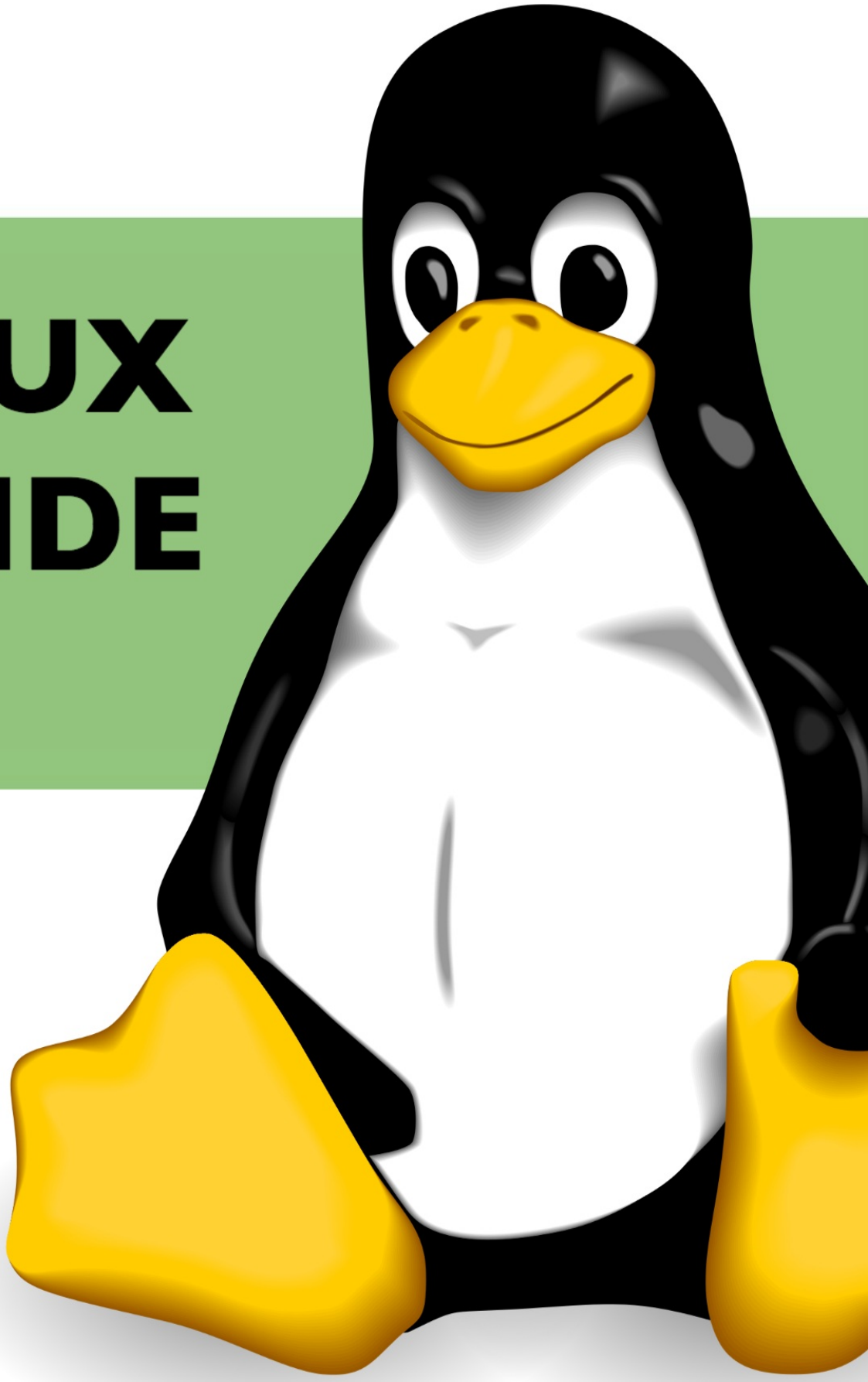
Published
with GitBook



LINUX INSIDE

By OxAX

目錄

Linux Insides

一系列关于Linux内核和其内在机理的帖子。

目的很简单 - 分享我对Linux内核内在机理的一点知识，帮助对linux内核内在机理感兴趣的人，和其他低级话题。

问题/建议: 通过在twitter上@0xAX，直接添加issue或者直接给我发邮件,请自由地向我提出任何问题或者建议。

翻译进度

章节	译者	翻译进度
Booting		正在进行
├1.1	@xinqiu	正在进行
├1.2		未开始
├1.3		未开始
├1.4		未开始
└1.5		未开始
Initialization	@lijiangsheng1	正在进行
Interrupts		未开始
System calls		未开始
Timers and time management		未开始
Memory management	@choleraehyq	正在进行
Concepts		未开始
DataStructures		正在进行
├9.1	@oska874 @mudongliang	正在进行
└9.2	Alick Guo	正在进行
Theory		正在进行
├10.1	@mudongliang	正在进行
└10.2	@mudongliang	已完成
Misc		正在进行
├12.1	@oska874	已完成
├12.2		未开始
└12.3		未开始

翻译认领规则

为了避免多个译者同时翻译相同章节的情况出现，请按照以下规则认领自己要翻译的章节：

- 开一个 issue，告诉大家你想翻译哪一章或者哪一节，并且确保 issue 页面里面没有其他人在翻译相同的章节。
- 开始翻译你认领的章节。
- 完成翻译之后，关闭 issue。

翻译前建议看TRANSLATION_NOTES.md 翻译约定，有任何问题或建议也请开 issue 讨论。

作者

@0xAX

贡献者

@xinqiu

@oska874

@mudongliang

@Alick Guo

LICENSE

内核引导过程

本章介绍了Linux内核引导过程。你将在这看到一些描述内核加载过程的整个周期的相关文章：

- 从引导加载程序内核 - 介绍了从启动计算机到内核执行第一条指令之前的所有阶段;
- 在内核安装代码的第一步 - 介绍了在内核设置代码的第一个步骤。你会看到堆的初始化，查询不同的参数，如EDD，IST和等...
- 视频模式初始化和转换到保护模式 - 介绍了视频模式初始化内核设置代码并过渡到保护模式。
- 过渡到64位模式 - 介绍了过渡到64位模式的准备并过渡到64位。
- 内核解压缩 - 介绍了内核解压缩之前的准备然后直接解压缩。

内核引导过程. Part 1.

