Contents

1 Kernel Boot Process

This chapter describes the linux kernel boot process. Here you will see a series of posts which describes 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 random address randomization - describes randomization of the Linux kernel load address.

This chapter coincides with `Linux kernel v4.17`.

2 Kernel booting process. Part 1.

2.1 From the bootloader to the kernel

If you read my previous blog posts, you might have noticed that I have been involved with low-level programming for some time. I wrote some posts about assembly programming for `x86_64` Linux and, at the same time, started to dive into the Linux kernel 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 the **x86_64** architecture.

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 github repo 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 official documentation, just learning and sharing knowledge.

Required knowledge

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

Anyway, if you're just starting to learn such tools, I will try to explain some parts during this and the following posts. Alright, this is the end of the simple introduction. Let's start to dive into the Linux kernel and low-level stuff!

I started writing these posts at the time of the `3.18` Linux kernel, and many things have changed since that time. If there are changes, I will update the posts accordingly.

2.2 The Magical Power Button, What happens next?

Although this is a series of posts about the Linux kernel, we won't start directly from the kernel code. As soon as you press the magical power button on your laptop or desktop computer, it starts working. The motherboard sends a signal to the power supply device. After receiving the signal, the power supply provides the proper amount of electricity to the computer. 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 predefined values for each of them.

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

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

The processor starts working in real mode. Let's back up a little and try to understand memory segmentation in this mode. Real mode is supported on all x86-compatible processors, from the 8086 CPU 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 `0-0xFFFFF` or `1 megabyte` address space. But it only has `16-bit` registers, which have a maximum address of `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 (64 KB). Since we cannot address memory above `64 KB` with 16-bit registers, an alternate method was devised.

An address consists of two parts: a segment selector, which has a 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 to it:

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

For example, if `CS:IP` is `0x2000:0x0010`, then the corresponding physical address will be:

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

But, if we take the largest segment selector and offset, `0xffff:0xffff`, then the resulting address will be:

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

which is `65520` bytes past the first megabyte. Since only one megabyte is accessible in real mode, `0x10ffef` becomes `0x00ffef` with the A20 line disabled.

Ok, now we know a little bit about real mode and its memory addressing. Let's get back to discussing 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` changes.

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

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

We get `0xfffffff0`, which is 16 bytes below 4GB. This point is called the reset vector. It's the memory location at which the CPU expects to find the first instruction to execute after reset. It contains a jump (`jmp`) instruction that usually points to the BIOS (Basic Input/Output System) entry point. For example, if we look in the coreboot source code (`src/cpu/x86/16bit/reset16.inc`), we see:

```
    .section ".reset", "ax", %progbits
    .code16
.globl  _start
_start:
    .byte  0xe9
    .int   _start16bit - ( . + 2 )
    ...
```

Here we can see the `jmp` instruction opcode, which is `0xe9`, and its destination address at `_start16bit - ( . + 2)`.

We also see that the `reset` section is `16` bytes and is compiled to start from the address `0xfffffff0` (`src/cpu/x86/16bit/reset16.ld`):

```
SECTIONS {
    /* Trigger an error if I have an unuseable start address */
    _bogus = ASSERT(_start16bit >= 0xffff0000, "_start16bit too low. Please report
    _ROMTOP = 0xfffffff0;
    . = _ROMTOP;
    .reset . : {
        *(.reset);
        . = 15;
        BYTE(0x00);
    }
}
```

Now the BIOS starts. After initializing and checking the hardware, the BIOS 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, where each sector is `512` bytes. The final two bytes of the first sector are `0x55` and `0xaa`, which designates to the BIOS that this device is bootable.

For example:

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

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

    int 0x10
    jmp $
```

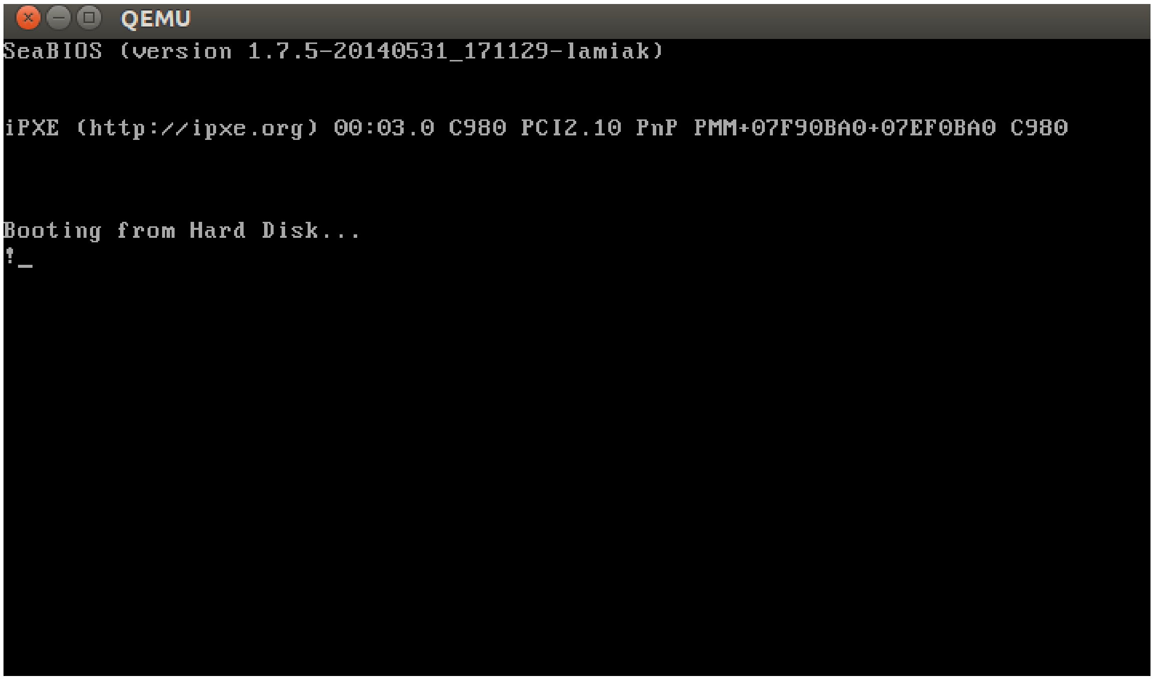

Figure 1: Simple bootloader which prints only `!`

```
times 510-($-$$) db 0

db 0x55
db 0xaa
```

Build and run this with:

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

This will instruct QEMU to use the `boot` binary that 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 it 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 remaining `510` bytes with zeros and finishes with the two magic bytes `0xaa` and `0x55`.

You can see a binary dump of this using the `objdump` utility:

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

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

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

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

just as explained above. We have only 16-bit general purpose registers, which has a maximum value of `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`. An 8086 processor (which was the first processor with real mode), in contrast, has a 20-bit address line. Since `2^20 = 1048576` is 1MB, this means that the actual available memory is 1MB.

In general, real mode's memory map is as follows:

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

At the beginning of this post, I wrote that the first instruction executed by the CPU is located at address `0xFFFFFFF0`, which is much larger than `0xFFFFF` (1MB). How can the CPU access this address in real mode? The answer is in the coreboot documentation:

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

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

2.3 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 a bootloader to implement Linux support. This example will describe GRUB 2.

Continuing from before, now that the BIOS has chosen a boot device and transferred control to the boot sector code, execution starts from boot.img. Its code is very simple, due to the limited amount of space available. It 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, which contains GRUB 2's kernel and drivers for handling filesystems, into memory. After loading the rest of the core image, it executes the grub_main function.

The `grub_main` function 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, the `grub_main` function moves grub to normal mode. The `grub_normal_execute` function (from the `grub-core/normal/main.c` source code file) completes the final preparations and shows a menu to select an operating system. When we select one of the grub menu entries, the `grub_menu_execute_entry` function runs, executing the grub `boot` command and 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 offset `0x01f1` from the kernel setup code. You may look at the boot linker script to confirm the value of this offset. 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 (which are only marked as being type **write** in the Linux boot protocol, such as in this example) with values either received from the command line or calculated during booting. (We will not go over full descriptions and explanations for all fields of the kernel setup header for now, but we shall do so when discussing how 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, memory will be mapped as follows 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            | <- Boot sector entry point 0x7C00
001000   +------------------------+
         | Reserved for MBR/BIOS  |
000800   +------------------------+
         | Typically used by MBR  |
000600   +------------------------+
         | BIOS use only          |
000000   +------------------------+
```

When the bootloader transfers control to the kernel, it starts at:

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

where `X` is the address of the kernel boot sector being loaded. In my case, `X` is `0x10000`, as we can see in a memory dump:

The bootloader has now loaded the Linux kernel into memory, filled the header fields, and then jumped to the corresponding memory address. We now move directly to the kernel setup code.

