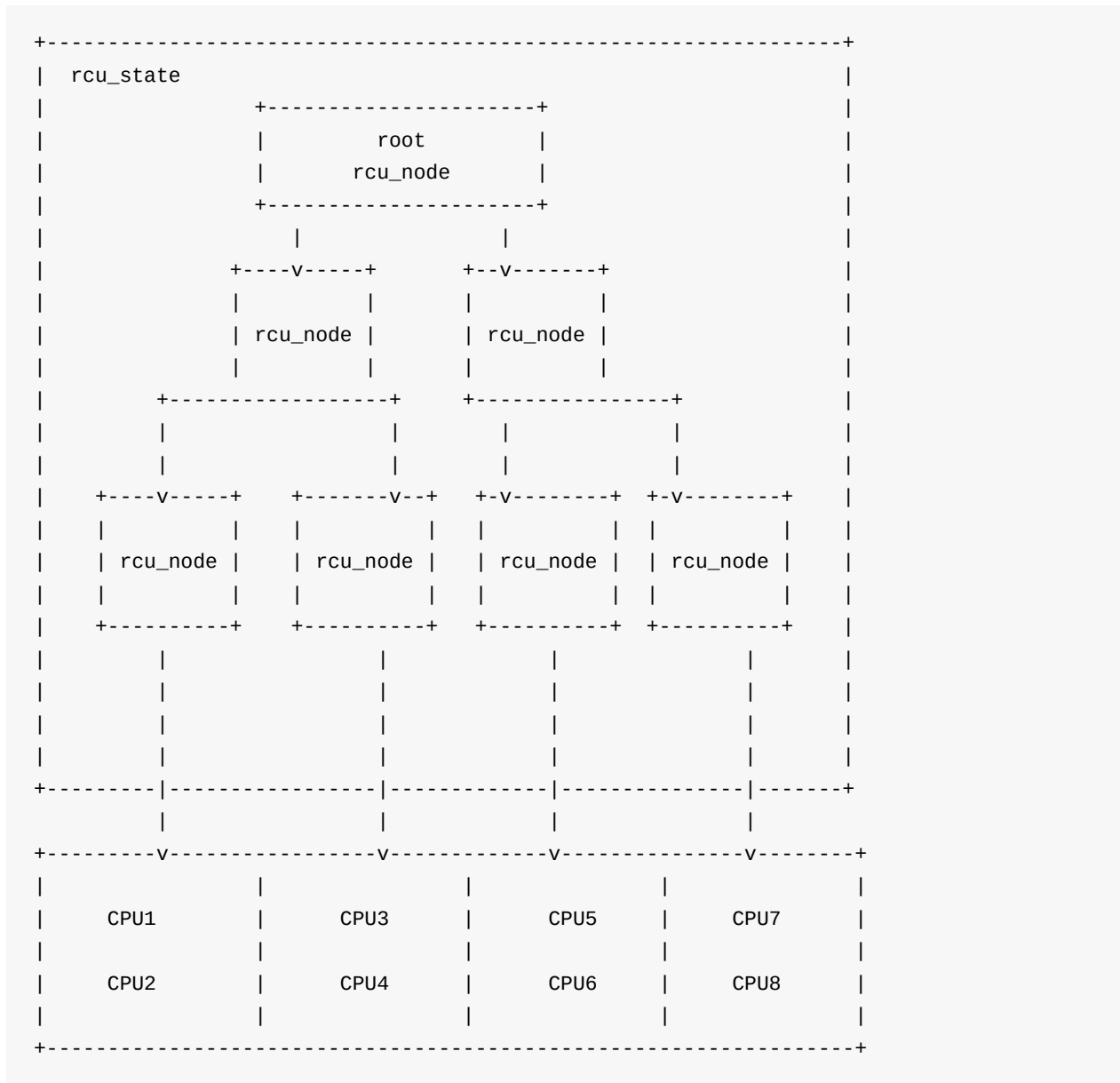
```
The root (first level) of the hierarchy is in ->node[0] (referenced by ->level[0]), the s
level in ->node[1] through ->node[m] (->node[1] referenced by ->level[1]), and the third
in ->node[m+1] and following (->node[m+1] referenced by ->level[2]). The number of level
determined by the number of CPUs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

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

The `rcu_node` structure is 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, etc. Every `rcu_node` contains a lock for a couple of CPUs. These `rcu_node` structures are embedded into a linear array in the `rcu_state` structure and represented as a tree with the root as the first element and 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_
```

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:



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 them 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 change (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 an `rcu_node` tree can handle with the given number of levels:

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

And in the last step we calculate 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 go 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 as 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 can be delayed for later execution when the system is less loaded. Pending interrupts is represented by the following structure:

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

which is defined in the [include/linux/interrupt.h](#) and contains only one field - handler of an interrupt. You can check about `softirqs` in the your system with the:

```
$ cat /proc/softirqs
```

	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU
HI:	2	0	0	1	0	2	
TIMER:	137779	108110	139573	107647	107408	114972	9965
NET_TX:	1127	0	4	0	1	1	
NET_RX:	334	221	132939	3076	451	361	29
BLOCK:	5253	5596	8	779	2016	37442	2
BLOCK_IOPOLL:	0	0	0	0	0	0	
TASKLET:	66	0	2916	113	0	24	2670
SCHED:	102350	75950	91705	75356	75323	82627	6927
HRTIMER:	510	302	368	260	219	255	24
RCU:	81290	68062	82979	69015	68390	69385	6330

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 is 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 systems 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 for different reasons. Some reasons not to dive into details are following:

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

After we initialized `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. You can read more about linux kernel trace system - [here](#).

After the `trace_init` , we can see the call of the `radix_tree_init` . If you are familiar 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 is defined in the [lib/radix-tree.c](#) and you can read more about it 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` , etc.) which are related to different timing and timers stuff. We will see more about these function in the chapter about timers.

The next couple of functions are related with the [perf](#) events - `perf_event_init` (there 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 leak [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 will go through 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 already wrote many times, we will see details of implementations in 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 the start of the first 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 find any mistakes please send me PR to [linux-insides](#).**

## 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 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 is 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/x86/x86\\_64/mm](#):

```
... unused hole ...
ffffff0000000000 - ffffffff7fffffff (=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` for the current CPU. After the `init_espfix_bsp` finished the 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 `THREAD_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 cache for the credentials (like `uid` , `gid` , 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 cache 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:

```

#ifdef CONFIG_ARCH_TASK_STRUCT_ALLOCATOR
#ifdef 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_ALLOCATOR` 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 cache 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 initialize **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 processes that user 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 is 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 initializes 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 is 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 in the previous functions. And calculate 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, etc. More details about [VFS](#) will be described in the separate part. After this we can see `signals_init` function. This function is 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 after been loaded into memory.

## Creation of the root for the procs

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 our system, we call the `proc_self_init` function from the [fs/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` symlink 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/lldisc", 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 skipped them, because they are not 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 different security stuff, `check_bugs` - fix some architecture-dependent bugs, `ftrace_init` function executes initialization of the [ftrace](#), `cgroup_init` makes initialization of the rest of the [cgroup](#) subsystem, etc. Many of these parts and subsystems will be described in the other chapters.

That's all. Finally we have 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 is 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 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 filesystem and it will start to execute `kernel_init` function. A kernel thread differs from a user thread that it runs in 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 `kthreadd`. We already know what is `init` process. Let's look on the `kthreadd`. It is a special kernel thread which manages and helps different parts of the kernel to create another kernel thread. We can see it in the output of the `ps` util:

```
$ ps -ef | grep kthreadd
root      2      0  0 Jan11 ?          00:00:00 [kthreadd]
```

Let's postpone `kernel_init` and `kthreadd` for now and 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 `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 is defined in the [include/linux/completion.h](#) and presents `completions` concept. Completions is a code synchronization mechanism which provides 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 task priority and tasks can be run only when the processor doesn't have anything to run besides this tasks. The second function

`schedule_preempt_disabled` disables preempt in `idle` tasks. And the third function `cpu_startup_entry` is 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 to consume the idle CPU cycles. When there is no 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 but just checks if there is an active task to switch to:

```
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 system is already running, set allowed `cpus/mems` to all CPUs and `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` , initialize `SMP` with the `smp_init` and initialize `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, initialize 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 opening of the `dev/console` and dup twice file descriptors from `0` to `2` after the `do_basic_setup` :

```
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:

```
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`:

```
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

```
system_state = SYSTEM_RUNNING;
```

And tries to run the `init` process:

```
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 is 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 either, 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;
```

Otherwise 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](#). It is not only the `tenth` part, but also 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 skipped details about different subsystem of the kernel, for example I almost did not cover scheduler, interrupts, exception handling, etc. From the next part we will start to dive to the different kernel subsystems. Hope it will be interesting.

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-insides](#).**

## 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 interrupts and 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](#) - describes 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.
- - this is the last part of the interrupts and interrupt handling chapter and here we will see a real hardware driver and interrupts related stuff.

# 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 it 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](#)). The 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
1	Hardware Reset and Machine Checks - RESET - Machine Check
2	Trap on Task Switch - T flag in TSS is set
3	External Hardware Interventions - FLUSH - STOPCLK - SMI - INIT
4	Traps on the Previous Instruction - 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
9	Faults from Decoding the Next Instruction Instruction length > 15 bytes 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));
}
```

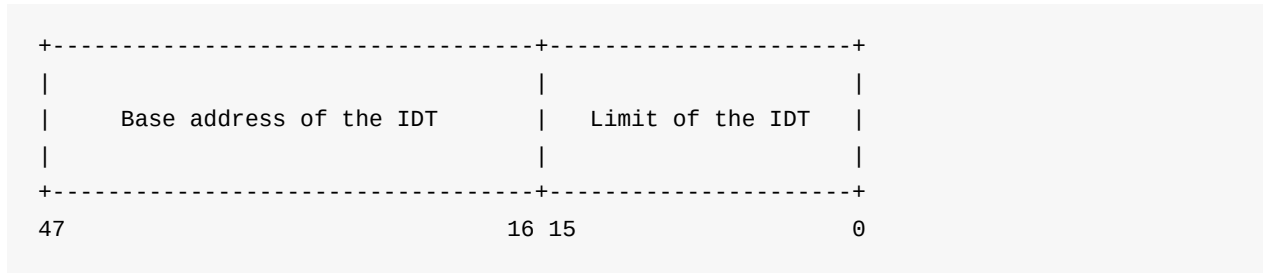
from the [arch/x86/boot/pm.c](#). The `Interrupt Descriptor table` can be located anywhere in the linear address space and the base address of it must be aligned on an 8-byte boundary on `x86` or 16-byte boundary on `x86_64`. The base address of the `IDT` is stored in the



special register - `IDTR` . There are two instructions on `x86` -compatible processors to modify the `IDTR` register:

- `LIDT`
- `SIDT`

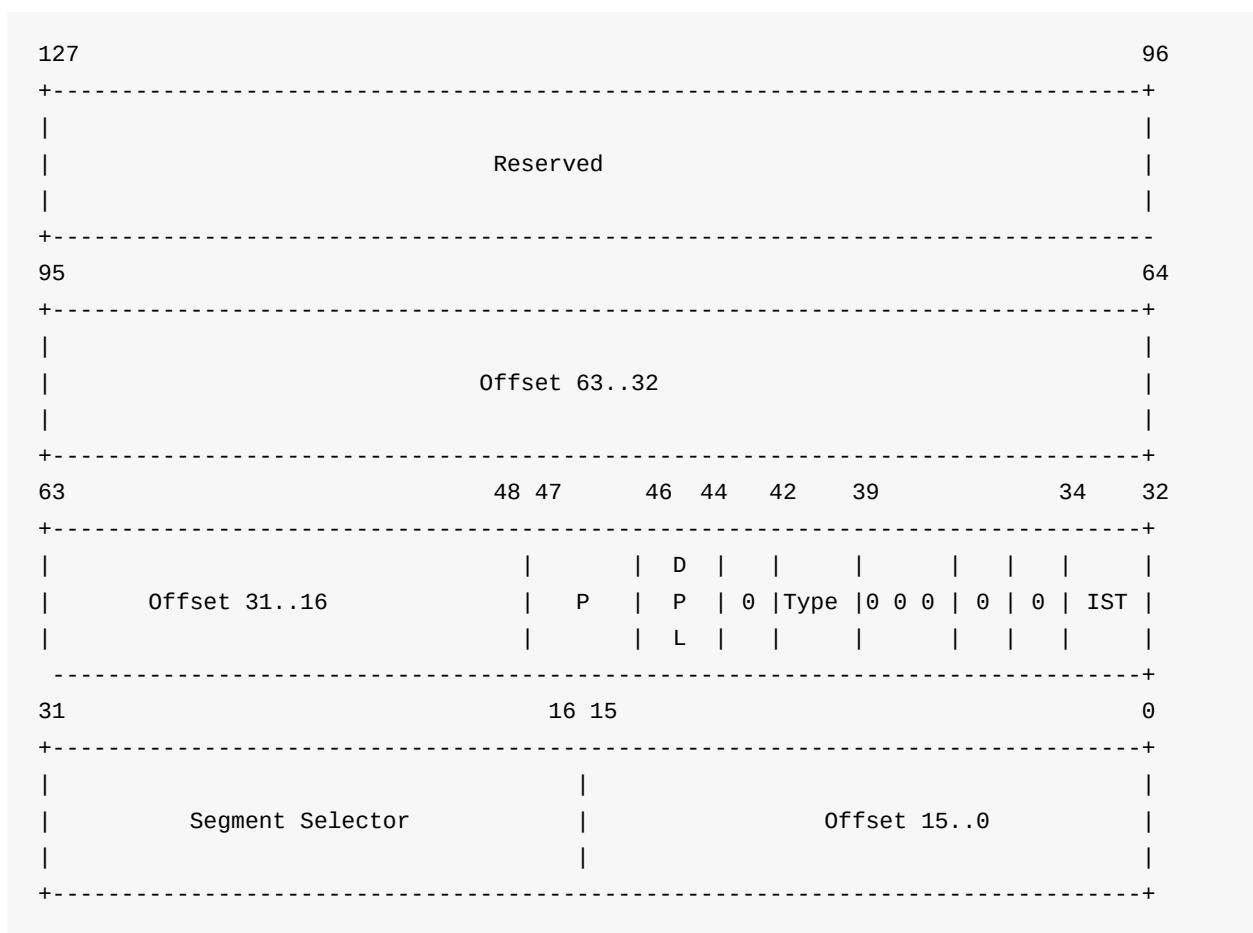
The first instruction `LIDT` is used to load the base-address of the `IDT` i.e. the specified operand into the `IDTR` . The second instruction `SIDT` is used to read and store the contents of the `IDTR` into the specified operand. The `IDTR` register is 48-bits on the `x86` and contains the following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



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:



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 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 is defined as:

```
#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 a 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 the per-cpu area. You already know that [segmented memory model](#) is abolished in the long mode, but we can set the 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 a 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, it's 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 one-by-one and setup `irq_stack_ptr`. This turns out to be equal to the top of the interrupt stack minus `64`. Why `64`? [TODO](#)  
[arch/x86/kernel/cpu/common.c](#) source code file is following:

```
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 the `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 the 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](#)

```
identry 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 `rp1` field is set to the new `cp1`. 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 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 to [linux-insides](#).**

## 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 interrupts and exception 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 in 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. In 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-specific 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 is defined in the same source code file as the `go_to_protected_mode` function and just loads the 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 a special 48-bit `GDTR` register which must contain the 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 the `GDTR` register, but `IDTR` since we set `Interrupt Descriptor Table` . You will not find an `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 differ only by 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 at this point. That's why we just fill the `IDT` with `NULL` .

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](#). You can read more about it in the [part](#) which describes the transition to protected mode.

We already know from the earliest parts that entry to protected mode is 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` is 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 conventions](#):

```
GLOBAL(protected_mode_jump)
...
...
...
.byte    0x66, 0xea        # ljmp1 opcode
2:       .long    in_pm32    # offset
        .word    __BOOT_CS   # segment
...
...
...
ENDPROC(protected_mode_jump)
```

where `in_pm32` contains a jump to the 32-bit entry point:

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

As you can remember the 32-bit entry point is in the [arch/x86/boot/compressed/head\\_64.S](#) assembly file, although it contains `_64` in 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 is the second file in our case. The first file is not even compiled for `x86_64` . Let's look at 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 the transition into [long mode](#) and directly jumps in to it. The `long mode` entry is located in `startup_64` and it makes preparations before the [kernel decompression](#) that occurs in the `decompress_kernel` from the [arch/x86/boot/compressed/misc.c](#). After the kernel is 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 the [NX bit](#), setup the `Extended Feature Enable Register` (see in links), and updated the early `Global Descriptor Table` with 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](#). 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 is declared in the [arch/x86/include/uapi/asm/msr-index.h](#) and looks like:

```
#define MSR_GS_BASE          0xc0000101
```

From this we can understand that `MSR_GS_BASE` defines the 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 a `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 the `init_per_cpu__` prefix with the given symbol. In our case we will get the `init_per_cpu__irq_stack_union` symbol. Let's look at 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 the address of the `init_per_cpu__irq_stack_union` will be `irq_stack_union + __per_cpu_load`. Now we need to understand where `init_per_cpu__irq_stack_union` and `__per_cpu_load` are what they mean. The first `irq_stack_union` is defined in the [arch/x86/include/asm/processor.h](#) with the `DECLARE_INIT_PER_CPU` macro which expands to call 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 expand all macros we will get the same `init_per_cpu__irq_stack_union` as we got after expanding the `INIT_PER_CPU` macro, but you can note that it is not just a symbol, but a variable. Let's look at the `typeof(per_cpu_var(var))` expression. Our `var` is `irq_stack_union` and the `per_cpu_var` macro is 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 are accessing `gs:irq_stack_union` and getting its type which is `irq_union`. Ok, we defined the first variable and know its address, now let's look at the second `__per_cpu_load` symbol. There are a couple of `per-cpu` variables which are located after this symbol. The `__per_cpu_load` is 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 the 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](#):

```
...
...
...
fffffffff819ed000 D __init_begin
fffffffff819ed000 D __per_cpu_load
fffffffff819ed000 A init_per_cpu__irq_stack_union
...
...
...
```

Now we know about `initial_gs` , so let's look at the code:

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

Here we specified a model specific register with `MSR_GS_BASE` , put the 64-bit address of the `initial_gs` to the `edx:eax` pair and execute the `wrmsr` instruction for filling the `gs` register with the base address of the `init_per_cpu_irq_stack_union` which will be at the 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 one of these preparations is filling the early `Interrupt Descriptor Table` with the interrupts handlers 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_array` :

```
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_array` 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 it 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 `union` 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 `softirq` state. In our case `lockdep` subsystem 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` field of the current process and increases 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\\_includes.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 an interrupt or an exception. Besides the `local_irq_disable`, as you already can know there is an inverse macro - `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) && irq_s_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:

```
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:

```
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:

```
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:

```
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 privilege 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 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-insides](#).

## 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](#)
- [thiscpu\\* 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 stopped 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 vector 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 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 specifier of the `gcc`. Actually for a `c` functions which are called from assembly, we need in explicit declaration of the function calling convention. In our case, if function made 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 occurred, 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 transferred to an interrupt handler;
- After an interrupt handler will finish its work and finishes with the `iret` instruction, old `ss` will be popped 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 error 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 occurs, 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 the `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 presents 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 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 switches 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, real 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` will 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 preparations 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 kernel 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` flag 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 disable `irqs`, decrease 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 make almost the same that in the `do_debug` handler. We get the previous state with the `ist_enter`, increase and decrease the

`debug_stack_usage` per-cpu variable, enable and disable local interrupts. But of course there is one difference between these two handlers. We need to lock and then 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 transferred 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 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-insides](#).**

## 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 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 fills [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 subsystem 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 the `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 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 beginning 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 condition:

```
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 `0x00007fffffffff000`. 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 `0x00007fffffffff000` 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 appropriate 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 kernel, 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 handlers 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 represents `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 indicates that a `BOUND-range-exceed` fault occurred 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:

- `#CS0` 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 `Segment 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-protection 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 beginning 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_ucose_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 `Global Descriptor 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 Segment` 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` exception and etc. You can note that we saw just initialization stuff, and will dive into details about handlers for these exceptions. In the next part we will start to do 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 find any mistakes please send me PR to [linux-insides](#).**

## Links

- [page fault](#)
- [Interrupt Descriptor Table](#)
- [Tracing](#)
- [cr2](#)
- [RCU](#)
- [thiscpu\\* operations](#)
- [kmemcheck](#)
- [prefetchw](#)
- [3DNow](#)
- [CPU caches](#)
- [VFS](#)
- [Linux kernel memory management](#)
- [Fix-Mapped Addresses and ioremap](#)
- [Extended Industry Standard Architecture](#)
- [INT instruction](#)
- [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:

<code>idtentry divide_error</code>	<code>do_divide_error</code>	<code>has_error_code</code>
<code>idtentry overflow</code>	<code>do_overflow</code>	<code>has_error_code</code>
<code>idtentry invalid_op</code>	<code>do_invalid_op</code>	<code>has_error_code</code>
<code>idtentry bounds</code>	<code>do_bounds</code>	<code>has_error_code</code>
<code>idtentry device_not_available</code>	<code>do_device_not_available</code>	<code>has_error_code</code>
<code>idtentry coprocessor_segment_overrun</code>	<code>do_coprocessor_segment_overrun</code>	<code>has_error_code=0</code>
<code>idtentry invalid_TSS</code>	<code>do_invalid_TSS</code>	<code>has_error_code=1</code>
<code>idtentry segment_not_present</code>	<code>do_segment_not_present</code>	<code>has_error_code=1</code>
<code>idtentry spurious_interrupt_bug</code>	<code>do_spurious_interrupt_bug</code>	<code>has_error_code</code>
<code>idtentry coprocessor_error</code>	<code>do_coprocessor_error</code>	<code>has_error_code=0</code>
<code>idtentry alignment_check</code>	<code>do_alignment_check</code>	<code>has_error_code=1</code>
<code>idtentry simd_coprocessor_error</code>	<code>do_simd_coprocessor_error</code>	<code>has_error_code</code>

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 third 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_ove
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 source 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 expand 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 `context_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 `atomic_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 `task_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 occurs 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 exception 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 split 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);
normal_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 exception 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` exception 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` flag 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. 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 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-insides](#).**

## 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 ignored 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 completed. 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` caught 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 then 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)
    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 `conditional_sti` function that checks `IF` flag and call the `local_irq_enable` depends on its value:

```
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:

```
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:

```
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`:



```
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 exception 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 finished 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 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-insides](#).**

## Links

- [General Protection Fault](#)
- [opcode](#)

- [Non-Maskable](#)
- [BOUND instruction](#)
- [CPU socket](#)
- [Interrupt Descriptor Table](#)
- [Interrupt Stack Table](#)
- [Paravirtualization](#)
- [.rept](#)
- [SIMD](#)
- [Coprocesor](#)
- [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 finished 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. Unfortunately 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 example 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 occurred:

- 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 initialization 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 files:

- `irq_common_data` - per irq and chip data passed down to chip functions;
- `status_use_accessors` - contains status of the interrupt source which is combination 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 mutiple 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 `CONFIG_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:

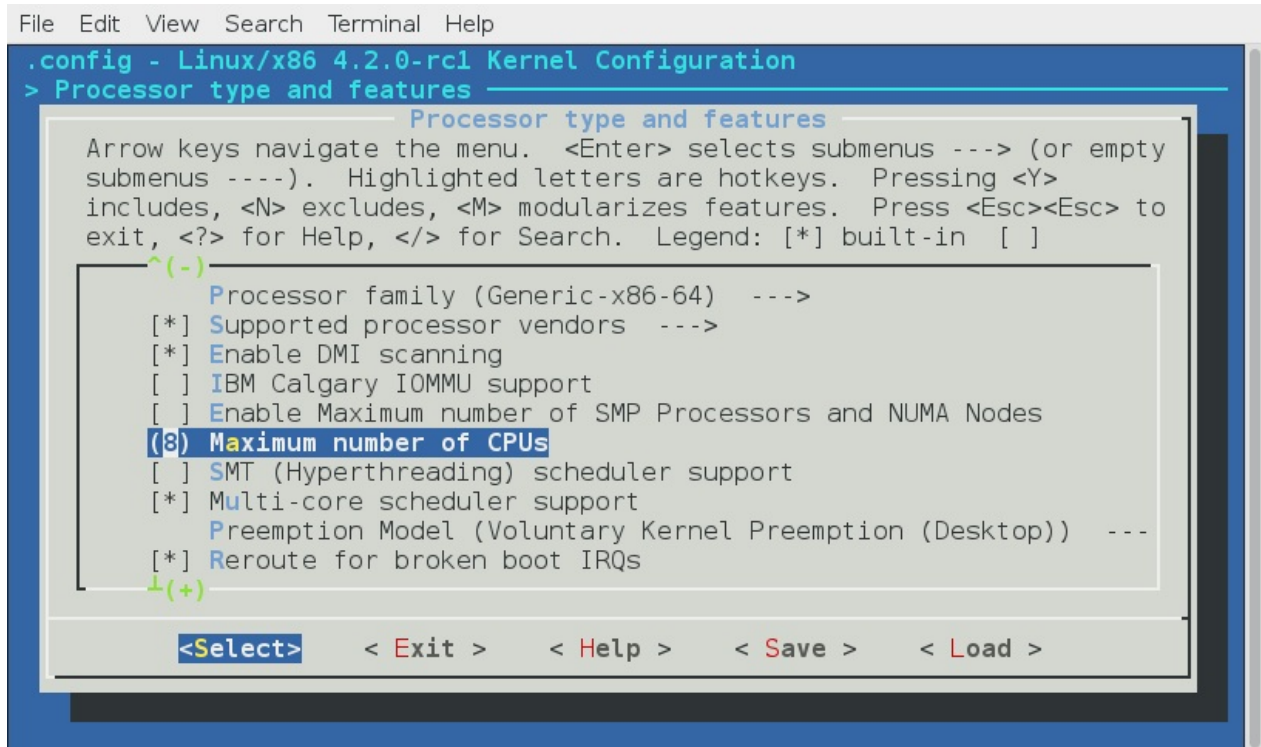
```
#define CPU_VECTOR_LIMIT              (64 * NR_CPUS)
#define NR_VECTORS                    256
#define IO_APIC_VECTOR_LIMIT         ( 32 * MAX_IO_APICS )
#define MAX_IO_APICS                 128

# 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 calculate 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 acquired 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 legacy 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 irq (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, now 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` represents 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 descriptor 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 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-insides](#).**

## 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-specific 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 `vector_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? Actually 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` structure:

```

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);
};
```

Actual default maximum number of the legacy interrupts represented 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 `vector_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` function 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
1:	2	0	0	0	0	0	0	0
8:	1	0	0	0	0	0	0	0

look on the last column;

- `(*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 initialize 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 about `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:

```

#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 predefined 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 calculate its offset on the programm's [stack frame](#). We subtract `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 interrupt gates from the `FIRST_EXTERNAL_VECTOR` or `0x20` to the `0x256`:

```

i = FIRST_EXTERNAL_VECTOR;

#ifdef 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 for 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 is defined in the [arch/x86/kernel/acpi/boot.c](#). This variable is set in the `acpi_set_irq_model_ioapic` function that is called during the processing of the [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 are 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 builds information about [APICs](#) in the [devicetree](#). Note that `of_ioapic` variable depends on the `CONFIG_OF` Linux kernel configuration option. 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 value 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 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 15` - 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 `/proc/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 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-insides](#).**

## 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 deferred interrupt? 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 `deferred 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 split 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 deferred



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 suggest, 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 remaining 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 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	CPU
HI:	5	0	0	0	0	0	
TIMER:	332519	310498	289555	272913	282535	279467	28289
NET_TX:	2320	0	0	2	1	1	
NET_RX:	270221	225	338	281	311	262	43
BLOCK:	134282	32	40	10	12	7	
BLOCK_IOPOLL:	0	0	0	0	0	0	
TASKLET:	196835	2	3	0	0	0	
SCHED:	161852	146745	129539	126064	127998	128014	12024
HRTIMER:	0	0	0	0	0	0	
RCU:	337707	289397	251874	239796	254377	254898	26749

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 deferred 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` flag 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 finished:

```

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 `exit_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 initialization 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 `tasklet_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 generally way. After this we enable interrupts for the local processor and go through the list of tasklets 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 fourth [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-insides](#).**

## Links

- [initcall](#)
- [IF](#)
- [eflags](#)
- [CPU masks](#)
- [per-cpu](#)
- [Workqueue](#)
- [Previous part](#)

# Interrupts and Interrupt Handling. Part 10.

## Last part

This is the tenth part of the [chapter](#) about interrupts and interrupt handling in the Linux kernel and in the previous [part](#) we saw a little about deferred interrupts and related concepts like `softirq`, `tasklet` and `workqueue`. In this part we will continue to dive into this theme and now it's time to look at real hardware driver.

Let's consider serial driver of the [StrongARM\\*\\* SA-110/21285 Evaluation Board](#) board for example and will look how this driver requests an [IRQ](#) line, what happens when an interrupt is triggered and etc. The source code of this driver is placed in the [drivers/tty/serial/21285.c](#) source code file. Ok, we have source code, let's start.

## Initialization of a kernel module

We will start to consider this driver as we usually did it with all new concepts that we saw in this book. We will start to consider it from the initialization. As you already may know, the Linux kernel provides two macros for initialization and finalization of a driver or a kernel module:

- `module_init` ;
- `module_exit` .

And we can find usage of these macros in our driver source code:

```
module_init(serial21285_init);
module_exit(serial21285_exit);
```

The most part of device drivers can be compiled as a loadable kernel [module](#) or in another way they can be statically linked into the Linux kernel. In the first case initialization of a device driver will be produced via the `module_init` and `module_exit` macros that are defined in the [include/linux/init.h](#):

```

#define module_init(initfn) \
    static inline initcall_t __inittest(void) \
    { return initfn; } \
    int init_module(void) __attribute__((alias(#initfn)));

#define module_exit(exitfn) \
    static inline exitcall_t __exittest(void) \
    { return exitfn; } \
    void cleanup_module(void) __attribute__((alias(#exitfn)));

```

and will be called by the `initcall` functions:

- `early_initcall`
- `pure_initcall`
- `core_initcall`
- `postcore_initcall`
- `arch_initcall`
- `subsys_initcall`
- `fs_initcall`
- `rootfs_initcall`
- `device_initcall`
- `late_initcall`

that are called in the `do_initcalls` from the `init/main.c`. Otherwise, if a device driver is statically linked into the Linux kernel, implementation of these macros will be following:

```

#define module_init(x) __initcall(x);
#define module_exit(x) __exitcall(x);

```

In this way implementation of module loading placed in the `kernel/module.c` source code file and initialization occurs in the `do_init_module` function. We will not dive into details about loadable modules in this chapter, but will see it in the special chapter that will describe Linux kernel modules. Ok, the `module_init` macro takes one parameter - the `serial21285_init` in our case. As we can understand from function's name, this function does stuff related to the driver initialization. Let's look at it:



```
static int __init serial21285_init(void)
{
    int ret;

    printk(KERN_INFO "Serial: 21285 driver\n");

    serial21285_setup_ports();

    ret = uart_register_driver(&serial21285_reg);
    if (ret == 0)
        uart_add_one_port(&serial21285_reg, &serial21285_port);

    return ret;
}
```

As we can see, first of all it prints information about the driver to the kernel buffer and the call of the `serial21285_setup_ports` function. This function setups the base `uart` clock of the `serial21285_port` device:

```
unsigned int mem_fclk_21285 = 500000000;

static void serial21285_setup_ports(void)
{
    serial21285_port.uartclk = mem_fclk_21285 / 4;
}
```

Here the `serial21285` is the structure that describes `uart` driver:

```
static struct uart_driver serial21285_reg = {
    .owner          = THIS_MODULE,
    .driver_name    = "ttyFB",
    .dev_name       = "ttyFB",
    .major          = SERIAL_21285_MAJOR,
    .minor          = SERIAL_21285_MINOR,
    .nr             = 1,
    .cons           = SERIAL_21285_CONSOLE,
};
```

If the driver registered successfully we attach the driver-defined port `serial21285_port` structure with the `uart_add_one_port` function from the [drivers/tty/serial/serial\\_core.c](#) source code file and return from the `serial21285_init` function:

```
if (ret == 0)
    uart_add_one_port(&serial21285_reg, &serial21285_port);

return ret;
```

That's all. Our driver is initialized. When an `uart` port will be opened with the call of the `uart_open` function from the [drivers/tty/serial/serial\\_core.c](#), it will call the `uart_startup` function to start up the serial port. This function will call the `startup` function that is part of the `uart_ops` structure. Each `uart` driver has the definition of this structure, in our case it is:

```
static struct uart_ops serial21285_ops = {
    ...
    .startup    = serial21285_startup,
    ...
}
```

`serial21285` structure. As we can see the `.startup` field references on the `serial21285_startup` function. Implementation of this function is very interesting for us, because it is related to the interrupts and interrupt handling.

## Requesting irq line

Let's look at the implementation of the `serial21285` function:

```
static int serial21285_startup(struct uart_port *port)
{
    int ret;

    tx_enabled(port) = 1;
    rx_enabled(port) = 1;

    ret = request_irq(IRQ_CONRX, serial21285_rx_chars, 0,
                     serial21285_name, port);
    if (ret == 0) {
        ret = request_irq(IRQ_CONTX, serial21285_tx_chars, 0,
                          serial21285_name, port);
        if (ret)
            free_irq(IRQ_CONRX, port);
    }

    return ret;
}
```

First of all about `TX` and `RX`. A serial bus of a device consists of just two wires: one for sending data and another for receiving. As such, serial devices should have two serial pins: the receiver - `RX`, and the transmitter - `TX`. With the call of first two macros: `tx_enabled` and `rx_enabled`, we enable these wires. The following part of these function is the greatest

interest for us. Note on `request_irq` functions. This function registers an interrupt handler and enables a given interrupt line. Let's look at the implementation of this function and get into the details. This function defined in the [include/linux/interrupt.h](#) header file and looks as:

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev)
{
    return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}
```

As we can see, the `request_irq` function takes five parameters:

- `irq` - the interrupt number that being requested;
- `handler` - the pointer to the interrupt handler;
- `flags` - the bitmask options;
- `name` - the name of the owner of an interrupt;
- `dev` - the pointer used for shared interrupt lines;

Now let's look at the calls of the `request_irq` functions in our example. As we can see the first parameter is `IRQ_CONRX`. We know that it is number of the interrupt, but what is it `CONRX`? This macro defined in the [arch/arm/mach-footbridge/include/mach/irqs.h](#) header file. We can find the full list of interrupts that the `21285` board can generate. Note that in the second call of the `request_irq` function we pass the `IRQ_CONTX` interrupt number. Both these interrupts will handle `RX` and `TX` event in our driver. Implementation of these macros is easy:

```
#define IRQ_CONRX          _DC21285_IRQ(0)
#define IRQ_CONTX          _DC21285_IRQ(1)
...
...
...
#define _DC21285_IRQ(x)    (16 + (x))
```

The [ISA](#) IRQs on this board are from `0` to `15`, so, our interrupts will have first two numbers: `16` and `17`. Second parameters for two calls of the `request_irq` functions are `serial21285_rx_chars` and `serial21285_tx_chars`. These functions will be called when an `RX` or `TX` interrupt occurred. We will not dive in this part into details of these functions, because this chapter covers the interrupts and interrupts handling but not device and drivers. The next parameter - `flags` and as we can see, it is zero in both calls of the `request_irq` function. All acceptable flags are defined as `IRQF_*` macros in the [include/linux/interrupt.h](#). Some of it:

- `IRQF_SHARED` - allows sharing the irq among several devices;

- `IRQF_PERCPU` - an interrupt is per cpu;
- `IRQF_NO_THREAD` - an interrupt cannot be threaded;
- `IRQF_NOBALANCING` - excludes this interrupt from irq balancing;
- `IRQF_IRQPOLL` - an interrupt is used for polling;
- and etc.

In our case we pass `0`, so it will be `IRQF_TRIGGER_NONE`. This flag means that it does not imply any kind of edge or level triggered interrupt behaviour. To the fourth parameter ( `name` ), we pass the `serial21285_name` that defined as:

```
static const char serial21285_name[] = "Footbridge UART";
```

and will be displayed in the output of the `/proc/interrupts`. And in the last parameter we pass the pointer to the our main `uart_port` structure. Now we know a little about `request_irq` function and its parameters, let's look at its implementation. As we can see above, the `request_irq` function just makes a call of the `request_threaded_irq` function inside. The `request_threaded_irq` function defined in the [kernel/irq/manage.c](#) source code file and allocates a given interrupt line. If we will look at this function, it starts from the definition of the `irqaction` and the `irq_desc`:

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                        irq_handler_t thread_fn, unsigned long irqflags,
                        const char *devname, void *dev_id)
{
    struct irqaction *action;
    struct irq_desc *desc;
    int retval;
    ...
    ...
    ...
}
```

We already saw the `irqaction` and the `irq_desc` structures in this chapter. The first structure represents per interrupt action descriptor and contains pointers to the interrupt handler, name of the device, interrupt number, etc. The second structure represents a descriptor of an interrupt and contains pointer to the `irqaction`, interrupt flags, etc. Note that the `request_threaded_irq` function called by the `request_irq` with the additional parameter: `irq_handler_t thread_fn`. If this parameter is not `NULL`, the `irq` thread will be created and the given `irq` handler will be executed in this thread. In the next step we need to make following checks:

```

if (((irqflags & IRQF_SHARED) && !dev_id) ||
    (!(irqflags & IRQF_SHARED) && (irqflags & IRQF_COND_SUSPEND)) ||
    ((irqflags & IRQF_NO_SUSPEND) && (irqflags & IRQF_COND_SUSPEND)))
    return -EINVAL;

```

First of all we check that real `dev_id` is passed for the shared interrupt and the `IRQF_COND_SUSPEND` only makes sense for shared interrupts. Otherwise we exit from this function with the `-EINVAL` error. After this we convert the given `irq` number to the `irq` descriptor with the help of the `irq_to_desc` function that is defined in the [kernel/irq/irqdesc.c](#) source code file and exit from this function with the `-EINVAL` error if it was not successful:

```

desc = irq_to_desc(irq);
if (!desc)
    return -EINVAL;

```

The `irq_to_desc` function checks that given `irq` number is less than maximum number of IRQs and returns the `irq` descriptor where the `irq` number is offset from the `irq_desc` array:

```

struct irq_desc *irq_to_desc(unsigned int irq)
{
    return (irq < NR_IRQS) ? irq_desc + irq : NULL;
}

```

As we have converted `irq` number to the `irq` descriptor we make the check the status of the descriptor that an interrupt can be requested:

```

if (!irq_settings_can_request(desc) || WARN_ON(irq_settings_is_per_cpu_devid(desc)))
    return -EINVAL;

```

and exit with the `-EINVAL` in other way. After this we check the given interrupt handler. If it was not passed to the `request_irq` function, we check the `thread_fn`. If both handlers are `NULL`, we return with the `-EINVAL`. If an interrupt handler was not passed to the `request_irq` function, but the `thread_fn` is not null, we set handler to the `irq_default_primary_handler`:

```

if (!handler) {
    if (!thread_fn)
        return -EINVAL;
    handler = irq_default_primary_handler;
}

```

In the next step we allocate memory for our `irqaction` with the `kzalloc` function and return from the function if this operation was not successful:

```
action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);
if (!action)
    return -ENOMEM;
```

More about `kzalloc` will be in the separate chapter about [memory management](#) in the Linux kernel. As we allocated space for the `irqaction`, we start to initialize this structure with the values of interrupt handler, interrupt flags, device name, etc:

```
action->handler = handler;
action->thread_fn = thread_fn;
action->flags = irqflags;
action->name = devname;
action->dev_id = dev_id;
```

In the end of the `request_threaded_irq` function we call the `__setup_irq` function from the [kernel/irq/manage.c](#) and registers a given `irqaction`. Release memory for the `irqaction` and return:

```
chip_bus_lock(desc);
retval = __setup_irq(irq, desc, action);
chip_bus_sync_unlock(desc);

if (retval)
    kfree(action);

return retval;
```

Note that the call of the `__setup_irq` function is placed between the `chip_bus_lock` and the `chip_bus_sync_unlock` functions. These functions lock/unlock access to slow bus (like [i2c](#)) chips. Now let's look at the implementation of the `__setup_irq` function. In the beginning of the `__setup_irq` function we can see a couple of different checks. First of all we check that the given interrupt descriptor is not `NULL`, `irqchip` is not `NULL` and that given interrupt descriptor module owner is not `NULL`. After this we check is interrupt nested into another interrupt thread or not, and if it is nested we replace the `irq_default_primary_handler` with the `irq_nested_primary_handler`.

In the next step we create an irq handler thread with the `kthread_create` function, if the given interrupt is not nested and the `thread_fn` is not `NULL`:

```

if (new->thread_fn && !nested) {
    struct task_struct *t;
    t = kthread_create(irq_thread, new, "irq/%d-%s", irq, new->name);
    ...
}

```

And fill the rest of the given interrupt descriptor fields in the end. So, our 16 and 17 interrupt request lines are registered and the `serial21285_rx_chars` and `serial21285_tx_chars` functions will be invoked when an interrupt controller will get event related to these interrupts. Now let's look at what happens when an interrupt occurs.

## Prepare to handle an interrupt

In the previous paragraph we saw the requesting of the irq line for the given interrupt descriptor and registration of the `irqaction` structure for the given interrupt. We already know that when an interrupt event occurs, an interrupt controller notifies the processor about this event and processor tries to find appropriate interrupt gate for this interrupt. If you have read the eighth [part](#) of this chapter, you may remember the `native_init_IRQ` function. This function makes initialization of the local [APIC](#). The following part of this function is the most interesting part for us right now:

```

for_each_clear_bit_from(i, used_vectors, first_system_vector) {
    set_intr_gate(i, irq_entries_start +
        8 * (i - FIRST_EXTERNAL_VECTOR));
}

```

Here we iterate over all the cleared bit of the `used_vectors` bitmap starting at `first_system_vector` that is:

```

int first_system_vector = FIRST_SYSTEM_VECTOR; // 0xef

```

and set interrupt gates with the `i` vector number and the `irq_entries_start + 8 * (i - FIRST_EXTERNAL_VECTOR)` start address. Only one thing is unclear here - the `irq_entries_start`. This symbol defined in the [arch/x86/entry/entry\\_64.S](#) assembly file and provides `irq` entries. Let's look at it:

```

        .align 8
ENTRY(irq_entries_start)
    vector=FIRST_EXTERNAL_VECTOR
    .rept (FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR)
    pushq    $(-vector+0x80)
    vector=vector+1
    jmp      common_interrupt
    .align   8
    .endr
END(irq_entries_start)

```

Here we can see the [GNU assembler](#) `.rept` instruction which repeats the sequence of lines that are before `.endr` - `FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR` times. As we already know, the `FIRST_SYSTEM_VECTOR` is `0xef`, and the `FIRST_EXTERNAL_VECTOR` is equal to `0x20`. So, it will work:

```

>>> 0xef - 0x20
207

```

times. In the body of the `.rept` instruction we push entry stubs on the stack (note that we use negative numbers for the interrupt vector numbers, because positive numbers already reserved to identify [system calls](#)), increase the `vector` variable and jump on the `common_interrupt` label. In the `common_interrupt` we adjust vector number on the stack and execute interrupt number with the `do_IRQ` parameter:

```

common_interrupt:
    addq    $-0x80, (%rsp)
    interrupt do_IRQ

```

The macro `interrupt` defined in the same source code file and saves [general purpose](#) registers on the stack, change the userspace `gs` on the kernel with the `SWAPGS` assembler instruction if need, increase [per-cpu](#) - `irq_count` variable that shows that we are in interrupt and call the `do_IRQ` function. This function defined in the [arch/x86/kernel/irq.c](#) source code file and handles our device interrupt. Let's look at this function. The `do_IRQ` function takes one parameter - `pt_regs` structure that stores values of the userspace registers:



```
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);
    unsigned vector = ~regs->orig_ax;
    unsigned irq;

    irq_enter();
    exit_idle();
    ...
    ...
    ...
}
```

At the beginning of this function we can see call of the `set_irq_regs` function that returns saved `per-cpu` `irq` register pointer and the calls of the `irq_enter` and `exit_idle` functions. The first function `irq_enter` enters to an interrupt context with the updating `__preempt_count` variable and the second function - `exit_idle` checks that current process is `idle` with `pid` - `0` and notify the `idle_notifier` with the `IDLE_END`.

In the next step we read the `irq` for the current `cpu` and call the `handle_irq` function:

```
irq = __this_cpu_read(vector_irq[vector]);

if (!handle_irq(irq, regs)) {
    ...
    ...
    ...
}
...
...
...
```

The `handle_irq` function defined in the [arch/x86/kernel/irq\\_64.c](#) source code file, checks the given interrupt descriptor and call the `generic_handle_irq_desc`:

```
desc = irq_to_desc(irq);
if (unlikely(!desc))
    return false;
generic_handle_irq_desc(irq, desc);
```

Where the `generic_handle_irq_desc` calls the interrupt handler:

```
static inline void generic_handle_irq_desc(unsigned int irq, struct irq_desc *desc)
{
    desc->handle_irq(irq, desc);
}
```

But stop... What is it `handle_irq` and why do we call our interrupt handler from the interrupt descriptor when we know that `irqaction` points to the actual interrupt handler? Actually the `irq_desc->handle_irq` is a high-level API for the calling interrupt handler routine. It setups during initialization of the [device tree](#) and [APIC](#) initialization. The kernel selects correct function and call chain of the `irq->action(s)` there. In this way, the `serial21285_tx_chars` or the `serial21285_rx_chars` function will be executed after an interrupt will occur.

In the end of the `do_IRQ` function we call the `irq_exit` function that will exit from the interrupt context, the `set_irq_regs` with the old userspace registers and return:

```
irq_exit();
set_irq_regs(old_regs);
return 1;
```

We already know that when an `IRQ` finishes its work, deferred interrupts will be executed if they exist.

## Exit from interrupt

Ok, the interrupt handler finished its execution and now we must return from the interrupt. When the work of the `do_IRQ` function will be finished, we will return back to the assembler code in the [arch/x86/entry/entry\\_64.S](#) to the `ret_from_intr` label. First of all we disable interrupts with the `DISABLE_INTERRUPTS` macro that expands to the `cli` instruction and decreases value of the `irq_count` [per-cpu](#) variable. Remember, this variable had value - 1, when we were in interrupt context:

```
DISABLE_INTERRUPTS(CLBR_NONE)
TRACE_IRQS_OFF
decl    PER_CPU_VAR(irq_count)
```

In the last step we check the previous context (user or kernel), restore it in a correct way and exit from an interrupt with the:

```
INTERRUPT_RETURN
```

where the `INTERRUPT_RETURN` macro is:

```
#define INTERRUPT_RETURN    jmp native_iret
```

and

```
ENTRY(native_iret)

.global native_irq_return_iret
native_irq_return_iret:
    iretq
```

That's all.

## Conclusion

It is the end of the tenth part of the [Interrupts and Interrupt Handling](#) chapter and as you have read in the beginning of this part - it is the last part of this chapter. This chapter started from the explanation of the theory of interrupts and we have learned what is it interrupt and kinds of interrupts, then we saw exceptions and handling of this kind of interrupts, deferred interrupts and finally we looked on the hardware interrupts and the handling of theirs in this part. Of course, this part and even this chapter does not cover full aspects of interrupts and interrupt handling in the Linux kernel. It is not realistic to do this. At least for me. It was the big part, I don't know how about you, but it was really big for me. This theme is much bigger than this chapter and I am not sure that somewhere there is a book that covers it. We have missed many part and aspects of interrupts and interrupt handling, but I think it will be good point to dive in the kernel code related to the interrupts and interrupts handling.

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-insides](#).**

## Links

- [Serial driver documentation](#)
- [StrongARM\\*\\* SA-110/21285 Evaluation Board](#)
- [IRQ](#)
- [module](#)
- [initcall](#)
- [uart](#)
- [ISA](#)
- [memory management](#)
- [i2c](#)
- [APIC](#)
- [GNU assembler](#)

- [Processor register](#)
- [per-cpu](#)
- [pid](#)
- [device tree](#)
- [system calls](#)
- [Previous part](#)

# System calls

This chapter describes the `system call` concept in the linux kernel.

- [Introduction to system call concept](#) - this part is introduction to the `system call` concept in the Linux kernel.
- [How the Linux kernel handles a system call](#) - this part describes how the Linux kernel handles a system call from an userspace application.
- [vsyscall and vDSO](#) - third part describes `vsyscall` and `vDSO` concepts.
- [How the Linux kernel runs a program](#) - this part describes startup process of a program.