从引导加载程序内核

如果你已经看过我之前的文章，就知道之前我开始和底层编程打交道。我写了一些关于Linux x86_64 汇编的文章。同时，我开始深入研究Linux源代码。底层是如果工作的，程序是如何在电脑上运行的，他们是如何在内存中定位的，内核是如何管理进程和内存，网络堆栈是如何在底层工作的等等，这些我都非常感兴趣。因此，我决定去写另外的一系列文章关于**x86_64**框架的Linux内核。

值得注意的是我不是一个专业的内核黑客并且我的工作不是为内核贡献代码。这只是小兴趣。我只是喜欢底层的东西，底层是如何工作的让我产生了很大的兴趣。如果你发现任何迷惑的地方或者你有任何问题/备注，twitter，email我或者提一个issue.(PS:翻译上的问题请mail我:xinqiu.94@gmail.com或github上@xinqiu)。我会很高兴。所有的文章也可以在linux-insides上看，如果你发现哪里英文或内容错误，随意提个PR。(PS:中文版地址：https://github.com/xinqiu/linux-insides)

注意这不是官方文档，只是学习和分享知识

需要的基础知识

- 理解 C 代码
- 理解 汇编语言 代码 （AT&T 语法）

不管怎样，如果你才开始学一些，我会在这些文章中尝试去解释一些部分。好了，小的介绍结束，我们开始深入内核和底层。

所有的代码实际上是内核 - 3.18.如果有任何改变，我将会做相应的更新。

神奇的电源按钮，接下来会发生什么？

尽管这一系列文章关于 Linux 内核，我们还没有从内核代码（至少在这一章）开始。好了，当你按下你笔记本或台式机的神奇电源按钮，它开始工作。在主板发送一个信号给电源，电源提供电脑适当量的电力。一旦主板收到了电源备妥信号,它会尝试运行 CPU 。CPU 复位寄存器里的所有剩余数据，设置预定义的值给每个寄存器。

80386 以及后来的 CPUs 在电脑复位后，在 CPU 寄存器中定义了如下预定义数据：

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

处理器开始在实模式工作，我们需要退回一点去理解在这种模式下的内存分割。所有 x86兼容处理器都支持实模式，从8086到现在的Intel 64位 CPU。8086处理器有一个20位寻址总线，这意味着它可以对0到2^20 位地址空间进行操作(1Mb).不过它只有16位的寄存器，通过这个16位寄存器最大寻址是2^16即 0xffff(64 Kb)。内存分配 被用来充分利用所有空闲地址空间。所有内存被分成固定的65535 字节或64 KB大小的小块。由于我们不能用16位寄存器寻址小于64KB的内存，一种替代的方法被设计出来了。一个地址包括两个部分：数据段起始地址和从该数据段起的偏移量。为了得到内存中的物理地址，我们要让数据段乘16并加上偏移量：

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

举个例子，如果 `CS:IP` 是 `0x2000:0x0010` , 相关的物理地址将会是：

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

不过如果我们让最大端进行偏移： `0xffff:0xffff` ，将会是：

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

这超出1MB65519字节。既然只有1MB在实模式中可以访问， `0x10ffef` 变成有A20缺陷的 `0x00ffef` 。

我们知道实模式和内存地址。回到复位后的寄存器值。

`CS` 寄存器包含两个部分：可视段选择器和隐含基址。 结合之前定义的 `CS` 基址和 `IP` 值，逻辑地址应该是：

```
0xffff0000:0xfff0
```

这种形式的起始地址为EIP寄存器里的值加上基址地址：

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

得到的 `0xfffffff0` 是4GB - 16 字节。这个地方是 复位向量(Reset vector) 。这是CPU在重置后期望执行的第一条指令的内存地址。它包含一个 jump 指令，这个指令通常指向BIOS入口点。举个例子，如果访问 coreboot 源代码，将看到：

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

跳转指令 opcode - 0xe9 到地址 `_start - ( . + 2)` 。 `reset` 段是16字节，起始于 `0xfffffff0` ：

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

现在BIOS已经开始工作了。在初始化和检查硬件之后，需要寻找可引导设备。引导顺序储存在BIOS配置中。引导顺序的功能是控制内核尝试引导的设备。对于尝试引导一个硬件，BIOS尝试寻找引导扇区。在硬盘分区上有一个MBR分区布局，引导扇区储存在第一个扇区(512字节)的起始446字节。剩下的第一个扇区的两个字节是 `0x55` 和 `0xaa` ，发出设备可引导的信号给BIOS。举个例子：

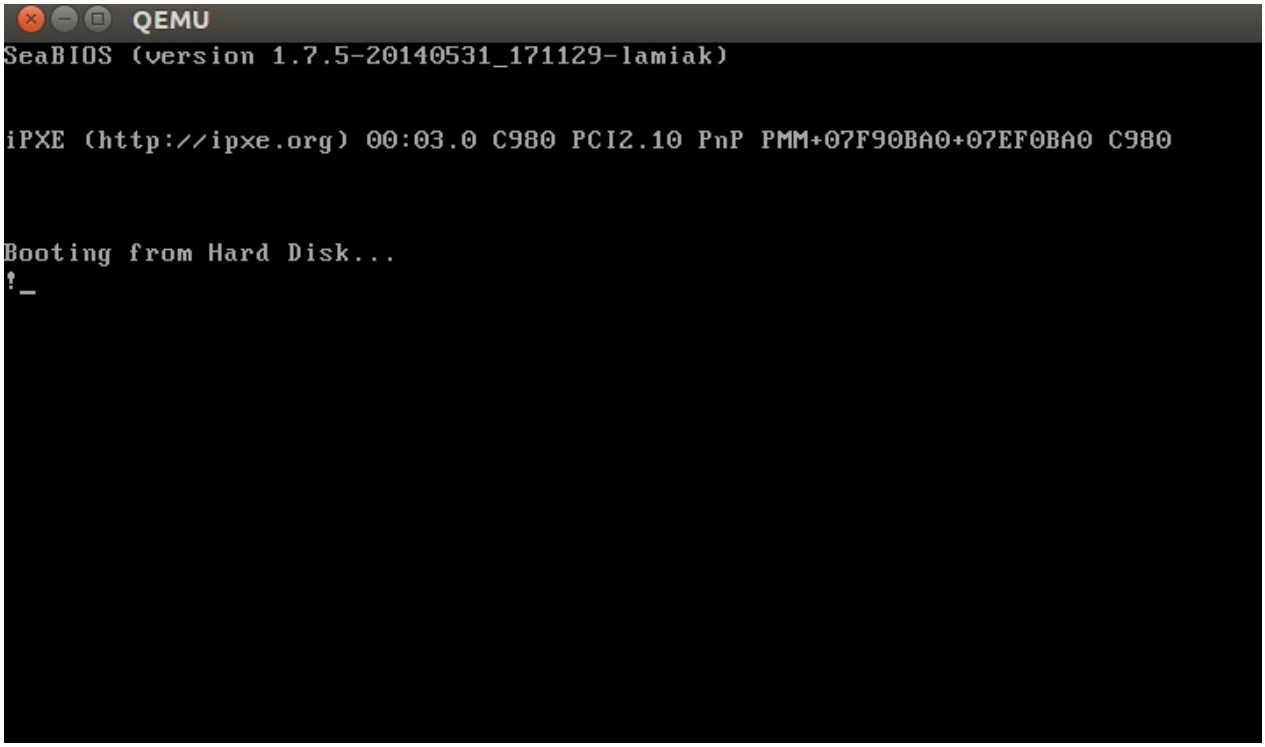
```
;
; Note: this example is written in Intel Assembly syntax
;
[BITS 16]
[ORG  0x7c00]

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

    int 0x10
    jmp $

times 510-($-$$) db 0

db 0x55
db 0xaa
```