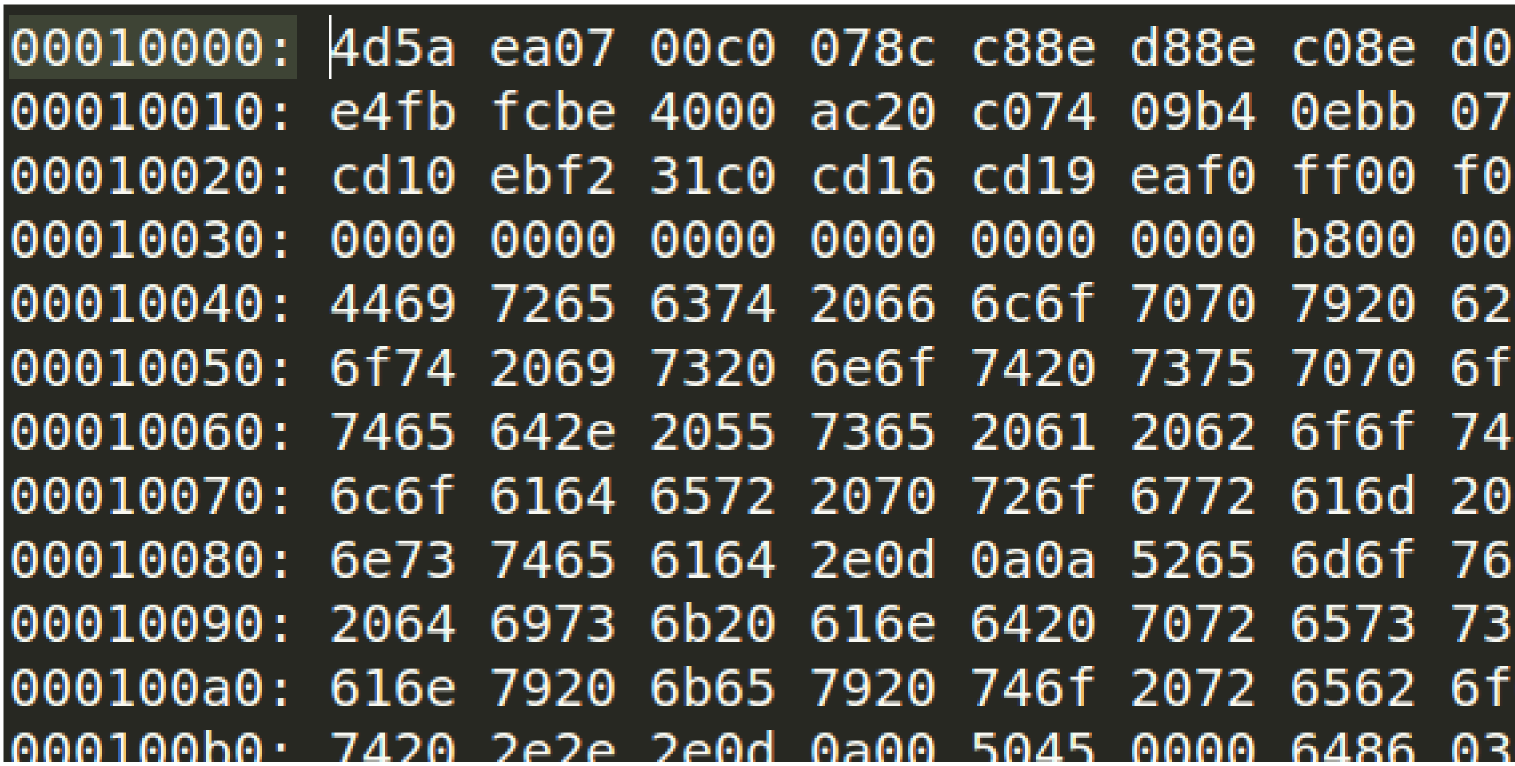

Figure 2: kernel first address

2.4 The Beginning of the Kernel Setup Stage

Finally, we are in the kernel! Technically, the kernel hasn't run yet. First, the kernel setup part must configure stuff such as the decompressor and some memory management related things, to name a few. After all these things are done, the kernel setup part will decompress the actual kernel and jump to it. Execution of the setup part starts from arch/x86/boot/header.S at the _start symbol.

It may look a bit strange at first sight, as there are several instructions before it. A long time ago, the Linux kernel had its own bootloader. Now, however, if you run, for example,

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

then you will see:

Actually, the file **header.S** starts with the magic number MZ (see image above), the error message that displays and, following that, the 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 support. We won't be looking into its inner workings right now but will cover it in upcoming chapters.

The actual kernel setup entry point is:

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

The bootloader (GRUB 2 and others) knows about this point (at an offset of `0x200` from `MZ`) and jumps directly to it, despite the fact that **header.S** starts from the **.bstext** section, which prints an error message:

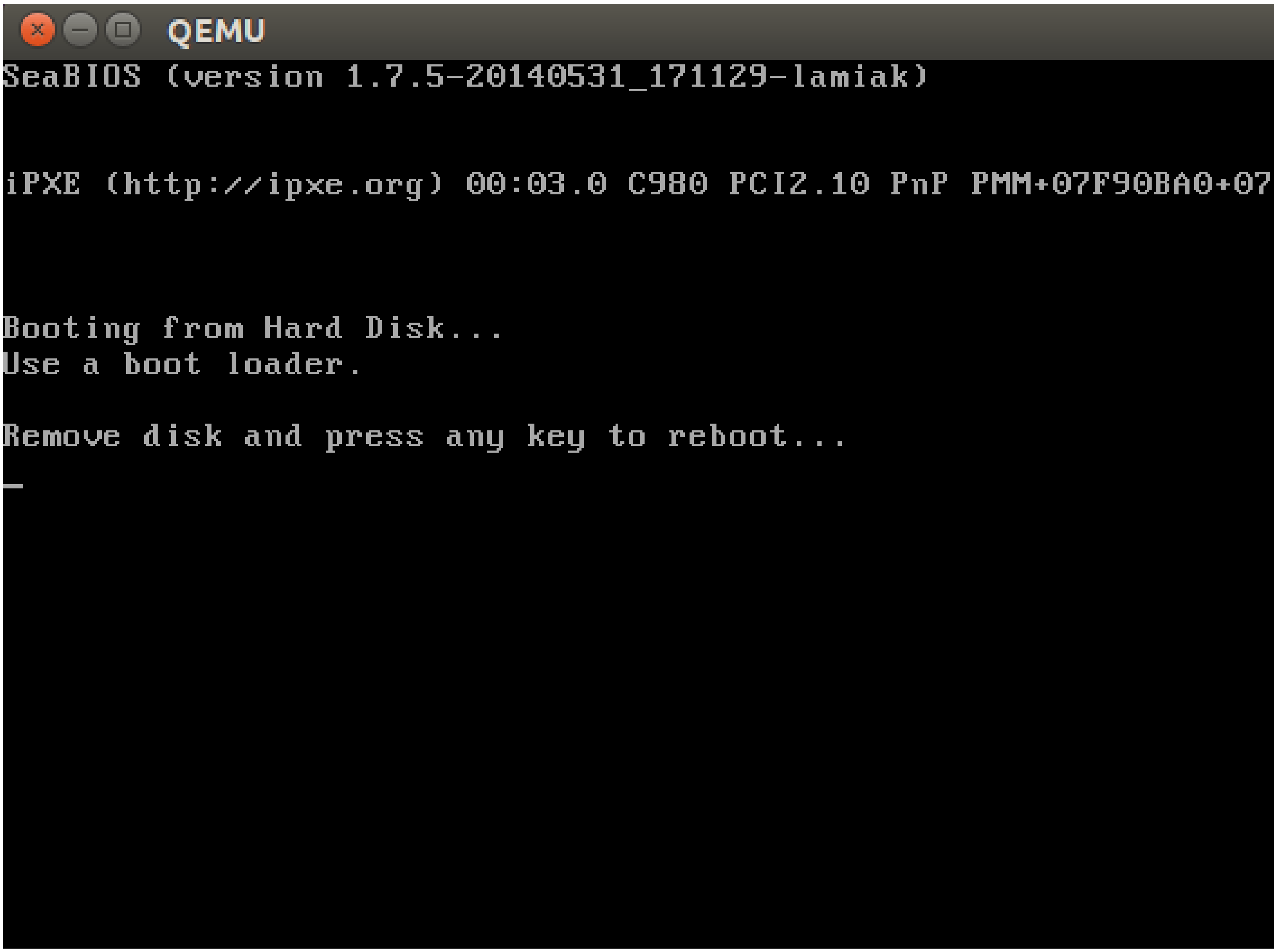

Figure 3: Try vmlinuz in qemu

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

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`) that jumps to the `start_of_setup-1f` point. In `Nf` notation, `2f`, for example, refers to the local label `2:`. In our case, it's label `1:` that is present right after the jump, and 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.

This is the first code that actually runs (aside from the previous jump instructions, of course). After the kernel setup part receives control from the bootloader, the first `jmp` instruction is located at the `0x200` offset from the start of the kernel real mode, i.e., after the first 512 bytes. This can be seen in both the Linux kernel boot protocol and the GRUB 2 source code:

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

In my case, the kernel is loaded at the physical address `0x10000`. This means that segment registers have the following values after kernel setup starts:

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

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

- Make sure that all segment register values are equal
- Set up a correct stack, if needed
- Set up bss
- Jump to the C code in arch/x86/boot/main.c

Let's look at the implementation.

2.5 Aligning the Segment Registers

First of all, the kernel ensures that the `ds` and `es` segment registers point to the same address. Next, it clears the direction flag using the `cld` instruction:

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

As I wrote earlier, `grub2` loads kernel setup code at address `0x10000` by default and `cs` at `0x1020` because execution doesn't start from the start of the file, but from the jump here:

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

which is at a `512` byte offset from 4d 5a. We also need to align `cs` from `0x1020` to `0x1000`, as well as all other segment registers. After that, we set up the stack:

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

which pushes the value of `ds` to the stack, followed by the address of the 6 label and executes the `lretw` instruction. When the `lretw` instruction is called, it loads the address of label `6` into the instruction pointer register and loads `cs` with the value of `ds`. Afterward, `ds` and `cs` will have the same values.

2.6 Stack Setup

Almost all of the setup code is for preparing the C language environment in real mode. The next step is checking the `ss` register's value and setting up 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 a valid value `0x1000` (as do all the other segment registers besides `cs`)
- `ss` is invalid and the `CAN_USE_HEAP` flag is set (see below)
- `ss` is invalid and the `CAN_USE_HEAP` flag is not set (see below)

Let's look at all three of these scenarios in turn:

- `ss` has a correct address (`0x1000`). In this case, we go to label 2:

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

Here we set the alignment of `dx` (which contains the value of `sp` as given by the bootloader) to `4` bytes and check if it is zero. If it is, we set `dx` to `0xfffc` (The last 4-byte aligned address in a 64KB segment). If it is not zero, we continue to use the value of `sp` given by the bootloader (`0xf7f4` in my case). Afterwards, we put the value of `ax` (`0x1000`) into `ss`. We now have a correct stack:

- The second scenario, (`ss` != `ds`). First, we put the value of _end (the address of the end of the setup code) into `dx` and check the `loadflags` header field using the `testb` instruction to see whether we can use the heap. loadflags is a bitmask header 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:

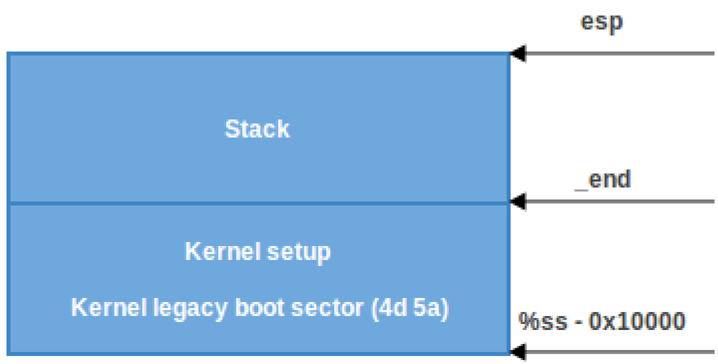

Figure 4: stack

```
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, we put `heap_end_ptr` into `dx` (which points to `_end`) and add `STACK_SIZE` (the minimum stack size, `1024` bytes) to it. After this, if `dx` is not carried (it will not be carried, `dx = _end + 1024`), 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`:

2.7 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:

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

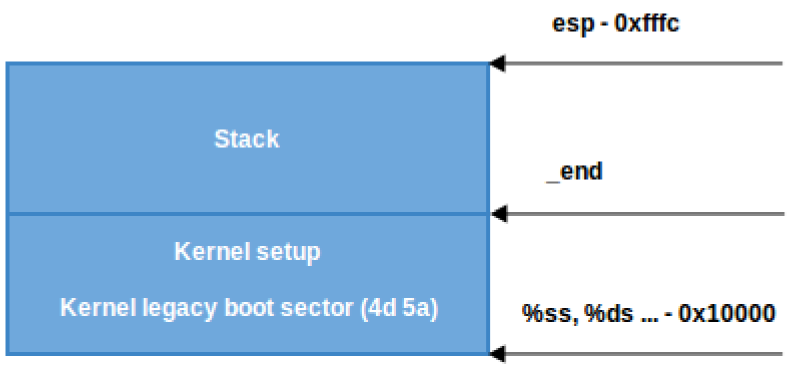

Figure 5: stack

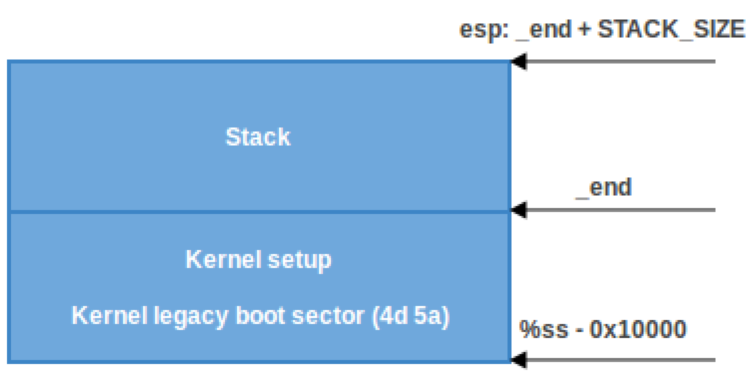

Figure 6: minimal stack

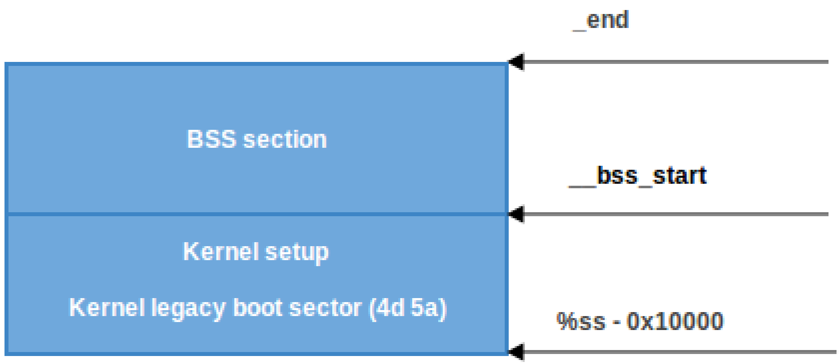

Figure 7: bss

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 zeroed using the following code:

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

First, the __bss_start address is moved into `di`. Next, the `_end + 3` address (+3 - aligns to 4 bytes) is moved into `cx`. The `eax` register is cleared (using the `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 used repeatedly, storing the value of `eax` (zero) into the address pointed to by `di`, automatically increasing `di` by four, repeating 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`:

2.8 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.

2.9 Conclusion

This is the end of the first part about Linux kernel insides. If you have questions or suggestions, ping me on Twitter 0xAX, drop me an email, or just create an issue. In the next part, we will see the first C code that executes in the Linux kernel setup, the implementation of memory routines such as `memset`, `memcpy`, `earlyprintk`, early console implementation and initialization, and much 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.

2.10 Links

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

3 Kernel booting process. Part 2.

3.1 First steps in the kernel setup

We started to dive into the linux kernel's 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 go over * what `protected mode` is, * the transition into it, * the initialization of the heap and the console, * memory detection, CPU validation and keyboard initialization * and much much more.

So, let's go ahead.

3.2 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 are only 220 bytes or 1 Megabyte, sometimes even only 640 Kilobytes of RAM available in 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 the 1 Megabyte in 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 talk about 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 Base * 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 the *Segment Descriptor*. These segment descriptors are stored in a data structure called the `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 how the GDT is loaded in the Linux kernel code. There will be an operation for loading it from memory, something like:

```
lgdt gdt
```

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

- the size(16-bit) of the global descriptor table;
- the 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:

```
 63         56         51   48    45           39        32
------------------------------------------------------------
|             | |B| |A|       | |   | |0|E|W|A|            |
| BASE 31:24  |G|/|L|V| LIMIT |P|DPL|S|  TYPE | BASE 23:16 |
|             | |D| |L| 19:16 | |   | |1|C|R|A|            |
------------------------------------------------------------

 31                         16 15                         0
------------------------------------------------------------
|                             |                            |
|       BASE 15:0             |       LIMIT 15:0           |
|                             |                            |
------------------------------------------------------------
```

Don't worry, I know it looks a little scary after Real mode, but it's easy. For example LIMIT 15:0 means that bits 0-15 of the segment limit are located at the beginning of the Descriptor. The rest of it is in LIMIT 19:16, which is located at bits 48-51 of the Descriptor. 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 split between bits 0-15 and 48-51. It defines the `length_of_segment - 1`. It depends on the `G`(Granularity) bit.

- if `G` (bit 55) is 0 and the segment limit is 0, the size of the segment is 1 Byte

- if **G** is 1 and the segment limit is 0, the size of the segment is 4096 Bytes
- if **G** is 0 and the segment limit is 0xfffff, the size of the segment is 1 Megabyte
- if **G** is 1 and the segment limit is 0xfffff, the size of the segment is 4 Gigabytes

So, what this means is * 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 232 = 4 Gigabytes.

2. Base[32-bits] is split between bits 16-31, 32-39 and 56-63. It defines the physical address of the segment's starting location.

3. Type/Attribute[5-bits] is represented by bits 40-44. It defines the type of segment and how it can be accessed.

- The **S** flag at bit 44 specifies the 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, access
| 13           1    1    1   0 | Code             | Execute/Read, conforming
| 15           1    1    1   1 | Code             | Execute/Read, conforming, access
--------------------------------------------------------------------------------
```

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) are either `EWA`(*E*xpansion *W*ritable *A*ccessible) or CRA(*C*onforming *R*eadable *A*ccessible). * if E(bit 42) is 0, expand up, otherwise, expand down. Read more here. * if W(bit 41)(for Data Segments) is 1, write access is allowed, and if it is 0, the segment is read-only. Note that read access is always allowed on data segments. * A(bit 40) controls whether the segment can be accessed by the processor or not. * C(bit 43) is the 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) controls read access to code segments; when it is 1, the segment can be read from. Write access is never granted for code segments.

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

5. The 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 from this segment.

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

7. The L flag(bit 53) indicates whether a code segment contains native 64-bit code. If it is set, then the code segment executes in 64-bit mode.

8. The D/B flag(bit 54) (Default/Big flag) represents the operand size i.e 16/32 bits. If set, operand size is 32 bits. Otherwise, it is 16 bits.

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:

```
 15              3 2  1     0
-----------------------------
```

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

Where, * **Index** stores the index number of the descriptor in the GDT. * **TI**(Table Indicator) indicates where to search for the descriptor. If it is 0 then the descriptor is searched for in the Global Descriptor Table(GDT). Otherwise, it will be searched for in the Local Descriptor Table(LDT). * And **RPL** contains the Requester's Privilege Level.

Every segment register has a visible and a hidden part. * Visible - The Segment Selector is stored here. * Hidden - The Segment Descriptor (which contains the base, limit, attributes & flags) is stored here.

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

- The segment selector must be loaded in one of the segment registers.
- The CPU tries to find a segment descriptor at the offset `GDT address + Index` from the selector and then loads the descriptor into the *hidden* part of the segment register.
- If paging is disabled, the linear address of the segment, or its physical address, is given by the formula: Base address (found in the descriptor obtained in the previous step) + Offset.

Schematically it will look like this:

The algorithm for the transition from real mode into protected mode is:

- Disable interrupts
- Describe and load the GDT with the `lgdt` instruction
- Set the 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.

3.3 Copying boot parameters into the "zeropage"

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

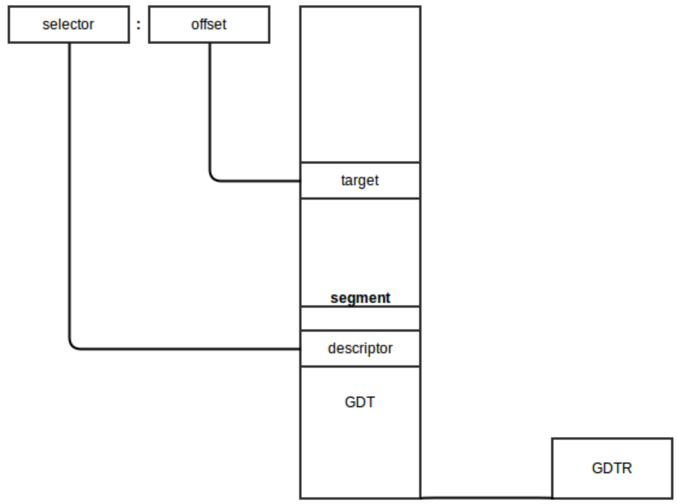

Figure 8: linear address

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

1. It copies **hdr** from header.S to the **setup_header** field in **boot_params** structure.
2. It updates the pointer to the kernel command line if the kernel was loaded with the old command line protocol.

Note that it copies **hdr** with the **memcpy** function, 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 the `globl` directive and its label. `ENDPROC` is described in include/linux/linkage.h and marks the `name` symbol as a function name and ends with the size of the `name` symbol.

The implementation of `memcpy` is simple. 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`. As we can see in the `REALMODE_CFLAGS` in `arch/x86/Makefile`, the kernel build system uses the `-mregparm=3` option of GCC, so functions get the first three parameters from `ax`, `dx` and `cx` registers. Calling `memcpy` looks like this:

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

So, * `ax` will contain the address of `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 `di` and saves `cx` on the stack. After this it shifts the value right 2 times (or divides it by 4) and copies four bytes from the address at `si` to the address at `di`. After this, we restore the size of `hdr` again, align it by 4 bytes and copy the rest of the bytes from the address at `si` to the address at `di` byte by byte (if there is more). Now the values of `si` and `di` are restored from the stack and the copying operation is finished.

3.4 Console initialization

After `hdr` is copied into `boot_params.hdr`, the next step is to initialize the console by calling the `console_init` function, 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. The value of the `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 prints 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 implementation of memset:

```
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 same calling conventions as the **memcpy** function, which means that the function gets its parameters from the **ax**, **dx** and **cx** registers.

The implementation of **memset** is similar to that of memcpy. It saves the value of the **di** register on the stack and puts the value of**ax**, which stores the address of the **biosregs** structure, into **di** . Next is the **movzbl** instruction, which copies the value of **dl** to the lowermost byte of the **eax** register. The remaining 3 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, if we need to fill a structure whose size is 4 bytes with the value **0x7** with memset, **eax** will contain the **0x00000007**. So if we multiply **eax** with **0x01010101**, we will get **0x07070707** and now we can copy these 4 bytes into the structure. **memset** uses the **rep; stosl** instruction to copy **eax** into **es:di**.

The rest of the `memset` function does almost the same thing 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.

3.5 Heap initialization

After the stack and bss section have been 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` structure 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 what it is used for, how to use it and how it is implemented in the next posts.

3.6 CPU validation

The next step as we can see is cpu validation through the `validate_cpu` function from arch/x86/boot/cpu.c source code file.

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;
}
```

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

at the next step, we may see a call to the `set_bios_mode` function after setup code found that a CPU is suitable. As we may see, this function is implemented only for the `x86_64` mode:

```
static void set_bios_mode(void)
{
#ifdef CONFIG_X86_64
    struct biosregs ireg;

    initregs(&ireg);
    ireg.ax = 0xec00;
    ireg.bx = 2;
    intcall(0x15, &ireg, NULL);
#endif
}
```

The `set_bios_mode` function executes the `0x15` BIOS interrupt to tell the BIOS that long mode (if `bx == 2`) will be used.