# System calls in the Linux kernel. Part 1.

## Introduction

This post opens up a new chapter in [linux-insides](#) book, and as you may understand from the title, this chapter will be devoted to the [System call](#) concept in the Linux kernel. The choice of topic for this chapter is not accidental. In the previous [chapter](#) we saw interrupts and interrupt handling. The concept of system calls is very similar to that of interrupts. This is because the most common way to implement system calls is as software interrupts. We will see many different aspects that are related to the system call concept. For example, we will learn what's happening when a system call occurs from userspace. We will see an implementation of a couple system call handlers in the Linux kernel, [VDSO](#) and [vsyscall](#) concepts and many many more.

Before we dive into Linux system call implementation, it is good to know some theory about system calls. Let's do it in the following paragraph.

## System call. What is it?

A system call is just a userspace request of a kernel service. Yes, the operating system kernel provides many services. When your program wants to write to or read from a file, start to listen for connections on a [socket](#), delete or create directory, or even to finish its work, a program uses a system call. In another words, a system call is just a [C](#) kernel space function that user space programs call to handle some request.

The Linux kernel provides a set of these functions and each architecture provides its own set. For example: the [x86\\_64](#) provides [322](#) system calls and the [x86](#) provides [358](#) different system calls. Ok, a system call is just a function. Let's look on a simple `Hello world` example that's written in the assembly programming language:

```
.data

msg:
    .ascii "Hello, world!\n"
    len = . - msg

.text
    .global _start

_start:
    movq $1, %rax
    movq $1, %rdi
    movq $msg, %rsi
    movq $len, %rdx
    syscall

    movq $60, %rax
    xorq %rdi, %rdi
    syscall
```

We can compile the above with the following commands:

```
$ gcc -c test.S
$ ld -o test test.o
```

and run it as follows:

```
./test
Hello, world!
```

Ok, what do we see here? This simple code represents `Hello world` assembly program for the Linux `x86_64` architecture. We can see two sections here:

- `.data`
- `.text`

The first section - `.data` stores initialized data of our program ( `Hello world` string and its length in our case). The second section - `.text` contains the code of our program. We can split the code of our program into two parts: first part will be before the first `syscall` instruction and the second part will be between first and second `syscall` instructions. First of all what does the `syscall` instruction do in our code and generally? As we can read in the [64-bit architectures software developer manual](#):

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32\_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32\_LSTAR MSR always contain a canonical address.)

...  
...  
...

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32\_STAR MSR. However, the CS and SS descriptor caches are not loaded from the descriptors (in GDT or LDT) referenced by those selectors.

Instead, the descriptor caches are loaded with fixed values. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

and we are initializing `syscalls` by the writing of the `entry_SYSCALL_64` that defined in the [arch/x86/entry/entry\\_64.S](#) assembler file and represents `SYSCALL` instruction entry to the `IA32_STAR` [Model specific register](#):

```
wrmsrl(MSR_LSTAR, entry_SYSCALL_64);
```

in the [arch/x86/kernel/cpu/common.c](#) source code file.

So, the `syscall` instruction invokes a handler of a given system call. But how does it know which handler to call? Actually it gets this information from the general purpose [registers](#). As you can see in the system call [table](#), each system call has a unique number. In our example, first system call is - `write` that writes data to the given file. Let's look in the system call table and try to find `write` system call. As we can see, the `write` system call has number - `1`. We pass the number of this system call through the `rax` register in our example. The next general purpose registers: `%rdi`, `%rsi` and `%rdx` take parameters of the `write` syscall. In our case, they are [file descriptor](#) (`1` is `stdout` in our case), second parameter is the pointer to our string, and the third is size of data. Yes, you heard right. Parameters for a system call. As I already wrote above, a system call is a just `c` function in the kernel space. In our case first system call is `write`. This system call defined in the [fs/read\\_write.c](#) source code file and looks like:

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                 size_t, count)
{
    ...
    ...
    ...
}
```



Or in other words:

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

Don't worry about the `SYSCALL_DEFINE3` macro for now, we'll come back to it.

The second part of our example is the same, but we call other system call. In this case we call `exit` system call. This system call gets only one parameter:

- Return value

and handles the way our program exits. We can pass the program name of our program to the `strace` util and we will see our system calls:

```
$ strace test
execve("./test", [ "./test" ], [ /* 62 vars */ ]) = 0
write(1, "Hello, world!\n", 14Hello, world!
)      = 14
_exit(0)                                = ?

+++ exited with 0 +++
```

In the first line of the `strace` output, we can see `execve` system call that executes our program, and the second and third are system calls that we have used in our program: `write` and `exit`. Note that we pass the parameter through the general purpose registers in our example. The order of the registers is not accidental. The order of the registers is defined by the following agreement - [x86-64 calling conventions](#). This and other agreement for the `x86_64` architecture explained in the special document - [System V Application Binary Interface. PDF](#). In a general way, argument(s) of a function are placed either in registers or pushed on the stack. The right order is:

- `rdi` ;
- `rsi` ;
- `rdx` ;
- `rcx` ;
- `r8` ;
- `r9` .

for the first six parameters of a function. If a function has more than six arguments, other parameters will be placed on the stack.

We do not use system calls in our code directly, but our program uses it when we want to print something, check access to a file or just write or read something to it.

For example:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    FILE *fp;
    char buff[255];

    fp = fopen("test.txt", "r");
    fgets(buff, 255, fp);
    printf("%s\n", buff);
    fclose(fp);

    return 0;
}
```

There are no `fopen`, `fgets`, `printf` and `fclose` system calls in the Linux kernel, but `open`, `read`, `write` and `close` instead. I think you know that these four functions `fopen`, `fgets`, `printf` and `fclose` are just functions that defined in the `c` [standard library](#). Actually these functions are wrappers for the system calls. We do not call system calls directly in our code, but using [wrapper](#) functions from the standard library. The main reason of this is simple: a system call must be performed quickly, very quickly. As a system call must be quick, it must be small. The standard library takes responsibility to perform system calls with the correct set parameters and makes different checks before it will call the given system call. Let's compile our program with the following command:

```
$ gcc test.c -o test
```

and look on it with the [ltrace](#) util:

```
$ ltrace ./test
__libc_start_main([ "./test" ] <unfinished ...>
fopen("test.txt", "r")                      = 0x602010
fgets("Hello World!\n", 255, 0x602010)      = 0x7ffd2745e700
puts("Hello World!\n"Hello World!

)   = 14
fclose(0x602010)                            = 0
+++ exited (status 0) +++
```

The `ltrace` util displays a set of userspace calls of a program. The `fopen` function opens the given text file, the `fgets` reads file content to the `buf` buffer, the `puts` function prints it to the `stdout` and the `fclose` function closes file by the given file descriptor. And as I

already wrote, all of these functions call an appropriate system call. For example `puts` calls the `write` system call inside, we can see it if we will add `-s` option to the `ltrace` program:

```
write@SYS(1, "Hello World!\n\n", 14) = 14
```

Yes, system calls are ubiquitous. Each program needs to open/write/read file, network connection, allocate memory and many other things that can be provided only by the kernel. The `proc` file system contains special files in a format: `/proc/pid/systemcall` that exposes the system call number and argument registers for the system call currently being executed by the process. For example, pid 1, that is `systemd` for me:

```
$ sudo cat /proc/1/comm
systemd

$ sudo cat /proc/1/syscall
232 0x4 0x7ffdf82e11b0 0x1f 0xffffffff 0x100 0x7ffdf82e11bf 0x7ffdf82e11a0 0x7f9114681193
```

the system call with number - `232` which is `epoll_wait` system call that waits for an I/O event on an `epoll` file descriptor. Or for example `emacs` editor where I'm writing this part:

```
$ ps ax | grep emacs
2093 ?          Sl      2:40 emacs

$ sudo cat /proc/2093/comm
emacs

$ sudo cat /proc/2093/syscall
270 0xf 0x7fff068a5a90 0x7fff068a5b10 0x0 0x7fff068a59c0 0x7fff068a59d0 0x7fff068a59b0 0x
```

the system call with the number `270` which is `sys_pselect6` system call that allows `emacs` to monitor multiple file descriptors.

Now we know a little about system call, what is it and why we need in it. So let's look at the `write` system call that our program used.

## Implementation of write system call

Let's look at the implementation of this system call directly in the source code of the Linux kernel. As we already know, the `write` system call is defined in the `fs/read_write.c` source code file and looks like this:

```

SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                 size_t, count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos = file_pos_read(f.file);
        ret = vfs_write(f.file, buf, count, &pos);
        if (ret >= 0)
            file_pos_write(f.file, pos);
        fdput_pos(f);
    }

    return ret;
}

```

First of all, the `SYSCALL_DEFINE3` macro is defined in the [include/linux/syscalls.h](#) header file and expands to the definition of the `sys_name(...)` function. Let's look at this macro:

```

#define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, _##name, __VA_ARGS__)

#define SYSCALL_DEFINEx(x, sname, ...) \
    SYSCALL_METADATA(sname, x, __VA_ARGS__) \
    __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)

```

As we can see the `SYSCALL_DEFINE3` macro takes `name` parameter which will represent name of a system call and variadic number of parameters. This macro just expands to the `SYSCALL_DEFINEx` macro that takes the number of the parameters the given system call, the `_##name` stub for the future name of the system call (more about tokens concatenation with the `##` you can read in the [documentation](#) of `gcc`). Next we can see the `SYSCALL_DEFINEx` macro. This macro expands to the two following macros:

- `SYSCALL_METADATA ;`
- `__SYSCALL_DEFINEx .`

Implementation of the first macro `SYSCALL_METADATA` depends on the `CONFIG_FTRACE_SYSCALLS` kernel configuration option. As we can understand from the name of this option, it allows to enable tracer to catch the syscall entry and exit events. If this kernel configuration option is enabled, the `SYSCALL_METADATA` macro executes initialization of the `syscall_metadata` structure that defined in the [include/trace/syscall.h](#) header file and contains different useful fields as name of a system call, number of a system call in the system call [table](#), number of parameters of a system call, list of parameter types and etc:

```
#define SYSCALL_METADATA(sname, nb, ...) \
... \
... \
... \
struct syscall_metadata __used \
__syscall_meta_##sname = { \
    .name          = "sys"#sname, \
    .syscall_nr     = -1, \
    .nb_args        = nb, \
    .types          = nb ? types_##sname : NULL, \
    .args           = nb ? args_##sname : NULL, \
    .enter_event    = &event_enter_##sname, \
    .exit_event     = &event_exit_##sname, \
    .enter_fields   = LIST_HEAD_INIT(__syscall_meta_##sname.enter_fields) \
};

static struct syscall_metadata __used \
__attribute__((section("__syscalls_metadata"))) \
*_p_syscall_meta_##sname = &__syscall_meta_##sname;
```

If the `CONFIG_FTRACE_SYSCALLS` kernel option does not enabled during kernel configuration, in this way the `SYSCALL_METADATA` macro expands to empty string:

```
#define SYSCALL_METADATA(sname, nb, ...)
```

The second macro `__SYSCALL_DEFINEX` expands to the definition of the five following functions:

```
#define __SYSCALL_DEFINE(x, name, ...)
asmlinkage long sys##name(__MAP(x,__SC_DECL,__VA_ARGS__))
    __attribute__((alias(__stringify(Sys##name))));

static inline long SYSC##name(__MAP(x,__SC_DECL,__VA_ARGS__));

asmlinkage long Sys##name(__MAP(x,__SC_LONG,__VA_ARGS__));

asmlinkage long Sys##name(__MAP(x,__SC_LONG,__VA_ARGS__))
{
    long ret = SYSC##name(__MAP(x,__SC_CAST,__VA_ARGS__));
    __MAP(x,__SC_TEST,__VA_ARGS__);
    __PROTECT(x, ret,__MAP(x,__SC_ARGS,__VA_ARGS__));
    return ret;
}

static inline long SYSC##name(__MAP(x,__SC_DECL,__VA_ARGS__))
```

The first `sys##name` is definition of the syscall handler function with the given name - `sys_system_call_name`. The `__SC_DECL` macro takes the `__VA_ARGS__` and combines call input parameter system type and the parameter name, because the macro definition is unable to determine the parameter types. And the `__MAP` macro applies `__SC_DECL` macro to the `__VA_ARGS__` arguments. The other functions that are generated by the `__SYSCALL_DEFINE3` macro are need to protect from the [CVE-2009-0029](#) and we will not dive into details about this here. Ok, as result of the `SYSCALL_DEFINE3` macro, we will have:

```
asmlinkage long sys_write(unsigned int fd, const char __user * buf, size_t count);
```

Now we know a little about the system call's definition and we can go back to the implementation of the `write` system call. Let's look on the implementation of this system call again:

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                size_t, count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos = file_pos_read(f.file);
        ret = vfs_write(f.file, buf, count, &pos);
        if (ret >= 0)
            file_pos_write(f.file, pos);
        fdput_pos(f);
    }

    return ret;
}
```

As we already know and can see from the code, it takes three arguments:

- `fd` - file descriptor;
- `buf` - buffer to write;
- `count` - length of buffer to write.

and writes data from a buffer declared by the user to a given device or a file. Note that the second parameter `buf`, defined with the `__user` attribute. The main purpose of this attribute is for checking the Linux kernel code with the [sparse](#) util. It is defined in the [include/linux/compiler.h](#) header file and depends on the `__CHECKER__` definition in the Linux kernel. That's all about useful meta-information related to our `sys_write` system call, let's try to understand how this system call is implemented. As we can see it starts from the definition of the `f` structure that has `fd` structure type that represent file descriptor in the

Linux kernel and we put the result of the call of the `fdget_pos` function. The `fdget_pos` function defined in the same [source](#) code file and just expands the call of the `__to_fd` function:

```
static inline struct fd fdget_pos(int fd)
{
    return __to_fd(__fdget_pos(fd));
}
```

The main purpose of the `fdget_pos` is to convert the given file descriptor which is just a number to the `fd` structure. Through the long chain of function calls, the `fdget_pos` function gets the file descriptor table of the current process, `current->files`, and tries to find a corresponding file descriptor number there. As we got the `fd` structure for the given file descriptor number, we check it and return if it does not exist. We get the current position in the file with the call of the `file_pos_read` function that just returns `f_pos` field of the our file:

```
static inline loff_t file_pos_read(struct file *file)
{
    return file->f_pos;
}
```

and call the `vfs_write` function. The `vfs_write` function defined in the [fs/read\\_write.c](#) source code file and does the work for us - writes given buffer to the given file starting from the given position. We will not dive into details about the `vfs_write` function, because this function is weakly related to the `system call` concept but mostly about [Virtual file system](#) concept which we will see in another chapter. After the `vfs_write` has finished its work, we check the result and if it was finished successfully we change the position in the file with the `file_pos_write` function:

```
if (ret >= 0)
    file_pos_write(f.file, pos);
```

that just updates `f_pos` with the given position in the given file:

```
static inline void file_pos_write(struct file *file, loff_t pos)
{
    file->f_pos = pos;
}
```

At the end of the our `write` system call handler, we can see the call of the following function:

```
fdput_pos(f);
```

unlocks the `f_pos_lock` mutex that protects file position during concurrent writes from threads that share file descriptor.

That's all.

We have seen the partial implementation of one system call provided by the Linux kernel. Of course we have missed some parts in the implementation of the `write` system call, because as I mentioned above, we will see only system calls related stuff in this chapter and will not see other stuff related to other subsystems, such as [Virtual file system](#).

## Conclusion

This concludes the first part covering system call concepts in the Linux kernel. We have covered the theory of system calls so far and in the next part we will continue to dive into this topic, touching Linux kernel code related to system calls.

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [system call](#)
- [vdso](#)
- [vsyscall](#)
- [general purpose registers](#)
- [socket](#)
- [C programming language](#)
- [x86](#)
- [x86\\_64](#)
- [x86-64 calling conventions](#)
- [System V Application Binary Interface. PDF](#)
- [GCC](#)
- [Intel manual. PDF](#)
- [system call table](#)
- [GCC macro documentation](#)



- [file descriptor](#)
- [stdout](#)
- [strace](#)
- [standard library](#)
- [wrapper functions](#)
- [ltrace](#)
- [sparse](#)
- [proc file system](#)
- [Virtual file system](#)
- [systemd](#)
- [epoll](#)
- [Previous chapter](#)

# System calls in the Linux kernel. Part 2.

## How does the Linux kernel handle a system call

The previous [part](#) was the first part of the chapter that describes the [system call](#) concepts in the Linux kernel. In the previous part we learned what a system call is in the Linux kernel, and in operating systems in general. This was introduced from a user-space perspective, and part of the [write](#) system call implementation was discussed. In this part we continue our look at system calls, starting with some theory before moving onto the Linux kernel code.

A user application does not make the system call directly from our applications. We did not write the `Hello world!` program like:

```
int main(int argc, char **argv)
{
    ...
    ...
    ...
    sys_write(fd1, buf, strlen(buf));
    ...
    ...
}
```

We can use something similar with the help of [C standard library](#) and it will look something like this:

```
#include <unistd.h>

int main(int argc, char **argv)
{
    ...
    ...
    ...
    write(fd1, buf, strlen(buf));
    ...
    ...
}
```

But anyway, `write` is not a direct system call and not a kernel function. An application must fill general purpose registers with the correct values in the correct order and use the `syscall` instruction to make the actual system call. In this part we will look at what occurs in

the Linux kernel when the `syscall` instruction is met by the processor.

## Initialization of the system calls table

From the previous part we know that system call concept is very similar to an interrupt. Furthermore, system calls are implemented as software interrupts. So, when the processor handles a `syscall` instruction from a user application, this instruction causes an exception which transfers control to an exception handler. As we know, all exception handlers (or in other words kernel C functions that will react on an exception) are placed in the kernel code. But how does the Linux kernel search for the address of the necessary system call handler for the related system call? The Linux kernel contains a special table called the `system call table`. The system call table is represented by the `sys_call_table` array in the Linux kernel which is defined in the `arch/x86/entry/syscall_64.c` source code file. Let's look at its implementation:

```
asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    [0 ... __NR_syscall_max] = &sys_ni_syscall,
    #include <asm/syscalls_64.h>
};
```

As we can see, the `sys_call_table` is an array of `__NR_syscall_max + 1` size where the `__NR_syscall_max` macro represents the maximum number of system calls for the given [architecture](#). This book is about the [x86\\_64](#) architecture, so for our case the `__NR_syscall_max` is `322` and this is the correct number at the time of writing (current Linux kernel version is `4.2.0-rc8+`). We can see this macro in the header file generated by [Kbuild](#) during kernel compilation - `include/generated/asm-offsets.h`:

```
#define __NR_syscall_max 322
```

There will be the same number of system calls in the `arch/x86/entry/syscalls/syscall_64.tbl` for the `x86_64`. There are two important topics here; the type of the `sys_call_table` array, and the initialization of elements in this array. First of all, the type. The `sys_call_ptr_t` represents a pointer to a system call table. It is defined as [typedef](#) for a function pointer that returns nothing and does not take arguments:

```
typedef void (*sys_call_ptr_t)(void);
```

The second thing is the initialization of the `sys_call_table` array. As we can see in the code above, all elements of our array that contain pointers to the system call handlers point to the `sys_ni_syscall`. The `sys_ni_syscall` function represents not-implemented system calls. To

start with, all elements of the `sys_call_table` array point to the not-implemented system call. This is the correct initial behaviour, because we only initialize storage of the pointers to the system call handlers, it is populated later on. Implementation of the `sys_ni_syscall` is pretty easy, it just returns `-errno` or `-ENOSYS` in our case:

```
asmlinkage long sys_ni_syscall(void)
{
    return -ENOSYS;
}
```

The `-ENOSYS` error tells us that:

ENOSYS	Function not implemented (POSIX.1)
--------	------------------------------------

Also a note on `...` in the initialization of the `sys_call_table`. We can do it with a [GCC](#) compiler extension called - [Designated Initializers](#). This extension allows us to initialize elements in non-fixed order. As you can see, we include the `asm/syscalls_64.h` header at the end of the array. This header file is generated by the special script at [arch/x86/entry/syscalls/syscalltbl.sh](#) and generates our header file from the [syscall table](#).

The `asm/syscalls_64.h` contains definitions of the following macros:

```
__SYSCALL_COMMON(0, sys_read, sys_read)
__SYSCALL_COMMON(1, sys_write, sys_write)
__SYSCALL_COMMON(2, sys_open, sys_open)
__SYSCALL_COMMON(3, sys_close, sys_close)
__SYSCALL_COMMON(5, sys_newfstat, sys_newfstat)
...
...
...
```

The `__SYSCALL_COMMON` macro is defined in the same source code [file](#) and expands to the `__SYSCALL_64` macro which expands to the function definition:

```
#define __SYSCALL_COMMON(nr, sym, compat) __SYSCALL_64(nr, sym, compat)
#define __SYSCALL_64(nr, sym, compat) [nr] = sym,
```

So, after this, our `sys_call_table` takes the following form:

```

asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    [0 ... __NR_syscall_max] = &sys_ni_syscall,
    [0] = sys_read,
    [1] = sys_write,
    [2] = sys_open,
    ...
    ...
    ...
};

```

After this all elements that point to the non-implemented system calls will contain the address of the `sys_ni_syscall` function that just returns `-ENOSYS` as we saw above, and other elements will point to the `sys_syscall_name` functions.

At this point, we have filled the system call table and the Linux kernel knows where each system call handler is. But the Linux kernel does not call a `sys_syscall_name` function immediately after it is instructed to handle a system call from a user space application. Remember the [chapter](#) about interrupts and interrupt handling. When the Linux kernel gets the control to handle an interrupt, it had to do some preparations like save user space registers, switch to a new stack and many more tasks before it will call an interrupt handler. There is the same situation with the system call handling. The preparation for handling a system call is the first thing, but before the Linux kernel will start these preparations, the entry point of a system call must be initialized and only the Linux kernel knows how to perform this preparation. In the next paragraph we will see the process of the initialization of the system call entry in the Linux kernel.

## Initialization of the system call entry

When a system call occurs in the system, where are the first bytes of code that starts to handle it? As we can read in the Intel manual - [64-ia-32-architectures-software-developer-vol-2b-manual](#):

```

SYSCALL invokes an OS system-call handler at privilege level 0.
It does so by loading RIP from the IA32_LSTAR MSR

```

it means that we need to put the system call entry in to the `IA32_LSTAR` [model specific register](#). This operation takes place during the Linux kernel initialization process. If you have read the fourth [part](#) of the chapter that describes interrupts and interrupt handling in the Linux kernel, you know that the Linux kernel calls the `trap_init` function during the initialization process. This function is defined in the [arch/x86/kernel/setup.c](#) source code file and executes the initialization of the `non-early` exception handlers like divide error,

`coprocessor` error etc. Besides the initialization of the `non-early` exceptions handlers, this function calls the `cpu_init` function from the `arch/x86/kernel/cpu/common.c` source code file which besides initialization of `per-cpu` state, calls the `syscall_init` function from the same source code file.

This function performs the initialization of the system call entry point. Let's look on the implementation of this function. It does not take parameters and first of all it fills two model specific registers:

```
wrmsrl(MSR_STAR, ((u64)__USER32_CS)<<48 | ((u64)__KERNEL_CS)<<32);
wrmsrl(MSR_LSTAR, entry_SYSCALL_64);
```

The first model specific register - `MSR_STAR` contains `63:48` bits of the user code segment. These bits will be loaded to the `cs` and `ss` segment registers for the `sysret` instruction which provides functionality to return from a system call to user code with the related privilege. Also the `MSR_STAR` contains `47:32` bits from the kernel code that will be used as the base selector for `cs` and `ss` segment registers when user space applications execute a system call. In the second line of code we fill the `MSR_LSTAR` register with the `entry_SYSCALL_64` symbol that represents system call entry. The `entry_SYSCALL_64` is defined in the `arch/x86/entry/entry_64.S` assembly file and contains code related to the preparation performed before a system call handler will be executed (I already wrote about these preparations, read above). We will not consider the `entry_SYSCALL_64` now, but will return to it later in this chapter.

After we have set the entry point for system calls, we need to set the following model specific registers:

- `MSR_CSTAR` - target `rip` for the compatibility mode callers;
- `MSR_IA32_SYSENTER_CS` - target `cs` for the `sysenter` instruction;
- `MSR_IA32_SYSENTER_ESP` - target `esp` for the `sysenter` instruction;
- `MSR_IA32_SYSENTER_EIP` - target `eip` for the `sysenter` instruction.

The values of these model specific register depend on the `CONFIG_IA32_EMULATION` kernel configuration option. If this kernel configuration option is enabled, it allows legacy 32-bit programs to run under a 64-bit kernel. In the first case, if the `CONFIG_IA32_EMULATION` kernel configuration option is enabled, we fill these model specific registers with the entry point for the system calls the compatibility mode:

```
wrmsrl(MSR_CSTAR, entry_SYSCALL_compat);
```

and with the kernel code segment, put zero to the stack pointer and write the address of the `entry_SYSENTER_compat` symbol to the [instruction pointer](#):

```
wrmsrl_safe(MSR_IA32_SYSENTER_CS, (u64)__KERNEL_CS);
wrmsrl_safe(MSR_IA32_SYSENTER_ESP, 0ULL);
wrmsrl_safe(MSR_IA32_SYSENTER_EIP, (u64)entry_SYSENTER_compat);
```

In another way, if the `CONFIG_IA32_EMULATION` kernel configuration option is disabled, we write `ignore_sysret` symbol to the `MSR_CSTAR` :

```
wrmsrl(MSR_CSTAR, ignore_sysret);
```

that is defined in the [arch/x86/entry/entry\\_64.S](#) assembly file and just returns `-ENOSYS` error code:

```
ENTRY(ignore_sysret)
    mov    $-ENOSYS, %eax
    sysret
END(ignore_sysret)
```

Now we need to fill `MSR_IA32_SYSENTER_CS` , `MSR_IA32_SYSENTER_ESP` , `MSR_IA32_SYSENTER_EIP` model specific registers as we did in the previous code when the `CONFIG_IA32_EMULATION` kernel configuration option was enabled. In this case (when the `CONFIG_IA32_EMULATION` configuration option is not set) we fill the `MSR_IA32_SYSENTER_ESP` and the `MSR_IA32_SYSENTER_EIP` with zero and put the invalid segment of the [Global Descriptor Table](#) to the `MSR_IA32_SYSENTER_CS` model specific register:

```
wrmsrl_safe(MSR_IA32_SYSENTER_CS, (u64)GDT_ENTRY_INVALID_SEG);
wrmsrl_safe(MSR_IA32_SYSENTER_ESP, 0ULL);
wrmsrl_safe(MSR_IA32_SYSENTER_EIP, 0ULL);
```

You can read more about the `Global Descriptor Table` in the second [part](#) of the chapter that describes the booting process of the Linux kernel.

At the end of the `syscall_init` function, we just mask flags in the [flags register](#) by writing the set of flags to the `MSR_SYSCALL_MASK` model specific register:

```
wrmsrl(MSR_SYSCALL_MASK,
        X86_EFLAGS_TF|X86_EFLAGS_DF|X86_EFLAGS_IF|
        X86_EFLAGS_IOPL|X86_EFLAGS_AC|X86_EFLAGS_NT);
```

These flags will be cleared during syscall initialization. That's all, it is the end of the `syscall_init` function and it means that system call entry is ready to work. Now we can see what will occur when a user application executes the `syscall` instruction.

# Preparation before system call handler will be called

As I already wrote, before a system call or an interrupt handler will be called by the Linux kernel we need to do some preparations. The `identry` macro performs the preparations required before an exception handler will be executed, the `interrupt` macro performs the preparations required before an interrupt handler will be called and the `entry_SYSCALL_64` will do the preparations required before a system call handler will be executed.

The `entry_SYSCALL_64` is defined in the [arch/x86/entry/entry\\_64.S](#) assembly file and starts from the following macro:

```
SWAPGS_UNSAFE_STACK
```

This macro is defined in the [arch/x86/include/asm/irqflags.h](#) header file and expands to the `swapgs` instruction:

```
#define SWAPGS_UNSAFE_STACK    swapgs
```

which exchanges the current GS base register value with the value contained in the `MSR_KERNEL_GS_BASE` model specific register. In other words we moved it on to the kernel stack. After this we point the old stack pointer to the `rsp_scratch` [per-cpu](#) variable and setup the stack pointer to point to the top of stack for the current processor:

```
movq    %rsp, PER_CPU_VAR(rsp_scratch)
movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp
```

In the next step we push the stack segment and the old stack pointer to the stack:

```
pushq    $__USER_DS
pushq    PER_CPU_VAR(rsp_scratch)
```

After this we enable interrupts, because interrupts are `off` on entry and save the general purpose [registers](#) (besides `bp`, `bx` and from `r12` to `r15`), flags, `-ENOSYS` for the non-implemented system call and code segment register on the stack:



```
ENABLE_INTERRUPTS(CLBR_NONE)
```

```
pushq    %r11
pushq    $__USER_CS
pushq    %rcx
pushq    %rax
pushq    %rdi
pushq    %rsi
pushq    %rdx
pushq    %rcx
pushq    $-ENOSYS
pushq    %r8
pushq    %r9
pushq    %r10
pushq    %r11
sub      $(6*8), %rsp
```

When a system call occurs from the user's application, general purpose registers have the following state:

- `rax` - contains system call number;
- `rcx` - contains return address to the user space;
- `r11` - contains register flags;
- `rdi` - contains first argument of a system call handler;
- `rsi` - contains second argument of a system call handler;
- `rdx` - contains third argument of a system call handler;
- `r10` - contains fourth argument of a system call handler;
- `r8` - contains fifth argument of a system call handler;
- `r9` - contains sixth argument of a system call handler;

Other general purpose registers (as `rbp`, `rbx` and from `r12` to `r15`) are callee-preserved in [C ABI](#)). So we push register flags on the top of the stack, then user code segment, return address to the user space, system call number, first three arguments, dump error code for the non-implemented system call and other arguments on the stack.

In the next step we check the `_TIF_WORK_SYSCALL_ENTRY` in the current `thread_info` :

```
testl    $_TIF_WORK_SYSCALL_ENTRY, ASM_THREAD_INFO(TI_flags, %rsp, sizeof_ptregs)
jnz      tracesys
```

The `_TIF_WORK_SYSCALL_ENTRY` macro is defined in the [arch/x86/include/asm/thread\\_info.h](#) header file and provides set of the thread information flags that are related to the system calls tracing:

```
#define _TIF_WORK_SYSCALL_ENTRY \
    (_TIF_SYSCALL_TRACE | _TIF_SYSCALL_EMU | _TIF_SYSCALL_AUDIT | \
     _TIF_SECCOMP | _TIF_SINGLESTEP | _TIF_SYSCALL_TRACEPOINT | \
     _TIF_NOHZ)
```

We will not consider debugging/tracing related stuff in this chapter, but will see it in the separate chapter that will be devoted to the debugging and tracing techniques in the Linux kernel. After the `tracesys` label, the next label is the `entry_SYSCALL_64_fastpath`. In the `entry_SYSCALL_64_fastpath` we check the `__SYSCALL_MASK` that is defined in the [arch/x86/include/asm/unistd.h](#) header file and

```
# ifdef CONFIG_X86_X32_ABI
#  define __SYSCALL_MASK (~(__X32_SYSCALL_BIT))
# else
#  define __SYSCALL_MASK (~0)
# endif
```

where the `__X32_SYSCALL_BIT` is

```
#define __X32_SYSCALL_BIT    0x40000000
```

As we can see the `__SYSCALL_MASK` depends on the `CONFIG_X86_X32_ABI` kernel configuration option and represents the mask for the 32-bit [ABI](#) in the 64-bit kernel.

So we check the value of the `__SYSCALL_MASK` and if the `CONFIG_X86_X32_ABI` is disabled we compare the value of the `rax` register to the maximum syscall number (`__NR_syscall_max`), alternatively if the `CONFIG_X86_X32_ABI` is enabled we mask the `eax` register with the `__X32_SYSCALL_BIT` and do the same comparison:

```
#if __SYSCALL_MASK == ~0
    cmpq    $__NR_syscall_max, %rax
#else
    andl    __SYSCALL_MASK, %eax
    cmpl    $__NR_syscall_max, %eax
#endif
```

After this we check the result of the last comparison with the `ja` instruction that executes if `CF` and `ZF` flags are zero:

```
ja    1f
```

and if we have the correct system call for this, we move the fourth argument from the `r10` to the `rcx` to keep [x86\\_64 C ABI](#) compliant and execute the `call` instruction with the address of a system call handler:

```
movq    %r10, %rcx
call    *sys_call_table(, %rax, 8)
```

Note, the `sys_call_table` is an array that we saw above in this part. As we already know the `rax` general purpose register contains the number of a system call and each element of the `sys_call_table` is 8-bytes. So we are using `*sys_call_table(, %rax, 8)` this notation to find the correct offset in the `sys_call_table` array for the given system call handler.

That's all. We did all the required preparations and the system call handler was called for the given interrupt handler, for example `sys_read`, `sys_write` or other system call handler that is defined with the `SYSCALL_DEFINE[N]` macro in the Linux kernel code.

## Exit from a system call

After a system call handler finishes its work, we will return back to the [arch/x86/entry/entry\\_64.S](#), right after where we have called the system call handler:

```
call    *sys_call_table(, %rax, 8)
```

The next step after we've returned from a system call handler is to put the return value of a system handler on to the stack. We know that a system call returns the result to the user program in the general purpose `rax` register, so we are moving its value on to the stack after the system call handler has finished its work:

```
movq    %rax, RAX(%rsp)
```

on the `RAX` place.

After this we can see the call of the `LOCKDEP_SYS_EXIT` macro from the [arch/x86/include/asm/irqflags.h](#):

```
LOCKDEP_SYS_EXIT
```

The implementation of this macro depends on the `CONFIG_DEBUG_LOCK_ALLOC` kernel configuration option that allows us to debug locks on exit from a system call. And again, we will not consider it in this chapter, but will return to it in a separate one. In the end of the

`entry_SYSCALL_64` function we restore all general purpose registers besides `rcx` and `r11`, because the `rcx` register must contain the return address to the application that called system call and the `r11` register contains the old [flags register](#). After all general purpose registers are restored, we fill `rcx` with the return address, `r11` register with the flags and `rsp` with the old stack pointer:

```
RESTORE_C_REGS_EXCEPT_RCX_R11
```

```
movq    RIP(%rsp), %rcx
movq    EFLAGS(%rsp), %r11
movq    RSP(%rsp), %rsp
```

```
USERGS_SYSRET64
```

In the end we just call the `USERGS_SYSRET64` macro that expands to the call of the `swapgs` instruction which exchanges again the user `GS` and kernel `GS` and the `sysretq` instruction which executes on exit from a system call handler:

```
#define USERGS_SYSRET64 \
    swapgs; \
    sysretq;
```

Now we know what occurs when a user application calls a system call. The full path of this process is as follows:

- User application contains code that fills general purposer register with the values (system call number and arguments of this system call);
- Processor switches from the user mode to kernel mode and starts execution of the system call entry - `entry_SYSCALL_64` ;
- `entry_SYSCALL_64` switches to the kernel stack and saves some general purpose registers, old stack and code segment, flags and etc... on the stack;
- `entry_SYSCALL_64` checks the system call number in the `rax` register, searches a system call handler in the `sys_call_table` and calls it, if the number of a system call is correct;
- If a system call is not correct, jump on exit from system call;
- After a system call handler will finish its work, restore general purposer registers, old stack, flags and return address and exit from the `entry_SYSCALL_64` with the `sysretq` instruction.

That's all.

## Conclusion

This is the end of the second part about the system calls concept in the Linux kernel. In the previous [part](#) we saw theory about this concept from the user application view. In this part we continued to dive into the stuff which is related to the system call concept and saw what the Linux kernel does when a system call occurs.

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [system call](#)
- [write](#)
- [C standard library](#)
- [list of cpu architectures](#)
- [x86\\_64](#)
- [kbuild](#)
- [typedef](#)
- [errno](#)
- [gcc](#)
- [model specific register](#)
- [intel 2b manual](#)
- [coprocessor](#)
- [instruction pointer](#)
- [flags register](#)
- [Global Descriptor Table](#)
- [per-cpu](#)
- [general purpose registers](#)
- [ABI](#)
- [x86\\_64 C ABI](#)
- [previous chapter](#)

# System calls in the Linux kernel. Part 3.

## vsyscalls and vDSO

This is the third part of the [chapter](#) that describes system calls in the Linux kernel and we saw preparations after a system call caused by an userspace application and process of handling of a system call in the previous [part](#). In this part we will look at two concepts that are very close to the system call concept, they are called `vsyscall` and `vdso`.

We already know what is a `system call`. This is special routine in the Linux kernel which userspace application asks to do privileged tasks, like to read or to write to a file, to open a socket and etc. As you may know, invoking a system call is an expensive operation in the Linux kernel, because the processor must interrupt the currently executing task and switch context to kernel mode, subsequently jumping again into userspace after the system call handler finishes its work. These two mechanisms - `vsyscall` and `vdso` are designed to speed up this process for certain system calls and in this part we will try to understand how these mechanisms work.

## Introduction to vsyscalls

The `vsyscall` or `virtual system call` is the first and oldest mechanism in the Linux kernel that is designed to accelerate execution of certain system calls. The principle of work of the `vsyscall` concept is simple. The Linux kernel maps into user space a page that contains some variables and the implementation of some system calls. We can find information about this memory space in the Linux kernel [documentation](#) for the [x86\\_64](#):

```
fffffffffff600000 - ffffffffdfdfdfdf (=8 MB) vsyscalls
```

or:

```
~$ sudo cat /proc/1/maps | grep vsyscall  
fffffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

After this, these system calls will be executed in userspace and this means that there will not be [context switching](#). Mapping of the `vsyscall` page occurs in the `map_vsyscall` function that is defined in the [arch/x86/entry/vsyscall/vsyscall\\_64.c](#) source code file. This function is

called during the Linux kernel initialization in the `setup_arch` function that is defined in the `arch/x86/kernel/setup.c` source code file (we saw this function in the fifth [part](#) of the Linux kernel initialization process chapter).

Note that implementation of the `map_vsyscall` function depends on the `CONFIG_X86_VSYSCALL_EMULATION` kernel configuration option:

```
#ifdef CONFIG_X86_VSYSCALL_EMULATION
extern void map_vsyscall(void);
#else
static inline void map_vsyscall(void) {}
#endif
```

As we can read in the help text, the `CONFIG_X86_VSYSCALL_EMULATION` configuration option:

Enable vsyscall emulation . Why emulate vsyscall ? Actually, the `vsyscall` is a legacy [ABI](#) due to security reasons. Virtual system calls have fixed addresses, meaning that `vsyscall` page is still at the same location every time and the location of this page is determined in the `map_vsyscall` function. Let's look on the implementation of this function:

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

As we can see, at the beginning of the `map_vsyscall` function we get the physical address of the `vsyscall` page with the `__pa_symbol` macro (we already saw implementation of this macro in the fourth [path](#) of the Linux kernel initialization process). The `__vsyscall_page` symbol defined in the `arch/x86/entry/vsyscall/vsyscall_emu_64.S` assembly source code file and have the following [virtual address](#):

```
fffffffff81881000 D __vsyscall_page
```

in the `.data..page_aligned, aw` [section](#) and contains call of the three following system calls:

- `gettimeofday ;`
- `time ;`
- `getcpu .`

Or:

```

__vsyscall_page:
    mov $__NR_gettimeofday, %rax
    syscall
    ret

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

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

```

Let's go back to the implementation of the `map_vsyscall` function and return to the implementation of the `__vsyscall_page`, later. After we receiving the physical address of the `__vsyscall_page`, we check the value of the `vsyscall_mode` variable and set the [fix-mapped](#) address for the `vsyscall` page with the `__set_fixmap` macro:

```

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

```

The `__set_fixmap` takes three arguments: The first is index of the `fixed_addresses` [enum](#). In our case `VSYSCALL_PAGE` is the first element of the `fixed_addresses` enum for the `x86_64` architecture:

```

enum fixed_addresses {
    ...
    ...
    ...
#ifdef CONFIG_X86_VSYSCALL_EMULATION
    VSYSCALL_PAGE = (FIXADDR_TOP - VSYSCALL_ADDR) >> PAGE_SHIFT,
#endif
    ...
    ...
    ...
}

```

It equal to the `511`. The second argument is the physical address of the page that has to be mapped and the third argument is the flags of the page. Note that the flags of the `VSYSCALL_PAGE` depend on the `vsyscall_mode` variable. It will be `PAGE_KERNEL_VSYSCALL` if



the `vsyscall_mode` variable is `NATIVE` and the `PAGE_KERNEL_VVAR` otherwise. Both macros (the `PAGE_KERNEL_VSYSCALL` and the `PAGE_KERNEL_VVAR`) will be expanded to the following flags:

```
#define __PAGE_KERNEL_VSYSCALL    (__PAGE_KERNEL_RX | _PAGE_USER)
#define __PAGE_KERNEL_VVAR       (__PAGE_KERNEL_RO | _PAGE_USER)
```

that represent access rights to the `vsyscall` page. Both flags have the same `_PAGE_USER` flag that means that the page can be accessed by a user-mode process running at lower privilege levels. The second flag depends on the value of the `vsyscall_mode` variable. The first flag (`__PAGE_KERNEL_VSYSCALL`) will be set in the case where `vsyscall_mode` is `NATIVE`. This means virtual system calls will be native `syscall` instructions. In other way the `vsyscall` will have `PAGE_KERNEL_VVAR` if the `vsyscall_mode` variable will be `emulate`. In this case virtual system calls will be turned into traps and are emulated reasonably. The `vsyscall_mode` variable gets its value in the `vsyscall_setup` function:

```
static int __init vsyscall_setup(char *str)
{
    if (str) {
        if (!strcmp("emulate", str))
            vsyscall_mode = EMULATE;
        else if (!strcmp("native", str))
            vsyscall_mode = NATIVE;
        else if (!strcmp("none", str))
            vsyscall_mode = NONE;
        else
            return -EINVAL;

        return 0;
    }

    return -EINVAL;
}
```

That will be called during early kernel parameters parsing:

```
early_param("vsyscall", vsyscall_setup);
```

More about `early_param` macro you can read in the sixth [part](#) of the chapter that describes process of the initialization of the Linux kernel.

In the end of the `vsyscall_map` function we just check that virtual address of the `vsyscall` page is equal to the value of the `VSYSCALL_ADDR` with the `BUILD_BUG_ON` macro:

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

That's all. `vsyscall` page is set up. The result of all the above is the following: If we pass `vsyscall=native` parameter to the kernel command line, virtual system calls will be handled as native `syscall` instructions in the [arch/x86/entry/vsyscall/vsyscall\\_emu\\_64.S](#). The `glibc` knows addresses of the virtual system call handlers. Note that virtual system call handlers are aligned by `1024` (or `0x400`) bytes:

```
__vsyscall_page:
    mov $__NR_gettimeofday, %rax
    syscall
    ret

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

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

And the start address of the `vsyscall` page is the `ffffffff600000` everytime. So, the `glibc` knows the addresses of all virtual system call handlers. You can find definition of these addresses in the `glibc` source code:

```
#define VSYSCALL_ADDR_vgettimeofday    0xffffffff600000
#define VSYSCALL_ADDR_vtime            0xffffffff600400
#define VSYSCALL_ADDR_vgetcpu          0xffffffff600800
```

All virtual system call requests will fall into the `__vsyscall_page` + `VSYSCALL_ADDR_vsyscall_name` offset, put the number of a virtual system call to the `rax` general purpose register and the native for the x86\_64 `syscall` instruction will be executed.

In the second case, if we pass `vsyscall=emulate` parameter to the kernel command line, an attempt to perform virtual system call handler will cause a `page fault` exception. Of course, remember, the `vsyscall` page has `__PAGE_KERNEL_VVAR` access rights that forbid execution. The `do_page_fault` function is the `#PF` or page fault handler. It tries to understand the reason of the last page fault. And one of the reasons can be a situation when a virtual system call is called and `vsyscall` mode is `emulate`. In this case `vsyscall` will be handled by the `emulate_vsyscall` function that is defined in the [arch/x86/entry/vsyscall/vsyscall\\_64.c](#) source code file.

The `emulate_vsyscall` function gets the number of a virtual system call, checks it, prints error and sends [segmentation fault](#) single:

```
...
...
...
vsyscall_nr = addr_to_vsyscall_nr(address);
if (vsyscall_nr < 0) {
    warn_bad_vsyscall(KERN_WARNING, regs, "misaligned vsyscall...");
    goto sigsegv;
}
...
...
...
sigsegv:
    force_sig(SIGSEGV, current);
    reutrn true;
```

As it checked number of a virtual system call, it does some yet another checks like `access_ok` violations and execute system call function depends on the number of a virtual system call:

```
switch (vsyscall_nr) {
    case 0:
        ret = sys_gettimeofday(
            (struct timeval __user *)regs->di,
            (struct timezone __user *)regs->si);
        break;
    ...
    ...
    ...
}
```

In the end we put the result of the `sys_gettimeofday` or another virtual system call handler to the `ax` general purpose register, as we did it with the normal system calls and restore the [instruction pointer](#) register and add `8` bytes to the [stack pointer](#) register. This operation emulates `ret` instruction.