构建并运行：

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

这让 QEMU 使用刚才新建的 `boot` 二进制文件作为磁盘镜像。由于这个二进制文件是由上述汇编语言产生，它满足引导扇区(起始设为 `0x7c00`，用Magic Sequence结束)的需求。QEMU将这个二进制文件作为磁盘镜像的主引导记录(MBR)。

将看到:



在这个例子中，这段代码被执行在16位的实模式，起始于内存0x7c00。之后调用 0x10 中断打印 `!` 符号。用0填充剩余的510字节并用两个Magic Bytes `0xaa` 和 `0x55` 结束。

可以使用 `objdump` 工具来查看转储信息：

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

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

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

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

Same as I mentioned before. But we have only 16 bit general purpose registers. The maximum value of 16 bit register is: `0xffff` ; So if we take the biggest values the result will be:

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

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

General real mode's memory map is:

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

But stop, at the beginning of post I wrote that first instruction executed by the CPU is located at the address `0xFFFFFFF0`, which is much bigger than `0xFFFFF` (1MB). How can CPU access it in real mode? As I write about it and you can read in coreboot documentation:

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

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

Bootloader

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

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

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

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

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

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

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

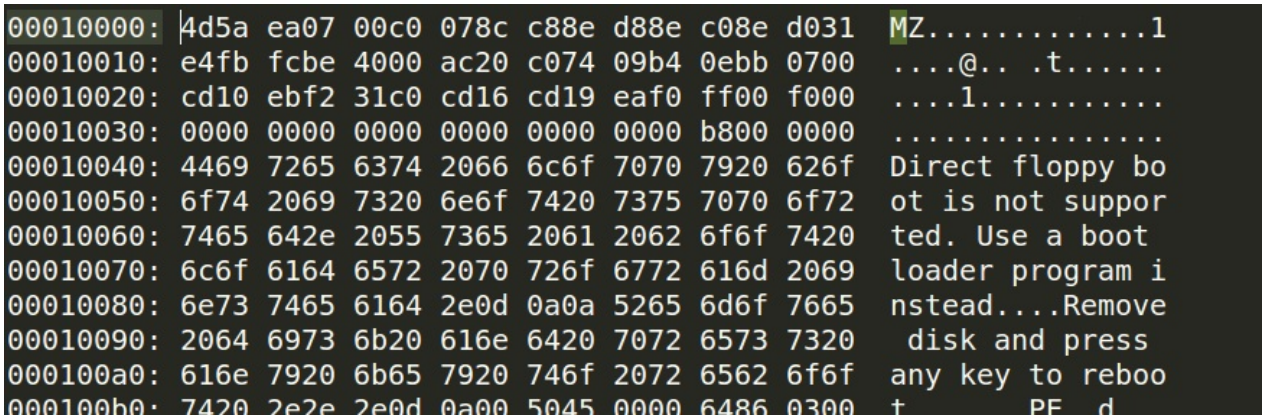
As we can see in kernel boot protocol, the memory map will be the following after kernel loading:

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

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

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

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



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

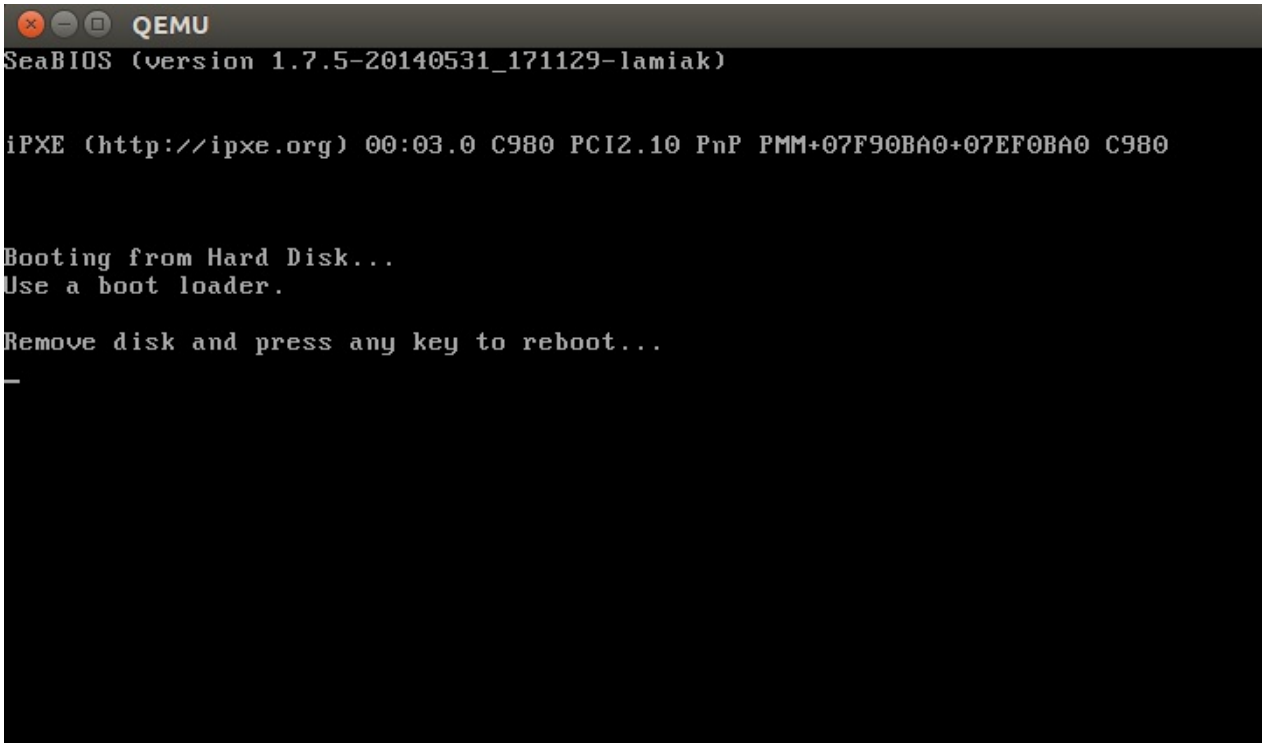
Start of Kernel Setup

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

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

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

You will see:



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

```
#ifdef CONFIG_EFI_STUB
# "MZ", MS-DOS header
.byte 0x4d
.byte 0x5a
#endif
...
...
...
pe_header:
    .ascii "PE"
    .word 0
```

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

So the actual kernel setup entry point is:

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

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

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

So kernel setup entry point is:

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

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

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

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

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

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

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

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

- Be sure that all values of all segment registers are equal
- Setup correct stack if needed
- Setup bss
- Jump to C code at main.c

Let's look at implementation.