3.7 Memory detection

The next step is memory detection through 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 the **0xE820** interface here.

Let's look at the implementation of the `detect_memory_e820` function 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` contains the size of the buffer which will contain data about the 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 with a call to the `0x15` BIOS interrupt, which writes one line from the address allocation table. For getting the next line we need to call this interrupt again (which we do in the loop). Before the next call `ebx` must contain the value returned previously:

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

Ultimately, this function collects data from the address allocation table and writes this data into the `e820_entry` array:

- start of memory segment
- size of memory segment
- type of memory segment (whether the particular segment is usable or reserved)

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

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

3.8 Keyboard initialization

The next step is the initialization of the keyboard with a call to the `keyboard_init` function. At first `keyboard_init` initializes registers using the `initregs` function. It then calls the 0x16 interrupt to query the status of the keyboard.

```
initregs(&ireg);
ireg.ah = 0x02;     /* Get keyboard status */
```

```
    intcall(0x16, &ireg, &oreg);
    boot_params.kbd_status = oreg.al;
```

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

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

3.9 Querying

The next couple of steps are queries for different parameters. We will not dive into details about these queries but we will get back to them in later parts. Let's take a short look at these functions:

The first step is getting Intel SpeedStep information by calling the `query_ist` function. It checks the CPU level and if it is correct, calls `0x15` to get the info and saves the result to `boot_params`.

Next, the 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 `0x15` finishes executing, the `query_apm_bios` functions check the `PM` signature (it must be `0x504d`), the carry flag (it must be 0 if `APM` supported) and the value of the `cx` register (if it's 0x02, the protected mode interface is supported).

Next, it calls `0x15` again, but with `ax = 0x5304` to disconnect the `APM` interface and connect 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 the `CONFIG_APM` or `CONFIG_APM_MODULE` compile time flag 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 at how `query_edd` is implemented.

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 the `EDD_MBR_SIG_MAX` macro is 16. It collects data into an 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.

3.10 Conclusion

This is the end of the second part about the insides of the Linux kernel. In the next part, we will see video mode setting and the rest of the preparations before the 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.

3.11 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

4 Kernel booting process. Part 3.

4.1 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 to the `set_video` routine from main.c.

In this part, we will look at:

- video mode initialization in the kernel setup code,
- the preparations made before switching into protected mode,
- the 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  Proto   Name        Meaning
/Size
01FA/2  ALL     vid_mode    Video mode control
```

As we can read from the linux kernel boot protocol:

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

So we can add the `vga` option to the grub (or another bootloader's) 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 `0xFFFD` or `ask`. If you pass `ask` to `vga`, you will see a menu like this:

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.

4.2 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 the source code of the kernel, you'll see these very often and so it will be good to remember them.

4.3 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 arch/x86/boot/boot.h header file.

This macro 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 macros and functions for managing the heap which are defined in `arch/x86/boot/boot.h` header file.

They are:

```
#define RESET_HEAP()
```

As we saw just above, it resets the heap by setting the `HEAP` variable to `_end`, where `_end` is just `extern char _end[];`

Next is the `GET_HEAP` macro:

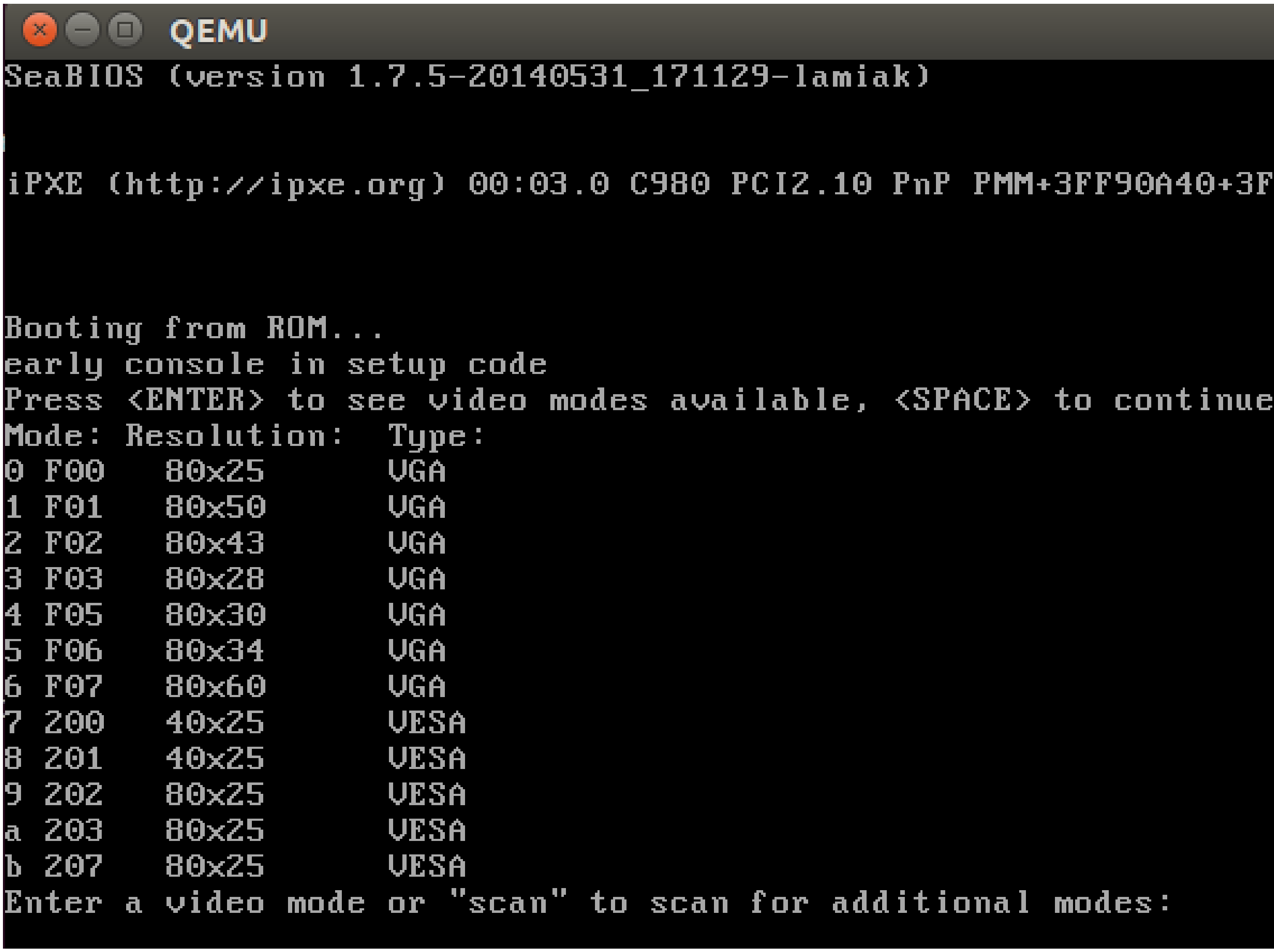

Figure 9: video mode setup menu

```
#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:

- the size of the datatype to be allocated for
- `__alignof__(type)` specifies how variables of this type are to be aligned
- `n` specifies how many items to allocate

The 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 we will further 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 assigned the address of the 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` pointer from the `heap_end` (we calculated it in the previous part) and returns 1 if there is enough memory available for `n`.

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

4.4 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 header file.

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

First of all, `store_cursor_position` initializes two variables which have type `biosregs` with `AH = 0x3`, and calls the `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 of 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, `store_mode_params` 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, the font size needs to be stored in `boot_params.screen_info.orig_video_po` 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 arch/x86/boot/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 the contents of the screen to the heap. This function collects all the data which we got in the previous functions (like the rows and columns, and stuff) 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 source code file. It goes over all video_cards and collects the number of modes provided by the cards. Here is the interesting part, 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. The answer is simple: every video mode presented in the x86 kernel setup code has a definition that looks 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 the `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 script, where we can find:

```
    .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 a loop to go over all of it. After `probe_cards` executes we have a bunch of structures like `static __videocard video_vga` with the `nmodes` (the number of video modes) filled in.

After the `probe_cards` function is done, we move to the main loop in the `set_video` function. There is an infinite loop which tries to set up the video mode with the `set_mode` function or prints a menu if we passed `vid_mode=ask` to the kernel command line or if 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 this value 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 selected card's `set_mode` function, 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 the above example of the `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 the video mode, we pass it to `boot_params.hdr.vid_mode`.

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

Having done this, the video mode setup is complete and now we can switch to the protected mode.

4.5 Last preparation before transition into protected mode

We can see the last function call - `go_to_protected_mode` - in arch/x86/boot/main.c. As the comment says: `Do the last things and invoke protected mode`, so let's see what these last things are and switch into protected mode.

The `go_to_protected_mode` function 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 it does 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 the `realmode_switch` hook (it isn't present for me) is checked, Non-Maskable Interrupts(NMI) is disabled:

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

At first, there is an inline assembly statement 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 such a 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 back to the interrupted instruction. Non-maskable interrupts (NMI) are interrupts which are always processed, independently of permission. They cannot be ignored and are typically used to signal for non-recoverable hardware errors. We will not dive into the details of interrupts now but we will be discussing them in the coming posts.

Let's get back to the code. We can see in the second line that we are writing the byte `0x80` (disabled bit) to `0x70` (the 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));
}
```

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

The next function is `enable_a20`, which enables the A20 line. This function is defined in arch/x86/boot/a2 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 = rdfs32(A20_TEST_ADDR);

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

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

First of all, we put `0x0000` in the `FS` register and `0xffff` in the `GS` register. Next, we read the value at the address `A20_TEST_ADDR` (it is `0x200`) and put this value into the variables `saved` and `ctr`.

Next, we write an updated `ctr` value into `fs:A20_TEST_ADDR` or `fs:0x200` with the `wrfs32` function, then delay for 1ms, and then read the value from the `GS` register into the address `A20_TEST_ADDR+0x10`. In a case when `a20` line is disabled, the address will be overlapped, in other case if it's not zero `a20` line is already enabled the A20 line.

If A20 is disabled, we try to enable it with a different method which you can find in `a20.c`. For example, it can be done with a call to the `0x15` BIOS interrupt with `AH=0x2041`.

If the `enable_a20` function finished with a failure, print an error message and call the 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.