```
regs->ax = ret;

do_ret:
    regs->ip = caller;
    regs->sp += 8;
    return true;
```

That's all. Now let's look on the modern concept - `vDSO`.

# Introduction to vDSO

As I already wrote above, `vsyscall` is an obsolete concept and replaced by the `vDSO` or `virtual dynamic shared object`. The main difference between the `vsyscall` and `vDSO` mechanisms is that `vDSO` maps memory pages into each process in a shared object [form](#), but `vsyscall` is static in memory and has the same address every time. For the `x86_64` architecture it is called - `linux-vdso.so.1`. All userspace applications linked with this shared library via the `glibc`. For example:

```
~$ ldd /bin/uname
linux-vdso.so.1 (0x00007ffe014b7000)
libc.so.6 => /lib64/libc.so.6 (0x00007fbfee2fe000)
/lib64/ld-linux-x86-64.so.2 (0x00005559aab7c000)
```

Or:

```
~$ sudo cat /proc/1/maps | grep vdso
7fff39f73000-7fff39f75000 r-xp 00000000 00:00 0          [vdso]
```

Here we can see that `uname` util was linked with the three libraries:

- `linux-vdso.so.1` ;
- `libc.so.6` ;
- `ld-linux-x86-64.so.2` .

The first provides `vDSO` functionality, the second is `c standard library` and the third is the program interpreter (more about this you can read in the part that describes [linkers](#)). So, the `vDSO` solves limitations of the `vsyscall`. Implementation of the `vDSO` is similar to `vsyscall`.

Initialization of the `vDSO` occurs in the `init_vdso` function that defined in the [arch/x86/entry/vdso/vma.c](#) source code file. This function starts from the initialization of the `vDSO` images for 32-bits and 64-bits depends on the `CONFIG_X86_X32_ABI` kernel configuration option:

```
static int __init init_vdso(void)
{
    init_vdso_image(&vdso_image_64);

#ifdef CONFIG_X86_X32_ABI
    init_vdso_image(&vdso_image_x32);
#endif
}
```

Both function initialize the `vdso_image` structure. This structure is defined in the two generated source code files: the [arch/x86/entry/vdso/vdso-image-64.c](#) and the [arch/x86/entry/vdso/vdso-image-64.c](#). These source code files generated by the `vdso2c` program from the different source code files, represent different approaches to call a system call like `int 0x80`, `sysenter` and etc. The full set of the images depends on the kernel configuration.

For example for the `x86_64` Linux kernel it will contain `vdso_image_64` :

```
#ifdef CONFIG_X86_64
extern const struct vdso_image vdso_image_64;
#endif
```

But for the `x86` - `vdso_image_32` :

```
#ifdef CONFIG_X86_X32
extern const struct vdso_image vdso_image_x32;
#endif
```

If our kernel is configured for the `x86` architecture or for the `x86_64` and compatibility mode, we will have ability to call a system call with the `int 0x80` interrupt, if compatibility mode is enabled, we will be able to call a system call with the native `syscall` instruction or `sysenter` instruction in other way:

```
#if defined CONFIG_X86_32 || defined CONFIG_COMPAT
extern const struct vdso_image vdso_image_32_int80;
#ifdef CONFIG_COMPAT
extern const struct vdso_image vdso_image_32_syscall;
#endif
extern const struct vdso_image vdso_image_32_sysenter;
#endif
```

As we can understand from the name of the `vdso_image` structure, it represents image of the `vdso` for the certain mode of the system call entry. This structure contains information about size in bytes of the `vdso` area that always a multiple of `PAGE_SIZE` ( 4096 bytes), pointer to the text mapping, start and end address of the `alternatives` (set of instructions with better alternatives for the certain type of the processor) and etc. For example `vdso_image_64` looks like this:

```
const struct vdso_image vdso_image_64 = {
    .data = raw_data,
    .size = 8192,
    .text_mapping = {
        .name = "[vdso]",
        .pages = pages,
    },
    .alt = 3145,
    .alt_len = 26,
    .sym_vvar_start = -8192,
    .sym_vvar_page = -8192,
    .sym_hpet_page = -4096,
};
```

Where the `raw_data` contains raw binary code of the 64-bit `vdso` system calls which are 2 page size:

```
static struct page *pages[2];
```

or 8 Kilobytes.

The `init_vdso_image` function is defined in the same source code file and just initializes the `vdso_image.text_mapping.pages`. First of all this function calculates the number of pages and initializes each `vdso_image.text_mapping.pages[number_of_page]` with the `virt_to_page` macro that converts given address to the `page` structure:

```
void __init init_vdso_image(const struct vdso_image *image)
{
    int i;
    int npages = (image->size) / PAGE_SIZE;

    for (i = 0; i < npages; i++)
        image->text_mapping.pages[i] =
            virt_to_page(image->data + i*PAGE_SIZE);
    ...
    ...
    ...
}
```

The `init_vdso` function passed to the `subsys_initcall` macro adds the given function to the `initcalls` list. All functions from this list will be called in the `do_initcalls` function from the `init/main.c` source code file:

```
subsys_initcall(init_vdso);
```

Ok, we just saw initialization of the `vdsO` and initialization of `page` structures that are related to the memory pages that contain `vdsO` system calls. But to where do their pages map? Actually they are mapped by the kernel, when it loads binary to the memory. The Linux kernel calls the `arch_setup_additional_pages` function from the [arch/x86/entry/vdso/vma.c](https://elixir.bootlin.com/linux/v5.10.10/source/arch/x86/entry/vdso/vma.c#L10) source code file that checks that `vdsO` enabled for the `x86_64` and calls the `map_vdsO` function:

```
int arch_setup_additional_pages(struct linux_binprm *bprm, int uses_interp)
{
    if (!vdsO64_enabled)
        return 0;

    return map_vdsO(&vdsO_image_64, true);
}
```

The `map_vdsO` function is defined in the same source code file and maps pages for the `vdsO` and for the shared `vdsO` variables. That's all. The main differences between the `vsyscall` and the `vdsO` concepts is that `vsyscal` has a static address of `ffffffffffff600000` and implements 3 system calls, whereas the `vdsO` loads dynamically and implements four system calls:

- `__vdsO_clock_gettime` ;
- `__vdsO_getcpu` ;
- `__vdsO_gettimeofday` ;
- `__vdsO_time` .

That's all.

## Conclusion

This is the end of the third part about the system calls concept in the Linux kernel. In the previous [part](#) we discussed the implementation of the preparation from the Linux kernel side, before a system call will be handled and implementation of the `exit` process from a system call handler. In this part we continued to dive into the stuff which is related to the system call concept and learned two new concepts that are very similar to the system call - the `vsyscall` and the `vdsO` .

After all of these three parts, we know almost all things that are related to system calls, we know what system call is and why user applications need them. We also know what occurs when a user application calls a system call and how the kernel handles system calls.

The next part will be the last part in this [chapter](#) and we will see what occurs when a user runs the program.

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [x86\\_64 memory map](#)
- [x86\\_64](#)
- [context switching](#)
- [ABI](#)
- [virtual address](#)
- [Segmentation](#)
- [enum](#)
- [fix-mapped addresses](#)
- [glibc](#)
- [BUILD\\_BUG\\_ON](#)
- [Processor register](#)
- [Page fault](#)
- [segmentation fault](#)
- [instruction pointer](#)
- [stack pointer](#)
- [uname](#)
- [Linkers](#)
- [Previous part](#)



# System calls in the Linux kernel. Part 4.

## How does the Linux kernel run a program

This is the fourth part of the [chapter](#) that describes [system calls](#) in the Linux kernel and as I wrote in the conclusion of the [previous](#) - this part will be last in this chapter. In the previous part we stopped at the two new concepts:

- `vsyscall` ;
- `vDSO` ;

that are related and very similar on system call concept.

This part will be last part in this chapter and as you can understand from the part's title - we will see what does occur in the Linux kernel when we run our programs. So, let's start.

## how do we launch our programs?

There are many different ways to launch an application from a user perspective. For example we can run a program from the [shell](#) or double-click on the application icon. It does not matter. The Linux kernel handles application launch regardless how we do launch this application.

In this part we will consider the way when we just launch an application from the shell. As you know, the standard way to launch an application from shell is the following: We just launch a [terminal emulator](#) application and just write the name of the program and pass or not arguments to our program, for example:

```
~$ ls --version
ls (GNU coreutils) 8.23
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Richard M. Stallman and David MacKenzie.
```

Let's consider what does occur when we launch an application from the shell, what does shell do when we write program name, what does Linux kernel do etc. But before we will start to consider these interesting things, I want to warn that this book is about the Linux kernel. That's why we will see Linux kernel insides related stuff mostly in this part. We will not consider in details what does shell do, we will not consider complex cases, for example subshells etc.

My default shell is - `bash`, so I will consider how do bash shell launches a program. So let's start. The `bash` shell as well as any program that written with `C` programming language starts from the `main` function. If you will look on the source code of the `bash` shell, you will find the `main` function in the `shell.c` source code file. This function makes many different things before the main thread loop of the `bash` started to work. For example this function:

- checks and tries to open `/dev/tty` ;
- check that shell running in debug mode;
- parses command line arguments;
- reads shell environment;
- loads `.bashrc` , `.profile` and other configuration files;
- and many many more.

After all of these operations we can see the call of the `reader_loop` function. This function defined in the `eval.c` source code file and represents main thread loop or in other words it reads and executes commands. As the `reader_loop` function made all checks and read the given program name and arguments, it calls the `execute_command` function from the `execute_cmd.c` source code file. The `execute_command` function through the chain of the functions calls:

```
execute_command
--> execute_command_internal
----> execute_simple_command
-----> execute_disk_command
-----> shell_execve
```

makes different checks like do we need to start `subshell` , was it builtin `bash` function or not etc. As I already wrote above, we will not consider all details about things that are not related to the Linux kernel. In the end of this process, the `shell_execve` function calls the `execve` system call:

```
execve (command, args, env);
```

The `execve` system call has the following signature:

```
int execve(const char *filename, char *const argv [], char *const envp[]);
```

and executes a program by the given filename, with the given arguments and [environment variables](#). This system call is the first in our case and only, for example:

```
$ strace ls
execve("/bin/ls", ["ls"], [/* 62 vars */]) = 0

$ strace echo
execve("/bin/echo", ["echo"], [/* 62 vars */]) = 0

$ strace uname
execve("/bin/uname", ["uname"], [/* 62 vars */]) = 0
```

So, a user application ( `bash` in our case) calls the system call and as we already know the next step is Linux kernel.

## execve system call

We saw preparation before a system call called by a user application and after a system call handler finished its work in the second [part](#) of this chapter. We stopped at the call of the `execve` system call in the previous paragraph. This system call defined in the [fs/exec.c](#) source code file and as we already know it takes three arguments:

```
SYSCALL_DEFINE3(execve,
    const char __user *, filename,
    const char __user *const __user *, argv,
    const char __user *const __user *, envp)
{
    return do_execve(getname(filename), argv, envp);
}
```

Implementation of the `execve` is pretty simple here, as we can see it just returns the result of the `do_execve` function. The `do_execve` function defined in the same source code file and do the following things:

- Initialize two pointers on a userspace data with the given arguments and environment variables;
- return the result of the `do_execveat_common`.

We can see its implementation:

```
struct user_arg_ptr argv = { .ptr.native = __argv };
struct user_arg_ptr envp = { .ptr.native = __envp };
return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
```

The `do_execveat_common` function does main work - it executes a new program. This function takes similar set of arguments, but as you can see it takes five arguments instead of three. The first argument is the file descriptor that represent directory with our application, in our case the `AT_FDCWD` means that the given pathname is interpreted relative to the current working directory of the calling process. The fifth argument is flags. In our case we passed `0` to the `do_execveat_common`. We will check in a next step, so will see it latter.

First of all the `do_execveat_common` function checks the `filename` pointer and returns if it is `NULL`. After this we check flags of the current process that limit of running processes is not exceed:

```
if (IS_ERR(filename))
    return PTR_ERR(filename);

if ((current->flags & PF_NPROC_EXCEEDED) &&
    atomic_read(&current_user()->processes) > rlimit(RLIMIT_NPROC)) {
    retval = -EAGAIN;
    goto out_ret;
}

current->flags &= ~PF_NPROC_EXCEEDED;
```

If these two checks were successful we unset `PF_NPROC_EXCEEDED` flag in the flags of the current process to prevent fail of the `execve`. You can see that in the next step we call the `unshare_files` function that defined in the [kernel/fork.c](#) and unshares the files of the current task and check the result of this function:

```
retval = unshare_files(&displaced);
if (retval)
    goto out_ret;
```

We need to call this function to eliminate potential leak of the `execve`'d binary's [file descriptor](#). In the next step we start preparation of the `bprm` that represented by the `struct linux_binprm` structure (defined in the [include/linux/binfmts.h](#) header file). The `linux_binprm` structure is used to hold the arguments that are used when loading binaries. For example it contains `vma` field which has `vm_area_struct` type and represents single memory area over a contiguous interval in a given address space where our application will be loaded, `mm` field which is memory descriptor of the binary, pointer to the top of memory and many other different fields.

First of all we allocate memory for this structure with the `kzalloc` function and check the result of the allocation:

```
bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
if (!bprm)
    goto out_files;
```

After this we start to prepare the `bprm` credentials with the call of the `prepare_bprm_creds` function:

```
retval = prepare_bprm_creds(bprm);
if (retval)
    goto out_free;

check_unsafe_exec(bprm);
current->in_execve = 1;
```

Initialization of the `bprm` credentials in other words is initialization of the `cred` structure that stored inside of the `linux_binprm` structure. The `cred` structure contains the security context of a task for example `real uid` of the task, `real guid` of the task, `uid` and `guid` for the `virtual file system` operations etc. In the next step as we executed preparation of the `bprm` credentials we check that now we can safely execute a program with the call of the `check_unsafe_exec` function and set the current process to the `in_execve` state.

After all of these operations we call the `do_open_execat` function that checks the flags that we passed to the `do_execveat_common` function (remember that we have `0` in the `flags` ) and searches and opens executable file on disk, checks that our we will load a binary file from `noexec` mount points (we need to avoid execute a binary from filesystems that do not contain executable binaries like `proc` or `sysfs`), initializes `file` structure and returns pointer on this structure. Next we can see the call the `sched_exec` after this:

```
file = do_open_execat(fd, filename, flags);
retval = PTR_ERR(file);
if (IS_ERR(file))
    goto out_unmark;

sched_exec();
```

The `sched_exec` function is used to determine the least loaded processor that can execute the new program and to migrate the current process to it.

After this we need to check `file descriptor` of the give executable binary. We try to check does the name of the our binary file starts from the `/` symbol or does the path of the given executable binary is interpreted relative to the current working directory of the calling process or in other words file descriptor is `AT_FDCWD` (read above about this).

If one of these checks is successful we set the binary parameter filename:

```

bprm->file = file;

if (fd == AT_FDCWD || filename->name[0] == '/') {
    bprm->filename = filename->name;
}

```

Otherwise if the filename is empty we set the binary parameter filename to the `/dev/fd/%d` or `/dev/fd/%d/%s` depends on the filename of the given executable binary which means that we will execute the file to which the file descriptor refers:

```

} else {
    if (filename->name[0] == '\0')
        pathbuf = kasprintf(GFP_TEMPORARY, "/dev/fd/%d", fd);
    else
        pathbuf = kasprintf(GFP_TEMPORARY, "/dev/fd/%d/%s",
                            fd, filename->name);

    if (!pathbuf) {
        retval = -ENOMEM;
        goto out_unmark;
    }

    bprm->filename = pathbuf;
}

bprm->interp = bprm->filename;

```

Note that we set not only the `bprm->filename` but also `bprm->interp` that will contain name of the program interpreter. For now we just write the same name there, but later it will be updated with the real name of the program interpreter depends on binary format of a program. You can read above that we already prepared `cred` for the `linux_binprm`. The next step is initialization of other fields of the `linux_binprm`. First of all we call the `bprm_mm_init` function and pass the `bprm` to it:

```

retval = bprm_mm_init(bprm);
if (retval)
    goto out_unmark;

```

The `bprm_mm_init` defined in the same source code file and as we can understand from the function's name, it makes initialization of the memory descriptor or in other words the `bprm_mm_init` function initializes `mm_struct` structure. This structure defined in the [include/linux/mm\\_types.h](#) header file and represents address space of a process. We will not consider implementation of the `bprm_mm_init` function because we do not know many important stuff related to the Linux kernel memory manager, but we just need to know that this function initializes `mm_struct` and populate it with a temporary stack `vm_area_struct`.

After this we calculate the count of the command line arguments which are were passed to the our executable binary, the count of the environment variables and set it to the `bprm->argc` and `bprm->envc` respectively:

```
bprm->argc = count(argv, MAX_ARG_STRINGS);
if ((retval = bprm->argc) < 0)
    goto out;

bprm->envc = count(envp, MAX_ARG_STRINGS);
if ((retval = bprm->envc) < 0)
    goto out;
```

As you can see we do this operations with the help of the `count` function that defined in the [same](#) source code file and calculates the count of strings in the `argv` array. The `MAX_ARG_STRINGS` macro defined in the [include/uapi/linux/binfmts.h](#) header file and as we can understand from the macro's name, it represents maximum number of strings that were passed to the `execve` system call. The value of the `MAX_ARG_STRINGS` :

```
#define MAX_ARG_STRINGS 0x7FFFFFFF
```

After we calculated the number of the command line arguments and environment variables, we call the `prepare_binprm` function. We already call the function with the similar name before this moment. This function is called `prepare_binprm_cred` and we remember that this function initializes `cred` structure in the `linux_bprm` . Now the `prepare_binprm` function:

```
retval = prepare_binprm(bprm);
if (retval < 0)
    goto out;
```

fills the `linux_binprm` structure with the `uid` from [inode](#) and read `128` bytes from the binary executable file. We read only first `128` from the executable file because we need to check a type of our executable. We will read the rest of the executable file in the later step. After the preparation of the `linux_bprm` structure we copy the filename of the executable binary file, command line arguments and environment variables to the `linux_bprm` with the call of the `copy_strings_kernel` function:

```

retval = copy_strings_kernel(1, &bprm->filename, bprm);
if (retval < 0)
    goto out;

retval = copy_strings(bprm->envc, envp, bprm);
if (retval < 0)
    goto out;

retval = copy_strings(bprm->argc, argv, bprm);
if (retval < 0)
    goto out;

```

And set the pointer to the top of new program's stack that we set in the `bprm_mm_init` function:

```
bprm->exec = bprm->p;
```

The top of the stack will contain the program filename and we store this filename to the `exec` field of the `linux_bprm` structure.

Now we have filled `linux_bprm` structure, we call the `exec_binprm` function:

```

retval = exec_binprm(bprm);
if (retval < 0)
    goto out;

```

First of all we store the `pid` and `pid` that seen from the `namespace` of the current task in the `exec_binprm`:

```

old_pid = current->pid;
rcu_read_lock();
old_vpid = task_pid_nr_ns(current, task_active_pid_ns(current->parent));
rcu_read_unlock();

```

and call the:

```
search_binary_handler(bprm);
```

function. This function goes through the list of handlers that contains different binary formats. Currently the Linux kernel supports following binary formats:

- `binfmt_script` - support for interpreted scripts that are starts from the `#!` line;
- `binfmt_misc` - support different binary formats, according to runtime configuration of the Linux kernel;



- `binfmt_elf` - support `elf` format;
- `binfmt_aout` - support `a.out` format;
- `binfmt_flat` - support for `flat` format;
- `binfmt_elf_fdpic` - Support for `elf FDPIC` binaries;
- `binfmt_em86` - support for Intel `elf` binaries running on `Alpha` machines.

So, the `search_binary_handler` tries to call the `load_binary` function and pass `linux_binprm` to it. If the binary handler supports the given executable file format, it starts to prepare the executable binary for execution:

```
int search_binary_handler(struct linux_binprm *bprm)
{
    ...
    ...
    ...
    list_for_each_entry(fmt, &formats, lh) {
        retval = fmt->load_binary(bprm);
        if (retval < 0 && !bprm->mm) {
            force_sigsegv(SIGSEGV, current);
            return retval;
        }
    }

    return retval;
}
```

Where the `load_binary` for example for the `elf` checks the magic number (each `elf` binary file contains magic number in the header) in the `linux_bprm` buffer (remember that we read first 128 bytes from the executable binary file): and exit if it is not `elf` binary:

```
static int load_elf_binary(struct linux_binprm *bprm)
{
    ...
    ...
    ...
    loc->elf_ex = *((struct elfhdr *)bprm->buf);

    if (memcmp(elf_ex.e_ident, ELF_MAGIC, SELF_MAGIC) != 0)
        goto out;
}
```

If the given executable file is in `elf` format, the `load_elf_binary` continues to execute. The `load_elf_binary` does many different things to prepare on execution executable file. For example it checks the architecture and type of the executable file:

```

if (loc->elf_ex.e_type != ET_EXEC && loc->elf_ex.e_type != ET_DYN)
    goto out;
if (!elf_check_arch(&loc->elf_ex))
    goto out;

```

and exit if there is wrong architecture and executable file non executable non shared. Tries to load the `program header table` :

```

elf_phdata = load_elf_phdrs(&loc->elf_ex, bprm->file);
if (!elf_phdata)
    goto out;

```

that describes [segments](#). Read the `program interpreter` and libraries that linked with the our executable binary file from disk and load it to memory. The `program interpreter` specified in the `.interp` section of the executable file and as you can read in the part that describes [Linkers](#) it is - `/lib64/ld-linux-x86-64.so.2` for the `x86_64` . It setups the stack and map `elf` binary into the correct location in memory. It maps the [bss](#) and the [brk](#) sections and does many many other different things to prepare executable file to execute.

In the end of the execution of the `load_elf_binary` we call the `start_thread` function and pass three arguments to it:

```

start_thread(regs, elf_entry, bprm->p);
retval = 0;
out:
    kfree(loc);
out_ret:
    return retval;

```

These arguments are:

- Set of [registers](#) for the new task;
- Address of the entry point of the new task;
- Address of the top of the stack for the new task.

As we can understand from the function's name, it starts new thread, but it is not so. The `start_thread` function just prepares new task's registers to be ready to run. Let's look on the implementation of this function:

```
void
start_thread(struct pt_regs *regs, unsigned long new_ip, unsigned long new_sp)
{
    start_thread_common(regs, new_ip, new_sp,
                        __USER_CS, __USER_DS, 0);
}
```

As we can see the `start_thread` function just makes a call of the `start_thread_common` function that will do all for us:

```
static void
start_thread_common(struct pt_regs *regs, unsigned long new_ip,
                   unsigned long new_sp,
                   unsigned int _cs, unsigned int _ss, unsigned int _ds)
{
    loadsegment(fs, 0);
    loadsegment(es, _ds);
    loadsegment(ds, _ds);
    load_gs_index(0);
    regs->ip      = new_ip;
    regs->sp      = new_sp;
    regs->cs      = _cs;
    regs->ss      = _ss;
    regs->flags    = X86_EFLAGS_IF;
    force_iret();
}
```

The `start_thread_common` function fills `fs` segment register with zero and `es` and `ds` with the value of the data segment register. After this we set new values to the [instruction pointer](#), `cs` segments etc. In the end of the `start_thread_common` function we can see the `force_iret` macro that force a system call return via `iret` instruction. Ok, we prepared new thread to run in userspace and now we can return from the `exec_binprm` and now we are in the `do_execveat_common` again. After the `exec_binprm` will finish its execution we release memory for structures that was allocated before and return.

After we returned from the `execve` system call handler, execution of our program will be started. We can do it, because all context related information already configured for this purpose. As we saw the `execve` system call does not return control to a process, but code, data and other segments of the caller process are just overwritten of the program segments. The exit from our application will be implemented through the `exit` system call.

That's all. From this point our program will be executed.

## Conclusion

This is the end of the fourth and last part of the about the system calls concept in the Linux kernel. We saw almost all related stuff to the `system call` concept in these four parts. We started from the understanding of the `system call` concept, we have learned what is it and why do users applications need in this concept. Next we saw how does the Linux handle a system call from a user application. We met two similar concepts to the `system call` concept, they are `vsyscall` and `vDSO` and finally we saw how does Linux kernel run a user program.

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [System call](#)
- [shell](#)
- [bash](#)
- [entry point](#)
- [C](#)
- [environment variables](#)
- [file descriptor](#)
- [real uid](#)
- [virtual file system](#)
- [procfs](#)
- [sysfs](#)
- [inode](#)
- [pid](#)
- [namespace](#)
- [#!](#)
- [elf](#)
- [a.out](#)
- [flat](#)
- [Alpha](#)
- [FDPIC](#)
- [segments](#)
- [Linkers](#)
- [Processor register](#)
- [instruction pointer](#)

- [Previous part](#)

# Timers and time management

This chapter describes timers and time management related concepts in the linux kernel.

- [Introduction](#) - this part is introduction to the timers in the Linux kernel.
- [Introduction to the clocksource framework](#) - this part describes `clocksource` framework in the Linux kernel.
- [The tick broadcast framework and dyntick](#) - this part describes tick broadcast framework and dyntick concept.
- [Introduction to timers](#) - this chapter describes timers in the Linux kernel.
- [Introduction to the clockevents framework](#) - this part describes yet another clock/time management related framework - `clockevents` .
- [x86 related clock sources](#) - this part describes `x86_64` related clock sources.
- [Time related system calls in the Linux kernel](#) - this part describes time related system calls.

# Timers and time management in the Linux kernel. Part 1.

## Introduction

This is yet another post that opens new chapter in the [linux-insides](#) book. The previous [part](#) was a list part of the chapter that describes [system call](#) concept and now time is to start new chapter. As you can understand from the post's title, this chapter will be devoted to the `timers` and `time management` in the Linux kernel. The choice of topic for the current chapter is not accidental. Timers and generally time management are very important and widely used in the Linux kernel. The Linux kernel uses timers for various tasks, different timeouts for example in [TCP](#) implementation, the kernel must know current time, scheduling asynchronous functions, next event interrupt scheduling and many many more.

So, we will start to learn implementation of the different time management related stuff in this part. We will see different types of timers and how do different Linux kernel subsystems use them. As always we will start from the earliest part of the Linux kernel and will go through initialization process of the Linux kernel. We already did it in the special [chapter](#) which describes initialization process of the Linux kernel, but as you may remember we missed some things there. And one of them is the initialization of timers.

Let's start.

## Initialization of non-standard PC hardware clock

After the Linux kernel was decompressed (more about this you can read in the [Kernel decompression](#) part) the architecture non-specific code starts to work in the `init/main.c` source code file. After initialization of the [lock validator](#), initialization of [cgroups](#) and setting [canary](#) value we can see the call of the `setup_arch` function.

As you may remember this function defined in the [arch/x86/kernel/setup.c](#) source code file and prepares/initializes architecture-specific stuff (for example it reserves place for [bss](#) section, reserves place for [initrd](#), parses kernel command line and many many other things). Besides this, we can find some time management related functions there.

The first is:

```
x86_init.timers.wallclock_init();
```

We already saw `x86_init` structure in the chapter that describes initialization of the Linux kernel. This structure contains pointers to the default setup functions for the different platforms like [Intel MID](#), [Intel CE4100](#) and etc. The `x86_init` structure defined in the [arch/x86/kernel/x86\\_init.c](#) and as you can see it determines standard PC hardware by default.

As we can see, the `x86_init` structure has `x86_init_ops` type that provides a set of functions for platform specific setup like reserving standard resources, platform specific memory setup, initialization of interrupt handlers and etc. This structure looks like:

```
struct x86_init_ops {
    struct x86_init_resources    resources;
    struct x86_init_mpparse     mpparse;
    struct x86_init_irqs        irq;
    struct x86_init_oem          oem;
    struct x86_init_paging       paging;
    struct x86_init_timers       timers;
    struct x86_init_iommu        iommu;
    struct x86_init_pci          pci;
};
```

We can note `timers` field that has `x86_init_timers` type and as we can understand by its name - this field is related to time management and timers. The `x86_init_timers` contains four fields which are all functions that returns pointer on [void](#):

- `setup_percpu_clockev` - set up the per cpu clock event device for the boot cpu;
- `tsc_pre_init` - platform function called before [TSC](#) init;
- `timer_init` - initialize the platform timer;
- `wallclock_init` - initialize the wallclock device.

So, as we already know, in our case the `wallclock_init` executes initialization of the wallclock device. If we will look on the `x86_init` structure, we will see that `wallclock_init` points to the `x86_init_noop` :



```
struct x86_init_ops x86_init __initdata = {
    ...
    ...
    ...
    .timers = {
        .wallclock_init      = x86_init_noop,
    },
    ...
    ...
    ...
}
```

Where the `x86_init_noop` is just a function that does nothing:

```
void __cpuinit x86_init_noop(void) { }
```

for the standard PC hardware. Actually, the `wallclock_init` function is used in the [Intel MID](#) platform. Initialization of the `x86_init.timers.wallclock_init` located in the [arch/x86/platform/intel-mid/intel-mid.c](#) source code file in the `x86_intel_mid_early_setup` function:

```
void __init x86_intel_mid_early_setup(void)
{
    ...
    ...
    ...
    x86_init.timers.wallclock_init = intel_mid_rtc_init;
    ...
    ...
    ...
}
```

Implementation of the `intel_mid_rtc_init` function is in the [arch/x86/platform/intel-mid/intel\\_mid\\_vrtc.c](#) source code file and looks pretty easy. First of all, this function parses [Simple Firmware Interface](#) M-Real-Time-Clock table for the getting such devices to the `sfi_mrtc_array` array and initialization of the `set_time` and `get_time` functions:

```

void __init intel_mid_rtc_init(void)
{
    unsigned long vrtc_paddr;

    sfi_table_parse(SFI_SIG_MRTC, NULL, NULL, sfi_parse_mrtc);

    vrtc_paddr = sfi_mrtc_array[0].phys_addr;
    if (!sfi_mrtc_num || !vrtc_paddr)
        return;

    vrtc_virt_base = (void __iomem *)set_fixmap_offset_nocache(FIX_LNW_VRTC,
  vrtc_paddr);

    x86_platform.get_wallclock = vrtc_get_time;
    x86_platform.set_wallclock = vrtc_set_mmss;
}

```

That's all, after this a device based on `Intel MID` will be able to get time from hardware clock. As I already wrote, the standard PC `x86_64` architecture does not support `x86_init_noop` and just do nothing during call of this function. We just saw initialization of the [real time clock](#) for the `Intel MID` architecture and now times to return to the general `x86_64` architecture and will look on the time management related stuff there.

## Acquainted with jiffies

If we will return to the `setup_arch` function which is located as you remember in the [arch/x86/kernel/setup.c](#) source code file, we will see the next call of the time management related function:

```
register_refined_jiffies(CLOCK_TICK_RATE);
```

Before we will look on the implementation of this function, we must know about [jiffy](#). As we can read on wikipedia:

```
Jiffy is an informal term for any unspecified short period of time
```

This definition is very similar to the `jiffy` in the Linux kernel. There is global variable with the `jiffies` which holds the number of ticks that have occurred since the system booted. The Linux kernel sets this variable to zero:

```
extern unsigned long volatile __jiffy_data jiffies;
```

during initialization process. This global variable will be increased each time during timer interrupt. Besides this, near the `jiffies` variable we can see definition of the similar variable

```
extern u64 jiffies_64;
```

Actually only one of these variables is in use in the Linux kernel. And it depends on the processor type. For the `x86_64` it will be `u64` use and for the `x86` is `unsigned long`. We will see this if we will look on the [arch/x86/kernel/vmlinux.lds.S](#) linker script:

```
#ifdef CONFIG_X86_32
...
jiffies = jiffies_64;
...
#else
...
jiffies_64 = jiffies;
...
#endif
```

In the case of `x86_32` the `jiffies` will be lower `32` bits of the `jiffies_64` variable. Schematically, we can imagine it as follows



Now we know a little theory about `jiffies` and we can return to the our function. There is no architecture-specific implementation for our function - the `register_refined_jiffies`. This function located in the generic kernel code - [kernel/time/jiffies.c](#) source code file. Main point of the `register_refined_jiffies` is registration of the jiffy `clocksource`. Before we will look on the implementation of the `register_refined_jiffies` function, we must know what is it `clocksource`. As we can read in the comments:

```
The `clocksource` is hardware abstraction for a free-running counter.
```

I'm not sure about you, but that description didn't give a good understanding about the `clocksource` concept. Let's try to understand what is it, but we will not go deeper because this topic will be described in a separate part in much more detail. The main point of the `clocksource` is timekeeping abstraction or in very simple words - it provides a time value to the kernel. We already know about `jiffies` interface that represents number of ticks that have occurred since the system booted. It represented by the global variable in the Linux kernel and increased each timer interrupt. The Linux kernel can use `jiffies` for time measurement. So why do we need in separate context like the `clocksource` ? Actually different hardware devices provide different clock sources that are widely in their capabilities. The availability of more precise techniques for time intervals measurement is hardware-dependent.

For example `x86` has on-chip a 64-bit counter that is called [Time Stamp Counter](#) and its frequency can be equal to processor frequency. Or for example [High Precision Event Timer](#) that consists of a `64-bit` counter of at least `10 MHz` frequency. Two different timers and they are both for `x86` . If we will add timers from other architectures, this only makes this problem more complex. The Linux kernel provides `clocksource` concept to solve the problem.

The `clocksource` concept represented by the `clocksource` structure in the Linux kernel. This structure defined in the [include/linux/clocksource.h](#) header file and contains a couple of fields that describe a time counter. For example it contains - `name` field which is the name of a counter, `flags` field that describes different properties of a counter, pointers to the `suspend` and `resume` functions, and many more.

Let's look on the `clocksource` structure for `jiffies` that defined in the [kernel/time/jiffies.c](#) source code file:

```
static struct clocksource clocksource_jiffies = {
    .name      = "jiffies",
    .rating     = 1,
    .read      = jiffies_read,
    .mask      = 0xffffffff,
    .mult      = NSEC_PER_JIFFY << JIFFIES_SHIFT,
    .shift     = JIFFIES_SHIFT,
    .max_cycles = 10,
};
```

We can see definition of the default name here - `jiffies` , the next is `rating` field allows the best registered clock source to be chosen by the clock source management code available for the specified hardware. The `rating` may have following value:

- `1-99` - Only available for bootup and testing purposes;
- `100-199` - Functional for real use, but not desired.

- 200-299 - A correct and usable clocksource.
- 300-399 - A reasonably fast and accurate clocksource.
- 400-499 - The ideal clocksource. A must-use where available;

For example rating of the [time stamp counter](#) is 300, but rating of the [high precision event timer](#) is 250. The next field is `read` - is pointer to the function that allows to read clocksource's cycle value or in other words it just returns `jiffies` variable with `cycle_t` type:

```
static cycle_t jiffies_read(struct clocksource *cs)
{
    return (cycle_t) jiffies;
}
```

that is just 64-bit unsigned type:

```
typedef u64 cycle_t;
```

The next field is the `mask` value ensures that subtraction between counters values from non 64 bit counters do not need special overflow logic. In our case the mask is `0xffffffff` and it is 32 bits. This means that `jiffy` wraps around to zero after 42 seconds:

```
>>> 0xffffffff
4294967295
# 42 nanoseconds
>>> 42 * pow(10, -9)
4.20000000000000006e-08
# 43 nanoseconds
>>> 43 * pow(10, -9)
4.3e-08
```

The next two fields `mult` and `shift` are used to convert the clocksource's period to nanoseconds per cycle. When the kernel calls the `clocksource.read` function, this function returns value in `machine` time units represented with `cycle_t` data type that we saw just now. To convert this return value to the [nanoseconds](#) we need in these two fields: `mult` and `shift`. The `clocksource` provides `clocksource_cyc2ns` function that will do it for us with the following expression:

```
((u64) cycles * mult) >> shift;
```

As we can see the `mult` field is equal:

```
NSEC_PER_JIFFY << JIFFIES_SHIFT

#define NSEC_PER_JIFFY ((NSEC_PER_SEC+HZ/2)/HZ)
#define NSEC_PER_SEC    1000000000L
```

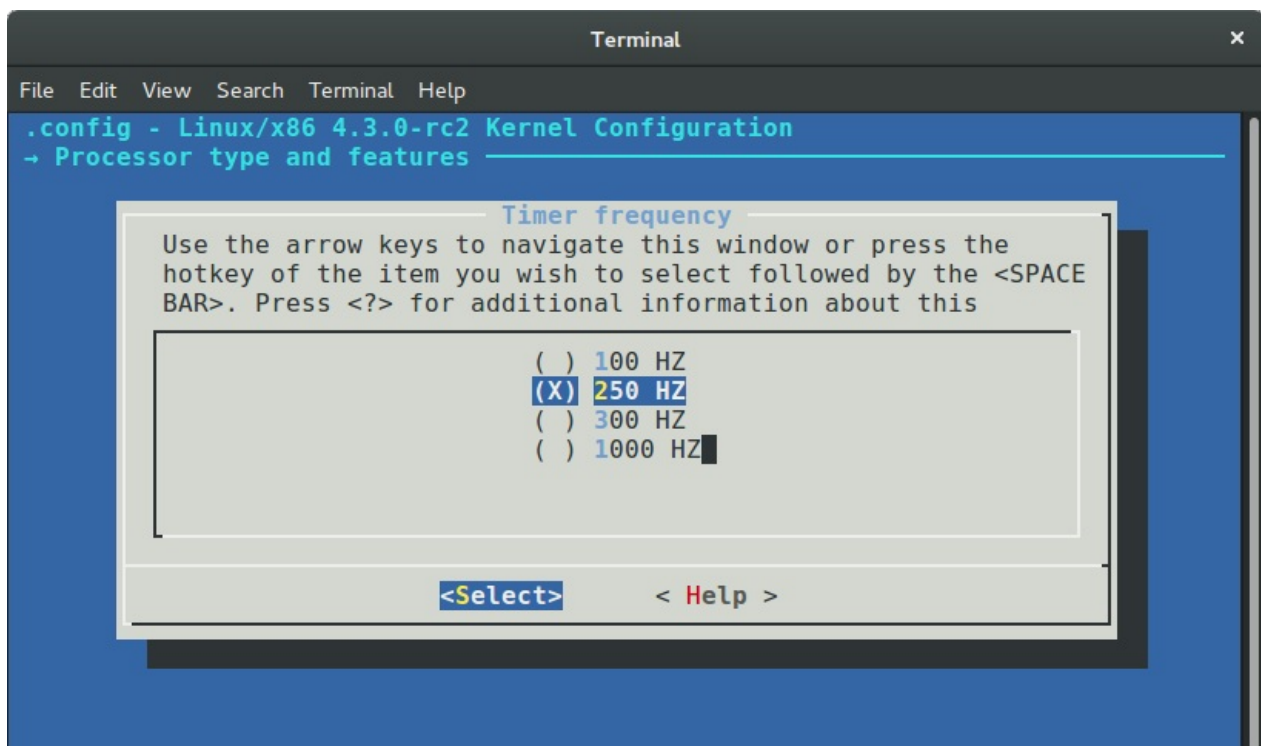
by default, and the `shift` is

```
#if HZ < 34
    #define JIFFIES_SHIFT    6
#elif HZ < 67
    #define JIFFIES_SHIFT    7
#else
    #define JIFFIES_SHIFT    8
#endif
```

The `jiffies` clock source uses the `NSEC_PER_JIFFY` multiplier conversion to specify the nanosecond over cycle ratio. Note that values of the `JIFFIES_SHIFT` and `NSEC_PER_JIFFY` depend on `HZ` value. The `HZ` represents the frequency of the system timer. This macro defined in the [include/asm-generic/param.h](#) and depends on the `CONFIG_HZ` kernel configuration option. The value of `HZ` differs for each supported architecture, but for `x86` it's defined like:

```
#define HZ    CONFIG_HZ
```

Where `CONFIG_HZ` can be one of the following values:



This means that in our case the timer interrupt frequency is `250 HZ` or occurs `250` times per second or one timer interrupt each `4ms`.

The last field that we can see in the definition of the `clocksource_jiffies` structure is the `max_cycles` that holds the maximum cycle value that can safely be multiplied without potentially causing an overflow.

Ok, we just saw definition of the `clocksource_jiffies` structure, also we know a little

```
register_refined_jiffies(CLOCK_TICK_RATE);
```

function from the [arch/x86/kernel/setup.c](#) source code file.

As I already wrote, the main purpose of the `register_refined_jiffies` function is to register `refined_jiffies` clocksource. We already saw the `clocksource_jiffies` structure represents standard `jiffies` clock source. Now, if you look in the [kernel/time/jiffies.c](#) source code file, you will find yet another clock source definition:

```
struct clocksource refined_jiffies;
```

There is one different between `refined_jiffies` and `clocksource_jiffies`: The standard `jiffies` based clock source is the lowest common denominator clock source which should function on all systems. As we already know, the `jiffies` global variable will be increased during each timer interrupt. This means that standard `jiffies` based clock source has the same resolution as the timer interrupt frequency. From this we can understand that standard `jiffies` based clock source may suffer from inaccuracies. The `refined_jiffies` uses `CLOCK_TICK_RATE` as the base of `jiffies` shift.

Let's look on the implementation of this function. First of all we can see that the `refined_jiffies` clock source based on the `clocksource_jiffies` structure:

```
int register_refined_jiffies(long cycles_per_second)
{
    u64 nsec_per_tick, shift_hz;
    long cycles_per_tick;

    refined_jiffies = clocksource_jiffies;
    refined_jiffies.name = "refined-jiffies";
    refined_jiffies.rating++;
    ...
    ...
    ...
}
```

Here we can see that we update the name of the `refined_jiffies` to `refined-jiffies` and increase the rating of this structure. As you remember, the `clocksource_jiffies` has rating - 1, so our `refined_jiffies` clocksource will have rating - 2. This means that the `refined_jiffies` will be best selection for clock source management code.

In the next step we need to calculate number of cycles per one tick:

```
cycles_per_tick = (cycles_per_second + HZ/2)/HZ;
```

Note that we have used `NSEC_PER_SEC` macro as the base of the standard `jiffies` multiplier. Here we are using the `cycles_per_second` which is the first parameter of the `register_refined_jiffies` function. We've passed the `CLOCK_TICK_RATE` macro to the `register_refined_jiffies` function. This macro defined in the [arch/x86/include/asm/timex.h](https://elixir.bootlin.com/linux/v4.18.202/source/arch/x86/include/asm/timex.h) header file and expands to the:

```
#define CLOCK_TICK_RATE      PIT_TICK_RATE
```

where the `PIT_TICK_RATE` macro expands to the frequency of the [Intel 8253](https://www.intel.com/content/www/us/en/processors/atom/atom-processor-family-32-bit-32-core-atom-c3000-series-processor.html):

```
#define PIT_TICK_RATE 1193182u1
```

After this we calculate `shift_hz` for the `register_refined_jiffies` that will store `hz << 8` or in other words frequency of the system timer. We shift left the `cycles_per_second` or frequency of the programmable interval timer on 8 in order to get extra accuracy:

```
shift_hz = (u64)cycles_per_second << 8;
shift_hz += cycles_per_tick/2;
do_div(shift_hz, cycles_per_tick);
```

In the next step we calculate the number of seconds per one tick by shifting left the `NSEC_PER_SEC` on 8 too as we did it with the `shift_hz` and do the same calculation as before:

```
nsec_per_tick = (u64)NSEC_PER_SEC << 8;
nsec_per_tick += (u32)shift_hz/2;
do_div(nsec_per_tick, (u32)shift_hz);
```

```
refined_jiffies.mult = ((u32)nsec_per_tick) << JIFFIES_SHIFT;
```



In the end of the `register_refined_jiffies` function we register new clock source with the `__clocksource_register` function that defined in the [include/linux/clocksource.h](#) header file and return:

```
__clocksource_register(&refined_jiffies);  
return 0;
```

The clock source management code provides the API for clock source registration and selection. As we can see, clock sources are registered by calling the `__clocksource_register` function during kernel initialization or from a kernel module. During registration, the clock source management code will choose the best clock source available in the system using the `clocksource.rating` field which we already saw when we initialized `clocksource` structure for `jiffies`.

## Using the jiffies

We just saw initialization of two `jiffies` based clock sources in the previous paragraph:

- standard `jiffies` based clock source;
- refined `jiffies` based clock source;

Don't worry if you don't understand the calculations here. They look frightening at first. Soon, step by step we will learn these things. So, we just saw initialization of `jiffies` based clock sources and also we know that the Linux kernel has the global variable `jiffies` that holds the number of ticks that have occurred since the kernel started to work. Now, let's look how to use it. To use `jiffies` we just can use `jiffies` global variable by its name or with the call of the `get_jiffies_64` function. This function defined in the [kernel/time/jiffies.c](#) source code file and just returns full 64-bit value of the `jiffies`:

```
u64 get_jiffies_64(void)  
{  
    unsigned long seq;  
    u64 ret;  
  
    do {  
        seq = read_seqbegin(&jiffies_lock);  
        ret = jiffies_64;  
    } while (read_seqretry(&jiffies_lock, seq));  
    return ret;  
}  
EXPORT_SYMBOL(get_jiffies_64);
```

Note that the `get_jiffies_64` function does not implemented as `jiffies_read` for example:

```
static cycle_t jiffies_read(struct clocksource *cs)
{
    return (cycle_t) jiffies;
}
```

We can see that implementation of the `get_jiffies_64` is more complex. The reading of the `jiffies_64` variable is implemented using [seqlocks](#). Actually this is done for machines that cannot atomically read the full 64-bit values.

If we can access the `jiffies` or the `jiffies_64` variable we can convert it to `human` time units. To get one second we can use following expression:

```
jiffies / HZ
```

So, if we know this, we can get any time units. For example:

```
/* Thirty seconds from now */
jiffies + 30*HZ

/* Two minutes from now */
jiffies + 120*HZ

/* One millisecond from now */
jiffies + HZ / 1000
```

That's all.

## Conclusion

This concludes the first part covering time and time management related concepts in the Linux kernel. We met first two concepts and its initialization in this part: `jiffies` and `clocksource`. In the next part we will continue to dive into this interesting theme and as I already wrote in this part we will acquainted and try to understand insides of these and other time management concepts in the Linux kernel.

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [system call](#)
- [TCP](#)
- [lock validator](#)
- [cgroups](#)
- [bss](#)
- [initrd](#)
- [Intel MID](#)
- [TSC](#)
- [void](#)
- [Simple Firmware Interface](#)
- [x86\\_64](#)
- [real time clock](#)
- [Jiffy](#)
- [high precision event timer](#)
- [nanoseconds](#)
- [Intel 8253](#)
- [seqlocks](#)
- [clocksource documentation](#)
- [Previous chapter](#)

# Timers and time management in the Linux kernel. Part 2.

## Introduction to the `clocksource` framework

The previous [part](#) was the first part in the current [chapter](#) that describes timers and time management related stuff in the Linux kernel. We got acquainted with two concepts in the previous part:

- `jiffies`
- `clocksource`

The first is the global variable that is defined in the [include/linux/jiffies.h](#) header file and represents the counter that is increased during each timer interrupt. So if we can access this global variable and we know the timer interrupt rate we can convert `jiffies` to the human time units. As we already know the timer interrupt rate represented by the compile-time constant that is called `HZ` in the Linux kernel. The value of `HZ` is equal to the value of the `CONFIG_HZ` kernel configuration option and if we will look into the [arch/x86/configs/x86\\_64\\_defconfig](#) kernel configuration file, we will see that:

```
CONFIG_HZ_1000=y
```

kernel configuration option is set. This means that value of `CONFIG_HZ` will be `1000` by default for the [x86\\_64](#) architecture. So, if we divide the value of `jiffies` by the value of `HZ`:

```
jiffies / HZ
```

we will get the amount of seconds that elapsed since the beginning of the moment the Linux kernel started to work or in other words we will get the system [uptime](#). Since `HZ` represents the amount of timer interrupts in a second, we can set a value for some time in the future. For example:

```
/* one minute from now */
unsigned long later = jiffies + 60*HZ;

/* five minutes from now */
unsigned long later = jiffies + 5*60*HZ;
```

This is a very common practice in the Linux kernel. For example, if you will look into the [arch/x86/kernel/smpboot.c](#) source code file, you will find the `do_boot_cpu` function. This function boots all processors besides bootstrap processor. You can find a snippet that waits ten seconds for a response from the application processor:

```
if (!boot_error) {
    timeout = jiffies + 10*HZ;
    while (time_before(jiffies, timeout)) {
        ...
        ...
        ...
        udelay(100);
    }
    ...
    ...
    ...
}
```

We assign `jiffies + 10*HZ` value to the `timeout` variable here. As I think you already understood, this means a ten seconds timeout. After this we are entering a loop where we use the `time_before` macro to compare the current `jiffies` value and our timeout.

Or for example if we look into the [sound/isa/sscape.c](#) source code file which represents the driver for the [Ensoniq Soundscape Elite](#) sound card, we will see the `obp_startup_ack` function that waits upto a given timeout for the On-Board Processor to return its start-up acknowledgement sequence:

```
static int obp_startup_ack(struct soundscape *s, unsigned timeout)
{
    unsigned long end_time = jiffies + msecs_to_jiffies(timeout);

    do {
        ...
        ...
        ...
        x = host_read_unsafe(s->io_base);
        ...
        ...
        ...
        if (x == 0xfe || x == 0xff)
            return 1;
        msleep(10);
    } while (time_before(jiffies, end_time));

    return 0;
}
```

As you can see, the `jiffies` variable is very widely used in the Linux kernel [code](#). As I already wrote, we met yet another new time management related concept in the previous part - `clocksource`. We have only seen a short description of this concept and the API for a clock source registration. Let's take a closer look in this part.

## Introduction to `clocksource`

The `clocksource` concept represents the generic API for clock sources management in the Linux kernel. Why do we need a separate framework for this? Let's go back to the beginning. The `time` concept is the fundamental concept in the Linux kernel and other operating system kernels. And the timekeeping is one of the necessities to use this concept. For example Linux kernel must know and update the time elapsed since system startup, it must determine how long the current process has been running for every processor and many many more. Where the Linux kernel can get information about time? First of all it is Real Time Clock or [RTC](#) that represents by the a nonvolatile device. You can find a set of architecture-independent real time clock drivers in the Linux kernel in the [drivers rtc](#) directory. Besides this, each architecture can provide a driver for the architecture-dependent real time clock, for example - CMOS/RTC - [arch/x86/kernel/rtc.c](#) for the [x86](#) architecture. The second is system timer - timer that excites [interrupts](#) with a periodic rate. For example, for [IBM PC](#) compatibles it was - [programmable interval timer](#).

We already know that for timekeeping purposes we can use `jiffies` in the Linux kernel. The `jiffies` can be considered as read only global variable which is updated with `HZ` frequency. We know that the `HZ` is a compile-time kernel parameter whose reasonable range is from `100` to `1000` [Hz](#). So, it is guaranteed to have an interface for time measurement with `1` - `10` milliseconds resolution. Besides standard `jiffies`, we saw the `refined_jiffies` clock source in the previous part that is based on the `i8253/i8254` [programmable interval timer](#) tick rate which is almost `1193182` hertz. So we can get something about `1` microsecond resolution with the `refined_jiffies`. In this time, [nanoseconds](#) are the favorite choice for the time value units of the given clock source.

The availability of more precise techniques for time intervals measurement is hardware-dependent. We just knew a little about `x86` dependent timers hardware. But each architecture provides own timers hardware. Earlier each architecture had own implementation for this purpose. Solution of this problem is an abstraction layer and associated API in a common code framework for managing various clock sources and independent of the timer interrupt. This common code framework became - `clocksource` framework.

Generic timeofday and clock source management framework moved a lot of timekeeping code into the architecture independent portion of the code, with the architecture-dependent portion reduced to defining and managing low-level hardware pieces of clocksources. It takes a large amount of funds to measure the time interval on different architectures with different hardware, and it is very complex. Implementation of the each clock related service is strongly associated with an individual hardware device and as you can understand, it results in similar implementations for different architectures.

Within this framework, each clock source is required to maintain a representation of time as a monotonically increasing value. As we can see in the Linux kernel code, nanoseconds are the favorite choice for the time value units of a clock source in this time. One of the main point of the clock source framework is to allow an user to select clock source among a range of available hardware devices supporting clock functions when configuring the system and selecting, accessing and scaling different clock sources.

## The clocksource structure

The fundamental of the `clocksource` framework is the `clocksource` structure that defined in the [include/linux/clocksource.h](#) header file. We already saw some fields that are provided by the `clocksource` strucutre in the previous [part](#). Let's look on the full definition of this structure and try to describe all of its fields:

```

struct clocksource {
    cycle_t (*read)(struct clocksource *cs);
    cycle_t mask;
    u32 mult;
    u32 shift;
    u64 max_idle_ns;
    u32 maxadj;
#ifdef CONFIG_ARCH_CLOCKSOURCE_DATA
    struct arch_clocksource_data archdata;
#endif
    u64 max_cycles;
    const char *name;
    struct list_head list;
    int rating;
    int (*enable)(struct clocksource *cs);
    void (*disable)(struct clocksource *cs);
    unsigned long flags;
    void (*suspend)(struct clocksource *cs);
    void (*resume)(struct clocksource *cs);
#ifdef CONFIG_CLOCKSOURCE_WATCHDOG
    struct list_head wd_list;
    cycle_t cs_last;
    cycle_t wd_last;
#endif
    struct module *owner;
} ____cacheline_aligned;

```

We already saw the first field of the `clocksource` structure in the previous part - it is pointer to the `read` function that returns best counter selected by the clocksource framework. For example we use `jiffies_read` function to read `jiffies` value:

```

static struct clocksource clocksource_jiffies = {
    ...
    .read      = jiffies_read,
    ...
}

```

where `jiffies_read` just returns:

```

static cycle_t jiffies_read(struct clocksource *cs)
{
    return (cycle_t) jiffies;
}

```

Or the `read_tsc` function:



```
static struct clocksource clocksource_tsc = {
    ...
    .read          = read_tsc,
    ...
};
```

for the [time stamp counter](#) reading.

The next field is `mask` that allows to ensure that subtraction between counters values from non `64 bit` counters do not need special overflow logic. After the `mask` field, we can see two fields: `mult` and `shift`. These are the fields that are base of mathematical functions that are provide ability to convert time values specific to each clock source. In other words these two fields help us to convert an abstract machine time units of a counter to nanoseconds.

After these two fields we can see the `64 bits max_idle_ns` field represents max idle time permitted by the clocksource in nanoseconds. We need in this field for the Linux kernel with enabled `CONFIG_NO_HZ` kernel configuration option. This kernel configuration option enables the Linux kernel to run without a regular timer tick (we will see full explanation of this in other part). The problem that dynamic tick allows the kernel to sleep for periods longer than a single tick, moreover sleep time could be unlimited. The `max_idle_ns` field represents this sleeping limit.

The next field after the `max_idle_ns` is the `maxadj` field which is the maximum adjustment value to `mult`. The main formula by which we convert cycles to the nanoseconds:

```
((u64) cycles * mult) >> shift;
```

is not `100%` accurate. Instead the number is taken as close as possible to a nanosecond and `maxadj` helps to correct this and allows clocksource API to avoid `mult` values that might overflow when adjusted. The next four fields are pointers to the function:

- `enable` - optional function to enable clocksource;
- `disable` - optional function to disable clocksource;
- `suspend` - suspend function for the clocksource;
- `resume` - resume function for the clocksource;

The next field is the `max_cycles` and as we can understand from its name, this field represents maximum cycle value before potential overflow. And the last field is `owner` represents reference to a kernel [module](#) that is owner of a clocksource. This is all. We just went through all the standard fields of the `clocksource` structure. But you can noted that we missed some fields of the `clocksource` structure. We can divide all of missed field on two types: Fields of the first type are already known for us. For example, they are `name` field

that represents name of a `clocksource`, the `rating` field that helps to the Linux kernel to select the best clocksource and etc. The second type, fields which are dependent from the different Linux kernel configuration options. Let's look on these fields.

The first field is the `archdata`. This field has `arch_clocksource_data` type and depends on the `CONFIG_ARCH_CLOCKSOURCE_DATA` kernel configuration option. This field is actual only for the `x86` and `IA64` architectures for this moment. And again, as we can understand from the field's name, it represents architecture-specific data for a clock source. For example, it represents `vds0` clock mode:

```
struct arch_clocksource_data {  
    int vclock_mode;  
};
```

for the `x86` architectures. Where the `vds0` clock mode can be one of the:

```
#define VCLOCK_NONE 0  
#define VCLOCK_TSC 1  
#define VCLOCK_HPET 2  
#define VCLOCK_PVCLOCK 3
```

The last three fields are `wd_list`, `cs_last` and the `wd_last` depends on the `CONFIG_CLOCKSOURCE_WATCHDOG` kernel configuration option. First of all let's try to understand what is it `whatchdog`. In a simple words, watchdog is a timer that is used for detection of the computer malfunctions and recovering from it. All of these three fields contain watchdog related data that is used by the `clocksource` framework. If we will grep the Linux kernel source code, we will see that only `arch/x86/KConfig` kernel configuration file contains the `CONFIG_CLOCKSOURCE_WATCHDOG` kernel configuration option. So, why do `x86` and `x86_64` need in `watchdog`? You already may know that all `x86` processors has special 64-bit register - `time stamp counter`. This register contains number of `cycles` since the reset. Sometimes the time stamp counter needs to be verified against another clock source. We will not see initialization of the `watchdog` timer in this part, before this we must learn more about timers.

That's all. From this moment we know all fields of the `clocksource` structure. This knowledge will help us to learn insides of the `clocksource` framework.

## New clock source registration

We saw only one function from the `clocksource` framework in the previous [part](#). This function was - `__clocksource_register`. This function defined in the [include/linux/clocksource.h](#) header file and as we can understand from the function's name, main point of this function is to register new clocksource. If we will look on the implementation of the `__clocksource_register` function, we will see that it just makes call of the `__clocksource_register_scale` function and returns its result:

```
static inline int __clocksource_register(struct clocksource *cs)
{
    return __clocksource_register_scale(cs, 1, 0);
}
```

Before we will see implementation of the `__clocksource_register_scale` function, we can see that `clocksource` provides additional API for a new clock source registration:

```
static inline int clocksource_register_hz(struct clocksource *cs, u32 hz)
{
    return __clocksource_register_scale(cs, 1, hz);
}

static inline int clocksource_register_khz(struct clocksource *cs, u32 khz)
{
    return __clocksource_register_scale(cs, 1000, khz);
}
```

And all of these functions do the same. They return value of the `__clocksource_register_scale` function but with different set of parameters. The `__clocksource_register_scale` function defined in the [kernel/time/clocksource.c](#) source code file. To understand difference between these functions, let's look on the parameters of the `clocksource_register_khz` function. As we can see, this function takes three parameters:

- `cs` - clocksource to be installed;
- `scale` - scale factor of a clock source. In other words, if we will multiply value of this parameter on frequency, we will get `hz` of a clocksource;
- `freq` - clock source frequency divided by scale.

Now let's look on the implementation of the `__clocksource_register_scale` function:

```
int __clocksource_register_scale(struct clocksource *cs, u32 scale, u32 freq)
{
    __clocksource_update_freq_scale(cs, scale, freq);
    mutex_lock(&clocksource_mutex);
    clocksource_enqueue(cs);
    clocksource_enqueue_watchdog(cs);
    clocksource_select();
    mutex_unlock(&clocksource_mutex);
    return 0;
}
```

First of all we can see that the `__clocksource_register_scale` function starts from the call of the `__clocksource_update_freq_scale` function that defined in the same source code file and updates given clock source with the new frequency. Let's look on the implementation of this function. In the first step we need to check given frequency and if it was not passed as `zero`, we need to calculate `mult` and `shift` parameters for the given clock source. Why do we need to check value of the `frequency`? Actually it can be zero. if you attentively looked on the implementation of the `__clocksource_register` function, you may have noticed that we passed `frequency` as `0`. We will do it only for some clock sources that have self defined `mult` and `shift` parameters. Look in the previous [part](#) and you will see that we saw calculation of the `mult` and `shift` for `jiffies`. The `__clocksource_update_freq_scale` function will do it for us for other clock sources.

So in the start of the `__clocksource_update_freq_scale` function we check the value of the `frequency` parameter and if is not zero we need to calculate `mult` and `shift` for the given clock source. Let's look on the `mult` and `shift` calculation:

```

void __clocksource_update_freq_scale(struct clocksource *cs, u32 scale, u32 freq)
{
    u64 sec;

    if (freq) {
        sec = cs->mask;
        do_div(sec, freq);
        do_div(sec, scale);

        if (!sec)
            sec = 1;
        else if (sec > 600 && cs->mask > UINT_MAX)
            sec = 600;

        clocks_calc_mult_shift(&cs->mult, &cs->shift, freq,
                               NSEC_PER_SEC / scale, sec * scale);
    }
    ...
    ...
    ...
}

```

Here we can see calculation of the maximum number of seconds which we can run before a clock source counter will overflow. First of all we fill the `sec` variable with the value of a clock source mask. Remember that a clock source's mask represents maximum amount of bits that are valid for the given clock source. After this, we can see two division operations. At first we divide our `sec` variable on a clock source frequency and then on scale factor. The `freq` parameter shows us how many timer interrupts will be occurred in one second. So, we divide `mask` value that represents maximum number of a counter (for example `jiffy`) on the frequency of a timer and will get the maximum number of seconds for the certain clock source. The second division operation will give us maximum number of seconds for the certain clock source depends on its scale factor which can be `1` hertz or `1` kilohertz ( $10^3$  Hz).

After we have got maximum number of seconds, we check this value and set it to `1` or `600` depends on the result at the next step. These values is maximum sleeping time for a clocksource in seconds. In the next step we can see call of the `clocks_calc_mult_shift`. Main point of this function is calculation of the `mult` and `shift` values for a given clock source. In the end of the `__clocksource_update_freq_scale` function we check that just calculated `mult` value of a given clock source will not cause overflow after adjustment, update the `max_idle_ns` and `max_cycles` values of a given clock source with the maximum nanoseconds that can be converted to a clock source counter and print result to the kernel buffer:

```
pr_info("%s: mask: 0x%llx max_cycles: 0x%llx, max_idle_ns: %lld ns\n",
        cs->name, cs->mask, cs->max_cycles, cs->max_idle_ns);
```

that we can see in the `dmesg` output:

```
$ dmesg | grep "clocksource:"
[ 0.000000] clocksource: refined-jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max
[ 0.000000] clocksource: hpet: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 1
[ 0.094084] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns
[ 0.205302] clocksource: acpi_pm: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 20
[ 1.452979] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x7350b459580, max_
```

After the `__clocksource_update_freq_scale` function will finish its work, we can return back to the `__clocksource_register_scale` function that will register new clock source. We can see the call of the following three functions:

```
mutex_lock(&clocksource_mutex);
clocksource_enqueue(cs);
clocksource_enqueue_watchdog(cs);
clocksource_select();
mutex_unlock(&clocksource_mutex);
```

Note that before the first will be called, we lock the `clocksource_mutex` [mutex](#). The point of the `clocksource_mutex` mutex is to protect `curr_clocksource` variable which represents currently selected `clocksource` and `clocksource_list` variable which represents list that contains registered `clocksources`. Now, let's look on these three functions.

The first `clocksource_enqueue` function and other two defined in the same source code [file](#). We go through all already registered `clocksources` or in other words we go through all elements of the `clocksource_list` and tries to find best place for a given `clocksource`:

```
static void clocksource_enqueue(struct clocksource *cs)
{
    struct list_head *entry = &clocksource_list;
    struct clocksource *tmp;

    list_for_each_entry(tmp, &clocksource_list, list)
        if (tmp->rating >= cs->rating)
            entry = &tmp->list;
    list_add(&cs->list, entry);
}
```

In the end we just insert new clocksource to the `clocksource_list`. The second function - `clocksource_enqueue_watchdog` does almost the same that previous function, but it inserts new clock source to the `wd_list` depends on flags of a clock source and starts new [watchdog](#) timer. As I already wrote, we will not consider `watchdog` related stuff in this part but will do it in next parts.

The last function is the `clocksource_select`. As we can understand from the function's name, main point of this function - select the best `clocksource` from registered clocksources. This function consists only from the call of the function helper:

```
static void clocksource_select(void)
{
    return __clocksource_select(false);
}
```

Note that the `__clocksource_select` function takes one parameter ( `false` in our case). This [bool](#) parameter shows how to traverse the `clocksource_list`. In our case we pass `false` that is meant that we will go through all entries of the `clocksource_list`. We already know that `clocksource` with the best rating will be the first in the `clocksource_list` after the call of the `clocksource_enqueue` function, so we can easily get it from this list. After we found a clock source with the best rating, we switch to it:

```
if (curr_clocksource != best && !timekeeping_notify(best)) {
    pr_info("Switched to clocksource %s\n", best->name);
    curr_clocksource = best;
}
```

The result of this operation we can see in the `dmesg` output:

```
$ dmesg | grep Switched
[  0.199688] clocksource: Switched to clocksource hpet
[  2.452966] clocksource: Switched to clocksource tsc
```

Note that we can see two clock sources in the `dmesg` output ( `hpet` and `tsc` in our case). Yes, actually there can be many different clock sources on a particular hardware. So the Linux kernel knows about all registered clock sources and switches to a clock source with a better rating each time after registration of a new clock source.

If we will look on the bottom of the [kernel/time/clocksource.c](#) source code file, we will see that it has [sysfs](#) interface. Main initialization occurs in the `init_clocksource_sysfs` function which will be called during device `initcalls`. Let's look on the implementation of the `init_clocksource_sysfs` function:

```
static struct bus_type clocksource_subsys = {
    .name = "clocksource",
    .dev_name = "clocksource",
};

static int __init init_clocksource_sysfs(void)
{
    int error = subsys_system_register(&clocksource_subsys, NULL);

    if (!error)
        error = device_register(&device_clocksource);
    if (!error)
        error = device_create_file(
            &device_clocksource,
            &dev_attr_current_clocksource);
    if (!error)
        error = device_create_file(&device_clocksource,
            &dev_attr_unbind_clocksource);
    if (!error)
        error = device_create_file(
            &device_clocksource,
            &dev_attr_available_clocksource);
    return error;
}
device_initcall(init_clocksource_sysfs);
```

First of all we can see that it registers a `clocksource` subsystem with the call of the `subsys_system_register` function. In other words, after the call of this function, we will have following directory:

```
$ pwd
/sys/devices/system/clocksource
```

After this step, we can see registration of the `device_clocksource` device which is represented by the following structure:

```
static struct device device_clocksource = {
    .id = 0,
    .bus = &clocksource_subsys,
};
```

and creation of three files:

- `dev_attr_current_clocksource` ;
- `dev_attr_unbind_clocksource` ;
- `dev_attr_available_clocksource` .



These files will provide information about current clock source in the system, available clock sources in the system and interface which allows to unbind the clock source.

After the `init_clocksource_sysfs` function will be executed, we will be able find some information about available clock sources in the:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

Or for example information about current clock source in the system:

```
$ cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

In the previous part, we saw API for the registration of the `jiffies` clock source, but didn't dive into details about the `clocksource` framework. In this part we did it and saw implementation of the new clock source registration and selection of a clock source with the best rating value in the system. Of course, this is not all API that `clocksource` framework provides. There are a couple additional functions like `clocksource_unregister` for removing given clock source from the `clocksource_list` and etc. But I will not describe these functions in this part, because they are not important for us right now. Anyway if you are interested in it, you can find it in the [kernel/time/clocksource.c](#).

That's all.

## Conclusion

This is the end of the second part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part got acquainted with the following two concepts: `jiffies` and `clocksource`. In this part we saw some examples of the `jiffies` usage and knew more details about the `clocksource` concept.

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [x86](#)

- [x86\\_64](#)
- [uptime](#)
- [Ensoniq Soundscape Elite](#)
- [RTC](#)
- [interrupts](#)
- [IBM PC](#)
- [programmable interval timer](#)
- [Hz](#)
- [nanoseconds](#)
- [dmesg](#)
- [time stamp counter](#)
- [loadable kernel module](#)
- [IA64](#)
- [watchdog](#)
- [clock rate](#)
- [mutex](#)
- [sysfs](#)
- [previous part](#)

# Timers and time management in the Linux kernel. Part 3.

## The tick broadcast framework and dyntick

This is third part of the [chapter](#) which describes timers and time management related stuff in the Linux kernel and we stopped on the `clocksource` framework in the previous [part](#). We have started to consider this framework because it is closely related to the special counters which are provided by the Linux kernel. One of these counters which we already saw in the first [part](#) of this chapter is - `jiffies`. As I already wrote in the first part of this chapter, we will consider time management related stuff step by step during the Linux kernel initialization. Previous step was call of the:

```
register_refined_jiffies(CLOCK_TICK_RATE);
```

function which defined in the [kernel/time/jiffies.c](#) source code file and executes initialization of the `refined_jiffies` clock source for us. Recall that this function is called from the `setup_arch` function that defined in the <https://github.com/torvalds/linux/blob/master/arch/x86/kernel/setup.c> source code and executes architecture-specific (`x86_64` in our case) initialization. Look on the implementation of the `setup_arch` and you will note that the call of the `register_refined_jiffies` is the last step before the `setup_arch` function will finish its work.

There are many different `x86_64` specific things already configured after the end of the `setup_arch` execution. For example some early [interrupt](#) handlers already able to handle interrupts, memory space reserved for the `initrd`, `DMI` scanned, the Linux kernel log buffer is already set and this means that the `printk` function is able to work, `e820` parsed and the Linux kernel already knows about available memory and many many other architecture specific things (if you are interesting, you can read more about the `setup_arch` function and Linux kernel initialization process in the second [chapter](#) of this book).

Now, the `setup_arch` finished its work and we can back to the generic Linux kernel code. Recall that the `setup_arch` function was called from the `start_kernel` function which is defined in the [init/main.c](#) source code file. So, we shall return to this function. You can see that there are many different function are called right after `setup_arch` function inside of the `start_kernel` function, but since our chapter is devoted to timers and time management related stuff, we will skip all code which is not related to this topic. The first function which is related to the time management in the Linux kernel is:

```
tick_init();
```

in the `start_kernel`. The `tick_init` function defined in the [kernel/time/tick-common.c](#) source code file and does two things:

- Initialization of `tick broadcast` framework related data structures;
- Initialization of `full` tickless mode related data structures.

We didn't see anything related to the `tick broadcast` framework in this book and didn't know anything about tickless mode in the Linux kernel. So, the main point of this part is to look on these concepts and to know what are they.

## The idle process

First of all, let's look on the implementation of the `tick_init` function. As I already wrote, this function defined in the [kernel/time/tick-common.c](#) source code file and consists from the two calls of following functions:

```
void __init tick_init(void)
{
    tick_broadcast_init();
    tick_nohz_init();
}
```

As you can understand from the paragraph's title, we are interesting only in the `tick_broadcast_init` function for now. This function defined in the [kernel/time/tick-broadcast.c](#) source code file and executes initialization of the `tick broadcast` framework related data structures. Before we will look on the implementation of the `tick_broadcast_init` function and will try to understand what does this function do, we need to know about `tick broadcast` framework.

Main point of a central processor is to execute programs. But sometimes a processor may be in a special state when it is not being used by any program. This special state is called - [idle](#). When the processor has no anything to execute, the Linux kernel launches `idle` task. We already saw a little about this in the last part of the [Linux kernel initialization process](#). When the Linux kernel will finish all initialization processes in the `start_kernel` function from the [init/main.c](#) source code file, it will call the `rest_init` function from the same source code file. Main point of this function is to launch kernel `init` thread and the `kthreadd` thread, to call the `schedule` function to start task scheduling and to go to sleep by calling the `cpu_idle_loop` function that defined in the [kernel/sched/idle.c](#) source code file.

The `cpu_idle_loop` function represents infinite loop which checks the need for rescheduling on each iteration. After the scheduler will find something to execute, the `idle` process will finish its work and the control will be moved to a new runnable task with the call of the `schedule_preempt_disabled` function:

```
static void cpu_idle_loop(void)
{
    while (1) {
        while (!need_resched()) {
            ...
            ...
            ...
            /* the main idle function */
            cpuidle_idle_call();
        }
        ...
        ...
        ...
        schedule_preempt_disabled();
    }
}
```

Of course, we will not consider full implementation of the `cpu_idle_loop` function and details of the `idle` state in this part, because it is not related to our topic. But there is one interesting moment for us. We know that the processor can execute only one task in one time. How does the Linux kernel decide to reschedule and stop `idle` process if the processor executes infinite loop in the `cpu_idle_loop`? The answer is system timer interrupts. When an interrupt occurs, the processor stops the `idle` thread and transfers control to an interrupt handler. After the system timer interrupt handler will be handled, the `need_resched` will return true and the Linux kernel will stop `idle` process and will transfer control to the current runnable task. But handling of the system timer interrupts is not effective for [power management](#), because if a processor is in `idle` state, there is little point in sending it a system timer interrupt.

By default, there is the `CONFIG_HZ_PERIODIC` kernel configuration option which is enabled in the Linux kernel and tells to handle each interrupt of the system timer. To solve this problem, the Linux kernel provides two additional ways of managing scheduling-clock interrupts:

The first is to omit scheduling-clock ticks on idle processors. To enable this behaviour in the Linux kernel, we need to enable the `CONFIG_NO_HZ_IDLE` kernel configuration option. This option allows Linux kernel to avoid sending timer interrupts to idle processors. In this case periodic timer interrupts will be replaced with on-demand interrupts. This mode is called - `dyntick-idle` mode. But if the kernel does not handle interrupts of a system timer, how can the kernel decide if the system has nothing to do?

Whenever the idle task is selected to run, the periodic tick is disabled with the call of the `tick_nohz_idle_enter` function that defined in the [kernel/time/tick-sched.c](#) source code file and enabled with the call of the `tick_nohz_idle_exit` function. There is special concept in the Linux kernel which is called - `clock event devices` that are used to schedule the next interrupt. This concept provides API for devices which can deliver interrupts at a specific time in the future and represented by the `clock_event_device` structure in the Linux kernel. We will not dive into implementation of the `clock_event_device` structure now. We will see it in the next part of this chapter. But there is one interesting moment for us right now.

The second way is to omit scheduling-clock ticks on processors that are either in `idle` state or that have only one runnable task or in other words busy processor. We can enable this feature with the `CONFIG_NO_HZ_FULL` kernel configuration option and it allows to reduce the number of timer interrupts significantly.

Besides the `cpu_idle_loop`, idle processor can be in a sleeping state. The Linux kernel provides special `cpuidle` framework. Main point of this framework is to put an idle processor to sleeping states. The name of the set of these states is - `C-states`. But how does a processor will be woken if local timer is disabled? The linux kernel provides `tick broadcast` framework for this. The main point of this framework is assign a timer which is not affected by the `C-states`. This timer will wake a sleeping processor.

Now, after some theory we can return to the implementation of our function. Let's recall that the `tick_init` function just calls two following functions:

```
void __init tick_init(void)
{
    tick_broadcast_init();
    tick_nohz_init();
}
```

Let's consider the first function. The first `tick_broadcast_init` function defined in the [kernel/time/tick-broadcast.c](#) source code file and executes initialization of the `tick broadcast` framework related data structures. Let's look on the implementation of the `tick_broadcast_init` function:

```

void __init tick_broadcast_init(void)
{
    zalloc_cpumask_var(&tick_broadcast_mask, GFP_NOWAIT);
    zalloc_cpumask_var(&tick_broadcast_on, GFP_NOWAIT);
    zalloc_cpumask_var(&tmpmask, GFP_NOWAIT);
#ifdef CONFIG_TICK_ONESHOT
    zalloc_cpumask_var(&tick_broadcast_oneshot_mask, GFP_NOWAIT);
    zalloc_cpumask_var(&tick_broadcast_pending_mask, GFP_NOWAIT);
    zalloc_cpumask_var(&tick_broadcast_force_mask, GFP_NOWAIT);
#endif
}

```

As we can see, the `tick_broadcast_init` function allocates different `cpumasks` with the help of the `zalloc_cpumask_var` function. The `zalloc_cpumask_var` function defined in the `lib/cpumask.c` source code file and expands to the call of the following function:

```

bool zalloc_cpumask_var(cpumask_var_t *mask, gfp_t flags)
{
    return alloc_cpumask_var(mask, flags | __GFP_ZERO);
}

```

Ultimately, the memory space will be allocated for the given `cpumask` with the certain flags with the help of the `kmalloc_node` function:

```

*mask = kmalloc_node(cpumask_size(), flags, node);

```

Now let's look on the `cpumasks` that will be initialized in the `tick_broadcast_init` function. As we can see, the `tick_broadcast_init` function will initialize six `cpumasks`, and moreover, initialization of the last three `cpumasks` will be depended on the `CONFIG_TICK_ONESHOT` kernel configuration option.

The first three `cpumasks` are:

- `tick_broadcast_mask` - the bitmap which represents list of processors that are in a sleeping mode;
- `tick_broadcast_on` - the bitmap that stores numbers of processors which are in a periodic broadcast state;
- `tmpmask` - this bitmap for temporary usage.

As we already know, the next three `cpumasks` depends on the `CONFIG_TICK_ONESHOT` kernel configuration option. Actually each clock event devices can be in one of two modes:

- `periodic` - clock events devices that support periodic events;
- `oneshot` - clock events devices that capable of issuing events that happen only once.

The linux kernel defines two mask for such clock events devices in the [include/linux/clockchips.h](#) header file:

```
#define CLOCK_EVT_FEAT_PERIODIC    0x000001
#define CLOCK_EVT_FEAT_ONESHOT    0x000002
```

So, the last three `cpumasks` are:

- `tick_broadcast_oneshot_mask` - stores numbers of processors that must be notified;
- `tick_broadcast_pending_mask` - stores numbers of processors that pending broadcast;
- `tick_broadcast_force_mask` - stores numbers of processors with enforced broadcast.

We have initialized six `cpumasks` in the `tick broadcast` framework, and now we can proceed to implementation of this framework.

## The `tick broadcast` framework

Hardware may provide some clock source devices. When a processor sleeps and its local timer stopped, there must be additional clock source device that will handle awakening of a processor. The Linux kernel uses these `special` clock source devices which can raise an interrupt at a specified time. We already know that such timers called `clock events` devices in the Linux kernel. Besides `clock events` devices. Actually, each processor in the system has its own local timer which is programmed to issue interrupt at the time of the next deferred task. Also these timers can be programmed to do a periodical job, like updating `jiffies` and etc. These timers represented by the `tick_device` structure in the Linux kernel. This structure defined in the [kernel/time/tick-sched.h](#) header file and looks:

```
struct tick_device {
    struct clock_event_device *evtdev;
    enum tick_device_mode mode;
};
```

Note, that the `tick_device` structure contains two fields. The first field - `evtdev` represents pointer to the `clock_event_device` structure that defined in the [include/linux/clockchips.h](#) header file and represents descriptor of a clock event device. A `clock event` device allows to register an event that will happen in the future. As I already wrote, we will not consider `clock_event_device` structure and related API in this part, but will see it in the next part.

The second field of the `tick_device` structure represents mode of the `tick_device`. As we already know, the mode can be one of the:



```

num tick_device_mode {
    TICKDEV_MODE_PERIODIC,
    TICKDEV_MODE_ONESHOT,
};

```

Each `clock events` device in the system registers itself by the call of the `clockevents_register_device` function or `clockevents_config_and_register` function during initialization process of the Linux kernel. During the registration of a new `clock events` device, the Linux kernel calls the `tick_check_new_device` function that defined in the [kernel/time/tick-common.c](#) source code file and checks the given `clock events` device should be used by the Linux kernel. After all checks, the `tick_check_new_device` function executes a call of the:

```

tick_install_broadcast_device(newdev);

```

function that checks that the given `clock event` device can be broadcast device and install it, if the given device can be broadcast device. Let's look on the implementation of the `tick_install_broadcast_device` function:

```

void tick_install_broadcast_device(struct clock_event_device *dev)
{
    struct clock_event_device *cur = tick_broadcast_device.evtdev;

    if (!tick_check_broadcast_device(cur, dev))
        return;

    if (!try_module_get(dev->owner))
        return;

    clockevents_exchange_device(cur, dev);

    if (cur)
        cur->event_handler = clockevents_handle_noop;

    tick_broadcast_device.evtdev = dev;

    if (!cpumask_empty(tick_broadcast_mask))
        tick_broadcast_start_periodic(dev);

    if (dev->features & CLOCK_EVT_FEAT_ONESHOT)
        tick_clock_notify();
}

```

First of all we get the current `clock event` device from the `tick_broadcast_device`. The `tick_broadcast_device` defined in the [kernel/time/tick-common.c](#) source code file:

```
static struct tick_device tick_broadcast_device;
```

and represents external clock device that keeps track of events for a processor. The first step after we got the current clock device is the call of the `tick_check_broadcast_device` function which checks that a given clock events device can be utilized as broadcast device. The main point of the `tick_check_broadcast_device` function is to check value of the `features` field of the given `clock events` device. As we can understand from the name of this field, the `features` field contains a clock event device features. Available values defined in the [include/linux/clockchips.h](#) header file and can be one of the `CLOCK_EVT_FEAT_PERIODIC` - which represents a clock events device which supports periodic events and etc. So, the `tick_check_broadcast_device` function check `features` flags for `CLOCK_EVT_FEAT_ONESHOT`, `CLOCK_EVT_FEAT_DUMMY` and other flags and returns `false` if the given clock events device has one of these features. In other way the `tick_check_broadcast_device` function compares `ratings` of the given clock event device and current clock event device and returns the best.

After the `tick_check_broadcast_device` function, we can see the call of the `try_module_get` function that checks module owner of the clock events. We need to do it to be sure that the given `clock events` device was correctly initialized. The next step is the call of the `clockevents_exchange_device` function that defined in the [kernel/time/clockevents.c](#) source code file and will release old clock events device and replace the previous functional handler with a dummy handler.

In the last step of the `tick_install_broadcast_device` function we check that the `tick_broadcast_mask` is not empty and start the given `clock events` device in periodic mode with the call of the `tick_broadcast_start_periodic` function:

```
if (!cpumask_empty(tick_broadcast_mask))
    tick_broadcast_start_periodic(dev);

if (dev->features & CLOCK_EVT_FEAT_ONESHOT)
    tick_clock_notify();
```

The `tick_broadcast_mask` filled in the `tick_device_uses_broadcast` function that checks a `clock events` device during registration of this `clock events` device:

```
int cpu = smp_processor_id();

int tick_device_uses_broadcast(struct clock_event_device *dev, int cpu)
{
    ...
    ...
    ...
    if (!tick_device_is_functional(dev)) {
        ...
        cpumask_set_cpu(cpu, tick_broadcast_mask);
        ...
    }
    ...
    ...
    ...
}
```

More about the `smp_processor_id` macro you can read in the fourth [part](#) of the Linux kernel initialization process chapter.

The `tick_broadcast_start_periodic` function check the given `clock event` device and call the `tick_setup_periodic` function:

```
static void tick_broadcast_start_periodic(struct clock_event_device *bc)
{
    if (bc)
        tick_setup_periodic(bc, 1);
}
```

that defined in the [kernel/time/tick-common.c](#) source code file and sets broadcast handler for the given `clock event` device by the call of the following function:

```
tick_set_periodic_handler(dev, broadcast);
```

This function checks the second parameter which represents broadcast state ( `on` or `off` ) and sets the broadcast handler depends on its value:

```
void tick_set_periodic_handler(struct clock_event_device *dev, int broadcast)
{
    if (!broadcast)
        dev->event_handler = tick_handle_periodic;
    else
        dev->event_handler = tick_handle_periodic_broadcast;
}
```

When an `clock event` device will issue an interrupt, the `dev->event_handler` will be called. For example, let's look on the interrupt handler of the [high precision event timer](#) which is located in the [arch/x86/kernel/hpet.c](#) source code file:

```
static irqreturn_t hpet_interrupt_handler(int irq, void *data)
{
    struct hpet_dev *dev = (struct hpet_dev *)data;
    struct clock_event_device *hevt = &dev->evt;

    if (!hevt->event_handler) {
        printk(KERN_INFO "Spurious HPET timer interrupt on HPET timer %d\n",
                dev->num);
        return IRQ_HANDLED;
    }

    hevt->event_handler(hevt);
    return IRQ_HANDLED;
}
```

The `hpet_interrupt_handler` gets the `irq` specific data and check the event handler of the `clock event` device. Recall that we just set in the `tick_set_periodic_handler` function. So the `tick_handler_periodic_broadcast` function will be called in the end of the high precision event timer interrupt handler.

The `tick_handler_periodic_broadcast` function calls the

```
bc_local = tick_do_periodic_broadcast();
```

function which stores numbers of processors which have asked to be woken up in the temporary `cpumask` and call the `tick_do_broadcast` function:

```
cpumask_and(tmpmask, cpu_online_mask, tick_broadcast_mask);
return tick_do_broadcast(tmpmask);
```

The `tick_do_broadcast` calls the `broadcast` function of the given clock events which sends [IPI](#) interrupt to the set of the processors. In the end we can call the event handler of the given `tick_device` :

```
if (bc_local)
    td->evtdev->event_handler(td->evtdev);
```

which actually represents interrupt handler of the local timer of a processor. After this a processor will wake up. That is all about `tick broadcast` framework in the Linux kernel. We have missed some aspects of this framework, for example reprogramming of a `clock event`

device and broadcast with the oneshot timer and etc. But the Linux kernel is very big, it is not real to cover all aspects of it. I think it will be interesting to dive into with yourself.

If you remember, we have started this part with the call of the `tick_init` function. We just consider the `tick_broadcast_init` function and related theory, but the `tick_init` function contains another call of a function and this function is - `tick_nohz_init`. Let's look on the implementation of this function.

## Initialization of dyntick related data structures

We already saw some information about `dyntick` concept in this part and we know that this concept allows kernel to disable system timer interrupts in the `idle` state. The `tick_nohz_init` function makes initialization of the different data structures which are related to this concept. This function defined in the [kernel/time/tick-sched.c](#) source code file and starts from the check of the value of the `tick_nohz_full_running` variable which represents state of the tick-less mode for the `idle` state and the state when system timer interrupts are disabled during a processor has only one runnable task:

```
if (!tick_nohz_full_running) {  
    if (tick_nohz_init_all() < 0)  
        return;  
}
```

If this mode is not running we call the `tick_nohz_init_all` function that defined in the same source code file and check its result. The `tick_nohz_init_all` function tries to allocate the `tick_nohz_full_mask` with the call of the `alloc_cpumask_var` that will allocate space for a `tick_nohz_full_mask`. The `tick_nohz_full_mask` will store numbers of processors that have enabled full `NO_HZ`. After successful allocation of the `tick_nohz_full_mask` we set all bits in the `tick_nohz_full_mask`, set the `tick_nohz_full_running` and return result to the `tick_nohz_init` function:

```
static int tick_nohz_init_all(void)
{
    int err = -1;
#ifdef CONFIG_NO_HZ_FULL_ALL
    if (!alloc_cpumask_var(&tick_nohz_full_mask, GFP_KERNEL)) {
        WARN(1, "NO_HZ: Can't allocate full dynticks cpumask\n");
        return err;
    }
    err = 0;
    cpumask_setall(tick_nohz_full_mask);
    tick_nohz_full_running = true;
#endif
    return err;
}
```

In the next step we try to allocate a memory space for the `housekeeping_mask` :

```
if (!alloc_cpumask_var(&housekeeping_mask, GFP_KERNEL)) {
    WARN(1, "NO_HZ: Can't allocate not-full dynticks cpumask\n");
    cpumask_clear(tick_nohz_full_mask);
    tick_nohz_full_running = false;
    return;
}
```

This `cpumask` will store number of processor for `housekeeping` or in other words we need at least in one processor that will not be in `NO_HZ` mode, because it will do timekeeping and etc. After this we check the result of the architecture-specific `arch_irq_work_has_interrupt` function. This function checks ability to send inter-processor interrupt for the certain architecture. We need to check this, because system timer of a processor will be disabled during `NO_HZ` mode, so there must be at least one online processor which can send inter-processor interrupt to awake offline processor. This function defined in the [arch/x86/include/asm/irq\\_work.h](#) header file for the `x86_64` and just checks that a processor has `APIC` from the `CPUID`:

```
static inline bool arch_irq_work_has_interrupt(void)
{
    return cpu_has_apic;
}
```

If a processor has not `APIC` , the Linux kernel prints warning message, clears the `tick_nohz_full_mask` `cpumask`, copies numbers of all possible processors in the system to the `housekeeping_mask` and resets the value of the `tick_nohz_full_running` variable:

```

if (!arch_irq_work_has_interrupt()) {
    pr_warning("NO_HZ: Can't run full dynticks because arch doesn't "
              "support irq work self-IPIs\n");
    cpumask_clear(tick_nohz_full_mask);
    cpumask_copy(housekeeping_mask, cpu_possible_mask);
    tick_nohz_full_running = false;
    return;
}

```

After this step, we get the number of the current processor by the call of the `smp_processor_id` and check this processor in the `tick_nohz_full_mask`. If the `tick_nohz_full_mask` contains a given processor we clear appropriate bit in the `tick_nohz_full_mask`:

```

cpu = smp_processor_id();

if (cpumask_test_cpu(cpu, tick_nohz_full_mask)) {
    pr_warning("NO_HZ: Clearing %d from nohz_full range for timekeeping\n", cpu);
    cpumask_clear_cpu(cpu, tick_nohz_full_mask);
}

```

Because this processor will be used for timekeeping. After this step we put all numbers of processors that are in the `cpu_possible_mask` and not in the `tick_nohz_full_mask`:

```

cpumask_andnot(housekeeping_mask,
               cpu_possible_mask, tick_nohz_full_mask);

```

After this operation, the `housekeeping_mask` will contain all processors of the system except a processor for timekeeping. In the last step of the `tick_nohz_init_all` function, we are going through all processors that are defined in the `tick_nohz_full_mask` and call the following function for an each processor:

```

for_each_cpu(cpu, tick_nohz_full_mask)
    context_tracking_cpu_set(cpu);

```

The `context_tracking_cpu_set` function defined in the [kernel/context\\_tracking.c](#) source code file and main point of this function is to set the `context_tracking.active` percpu variable to `true`. When the `active` field will be set to `true` for the certain processor, all [context switches](#) will be ignored by the Linux kernel context tracking subsystem for this processor.

That's all. This is the end of the `tick_nohz_init` function. After this `NO_HZ` related data structures will be initialized. We didn't see API of the `NO_HZ` mode, but will see it soon.

## Conclusion

This is the end of the third part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part got acquainted with the `clocksource` concept in the Linux kernel which represents framework for managing different clock source in a interrupt and hardware characteristics independent way. We continued to look on the Linux kernel initialization process in a time management context in this part and got acquainted with two new concepts for us: the `tick broadcast` framework and `tick-less` mode. The first concept helps the Linux kernel to deal with processors which are in deep sleep and the second concept represents the mode in which kernel may work to improve power management of `idle` processors.

In the next part we will continue to dive into timer management related things in the Linux kernel and will see new concept for us - `timers`.

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [x86\\_64](#)
- [initrd](#)
- [interrupt](#)
- [DMI](#)
- [printk](#)
- [CPU idle](#)
- [power management](#)
- [NO\\_HZ documentation](#)
- [cpumasks](#)
- [high precision event timer](#)
- [irq](#)
- [IPI](#)
- [CUID](#)
- [APIC](#)
- [percpu](#)
- [context switches](#)
- [Previous part](#)



# Timers and time management in the Linux kernel. Part 4.

## Timers

This is fourth part of the [chapter](#) which describes timers and time management related stuff in the Linux kernel and in the previous [part](#) we knew about the `tick broadcast` framework and `NO_HZ` mode in the Linux kernel. We will continue to dive into the time management related stuff in the Linux kernel in this part and will be acquainted with yet another concept in the Linux kernel - `timers`. Before we will look at timers in the Linux kernel, we have to learn some theory about this concept. Note that we will consider software timers in this part.

The Linux kernel provides a `software timer` concept to allow to kernel functions could be invoked at future moment. Timers are widely used in the Linux kernel. For example, look in the [net/netfilter/ipset/ip\\_set\\_list\\_set.c](#) source code file. This source code file provides implementation of the framework for the managing of groups of [IP](#) addresses.

We can find the `list_set` structure that contains `gc` filed in this source code file:

```
struct list_set {  
    ...  
    struct timer_list gc;  
    ...  
};
```

Not that the `gc` filed has `timer_list` type. This structure defined in the [include/linux/timer.h](#) header file and main point of this structure is to store `dynamic` timers in the Linux kernel. Actually, the Linux kernel provides two types of timers called dynamic timers and interval timers. First type of timers is used by the kernel, and the second can be used by user mode. The `timer_list` structure contains actual `dynamic` timers. The `list_set` contains `gc` timer in our example represents timer for garbage collection. This timer will be initialized in the `list_set_gc_init` function:

```
static void
list_set_gc_init(struct ip_set *set, void (*gc)(unsigned long ul_set))
{
    struct list_set *map = set->data;
    ...
    ...
    ...
    map->gc.function = gc;
    map->gc.expires = jiffies + IPSET_GC_PERIOD(set->timeout) * HZ;
    ...
    ...
    ...
}
```

A function that is pointed by the `gc` pointer, will be called after timeout which is equal to the `map->gc.expires`.

Ok, we will not dive into this example with the [netfilter](#), because this chapter is not about [network](#) related stuff. But we saw that timers are widely used in the Linux kernel and learned that they represent concept which allows to functions to be called in future.

Now let's continue to research source code of Linux kernel which is related to the timers and time management stuff as we did it in all previous chapters.

## Introduction to dynamic timers in the Linux kernel

As I already wrote, we knew about the `tick broadcast` framework and `NO_HZ` mode in the previous [part](#). They will be initialized in the [init/main.c](#) source code file by the call of the `tick_init` function. If we will look at this source code file, we will see that the next time management related function is:

```
init_timers();
```

This function defined in the [kernel/time/timer.c](#) source code file and contains calls of four functions:

```
void __init init_timers(void)
{
    init_timer_cpus();
    init_timer_stats();
    timer_register_cpu_notifier();
    open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
}
```

Let's look on implementation of each function. The first function is `init_timer_cpus` defined in the [same](#) source code file and just calls the `init_timer_cpu` function for each possible processor in the system:

```
static void __init init_timer_cpus(void)
{
    int cpu;

    for_each_possible_cpu(cpu)
        init_timer_cpu(cpu);
}
```

If you do not know or do not remember what is it a `possible` cpu, you can read the special [part](#) of this book which describes `cpumask` concept in the Linux kernel. In short words, a `possible` processor is a processor which can be plugged in anytime during the life of the system.

The `init_timer_cpu` function does main work for us, namely it executes initialization of the `tvec_base` structure for each processor. This structure defined in the [kernel/time/timer.c](#) source code file and stores data related to a `dynamic` timer for a certain processor. Let's look on the definition of this structure:

```
struct tvec_base {
    spinlock_t lock;
    struct timer_list *running_timer;
    unsigned long timer_jiffies;
    unsigned long next_timer;
    unsigned long active_timers;
    unsigned long all_timers;
    int cpu;
    bool migration_enabled;
    bool nohz_active;
    struct tvec_root tv1;
    struct tvec tv2;
    struct tvec tv3;
    struct tvec tv4;
    struct tvec tv5;
} ____cacheline_aligned;
```

The `thecc_base` structure contains following fields: The `lock` for `tvec_base` protection, the next `running_timer` field points to the currently running timer for the certain processor, the `timer_jiffies` fields represents the earliest expiration time (it will be used by the Linux kernel to find already expired timers). The next field - `next_timer` contains the next pending timer for a next timer [interrupt](#) in a case when a processor goes to sleep and the `NO_HZ` mode is enabled in the Linux kernel. The `active_timers` field provides accounting of non-deferrable timers or in other words all timers that will not be stopped during a processor will go to sleep. The `all_timers` field tracks total number of timers or `active_timers` + deferrable timers. The `cpu` field represents number of a processor which owns timers. The `migration_enabled` and `nohz_active` fields are represent opportunity of timers migration to another processor and status of the `NO_HZ` mode respectively.

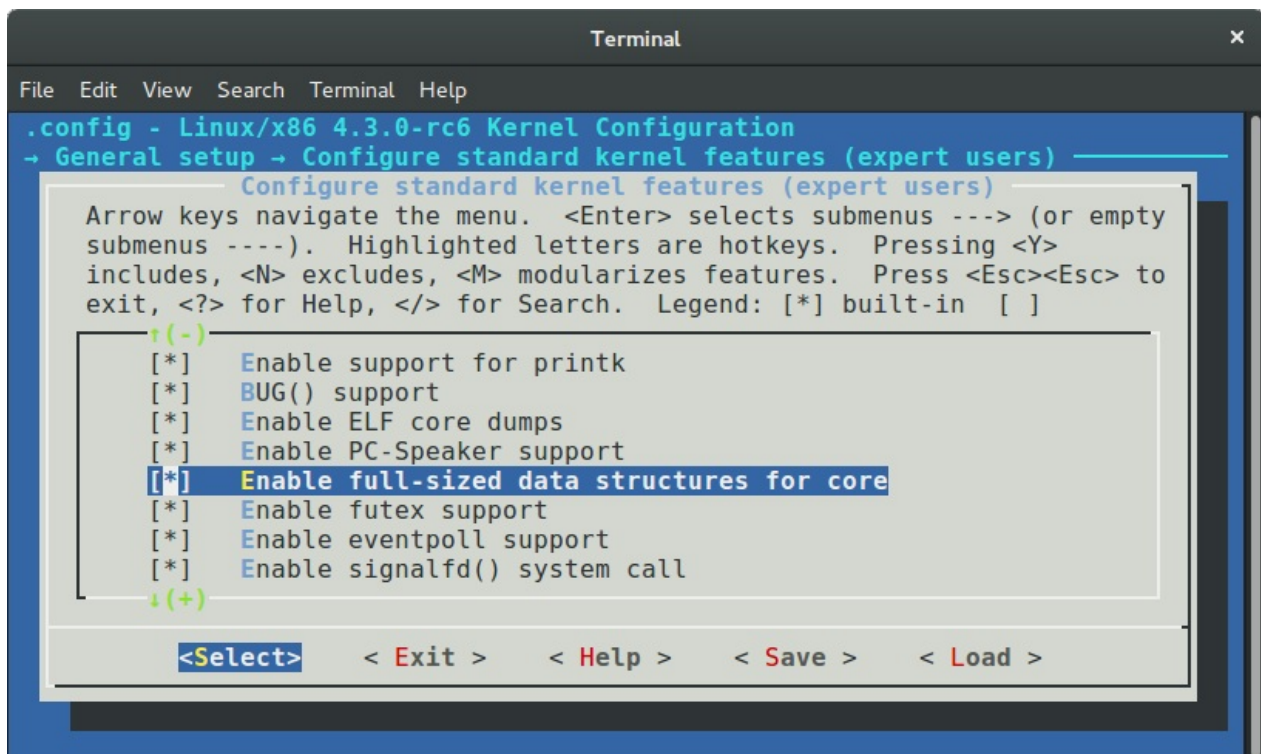
The last five fields of the `tvec_base` structure represent lists of dynamic timers. The first `tv1` field has:

```
#define TVR_SIZE (1 << TVR_BITS)
#define TVR_BITS (CONFIG_BASE_SMALL ? 6 : 8)

...
...
...

struct tvec_root {
    struct hlist_head vec[TVR_SIZE];
};
```

type. Note that the value of the `TVR_SIZE` depends on the `CONFIG_BASE_SMALL` kernel configuration option:



that reduces size of the kernel data structures if disabled. The `tv1` is array that may contain 64 or 256 elements where each element represents a dynamic timer that will decay within the next 255 system timer interrupts. Next three fields: `tv2`, `tv3` and `tv4` are lists with dynamic timers too, but they store dynamic timers which will decay the next  $2^{14} - 1$ ,  $2^{20} - 1$  and  $2^{26}$  respectively. The last `tv5` field represents list which stores dynamic timers with a large expiring period.