Segment registers align

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

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

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

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

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

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

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

Stack Setup

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

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

Generally, it can be 3 different cases:

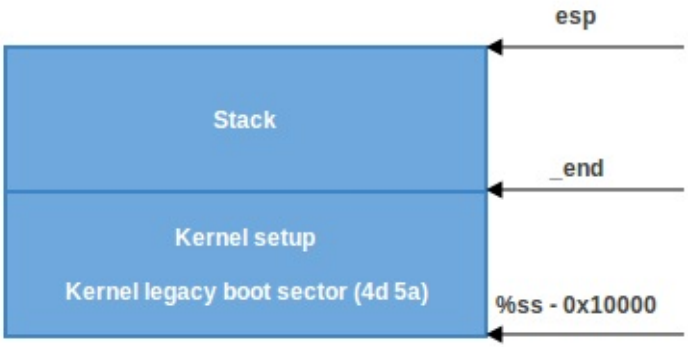
- `ss` has valid value 0x10000 (as all other segment registers beside `cs`)
- `ss` is invalid and `CAN_USE_HEAP` flag is set (see below)
- `ss` is invalid and `CAN_USE_HEAP` flag is not set (see below)

Let's look at all of these cases:

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

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

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



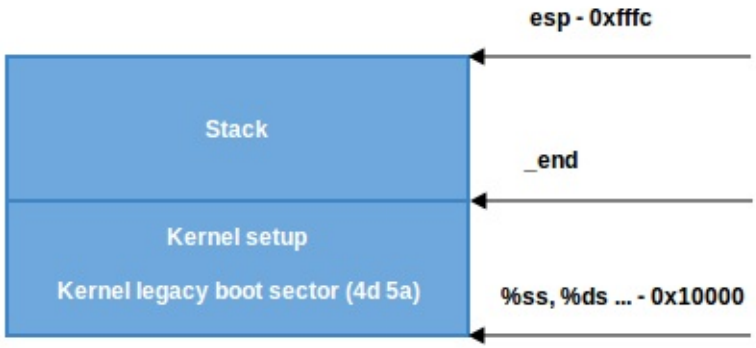
1. In the second case (`ss` != `ds`), first of all put _end (address of end of setup code) value in `dx` . And check `loadflags` header field with `testb` instruction too see if we can use heap or not. loadflags is a bitmask header which is defined as:

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

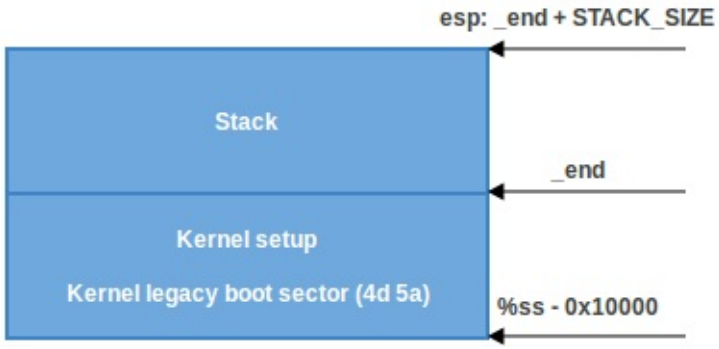
And as we can read in the boot protocol:

```
Field name:    loadflags

  This field is a bitmask.

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

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



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



BSS Setup

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

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

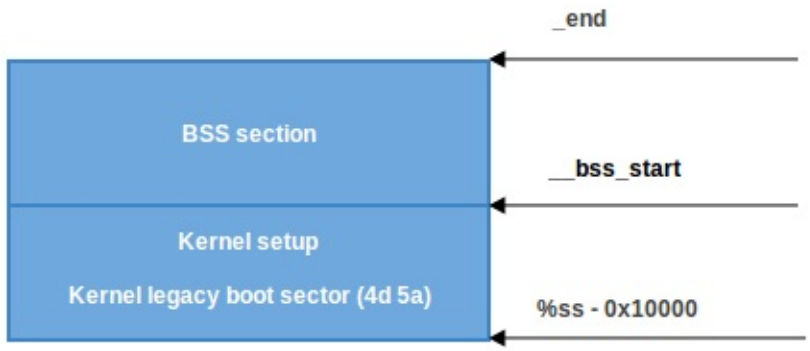
This simply consists of comparing the setup_sig against the magic number `0x5a5aaa55`. If they are not equal, a fatal error is reported.

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

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

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

First of all the __bss_start address is moved into `di`, and the `_end + 3` address (+3 - aligns to 4 bytes) is moved into `cx`. The `eax` register is cleared (using an `xor` instruction), and the bss section size (`cx` - `di`) is calculated and put into `cx`. Then, `cx` is divided by four (the size of a 'word'), and the `stosl` instruction is repeatedly used, storing the value of `eax` (zero) into the address pointed to by `di`, and automatically increasing `di` by four (this occurs until `cx` reaches zero). The net effect of this code, is that zeros are written through all words in memory from `__bss_start` to `_end`:



Jump to main

That's all, we have the stack, BSS and now we can jump to the `main()` C function:

```
    calll main
```

The `main()` function is located in arch/x86/boot/main.c. What will be there? We will see it in the next part.

Conclusion

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

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

Links

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

Linux内核中的数据结构

Linux内核对很多数据结构提供不同的实现方法，比如，双向链表，B+树，具有优先级的堆等等。

这部分考虑这些数据结构和算法。

- 双向链表
- 基数树

Linux内核中的数据结构

双向链表

Linux kernel provides its own doubly linked list implementation which you can find in the include/linux/list.h. We will start `Data Structures in the Linux kernel` from the doubly linked list data structure. Why? Because it is very popular in the kernel, just try to search

First of all let's look on the main structure:

```
struct list_head {
    struct list_head *next, *prev;
};
```

You can note that it is different from many lists implementations which you have seen. For example this doubly linked list structure from the glib:

```
struct GList {
  gpointer data;
  GList *next;
  GList *prev;
};
```

Usually a linked list structure contains a pointer to the item. Linux kernel implementation of the list does not. So the main question is - `where does the list store the data?` . The actual implementation of lists in the kernel is - `Intrusive list` . An intrusive linked list does not contain data in its nodes - A node just contains pointers to the next and previous node and list nodes part of the data that are added to the list. This makes the data structure generic, so it does not care about entry data type anymore.

For example:

```
struct nmi_desc {
    spinlock_t lock;
    struct list_head head;
};
```

Let's look at some examples to understand how `list_head` is used in the kernel. As I already wrote about, there are many, really many different places where lists are used in the kernel. Let's look for example in miscellaneous character drivers. Misc character drivers API

from the drivers/char/misc.c is used for writing small drivers for handling simple hardware or virtual devices. This drivers share major number:

```
#define MISC_MAJOR              10
```

but have their own minor number. For example you can see it with:

```
ls -l /dev |  grep 10
crw-------   1 root root     10, 235 Mar 21 12:01 autofs
drwxr-xr-x  10 root root         200 Mar 21 12:01 cpu
crw-------   1 root root     10,  62 Mar 21 12:01 cpu_dma_latency
crw-------   1 root root     10, 203 Mar 21 12:01 cuse
drwxr-xr-x   2 root root         100 Mar 21 12:01 dri
crw-rw-rw-   1 root root     10, 229 Mar 21 12:01 fuse
crw-------   1 root root     10, 228 Mar 21 12:01 hpet
crw-------   1 root root     10, 183 Mar 21 12:01 hwrng
crw-rw----+  1 root kvm      10, 232 Mar 21 12:01 kvm
crw-rw----   1 root disk     10, 237 Mar 21 12:01 loop-control
crw-------   1 root root     10, 227 Mar 21 12:01 mcelog
crw-------   1 root root     10,  59 Mar 21 12:01 memory_bandwidth
crw-------   1 root root     10,  61 Mar 21 12:01 network_latency
crw-------   1 root root     10,  60 Mar 21 12:01 network_throughput
crw-r-----   1 root kmem     10, 144 Mar 21 12:01 nvram
brw-rw----   1 root disk      1,  10 Mar 21 12:01 ram10
crw--w----   1 root tty       4,  10 Mar 21 12:01 tty10
crw-rw----   1 root dialout   4,  74 Mar 21 12:01 ttyS10
crw-------   1 root root     10,  63 Mar 21 12:01 vga_arbiter
crw-------   1 root root     10, 137 Mar 21 12:01 vhci
```

Now let's have a close look at how lists are used in the misc device drivers. First of all let's look on `miscdevice` structure:

```
struct miscdevice
{
      int minor;
      const char *name;
      const struct file_operations *fops;
      struct list_head list;
      struct device *parent;
      struct device *this_device;
      const char *nodename;
      mode_t mode;
};
```

We can see the fourth field in the `miscdevice` structure - `list` which is a list of registered devices. In the beginning of the source code file we can see the definition of misc_list:

```
static LIST_HEAD(misc_list);
```

which expands to definition of the variables with `list_head` type:

```
#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)
```

and initializes it with the `LIST_HEAD_INIT` macro which set previous and next entries:

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }
```

Now let's look on the `misc_register` function which registers a miscellaneous device. At the start it initializes `miscdevice->list` with the `INIT_LIST_HEAD` function:

```
INIT_LIST_HEAD(&misc->list);
```

which does the same as the `LIST_HEAD_INIT` macro:

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}
```

In the next step after device created with the `device_create` function we add it to the miscellaneous devices list with:

```
list_add(&misc->list, &misc_list);
```

Kernel `list.h` provides this API for the addition of new entry to the list. Let's look on it's implementation:

```
static inline void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}
```

It just calls internal function `__list_add` with the 3 given parameters:

- new - new entry;
- head - list head after which the new item will be inserted

- head->next - next item after list head.

Implementation of the `__list_add` is pretty simple:

```
static inline void __list_add(struct list_head *new,
                  struct list_head *prev,
                  struct list_head *next)
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}
```

Here we set new item between `prev` and `next`. So `misc` list which we defined at the start with the `LIST_HEAD_INIT` macro will contain previous and next pointers to the `miscdevice->list`.

There is still one question: how to get list's entry. There is a special macro:

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

which gets three parameters:

- ptr - the structure list_head pointer;
- type - structure type;
- member - the name of the list_head within the structure;

For example:

```
const struct miscdevice *p = list_entry(v, struct miscdevice, list)
```

After this we can access to any `miscdevice` field with `p->minor` or `p->name` and etc... Let's look on the `list_entry` implementation:

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

As we can see it just calls `container_of` macro with the same arguments. At first sight, the `container_of` looks strange:

```
#define container_of(ptr, type, member) ({                      \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

First of all you can note that it consists of two expressions in curly brackets. Compiler will evaluate the whole block in the curly braces and use the value of the last expression.

For example:

```
#include <stdio.h>

int main() {
    int i = 0;
    printf("i = %d\n", ({++i; ++i;}));
    return 0;
}
```

will print `2`.

The next point is `typeof`, it's simple. As you can understand from its name, it just returns the type of the given variable. When I first saw the implementation of the `container_of` macro, the strangest thing for me was the zero in the `((type *)0)` expression. Actually this pointer magic calculates the offset of the given field from the address of the structure, but as we have `0` here, it will be just a zero offset alongwith the field width. Let's look at a simple example:

```
#include <stdio.h>

struct s {
        int field1;
        char field2;
        char field3;
};

int main() {
    printf("%p\n", &((struct s*)0)->field3);
    return 0;
}
```

will print `0x5`.

The next offsetof macro calculates offset from the beginning of the structure to the given structure's field. Its implementation is very similar to the previous code:

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

Let's summarize all about `container_of` macro. `container_of` macro returns address of the structure by the given address of the structure's field with `list_head` type, the name of the structure field with `list_head` type and type of the container structure. At the first line this macro declares the `__mptr` pointer which points to the field of the structure that `ptr` points to and assigns `ptr` to it. Now `ptr` and `__mptr` point to the same address. Technically we don't need this line but its useful for type checking. First line ensures that that given structure (`type` parameter) has a member called `member` . In the second line it calculates offset of the field from the structure with the `offsetof` macro and subtracts it from the structure address. That's all.

Of course `list_add` and `list_entry` is not the only functions which `<linux/list.h>` provides. Implementation of the doubly linked list provides the following API:

- list_add
- list_add_tail
- list_del
- list_replace
- list_move
- list_is_last
- list_empty
- list_cut_position
- list_splice

and many more.