4.6 Set up the Interrupt Descriptor Table

Now we set up the Interrupt Descriptor table (IDT) in the `setup_idt` function:

```
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 the IDT with the `lidtl` instruction. `null_idt` contains the address and size of the IDT, but for 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).

4.7 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 the GDT (you can read about it in the post 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, 0xfffff),
    [GDT_ENTRY_BOOT_DS] = GDT_ENTRY(0xc093, 0, 0xfffff),
    [GDT_ENTRY_BOOT_TSS] = GDT_ENTRY(0x0089, 4096, 103),
};
```

for code, data and TSS (Task State Segment). We will not use 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 the 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 in size, but an `aligned` structure will need 16 bytes to store in memory:

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

The `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 the 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, limit and builds a GDT entry. For example, let's look at the code segment entry. `GDT_ENTRY` takes the following values:

- base - 0
- limit - 0xfffff

- flags - 0xc09b

What does this mean? The segment's base address is 0, and the limit (size of segment) is - `0xfffff` (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 real mode now).

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

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

4.8 Actual transition into protected mode

This is the end of the `go_to_protected_mode` function. We loaded the IDT and GDT, disabled interrupts 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 the 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 the second one is in `edx`.

First of all, we put the address of `boot_params` in the `esi` register and the address of the code segment register `cs` in `bx`. After this, we shift `bx` by 4 bits and add it to the memory location labeled `2` (which is `(cs << 4) + in_pm32`, the physical address to jump after transitioned to 32-bit mode) and jump to label `1`. So after this `in_pm32` in label `2` will be overwritten with `(cs << 4) + in_pm32`.

Next we put the data segment and the task state segment in the `cx` 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 under protect mode, which has value `(cs << 4) + in_pm32` derived from real mode
- `__BOOT_CS` is the code segment we want to jump to.

After this we are finally in protected mode:

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

Let's look at the first steps taken 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`).

And setup a valid stack for debugging purposes:

```
addl    %ebx, %esp
```

The last step before the jump into 32-bit entry point is to clear the general purpose registers:

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

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

```
jmpl    *%eax
```

Remember that `eax` contains the address of the 32-bit entry (we passed it as the first parameter into `protected_mode_jump`).

That's all. We're in protected mode and stop at its entry point. We will see what happens next in the next part.

4.9 Conclusion

This is the end of the third part about linux kernel insides. In the next part, we will look at the first steps we take in protected mode and transition into 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.

4.10 Links

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

5 Kernel booting process. Part 4.

5.1 The Transition to 64-bit mode

This is the fourth part of the `Kernel booting process`. Here, we will learn about the first steps taken in protected mode, like checking if the CPU supports long mode and SSE. We will initialize the page tables with paging and, at the end, transition the CPU to long mode.

NOTE: there will be lots of assembly code in this part, so if you are not familiar with that, you might want to consult 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:

```
jmpl    *%eax
```

You will recall that the `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 this is so 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 the `cs` register contains a value of `0x10` (as you maight recall from the previous part, this is the second index in the `Global Descriptor Table`), the `eip` register contains the value `0x100000` and the base address of all segments including the code segment are zero.

So, the physical address where the kernel is loaded would be `0:0x100000` or just `0x100000`, as specified by the boot protocol. Now let's start with the `32-bit` entry point.

5.2 The 32-bit entry point

The `32-bit` entry point is defined in the arch/x86/boot/compressed/head_64.S assembly source code file:

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

First, why is the directory named `compressed`? The answer to that is that `bzimage` is a gzipped package consisting of `vmlinux`, `header` and `kernel setup code`. We looked at kernel setup code in all of the previous parts. The main goal of the code in `head_64.S` is to prepare to enter long mode, enter it and then decompress the kernel. We will look at all of the steps leading to kernel decompression in this part.

You will find two files in the `arch/x86/boot/compressed` directory:

- head_32.S
- head_64.S

but we will consider only the `head_64.S` source code file because, as you may remember, this book is only `x86_64` related; Let's look at arch/x86/boot/compressed/Makefile. We can find the following `make` target here:

```
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
```

The first line contains this- `$(obj)/head_$(BITS).o`.

This means that we will select which file to link based on what `$(BITS)` is set to, either `head_32.o` or `head_64.o`. The `$(BITS)` variable is defined elsewhere in arch/x86/Makefile based on the kernel configuration:

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

Now that we know where to start, let's get to it.

5.3 Reload the segments if needed

As indicated above, we start in the arch/x86/boot/compressed/head_64.S assembly source code file. We first see the definition of a special section attribute before the definition of the `startup_32` function:

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

`__HEAD` is a macro defined in the include/linux/init.h header file and expands to the definition of the following section:

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

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

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

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

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

Now that we have our bearings, let's look at the contents of 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 the direction flag is clear, all string operations like stos, scas and others will increment the index registers `esi` or `edi`. We need to clear the direction flag because later we will use strings operations to perform various operations such as clearing space for page tables.

After we have cleared the `DF` bit, the next step is to check the `KEEP_SEGMENTS` flag in the `loadflags` kernel setup header field. If you remember, we already talked about `loadflags` in the very first part of this book. There we checked the `CAN_USE_HEAP` flag to query the ability to use the 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 `loadflags`, we need to set the `ds`, `ss` and `es` segment registers to the index of the data segment with a base of `0`. That we do:

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

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

Remember that `__BOOT_DS` is `0x18` (the index of the 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 they are not set. This is all pretty easy, but here's something to consider. 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 the values in the segment registers again? The answer is easy. The Linux kernel also has a 32-bit boot protocol and if a bootloader uses *that* to load the Linux kernel, all the code before the `startup_32` function will be missed. In this case, the `startup_32` function would be the first entry point to the Linux kernel right after the bootloader and there are no guarantees that the segment registers will be in a known state.

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

```
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` function is `0` but that isn't so. We now need to know where we actually are. This is pretty simple to do in long mode because it supports `rip` relative addressing, but currently we are in protected mode. We will use a common pattern to find the address of the `startup_32` function. We need to define a label, make a call to it and pop the top of the stack to a register:

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

After this, the register indicated by `%reg` will contain the address of `label`. Let's look at the code which uses this pattern to search for the `startup_32` function 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 an offset of `0x1e4`. This four byte field is a temporary stack for the `call` instruction. We set `esp` to the address four bytes after the `BP_scratch` field of the `boot_params` structure. 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 the top to bottom in the `x86_64` architecture. So our stack pointer will point to the top of the temporary stack. Next, we can see the pattern that I've described above. We make a call to the **1f** label and pop the top of the stack onto **ebp**. This works because **call** stores the return address of the current function on the top of the stack. We now have the address of the **1f** label and can now easily get the address of the **startup_32** function. We just need to subtract the address of the label from the address we got from the stack:

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

The **startup_32** function is linked to run at the address **0x0** and this means that **1f** has the address **0x0 + offset to 1f**, which is approximately **0x21** bytes. The **ebp** register contains the real physical address of the **1f** label. So, if we subtract **1f** from the **ebp** register, we will get the real physical address of the **startup_32** function. The Linux kernel boot protocol saysthe base of the protected mode kernel is **0x100000**. We can verify this with gdb. Let's start the debugger and add a breakpoint at the address of **1f**, which is **0x100021**. If this is correct we will see the value **0x100021** in the **ebp** register:

```
$ gdb
(gdb)$ target remote :1234
Remote debugging using :1234
0x0000fff0 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 execute the next instruction, `subl $1b, %ebp`, we will see:

```
(gdb) nexti
...
...
...
ebp            0x100000 0x100000
...
...
...
```

Ok, we've verified that the address of the `startup_32` function is `0x100000`. After we know the address of the `startup_32` label, we can 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.

5.4 Stack setup and CPU verification

We can't set up the stack until we know where in memory the `startup_32` label is. If we imagine the stack as an array, the stack pointer register `esp` must point to the end of it. Of course, we can define an array in our code, but we need to know its actual address to configure the stack pointer correctly. Let's look at the code:

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

The `boot_stack_end` label is also defined in the arch/x86/boot/compressed/head_64.S assembly source code file and is 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, so the `eax` register contains the address of `boot_stack_end` as it was linked, which is `0x0 + boot_stack_end`. To get the real address of `boot_stack_end`, we need to add the real address of the `startup_32` function. We've already found this address and put it into the `ebp` register. In the end, the `eax` register will contain the real address of `boot_stack_end` and we just need to set the stack pointer to it.

After we have set up the stack, the next step is CPU verification. Since we are transitioning to `long mode`, we need to check that the CPU supports `long mode` and `SSE`. We will do this with a call to the `verify_cpu` function:

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

This function is 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 to get information about the processor. In our case, it checks for `long mode` and `SSE` support and sets the `eax` register to `0` on success and `1` on failure.

If the value of `eax` is not zero, we jump to the `no_longmode` label which just stops the CPU with the `hlt` instruction while no hardware interrupt can happen:

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

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

5.5 Calculate the relocation address

The next step is to calculate the 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. The default base address of the Linux kernel is determined by the value of the `CONFIG_PHYSICAL_START` kernel configuration option. Its default value is `0x1000000` or `16 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 a 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.
```

Now that we know where to start, let's get to it.

5.6 Reload the segments if needed

As indicated above, we start in the arch/x86/boot/compressed/head_64.S assembly source code file. We first see the definition of a special section attribute before the definition of the `startup_32` function:

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

`__HEAD` is a macro defined in the include/linux/init.h header file and expands to the definition of the following section:

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

Here, `.head.text` is the name of the section and `ax` is a set of flags. In our case, these flags show us that this section is [executable](https://en.wikipedia.org/wiki/Executable

In simple terms, this means that a Linux kernel with this option set can be booted from different addresses. Technically, this is done by compiling the decompressor as position independent code. If we look at arch/x86/boot/compressed/Makefile, we can 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 is obtained by adding the address field of the instruction to the value of the program counter. We can load code which 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. The calculation of this address depends on the `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
```

Remember that the 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 in the `ebx` register, align it to a multiple of `2MB` and compare it with the result of the `LOAD_PHYSICAL_ADDR` macro. `LOAD_PHYSICAL_ADDR` is 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 the physical address where the kernel will be loaded. After comparing `LOAD_PHYSICAL_ADDR` and the value of the `ebx` register, we add the offset from `startup_32` where we will decompress the compressed kernel image. If the `CONFIG_RELOCATABLE` option is not enabled during kernel configuration, we just add `z_extract_offset` to the default address where the kernel is loaded.

After all of these calculations, `ebp` will contain the address where we loaded the kernel and `ebx` will contain the address where the decompressed kernel will be relocated. But that is not the end. The compressed kernel image should be moved to the end of the decompression buffer to simplify calculations regarding where the kernel will be located later. For this:

```
1:
    movl    BP_init_size(%esi), %eax
    subl    $_end, %eax
    addl    %eax, %ebx
```

we put the value from the `boot_params.BP_init_size` field (or the kernel setup header value from `hdr.init_size`) in the `eax` register. The `BP_init_size` field contains the larger of the compressed and uncompressed vmlinux sizes. Next we subtract the address of the `_end` symbol from this value and add the result of the subtraction to the `ebx` register which will store the base address for kernel decompression.