So, now we saw the `tvec_base` structure and description of its fields and we can look on the implementation of the `init_timer_cpu` function. As I already wrote, this function defined in the [kernel/time/timer.c](#) source code file and executes initialization of the `tvec_bases`:

```
static void __init init_timer_cpu(int cpu)
{
    struct tvec_base *base = per_cpu_ptr(&tvec_bases, cpu);

    base->cpu = cpu;
    spin_lock_init(&base->lock);

    base->timer_jiffies = jiffies;
    base->next_timer = base->timer_jiffies;
}
```

The `tvec_bases` represents per-cpu variable which represents main data structure for a dynamic timer for a given processor. This per-cpu variable defined in the same source code file:

```
static DEFINE_PER_CPU(struct tvec_base, tvec_bases);
```

First of all we're getting the address of the `tvec_bases` for the given processor to `base` variable and as we got it, we are starting to initialize some of the `tvec_base` fields in the `init_timer_cpu` function. After initialization of the `per-cpu` dynamic timers with the `jiffies` and the number of a possible processor, we need to initialize a `tstats_lookup_lock` `spinlock` in the `init_timer_stats` function:

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

    for_each_possible_cpu(cpu)
        raw_spin_lock_init(&per_cpu(tstats_lookup_lock, cpu));
}
```

The `tstats_lookup_lock` variable represents `per-cpu` raw spinlock:

```
static DEFINE_PER_CPU(raw_spinlock_t, tstats_lookup_lock);
```

which will be used for protection of operation with statistics of timers that can be accessed through the `procfs`:

```
static int __init init_tstats_procfs(void)
{
    struct proc_dir_entry *pe;

    pe = proc_create("timer_stats", 0644, NULL, &tstats_fops);
    if (!pe)
        return -ENOMEM;
    return 0;
}
```

For example:

```
$ cat /proc/timer_stats
Timerstats sample period: 3.888770 s
 12,      0 swapper      hrtimer_stop_sched_tick (hrtimer_sched_tick)
 15,      1 swapper      hcd_submit_urb (rh_timer_func)
  4,    959 kded         schedule_timeout (process_timeout)
  1,      0 swapper      page_writeback_init (wb_timer_fn)
 28,      0 swapper      hrtimer_stop_sched_tick (hrtimer_sched_tick)
 22,   2948 IRQ 4        tty_flip_buffer_push (delayed_work_timer_fn)
...
...
...
```

The next step after initialization of the `tstats_lookup_lock` spinlock is the call of the `timer_register_cpu_notifier` function. This function depends on the `CONFIG_HOTPLUG_CPU` kernel configuration option which enables support for [hotplug](#) processors in the Linux kernel.

When a processor will be logically offlined, a notification will be sent to the Linux kernel with the `CPU_DEAD` or the `CPU_DEAD_FROZEN` event by the call of the `cpu_notifier` macro:

```
#ifdef CONFIG_HOTPLUG_CPU
...
...
static inline void timer_register_cpu_notifier(void)
{
    cpu_notifier(timer_cpu_notify, 0);
}
...
...
#else
...
...
static inline void timer_register_cpu_notifier(void) { }
...
...
#endif /* CONFIG_HOTPLUG_CPU */
```

In this case the `timer_cpu_notify` will be called which checks an event type and will call the `migrate_timers` function:

```
static int timer_cpu_notify(struct notifier_block *self,
                           unsigned long action, void *hcpu)
{
    switch (action) {
        case CPU_DEAD:
        case CPU_DEAD_FROZEN:
            migrate_timers((long)hcpu);
            break;
        default:
            break;
    }

    return NOTIFY_OK;
}
```

This chapter will not describe `hotplug` related events in the Linux kernel source code, but if you are interesting in such things, you can find implementation of the `migrate_timers` function in the [kernel/time/timer.c](#) source code file.

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

```
open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
```

function. The `open_softirq` function may be already familiar to you if you have read the ninth [part](#) about the interrupts and interrupt handling in the Linux kernel. In short words, the `open_softirq` function defined in the [kernel/softirq.c](#) source code file and executes initialization of the deferred interrupt handler.

In our case the deferred function is the `run_timer_softirq` function that is will be called after a hardware interrupt in the `do_IRQ` function which defined in the [arch/x86/kernel/irq.c](#) source code file. The main point of this function is to handle a software dynamic timer. The Linux kernel does not do this thing during the hardware timer interrupt handling because this is time consuming operation.

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

```
static void run_timer_softirq(struct softirq_action *h)
{
    struct tvec_base *base = this_cpu_ptr(&tvec_bases);

    if (time_after_eq(jiffies, base->timer_jiffies))
        __run_timers(base);
}
```



At the beginning of the `run_timer_softirq` function we get a `dynamic` timer for a current processor and compares the current value of the `jiffies` with the value of the `timer_jiffies` for the current structure by the call of the `time_after_eq` macro which is defined in the `include/linux/jiffies.h` header file:

```
#define time_after_eq(a,b) \
    (typecheck(unsigned long, a) && \
     typecheck(unsigned long, b) && \
     ((long)((a) - (b)) >= 0))
```

Reclaim that the `timer_jiffies` field of the `tvec_base` structure represents the relative time when functions delayed by the given timer will be executed. So we compare these two values and if the current time represented by the `jiffies` is greater than `base->timer_jiffies`, we call the `__run_timers` function that defined in the same source code file. Let's look on the implementation of this function.

As I just wrote, the `__run_timers` function runs all expired timers for a given processor. This function starts from the acquiring of the `tvec_base's` lock to protect the `tvec_base` structure

```
static inline void __run_timers(struct tvec_base *base)
{
    struct timer_list *timer;

    spin_lock_irq(&base->lock);
    ...
    ...
    ...
    spin_unlock_irq(&base->lock);
}
```

After this it starts the loop while the `timer_jiffies` will not be greater than the `jiffies`:

```
while (time_after_eq(jiffies, base->timer_jiffies)) {
    ...
    ...
    ...
}
```

We can find many different manipulations in the our loop, but the main point is to find expired timers and call delayed functions. First of all we need to calculate the `index` of the `base->tv1` list that stores the next timer to be handled with the following expression:

```
index = base->timer_jiffies & TVR_MASK;
```

where the `TVR_MASK` is a mask for the getting of the `tvec_root->vec` elements. As we got the index with the next timer which must be handled we check its value. If the index is zero, we go through all lists in our cascade table `tv2`, `tv3` and etc., and rehashing it with the call of the `cascade` function:

```
if (!index &&
    (!cascade(base, &base->tv2, INDEX(0))) &&
    (!cascade(base, &base->tv3, INDEX(1))) &&
    !cascade(base, &base->tv4, INDEX(2)))
    cascade(base, &base->tv5, INDEX(3));
```

After this we increase the value of the `base->timer_jiffies` :

```
++base->timer_jiffies;
```

In the last step we are executing a corresponding function for each timer from the list in a following loop:

```
hlist_move_list(base->tv1.vec + index, head);

while (!hlist_empty(head)) {
    ...
    ...
    ...
    timer = hlist_entry(head->first, struct timer_list, entry);
    fn = timer->function;
    data = timer->data;

    spin_unlock(&base->lock);
    call_timer_fn(timer, fn, data);
    spin_lock(&base->lock);

    ...
    ...
    ...
}
```

where the `call_timer_fn` just call the given function:

```
static void call_timer_fn(struct timer_list *timer, void (*fn)(unsigned long),
                        unsigned long data)
{
    ...
    ...
    ...
    fn(data);
    ...
    ...
    ...
}
```

That's all. The Linux kernel has infrastructure for `dynamic timers` from this moment. We will not dive into this interesting theme. As I already wrote the `timers` is a [widely](#) used concept in the Linux kernel and nor one part, nor two parts will not cover understanding of such things how it implemented and how it works. But now we know about this concept, why does the Linux kernel needs in it and some data structures around it.

Now let's look usage of `dynamic timers` in the Linux kernel.

## Usage of dynamic timers

As you already can noted, if the Linux kernel provides a concept, it also provides API for managing of this concept and the `dynamic timers` concept is not exception here. To use a timer in the Linux kernel code, we must define a variable with a `timer_list` type. We can initialize our `timer_list` structure in two ways. The first is to use the `init_timer` macro that defined in the [include/linux/timer.h](#) header file:

```
#define init_timer(timer) \
    __init_timer((timer), 0)

#define __init_timer(_timer, _flags) \
    init_timer_key((_timer), (_flags), NULL, NULL)
```

where the `init_timer_key` function just calls the:

```
do_init_timer(timer, flags, name, key);
```

function which fields the given `timer` with default values. The second way is to use the:

```
#define TIMER_INITIALIZER(_function, _expires, _data) \
    __TIMER_INITIALIZER((_function), (_expires), (_data), 0)
```

macro which will initialize the given `timer_list` structure too.

After a `dynamic timer` is initialized we can start this `timer` with the call of the:

```
void add_timer(struct timer_list * timer);
```

function and stop it with the:

```
int del_timer(struct timer_list * timer);
```

function.

That's all.

## Conclusion

This is the end of the fourth part of the chapter that describes timers and timer management related stuff in the Linux kernel. In the previous part we got acquainted with the two new concepts: the `tick broadcast` framework and the `NO_HZ` mode. In this part we continued to dive into time management related stuff and got acquainted with the new concept - `dynamic timer` or software timer. We didn't see implementation of a `dynamic timers` management code in details in this part but saw data structures and API around this concept.

In the next part we will continue to dive into timer management related things in the Linux kernel and will see new concept for us - `timers`.

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [IP](#)
- [netfilter](#)
- [network](#)
- [cpumask](#)
- [interrupt](#)
- [jiffies](#)

- [per-cpu](#)
- [spinlock](#)
- [procfs](#)
- [previous part](#)

# Timers and time management in the Linux kernel. Part 5.

## Introduction to the `clockevents` framework

This is fifth part of the [chapter](#) which describes timers and time management related stuff in the Linux kernel. As you might noted from the title of this part, the `clockevents` framework will be discussed. We already saw one framework in the [second](#) part of this chapter. It was `clocksource` framework. Both of these frameworks represent timekeeping abstractions in the Linux kernel.

At first let's refresh your memory and try to remember what is it `clocksource` framework and what its purpose. The main goal of the `clocksource` framework is to provide `timeline`. As described in the [documentation](#):

For example issuing the command 'date' on a Linux system will eventually read the clock source to determine exactly what time it is.

The Linux kernel supports many different clock sources. You can find some of them in the [drivers/clocksource](#). For example old good [Intel 8253 - programmable interval timer](#) with `1193182` Hz frequency, yet another one - [ACPI PM timer](#) with `3579545` Hz frequency. Besides the [drivers/clocksource](#) directory, each architecture may provide own architecture-specific clock sources. For example [x86](#) architecture provides [High Precision Event Timer](#), or for example [powerpc](#) provides access to the processor timer through `timebase` register.

Each clock source provides monotonic atomic counter. As I already wrote, the Linux kernel supports a huge set of different clock source and each clock source has own parameters like [frequency](#). The main goal of the `clocksource` framework is to provide [API](#) to select best available clock source in the system i.e. a clock source with the highest frequency. Additional goal of the `clocksource` framework is to represent an atomic counter provided by a clock source in human units. In this time, nanoseconds are the favorite choice for the time value units of the given clock source in the Linux kernel.

The `clocksource` framework represented by the `clocksource` structure which is defined in the [include/linux/clocksource.h](#) header code file which contains `name` of a clock source, rating of certain clock source in the system (a clock source with the higher frequency has the biggest rating in the system), `list` of all registered clock source in the system, `enable` and `disable` fields to enable and disable a clock source, pointer to the `read` function which must return an atomic counter of a clock source and etc.

Additionally the `clocksource` structure provides two fields: `mult` and `shift` which are needed for translation of an atomic counter which is provided by a certain clock source to the human units, i.e. [nanoseconds](#). Translation occurs via following formula:

```
ns ~= (clocksource * mult) >> shift
```

As we already know, besides the `clocksource` structure, the `clocksource` framework provides an API for registration of clock source with different frequency scale factor:

```
static inline int clocksource_register_hz(struct clocksource *cs, u32 hz)
static inline int clocksource_register_khz(struct clocksource *cs, u32 khz)
```

A clock source unregistration:

```
int clocksource_unregister(struct clocksource *cs)
```

and etc.

Additionally to the `clocksource` framework, the Linux kernel provides `clockevents` framework. As described in the [documentation](#):

Clock events are the conceptual reverse of clock sources

Main goal of the is to manage clock event devices or in other words - to manage devices that allow to register an event or in other words [interrupt](#) that is going to happen at a defined point of time in the future.

Now we know a little about the `clockevents` framework in the Linux kernel, and now time is to see on it [API](#).

## API of `clockevents` framework

The main structure which described a clock event device is `clock_event_device` structure. This structure is defined in the [include/linux/clockchips.h](#) header file and contains a huge set of fields. as well as the `clocksource` structure it has `name` fields which contains human readable name of a clock event device, for example [local APIC](#) timer:

```
static struct clock_event_device lapic_clockevent = {
    .name           = "lapic",
    ...
    ...
    ...
}
```

Addresses of the `event_handler`, `set_next_event`, `next_event` functions for a certain clock event device which are an [interrupt handler](#), setter of next event and local storage for next event respectively. Yet another field of the `clock_event_device` structure is - `features` field. Its value maybe on of the following generic features:

```
#define CLOCK_EVT_FEAT_PERIODIC    0x000001
#define CLOCK_EVT_FEAT_ONESHOT     0x000002
```

Where the `CLOCK_EVT_FEAT_PERIODIC` represents device which may be programmed to generate events periodically. The `CLOCK_EVT_FEAT_ONESHOT` represents device which may generate an event only once. Besides these two features, there are also architecture-specific features. For example [x86\\_64](#) supports two additional features:

```
#define CLOCK_EVT_FEAT_C3STOP      0x000008
```

The first `CLOCK_EVT_FEAT_C3STOP` means that a clock event device will be stopped in the [C3](#) state. Additionally the `clock_event_device` structure has `mult` and `shift` fields as well as `clocksource` structure. The `clocksource` structure also contains other fields, but we will consider it later.

After we considered part of the `clock_event_device` structure, time is to look at the `API` of the `clockevents` framework. To work with a clock event device, first of all we need to initialize `clock_event_device` structure and register a clock events device. The `clockevents` framework provides following `API` for registration of clock event devies:

```
void clockevents_register_device(struct clock_event_device *dev)
{
    ...
    ...
    ...
}
```

This function defined in the [kernel/time/clockevents.c](#) source code file and as we may see, the `clockevents_register_device` function takes only one parameter:

- address of a `clock_event_device` structure which represents a clock event device.



So, to register a clock event device, at first we need to initialize `clock_event_device` structure with parameters of a certain clock event device. Let's take a look at one random clock event device in the Linux kernel source code. We can find one in the [drivers/clocksource](#) directory or try to take a look at an architecture-specific clock event device. Let's take for example - [Periodic Interval Timer \(PIT\) for at91sam926x](#). You can find its implementation in the [drivers/clocksource](#).

First of all let's look at initialization of the `clock_event_device` structure. This occurs in the `at91sam926x_pit_common_init` function:

```
struct pit_data {
    ...
    ...
    struct clock_event_device    clkevt;
    ...
    ...
};

static void __init at91sam926x_pit_common_init(struct pit_data *data)
{
    ...
    ...
    ...
    data->clkevt.name = "pit";
    data->clkevt.features = CLOCK_EVT_FEAT_PERIODIC;
    data->clkevt.shift = 32;
    data->clkevt.mult = div_sc(pit_rate, NSEC_PER_SEC, data->clkevt.shift);
    data->clkevt.rating = 100;
    data->clkevt.cpumask = cpumask_of(0);

    data->clkevt.set_state_shutdown = pit_clkevt_shutdown;
    data->clkevt.set_state_periodic = pit_clkevt_set_periodic;
    data->clkevt.resume = at91sam926x_pit_resume;
    data->clkevt.suspend = at91sam926x_pit_suspend;
    ...
}
```

Here we can see that `at91sam926x_pit_common_init` takes one parameter - pointer to the `pit_data` structure which contains `clock_event_device` structure which will contain clock event related information of the `at91sam926x` [periodic Interval Timer](#). At the start we fill `name` of the timer device and its `features`. In our case we deal with periodic timer which as we already know may be programmed to generate events periodically.

The next two fields `shift` and `mult` are familiar to us. They will be used to translate counter of our timer to nanoseconds. After this we set rating of the timer to `100`. This means if there will not be timers with higher rating in the system, this timer will be used for

timekeeping. The next field - `cpumask` indicates for which processors in the system the device will work. In our case, the device will work for the first processor. The `cpumask_of` macro defined in the [include/linux/cpumask.h](#) header file and just expands to the call of the:

```
#define cpumask_of(cpu) (get_cpu_mask(cpu))
```

Where the `get_cpu_mask` returns the cpumask containing just a given `cpu` number. More about `cpumasks` concept you may read in the [CPU masks in the Linux kernel](#) part. In the last four lines of code we set callbacks for the clock event device suspend/resume, device shutdown and update of the clock event device state.

After we finished with the initialization of the `at91sam926x` periodic timer, we can register it by the call of the following functions:

```
clockevents_register_device(&data->clkev);
```

Now we can consider implementation of the `clockevent_register_device` function. As I already wrote above, this function is defined in the [kernel/time/clockevents.c](#) source code file and starts from the initialization of the initial event device state:

```
clockevent_set_state(dev, CLOCK_EVT_STATE_DETACHED);
```

Actually, an event device may be in one of this states:

```
enum clock_event_state {
    CLOCK_EVT_STATE_DETACHED,
    CLOCK_EVT_STATE_SHUTDOWN,
    CLOCK_EVT_STATE_PERIODIC,
    CLOCK_EVT_STATE_ONESHOT,
    CLOCK_EVT_STATE_ONESHOT_STOPPED,
};
```

Where:

- `CLOCK_EVT_STATE_DETACHED` - a clock event device is not not used by `clockevents` framework. Actually it is initial state of all clock event devices;
- `CLOCK_EVT_STATE_SHUTDOWN` - a clock event device is powered-off;
- `CLOCK_EVT_STATE_PERIODIC` - a clock event device may be programmed to generate event periodically;
- `CLOCK_EVT_STATE_ONESHOT` - a clock event device may be programmed to generate event only once;
- `CLOCK_EVT_STATE_ONESHOT_STOPPED` - a clock event device was programmed to generate

event only once and now it is temporary stopped.

The implementation of the `clock_event_set_state` function is pretty easy:

```
static inline void clockevent_set_state(struct clock_event_device *dev,
                                       enum clock_event_state state)
{
    dev->state_use_accessors = state;
}
```

As we can see, it just fills the `state_use_accessors` field of the given `clock_event_device` structure with the given value which is in our case is `CLOCK_EVT_STATE_DETACHED`. Actually all clock event devices has this initial state during registration. The `state_use_accessors` field of the `clock_event_device` structure provides current state of the clock event device.

After we have set initial state of the given `clock_event_device` structure we check that the `cpumask` of the given clock event device is not zero:

```
if (!dev->cpumask) {
    WARN_ON(num_possible_cpus() > 1);
    dev->cpumask = cpumask_of(smp_processor_id());
}
```

Remember that we have set the `cpumask` of the `at91sam926x` periodic timer to first processor. If the `cpumask` field is zero, we check the number of possible processors in the system and print warning message if it is less than one. Additionally we set the `cpumask` of the given clock event device to the current processor. If you are interested in how the `smp_processor_id` macro is implemented, you can read more about it in the fourth [part](#) of the Linux kernel initialization process chapter.

After this check we lock the actual code of the clock event device registration by the call following macros:

```
raw_spin_lock_irqsave(&clockevents_lock, flags);
...
...
...
raw_spin_unlock_irqrestore(&clockevents_lock, flags);
```

Additionally the `raw_spin_lock_irqsave` and the `raw_spin_unlock_irqrestore` macros disable local interrupts, however interrupts on other processors still may occur. We need to do it to prevent potential [deadlock](#) if we adding new clock event device to the list of clock event devices and an interrupt occurs from other clock event device.

We can see following code of clock event device registration between the

`raw_spin_lock_irqsave` and `raw_spin_unlock_irqrestore` macros:

```
list_add(&dev->list, &clockevent_devices);
tick_check_new_device(dev);
clockevents_notify_released();
```

First of all we add the given clock event device to the list of clock event devices which is represented by the `clockevent_devices` :

```
static LIST_HEAD(clockevent_devices);
```

At the next step we call the `tick_check_new_device` function which is defined in the [kernel/time/tick-common.c](#) source code file and checks do the new registered clock event device should be used or not. The `tick_check_new_device` function checks the given `clock_event_device` gets the current registered tick device which is represented by the `tick_device` structure and compares their ratings and features. Actually `CLOCK_EVT_STATE_ONESHOT` is preferred:

```
static bool tick_check_preferred(struct clock_event_device *curdev,
                                struct clock_event_device *newdev)
{
    if (!(newdev->features & CLOCK_EVT_FEAT_ONESHOT)) {
        if (curdev && (curdev->features & CLOCK_EVT_FEAT_ONESHOT))
            return false;
        if (tick_oneshot_mode_active())
            return false;
    }

    return !curdev ||
        newdev->rating > curdev->rating ||
        !cpumask_equal(curdev->cpumask, newdev->cpumask);
}
```

If the new registered clock event device is more preferred than old tick device, we exchange old and new registered devices and install new device:

```
clockevents_exchange_device(curdev, newdev);
tick_setup_device(td, newdev, cpu, cpumask_of(cpu));
```

The `clockevents_exchange_device` function releases or in other words deleted the old clock event device from the `clockevent_devices` list. The next function - `tick_setup_device` as we may understand from its name, setups new tick device. This function check the mode of the

new registered clock event device and call the `tick_setup_periodic` function or the `tick_setup_oneshot` depends on the tick device mode:

```
if (td->mode == TICKDEV_MODE_PERIODIC)
    tick_setup_periodic(newdev, 0);
else
    tick_setup_oneshot(newdev, handler, next_event);
```

Both of this functions calls the `clockevents_switch_state` to change state of the clock event device and the `clockevents_program_event` function to set next event of clock event device based on delta between the maximum and minimum difference current time and time for the next event. The `tick_setup_periodic` :

```
clockevents_switch_state(dev, CLOCK_EVT_STATE_PERIODIC);
clockevents_program_event(dev, next, false);
```

and the `tick_setup_oneshot_periodic` :

```
clockevents_switch_state(newdev, CLOCK_EVT_STATE_ONESHOT);
clockevents_program_event(newdev, next_event, true);
```

The `clockevents_switch_state` function checks that the clock event device is not in the given state and calls the `__clockevents_switch_state` function from the same source code file:

```
if (clockevent_get_state(dev) != state) {
    if (__clockevents_switch_state(dev, state))
        return;
```

The `__clockevents_switch_state` function just makes a call of the certain callback depends on the given state:

```

static int __clockevents_switch_state(struct clock_event_device *dev,
                                     enum clock_event_state state)
{
    if (dev->features & CLOCK_EVT_FEAT_DUMMY)
        return 0;

    switch (state) {
    case CLOCK_EVT_STATE_DETACHED:
    case CLOCK_EVT_STATE_SHUTDOWN:
        if (dev->set_state_shutdown)
            return dev->set_state_shutdown(dev);
        return 0;

    case CLOCK_EVT_STATE_PERIODIC:
        if (!(dev->features & CLOCK_EVT_FEAT_PERIODIC))
            return -ENOSYS;
        if (dev->set_state_periodic)
            return dev->set_state_periodic(dev);
        return 0;
    ...
    ...
    ...

```

In our case for `at91sam926x` periodic timer, the state is the `CLOCK_EVT_FEAT_PERIODIC` :

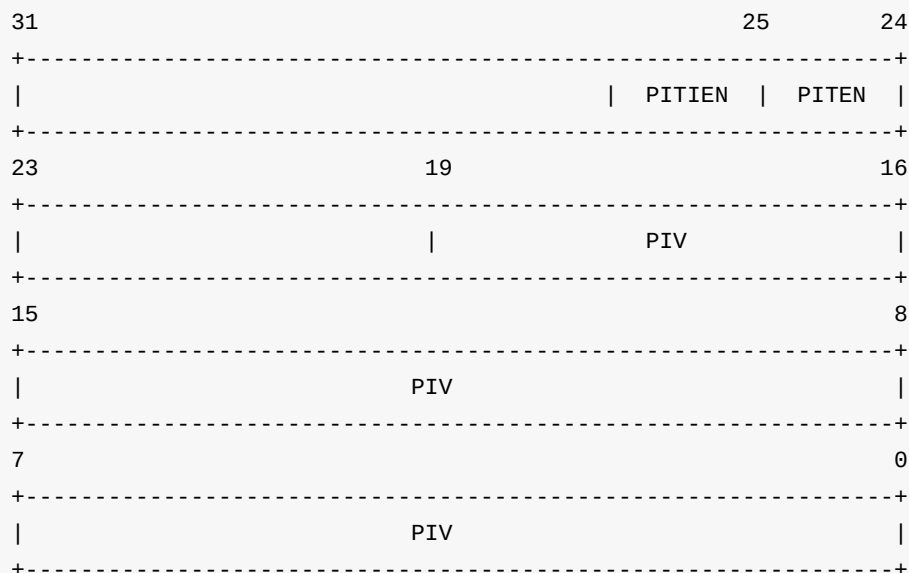
```

data->clkevt.features = CLOCK_EVT_FEAT_PERIODIC;
data->clkevt.set_state_periodic = pit_clkevt_set_periodic;

```

So, for the `pit_clkevt_set_periodic` callback will be called. If we will read the documentation of the [Periodic Interval Timer \(PIT\) for at91sam926x](#), we will see that there is `Periodic Interval Timer Mode Register` which allows us to control of periodic interval timer.

It looks like:



Where `PIV` or `Periodic Interval Value` - defines the value compared with the primary 20-bit counter of the Periodic Interval Timer. The `PITEN` or `Period Interval Timer Enabled` if the bit is `1` and the `PITIEN` or `Periodic Interval Timer Interrupt Enable` if the bit is `1`. So, to set periodic mode, we need to set `24`, `25` bits in the `Periodic Interval Timer Mode Register`. And we are doing it in the `pit_clkevt_set_periodic` function:

```
static int pit_clkevt_set_periodic(struct clock_event_device *dev)
{
    struct pit_data *data = clkevt_to_pit_data(dev);
    ...
    ...
    ...
    pit_write(data->base, AT91_PIT_MR,
              (data->cycle - 1) | AT91_PIT_PITEN | AT91_PIT_PITIEN);

    return 0;
}
```

Where the `AT91_PT_MR` , `AT91_PT_PITEN` and the `AT91_PIT_PITIEN` are declared as:

```
#define AT91_PIT_MR                0x00
#define AT91_PIT_PITIEN            BIT(25)
#define AT91_PIT_PITEN              BIT(24)
```

After the setup of the new clock event device is finished, we can return to the `clockevents_register_device` function. The last function in the `clockevents_register_device` function is:

```
clockevents_notify_released();
```

This function checks the `clockevents_released` list which contains released clock event devices (remember that they may occur after the call of the `clockevents_exchange_device` function). If this list is not empty, we go through clock event devices from the

`clock_events_released` list and delete it from the `clockevent_devices` :

```
static void clockevents_notify_released(void)
{
    struct clock_event_device *dev;

    while (!list_empty(&clockevents_released)) {
        dev = list_entry(clockevents_released.next,
                        struct clock_event_device, list);
        list_del(&dev->list);
        list_add(&dev->list, &clockevent_devices);
        tick_check_new_device(dev);
    }
}
```

That's all. From this moment we have registered new clock event device. So the usage of the `clockevents` framework is simple and clear. Architectures registered their clock event devices, in the clock events core. Users of the clockevents core can get clock event devices for their use. The `clockevents` framework provides notification mechanisms for various clock related management events like a clock event device registered or unregistered, a processor is offlined in system which supports [CPU hotplug](#) and etc.

We saw implementation only of the `clockevents_register_device` function. But generally, the clock event layer [API](#) is small. Besides the `API` for clock event device registration, the `clockevents` framework provides functions to schedule the next event interrupt, clock event device notification service and support for suspend and resume for clock event devices.

If you want to know more about `clockevents` `API` you can start to research following source code and header files: [kernel/time/tick-common.c](#), [kernel/time/clockevents.c](#) and [include/linux/clockchips.h](#).

That's all.

## Conclusion

This is the end of the fifth part of the [chapter](#) that describes timers and timer management related stuff in the Linux kernel. In the previous part got acquainted with the `timers` concept. In this part we continued to learn time management related stuff in the Linux kernel



and saw a little about yet another framework - `clockevents` .

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [timekeeping documentation](#)
- [Intel 8253](#)
- [programmable interval timer](#)
- [ACPI pdf](#)
- [x86](#)
- [High Precision Event Timer](#)
- [powerpc](#)
- [frequency](#)
- [API](#)
- [nanoseconds](#)
- [interrupt](#)
- [interrupt handler](#)
- [local APIC](#)
- [C3 state](#)
- [Periodic Interval Timer \(PIT\) for at91sam926x](#)
- [CPU masks in the Linux kernel](#)
- [deadlock](#)
- [CPU hotplug](#)
- [previous part](#)

# Timers and time management in the Linux kernel. Part 6.

## x86\_64 related clock sources

This is sixth part of the [chapter](#) which describes timers and time management related stuff in the Linux kernel. In the previous [part](#) we saw `clockevents` framework and now we will continue to dive into time management related stuff in the Linux kernel. This part will describe implementation of `x86` architecture related clock sources (more about `clocksource` concept you can read in the [second part](#) of this chapter).

First of all we must know what clock sources may be used at `x86` architecture. It is easy to know from the `sysfs` or from content of the

```
/sys/devices/system/clocksource/clocksource0/available_clocksource . The  
/sys/devices/system/clocksource/clocksourceN provides two special files to achieve this:
```

- `available_clocksource` - provides information about available clock sources in the system;
- `current_clocksource` - provides information about currently used clock source in the system.

So, let's look:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource  
tsc hpet acpi_pm
```

We can see that there are three registered clock sources in my system:

- `tsc` - [Time Stamp Counter](#);
- `hpet` - [High Precision Event Timer](#);
- `acpi_pm` - [ACPI Power Management Timer](#).

Now let's look at the second file which provides best clock source (a clock source which has the best rating in the system):

```
$ cat /sys/devices/system/clocksource/clocksource0/current_clocksource  
tsc
```

For me it is [Time Stamp Counter](#). As we may know from the [second part](#) of this chapter, which describes internals of the `clocksource` framework in the Linux kernel, the best clock source in a system is a clock source with the best (highest) rating or in other words with the highest [frequency](#).

Frequency of the [ACPI](#) power management timer is `3.579545 MHz`. Frequency of the [High Precision Event Timer](#) is at least `10 MHz`. And the frequency of the [Time Stamp Counter](#) depends on processor. For example On older processors, the `Time Stamp Counter` was counting internal processor clock cycles. This means its frequency changed when the processor's frequency scaling changed. The situation has changed for newer processors. Newer processors have an `invariant Time Stamp counter` that increments at a constant rate in all operational states of processor. Actually we can get its frequency in the output of the `/proc/cpuinfo`. For example for the first processor in the system:

```
$ cat /proc/cpuinfo
...
model name      : Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz
...
```

And although Intel manual says that the frequency of the `Time Stamp Counter`, while constant, is not necessarily the maximum qualified frequency of the processor, or the frequency given in the brand string, anyway we may see that it will be much more than frequency of the `ACPI PM` timer or `High Precision Event Timer`. And we can see that the clock source with the best rating or highest frequency is current in the system.

You can note that besides these three clock source, we don't see yet another two familiar us clock sources in the output of the

`/sys/devices/system/clocksource/clocksource0/available_clocksource`. These clock sources are `jiffy` and `refined_jiffies`. We don't see them because this file maps only high resolution clock sources or in other words clock sources with the [CLOCK\\_SOURCE\\_VALID\\_FOR\\_HRES](#) flag.

As I already wrote above, we will consider all of these three clock sources in this part. We will consider it in order of their initialization or:

- `hpet` ;
- `acpi_pm` ;
- `tsc` .

We can make sure that the order is exactly like this in the output of the [dmesg](#) util:

```
$ dmesg | grep clocksource
[ 0.000000] clocksource: refined-jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max
[ 0.000000] clocksource: hpet: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 1
[ 0.094369] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns
[ 0.186498] clocksource: Switched to clocksource hpet
[ 0.196827] clocksource: acpi_pm: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 20
[ 1.413685] tsc: Refined TSC clocksource calibration: 3999.981 MHz
[ 1.413688] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x73509721780, max_
[ 2.413748] clocksource: Switched to clocksource tsc
```

The first clock source is the [High Precision Event Timer](#), so let's start from it.

## High Precision Event Timer

The implementation of the `High Precision Event Timer` for the `x86` architecture is located in the `arch/x86/kernel/hpet.c` source code file. Its initialization starts from the call of the `hpet_enable` function. This function is called during Linux kernel initialization. If we will look into `start_kernel` function from the `init/main.c` source code file, we will see that after the all architecture-specific stuff initialized, early console is disabled and time management subsystem already ready, call of the following function:

```
if (late_time_init)
    late_time_init();
```

which does initialization of the late architecture specific timers after early jiffy counter already initialized. The definition of the `late_time_init` function for the `x86` architecture is located in the `arch/x86/kernel/time.c` source code file. It looks pretty easy:

```
static __init void x86_late_time_init(void)
{
    x86_init.timers.timer_init();
    tsc_init();
}
```

As we may see, it does initialization of the `x86` related timer and initialization of the `Time Stamp Counter`. The seconds we will see in the next paragraph, but now let's consider the call of the `x86_init.timers.timer_init` function. The `timer_init` points to the `hpet_time_init` function from the same source code file. We can verify this by looking on the definition of the `x86_init` structure from the `arch/x86/kernel/x86_init.c`:

```

struct x86_init_ops x86_init __initdata = {
    ...
    ...
    ...
    .timers = {
        .setup_percpu_clockev    = setup_boot_APIC_clock,
        .timer_init              = hpet_time_init,
        .wallclock_init          = x86_init_noop,
    },
    ...
    ...
    ...
}

```

The `hpet_time_init` function does setup of the [programmable interval timer](#) if we can not enable `High Precision Event Timer` and setups default timer [IRQ](#) for the enabled timer:

```

void __init hpet_time_init(void)
{
    if (!hpet_enable())
        setup_pit_timer();
    setup_default_timer_irq();
}

```

First of all the `hpet_enable` function check we can enable `High Precision Event Timer` in the system by the call of the `is_hpet_capable` function and if we can, we map a virtual address space for it:

```

int __init hpet_enable(void)
{
    if (!is_hpet_capable())
        return 0;

    hpet_set_mapping();
}

```

The `is_hpet_capable` function checks that we didn't pass `hpet=disable` to the kernel command line and the `hpet_address` is received from the [ACPI HPET](#) table. The `hpet_set_mapping` function just maps the virtual address spaces for the timer registers:

```

hpet_virt_address = ioremap_nocache(hpet_address, HPET_MMAP_SIZE);

```

As we can read in the [IA-PC HPET \(High Precision Event Timers\) Specification](#):

The timer register space is 1024 bytes

So, the `HPET_MMAP_SIZE` is `1024` bytes too:

```
#define HPET_MMAP_SIZE    1024
```

After we mapped virtual space for the `High Precision Event Timer`, we read `HPET_ID` register to get number of the timers:

```
id = hpet_readl(HPET_ID);

last = (id & HPET_ID_NUMBER) >> HPET_ID_NUMBER_SHIFT;
```

We need to get this number to allocate correct amount of space for the `General Configuration Register` of the `High Precision Event Timer` :

```
cfg = hpet_readl(HPET_CFG);

hpet_boot_cfg = kmalloc((last + 2) * sizeof(*hpet_boot_cfg), GFP_KERNEL);
```

After the space is allocated for the configuration register of the `High Precision Event Timer`, we allow to main counter to run, and allow timer interrupts if they are enabled by the setting of `HPET_CFG_ENABLE` bit in the configuration register for all timers. In the end we just register new clock source by the call of the `hpet_clocksource_register` function:

```
if (hpet_clocksource_register())
    goto out_nohpet;
```

which just calls already familiar

```
clocksource_register_hz(&clocksource_hpet, (u32)hpet_freq);
```

function. Where the `clocksource_hpet` is the `clocksource` structure with the rating `250` (remember rating of the previous `refined_jiffies` clock source was `2`), name - `hpet` and `read_hpet` callback for the reading of atomic counter provided by the `High Precision Event Timer` :

```
static struct clocksource clocksource_hpet = {
    .name      = "hpet",
    .rating     = 250,
    .read      = read_hpet,
    .mask      = HPET_MASK,
    .flags     = CLOCK_SOURCE_IS_CONTINUOUS,
    .resume    = hpet_resume_counter,
    .archdata  = { .vclock_mode = VCLOCK_HPET },
};
```

After the `clocksource_hpet` is registered, we can return to the `hpet_time_init()` function from the [arch/x86/kernel/time.c](#) source code file. We can remember that the last step is the call of the:

```
setup_default_timer_irq();
```

function in the `hpet_time_init()`. The `setup_default_timer_irq` function checks existence of legacy IRQs or in other words support for the [i8259](#) and setups [IRQ0](#) depends on this.

That's all. From this moment the [High Precision Event Timer](#) clock source registered in the Linux kernel `clock source` framework and may be used from generic kernel code via the `read_hpet`:

```
static cycle_t read_hpet(struct clocksource *cs)
{
    return (cycle_t)hpet_readl(HPET_COUNTER);
}
```

function which just reads and returns atomic counter from the `Main Counter Register`.

## ACPI PM timer

The seconds clock source is [ACPI Power Management Timer](#). Implementation of this clock source is located in the [drivers/clocksource/acpi\\_pm.c](#) source code file and starts from the call of the `init_acpi_pm_clocksource` function during `fs initcall`.

If we will look at implementation of the `init_acpi_pm_clocksource` function, we will see that it starts from the check of the value of `pmtmr_ioport` variable:

```
static int __init init_acpi_pm_clocksource(void)
{
    ...
    ...
    ...
    if (!pmtmr_ioport)
        return -ENODEV;
    ...
    ...
    ...
}
```

This `pmtmr_ioport` variable contains extended address of the Power Management Timer Control Register Block . It gets its value in the `acpi_parse_fadt` function which is defined in the [arch/x86/kernel/acpi/boot.c](#) source code file. This function parses FADT or Fixed ACPI Description Table ACPI table and tries to get the values of the `X_PM_TMR_BLK` field which contains extended address of the Power Management Timer Control Register Block , represented in Generic Address Structure format:

```
static int __init acpi_parse_fadt(struct acpi_table_header *table)
{
    #ifdef CONFIG_X86_PM_TIMER
        ...
        ...
        ...
        pmtmr_ioport = acpi_gbl_FADT.xpm_timer_block.address;
        ...
        ...
        ...
    #endif
    return 0;
}
```

So, if the `CONFIG_X86_PM_TIMER` Linux kernel configuration option is disabled or something going wrong in the `acpi_parse_fadt` function, we can't access the Power Management Timer register and return from the `init_acpi_pm_clocksource` . In other way, if the value of the `pmtmr_ioport` variable is not zero, we check rate of this timer and register this clock source by the call of the:

```
clocksource_register_hz(&clocksource_acpi_pm, PMTMR_TICKS_PER_SEC);
```

function. After the call of the `clocksource_register_hz` , the `acpi_pm` clock source will be registered in the `clocksource` framework of the Linux kernel:



```
static struct clocksource clocksource_acpi_pm = {
    .name      = "acpi_pm",
    .rating     = 200,
    .read      = acpi_pm_read,
    .mask      = (cycle_t)ACPI_PM_MASK,
    .flags     = CLOCK_SOURCE_IS_CONTINUOUS,
};
```

with the rating - 200 and the `acpi_pm_read` callback to read atomic counter provided by the `acpi_pm` clock source. The `acpi_pm_read` function just executes `read_pmtmr` function:

```
static cycle_t acpi_pm_read(struct clocksource *cs)
{
    return (cycle_t)read_pmtmr();
}
```

which reads value of the `Power Management Timer` register. This register has following structure:

+-----+-----+	
eupper eight bits of a	running count of the
32-bit power management timer	power management timer
+-----+-----+	
31	0
E_TMR_VAL	TMR_VAL

Address of this register is stored in the `Fixed ACPI Description Table` [ACPI](#) table and we already have it in the `pmtmr_ioport`. So, the implementation of the `read_pmtmr` function is pretty easy:

```
static inline u32 read_pmtmr(void)
{
    return inl(pmtmr_ioport) & ACPI_PM_MASK;
}
```

We just read the value of the `Power Management Timer` register and mask its 24 bits.

That's all. Now we move to the last clock source in this part - `Time Stamp Counter`.

## Time Stamp Counter

The third and last clock source in this part is - [Time Stamp Counter](#) clock source and its implementation is located in the [arch/x86/kernel/tsc.c](#) source code file. We already saw the `x86_early_time_init` function in this part and initialization of the [Time Stamp Counter](#) starts from this place. This function calls the `tsc_init()` function from the [arch/x86/kernel/tsc.c](#) source code file.

At the beginning of the `tsc_init` function we can see check, which checks that a processor has support of the `Time Stamp Counter` :

```
void __init tsc_init(void)
{
    u64 lpj;
    int cpu;

    if (!cpu_has_tsc) {
        setup_clear_cpu_cap(X86_FEATURE_TSC_DEADLINE_TIMER);
        return;
    }
    ...
    ...
    ...
}
```

The `cpu_has_tsc` macro expands to the call of the `cpu_has` macro:

```
#define cpu_has_tsc        boot_cpu_has(X86_FEATURE_TSC)

#define boot_cpu_has(bit)   cpu_has(&boot_cpu_data, bit)

#define cpu_has(c, bit)    \
    (__builtin_constant_p(bit) && REQUIRED_MASK_BIT_SET(bit) ? 1 : \
    test_cpu_cap(c, bit))
```

which check the given bit (the `X86_FEATURE_TSC_DEADLINE_TIMER` in our case) in the `boot_cpu_data` array which is filled during early Linux kernel initialization. If the processor has support of the `Time Stamp Counter` , we get the frequency of the `Time Stamp Counter` by the call of the `calibrate_tsc` function from the same source code file which tries to get frequency from the different source like [Model Specific Register](#), [programmable interval timer](#) and etc, after this we initialize frequency and scale factor for the all processors in the system:

```
tsc_khz = x86_platform.calibrate_tsc();
cpu_khz = tsc_khz;

for_each_possible_cpu(cpu) {
    cyc2ns_init(cpu);
    set_cyc2ns_scale(cpu_khz, cpu);
}
```

because only first bootstrap processor will call the `tsc_init`. After this we check that `Time Stamp Counter` is not disabled:

```
if (tsc_disabled > 0)
    return;
...
...
...
check_system_tsc_reliable();
```

and call the `check_system_tsc_reliable` function which sets the `tsc_clocksource_reliable` if bootstrap processor has the `X86_FEATURE_TSC_RELIABLE` feature. Note that we went through the `tsc_init` function, but did not register our clock source. Actual registration of the `Time Stamp Counter` clock source occurs in the:

```
static int __init init_tsc_clocksource(void)
{
    if (!cpu_has_tsc || tsc_disabled > 0 || !tsc_khz)
        return 0;
    ...
    ...
    ...
    if (boot_cpu_has(X86_FEATURE_TSC_RELIABLE)) {
        clocksource_register_khz(&clocksource_tsc, tsc_khz);
        return 0;
    }
}
```

function. This function called during the `device initcall`. We do it to be sure that the `Time Stamp Counter` clock source will be registered after the [High Precision Event Timer](#) clock source.

After these all three clock sources will be registered in the `clocksource` framework and the `Time Stamp Counter` clock source will be selected as active, because it has the highest rating among other clock sources:

```
static struct clocksource clocksource_tsc = {  
    .name           = "tsc",  
    .rating          = 300,  
    .read            = read_tsc,  
    .mask            = CLOCKSOURCE_MASK(64),  
    .flags           = CLOCK_SOURCE_IS_CONTINUOUS | CLOCK_SOURCE_MUST_VERIFY,  
    .archdata        = { .vclock_mode = VCLOCK_TSC },  
};
```

That's all.

## Conclusion

This is the end of the sixth part of the [chapter](#) that describes timers and timer management related stuff in the Linux kernel. In the previous part got acquainted with the `clockevents` framework. In this part we continued to learn time management related stuff in the Linux kernel and saw a little about three different clock sources which are used in the [x86](#) architecture. The next part will be last part of this [chapter](#) and we will see some user space related stuff, i.e. how some time related [system calls](#) implemented in the Linux kernel.

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [x86](#)
- [sysfs](#)
- [Time Stamp Counter](#)
- [High Precision Event Timer](#)
- [ACPI Power Management Timer \(PDF\)](#)
- [frequency](#).
- [dmesg](#)
- [programmable interval timer](#)
- [IRQ](#)
- [IA-PC HPET \(High Precision Event Timers\) Specification](#)
- [IRQ0](#)
- [i8259](#)
- [initcall](#)

- [previous part](#)

# Timers and time management in the Linux kernel. Part 7.

## Time related system calls in the Linux kernel

This is the seventh and last part [chapter](#) which describes timers and time management related stuff in the Linux kernel. In the previous [part](#) we saw some `x86_64` like [High Precision Event Timer](#) and [Time Stamp Counter](#). Internal time management is interesting part of the Linux kernel, but of course not only the kernel needs in the `time` concept. Our programs need to know time too. In this part, we will consider implementation of some time management related [system calls](#). These system calls are:

- `clock_gettime` ;
- `gettimeofday` ;
- `nanosleep` .

We will start from simple userspace `C` program and see all way from the call of the [standard library](#) function to the implementation of certain system call. As each [architecture](#) provides its own implementation of certain system call, we will consider only `x86_64` specific implementations of system calls, as this book is related to this architecture.

Additionally we will not consider concept of system calls in this part, but only implementations of these three system calls in the Linux kernel. If you are interested in what is it a `system call` , there is special [chapter](#) about this.

So, let's from the `gettimeofday` system call.

## Implementation of the `gettimeofday` system call

As we can understand from the name of the `gettimeofday` , this function returns current time. First of all, let's look on the following simple example:

```
#include <time.h>
#include <sys/time.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    char buffer[40];
    struct timeval time;

    gettimeofday(&time, NULL);

    strftime(buffer, 40, "Current date/time: %m-%d-%Y/%T", localtime(&time.tv_sec));
    printf("%s\n", buffer);

    return 0;
}
```

As you can see, here we call the `gettimeofday` function which takes two parameters: pointer to the `timeval` structure which represents an elapsed time:

```
struct timeval {
    time_t      tv_sec;      /* seconds */
    suseconds_t tv_usec;     /* microseconds */
};
```

The second parameter of the `gettimeofday` function is pointer to the `timezone` structure which represents a timizone. In our example, we pass address of the `timeval time` to the `gettimeofday` function, the Linux kernel fills the given `timeval` structure and returns it back to us. Additionally, we format the time with the `strftime` function to get something more human readable than elapsed microseconds. Let's see on result:

```
~$ gcc date.c -o date
~$ ./date
Current date/time: 03-26-2016/16:42:02
```

As you already may know, an userspace application does not call a system call directly from the kernel space. Before the actual system call entry will be called, we call a function from the standard library. In my case it is `glibc`, so I will consider this case. The implementation of the `gettimeofday` function is located in this [sysdeps/unix/sysv/linux/x86/gettimeofday.c](#) source code file. As you already may know, the `gettimeofday` is not usual system call. It is located in the special area which is called `vdsO` (you can read more about it in the [part](#) which describes this concept).

The `glibc` implementation of the `gettimeofday` tries to resolve the given symbol, in our case this symbol is `__vdso_gettimeofday` by the call of the `_dl_vdso_vsym` internal function. If the symbol will not be resolved, it returns `NULL` and we fallback to the call of the usual system call:

```
return (_dl_vdso_vsym ("__vdso_gettimeofday", &linux26)
?: (void*) (&__gettimeofday_syscall));
```

The `gettimeofday` entry is located in the [arch/x86/entry/vdso/vclock\\_gettime.c](#) source code file. As we can see the `gettimeofday` is weak alias of the `__vdso_gettimeofday`:

```
int gettimeofday(struct timeval *, struct timezone *)
    __attribute__((weak, alias("__vdso_gettimeofday")));
```

The `__vdso_gettimeofday` is defined in the same source code file and calls the `do_realtime` function if the given `timeval` is not null:

```
notrace int __vdso_gettimeofday(struct timeval *tv, struct timezone *tz)
{
    if (likely(tv != NULL)) {
        if (unlikely(do_realtime((struct timespec *)tv) == VCLOCK_NONE))
            return vdso_fallback_gtod(tv, tz);
        tv->tv_usec /= 1000;
    }
    if (unlikely(tz != NULL)) {
        tz->tz_minuteswest = gtod->tz_minuteswest;
        tz->tz_dsttime = gtod->tz_dsttime;
    }

    return 0;
}
```

If the `do_realtime` will fail, we fallback to the real system call via call the `syscall` instruction and passing the `__NR_gettimeofday` system call number and the given `timeval` and `timezone`:

```
notrace static long vdso_fallback_gtod(struct timeval *tv, struct timezone *tz)
{
    long ret;

    asm("syscall" : "=a" (ret) :
        "0" (__NR_gettimeofday), "D" (tv), "S" (tz) : "memory");
    return ret;
}
```



The `do_realtime` function gets the time data from the `vsyscall_gtod_data` structure which is defined in the [arch/x86/include/asm/vgtod.h](#) header file and contains mapping of the `timespec` structure and a couple of fields which are related to the current clock source in the system. This function fills the given `timeval` structure with values from the `vsyscall_gtod_data` which contains a time related data which is updated via timer interrupt.