Linux内核中的数据结构

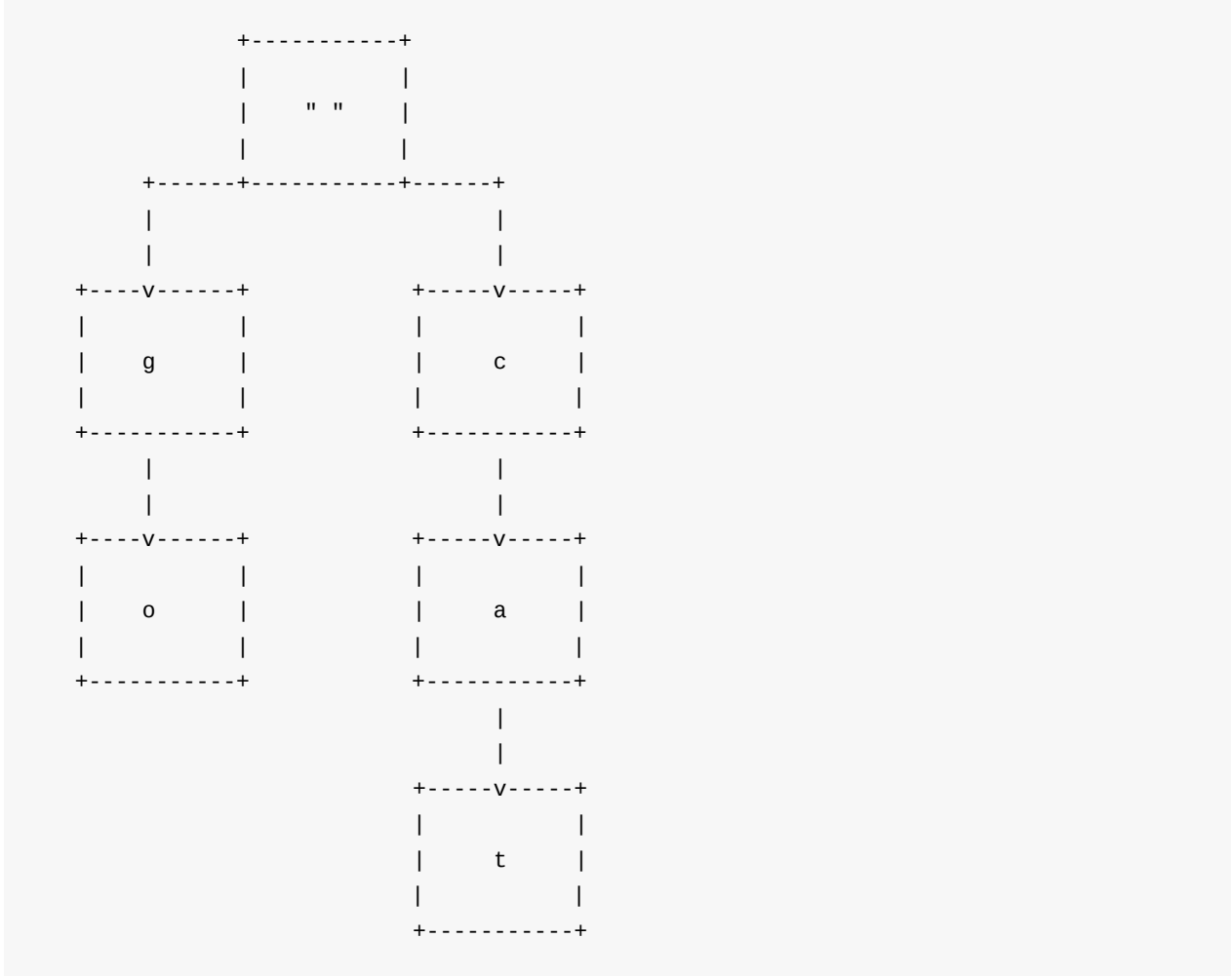
基数树

正如你所知道的 Linux 内核通过许多不同库以及函数提供各种数据结构以及算法实现。 这个部分我们将介绍其中一个数据结构 Radix tree。Linux 内核中有两个文件与 `radix tree` 的实现和API相关：

- include/linux/radix-tree.h
- lib/radix-tree.c

首先说明一下什么是 `radix tree` 。Radix tree 是一种 `压缩 trie`，其中 trie 是一种通过保存关联数组（associative array）来提供 `关键字-值（key-value）` 存储与查找的数据结构。通常关键字是字符串，不过也可以是其他数据类型。

trie 结构的节点与 `n-tree` 不同，其节点中并不存储关键字，取而代之的是存储单个字符标签。关键字查找时，通过从树的根开始遍历关键字相关的所有字符标签节点，直至到达最终的叶子节点。下面是个例子：



这个例子中，我们可以看到 `trie` 所存储的关键字信息 `go` 与 `cat`，压缩 trie 或 `radix tree` 与 `trie` 所不同的是，所有只存在单个孩子的中间节点将被压缩。

Linux 内核中的 Radix 树将值映射为整型关键字，Radix 的数据结构定义在 include/linux/radix-tree.h 文件中：

```
struct radix_tree_root {
        unsigned int            height;
        gfp_t                   gfp_mask;
        struct radix_tree_node  __rcu *rnode;
};
```

上面这个是 radix 树的 root 节点的结构体，它包括三个成员：

- `height` - 从叶节点向上计算出的树高度。
- `gfp_mask` - 内存分配标识。
- `rnode` - 子节点指针。

这里我们先讨论的结构体成员是 `gfp_mask`：

Linux 底层的内存申请接口需要提供一类标识（flag）- `gfp_mask` ，用于描述内存申请的行为。这个以 `GFP_` 前缀开头的内存申请控制标识主要包括，`GFP_NOIO` 禁止所有IO操作但允许睡眠等待内存，`__GFP_HIGHMEM` 允许申请内核的高端内存，`GFP_ATOMIC` 高优先级申请内存且操作不允许被睡眠。

接下来说的结构体成员是 `rnode` ：

```
struct radix_tree_node {
        unsigned int    path;
        unsigned int    count;
        union {
                struct {
                        struct radix_tree_node *parent;
                        void *private_data;
                };
                struct rcu_head rcu_head;
        };
        /* For tree user */
        struct list_head private_list;
        void __rcu      *slots[RADIX_TREE_MAP_SIZE];
        unsigned long   tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];
};
```

这个结构体中包括这几个内容，节点与父节点的偏移以及到树底端的高度，子节点的个数，节点的存储数据域，具体描述如下：

- `path` - 从叶节点
- `count` - 子节点的个数。
- `parent` - 父节点的指针。
- `private_data` - 存储数据内容缓冲区。
- `rcu_head` - 用于节点释放的RCU链表。
- `private_list` - 存储数据。

结构体 `radix_tree_node` 的最后两个成员 `tags` 与 `slots` 是非常重要且需要特别注意的。每个 Radix 树节点都可以包括一个指向存储数据指针的 slots 集合，空闲 slots 的指针指向 NULL。 Linux 内核的 Radix 树结构体中还包含用于记录节点存储状态的标签 `tags` 成员，标签通过位设置指示 Radix 树的数据存储状态。