5.7 Preparation before entering long mode

After we get the address to relocate the compressed kernel image to, we need to do one last step before we can transition to 64-bit mode. First, we need to update the Global Descriptor Table with 64-bit segments because a relocatable kernel is runnable at any address below 512GB:

```
    addl    %ebp, gdt+2(%ebp)
    lgdt    gdt(%ebp)
```

Here we adjust the base address of the Global Descriptor table to the address where we actually loaded the kernel 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
gdt64:
	.word	gdt_end - gdt
	.long	0
	.word	0
	.quad	0
gdt:
	.word	gdt_end - gdt
	.long	gdt
	.word	0
	.quad	0x00cf9a000000ffff	/* __KERNEL32_CS */
	.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: the first is a `32-bit` descriptor for the kernel code segment, a `64-bit` kernel segment, a 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 we set descriptors to use `CS.L = 1` and `CS.D = 0` for execution in `64` bit mode. As we can see, the definition of the `gdt` starts with a two byte value: `gdt_end - gdt` which represents the address of the last byte in the `gdt` table or the table limit. The next four bytes contain the base address of the `gdt`.

After we have loaded the `Global Descriptor Table` with the `lgdt` instruction, we must enable PAE by putting the value of the `cr4` register into `eax`, setting the 5th bit and loading it back into `cr4`:

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

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

5.8 Long mode

Long mode is the native mode for x86_64 processors. First, let's look at some differences between `x86_64` and `x86`.

`64-bit` mode provides the following features:

- 8 new general purpose registers from `r8` to `r15`
- All general purpose registers are 64-bit now
- A 64-bit instruction pointer - `RIP`
- A new operating mode - Long mode;
- 64-Bit Addresses and Operands;
- RIP Relative Addressing (we will see an example of this in the coming parts).

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

- 64-bit mode;
- compatibility mode.

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

- Enable PAE;
- Build page tables and load the address of the top level page table into the `cr3` register;
- Enable `EFER.LME`;
- Enable paging.

We already enabled `PAE` by setting the `PAE` bit in the `cr4` control register. Our next goal is to build the structure for paging. We will discuss this in the next paragraph.

5.9 Early page table initialization

We already know that before we can move into `64-bit` mode, we need to build page tables. Let's look at how the early `4G` boot page tables are built.

NOTE: I will not describe the theory of virtual memory here. If you want to know more about virtual memory, check out the links at the end of this part.

The Linux kernel uses `4-level` paging, and we generally 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 a total of `2048` entries.

Let's look at how this is implemented. First, we clear the buffer for the page tables in memory. Every table is `4096` bytes, so we need clear a `24` kilobyte buffer:

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

We put the address of `pgtable` with an offset of `ebx` (remember that `ebx` points to the location in memory where the kernel will be decompressed later) into the `edi` register, clear the `eax` register and set the `ecx` register to `6144`.

The `rep stosl` instruction will write the value of `eax` to `edi`, add `4` to `edi` and decrement `ecx` by `1`. This operation will be repeated while the value of the `ecx` register is greater than zero. That's why we put `6144` or `BOOT_INIT_PGT_SIZE/4` in `ecx`.

`pgtable` is defined at the end of the arch/x86/boot/compressed/head_64.S assembly file:

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

As we can see, it is located in the `.pgtable` section and its size depends on the `CONFIG_X86_VERBOSE_BOOTUP` kernel configuration option:

```
#   ifdef CONFIG_X86_VERBOSE_BOOTUP
#    define BOOT_PGT_SIZE    (19*4096)
#   else /* !CONFIG_X86_VERBOSE_BOOTUP */
#    define BOOT_PGT_SIZE    (17*4096)
#   endif
# else /* !CONFIG_RANDOMIZE_BASE */
#   define BOOT_PGT_SIZE     BOOT_INIT_PGT_SIZE
# endif
```

After we have a 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 `pgtable` relative to `ebx` or in other words relative to address of `startup_32` in the `edi` register. Next, we put this address with an offset of `0x1007` into the `eax` register. `0x1007` is the result of adding the size of the `PML4` table which is `4096` or `0x1000` bytes with `7`. The `7` here represents the flags associated with the `PML4` entry. In our case, these flags are `PRESENT+RW+USER`. In the end, we just write the address of the first `PDP` entry to the `PML4` table.

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 set `edi` to the base address of the page directory pointer which is at an offset of `4096` or `0x1000` bytes from the `pgtable` table and `eax` to the address of the first page directory pointer entry. We also set `ecx` to `4` to act as 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 calculate the address of the following page directory pointer entries — each entry is `8` bytes — and write their addresses to `eax`. The last step in building the paging structure is to build 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 things that we did in the previous example, all entries are associated with these flags - `$0x00000183` - `PRESENT + WRITE + MBZ`. In the end, we will have a page table with `2048` `2-MByte` pages, which represents a 4 Gigabyte block of memory:

```
>>> 2048 * 0x00200000
4294967296
```

Since we've just finished building our early page table structure which maps `4` gigabytes of memory, we can put the address of the high-level page table - `PML4` - into the `cr3` control register:

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

That's all. We are now prepared to transition to long mode.

5.10 The transition to 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/asm/msr-index.h) in the `ecx` register and execute the `rdmsr` instruction which reads the MSR register. After `rdmsr` executes, the resulting data is stored in `edx:eax` according to the `MSR` register specified in `ecx`. We check the `EFER_LME` bit with the `btsl` instruction and write data from `edx:eax` back 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 `eax` to the stack and enable paging by setting the `PG` and `PE` bits in the `cr0` register:

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

We then execute the `lret` instruction:

```
lret
```

Remember that we pushed the address of the `startup_64` function to the stack in the previous step. The CPU extracts `startup_64`'s address from the stack and jumps there.

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

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

That's all!

5.11 Conclusion

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

In the next part, we will learn about many things, including how kernel decompression works.

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

5.12 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

6 Kernel booting process. Part 5.

6.1 Kernel Decompression

This is the fifth part of the `Kernel booting process` series. We went over the transition to 64-bit mode in the previous part and we will continue where we left off in this part. We will study the steps taken to prepare for kernel decompression, relocation and the process of kernel decompression itself. So… let's dive into the kernel code again.

6.2 Preparing to Decompress the Kernel

We stopped right before the jump to the `64-bit` entry point - `startup_64` which is located in the arch/x86/boot/compressed/head_64.S source code file. We already covered the jump to `startup_64` from `startup_32` in the previous part:

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

Since we have loaded a new `Global Descriptor Table` and the CPU has transitioned to a new mode (`64-bit` mode in our case), we set up the segment registers again at the beginning of the `startup_64` function:

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

All segment registers besides the `cs` register are now reset in `long mode`.

The next step is to compute the difference between the location the kernel was compiled to be loaded at and the location where it is actually 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:
    movl    BP_init_size(%rsi), %ebx
    subl    $_end, %ebx
    addq    %rbp, %rbx
```

The `rbp` register contains the decompressed kernel's start address. After this code executes, the `rbx` register will contain the address where the kernel code will be relocated to for decompression. We've already done this before in the `startup_32` function (you can read about this in the previous part - Calculate relocation address), but we need to do this calculation again because the bootloader can use the 64-bit boot protocol now and `startup_32` is no longer being executed.

In the next step we set up the stack pointer, reset the flags register and set up the `GDT` again to overwrite the `32-bit` specific values with those from the `64-bit` protocol:

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

    leaq    gdt(%rip), %rax
    movq    %rax, gdt64+2(%rip)
    lgdt    gdt64(%rip)

    pushq   $0
    popfq
```

If you take a look at the code after the `lgdt gdt64(%rip)` instruction, you will see that there is some additional code. This code builds the trampoline to enable 5-level paging if needed. We will only consider 4-level paging in this book, so this code will be omitted.

As you can see above, the **rbx** register contains the start address of the kernel decompressor code and we just put this address with an offset of **boot_stack_end** in the **rsp** register which points to the top of the stack. After this step, the stack will be correct. You can find the definition of the **boot_stack_end** constant in the end of the 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 **.pgtable**. If you peek inside the arch/x86/boot/compressed/v
linker script, you will find the definitions of **.bss** and **.pgtable** there.

Since the stack is now correct, we can copy the compressed kernel to the address that we got above, when we calculated the relocation address of the decompressed kernel. Before we get into the details, let's take a 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
```

This set of instructions copies the compressed kernel over to where it will be decompressed.

First of all we push **rsi** to the stack. We need preserve the value of **rsi**, because this register now stores a pointer to **boot_params** which is a real mode structure that contains booting related data (remember, this structure was populated at the start of the kernel setup). We pop the pointer to **boot_params** back to **rsi** after we execute this code.

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

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

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

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

The `.text` section contains the decompression code:

```
    .text
relocated:
...
...
```

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

And `.rodata..compressed` contains the compressed kernel image. So `rsi` will contain the absolute address of `_bss - 8`, and `rdi` will contain the relocation relative address of `_bss - 8`. In the same way we store these addresses in registers, we put the address of `_bss` in the `rcx` register. As you can see in the `vmlinux.lds.S` linker script, it's located at the end of all sections with the setup/kernel code. Now we can start copying data from `rsi` to `rdi`, `8` bytes at a time, with the `movsq` instruction.

Note that we execute an `std` instruction before copying the data. This sets the `DF` flag, which means that `rsi` and `rdi` will be decremented. In other words, we will copy the bytes backwards. At the end, we clear the `DF` flag with the `cld` instruction, and restore the `boot_params` structure to `rsi`.

Now we have a pointer to the `.text` section's address after relocation, and we can jump to it:

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

6.3 The final touches before kernel decompression

In the previous paragraph we saw that the `.text` section starts with the `relocated` label. The first thing we do is to clear 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 we'll soon jump to C code. Here we just clear `eax`, put the addresses of `_bss` in `rdi` and `_ebss` in `rcx`, and fill `.bss` with zeros with the `rep stosq` instruction.

At the end, we can see a call to the `extract_kernel` function:

```
pushq   %rsi
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    extract_kernel
popq    %rsi
```

Like before, we push `rsi` onto the stack to preserve the pointer to `boot_params`. We also copy the contents of `rsi` to `rdi`. Then, we set `rsi` to point to the area where the kernel will be decompressed. The last step is to prepare the parameters for the `extract_kernel` function and call it to decompress the kernel. The `extract_kernel` function is defined in the arch/x86/boot/compressed/misc.c source code file and takes six arguments:

- `rmode` - a pointer to the boot_params structure which is filled by either the bootloader or during early kernel initialization;
- `heap` - a pointer to `boot_heap` which represents the start address of the early boot heap;
- `input_data` - a pointer to the start of the compressed kernel or in other words, a pointer to the `arch/x86/boot/compressed/vmlinux.bin.bz2` file;
- `input_len` - the size of the compressed kernel;
- `output` - the start address of the decompressed kernel;
- `output_len` - the size of the decompressed kernel;

All arguments will be passed through registers as per the System V Application Binary Interface. We've finished all the preparations and can now decompress the kernel.

6.4 Kernel decompression

As we saw in the previous paragraph, the `extract_kernel` function is defined in the arch/x86/boot/compre source code file and takes six arguments. This function starts with the video/console initialization that we already saw in the previous parts. We need to do this again because we don't know if we started in real mode or if a bootloader was used, or whether the bootloader used the `32` or `64-bit` boot protocol.

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;
```

Here, `heap` is the second parameter of the `extract_kernel` function as passed to it in arch/x86/boot/compressed/

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

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

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

where `BOOT_HEAP_SIZE` is a macro which expands to `0x10000` (`0x400000` in thecase of a `bzip2` kernel) and represents the size of the heap.