First of all we try to access the `gtod` or global time of day the `vsyscall_gtod_data` structure via the call of the `gtod_read_begin` and will continue to do it until it will be successful:

```
do {
    seq = gtod_read_begin(gtod);
    mode = gtod->vclock_mode;
    ts->tv_sec = gtod->wall_time_sec;
    ns = gtod->wall_time_nsec;
    ns += vgetsns(&mode);
    ns >>= gtod->shift;
} while (unlikely(gtod_read_retry(gtod, seq)));

ts->tv_sec += __iter_div_u64_rem(ns, NSEC_PER_SEC, &ns);
ts->tv_nsec = ns;
```

As we got access to the `gtod`, we fill the `ts->tv_sec` with the `gtod->wall_time_sec` which stores current time in seconds gotten from the [real time clock](#) during initialization of the timekeeping subsystem in the Linux kernel and the same value but in nanoseconds. In the end of this code we just fill the given `timespec` structure with the resulted values.

That's all about the `gettimeofday` system call. The next system call in our list is the `clock_gettime`.

## Implementation of the `clock_gettime` system call

The `clock_gettime` function gets the time which is specified by the second parameter. Generally the `clock_gettime` function takes two parameters:

- `clk_id` - clock identifier;
- `timespec` - address of the `timespec` structure which represent elapsed time.

Let's look on the following simple example:

```
#include <time.h>
#include <sys/time.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    struct timespec elapsed_from_boot;

    clock_gettime(CLOCK_BOOTTIME, &elapsed_from_boot);

    printf("%d - seconds elapsed from boot\n", elapsed_from_boot.tv_sec);

    return 0;
}
```

which prints `uptime` information:

```
~$ gcc uptime.c -o uptime
~$ ./uptime
14180 - seconds elapsed from boot
```

We can easily check the result with the help of the `uptime` util:

```
~$ uptime
up 3:56
```

The `elapsed_from_boot.tv_sec` represents elapsed time in seconds, so:

```
>>> 14180 / 60
236
>>> 14180 / 60 / 60
3
>>> 14180 / 60 % 60
56
```

The `clock_id` maybe one of the following:

- `CLOCK_REALTIME` - system wide clock which measures real or wall-clock time;
- `CLOCK_REALTIME_COARSE` - faster version of the `CLOCK_REALTIME` ;
- `CLOCK_MONOTONIC` - represents monotonic time since some unspecified starting point;
- `CLOCK_MONOTONIC_COARSE` - faster version of the `CLOCK_MONOTONIC` ;
- `CLOCK_MONOTONIC_RAW` - the same as the `CLOCK_MONOTONIC` but provides non [NTP](#) adjusted time.
- `CLOCK_BOOTTIME` - the same as the `CLOCK_MONOTONIC` but plus time that the system was suspended;

- `CLOCK_PROCESS_CPUTIME_ID` - per-process time consumed by all threads in the process;
- `CLOCK_THREAD_CPUTIME_ID` - thread-specific clock.

The `clock_gettime` is not usual syscall too, but as the `gettimeofday`, this system call is placed in the `vdso` area. Entry of this system call is located in the same source code file - [arch/x86/entry/vdso/vclock\\_gettime.c](#)) as for `gettimeofday`.

The Implementation of the `clock_gettime` depends on the clock id. If we have passed the `CLOCK_REALTIME` clock id, the `do_realtime` function will be called:

```
notrace int __vdso_clock_gettime(clockid_t clock, struct timespec *ts)
{
    switch (clock) {
        case CLOCK_REALTIME:
            if (do_realtime(ts) == VCLOCK_NONE)
                goto fallback;
            break;
        ...
        ...
        ...
    fallback:
        return vdso_fallback_gettime(clock, ts);
    }
```

In other cases, the `do_{name_of_clock_id}` function is called. Implementations of some of them is similar. For example if we will pass the `CLOCK_MONOTONIC` clock id:

```
...
...
...
case CLOCK_MONOTONIC:
    if (do_monotonic(ts) == VCLOCK_NONE)
        goto fallback;
    break;
...
...
...
```

the `do_monotonic` function will be called which is very similar on the implementation of the `do_realtime`:

```

notrace static int __always_inline do_monotonic(struct timespec *ts)
{
    do {
        seq = gtod_read_begin(gtod);
        mode = gtod->vclock_mode;
        ts->tv_sec = gtod->monotonic_time_sec;
        ns = gtod->monotonic_time_nsec;
        ns += vgetsns(&mode);
        ns >>= gtod->shift;
    } while (unlikely(gtod_read_retry(gtod, seq)));

    ts->tv_sec += __iter_div_u64_rem(ns, NSEC_PER_SEC, &ns);
    ts->tv_nsec = ns;

    return mode;
}

```

We already saw a little about the implementation of this function in the previous paragraph about the `gettimeofday`. There is only one difference here, that the `sec` and `nsec` of our `timespec` value will be based on the `gtod->monotonic_time_sec` instead of `gtod->wall_time_sec` which maps the value of the `tk->tkr_mono.xtime_nsec` or number of **nanoseconds** elapsed.

That's all.

## Implementation of the `nanosleep` system call

The last system call in our list is the `nanosleep`. As you can understand from its name, this function provides `sleeping` ability. Let's look on the following simple example:

```

#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main (void)
{
    struct timespec ts = {5,0};

    printf("sleep five seconds\n");
    nanosleep(&ts, NULL);
    printf("end of sleep\n");

    return 0;
}

```

If we will compile and run it, we will see the first line

```
~$ gcc sleep_test.c -o sleep
~$ ./sleep
sleep five seconds
end of sleep
```

and the second line after five seconds.

The `nanosleep` is not located in the `vDSO` area like the `gettimeofday` and the `clock_gettime` functions. So, let's look how the `real` system call which is located in the kernel space will be called by the standard library. The implementation of the `nanosleep` system call will be called with the help of the `syscall` instruction. Before the execution of the `syscall` instruction, parameters of the system call must be put in processor `registers` according to order which is described in the [System V Application Binary Interface](#) or in other words:

- `rdi` - first parameter;
- `rsi` - second parameter;
- `rdx` - third parameter;
- `r10` - fourth parameter;
- `r8` - fifth parameter;
- `r9` - sixth parameter.

The `nanosleep` system call has two parameters - two pointers to the `timespec` structures. The system call suspends the calling thread until the given timeout has elapsed. Additionally it will finish if a signal interrupts its execution. It takes two parameters, the first is `timespec` which represents timeout for the sleep. The second parameter is the pointer to the `timespec` structure too and it contains remainder of time if the call of the `nanosleep` was interrupted.

As `nanosleep` has two parameters:

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

To call system call, we need put the `req` to the `rdi` register, and the `rem` parameter to the `rsi` register. The `glibc` does these job in the `INTERNAL_SYSCALL` macro which is located in the `sysdeps/unix/sysv/linux/x86_64/sysdep.h` header file.

```
# define INTERNAL_SYSCALL(name, err, nr, args...) \
INTERNAL_SYSCALL_NCS (__NR_##name, err, nr, ##args)
```

which takes the name of the system call, storage for possible error during execution of system call, number of the system call (all `x86_64` system calls you can find in the [system calls table](#)) and arguments of certain system call. The `INTERNAL_SYSCALL` macro just expands to the call of the `INTERNAL_SYSCALL_NCS` macro, which prepares arguments of system call (puts them into the processor registers in correct order), executes `syscall` instruction and returns the result:

```
# define INTERNAL_SYSCALL_NCS(name, err, nr, args...) \
({ \
    unsigned long int resultvar; \
    LOAD_ARGS_##nr (args) \
    LOAD_REGS_##nr \
    asm volatile ( \
        "syscall\n\t" \
        : "=a" (resultvar) \
        : "0" (name) ASM_ARGS_##nr : "memory", REGISTERS_CLOBBERED_BY_SYSCALL); \
    (long int) resultvar; })
```

The `LOAD_ARGS_##nr` macro calls the `LOAD_ARGS_N` macro where the `N` is number of arguments of the system call. In our case, it will be the `LOAD_ARGS_2` macro. Ultimately all of these macros will be expanded to the following:

```
# define LOAD_REGS_TYPES_1(t1, a1) \
    register t1 _a1 asm ("rdi") = __arg1; \
    LOAD_REGS_0

# define LOAD_REGS_TYPES_2(t1, a1, t2, a2) \
    register t2 _a2 asm ("rsi") = __arg2; \
    LOAD_REGS_TYPES_1(t1, a1)
...
...
...
```

After the `syscall` instruction will be executed, the [context switch](#) will occur and the kernel will transfer execution to the system call handler. The system call handler for the `nanosleep` system call is located in the [kernel/time/hrtimer.c](#) source code file and defined with the `SYSCALL_DEFINE2` macro helper:

```

SYSCALL_DEFINE2(nanosleep, struct timespec __user *, rqtp,
                struct timespec __user *, rmtp)
{
    struct timespec tu;

    if (copy_from_user(&tu, rqtp, sizeof(tu)))
        return -EFAULT;

    if (!timespec_valid(&tu))
        return -EINVAL;

    return hrtimer_nanosleep(&tu, rmtp, HRTIMER_MODE_REL, CLOCK_MONOTONIC);
}

```

More about the `SYSCALL_DEFINE2` macro you may read in the [chapter](#) about system calls. If we look at the implementation of the `nanosleep` system call, first of all we will see that it starts from the call of the `copy_from_user` function. This function copies the given data from the userspace to kernelspace. In our case we copy timeout value to sleep to the kernelspace `timespec` structure and check that the given `timespec` is valid by the call of the `timespec_valid` function:

```

static inline bool timespec_valid(const struct timespec *ts)
{
    if (ts->tv_sec < 0)
        return false;
    if ((unsigned long)ts->tv_nsec >= NSEC_PER_SEC)
        return false;
    return true;
}

```

which just checks that the given `timespec` does not represent date before 1970 and nanoseconds does not overflow 1 second. The `nanosleep` function ends with the call of the `hrtimer_nanosleep` function from the same source code file. The `hrtimer_nanosleep` function creates a [timer](#) and calls the `do_nanosleep` function. The `do_nanosleep` does main job for us. This function provides loop:

```
do {
    set_current_state(TASK_INTERRUPTIBLE);
    hrtimer_start_expires(&t->timer, mode);

    if (likely(t->task))
        freezable_schedule();

} while (t->task && !signal_pending(current));

__set_current_state(TASK_RUNNING);
return t->task == NULL;
```

Which freezes current task during sleep. After we set `TASK_INTERRUPTIBLE` flag for the current task, the `hrtimer_start_expires` function starts the give high-resolution timer on the current processor. As the given high resolution timer will expire, the task will be again running.

That's all.

## Conclusion

This is the end of the seventh part of the [chapter](#) that describes timers and timer management related stuff in the Linux kernel. In the previous part we saw [x86\\_64](#) specific clock sources. As I wrote in the beginning, this part is the last part of this chapter. We saw important time management related concepts like `clocksource` and `clockevents` frameworks, `jiffies` counter and etc., in this chpater. Of course this does not cover all of the time management in the Linux kernel. Many parts of this mostly related to the scheduling which we will see in other chapter.

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [system call](#)
- [C programming language](#)
- [standard library](#)
- [glibc](#)
- [real time clock](#)



- [NTP](#)
- [nanoseconds](#)
- [register](#)
- [System V Application Binary Interface](#)
- [context switch](#)
- [Introduction to timers in the Linux kernel](#)
- [uptime](#)
- [system calls table for x86\\_64](#)
- [High Precision Event Timer](#)
- [Time Stamp Counter](#)
- [x86\\_64](#)
- [previous part](#)

# Synchronization primitives in the Linux kernel.

This chapter describes synchronization primitives in the Linux kernel.

- [Introduction to spinlocks](#) - the first part of this chapter describes implementation of spinlock mechanism in the Linux kernel.

# Synchronization primitives in the Linux kernel. Part 1.

## Introduction

This part opens new chapter in the [linux-insides](#) book. Timers and time management related stuff was described in the previous [chapter](#). Now time to go next. As you may understand from the part's title, this chapter will describe [synchronization](#) primitives in the Linux kernel.

As always, before we will consider something synchronization related, we will try to know what is `synchronization primitive` in general. Actually, synchronization primitive is a software mechanism which provides ability to two or more [parallel](#) processes or threads to not execute simultaneously one the same segment of a code. For example let's look on the following piece of code:

```
mutex_lock(&clocksource_mutex);
...
...
...
clocksource_enqueue(cs);
clocksource_enqueue_watchdog(cs);
clocksource_select();
...
...
...
mutex_unlock(&clocksource_mutex);
```

from the [kernel/time/clocksource.c](#) source code file. This code is from the `__clocksource_register_scale` function which adds the given [clocksource](#) to the list clock sources list. This function produces different operations on a list with registered clock sources. For example the `clocksource_enqueue` function adds the given clock source to the list with registered clocksources - `clocksource_list`. Note that these linse of code wrapped to two functions: `mutex_lock` and `mutex_unlock` which are takes one parameter - the `clocksource_mutex` in our case.

These functions represents locking and unlocking based on [mutex](#) synchronization primitive. As `mutex_lock` will be executed, it allows us to prevent situation when two or more threads will execute this code while the `mute_unlock` will not be executed by process-owner of the mutex. In other words, we prevent parallel operations on a `clocksource_list`. Why do we need `mutex` here? What if two parallel processes will try to register a clock source. As we

already know, the `clocksource_enqueue` function adds the given clock source to the `clocksource_list` list right after a clock source in the list which has the biggest rating (a registered clock source which has the highest frequency in the system):

```
static void clocksource_enqueue(struct clocksource *cs)
{
    struct list_head *entry = &clocksource_list;
    struct clocksource *tmp;

    list_for_each_entry(tmp, &clocksource_list, list)
        if (tmp->rating >= cs->rating)
            entry = &tmp->list;
    list_add(&cs->list, entry);
}
```

If two parallel processes will try to do it simultaneously, both process may found the same `entry` may occur [race condition](#) or in other words, the second process which will execute `list_add`, will overwrite a clock source from first thread.

Besides this simple example, synchronization primitives are ubiquitous in the Linux kernel. If we will go through the previous [chapter](#) or other chapters again or if we will look at the Linux kernel source code in general, we will meet many places like this. We will not consider how `mutex` is implemented in the Linux kernel. Actually, the Linux kernel provides a set of different synchronization primitives like:

- `mutex` ;
- `semaphores` ;
- `seqlocks` ;
- `atomic operations` ;
- etc.

We will start this chapter from the `spinlock`.

## Spinlocks in the Linux kernel.

The `spinlock` is a low-level synchronization mechanism which in simple words, represents a variable which can be in two states:

- `acquired` ;
- `released` .

Each process which wants to acquire a `spinlock`, must write a value which represents `spinlock acquired` state to this variable and write `spinlock released` state to the variable. If a process tries to execute code which is protected by a `spinlock`, it will be locked while a

process which holds this lock will release it. In this case all related operations must be **atomic** to prevent **race conditions** state. The `spinlock` is represented by the `spinlock_t` type in the Linux kernel. If we will look at the Linux kernel code, we will see that this type is **widely** used. The `spinlock_t` is defined as:

```
typedef struct spinlock {
    union {
        struct raw_spinlock rlock;

#ifdef CONFIG_DEBUG_LOCK_ALLOC
        #define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
        struct {
            u8 __padding[LOCK_PADSIZE];
            struct lockdep_map dep_map;
        };
#endif
    };
} spinlock_t;
```

and located in the `include/linux/spinlock_types.h` header file. We may see that its implementation depends on the state of the `CONFIG_DEBUG_LOCK_ALLOC` kernel configuration option. We will skip this now, because all debugging related stuff will be in the end of this part. So, if the `CONFIG_DEBUG_LOCK_ALLOC` kernel configuration option is disabled, the `spinlock_t` contains **union** with one field which is - `raw_spinlock` :

```
typedef struct spinlock {
    union {
        struct raw_spinlock rlock;
    };
} spinlock_t;
```

The `raw_spinlock` structure defined in the **same** header file and represents implementation of **normal** `spinlock`. Let's look how the `raw_spinlock` structure is defined:

```
typedef struct raw_spinlock {
    arch_spinlock_t raw_lock;
#ifdef CONFIG_GENERIC_LOCKBREAK
    unsigned int break_lock;
#endif
} raw_spinlock_t;
```

where the `arch_spinlock_t` represents architecture-specific `spinlock` implementation and the `break_lock` field which holds value - `1` in a case when one processor starts to wait while the lock is held on another processor on **SMP** systems. This allows prevent long time

locking. As consider the `x86_64` architecture in this books, so the `arch_spinlock_t` is defined in the `arch/x86/include/asm/spinlock_types.h` header file and looks:

```
#ifdef CONFIG_QUEUEUED_SPINLOCKS
#include <asm-generic/qspinlock_types.h>
#else
typedef struct arch_spinlock {
    union {
        __ticketpair_t head_tail;
        struct __raw_tickets {
            __ticket_t head, tail;
        } tickets;
    };
} arch_spinlock_t;
```

As we may see, the definition of the `arch_spinlock` structure depends on the value of the `COFNIG_QUEUEUED_SPINLOCKS` kernel configuration option. This configuration option the Linux kernel supports `spinlocks` with queue. This special type of `spinlocks` which instead of acquired and released `atomic` values used `atomic` operation on a `queue` . If the `CONFIG_QUEUEUED_SPINLOCKS` kernel configuration option is enabled, the `arch_spinlock_t` will be represented by the following structure:

```
typedef struct qspinlock {
    atomic_t    val;
} arch_spinlock_t;
```

from the `include/asm-generic/qspinlock_types.h` header file.

We will not stop on this structures for now and before we will consider both `arch_spinlock` and the `qspinlock` , let's look at the operations on a spinlock. The Linux kernel provides following main operations on a `spinlock` :

- `spin_lock_init` - produces initialization of the given `spinlock` ;
- `spin_lock` - acquires given `spinlock` ;
- `spin_lock_bh` - disables software `interrupts` and acquire given `spinlock` .
- `spin_lock_irqsave` and `spin_lock_irq` - disable interrupts on local processor and preserve/not preserve previous interrupt state in the `flags` ;
- `spin_unlock` - releases given `spinlock` ;
- `spin_unlock_bh` - releases given `spinlock` and enables software interrupts;
- `spin_is_locked` - returns the state of the given `spinlock` ;
- and etc.

Let's look on the implementation of the `spin_lock_init` macro. As I already wrote, this and other macro are defined in the `include/linux/spinlock.h` header file and the `spin_lock_init` macro looks:

```
#define spin_lock_init(_lock) \
do { \
    spinlock_check(_lock); \
    raw_spin_lock_init(&(_lock)->rlock); \
} while (0)
```

As we may see, the `spin_lock_init` macro takes a `spinlock` and executes two operations: check the given `spinlock` and execute the `raw_spin_lock_init`. The implementation of the `spinlock_check` is pretty easy, this function just returns the `raw_spinlock_t` of the given `spinlock` to be sure that we got exactly `normal` raw spinlock:

```
static __always_inline raw_spinlock_t *spinlock_check(spinlock_t *lock)
{
    return &lock->rlock;
}
```

The `raw_spin_lock_init` macro:

```
# define raw_spin_lock_init(lock) \
do { \
    *(lock) = __RAW_SPIN_LOCK_UNLOCKED(lock); \
} while (0)
```

assigns the value of the `__RAW_SPIN_LOCK_UNLOCKED` with the given `spinlock` to the given `raw_spinlock_t`. As we may understand from the name of the `__RAW_SPIN_LOCK_UNLOCKED` macro, this macro does initialization of the given `spinlock` and set it to `released` state. This macro defined in the `include/linux/spinlock_types.h` header file and expands to the following macros:

```
#define __RAW_SPIN_LOCK_UNLOCKED(lockname) \
    (raw_spinlock_t) __RAW_SPIN_LOCK_INITIALIZER(lockname)

#define __RAW_SPIN_LOCK_INITIALIZER(lockname) \
{ \
    .raw_lock = __ARCH_SPIN_LOCK_UNLOCKED, \
    SPIN_DEBUG_INIT(lockname) \
    SPIN_DEP_MAP_INIT(lockname) \
}
```

As I already wrote above, we will not consider stuff which is related to debugging of synchronization primitives. In this case we will not consider the `SPIN_DEBUG_INIT` and the `SPIN_DEP_MAP_INIT` macros. So the `__RAW_SPINLOCK_UNLOCKED` macro will be expanded to the:

```
*(&(_lock)->rlock) = __ARCH_SPIN_LOCK_UNLOCKED;
```

where the `__ARCH_SPIN_LOCK_UNLOCKED` is:

```
#define __ARCH_SPIN_LOCK_UNLOCKED    { { 0 } }
```

and:

```
#define __ARCH_SPIN_LOCK_UNLOCKED    { ATOMIC_INIT(0) }
```

for the `x86_64` architecture. if the `CONFIG_QUEUED_SPINLOCKS` kernel configuration option is enabled. So, after the expansion of the `spin_lock_init` macro, a given `spinlock` will be initialized and its state will be - `unlocked` .

From this moment we know how to initialize a `spinlock` , now let's consider [API](#) which Linux kernel provides for manipulations of `spinlocks` . The first is:

```
static __always_inline void spin_lock(spinlock_t *lock)
{
    raw_spin_lock(&lock->rlock);
}
```

function which allows us to `acquire` a `spinlock`. The `raw_spin_lock` macro is defined in the same header file and expands to the call of the `_raw_spin_lock` function:

```
#define raw_spin_lock(lock)    _raw_spin_lock(lock)
```

As we may see in the [include/linux/spinlock.h](#) header file, definition of the `_raw_spin_lock` macro depends on the `CONFIG_SMP` kernel configuration parameter:

```
#if defined(CONFIG_SMP) || defined(CONFIG_DEBUG_SPINLOCK)
# include <linux/spinlock_api_smp.h>
#else
# include <linux/spinlock_api_up.h>
#endif
```

So, if the [SMP](#) is enabled in the Linux kernel, the `_raw_spin_lock` macro is defined in the [arch/x86/include/asm/spinlock.h](#) header file and looks like:



```
#define __raw_spin_lock(lock) __raw_spin_lock(lock)
```

The `__raw_spin_lock` function looks:

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
    LOCK_CONTENTED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}
```

As you may see, first of all we disable [preemption](#) by the call of the `preempt_disable` macro from the [include/linux/preempt.h](#) (more about this you may read in the ninth [part](#) of the Linux kernel initialization process chapter). When we will unlock the given `spinlock`, preemption will be enabled again:

```
static inline void __raw_spin_unlock(raw_spinlock_t *lock)
{
    ...
    ...
    ...
    preempt_enable();
}
```

We need to do this while a process is spinning on a lock, other processes must be prevented to preempt the process which acquired a lock. The `spin_acquire` macro which through a chain of other macros expands to the call of the:

```
#define spin_acquire(l, s, t, i)          lock_acquire_exclusive(l, s, t, NULL, i)
#define lock_acquire_exclusive(l, s, t, n, i)  lock_acquire(l, s, t, 0, 1, n, i)
```

`lock_acquire` function:

```

void lock_acquire(struct lockdep_map *lock, unsigned int subclass,
                 int trylock, int read, int check,
                 struct lockdep_map *nest_lock, unsigned long ip)
{
    unsigned long flags;

    if (unlikely(current->lockdep_recursion))
        return;

    raw_local_irq_save(flags);
    check_flags(flags);

    current->lockdep_recursion = 1;
    trace_lock_acquire(lock, subclass, trylock, read, check, nest_lock, ip);
    __lock_acquire(lock, subclass, trylock, read, check,
                  irqs_disabled_flags(flags), nest_lock, ip, 0, 0);
    current->lockdep_recursion = 0;
    raw_local_irq_restore(flags);
}

```

As I wrote above, we will not consider stuff here which is related to debuggin or tracing. The main point of the `lock_acquire` function is to disable hardware interrupts by the call of the `raw_local_irq_save` macro, because the given spinlock might be acquired with enabled hardware interrupts. In this way the process will not be preempted. Note that in the end of the `lock_acquire` function we will enable hardware interrupts again with the help of the `raw_local_irq_restore` macro. As you already may guess, the main work will be in the `__lock_acquire` function which is defined in the [kernel/locking/lockdep.c](#) source code file.

The `__lock_acquire` function looks big. We will try to understand what does this function do, but not in this part. Actually this function mostly related to the Linux kernel [lock validator](#) and it is not topic of this part. If we will return to the definition of the `__raw_spin_lock` function, we will see that it contains the following definition in the end:

```
LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
```

The `LOCK_CONTENDED` macro is defined in the [include/linux/lockdep.h](#) header file and just calls the given function with the given `spinlock` :

```

#define LOCK_CONTENDED(_lock, try, lock) \
    lock(_lock)

```

In our case, the `lock` is `do_raw_spin_lock` function from the [include/linux/spinlock.h](#) header file and the `_lock` is the given `raw_spinlock_t` :

```
static inline void do_raw_spin_lock(raw_spinlock_t *lock) __acquires(lock)
{
    __acquire(lock);
    arch_spin_lock(&lock->raw_lock);
}
```

The `__acquire` here is just `sparse` related macro and we are not interesting in it in this moment. Location of the definition of the `arch_spin_lock` function depends on two things: the first is architecture of system and the second do we use `queued spinlocks` or not. In our case we consider only `x86_64` architecture, so the definition of the `arch_spin_lock` is represented as the macro from the `include/asm-generic/qspinlock.h` header file:

```
#define arch_spin_lock(l)          queued_spin_lock(l)
```

if we are using `queued spinlocks`. Or in other case, the `arch_spin_lock` function is defined in the `arch/x86/include/asm/spinlock.h` header file. Now we will consider only `normal spinlock` and information related to `queued spinlocks` we will see later. Let's look again on the definition of the `arch_spinlock` structure, to understand implementation of the `arch_spin_lock` function:

```
typedef struct arch_spinlock {
    union {
        __ticketpair_t head_tail;
        struct __raw_tickets {
            __ticket_t head, tail;
        } tickets;
    };
} arch_spinlock_t;
```

This variant of `spinlock` is called - `ticket spinlock`. As we may see, it consists from two parts. When lock is acquired, it increments a `tail` by one everytime when a process wants to hold a `spinlock`. If the `tail` is not equal to `head`, the process will be locked, until values of these variables will not be equal. Let's look on the implementation of the `arch_spin_lock` function:

```

static __always_inline void arch_spin_lock(arch_spinlock_t *lock)
{
    register struct __raw_tickets inc = { .tail = TICKET_LOCK_INC };

    inc = xadd(&lock->tickets, inc);

    if (likely(inc.head == inc.tail))
        goto out;

    for (;;) {
        unsigned count = SPIN_THRESHOLD;

        do {
            inc.head = READ_ONCE(lock->tickets.head);
            if (__tickets_equal(inc.head, inc.tail))
                goto clear_slowpath;
            cpu_relax();
        } while (--count);
        __ticket_lock_spinning(lock, inc.tail);
    }
clear_slowpath:
    __ticket_check_and_clear_slowpath(lock, inc.head);
out:
    barrier();
}

```

At the beginning of the `arch_spin_lock` function we can initialization of the `__raw_tickets` structure with `tail - 1` :

```

#define __TICKET_LOCK_INC    1

```

In the next line we execute `xadd` operation on the `inc` and `lock->tickets` . After this operation the `inc` will store value of the `tickets` of the given `lock` and the `tickets.tail` will be increased on `inc` or `1` . The `tail` value was increased on `1` which means that one process started to try to hold a lock. In the next step we do the check that checks that `head` and `tail` have the same value. If these values are equal, this means that nobody holds lock and we go to the `out` label. In the end of the `arch_spin_lock` function we may see the `barrier` macro which represents `barrier` instruction which guarantees that compiler will not change order of operations that access memory (more about memory barriers you can read in the kernel [documentation](#)).

If one process held a lock and a second process started to execute the `arch_spin_lock` function, the `head` will not be equal to `tail` , because the `tail` will be greater than `head` on `1` . In this way, process will occur in the loop. There will be comparison between `head` and the `tail` values at each loop iteration. If these values are not equal, the `cpu_relax` will be called which is just `NOP` instruction:

```
#define cpu_relax()      asm volatile("rep; nop")
```

and the next iteration of the loop will be started. If these values will be equal, this means that the process which held this lock, released this lock and the next process may acquire the lock.

The `spin_unlock` operation goes through the all macros/function as `spin_lock`, ofcourse with `unlock` prefix. In the end the `arch_spin_unlock` function will be called. If we will look at the implementation of the `arch_spin_lock` function, we will see that it increases `head` of the lock tickets list:

```
__add(&lock->tickets.head, TICKET_LOCK_INC, UNLOCK_LOCK_PREFIX);
```

In a combination of the `spin_lock` and `spin_unlock`, we get kind of queue where `head` contains an index number which maps currently executed process which holds a lock and the `tail` which contains an index number which maps last process which tried to hold the lock:

```

+-----+   +-----+
|       |   |       |
head |  7  | - - - |  7  | tail
|       |   |       |
+-----+   +-----+
                |
                +-----+
                |       |
                |  8  |
                |       |
                +-----+
                |
                +-----+
                |       |
                |  9  |
                |       |
                +-----+

```

That's all for now. We didn't cover `spinlock` API in full in this part, but I think that the main idea behind this concept must be clear now.

## Conclusion

This concludes the first part covering synchronization primitives in the Linux kernel. In this part, we met first synchronization primitive `spinlock` provided by the Linux kernel. In the next part we will continue to dive into this interesting theme and will see other `synchronization` related stuff.

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [Concurrent computing](#)
- [Synchronization](#)
- [Clocksource framework](#)
- [Mutex](#)
- [Race condition](#)
- [Atomic operations](#)
- [SMP](#)
- [x86\\_64](#)
- [Interrupts](#)
- [Preemption](#)
- [Linux kernel lock validator](#)
- [Sparse](#)
- [xadd instruction](#)
- [NOP](#)
- [Memory barriers](#)
- [Previous chapter](#)

# 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 complicated 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 it's 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:

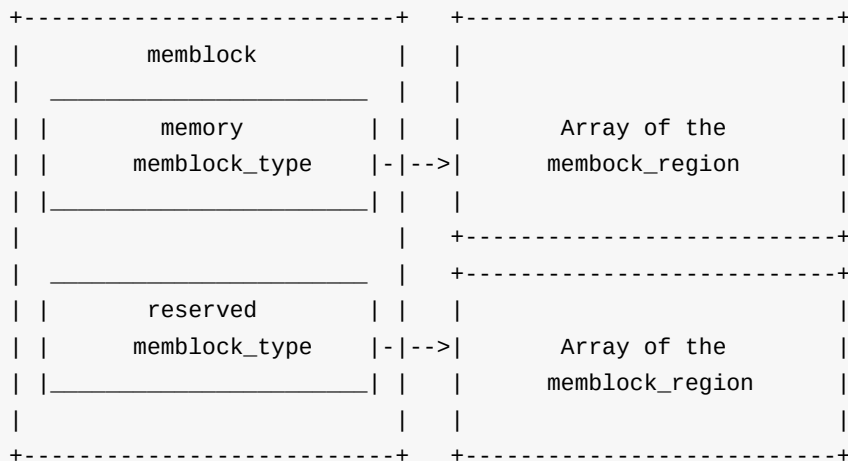
```
struct memblock_region {
    phys_addr_t base;
    phys_addr_t size;
    unsigned long flags;
#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
    int nid;
#endif
};
```

`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:



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` are 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 the `__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] __initd
static struct memblock_region memblock_reserved_init_regions[INIT_MEMBLOCK_REGIONS] __ini
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
static struct memblock_region memblock_physmem_init_regions[INIT_PHYSMEM_REGIONS] __initd
#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` structure initialization (read above if you've forgotten).

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 the initialization of the `memblock` structure has been finished and we can look on the Memblock API.

## Memblock API

Ok we have finished with initialization 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` is presented in the [mm/memblock.c](#). To understand how `memblock` works and how it is implemented, let's look at its usage first. 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 is 1 if `CONFIG_NODES_SHIFT` is not set in the configuration file or `1 << 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 `memblock` 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, etc. and increases 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;
```

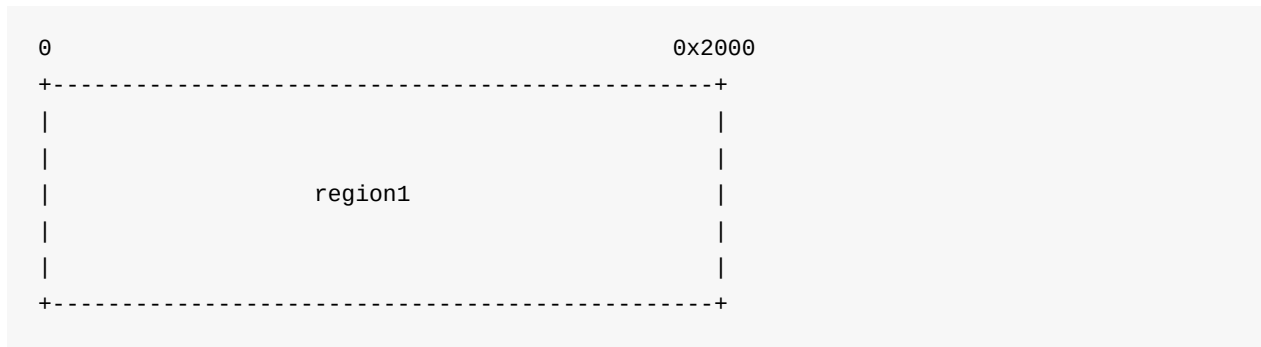
As we update the size of the first memory region with the size of the next memory region, we move all memory regions which are after the (`next`) memory region one index backward with the `memmove` function:

```
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:



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 APIs 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 `memblock`. 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/phymem`

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-insides](#).**

## 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](#), etc. In virtual memory fix-mapped area is placed in the modules area:

kernel text	kernel text	Modules	vsyscalls
mapping	data		fix-mapped
from phys 0			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 kilobytes. In your case it might be a different number, because the size depends on amount of the fix-mapped addresses which depends on your kernel's configuration.

The second `FIXADDR_START` macro just subtracts fix-mapped area size from the last address of the fix-mapped area to get its base virtual address. `FIXADDR_TOP` is a 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 by 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 of `fix-mapped` area (`FIXADDR_TOP`) and shifts the result right 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 shifting 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_TB00T_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 use `fix-mapped` area in the early `ioremap` initialization. Let's look on it and try to understand what is `ioremap`, how it is implemented in the kernel and how it is related 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 defined as:

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

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 is 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, etc. Every `resource` structure contains pointers to the `parent`, `sibling` and `child` resources. As it has parent and child, it means that every subset of resources 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 is 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 is 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 functions for checking, requesting and release of a memory region as 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-deffeffff : 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 is 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 will be inserted to the `iomem` resource 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 them 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 into 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_BITMAP_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 its work, we're getting page middle directory where early ioremap begins 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 fill `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_populate_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 virtual addresses. Technically 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 physical address of the `I/O` memory region to map on virtual addresses;
- `size` - size of the `I/O` memory 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` except the first 12 bits. `PAGE_MASK` macro is 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 the 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` area 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 is 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 is not informed of the changes, that's why we need to do it manually. There are two ways 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 to invalidate 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 finished and we can back to the `__early_ioremap` implementation. As we have set fixmap area for the given address, we 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 physical 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 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-insides](#).**

## 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](#)
- [The initcall mechanism](#)

# 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. In this part, we take a closer look at this feature and try to understand how it is implemented and how it works.

The kernel provides an API for creating per-cpu variables - the `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 variables, for example like this:

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

We pass the type and the name of our variable. `DEFINE_PER_CPU` calls the `DEFINE_PER_CPU_SECTION` macro and passes the same two parameters 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 a global per-cpu variable:

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

It means that we will have a `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 the `DEFINE_PER_CPU` macro, a per-cpu variable in the `.data..percpu` section will be created. When the kernel initializes it calls the `setup_per_cpu_areas` function which loads the `.data..percpu` section multiple times, one section per CPU.

Let's look at the per-CPU areas initialization process. It starts in the [init/main.c](#) from the call of the `setup_per_cpu_areas` function which is 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 the `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 the `percpu` first chunk allocator. All percpu areas are allocated in chunks. The first chunk is used for the static percpu variables. The Linux kernel has `percpu_alloc` command line parameters which provides the 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 the handler of this command line option:

```
early_param("percpu_alloc", percpu_alloc_setup);
```

Where the `percpu_alloc_setup` function sets the `pcpu_chosen_fc` variable depends on the `percpu_alloc` parameter value. By default the first chunk allocator is `auto` :

```
enum pcpu_fc pcpu_chosen_fc __initdata = PCPU_FC_AUTO;
```

If the `percpu_alloc` parameter is not given to the kernel command line, the `embed` allocator will be used which embeds the first percpu chunk into bootmem with the `memblock`. The last allocator is the first chunk `page` allocator which maps the 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 the `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 `pcpu_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 bytes;
- `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 these parameters we calculate before the call of the `pcpu_embed_first_chunk` :

```

const size_t dyn_size = PERCPU_MODULE_RESERVE + PERCPU_DYNAMIC_RESERVE - PERCPU_FIRST_CHUNK;
size_t atom_size;
#ifdef CONFIG_X86_64
    atom_size = PMD_SIZE;
#else
    atom_size = PAGE_SIZE;
#endif

```

If the 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 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 finishes the initialization process, we will have loaded `N` `.data..percpu` sections, where `N` is the number of CPUs, and the section used by the bootstrap processor will contain an uninitialized variable created with the `DEFINE_PER_CPU` macro.

The kernel provides an API for per-cpu variables manipulating:

- `get_cpu_var(var)`
- `put_cpu_var(var)`

Let's look at the `get_cpu_var` implementation:

```

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

```

The Linux kernel is preemptible and accessing a per-cpu variable requires us to know which processor the 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 a call of the `preempt_disable` function. After this we can see a call of the `this_cpu_ptr` macro, which looks like:

```

#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 we've created a per-cpu variable and made modifications to it, we must call the `put_cpu_var` macro which enables preemption with a call of `preempt_enable` function. So the typical usage of a per-cpu variable is as follows:

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

Let's look at the `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 a 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 the 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 second we pass the cpu number to the `per_cpu_offset` macro:

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

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

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

where `NR_CPUS` is the number of CPUs. The `__per_cpu_offset` array is filled with the distances between cpu-variable copies. For example all per-cpu data is `x` bytes in size, so if we access `__per_cpu_offset[Y]`, `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 return a pointer to the variable.

That's all! Of course it is not the full API, but a general overview. It can be hard to start with, but to understand per-cpu variables you mainly need to understand the [include/linux/percpu-defs.h](#) magic.

Let's again look at the algorithm of getting a pointer to a per-cpu variable:

- The kernel creates multiple `.data..percpu` sections (one per-cpu) during initialization process;
- All variables created with the `DEFINE_PER_CPU` macro will be relocated to the first section or for CPU0;
- `__per_cpu_offset` array filled with the distance ( `BOOT_PERCPU_OFFSET` ) between `.data..percpu` sections;
- When the `per_cpu_ptr` is called, for example for getting a pointer on a certain per-cpu variable for the third CPU, the `__per_cpu_offset` array will be accessed, where every index points to the required 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 a 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`. The implementations of all of these functions are very similar. Every function checks the second parameter. If it is `true`, it calls `cpumask_set_cpu` otherwise it calls `cpumask_clear_cpu`.

There are two ways for a `cpumask` creation. First is to use `cpumask_t`. It is defined as:

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

It wraps the `cpumask` structure which contains one bitmak `bits` field. The `DECLARE_BITMAP` macro gets two parameters:

- bitmap name;
- number of bits.



and creates an array of `unsigned long` with the given name. Its implementation is pretty easy:

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

where `BITS_TO_LONGS` :

```
#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 are focussing on the `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 represents the number of CPUs in the system and depends on the `CONFIG_NR_CPUS` macro which is defined in [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 the `DECLARE_BITMAP` macro directly and the `to_cpumask` macro which converts the given bitmap to `struct cpumask *` :

```
#define to_cpumask(bitmap) \
    ((struct cpumask *) (1 ? (bitmap) \
    : (void *) sizeof(__check_is_bitmap(bitmap))))
```

We can see the ternary operator here which is `true` every time.

`__check_is_bitmap` inline function is defined as:

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

And returns `1` every time. We need it here for only one purpose: at compile time it checks that a given `bitmap` is a bitmap, or in other words it checks that a given `bitmap` has type - `unsigned long *`. So we just pass `cpu_possible_bits` to the `to_cpumask` macro for converting an 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 a bitmap and defined as:

```
static DECLARE_BITMAP(cpu_online_bits, CONFIG_NR_CPUS) __read_mostly;
```

The `cpumask_set_cpu` function makes only one call to the `set_bit` function:

```
static inline void cpumask_set_cpu(unsigned int cpu, struct cpumask *dstp)
{
    set_bit(cpumask_check(cpu), cpumask_bits(dstp));
}
```

The `set_bit` function takes two parameters 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 parameters will be passed to the

- `cpumask_check`;
- `cpumask_bits`.

Let's consider these two macros. First if `cpumask_check` does nothing in our case and just returns given parameter. The second `cpumask_bits` just returns the `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 scary, 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 calls 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, the `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 the cpu to occupy the system bus while the instruction(s) will be executed. This allows the CPU to synchronize memory access, preventing simultaneous access of multiple processors (or devices - the DMA controller for example) to one memory cell.

`BITOP_ADDR` casts the given parameter to the `(*(volatile long *))` and adds `+m` constraints. `+` means that this operand is both read and written by the instruction. `m` shows that this is a 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` - immediate register operand.

The `bts` instruction sets a given bit in a bit string and stores the value of a given bit in the `CF` flag. So we passed the `cpu` number which is zero in our case and after `set_bit` is executed, it sets the zero bit in the `cpu_online_bits` `cpumask`. It means that the first `cpu` is online at this moment.

Besides the `set_cpu_*` API, `cpumask` of course provides another API for `cpumasks` manipulation. Let's consider it in short.

## Additional cpumask API

`cpumask` provides a set of macros for getting the numbers of CPUs in various states. For example:

```
#define num_online_cpus() cpumask_weight(cpu_online_mask)
```

This macro returns the amount of `online` CPUs. It calls the `cpumask_weight` function with the `cpu_online_mask` bitmap (read about it). The `cpumask_weight` function makes one call of the `bitmap_weight` 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 the number of 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 the Linux kernel provides the following API for the manipulation 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](#)

# The initcall mechanism

## Introduction

As you may understand from the title, this part will cover interesting and important concept in the Linux kernel which is called - `initcall` . We already saw definitions like these:

```
early_param("debug", debug_kernel);
```

or

```
arch_initcall(init_pit_clocksource);
```

in some parts of the Linux kernel. Before we see how this mechanism is implemented in the Linux kernel, we must know actually what is it and how the Linux kernel uses it. Definitions like these represent a [callback](#) function which will be called during initialization of the Linux kernel or right after it. Actually the main point of the `initcall` mechanism is to determine correct order of the built-in modules and subsystems initialization. For example let's look at the following function:

```
static int __init nmi_warning_debugfs(void)
{
    debugfs_create_u64("nmi_longest_ns", 0644,
                      arch_debugfs_dir, &nmi_longest_ns);
    return 0;
}
```

from the [arch/x86/kernel/nmi.c](#) source code file. As we may see it just creates the `nmi_longest_ns` [debugfs](#) file in the `arch_debugfs_dir` directory. Actually, this `debugfs` file may be created only after the `arch_debugfs_dir` will be created. Creation of this directory occurs during the architecture-specific initialization of the Linux kernel. Actually this directory will be created in the `arch_kdebugfs_init` function from the [arch/x86/kernel/kdebugfs.c](#) source code file. Note that the `arch_kdebugfs_init` function is marked as `initcall` too:

```
arch_initcall(arch_kdebugfs_init);
```

The Linux kernel calls all architecture-specific `initcalls` before the `fs` related `initcalls`. So, our `nmi_longest_ns` file will be created only after the `arch_kdebugfs_dir` directory will be created. Actually, the Linux kernel provides eight levels of main `initcalls`:

- `early` ;
- `core` ;
- `postcore` ;
- `arch` ;
- `susys` ;
- `fs` ;
- `device` ;
- `late` .

All of their names are represented by the `initcall_level_names` array which is defined in the [init/main.c](#) source code file:

```
static char *initcall_level_names[] __initdata = {
    "early",
    "core",
    "postcore",
    "arch",
    "subsys",
    "fs",
    "device",
    "late",
};
```

All functions which are marked as `initcall` by these identifiers, will be called in the same order or at first `early initcalls` will be called, at second `core initcalls` and etc. From this moment we know a little about `initcall` mechanism, so we can start to dive into the source code of the Linux kernel to see how this mechanism is implemented.

## Implementation initcall mechanism in the Linux kernel

The Linux kernel provides a set of macros from the [include/linux/init.h](#) header file to mark a given function as `initcall`. All of these macros are pretty simple:

```

#define early_initcall(fn)      __define_initcall(fn, early)
#define core_initcall(fn)      __define_initcall(fn, 1)
#define postcore_initcall(fn)  __define_initcall(fn, 2)
#define arch_initcall(fn)      __define_initcall(fn, 3)
#define subsys_initcall(fn)    __define_initcall(fn, 4)
#define fs_initcall(fn)        __define_initcall(fn, 5)
#define device_initcall(fn)    __define_initcall(fn, 6)
#define late_initcall(fn)      __define_initcall(fn, 7)

```

and as we may see these macros just expands to the call of the `__define_initcall` macro from the same header file. As we may see, the `__define_initcall` macro takes two arguments:

- `fn` - callback function which will be called during call of `initcalls` of the certain level;
- `id` - identifier to identify `initcall` to prevent error when two the same `initcalls` point to the same handler.

The implementation of the `__define_initcall` macro looks like:

```

#define __define_initcall(fn, id) \
    static initcall_t __initcall_##fn##id __used \
    __attribute__((__section__(".initcall" #id ".init"))) = fn; \
    LTO_REFERENCE_INITCALL(__initcall_##fn##id)

```

To understand the `__define_initcall` macro, first of all let's look at the `initcall_t` type. This type is defined in the same [header](#) file and represents pointer to a function which returns pointer to [integer](#) which will be result of the `initcall`:

```

typedef int (*initcall_t)(void);

```

Now let's return to the `__define_initcall` macro. The `##` provides ability to concatenate two symbols. In our case, the first line of the `__define_initcall` macro produces definition of the given function which is located in the `.initcall id .init` [ELF section](#) and marked with the following [gcc](#) attributes: `__initcall_function_name_id` and `__used`. If we will look in the [include/asm-generic/vmlinux.lds.h](#) header file which represents data for the kernel [linker](#) script, we will see that all of `initcalls` sections will be placed in the `.data` section:



```

#define INIT_CALLS \
    VMLINUX_SYMBOL(__initcall_start) = .; \
    *(.initcallearly.init) \
    INIT_CALLS_LEVEL(0) \
    INIT_CALLS_LEVEL(1) \
    INIT_CALLS_LEVEL(2) \
    INIT_CALLS_LEVEL(3) \
    INIT_CALLS_LEVEL(4) \
    INIT_CALLS_LEVEL(5) \
    INIT_CALLS_LEVEL(rootfs) \
    INIT_CALLS_LEVEL(6) \
    INIT_CALLS_LEVEL(7) \
    VMLINUX_SYMBOL(__initcall_end) = .;

#define INIT_DATA_SECTION(initsetup_align) \
    .init.data : AT(ADDR(.init.data) - LOAD_OFFSET) { \
        ... \
        INIT_CALLS \
        ... \
    }

```

The second attribute - `__used` is defined in the [include/linux/compiler-gcc.h](#) header file and just expands to the definition of the following `gcc` attribute:

```

#define __used __attribute__((__used__))

```

which prevents `variable defined but not used` warning. The last line of the `__define_initcall` macro is:

```

LTO_REFERENCE_INITCALL(__initcall_##fn##id)

```

depends on the `CONFIG_LTO` kernel configuration option and just provides stub for the compiler [Link time optimization](#):

```

#ifdef CONFIG_LTO
#define LTO_REFERENCE_INITCALL(x) \
    static __used __exit void *reference_##x(void) \
    { \
        return &x; \
    }
#else
#define LTO_REFERENCE_INITCALL(x)
#endif

```

to prevent problem when there is no reference to a variable in a module it will be moved to the end of the program. That's all about the `__define_initcall` macro. So, all of the `*_initcall` macros will be expanded during compilation of the Linux kernel, and all `initcalls` will be placed in their sections and all of them will be available from the `.data` section and the Linux kernel will know where to find a certain `initcall` to call it during initialization process.

As `initcalls` can be called by the Linux kernel, let's look how the Linux kernel does this. This process starts in the `do_basic_setup` function from the [init/main.c](#) source code file:

```
static void __init do_basic_setup(void)
{
    ...
    ...
    ...
    do_initcalls();
    ...
    ...
    ...
}
```

which is called during the initialization of the Linux kernel, right after main steps of initialization like memory manager related initializations, `CPU` subsystem and other already finished. The `do_initcalls` function just goes through the array of `initcall` levels and call the `do_initcall_level` function for each level:

```
static void __init do_initcalls(void)
{
    int level;

    for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++)
        do_initcall_level(level);
}
```

The `initcall_levels` array is defined in the same source code [file](#) and contains pointers to the sections which were defined in the `__define_initcall` macro:

```
static initcall_t *initcall_levels[] __initdata = {
    __initcall0_start,
    __initcall1_start,
    __initcall2_start,
    __initcall3_start,
    __initcall4_start,
    __initcall5_start,
    __initcall6_start,
    __initcall7_start,
    __initcall_end,
};
```

If you are interested, you can find these sections in the `arch/x86/kernel/vmlinux.lds` linker script which is generated after the Linux kernel compilation:

```
.init.data : AT(ADDR(.init.data) - 0xffffffff80000000) {
    ...
    ...
    ...
    ...
    __initcall_start = .;
    *(.initcallearly.init)
    __initcall0_start = .;
    *(.initcall0.init)
    *(.initcall0s.init)
    __initcall1_start = .;
    ...
    ...
}
```

If this is not familiar for you, you can know more about [linkers](#) in the special [part](#) of this book.

As we just saw, the `do_initcall_level` function takes one parameter - level of `initcall` and does two following things: First of all this function parses the `initcall_command_line` which is copy of usual kernel [command line](#) which may contain parameters for modules with the `parse_args` function from the [kernel/params.c](#) source code file and call the `do_on_initcall` function for each level:

```
for (fn = initcall_levels[level]; fn < initcall_levels[level+1]; fn++)
    do_one_initcall(*fn);
```

The `do_on_initcall` does all main job for us. As we may see, this function takes one parameter which represent `initcall` callback function and does the call of the given callback:

```

int __init_or_module do_one_initcall(initcall_t fn)
{
    int count = preempt_count();
    int ret;
    char msgbuf[64];

    if (initcall_blacklisted(fn))
        return -EPERM;

    if (initcall_debug)
        ret = do_one_initcall_debug(fn);
    else
        ret = fn();

    msgbuf[0] = 0;

    if (preempt_count() != count) {
        sprintf(msgbuf, "preemption imbalance ");
        preempt_count_set(count);
    }
    if (irqs_disabled()) {
        strlcat(msgbuf, "disabled interrupts ", sizeof(msgbuf));
        local_irq_enable();
    }
    WARN(msgbuf[0], "initcall %pF returned with %s\n", fn, msgbuf);

    return ret;
}

```

Let's try to understand what does the `do_on_initcall` function does. First of all we increase `preemption` counter to check it later to be sure that it is not imbalanced. After this step we can see the call of the `initcall_blacklist` function which goes over the `blacklisted_initcalls` list which stores blacklisted `initcalls` and releases the given `initcall` if it is located in this list:

```

list_for_each_entry(entry, &blacklisted_initcalls, next) {
    if (!strcmp(fn_name, entry->buf)) {
        pr_debug("initcall %s blacklisted\n", fn_name);
        kfree(fn_name);
        return true;
    }
}

```

The blacklisted `initcalls` stored in the `blacklisted_initcalls` list and this list is filled during early Linux kernel initialization from the Linux kernel command line.

After the blacklisted `initcalls` will be handled, the next part of code does directly the call of the `initcall`:

```
if (initcall_debug)
    ret = do_one_initcall_debug(fn);
else
    ret = fn();
```

Depends on the value of the `initcall_debug` variable, the `do_one_initcall_debug` function will call `initcall` or this function will do it directly via `fn()`. The `initcall_debug` variable is defined in the [same](#) source code file:

```
bool initcall_debug;
```

and provides ability to print some information to the kernel [log buffer](#). The value of the variable can be set from the kernel commands via the `initcall_debug` parameter. As we can read from the [documentation](#) of the Linux kernel command line:

```
initcall_debug    [KNL] Trace initcalls as they are executed. Useful
                  for working out where the kernel is dying during
                  startup.
```

And that's true. If we will look at the implementation of the `do_one_initcall_debug` function, we will see that it does the same as the `do_one_initcall` function or i.e. the `do_one_initcall_debug` function calls the given `initcall` and prints some information (like the [pid](#) of the currently running task, duration of execution of the `initcall` and etc.) related to the execution of the given `initcall`:

```
static int __init_or_module do_one_initcall_debug(initcall_t fn)
{
    ktime_t calltime, delta, rettime;
    unsigned long long duration;
    int ret;

    printk(KERN_DEBUG "calling %pF @ %i\n", fn, task_pid_nr(current));
    calltime = ktime_get();
    ret = fn();
    rettime = ktime_get();
    delta = ktime_sub(rettime, calltime);
    duration = (unsigned long long) ktime_to_ns(delta) >> 10;
    printk(KERN_DEBUG "initcall %pF returned %d after %lld usecs\n",
           fn, ret, duration);

    return ret;
}
```

As an `initcall` was called by the one of the `do_one_initcall` or `do_one_initcall_debug` functions, we may see two checks in the end of the `do_one_initcall` function. The first one checks the amount of possible `__preempt_count_add` and `__preempt_count_sub` calls inside of the executed `initcall`, and if this value is not equal to the previous value of the preemptible counter, we add the `preemption imbalance` string to the message buffer and set correct value of the preemptible counter:

```
if (preempt_count() != count) {
    sprintf(msgbuf, "preemption imbalance ");
    preempt_count_set(count);
}
```

Later this error string will be printed. The last check the state of local `IRQs` and if they are disabled, we add the `disabled interrupts` strings to the our message buffer and enable `IRQs` for the current processor to prevent the state when `IRQs` were disabled by an `initcall` and didn't enabled again:

```
if (irqs_disabled()) {
    strlcat(msgbuf, "disabled interrupts ", sizeof(msgbuf));
    local_irq_enable();
}
```

That's all. In this way the Linux kernel does initialization of many subsystems in a correct order. From now we know what is it `initcall` mechanism in the Linux kernel. We saw main general part of the `initcall` mechanism in this part. But we avoided some important concepts. Let's make a short look at these concepts.

First of all, we have missed one level of `initcalls`, this is `rootfs initcalls`. You can find definition of the `rootfs_initcall` in the [include/linux/init.h](#) header file together with all similar macros which we saw in this part:

```
#define rootfs_initcall(fn)    __define_initcall(fn, rootfs)
```

As we may understand from the macro's name, its main purpose is to store callbacks which are related to the `rootfs`. Besides this goal, it may be useful to initialize other stuffs after initialization related to filesystems level, but only before devices related stuff are not initialized. For example, the decompression of the `initramfs` which occurred in the `populate_rootfs` function from the [init/initramfs.c](#) source code file:

```
rootfs_initcall(populate_rootfs);
```

From this place, we may see familiar output:

```
[ 0.199960] Unpacking initramfs...
```

Besides the `rootfs_initcall` level, there are additional `console_initcall`, `security_initcall` and other secondary `initcall` levels. The last thing that we have missed is the set of the `*_initcall_sync` levels. Almost each `*_initcall` macro that we have seen in this part, has macro companion with the `_sync` prefix:

```
#define core_initcall_sync(fn)      __define_initcall(fn, 1s)
#define postcore_initcall_sync(fn)  __define_initcall(fn, 2s)
#define arch_initcall_sync(fn)      __define_initcall(fn, 3s)
#define subsys_initcall_sync(fn)    __define_initcall(fn, 4s)
#define fs_initcall_sync(fn)        __define_initcall(fn, 5s)
#define device_initcall_sync(fn)    __define_initcall(fn, 6s)
#define late_initcall_sync(fn)      __define_initcall(fn, 7s)
```

The main goal of these additional levels is to wait for completion of all a module related initialization routines for a certain level.

That's all.

## Conclusion

In this part we saw the important mechanism of the Linux kernel which allows to call a function which depends on the current state of the Linux kernel during its initialization.

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

**Please note that English is not my first language and I am really sorry for any inconvenience. If you found any mistakes please send me PR to [linux-insides](#).**

## Links

- [callback](#)
- [debugfs](#)
- [integer type](#)
- [symbols concatenation](#)
- [GCC](#)
- [Link time optimization](#)
- [Introduction to linkers](#)
- [Linux kernel command line](#)

- [Process identifier](#)
- [IRQs](#)
- [rootfs](#)
- [previous part](#)



# Data Structures in the Linux Kernel

Linux kernel provides different implementations of data structures like doubly linked list, B+ tree, priority heap and many many more.

This part considers the following data structures and algorithms:

- [Doubly linked list](#)
- [Radix tree](#)
- [Bit arrays](#)

# Data Structures in the Linux Kernel

## Doubly linked list

Linux kernel provides its own implementation of doubly linked list, 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 in the [include/linux/types.h](#):

```
struct list_head {
    struct list_head *next, *prev;
};
```

You can note that it is different from many implementations of doubly linked list which you have seen. For example, this doubly linked list structure from the [glib](#) library looks like :

```
struct GList {
    gpointer data;
    GList *next;
    GList *prev;
};
```

Usually a linked list structure contains a pointer to the item. The implementation of linked list in Linux kernel does not. So the main question is - `where does the list store the data?` .

The actual implementation of linked list 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 to understand 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 an example in miscellaneous character drivers. Misc character drivers

API from the [drivers/char/misc.c](#) is used for writing small drivers for handling simple hardware or virtual devices. Those drivers share same major number:

```
#define MISC_MAJOR      10
```

but have their 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 have a close look at 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 a list of registered devices. In the beginning of the source code file we can see the definition of `misc_list`:

```
static LIST_HEAD(misc_list);
```

which expands to the definition of 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 sets previous and next entries with the address of variable - name:

```
#define LIST_HEAD_INIT(name) { &(amp;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 as the `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 a device is created by 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 a new entry to the list. Let's look at its 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 the new item will be inserted.
- 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 add a 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 one question: how to get list's entry. There is a special macro:

```
#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 structure;

For example:

```
const struct miscdevice *p = list_entry(v, struct miscdevice, list)
```

After this we can access to 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. At first sight, the `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 of two expressions in curly brackets. The 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 I found 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 along with 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. The `container_of` macro returns the 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 `ptr` to it. Now `ptr` and `__mptr` point to the same address. Technically we don't need this line but it's useful for type checking. The first line ensures that the 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 the only functions which `<linux/list.h>` provides. 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`
- `list_for_each`
- `list_for_each_entry`

and many more.

# Data Structures in the Linux Kernel

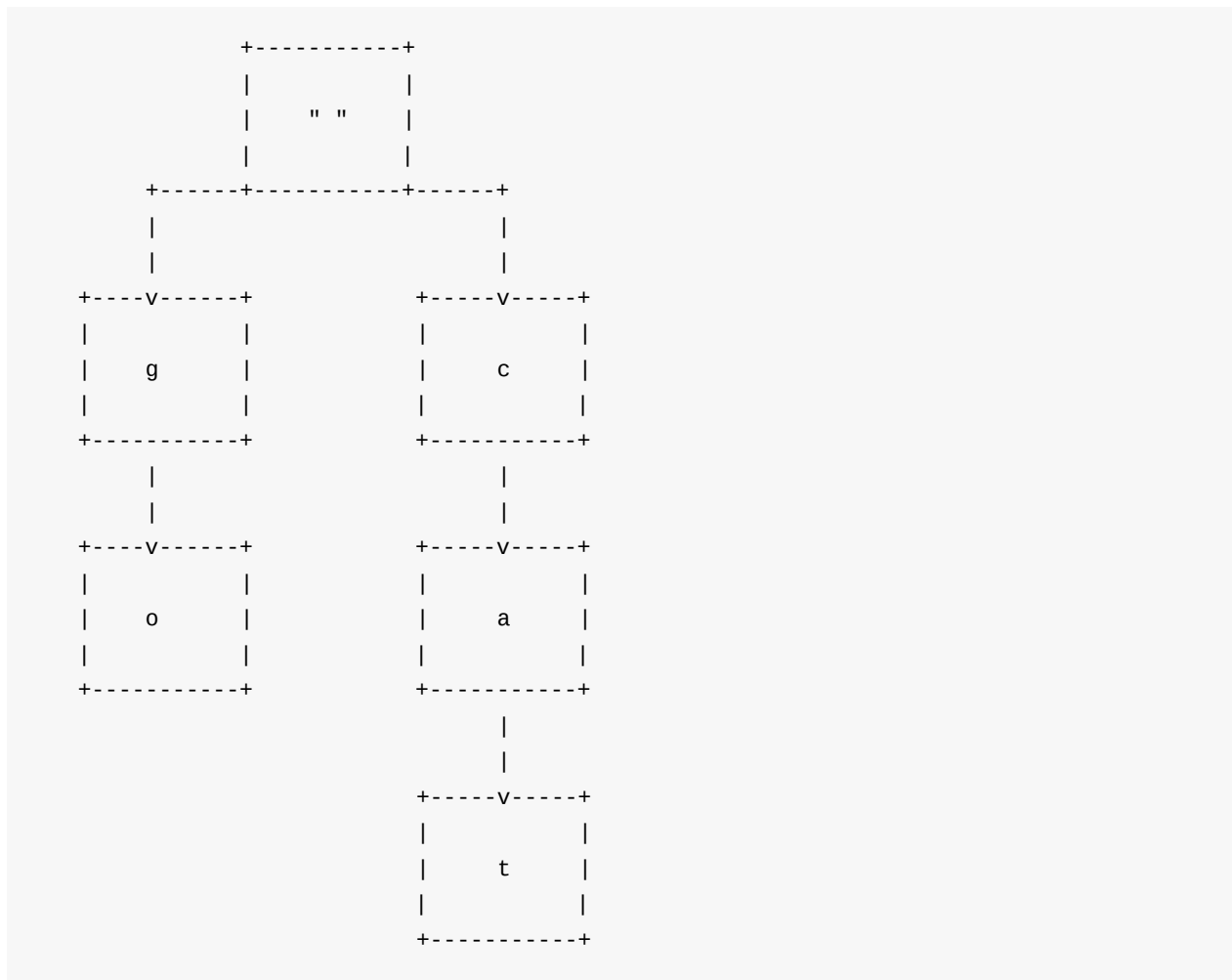
## Radix tree

As you already know linux kernel provides many different libraries and functions which implement different data structures and algorithms. 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 a `radix tree` is. Radix tree is a `compressed trie` where a [trie](#) is a data structure which implements an interface of an associative array and allows to store values as `key-value`. The keys are usually strings, but any data type can be used. A trie is different from an `n-tree` because of 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:





So in this example, we can see the `trie` with keys, `go` and `cat`. The compressed trie or `radix tree` differs from `trie` in that all intermediates nodes which have only one child are removed.

Radix tree in linux kernel is the datastructure which maps values to integer keys. 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 will be performed;
- `rnode` - pointer to the child node.

The first field 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) means sleep and wait for memory, ( `__GFP_HIGHMEM` flag) means high memory can be used, ( `GFP_ATOMIC` flag) means the allocation process has high-priority and can't sleep etc.

- `GFP_NOIO` - can sleep and wait for memory;
- `__GFP_HIGHMEM` - high memory can be used;
- `GFP_ATOMIC` - allocation process is high-priority and can't sleep;

etc.

The next field 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. This 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 trees in the linux kernel also supports tags which are associated with the `tags` fields in the `radix_tree_node` structure. Tags allow individual bits to be set on records which are stored in the radix tree.

Now that we know about radix tree structure, it is time to look on its API.

## Linux kernel radix tree API

We start from the datastructure initialization. There are two ways to initialize a 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 initialization with default values as it does `RADIX_TREE_INIT` macro.

The next are two functions for 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 return the record associated 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 not be 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](#)

# Data Structures in the Linux Kernel

## Bit arrays and bit operations in the Linux kernel

Besides different [linked](#) and [tree](#) based data structures, the Linux kernel provides [API](#) for [bit arrays](#) or `bitmap`. Bit arrays are heavily used in the Linux kernel and following source code files contain common `API` for work with such structures:

- [lib/bitmap.c](#)
- [include/linux/bitmap.h](#)

Besides these two files, there is also architecture-specific header file which provides optimized bit operations for certain architecture. We consider [x86\\_64](#) architecture, so in our case it will be:

- [arch/x86/include/asm/bitops.h](#)

header file. As I just wrote above, the `bitmap` is heavily used in the Linux kernel. For example a `bit array` is used to store set of online/offline processors for systems which support [hot-plug](#) cpu (more about this you can read in the [cpumasks](#) part), a `bit array` stores set of allocated [irqs](#) during initialization of the Linux kernel and etc.

So, the main goal of this part is to see how `bit arrays` are implemented in the Linux kernel. Let's start.

## Declaration of bit array

Before we will look on `API` for bitmaps manipulation, we must know how to declare it in the Linux kernel. There are two common method to declare own bit array. The first simple way to declare a bit array is to array of `unsigned long`. For example:

```
unsigned long my_bitmap[8]
```

The second way is to use the `DECLARE_BITMAP` macro which is defined in the [include/linux/types.h](#) header file:

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

We can see that `DECLARE_BITMAP` macro takes two parameters:

- `name` - name of bitmap;
- `bits` - amount of bits in bitmap;

and just expands to the definition of `unsigned long` array with `BITS_TO_LONGS(bits)` elements, where the `BITS_TO_LONGS` macro converts a given number of bits to number of longs or in other words it calculates how many 8 byte elements in `bits`:

```
#define BITS_PER_BYTE      8
#define DIV_ROUND_UP(n,d) (((n) + (d) - 1) / (d))
#define BITS_TO_LONGS(nr)  DIV_ROUND_UP(nr, BITS_PER_BYTE * sizeof(long))
```

So, for example `DECLARE_BITMAP(my_bitmap, 64)` will produce:

```
>>> (((64) + (64) - 1) / (64))
1
```

and:

```
unsigned long my_bitmap[1];
```

After we are able to declare a bit array, we can start to use it.

## Architecture-specific bit operations

We already saw above a couple of source code and header files which provide [API](#) for manipulation of bit arrays. The most important and widely used API of bit arrays is architecture-specific and located as we already know in the [arch/x86/include/asm/bitops.h](#) header file.

First of all let's look at the two most important functions:

- `set_bit` ;
- `clear_bit` .

I think that there is no need to explain what these functions do. This is already clear from their name. Let's look on their implementation. If you will look into the [arch/x86/include/asm/bitops.h](#) header file, you will note that each of these functions represented by two variants: `atomic` and not. Before we will start to dive into implementations of these functions, first of all we must to know a little about `atomic` operations.

In simple words atomic operations guarantees that two or more operations will not be performed on the same data concurrently. The `x86` architecture provides a set of atomic instructions, for example `xchg` instruction, `cmpxchg` instruction and etc. Besides atomic instructions, some of non-atomic instructions can be made atomic with the help of the `lock` instruction. It is enough to know about atomic operations for now, so we can begin to consider implementation of `set_bit` and `clear_bit` functions.

First of all, let's start to consider `non-atomic` variants of this function. Names of non-atomic `set_bit` and `clear_bit` starts from double underscore. As we already know, all of these functions are defined in the [arch/x86/include/asm/bitops.h](#) header file and the first function is `__set_bit`:

```
static inline void __set_bit(long nr, volatile unsigned long *addr)
{
    asm volatile("bts %1,%0" : ADDR : "Ir" (nr) : "memory");
}
```

As we can see it takes two arguments:

- `nr` - number of bit in a bit array.
- `addr` - address of a bit array where we need to set bit.

Note that the `addr` parameter is defined with `volatile` keyword which tells to compiler that value maybe changed by the given address. The implementation of the `__set_bit` is pretty easy. As we can see, it just contains one line of `inline assembler` code. In our case we are using the `bts` instruction which selects a bit which is specified with the first operand (`nr` in our case) from the bit array, stores the value of the selected bit in the `CF` flags register and set this bit.

Note that we can see usage of the `nr`, but there is `addr` here. You already might guess that the secret is in `ADDR`. The `ADDR` is the macro which is defined in the same header code file and expands to the string which contains value of the given address and `+m` constraint:

```
#define ADDR          BITOP_ADDR(addr)
#define BITOP_ADDR(x) "+m" (*(volatile long *) (x))
```

Besides the `+m`, we can see other constraints in the `__set_bit` function. Let's look on they and try to understand what do they mean:

- `+m` - represents memory operand where `+` tells that the given operand will be input and output operand;
- `I` - represents integer constant;
- `r` - represents register operand

Besides these constraint, we also can see - the `memory` keyword which tells compiler that this code will change value in memory. That's all. Now let's look at the same function but at `atomic` variant. It looks more complex that its `non-atomic` variant:

```
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");
    }
}
```

First of all note that this function takes the same set of parameters that `__set_bit`, but additionally marked with the `__always_inline` attribute. The `__always_inline` is macro which defined in the [include/linux/compiler-gcc.h](#) and just expands to the `always_inline` attribute:

```
#define __always_inline inline __attribute__((always_inline))
```

which means that this function will be always inlined to reduce size of the Linux kernel image. Now let's try to understand implementation of the `set_bit` function. First of all we check a given number of bit at the beginning of the `set_bit` function. The `IS_IMMEDIATE` macro defined in the same [header](#) file and expands to the call of the builtin `gcc` function:

```
#define IS_IMMEDIATE(nr)      (__builtin_constant_p(nr))
```

The `__builtin_constant_p` builtin function returns `1` if the given parameter is known to be constant at compile-time and returns `0` in other case. We no need to use slow `bts` instruction to set bit if the given number of bit is known in compile time constant. We can just apply `bitwise or` for byte from the give address which contains given bit and masked number



of bits where high bit is `1` and other is zero. In other case if the given number of bit is not known constant at compile-time, we do the same as we did in the `__set_bit` function. The

`CONST_MASK_ADDR` macro:

```
#define CONST_MASK_ADDR(nr, addr)    BITOP_ADDR((void *)(addr) + ((nr)>>3))
```

expands to the give address with offset to the byte which contains a given bit. For example we have address `0x1000` and the number of bit is `0x9`. So, as `0x9` is one byte + one bit our address with be `addr + 1`:

```
>>> hex(0x1000 + (0x9 >> 3))
'0x1001'
```

The `CONST_MASK` macro represents our given number of bit as byte where high bit is `1` and other bits are `0`:

```
#define CONST_MASK(nr)                (1 << ((nr) & 7))
```

```
>>> bin(1 << (0x9 & 7))
'0b10'
```

In the end we just apply bitwise `or` for these values. So, for example if our address will be `0x4097` and we need to set `0x9` bit:

```
>>> bin(0x4097)
'0b100000010010111'
>>> bin((0x4097 >> 0x9) | (1 << (0x9 & 7)))
'0b100010'
```

the `ninth` bit will be set.

Note that all of these operations are marked with `LOCK_PREFIX` which is expands to the `lock` instruction which guarantees atomicity of this operation.

As we already know, besides the `set_bit` and `__set_bit` operations, the Linux kernel provides two inverse functions to clear bit in atomic and non-atomic context. They are `clear_bit` and `__clear_bit`. Both of these functions are defined in the same [header file](#) and takes the same set of arguments. But not only arguments are similar. Generally these functions are very similar on the `set_bit` and `__set_bit`. Let's look on the implementation of the non-atomic `__clear_bit` function:

```
static inline void __clear_bit(long nr, volatile unsigned long *addr)
{
    asm volatile("btr %1,%0" : ADDR : "Ir" (nr));
}
```

Yes. As we see, it takes the same set of arguments and contains very similar block of inline assembler. It just uses the `btr` instruction instead of `bts`. As we can understand from the function's name, it clears a given bit by the given address. The `btr` instruction acts like `bts`. This instruction also selects a given bit which is specified in the first operand, stores its value in the `CF` flag register and clears this bit in the given bit array which is specified with second operand.

The atomic variant of the `__clear_bit` is `clear_bit`:

```
static __always_inline void
clear_bit(long nr, volatile unsigned long *addr)
{
    if (IS_IMMEDIATE(nr)) {
        asm volatile(LOCK_PREFIX "andb %1,%0"
                     : CONST_MASK_ADDR(nr, addr)
                     : "iq" ((u8)~CONST_MASK(nr)));
    } else {
        asm volatile(LOCK_PREFIX "btr %1,%0"
                     : BITOP_ADDR(addr)
                     : "Ir" (nr));
    }
}
```

and as we can see it is very similar on `set_bit` and just contains two differences. The first difference it uses `btr` instruction to clear bit when the `set_bit` uses `bts` instruction to set bit. The second difference it uses negated mask and `and` instruction to clear bit in the given byte when the `set_bit` uses `or` instruction.

That's all. Now we can set and clear bit in any bit array and and we can go to other operations on bitmaps.

Most widely used operations on a bit arrays are set and clear bit in a bit array in the Linux kernel. But besides this operations it is useful to do additional operations on a bit array. Yet another widely used operation in the Linux kernel - is to know is a given bit set or not in a bit array. We can achieve this with the help of the `test_bit` macro. This macro is defined in the [arch/x86/include/asm/bitops.h](#) header file and expands to the call of the `constant_test_bit` or `variable_test_bit` depends on bit number:

```
#define test_bit(nr, addr) \
    (__builtin_constant_p(nr) \
     ? constant_test_bit((nr), (addr)) \
     : variable_test_bit((nr), (addr)))
```

So, if the `nr` is known in compile time constant, the `test_bit` will be expanded to the call of the `constant_test_bit` function or `variable_test_bit` in other case. Now let's look at implementations of these functions. Let's start from the `variable_test_bit` :

```
static inline int variable_test_bit(long nr, volatile const unsigned long *addr)
{
    int oldbit;

    asm volatile("bt %2,%1\n\t"
                 "sbb %0,%0"
                 : "=r" (oldbit)
                 : "m" (*(unsigned long *)addr), "Ir" (nr));

    return oldbit;
}
```

The `variable_test_bit` function takes similar set of arguments as `set_bit` and other function take. We also may see inline assembly code here which executes `bt` and `sbb` instruction. The `bt` or `bit test` instruction selects a given bit which is specified with first operand from the bit array which is specified with the second operand and stores its value in the `CF` bit of flags register. The second `sbb` instruction subtracts first operand from second and subtract value of the `CF`. So, here write a value of a given bit number from a given bit array to the `CF` bit of flags register and execute `sbb` instruction which calculates: `00000000 - CF` and writes the result to the `oldbit`.

The `constant_test_bit` function does the same as we saw in the `set_bit` :

```
static __always_inline int constant_test_bit(long nr, const volatile unsigned long *addr)
{
    return ((1UL << (nr & (BITS_PER_LONG-1))) &
            (addr[nr >> _BITOPS_LONG_SHIFT])) != 0;
}
```

It generates a byte where high bit is `1` and other bits are `0` (as we saw in `CONST_MASK`) and applies bitwise `and` to the byte which contains a given bit number.

The next widely used bit array related operation is to change bit in a bit array. The Linux kernel provides two helper for this:

- `__change_bit` ;
- `change_bit` .

As you already can guess, these two variants are atomic and non-atomic as for example `set_bit` and `__set_bit` . For the start, let's look at the implementation of the `__change_bit` function:

```
static inline void __change_bit(long nr, volatile unsigned long *addr)
{
    asm volatile("btc %1,%0" : ADDR : "Ir" (nr));
}
```

Pretty easy, is not it? The implementation of the `__change_bit` is the same as `__set_bit` , but instead of `bts` instruction, we are using `btc`. This instruction selects a given bit from a given bit array, stores its value in the `CF` and changes its value by the applying of complement operation. So, a bit with value `1` will be `0` and vice versa:

```
>>> int(not 1)
0
>>> int(not 0)
1
```

The atomic version of the `__change_bit` is the `change_bit` function:

```
static inline void change_bit(long nr, volatile unsigned long *addr)
{
    if (IS_IMMEDIATE(nr)) {
        asm volatile(LOCK_PREFIX "xorb %1,%0"
                     : CONST_MASK_ADDR(nr, addr)
                     : "iq" ((u8)CONST_MASK(nr)));
    } else {
        asm volatile(LOCK_PREFIX "btc %1,%0"
                     : BITOP_ADDR(addr)
                     : "Ir" (nr));
    }
}
```

It is similar on `set_bit` function, but also has two differences. The first difference is `xor` operation instead of `or` and the second is `bts` instead of `bts` .

For this moment we know the most important architecture-specific operations with bit arrays. Time to look at generic bitmap API.

## Common bit operations

Besides the architecture-specific API from the [arch/x86/include/asm/bitops.h](#) header file, the Linux kernel provides common API for manipulation of bit arrays. As we know from the beginning of this part, we can find it in the [include/linux/bitmap.h](#) header file and additionally in the \* [lib/bitmap.c](#) source code file. But before these source code files let's look into the [include/linux/bitops.h](#) header file which provides a set of useful macro. Let's look on some of they.

First of all let's look at following four macros:

- `for_each_set_bit`
- `for_each_set_bit_from`
- `for_each_clear_bit`
- `for_each_clear_bit_from`

All of these macros provide iterator over certain set of bits in a bit array. The first macro iterates over bits which are set, the second does the same, but starts from a certain bits. The last two macros do the same, but iterates over clear bits. Let's look on implementation of the `for_each_set_bit` macro:

```
#define for_each_set_bit(bit, addr, size) \
    for ((bit) = find_first_bit((addr), (size)); \
         (bit) < (size); \
         (bit) = find_next_bit((addr), (size), (bit) + 1))
```

As we may see it takes three arguments and expands to the loop from first set bit which is returned as result of the `find_first_bit` function and to the last bit number while it is less than given size.

Besides these four macros, the [arch/x86/include/asm/bitops.h](#) provides API for rotation of `64-bit` or `32-bit` values and etc.

The next [header](#) file which provides API for manipulation with a bit arrays. For example it provides two functions:

- `bitmap_zero` ;
- `bitmap_fill` .

To clear a bit array and fill it with `1` . Let's look on the implementation of the `bitmap_zero` function:

```
static inline void bitmap_zero(unsigned long *dst, unsigned int nbits)
{
    if (small_const_nbbits(nbbits))
        *dst = 0UL;
    else {
        unsigned int len = BITS_TO_LONGS(nbbits) * sizeof(unsigned long);
        memset(dst, 0, len);
    }
}
```

First of all we can see the check for `nbits`. The `small_const_nbbits` is macro which defined in the same header [file](#) and looks:

```
#define small_const_nbbits(nbbits) \
    (__builtin_constant_p(nbbits) && (nbits) <= BITS_PER_LONG)
```

As we may see it checks that `nbits` is known constant in compile time and `nbits` value does not overflow `BITS_PER_LONG` or `64`. If bits number does not overflow amount of bits in a `long` value we can just set to zero. In other case we need to calculate how many `long` values do we need to fill our bit array and fill it with `memset`.

The implementation of the `bitmap_fill` function is similar on implementation of the `bitmap_zero` function, except we fill a given bit array with `0xff` values or `0b11111111`:

```
static inline void bitmap_fill(unsigned long *dst, unsigned int nbits)
{
    unsigned int nlongs = BITS_TO_LONGS(nbbits);
    if (!small_const_nbbits(nbbits)) {
        unsigned int len = (nlongs - 1) * sizeof(unsigned long);
        memset(dst, 0xff, len);
    }
    dst[nlongs - 1] = BITMAP_LAST_WORD_MASK(nbbits);
}
```

Besides the `bitmap_fill` and `bitmap_zero` functions, the [include/linux/bitmap.h](#) header file provides `bitmap_copy` which is similar on the `bitmap_zero`, but just uses `memcpy` instead of `memset`. Also it provides bitwise operations for bit array like `bitmap_and`, `bitmap_or`, `bitmap_xor` and etc. We will not consider implementation of these functions because it is easy to understand implementations of these functions if you understood all from this part. Anyway if you are interested how did these function implemented, you may open [include/linux/bitmap.h](#) header file and start to research.

That's all.

# Links

- [bitmap](#)
- [linked data structures](#)
- [tree data structures](#)
- [hot-plug](#)
- [cpumasks](#)
- [IRQs](#)
- [API](#)
- [atomic operations](#)
- [xchg instruction](#)
- [cmpxchg instruction](#)
- [lock instruction](#)
- [bts instruction](#)
- [btr instruction](#)
- [bt instruction](#)
- [sbb instruction](#)
- [btc instruction](#)
- [man memcpy](#)
- [man memset](#)
- [CF](#)
- [inline assembler](#)
- [gcc](#)

# 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 learned about what the kernel does in its earliest stage. In the next step the kernel will initialize different things like `initrd` mounting, lockdep initialization, and many many other things, before we can see how the kernel runs 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 parts of the Linux kernel and in system programming in general. This is why we need to get acquainted with paging, before we proceed with the kernel initialization stuff.

**Paging** is a mechanism that translates a linear memory address to a physical address. If you have read the previous parts of this book, you may remember that we saw segmentation in real mode when physical addresses are calculated by shifting a segment register by four and adding an offset. We also saw segmentation in protected mode, where we used the descriptor tables and base addresses from descriptors with offsets to calculate the physical addresses. Now we will see paging in 64-bit mode.

As the 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... In this post I will try to explain the theory behind paging. Of course it will be closely related to the `x86_64` version of the Linux kernel, but we will not go into too much details (at least in this post).

## Enabling paging

There are three paging modes:

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

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

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

We already saw where those bits were set in [arch/x86/boot/compressed/head\\_64.S](#):

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

and

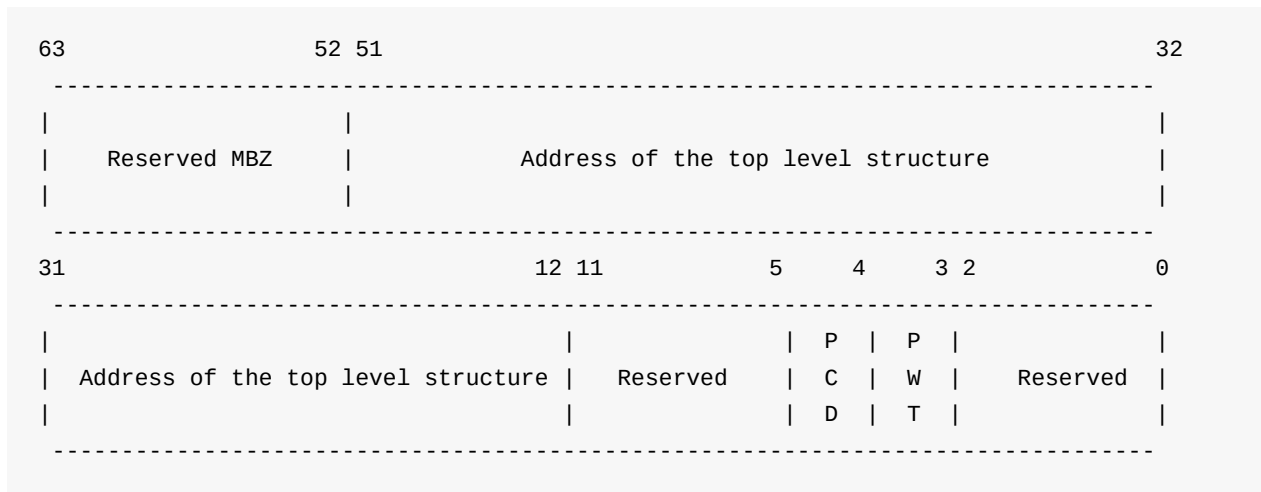
```
movl    $MSR_EFER, %ecx
rdmsr
btsl    $_EFER_LME, %eax
wrmsr
```

## Paging structures

Paging divides the linear address space into fixed-size pages. Pages can be mapped into the physical address space or external storage. This fixed size is `4096` bytes for the `x86_64` Linux kernel. To perform the translation from linear address to physical address, special structures are used. Every structure is `4096` bytes and contains `512` entries (this only for `PAE` and `IA32_EFER.LME` modes). Paging structures are hierarchical and the Linux kernel uses 4 level of paging in the `x86_64` architecture. The CPU uses a part of linear addresses to identify the entry in another paging structure which is at the lower level, 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 have already seen this in [arch/x86/boot/compressed/head\\_64.S](#):

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

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



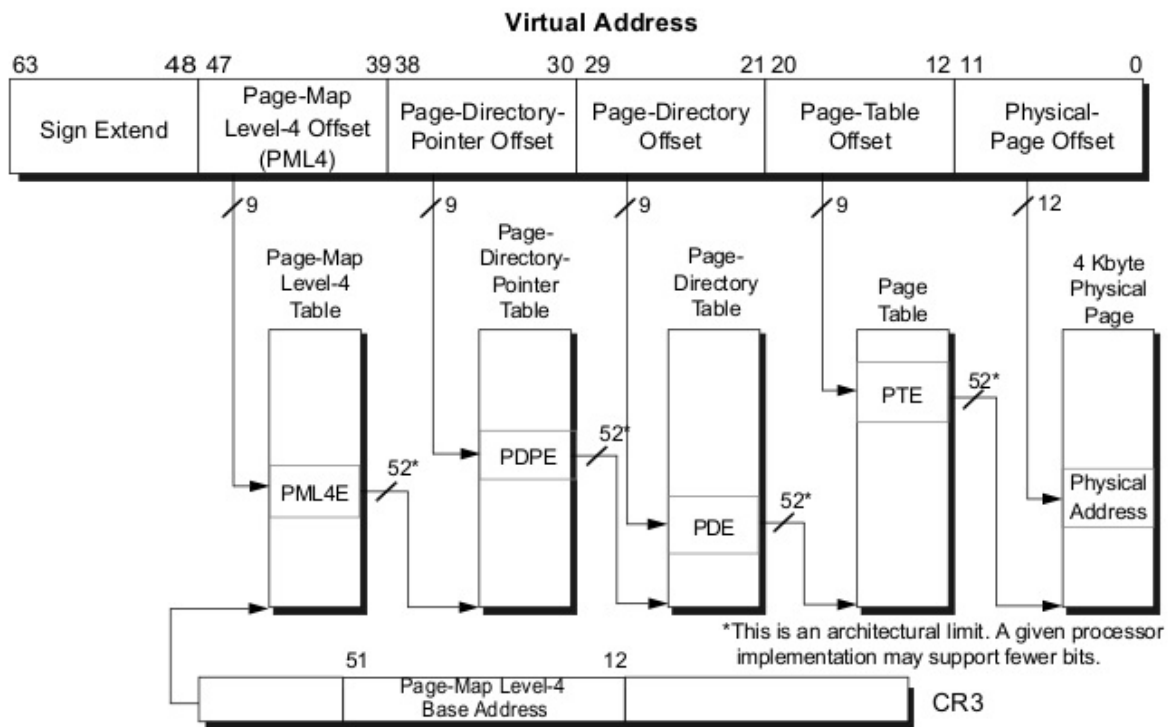
These fields have the following meanings:

- Bits 63:52 - reserved must be 0.
- Bits 51:12 - stores the address of the top level paging structure;
- Reserved - reserved must be 0;
- Bits 4 : 3 - 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;
- Bits 2 : 0 - ignored;

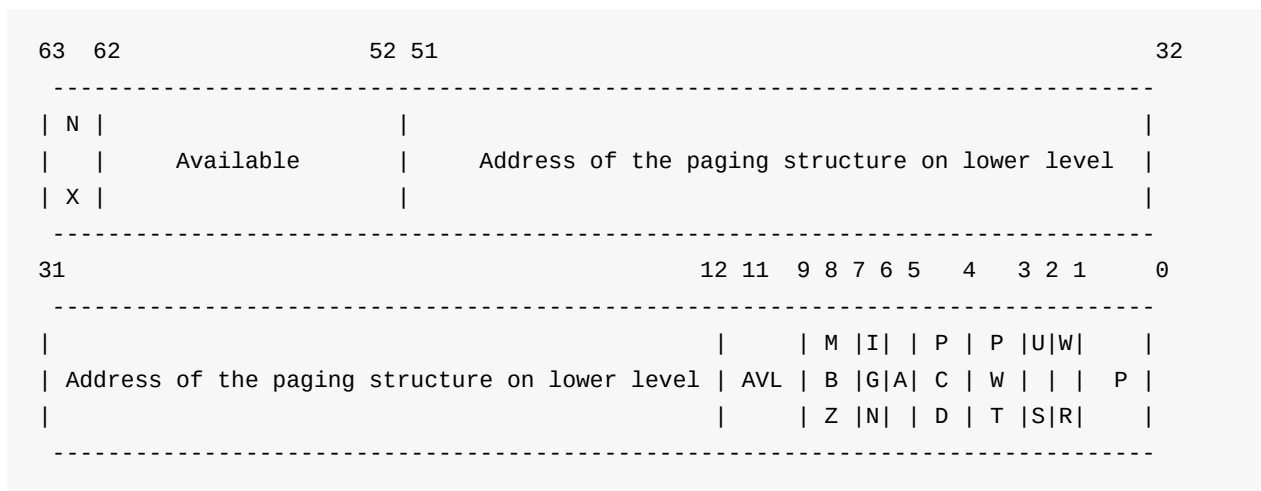
The linear address translation is following:

- A given linear address arrives to the [MMU](#) instead of memory bus.
- 64-bit linear address is split into 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 store an index into the paging structure level-4, `38:30` bits store index into the paging structure level-3, `29:21` bits store an index into the paging structure level-2, `20:12` bits store an index into the paging structure level-1 and `11:0` bits provide the offset into the physical page in byte.

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 is determined by the `CPL` (current privilege level). If `CPL < 3` it is a supervisor mode access level, otherwise it is a user mode access level. For example, the top level page table entry contains access bits and has the following structure:



Where:

- 63 bit - N/X bit (No Execute Bit) which 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;
- 11: 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 all the physical pages mapped by this table entry;
- R/W - read/write bit controls read/write access to all the 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, we know about the paging structures and their entries. Now let's see some details about 4-level paging in the Linux kernel.

## Paging structures in the Linux kernel

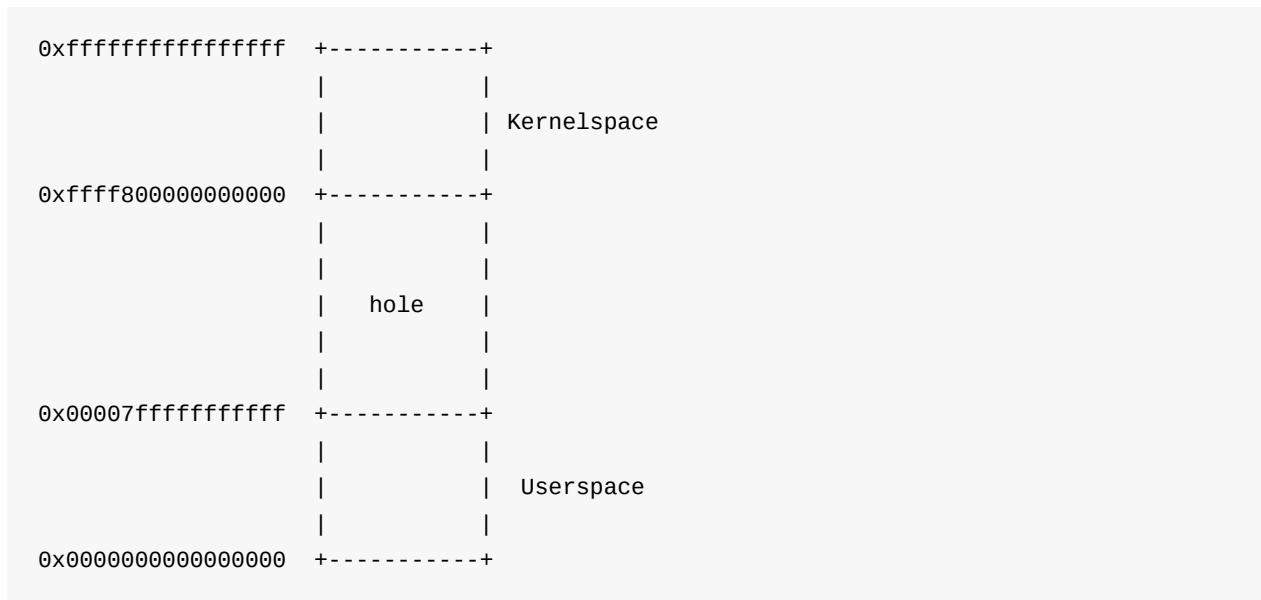
As we've seen, the Linux kernel in `x86_64` uses 4-level page tables. Their names are:

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

After you've compiled and installed the Linux kernel, you can see the `System.map` file which stores the virtual addresses of the functions that are used by the kernel. For example:

```
$ grep "start_kernel" System.map
ffffffff81efe497 T x86_64_start_kernel
ffffffff81efea2 T start_kernel
```

We can see `0xffffffff81efe497` here. I doubt you really have that much RAM installed. But anyway, `start_kernel` and `x86_64_start_kernel` will be executed. The address space in `x86_64` is  $2^{64}$  wide, but it's too large, that's why a smaller address space is used, only 48-bits wide. So we have a situation where the physical address space is limited to 48 bits, but addressing still performs with 64 bit pointers. How is this problem solved? Look at this diagram:



This solution is `sign extension`. Here we can see that the lower 48 bits of a virtual address can be used for addressing. Bits `63:48` can be either only zeroes or only ones. Note that the virtual address space is split into 2 parts:

- Kernel space
- Userspace

Userspace occupies the lower part of the virtual address space, from `0x0000000000000000` to `0x00007fffffffffff` and kernel space occupies the highest part from `0xffff800000000000` 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 other words which higher `63:48` bits are zeroes or ones are called `canonical` addresses. There is a `non-canonical` area between these memory regions. Together these two memory regions (kernel space and user space) are exactly  $2^{48}$  bits wide. We can find the 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 - ffff87fffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7fffffffffff (=64 TB) direct mapping of all phys. memory
fffc800000000000 - ffffc8fffffffffff (=40 bits) hole
fffc900000000000 - ffffe8fffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9fffffffffff (=40 bits) hole
ffffea0000000000 - ffffea0000000000 (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffc0000000000 (=44 bits) kasan shadow memory (16TB)
... unused hole ...
ffffff0000000000 - ffffffff7fffffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffff80000000 - ffffffff80000000 (=512 MB) kernel text mapping, from phys 0
ffffffff80000000 - ffffffff80000000 (=1525 MB) module mapping space
ffffffffffff600000 - fffffffffffffdfffff (=8 MB) vsyscalls
ffffffffffffe00000 - fffffffffffffffffff (=2 MB) unused hole

```

We can see here the memory map for user space, kernel space and the non-canonical area in-between them. The user space memory map is simple. Let's take a closer look at the kernel space. We can see that it starts from the guard hole which is reserved for the hypervisor. We can find the definition of this guard hole in [arch/x86/include/asm/page\\_64\\_types.h](#):

```
#define __PAGE_OFFSET _AC(0xffff800000000000, UL)
```

Previously this guard hole and `__PAGE_OFFSET` was from `0xffff800000000000` to `0xffff800000000000` to prevent access to non-canonical area, but was later extended by 3 bits for the hypervisor.

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

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

Usually kernel's `.text` starts here with the `CONFIG_PHYSICAL_START` offset. We have seen 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 check `vmlinux` with `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 there is the virtual memory region for kernel module, `vsyscalls` and an unused hole of 2 megabytes.

We've seen how virtual memory map in the kernel is laid out and how a virtual address is translated into a physical one. Let's take the following address as example:

```
0xffffffff81000000
```

In binary it will be:

```
1111111111111111 11111111 111111110 000001000 000000000 0000000000000
63:48      47:39      38:30      29:21      20:12      11:0
```

This virtual address is split in parts as described above:

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

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

## Conclusion

It's the end of this short part about paging theory. Of course this post doesn't cover every detail of paging, but soon we'll see in practice how the Linux kernel builds paging structures and works with them.

**Please note that English is not my first language and I am really sorry for any inconvenience. If you've found any mistakes please send me PR to [linux-insides](#).**



## 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, object code, shared libraries and core dumps. Linux and many UNIX-like operating systems use this format. Let's look at the structure of the ELF-64 Object File Format and some definitions in the linux kernel source code which 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 of the remaining parts, etc...;
- Program header table - lists 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 the description of the sections.

Now let's have a closer look on these components.

## ELF header

The ELF header is located at the beginning of the object file. Its main purpose is to locate all other parts of the object file. The File header contains the following fields:

- ELF identification - array of bytes which helps 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, an 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 the `elf64_hdr` structure which presents ELF64 header in the linux kernel source code:

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

This structure defined in the [elf.h](#)

## Sections

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

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

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

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

[elf.h](#)

## 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](#).

The ELF object file also contains other fields/structures which you can find in the [Documentation](#). Now let's a look at the `vmlinux` ELF object.