至此，我们了解到 radix 树的结构，接下来看一下 radix 树所提供的 API。

Linux 内核基数树 API

我们从数据结构的初始化开始看，radix 树支持两种方式初始化。

第一个是使用宏 `RADIX_TREE` ：

```
RADIX_TREE(name, gfp_mask);
`
```

正如你看到，只需要提供 `name` 参数，就能够使用 `RADIX_TREE` 宏完成 radix 的定义以及初始化，`RADIX_TREE` 宏的实现非常简单：

```
#define RADIX_TREE(name, mask) \
         struct radix_tree_root name = RADIX_TREE_INIT(mask)

#define RADIX_TREE_INIT(mask)   { \
        .height = 0,              \
        .gfp_mask = (mask),       \
        .rnode = NULL,            \
}
```

`RADIX_TREE` 宏首先使用 `name` 定义了一个 `radix_tree_root` 实例并用 `RADIX_TREE_INIT` 宏带参数 `mask` 进行初始化。宏 `RADIX_TREE_INIT` 将 `radix_tree_root` 初始化为默认属性并将 gfp_mask 初始化为入参 `mask` 。第二种方式是手工定义 `radix_tree_root` 变量，之后再使用 `mask` 调用 `INIT_RADIX_TREE` 宏对变量进行初始化。

```
struct radix_tree_root my_radix_tree;
INIT_RADIX_TREE(my_tree, gfp_mask_for_my_radix_tree);
```

`INIT_RADIX_TREE` 宏定义：

```
#define INIT_RADIX_TREE(root, mask)  \
do {                                 \
        (root)->height = 0;          \
        (root)->gfp_mask = (mask);   \
        (root)->rnode = NULL;        \
} while (0)
```

宏 `INIT_RADIX_TREE` 所初始化的属性与 `RADIX_TREE_INIT` 一致

接下来是 radix 树的节点插入以及删除，这两个函数：

- `radix_tree_insert`;
- `radix_tree_delete`.

第一个函数 `radix_tree_insert` 需要三个入参：

- radix 树 root 节点结构
- 索引关键字
- 需要插入存储的数据

第二个函数 `radix_tree_delete` 除了不需要存储数据参数外，其他与 `radix_tree_insert` 一致。

radix 树的查找实现有以下几个函数：The search in a radix tree implemented in two ways:

- `radix_tree_lookup`;
- `radix_tree_gang_lookup`;
- `radix_tree_lookup_slot`.

第一个函数 `radix_tree_lookup` 需要两个参数：

- radix 树 root 节点结构
- 索引关键字

这个函数通过给定的关键字查找 radix 树，并返关键字所对应的结点。

第二个函数 `radix_tree_gang_lookup` 具有以下特征：

```
unsigned int radix_tree_gang_lookup(struct radix_tree_root *root,
                                    void **results,
                                    unsigned long first_index,
                                    unsigned int max_items);
```

函数返回查找到记录的条目数，并根据关键字进行排序，返回的总结点数不超过入参 `max_items` 的大小。

最后一个函数 `radix_tree_lookup_slot` 返回结点 slot 中所存储的数据。

链接

- Radix tree
- Trie

理论

这一章描述各种理论性概念和那些不直接涉及实践，但是知道了会很有用的概念。

- 分页
- Elf64 格式

分页

Introduction

In the fifth part of the series `Linux kernel booting process` we learned about what the kernel does in its earliest stage. In the next step the kernel will initialize different things like `initrd` mounting, lockdep initialization, and many many others things, before we can see how the kernel runs the first init process.

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

In my view, memory management is one of the most complex part of the linux kernel and in system programming in general. This is why before we proceed with the kernel initialization stuff, we need to get acquainted with paging.

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

As the Intel manual says:

> Paging provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed.

So... In this post I will try to explain the theory behind paging. Of course it will be closely related to the `x86_64` version of the linux kernel for, but we will not go into too much details (at least in this post).

Enabling paging

There are three paging modes:

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

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

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

We already saw where those this bits were set in arch/x86/boot/compressed/head_64.S:

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

and

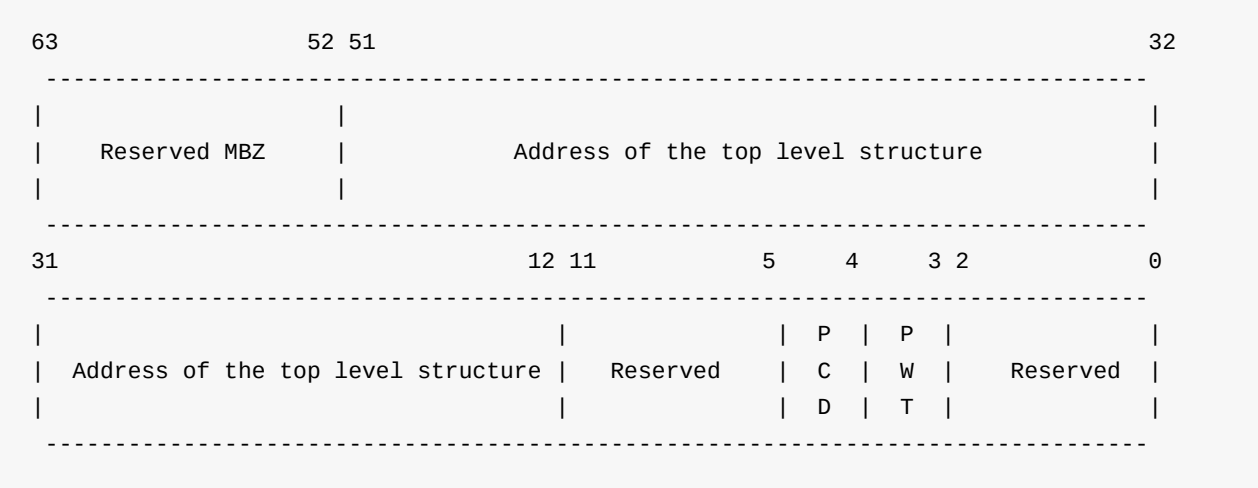
```
movl    $MSR_EFER, %ecx
rdmsr
btsl    $_EFER_LME, %eax
wrmsr
```