After we initialize the heap pointers, the next step is to call the `choose_random_location` function from the arch/x86/boot/compressed/kaslr.c source code file. As we can guess from the function name, it chooses a memory location to write the decompressed kernel to. It may look weird that we need to find or even `choose` where to decompress the compressed kernel image, but the Linux kernel supports kASLR which allows decompression of the kernel into a random address, for security reasons.

We'll take a look at how the kernel's load address is randomized in the next part.

Now let's get back to misc.c. After getting the address for the kernel image, we need to check that the random address we got is correctly aligned, and in general, not wrong:

```
if ((unsigned long)output & (MIN_KERNEL_ALIGN - 1))
    error("Destination physical address inappropriately aligned");

if (virt_addr & (MIN_KERNEL_ALIGN - 1))
    error("Destination virtual address inappropriately aligned");

if (heap > 0x3fffffffffffUL)
    error("Destination address too large");

if (virt_addr + max(output_len, kernel_total_size) > KERNEL_IMAGE_SIZE)
    error("Destination virtual address is beyond the kernel mapping area");

if ((unsigned long)output != LOAD_PHYSICAL_ADDR)
    error("Destination address does not match LOAD_PHYSICAL_ADDR");
```

```
if (virt_addr != LOAD_PHYSICAL_ADDR)
    error("Destination virtual address changed when not relocatable");
```

After all these checks we will see the familiar message:

```
Decompressing Linux...
```

Now, we call the `__decompress` function to decompress the kernel:

```
__decompress(input_data, input_len, NULL, NULL, output, output_len, NULL, error);
```

The implementation of 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 the kernel is decompressed, two more functions are called: `parse_elf` and `handle_relocations`. The main point of these functions is to move the decompressed kernel image to its correct place in memory. This is because the decompression is done in-place, and we still need to move the kernel to the correct address. As we already know, the kernel image is an ELF executable. The main goal of the `parse_elf` function is to move loadable segments to the correct address. We can see the kernel's loadable segments in the output of the `readelf` program:

```
readelf -l vmlinux

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

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000200000 0xffffffff81000000 0x0000000001000000
                 0x0000000000893000 0x0000000000893000  R E    200000
  LOAD           0x0000000000a93000 0xffffffff81893000 0x0000000001893000
                 0x000000000016d000 0x000000000016d000  RW     200000
  LOAD           0x0000000000c00000 0x0000000000000000 0x0000000001a00000
                 0x00000000000152d8 0x00000000000152d8  RW     200000
  LOAD           0x0000000000c16000 0xffffffff81a16000 0x0000000001a16000
                 0x0000000000138000 0x000000000029b000  RWE    200000
```

The goal of the `parse_elf` function is to load these segments to the `output` address we got from the `choose_random_location` function. This function starts by checking 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;
}
```

If the ELF header is not valid, it prints an error message and halts. If we have a valid `ELF` file, we go through all the program headers from the given `ELF` file and copy all loadable segments with correct 2 megabyte aligned addresses to the output buffer:

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

        switch (phdr->p_type) {
        case PT_LOAD:
#ifdef CONFIG_X86_64
            if ((phdr->p_align % 0x200000) != 0)
                error("Alignment of LOAD segment isn't multiple of 2MB");
#endif
#ifdef CONFIG_RELOCATABLE
            dest = output;
            dest += (phdr->p_paddr - LOAD_PHYSICAL_ADDR);
#else
            dest = (void *)(phdr->p_paddr);
#endif
            memmove(dest, output + phdr->p_offset, phdr->p_filesz);
            break;
        default:
            break;
        }
    }
```

That's all.

From this moment, all loadable segments are in the correct place.

The next step after the `parse_elf` function is to call the `handle_relocations` function. The implementation of this function depends on the `CONFIG_X86_NEED_RELOCS` kernel configuration option and if it is enabled, this function adjusts addresses in the kernel image. This function is also only called if the `CONFIG_RANDOMIZE_BASE` configuration option was enabled during kernel configuration. The implementation of the `handle_relocations` function is easy enough. This function subtracts the value of `LOAD_PHYSICAL_ADDR` from the value of the base load address of the kernel and thus we obtain the difference between where the kernel was linked to load and where it was actually loaded. After this we can relocate the kernel since we know the actual address where the kernel was loaded, the address where it was linked to run and the relocation table which is at the end of the kernel image.

After the kernel is relocated, we return from the `extract_kernel` function to arch/x86/boot/compressed/head_64

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!

6.5 Conclusion

This is the end of the fifth part about the linux kernel booting process. We will not see any more posts about the kernel booting process (there may be updates to this and previous posts though), but there will be many posts about other kernel internals.

The Next chapter will describe more advanced details about linux kernel booting process, like load address randomization and etc.

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.

6.6 Links

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

7 Kernel booting process. Part 6.

7.1 Introduction

This is the sixth part of the `Kernel booting process` series. In the previous part we took a look at the final stages of the Linux kernel boot process. But we have skipped some important, more advanced parts.

As you may remember, the entry point of the Linux kernel is the `start_kernel` function defined in the main.c source code file. This function is executed at the address stored in `LOAD_PHYSICAL_ADDR`. and depends on the `CONFIG_PHYSICAL_START` kernel configuration option, which is `0x1000000` by default:

```
config PHYSICAL_START
    hex "Physical address where the kernel is loaded" if (EXPERT || CRASH_DUMP)
    default "0x1000000"
    ---help---
      This gives the physical address where the kernel is loaded.
      ...
      ...
      ...
```

This value may be changed during kernel configuration, but the load address can also be configured to be a random value. For this purpose, the `CONFIG_RANDOMIZE_BASE` kernel configuration option should be enabled during kernel configuration.

Now, the physical address where the Linux kernel image will be decompressed and loaded will be randomized. This part considers the case when the `CONFIG_RANDOMIZE_BASE` option is enabled and the load address of the kernel image is randomized for security reasons.

7.2 Page Table Initialization

Before the kernel decompressor can look for a random memory range to decompress and load the kernel to, the identity mapped page tables should be initialized. If the bootloader used the 16-bit or 32-bit boot protocol, we already have page tables. But, there may be problems if the kernel decompressor selects a memory range which is valid only in a 64-bit context. That's why we need to build new identity mapped page tables.

Indeed, the first step in randomizing the kernel load address is to build new identity mapped page tables. But first, let's reflect on how we got to this point.

In the previous part, we followed the transition to long mode and jumped to the kernel decompressor entry point - the `extract_kernel` function. The randomization stuff begins with a call to this function:

```
void choose_random_location(unsigned long input,
                            unsigned long input_size,
                            unsigned long *output,
```

```
                                    unsigned long output_size,
                                    unsigned long *virt_addr)
{}
```

This function takes five parameters:

- `input`;
- `input_size`;
- `output`;
- `output_isze`;
- `virt_addr`.

Let's try to understand what these parameters are. The first parameter, `input` is just the `input_data` parameter of the `extract_kernel` function from the arch/x86/boot/compressed/misc.c source code file, cast to `unsigned long`:

```
asmlinkage __visible void *extract_kernel(void *rmode, memptr heap,
                                          unsigned char *input_data,
                                          unsigned long input_len,
                                          unsigned char *output,
                                          unsigned long output_len)
{
  ...
  ...
  ...
  choose_random_location((unsigned long)input_data, input_len,
                         (unsigned long *)&output,
                         max(output_len, kernel_total_size),
                         &virt_addr);
  ...
  ...
  ...
}
```

This parameter is passed through assembly from the arch/x86/boot/compressed/head_64.Ssource code file:

```
leaq    input_data(%rip), %rdx
```

`input_data` is generated by the little mkpiggy program. If you've tried compiling the Linux kernel yourself, you may find the output generated by this program in the `linux/arch/x86/boot/compres` source code file. In my case this file looks like this:

```
.section ".rodata..compressed","a",@progbits
.globl z_input_len
z_input_len = 6988196
.globl z_output_len
z_output_len = 29207032
.globl input_data, input_data_end
input_data:
.incbin "arch/x86/boot/compressed/vmlinux.bin.gz"
input_data_end:
```

As you can see, it contains four global symbols. The first two, `z_input_len` and `z_output_len` are the sizes of the compressed and uncompressed `vmlinux.bin.gz` archive. The third is our `input_data` parameter which points to the linux kernel image's raw binary (stripped of all debugging symbols, comments and relocation information). The last parameter, `input_data_end`, points to the end of the compressed linux image.

So, the first parameter to the `choose_random_location` function is the pointer to the compressed kernel image that is embedded into the `piggy.o` object file.

The second parameter of the `choose_random_location` function is `z_input_len`.

The third and fourth parameters of the `choose_random_location` function are the address of the decompressed kernel image and its length respectively. The decompressed kernel's address came from the arch/x86/boot/compressed/head_64.S source code file and is the address of the `startup_32` function aligned to a 2 megabyte boundary. The size of the decompressed kernel is given by `z_output_len` which, again, is found in `piggy.S`.