## vmlinux

`vmlinux` is also a relocatable ELF object file . We can take a look at it with the `readelf` util. First of all let's look at the 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:                    0x10000000
  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 a 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
```

We can then look this address up in the `vmlinux` ELF object 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 the address of the `startup_64` routine is not `ffffffff80000000`, but `fffffff81000000` 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 the documentation - `ffffffff80000000` and  
`__PHYSICAL_START` is `0x1000000` . That's why address of the `startup_64` is  
`ffffffff81000000` .

And at 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 FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
LOAD	0x0000000000200000	0xfffffffff81000000	0x0000000001000000 R E 200000
LOAD	0x0000000000cfd000	0x0000000000cfd000	0x0000000001e00000 RW 200000
LOAD	0x0000000001000000	0xfffffffff81e00000	0x0000000001f00000 RW 200000
LOAD	0x0000000001200000	0x0000000000000000	0x00000000014d98 RW 200000
LOAD	0x0000000001315000	0x00000000014d98	0x0000000001f15000 RWE 200000
NOTE	0x00000000011d000	0x000000000279000	0x0000000001917284 0x000000000000024 0x000000000000024 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. You can find all of these sections 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 want to know more, you can find the documentation - [here](#)

# Misc

This chapter contains parts which are not directly related to the Linux kernel source 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 execute `make` in the root 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 part and the kernel was the [4.2.0-rc3](#) release.

This makefile is the top makefile in the Linux kernel source code and the kernel building starts here. Yes, it is big, but moreover, if you've read the source code of the Linux kernel you may have noted that all directories containing source code has its own makefile. Of course it is not possible to describe how each source file is compiled and linked, so we will only study the standard compilation case. You will not find here building of the kernel's documentation, cleaning of the kernel source code, [tags](#) generation, [cross-compilation](#) related stuff, 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 better if you're already familiar with the [make](#) util, but I will try to describe every piece of code in this part anyway.

So let's start.

## Preparation before the kernel compilation

There are many things to prepare before the kernel compilation can be started. The main point here is to find and configure the type of compilation, to parse command line arguments that are passed to `make`, etc... So let's dive into the top `Makefile` of Linux kernel.

The top `Makefile` of Linux kernel 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 with the definition of following variables:

```

VERSION = 4
PATCHLEVEL = 2
SUBLEVEL = 0
EXTRAVERSION = -rc3
NAME = Hurr durr I'ma sheep

```

These variables determine the current version of Linux kernel and are used in different places, for example in the forming of the `KERNELVERSION` variable in the same `Makefile` :

```

KERNELVERSION = $(VERSION)$(if $(PATCHLEVEL),.$(PATCHLEVEL)$(if $(SUBLEVEL),.$(SUBLEVEL)))

```

After this we can see a couple of `ifeq` conditions 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` => verbose build. The first `ifeq` checks whether 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 `v` option. Otherwise we set the `KBUILD_VERBOSE` variable to zero. After this we check the value of `KBUILD_VERBOSE` variable and set values of the `quiet` and `q` variables depending on the value of `KBUILD_VERBOSE` variable. The `@` symbols suppress the output of command. And if it is present before a command the output will be something like this: `cc scripts/mod/empty.o` instead of `Compiling .... scripts/mod/empty.o` . In the end we just export all of these variables. The next `ifeq` statement checks that `0=/dir` option was passed to the `make` . This option allows to locate all output files in the given `dir` :

```

ifeq ($(KBUILD_SRC),)

ifeq ("$(origin 0)", "command line")
    KBUILD_OUTPUT := $(0)
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 represents the top directory of the kernel source code and whether it is empty (it is empty when the makefile is executed for the first time). We then set the `KBUILD_OUTPUT` variable to the value passed with the `o` option (if this option was passed). In the next step we check this `KBUILD_OUTPUT` variable and if it is set, we do following things:

- Store the value of `KBUILD_OUTPUT` in the temporary `saved-output` variable;
- Try to create the given output directory;
- Check that directory created, in other way print error message;
- If the custom output directory was created successfully, execute `make` again with the new directory (see the `-c` option).

The next `ifeq` statements check that the `c` or `M` options passed to `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 `c` option tells the `makefile` that we 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). We also check whether the `KBUILD_SRC` variable is set, and if it isn't, we set the `srctree` variable to `.`:

```
ifeq ($(KBUILD_SRC),)
    srctree := .
endif

objtree := .
src := $(srctree)
obj := $(objtree)

export srctree objtree VPATH
```

That tells `Makefile` that the kernel source tree will be in the current directory where `make` was executed. We then set `objtree` and other variables to this directory and export them. The next step is to get value for the `SUBARCH` variable that represents 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 the `uname` util that prints information about machine, operating system and architecture. As it gets the output of `uname`, it parses the output and assigns the result to the `SUBARCH` variable. Now that we have `SUBARCH`, we set the `SRCARCH` variable that provides the directory of the certain architecture and `hdr-arch` that provides the directory for the header files:

```
ifeq ($(ARCH),i386)
    SRCARCH := x86
endif
ifeq ($(ARCH),x86_64)
    SRCARCH := x86
endif

hdr-arch := $(SRCARCH)
```

Note `ARCH` is an alias for `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 is set to `.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 are related to the compilers used during Linux kernel compilation. We set the host compilers for the `c` and `c++` and the flags to be used with them:

```
HOSTCC      = gcc
HOSTCXX     = g++
HOSTCFLAGS  = -Wall -Wmissing-prototypes -Wstrict-prototypes -O2 -fomit-frame-pointer -s
HOSTCXXFLAGS = -O2
```

Next we get to the `cc` variable that represents compiler too, so why do we need the `HOST*` variables? `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 the definition of `KBUILD_MODULES` and `KBUILD_BUILTIN` variables that are used to determine what to compile (modules, kernel, 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 `KBUILD_BUILTIN` variable will depend on the `CONFIG_MODVERSIONS` kernel configuration parameter if we pass only `modules` to `make`. The next step is to include the `kbuild` file.

```
include scripts/Kbuild.include
```

The `Kbuild` or `Kernel Build System` is the special infrastructure to manage the build of the kernel and its modules. The `kbuild` files has the same syntax that makefiles do. 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](#), 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
...
...
...
```

We then define two other variables: `USERINCLUDE` and `LINUXINCLUDE`. They contain the 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 not the 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

We have now finished all the preparations, and next step in the main makefile is related to the kernel build. Before this moment, nothing has been printed to the terminal by `make`. But now the first steps of the compilation are started. We need to go to line 598 of the Linux kernel top makefile and we will find the `vmlinux` target there:

```
all: vmlinux
    include arch/$(SRCARCH)/Makefile
```

Don't worry that we have missed many lines in Makefile that are between `export RCS_FIND_IGNORE.....` and `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 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 `vmlinux-dirs` depends on two targets: `prepare` and `scripts` . `prepare` is 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](#), 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 `bin2c` :

```
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 [gcc](#) that tells make when to remake a source code file. The second program is `bin2c`, which depends on the value of the `CONFIG_BUILD_BIN2C` kernel configuration option and is a very little C program that allows to convert a binary on stdin to a C include on stdout. You can note here a strange notation: `hostprogs-y`, etc... This notation is used in the all `kbuild` files and you can read more about it in the [documentation](#). In our case `hostprogs-y` tells `kbuild` that there is one host program named `fixdep` that will be built from `fixdep.c` that is located in the same directory where the `Makefile` is. The first output after we execute `make` 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`, 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 (depends 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 built-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](#), etc... but I will not describe it here 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` file is the compressed Linux kernel image. We can get it by executing `make bzImage` after `vmlinux` is built. That, or we can just execute `make` without any argument and we 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 is to build the source code in the `arch/x86/boot` and `arch/x86/boot/compressed` directories, build `setup.bin` and `vmlinux.bin`, and build the `bzImage` from them 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` variable that 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 file is `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 V0__end 0xffffffff82ab0000
#define V0__text 0xffffffff81000000
```

They are the start and the end of the kernel. The second is `zoffset.h` depends 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 program - `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 `vmlinux.bin` is the `vmlinux` file with debugging information and comments stripped 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 the computed offset from the 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
```

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
```

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
```

Actually the `bzImage` is the concatenated `setup.bin` and the `vmlinux.bin` . In the end we will see the output which is familiar to all who once built 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 understand the process of building the Linux kernel.

## 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 the `Linker` page on Wikipedia, we will see 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 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 a simple project with the following structure:

```
*-linkers
*-main.c
*-lib.c
*-lib.h
```

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,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 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  %eax
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 the `e8 00 00 00 00` on the first line. The `e8` is the `opcode` of the `call`, and the remainder of the line is 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. Why only 4-bytes if an address can be 8-bytes in a `x86_64` (64-bit) machine? Actually we compiled the `main.c` source code file with the `-mmodel=small` ! From the `gcc` man page:

```
-mmodel=small
```

Generate code for the small code model: the program and its symbols must be linked in the

Of course we didn't pass this option to the `gcc` when we compiled the `main.c`, but it is the default. We know that our program will be linked in the lower 2 GB of the address space from the `gcc` manual extract above. Four bytes is therefore enough for this. So we have opcode of the `call` instruction and an unknown address. When we compile `main.c` with all its dependencies to an executable file, and then look at the factorial call we 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 start from `0x0`? You may already know that standard C programs are linked with the `glibc` C standard library (assuming the `-nostdlib` was not passed to the `gcc`). The compiled code for a program includes constructor 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 start from the code placed in the special `.init` section. We can see this 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 <_DYNALINK_0>
```

Not that it starts at the `0x00000000004003a8` address relative to the `glibc` code. We can check it also in the [ELF](#) output by running `readelf`:

```
$ readelf -d factorial | grep \((INIT\)
0x000000000000000c (INIT)          0x4003a8
```

So, the address of the `main` function is `0000000000400506` and 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 `x86_64`, so 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 long (`e8 18 00 00 00`) and the `0x18` is the offset of the call after 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, these will overlap.

What we have seen in this section is the `relocation` process. 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 that we know a little about linkers and relocation it is time to learn more about linkers by linking our object files.

## 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

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_
/tmp/buildd/glibc-2.19/csu/../sysdeps/x86_64/start.S:116: undefined reference to `__libc_
/tmp/buildd/glibc-2.19/csu/../sysdeps/x86_64/start.S:122: undefined reference to `__libc_
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:

- `crt0.o` ;
- `crti.o` .

define the function prologs/epilogs for the `.init` and `.fini` sections (with the `_init` and `_fini` symbols respectively).

The `crt0.o` object file contains these `.init` and `.fini` sections:

```
$ objdump -S /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../../x86_64-linux-gnu/crt0.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/crt0.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` environment variable. Let's try to link again with 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/crt0.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 FileSiz	VirtAddr MemSiz	PhysAddr Flags Align
PHDR	0x0000000000000040	0x0000000000400040	0x0000000000400040
	0x0000000000000188	0x0000000000000188	R E 8
INTERP	0x00000000000001c8	0x00000000004001c8	0x00000000004001c8
	0x00000000000001c	0x00000000000001c	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.d
03 .dynamic .got .got.plt .data
04 .dynamic
05 .note.ABI-tag
06
```

Note on the strange line:

```
INTERP      0x00000000000001c8 0x00000000004001c8 0x00000000004001c8
            0x00000000000001c 0x00000000000001c 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/../../../../
...
...
...
.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 a simple assembly program and 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 specify the name of our executable file and its format. The next command, `INPUT`, specifies 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 inside 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 = 0x2000000;
DATA_OFFSET   = 0x2000000;

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 insides](#)
- [linker](#)
- [object files](#)
- [glibc](#)
- [opcode](#)
- [ELF](#)
- [GNU linker](#)
- [My posts about assembly programming for x86\\_64](#)
- [readelf](#)

# Linux kernel development

## Introduction

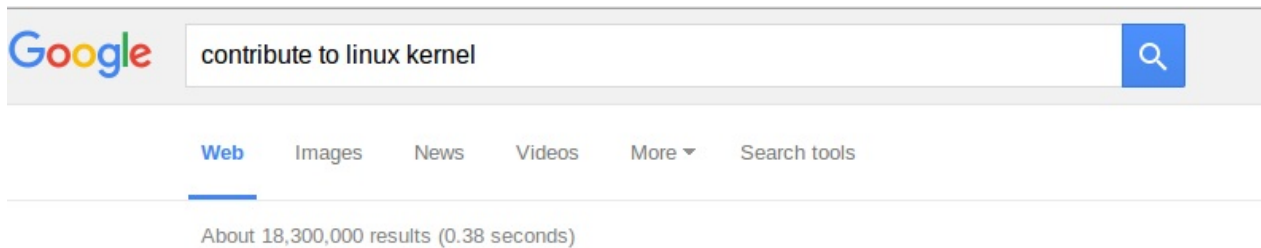
As you already may know, I've started a series of [blog posts](#) about assembler programming for `x86_64` architecture in the last year. I have never written a line of low-level code before this moment, except for a couple of toy `Hello World` examples in university. It was a long time ago and, as I already said, I didn't write low-level code at all. Some time ago I became interested in such things. I understood that I can write programs, but didn't actually understand how my program is arranged.

After writing some assembler code I began to understand how my program looks after compilation, **approximately**. But anyway, I didn't understand many other things. For example: what occurs when the `syscall` instruction is executed in my assembler, what occurs when the `printf` function starts to work or how can my program talk with other computers via network. [Assembler](#) programming language didn't give me answers to my questions and I decided to go deeper in my research. I started to learn from the source code of the Linux kernel and tried to understand the things that I'm interested in. The source code of the Linux kernel didn't give me the answers to **all** of my questions, but now my knowledge about the Linux kernel and the processes around it is much better.

I'm writing this part nine and a half months after I've started to learn from the source code of the Linux kernel and published the first [part](#) of this book. Now it contains forty parts and it is not the end. I decided to write this series about the Linux kernel mostly for myself. As you know the Linux kernel is very huge piece of code and it is easy to forget what does this or that part of the Linux kernel mean and how does it implement something. But soon the [linux-insides](#) repo became popular and after nine months it has `9096` stars:



It seems that people are interested in the insides of the Linux kernel. Besides this, in all the time that I have been writing `linux-insides`, I have received many questions from different people about how to begin contributing to the Linux kernel. Generally people are interested in contributing to open source projects and the Linux kernel is not an exception:



So, it seems that people are interested in the Linux kernel development process. I thought it would be strange if a book about the Linux kernel would not contain a part describing how to take a part in the Linux kernel development and that's why I decided to write it. You will not find information about why you should be interested in contributing to the Linux kernel in this part. But if you are interested how to start with Linux kernel development, this part is for you.

Let's start.

## How to start with Linux kernel

First of all, let's see how to get, build, and run the Linux kernel. You can run your custom build of the Linux kernel in two ways:

- Run the Linux kernel on a virtual machine;
- Run the Linux kernel on real hardware.

I'll provide descriptions for both methods. Before we start doing anything with the Linux kernel, we need to get it. There are a couple of ways to do this depending on your purpose. If you just want to update the current version of the Linux kernel on your computer, you can use the instructions specific to your Linux [distro](#).

In the first case you just need to download new version of the Linux kernel with the [package manager](#). For example, to upgrade the version of the Linux kernel to `4.1` for [Ubuntu \(Vivid Vervet\)](#), you will just need to execute the following commands:

```
$ sudo add-apt-repository ppa:kernel-ppa/ppa
$ sudo apt-get update
```

After this execute this command:

```
$ apt-cache showpkg linux-headers
```

and choose the version of the Linux kernel in which you are interested. In the end execute the next command and replace `${version}` with the version that you chose in the output of the previous command:

```
$ sudo apt-get install linux-headers-${version} linux-headers-${version}-generic linux-im
```

and reboot your system. After the reboot you will see the new kernel in the [grub](#) menu.

In the other way if you are interested in the Linux kernel development, you will need to get the source code of the Linux kernel. You can find it on the [kernel.org](#) website and download an archive with the Linux kernel source code. Actually the Linux kernel development process is fully built around `git` [version control system](#). So you can get it with `git` from the `kernel.org` :

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

I don't know how about you, but I prefer `github` . There is a [mirror](#) of the Linux kernel mainline repository, so you can clone it with:

```
$ git clone git@github.com:torvalds/linux.git
```

I use my own [fork](#) for development and when I want to pull updates from the main repository I just execute the following command:

```
$ git checkout master
$ git pull upstream master
```

Note that the remote name of the main repository is `upstream` . To add a new remote with the main Linux repository you can execute:

```
git remote add upstream git@github.com:torvalds/linux.git
```

After this you will have two remotes:

```
~/dev/linux (master) $ git remote -v
origin    git@github.com:0xAX/linux.git (fetch)
origin    git@github.com:0xAX/linux.git (push)
upstream  https://github.com/torvalds/linux.git (fetch)
upstream  https://github.com/torvalds/linux.git (push)
```

One is of your fork ( `origin` ) and the second is for the main repository ( `upstream` ).

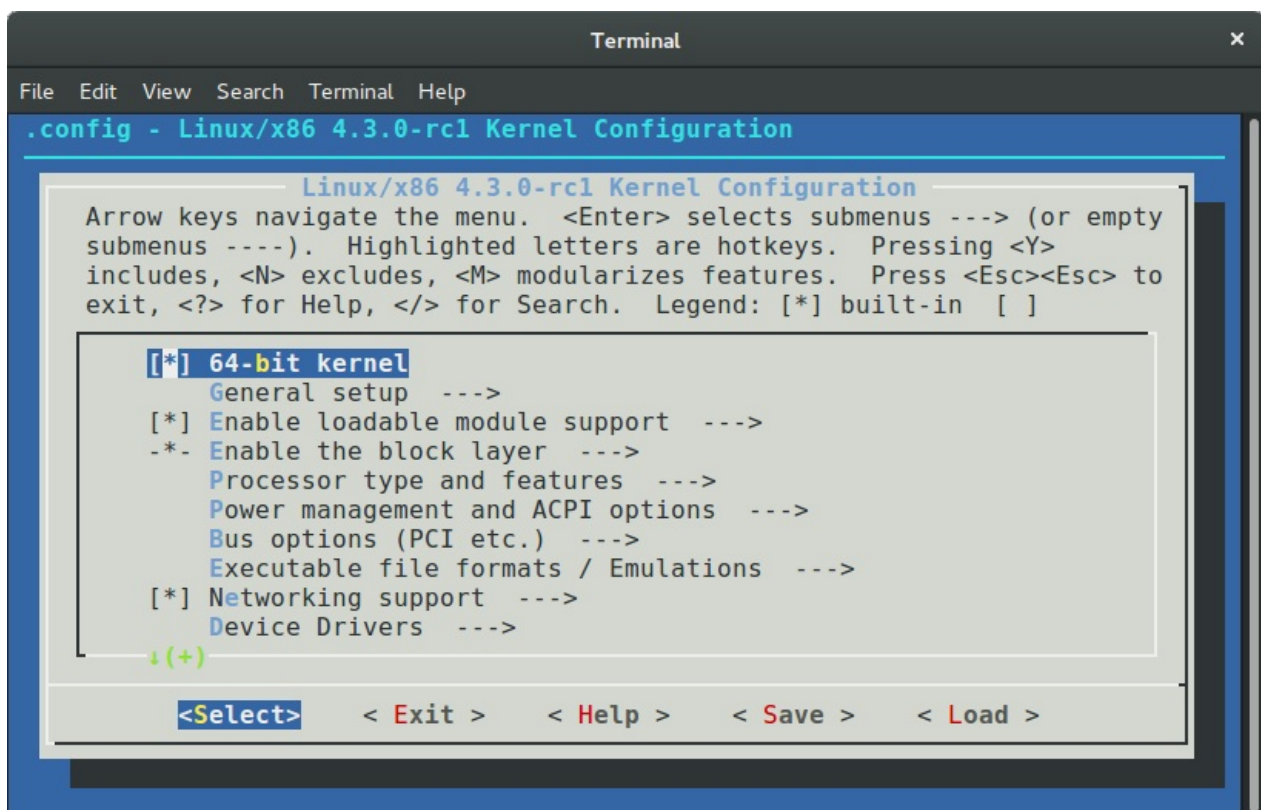
Now that we have a local copy of the Linux kernel source code, we need to configure and build it. The Linux kernel can be configured in different ways. The simplest way is to just copy the configuration file of the already installed kernel that is located in the `/boot` directory:

```
$ sudo cp /boot/config-$(uname -r) ~/dev/linux/.config
```

If your current Linux kernel was built with the support for access to the `/proc/config.gz` file, you can copy your actual kernel configuration file with this command:

```
$ cat /proc/config.gz | gunzip > ~/dev/linux/.config
```

If you are not satisfied with the standard kernel configuration that is provided by the maintainers of your distro, you can configure the Linux kernel manually. There are a couple of ways to do it. The Linux kernel root [Makefile](#) provides a set of targets that allows you to configure it. For example `menuconfig` provides a menu-driven interface for the kernel configuration:



The `defconfig` argument generates the default kernel configuration file for the current architecture, for example `x86_64 defconfig`. You can pass the `ARCH` command line argument to `make` to build `defconfig` for the given architecture:

```
$ make ARCH=arm64 defconfig
```

The `allnoconfig`, `allyesconfig` and `allmodconfig` arguments allow you to generate a new configuration file where all options will be disabled, enabled, and enabled as modules respectively. The `nconfig` command line arguments that provides `ncurses` based program with menu to configure Linux kernel:

```

Terminal
File Edit View Search Terminal Help

.config - Linux/x86 4.3.0-rc1 Kernel Configuration
Linux/x86 4.3.0-rc1 Kernel Configuration

[*] 64-bit kernel
    General setup --->
[*] Enable loadable module support --->
-*- Enable the block layer --->
    Processor type and features --->
    Power management and ACPI options --->
    Bus options (PCI etc.) --->
    Executable file formats / Emulations --->
[*] Networking support --->
    Device Drivers --->
    Firmware Drivers --->
    File systems --->
    Kernel hacking --->
    Security options --->
-*- Cryptographic API --->
[*] Virtualization --->
    Library routines --->

F1 Help F2 SymInfo F3 Help 2 F4 ShowAll F5 Back F6 Save F7 Load F8 SymSearch F9 Exit

```

And even `randconfig` to generate random Linux kernel configuration file. I will not write about how to configure the Linux kernel or which options to enable because it makes no sense to do so for two reasons: First of all I do not know your hardware and second, if you know your hardware, the only remaining task is to find out how to use programs for kernel configuration, and all of them are pretty simple to use.

OK, we now have the source code of the Linux kernel and configured it. The next step is the compilation of the Linux kernel. The simplest way to compile Linux kernel is to just execute:

```
$ make
scripts/kconfig/conf  --silentoldconfig Kconfig
#
# configuration written to .config
#
CHK      include/config/kernel.release
UPD      include/config/kernel.release
CHK      include/generated/uapi/linux/version.h
CHK      include/generated/utsrelease.h
...
...
...
OBJCOPY  arch/x86/boot/vmlinux.bin
AS        arch/x86/boot/header.o
LD        arch/x86/boot/setup.elf
OBJCOPY  arch/x86/boot/setup.bin
BUILD    arch/x86/boot/bzImage
Setup is 15740 bytes (padded to 15872 bytes).
System is 4342 kB
CRC 82703414
Kernel: arch/x86/boot/bzImage is ready  (#73)
```

To increase the speed of kernel compilation you can pass `-jN` command line argument to `make`, where `N` specifies the number of commands to run simultaneously:

```
$ make -j8
```

If you want to build Linux kernel for an architecture that differs from your current, the simplest way to do it pass two arguments:

- `ARCH` command line argument and the name of the target architecture;
- `CROSS_COMPILER` command line argument and the cross-compiler tool prefix;

For example if we want to compile the Linux kernel for the [arm64](#) with default kernel configuration file, we need to execute following command:

```
$ make -j4 ARCH=arm64 CROSS_COMPILER=aarch64-linux-gnu- defconfig
$ make -j4 ARCH=arm64 CROSS_COMPILER=aarch64-linux-gnu-
```

As result of compilation we can see the compressed kernel - `arch/x86/boot/bzImage`. Now that we have compiled the kernel, we can either install it on our computer or just run it in an emulator.

## Installing Linux kernel



As I already wrote we will consider two ways how to launch new kernel: In the first case we can install and run the new version of the Linux kernel on the real hardware and the second is launch the Linux kernel on a virtual machine. In the previous paragraph we saw how to build the Linux kernel from source code and as a result we have got compressed image:

```
...  
...  
...  
Kernel: arch/x86/boot/bzImage is ready (#73)
```

After we have got the `bzImage` we need to install `headers`, `modules` of the new Linux kernel with the:

```
$ sudo make headers_install  
$ sudo make modules_install
```

and directly the kernel itself:

```
$ sudo make install
```

From this moment we have installed new version of the Linux kernel and now we must tell the `bootloader` about it. Of course we can add it manually by the editing of the `/boot/grub2/grub.cfg` configuration file, but I prefer to use a script for this purpose. I'm using two different Linux distros: Fedora and Ubuntu. There are two different ways to update the `grub` configuration file. I'm using following script for this purpose:

```
#!/bin/bash  
  
source "term-colors"  
  
DISTRIBUTIVE=$(cat /etc/*-release | grep NAME | head -1 | sed -n -e 's/NAME=//p')  
echo -e "Distributive: ${Green}${DISTRIBUTIVE}${Color_Off}"  
  
if [[ "$DISTRIBUTIVE" == "Fedora" ]] ;  
then  
    su -c 'grub2-mkconfig -o /boot/grub2/grub.cfg'  
else  
    sudo update-grub  
fi  
  
echo "${Green}Done.${Color_Off}"
```

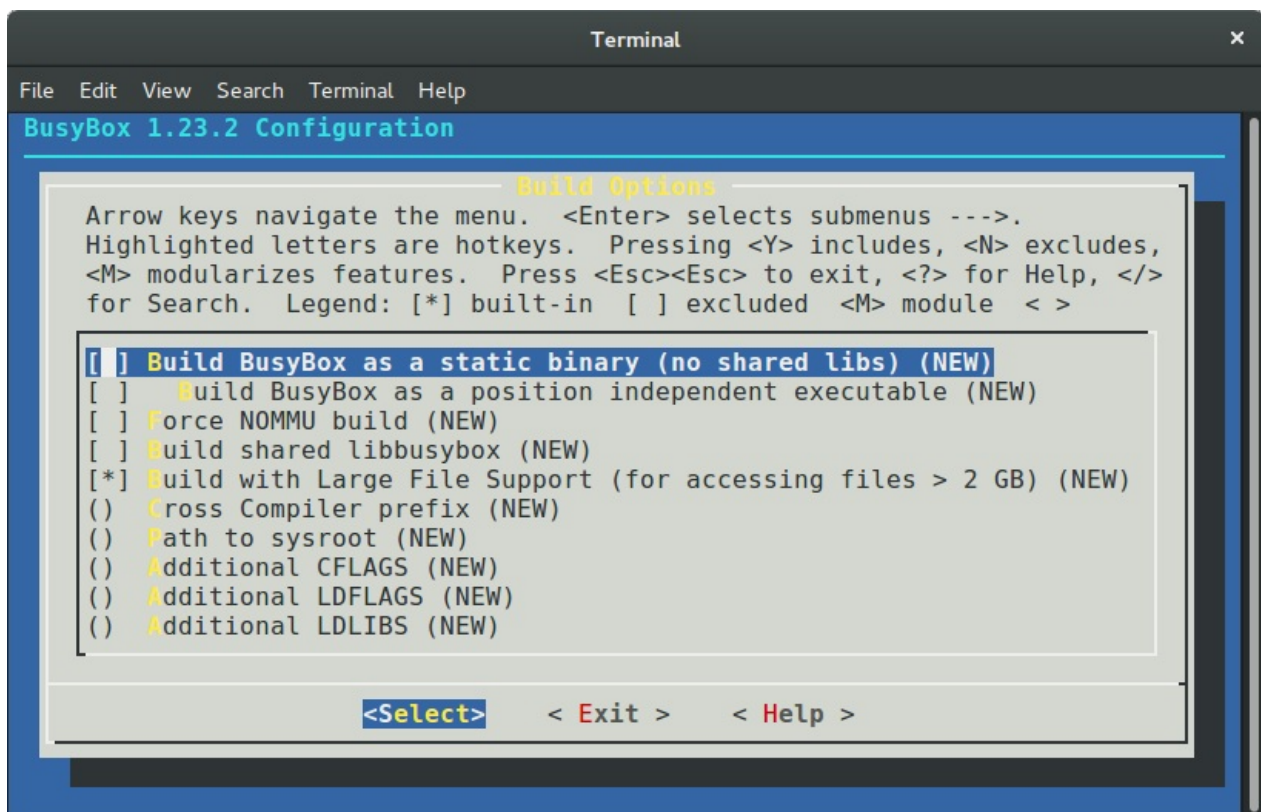
This is the last step of the new Linux kernel installation and after this you can reboot your computer and select new version of the kernel during boot.

The second case is to launch new Linux kernel in the virtual machine. I prefer [qemu](#). First of all we need to build initial ramdisk - [initrd](#) for this. The `initrd` is a temporary root file system that is used by the Linux kernel during initialization process while other filesystems are not mounted. We can build `initrd` with the following commands:

First of all we need to download [busybox](#) and run `menuconfig` for its configuration:

```
$ mkdir initrd
$ cd initrd
$ curl http://busybox.net/downloads/busybox-1.23.2.tar.bz2 | tar xjf -
$ cd busybox-1.23.2/
$ make menuconfig
$ make -j4
```

`busybox` is an executable file - `/bin/busybox` that contains a set of standard tools like [coreutils](#). In the `busysbox` menu we need to enable: `Build BusyBox as a static binary (no shared libs)` option:



We can find this menu in the:

```
Busybox Settings
--> Build Options
```

After this we exit from the `busysbox` configuration menu and execute following commands for building and installation of it:

```
$ make -j4
$ sudo make install
```

Now that `busybox` is installed, we can begin building our `initrd`. To do this, we go to the previous `initrd` directory and:

```
$ cd ..
$ mkdir -p initramfs
$ cd initramfs
$ mkdir -pv {bin,sbin,etc,proc,sys,usr/{bin,sbin}}
$ cp -av ../busybox-1.23.2/_install/* .
```

copy `busybox` fields to the `bin`, `sbin` and other directories. Now we need to create executable `init` file that will be executed as a first process in the system. My `init` file just mounts `procfs` and `sysfs` filesystems and executed shell:

```
#!/bin/sh

mount -t proc none /proc
mount -t sysfs none /sys

exec /bin/sh
```

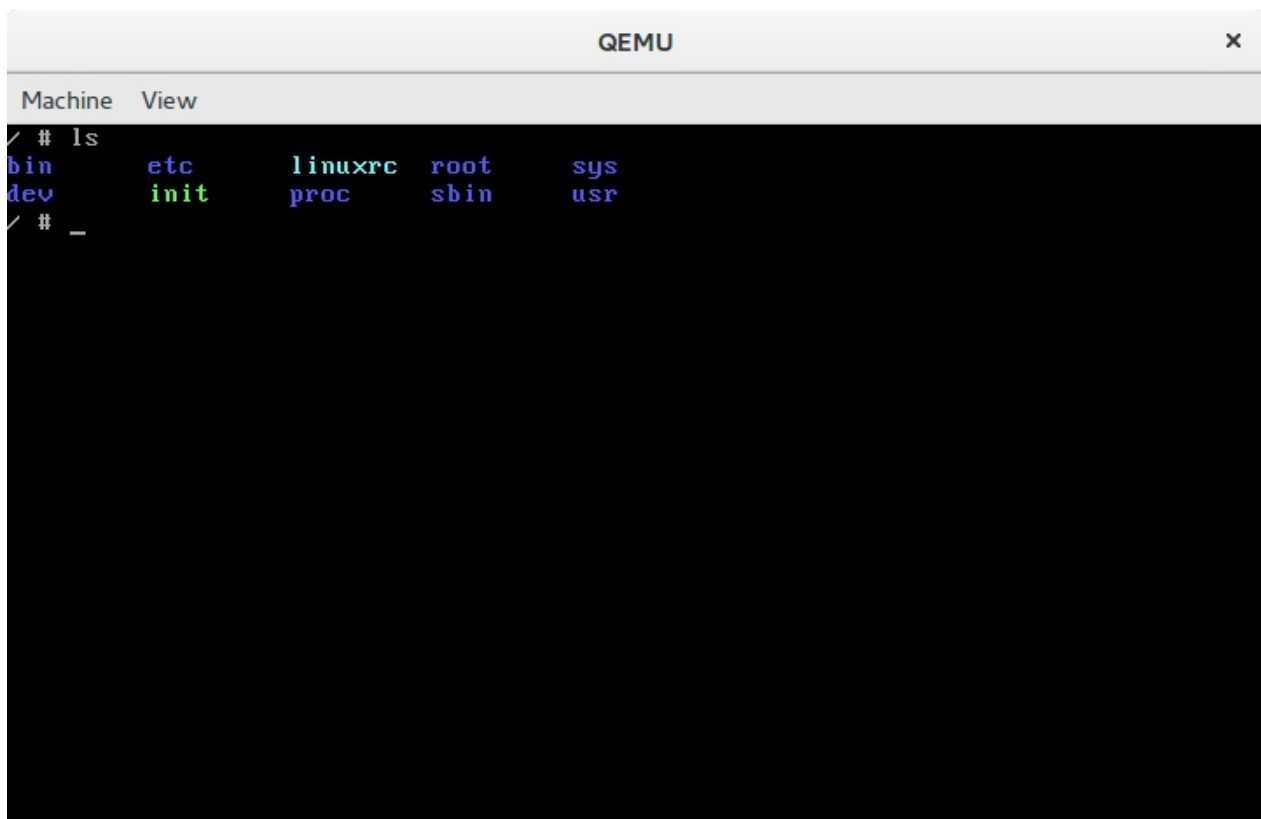
Now we can create an archive that will be our `initrd`:

```
$ find . -print0 | cpio --null -ov --format=newc | gzip -9 > ~/dev/initrd_x86_64.gz
```

We can now run our kernel in the virtual machine. As I already wrote I prefer `qemu` for this. We can run our kernel with the following command:

```
$ qemu-system-x86_64 -snapshot -m 8GB -serial stdio -kernel ~/dev/linux/arch/x86_64/boot/
```





From now we can run the Linux kernel in the virtual machine and this means that we can begin to change and test the kernel.

Consider using [ivandaviov/minimal](#) to automate the process of generating initrd.

## Getting started with the Linux Kernel Development

The main point of this paragraph is to answer two questions: What to do and what not to do before sending your first patch to the Linux kernel. Please, do not confuse this `to do` with `todo`. I have no answer what you can fix in the Linux kernel. I just want to tell you my workflow during experimenting with the Linux kernel source code.

First of all I pull the latest updates from Linus's repo with the following commands:

```
$ git checkout master
$ git pull upstream master
```

After this my local repository with the Linux kernel source code is synced with the [mainline](#) repository. Now we can make some changes in the source code. As I already wrote, I have no advice for you where you can start and what `TODO` in the Linux kernel. But the best place for newbies is `staging` tree. In other words the set of drivers from the [drivers/staging](#). The

maintainer of the `staging` tree is [Greg Kroah-Hartman](#) and the `staging` tree is that place where your trivial patch can be accepted. Let's look on a simple example that describes how to generate patch, check it and send to the [Linux kernel mail listing](#).

If we look in the driver for the [Digi International EPCA PCI](#) based devices, we will see the `dgap_sindex` function on line 295:

```
static char *dgap_sindex(char *string, char *group)
{
    char *ptr;

    if (!string || !group)
        return NULL;

    for (; *string; string++) {
        for (ptr = group; *ptr; ptr++) {
            if (*ptr == *string)
                return string;
        }
    }

    return NULL;
}
```

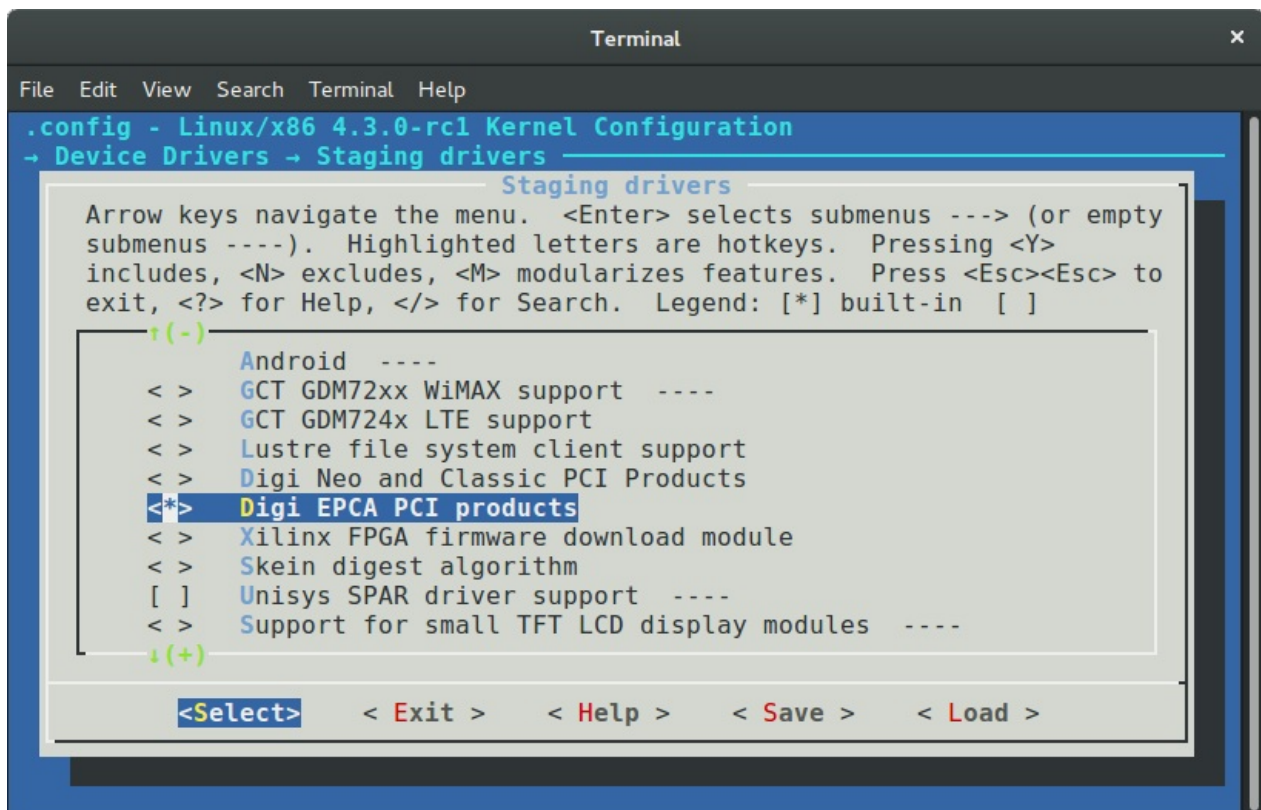
This function looks for a match of any character in the group and returns that position. During research of source code of the Linux kernel, I have noted that the [lib/string.c](#) source code file contains the implementation of the `strpbrk` function that does the same thing as `dgap_sindex`. It is not a good idea to use a custom implementation of a function that already exists, so we can remove the `dgap_sindex` function from the [drivers/staging/dgap/dgap.c](#) source code file and use the `strpbrk` instead.

First of all let's create new `git` branch based on the current master that synced with the Linux kernel mainline repo:

```
$ git checkout -b "dgap-remove-dgap_sindex"
```

And now we can replace the `dgap_sindex` with the `strpbrk`. After we did all changes we need to recompile the Linux kernel or just `dgap` directory. Do not forget to enable this driver in the kernel configuration. You can find it in the:

```
Device Drivers
--> Staging drivers
----> Digi EPCA PCI products
```



Now is time to make commit. I'm using following combination for this:

```
$ git add .
$ git commit -s -v
```

After the last command an editor will be opened that will be chosen from `$GIT_EDITOR` or `$EDITOR` environment variable. The `-s` command line argument will add `Signed-off-by` line by the committer at the end of the commit log message. You can find this line in the end of each commit message, for example - [00cc1633](#). The main point of this line is the tracking of who did a change. The `-v` option show unified diff between the HEAD commit and what would be committed at the bottom of the commit message. It is not necessary, but very useful sometimes. A couple of words about commit message. Actually a commit message consists from two parts:

The first part is on the first line and contains short description of changes. It starts from the `[PATCH]` prefix followed by a subsystem, driver or architecture name and after `:` symbol short description. In our case it will be something like this:

```
[PATCH] staging/dgap: Use strpbrk() instead of dgap_sindex()
```

After short description usually we have an empty line and full description of the commit. In our case it will be:

```
The <linux/string.h> provides strpbrk() function that does the same that the
dgap_sindex(). Let's use already defined function instead of writing custom.
```

And the `Sign-off-by` line in the end of the commit message. Note that each line of a commit message must not be longer than 80 symbols and commit message must describe your changes in details. Do not just write a commit message like: `Custom function removed`, you need to describe what you did and why. The patch reviewers must know what they review. Besides this commit messages in this view are very helpful. Each time when we can't understand something, we can use [git blame](#) to read description of changes.

After we have committed changes time to generate patch. We can do it with the `format-patch` command:

```
$ git format-patch master
0001-staging-dgap-Use-strpbrk-instead-of-dgap_sindex.patch
```

We've passed name of the branch (`master` in this case) to the `format-patch` command that will generate a patch with the last changes that are in the `dgap-remove-dgap_sindex` branch and not are in the `master` branch. As you can note, the `format-patch` command generates file that contains last changes and has name that is based on the commit short description. If you want to generate a patch with the custom name, you can use `--stdout` option:

```
$ git format-patch master --stdout > dgap-patch-1.patch
```

The last step after we have generated our patch is to send it to the Linux kernel mailing list. Of course, you can use any email client, `git` provides a special command for this: `git send-email`. Before you send your patch, you need to know where to send it. Yes, you can just send it to the Linux kernel mailing list address which is `linux-kernel@vger.kernel.org`, but it is very likely that the patch will be ignored, because of the large flow of messages. The better choice would be to send the patch to the maintainers of the subsystem where you have made changes. To find the names of these maintainers use the `get_maintainer.pl` script. All you need to do is pass the file or directory where you wrote code.

```
$ ./scripts/get_maintainer.pl -f drivers/staging/dgap/dgap.c
Lidza Louina <lidza.louina@gmail.com> (maintainer:DIGI EPCA PCI PRODUCTS)
Mark Hounscheil <markh@compro.net> (maintainer:DIGI EPCA PCI PRODUCTS)
Daeseok Youn <daeseok.youn@gmail.com> (maintainer:DIGI EPCA PCI PRODUCTS)
Greg Kroah-Hartman <gregkh@linuxfoundation.org> (supporter:STAGING SUBSYSTEM)
driverdev-devel@linuxdriverproject.org (open list:DIGI EPCA PCI PRODUCTS)
devel@driverdev.osuosl.org (open list:STAGING SUBSYSTEM)
linux-kernel@vger.kernel.org (open list)
```

You will see the set of the names and related emails. Now we can send our patch with:

```
$ git send-email --to "Lidza Louina <lidza.louina@gmail.com>" \
--cc "Mark Hounscheil <markh@compro.net>" \
--cc "Daeseok Youn <daeseok.youn@gmail.com>" \
--cc "Greg Kroah-Hartman <gregkh@linuxfoundation.org>" \
--cc "driverdev-devel@linuxdriverproject.org" \
--cc "devel@driverdev.osuosl.org" \
--cc "linux-kernel@vger.kernel.org"
```

That's all. The patch is sent and now you only have to wait for feedback from the Linux kernel developers. After you send a patch and a maintainer accepts it, you will find it in the maintainer's repository (for example [patch](#) that you saw in this part) and after some time the maintainer will send a pull request to Linus and you will see your patch in the mainline repository.

That's all.

## Some advice

In the end of this part I want to give you some advice that will describe what to do and what not to do during development of the Linux kernel:

- Think, Think, Think. And think again before you decide to send a patch.
- Each time when you have changed something in the Linux kernel source code - compile it. After any changes. Again and again. Nobody likes changes that don't even compile.
- The Linux kernel has a coding style [guide](#) and you need to comply with it. There is great script which can help to check your changes. This script is - [scripts/checkpatch.pl](#). Just pass source code file with changes to it and you will see:

```
$ ./scripts/checkpatch.pl -f drivers/staging/dgap/dgap.c
WARNING: Block comments use * on subsequent lines
#94: FILE: drivers/staging/dgap/dgap.c:94:
+/*
+   SUPPORTED PRODUCTS

CHECK: spaces preferred around that '|' (ctx:VxV)
#143: FILE: drivers/staging/dgap/dgap.c:143:
+   { PPCM,          PCI_DEV_XEM_NAME,      64, (T_PCXM|T_PCLITE|T_PCIBUS) },
```

Also you can see problematic places with the help of the `git diff` :



```
~/dev/linux (dgap-remove-dgap_sindex) $ git diff
diff --git a/init/main.c b/init/main.c
index 9e64d70..af379a5 100644
--- a/init/main.c
+++ b/init/main.c
@@ -153,6 +153,8 @@ EXPORT_SYMBOL(reset_devices);
static int __init set_reset_devices(char *str)
{
    reset_devices = 1;
+
+
    return 1;
}
```

- [Linus doesn't accept github pull requests](#)
- If your change consists from some different and unrelated changes, you need to split the changes via separate commits. The `git format-patch` command will generate patches for each commit and the subject of each patch will contain a `vN` prefix where the `N` is the number of the patch. If you are planning to send a series of patches it will be helpful to pass the `--cover-letter` option to the `git format-patch` command. This will generate an additional file that will contain the cover letter that you can use to describe what your patchset changes. It is also a good idea to use the `--in-reply-to` option in the `git send-email` command. This option allows you to send your patch series in reply to your cover message. The structure of the your patch will look like this for a maintainer:

```
|--> cover letter
|----> patch_1
|----> patch_2
```

You need to pass `message-id` as an argument of the `--in-reply-to` option that you can find in the output of the `git send-email` :

It's important that your email be in the [plain text](#) format. Generally, `send-email` and `format-patch` are very useful during development, so look at the documentation for the commands and you'll find some useful options such as: [git send-email](#) and [git format-patch](#).

- Do not be surprised if you do not get an immediate answer after you send your patch. Maintainers can be very busy.
- The [scripts](#) directory contains many different useful scripts that are related to Linux kernel development. We already saw two scripts from this directory: the `checkpatch.pl` and the `get_maintainer.pl` scripts. Outside of those scripts, you can find the

[stackusage](#) script that will print usage of the stack, [extract-vmlinux](#) for extracting an uncompressed kernel image, and many others. Outside of the `scripts` directory you can find some very useful [scripts](#) by [Lorenzo Stoakes](#) for kernel development.

- Subscribe to the Linux kernel mailing list. There are a large number of letters every day on `lkm1`, but it is very useful to read them and understand things such as the current state of the Linux kernel. Other than `lkm1` there are [set](#) mailing listings which are related to the different Linux kernel subsystems.
- If your patch is not accepted the first time and you receive feedback from Linux kernel developers, make your changes and resend the patch with the `[PATCH vN]` prefix (where `N` is the number of patch version). For example:

```
[PATCH v2] staging/dgap: Use strpbrk() instead of dgap_sindex()
```

Also it must contain a changelog that describes all changes from previous patch versions. Of course, this is not an exhaustive list of requirements for Linux kernel development, but some of the most important items were addressed.

Happy Hacking!

## Conclusion

I hope this will help others join the Linux kernel community! If you have any questions or suggestions, write me at [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

- [blog posts about assembly programming for x86\\_64](#)
- [Assembler](#)
- [distro](#)
- [package manager](#)
- [grub](#)
- [kernel.org](#)
- [version control system](#)
- [arm64](#)
- [bzImage](#)
- [qemu](#)

- [initrd](#)
- [busybox](#)
- [coreutils](#)
- [procfs](#)
- [sysfs](#)
- [Linux kernel mail listing archive](#)
- [Linux kernel coding style guide](#)
- [How to Get Your Change Into the Linux Kernel](#)
- [Linux Kernel Newbies](#)
- [plain text](#)

## 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](#)

## Useful links

- [Linux x86 Program Start Up](#)
- [Memory Layout in Program Execution \(32 bits\)](#)

## 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](#)
- [DongLiang Mu](#)
- [Johan Manuel](#)
- [Brian Rak](#)
- [Robin Peiremans](#)
- [xiaoqiang zhao](#)
- [aouete](#)
- [Dennis Birkholz](#)
- [Anton Tyurin](#)
- [Bogdan Kulbida](#)
- [Matt Hudgins](#)
- [Ruth Grace Wong](#)
- [Jeremy Lacomis](#)

- [Dubyah](#)
- [Matthieu Tardy](#)
- [michaelian ennis](#)
- [Amitay Stern](#)
- [Matt Todd](#)
- [Piyush Pangtey](#)