The last parameter of the `choose_random_location` function is the virtual address of the kernel load address. As can be seen, by default, it coincides with the default physical load address:

```
unsigned long virt_addr = LOAD_PHYSICAL_ADDR;
```

The physical load address is defined by the configuration options:

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

We've covered `choose_random_location`'s parameters, so let's look at its implementation. This function starts by checking the `nokaslr` option in the kernel command line:

```
if (cmdline_find_option_bool("nokaslr")) {
    warn("KASLR disabled: 'nokaslr' on cmdline.");
    return;
}
```

We exit `choose_random_location` if the option is specified, leaving the kernel load address unrandomized. Information related to this can be found in the kernel's 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.
```

Let's assume that we didn't pass `nokaslr` to the kernel command line and the `CONFIG_RANDOMIZE_BASE` kernel configuration option is enabled. In this case we add `kASLR` flag to kernel load flags:

```
boot_params->hdr.loadflags |= KASLR_FLAG;
```

Now, we call another function:

```
initialize_identity_maps();
```

The `initialize_identity_maps` function is defined in the arch/x86/boot/compressed/kaslr_64.c source code file. This function starts by initialising an instance of the `x86_mapping_info` structure called `mapping_info`:

```
mapping_info.alloc_pgt_page = alloc_pgt_page;
mapping_info.context = &pgt_data;
mapping_info.page_flag = __PAGE_KERNEL_LARGE_EXEC | sev_me_mask;
mapping_info.kernpg_flag = _KERNPG_TABLE;
```

The `x86_mapping_info` structure is defined in the arch/x86/include/asm/init.h header file and looks like this:

```
struct x86_mapping_info {
    void *(*alloc_pgt_page)(void *);
    void *context;
    unsigned long page_flag;
    unsigned long offset;
    bool direct_gbpages;
    unsigned long kernpg_flag;
};
```

This structure provides information about memory mappings. As you may remember from the previous part, we have already set up page tables to cover the range `0` to `4G`. This won't do since we might generate a randomized address outside of the 4 gigabyte range. So, the `initialize_identity_maps` function initializes the memory for a new page table entry. First, let's take a look at the definition of the `x86_mapping_info` structure.

`alloc_pgt_page` is a callback function that is called to allocate space for a page table entry. The `context` field is an instance of the `alloc_pgt_data` structure. We use it to track allocated page tables. The `page_flag` and `kernpg_flag` fields are page flags. The first represents flags for `PMD` or `PUD` entries. The `kernpg_flag` field represents overridable flags for kernel pages. The `direct_gbpages` field is used to check if huge pages are supported and the last field, `offset`, represents the offset between the kernel's virtual addresses and its physical addresses up to the `PMD` level.

The `alloc_pgt_page` callback just checks that there is space for a new page, allocates it in the `pgt_buf` field of the `alloc_pgt_data` structure and returns the address of the new page:

```
entry = pages->pgt_buf + pages->pgt_buf_offset;
pages->pgt_buf_offset += PAGE_SIZE;
```

Here's what the `alloc_pgt_data` structure looks like:

```
struct alloc_pgt_data {
    unsigned char *pgt_buf;
    unsigned long pgt_buf_size;
    unsigned long pgt_buf_offset;
};
```

The last goal of the `initialize_identity_maps` function is to initialize `pgdt_buf_size` and `pgt_buf_offset`. As we are only in the initialization phase, the `initialze_identity_maps` function sets `pgt_buf_offset` to zero:

```
pgt_data.pgt_buf_offset = 0;
```

`pgt_data.pgt_buf_size` will be set to `77824` or `69632` depending on which boot protocol was used by the bootloader (64-bit or 32-bit). The same is done for `pgt_data.pgt_buf`. If a bootloader loaded the kernel at `startup_32`, `pgdt_data.pgdt_buf` will point to the end of the already initialzed page table in the arch/x86/boot/compressed/head_64.S source code file:

```
pgt_data.pgt_buf = _pgtable + BOOT_INIT_PGT_SIZE;
```

Here, `_pgtable` points to the beginning of _pgtable. On the other hand, if the bootloader used the 64-bit boot protocol and loaded the kernel at `startup_64`, the early page tables should already be built by the bootloader itself and `_pgtable` will just point to those instead:

```
pgt_data.pgt_buf = _pgtable
```

As the buffer for new page tables is initialized, we may return to the `choose_random_location` function.

7.3 Avoiding Reserved Memory Ranges

After the stuff related to identity page tables is initilized, we can choose a random memory location to extract the kernel image to. But as you may have guessed, we can't just choose any address. There are certain reseved memory regions which are occupied by important things like the initrd and the kernel command line which must be avoided. The `mem_avoid_init` function will help us do this:

```
mem_avoid_init(input, input_size, *output);
```

All unsafe memory regions will be collected in an array called `mem_avoid`:

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

static struct mem_vector mem_avoid[MEM_AVOID_MAX];
```

Here, `MEM_AVOID_MAX` is from the `mem_avoid_index` enum which represents different types of reserved memory regions:

```
enum mem_avoid_index {
    MEM_AVOID_ZO_RANGE = 0,
    MEM_AVOID_INITRD,
    MEM_AVOID_CMDLINE,
    MEM_AVOID_BOOTPARAMS,
    MEM_AVOID_MEMMAP_BEGIN,
    MEM_AVOID_MEMMAP_END = MEM_AVOID_MEMMAP_BEGIN + MAX_MEMMAP_REGIONS - 1,
    MEM_AVOID_MAX,
};
```

Both are defined in the arch/x86/boot/compressed/kaslr.c source code file.

Let's look at the implementation of the `mem_avoid_init` function. The main goal of this function is to store information about reseved memory regions with descriptions given by the `mem_avoid_index` enum in the `mem_avoid` array and to create new pages for such regions in our new identity mapped buffer. The `mem_avoid_index` function does the same thing for all elements in the `mem_avoid_index`enum, so let's look at a typical example of the process:

```
mem_avoid[MEM_AVOID_ZO_RANGE].start = input;
mem_avoid[MEM_AVOID_ZO_RANGE].size = (output + init_size) - input;
add_identity_map(mem_avoid[MEM_AVOID_ZO_RANGE].start,
         mem_avoid[MEM_AVOID_ZO_RANGE].size);
```

THe `mem_avoid_init` function first tries to avoid memory regions currently used to decompress the kernel. We fill an entry from the `mem_avoid` array with the start address and the size of the relevant region and call the `add_identity_map` function, which builds the identity mapped pages for this region. The `add_identity_map` function is defined in the arch/x86/boot/compressed/kaslr_64.c source code file and looks like this:

```
void add_identity_map(unsigned long start, unsigned long size)
{
    unsigned long end = start + size;

    start = round_down(start, PMD_SIZE);
    end = round_up(end, PMD_SIZE);
    if (start >= end)
        return;

    kernel_ident_mapping_init(&mapping_info, (pgd_t *)top_level_pgt,
```

```
                    start, end);
}
```

The `round_up` and `round_down` functions are used to align the start and end addresses to a 2 megabyte boundary.

In the end this function calls the `kernel_ident_mapping_init` function from the arch/x86/mm/ident_map.c source code file and passes the previously initialized `mapping_info` instance, the address of the top level page table and the start and end addresses of the memory region for which a new identity mapping should be built.

The `kernel_ident_mapping_init` function sets default flags for new pages if they were not already set:

```
if (!info->kernpg_flag)
    info->kernpg_flag = _KERNPG_TABLE;
```

It then starts to build new 2-megabyte (because of the `PSE` bit in `mapping_info.page_flag`) page entries (`PGD -> P4D -> PUD -> PMD` if we're using five-level page tables or `PGD -> PUD -> PMD` if four-level page tables are used) associated with the given addresses.

```
for (; addr < end; addr = next) {
    p4d_t *p4d;

    next = (addr & PGDIR_MASK) + PGDIR_SIZE;
    if (next > end)
        next = end;

    p4d = (p4d_t *)info->alloc_pgt_page(info->context);
    result = ident_p4d_init(info, p4d, addr, next);

    return result;
}
```

The first thing this for loop does is to find the next entry of the `Page Global Directory` for the given address. If the entry's address is greater than the `end` of the given memory region, we set its size to `end`. After this, we allocate a new page with the `x86_mapping_info` callback that we looked at previously and call the `ident_p4d_init` function. The `ident_p4d_init` function will do the same thing, but for the lower level page directories (`p4d` -> `pud` -> `pmd`).

That's all.

We now have new page entries related to reserved addresses in our page tables. We haven't reached the end of the `mem_avoid_init` function, but the rest is similar. It builds pages for the initrd and the kernel command line, among other things.

Now we may return to the `choose_random_location` function.

7.4 Physical address randomization

After the reserved memory regions have been stored in the `mem_avoid` array and identity mapped pages are built for them, we select the region with the lowest available address to decompress the kernel to:

```
min_addr = min(*output, 512UL << 20);
```

You will notice that the address should be within the first `512` megabytes. A limit of `512` megabytes was selected to avoid unknown things in lower memory.

The next step is to select random physical and virtual addresses to load the kernel to. The first is the physical addresses:

```
random_addr = find_random_phys_addr(min_addr, output_size);
```

The `find_random_phys_addr` function is defined in the same source code file as `choose_random_loca`

```
static unsigned long find_random_phys_addr(unsigned long minimum,
                                           unsigned long image_size)
{
    minimum = ALIGN(minimum, CONFIG_PHYSICAL_ALIGN);

    if (process_efi_entries(minimum, image_size))
        return slots_fetch_random();

    process_e820_entries(minimum, image_size);
    return slots_fetch_random();
}
```

The main goal of the `process_efi_entries` function is to find all suitable memory ranges in fully accessible memory to load kernel. If the kernel is compiled and run on a system without EFI support, we continue to search for such memory regions in the e820 region. All memory regions found will be stored in the `slot_areas` array:

```
struct slot_area {
    unsigned long addr;
    int num;
};

#define MAX_SLOT_AREA 100

static struct slot_area slot_areas[MAX_SLOT_AREA];
```

The kernel will select a random index from this array to decompress the kernel to. The selection process is conducted by the `slots_fetch_random` function. The main goal of the `slots_fetch_random` function is to select a random memory range from the `slot_areas` array via the `kaslr_get_random_long` function:

```
slot = kaslr_get_random_long("Physical") % slot_max;
```

The `kaslr_get_random_long` function is defined in the arch/x86/lib/kaslr.c source code file and as its name suggests, returns a random number. Note that the random number can be generated in a number of ways depending on kernel configuration and features present in the system (For example, using the time stamp counter, or rdrand or some other method).

We now have a random physical address to decompress the kernel to.

7.5 Virtual address randomization

After selecting a random physical address for the decompressed kernel, we generate identity mapped pages for the region:

```
random_addr = find_random_phys_addr(min_addr, output_size);

if (*output != random_addr) {
        add_identity_map(random_addr, output_size);
        *output = random_addr;
}
```

From now on, `output` will store the base address of the memory region where kernel will be decompressed. Currrently, we have only randomized the physical address. We can randomize the virtual address as well on the x86_64 architecture:

```
if (IS_ENABLED(CONFIG_X86_64))
    random_addr = find_random_virt_addr(LOAD_PHYSICAL_ADDR, output_size);

*virt_addr = random_addr;
```

In architectures other than `x86_64`, the randomized physical and virtual addresses are the same. The `find_random_virt_addr` function calculates the number of virtual memory ranges needed to hold the kernel image. It calls the `kaslr_get_random_long` function, which we have already seen being used to generate a random `physical` address.

At this point we have randomized both the base physical (`*output`) and virtual (`*virt_addr`) addresses for the decompressed kernel.

That's all.

7.6 Conclusion

This is the end of the sixth and last part concerning the linux kernel's booting process. We will not see any more posts about kernel booting (though there may be updates to this and previous posts). We will now turn to other parts of the linux kernel instead.

The next chapter will be about kernel initialization and we will study the first steps take 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.

7.7 Links

- Address space layout randomization
- Linux kernel boot protocol
- long mode
- initrd
- Enumerated type
- four-level page tables
- five-level page tables
- EFI
- e820
- time stamp counter
- rdrand

- x86_64
- Previous part