Paging structures

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

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

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



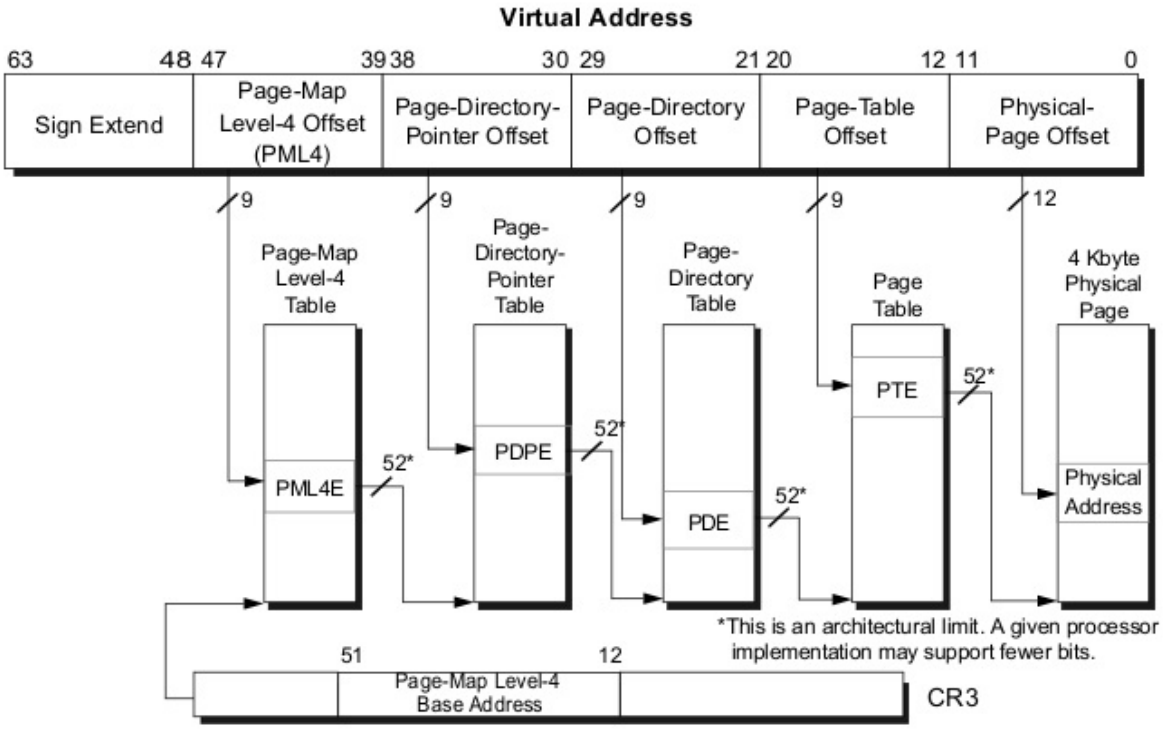
These fields have the following meanings:

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

The linear address translation address is following:

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

schematically, we can imagine it like this:



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

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

Where:

- 63 bit - N/X bit (No Execute Bit) - presents ability to execute the code from physical pages mapped by the table entry;
- 62:52 bits - ignored by CPU, used by system software;
- 51:12 bits - stores physical address of the lower level paging structure;
- 12:9 bits - ignored by CPU;
- MBZ - must be zero bits;

- Ignored bits;
- A - accessed bit indicates was physical page or page structure accessed;
- PWT and PCD used for cache;
- U/S - user/supervisor bit controls user access to the all physical pages mapped by this table entry;
- R/W - read/write bit controls read/write access to the all physical pages mapped by this table entry;
- P - present bit. Current bit indicates was page table or physical page loaded into primary memory or not.

Ok, we know about the paging structures and their entries. Now let's see some details about 4-level paging in the linux kernel.

Paging structures in the linux kernel

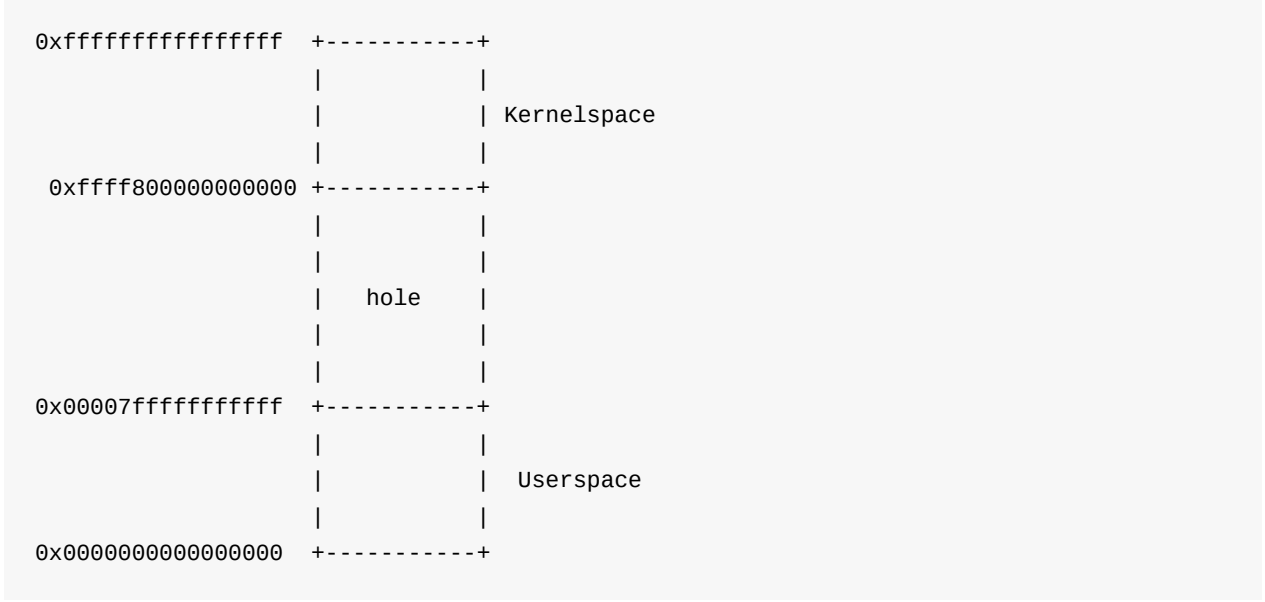
As we've seen, the linux kernel in `x86_64` uses 4-level page tables. Their names are:

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

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

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

We can see `0xffffffff81efe497` here. I doubt you really have that much RAM installed. But anyway, `start_kernel` and `x86_64_start_kernel` will be executed. The address space in `x86_64` is `2^64` size, but it's too large, that's why a smaller address space is used, only 48-bits wide. So we have a situation where the physical address space is limited to 48 bits, but addressing still performed with 64 bit pointers. How is this problem solved? Look at this diagram:



This solution is `sign extension`. Here we can see that the lower 48 bits of a virtual address can be used for addressing. Bits `63:48` can be either only zeroes or only ones. Note that the virtual address space is split in 2 parts:

- Kernel space
- Userspace

Userspace occupies the lower part of the virtual address space, from `0x0000000000000000` to `0x00007fffffffffff` and kernel space occupies the highest part from `0xffff8000000000` to `0xffffffffffffffff`. Note that bits `63:48` is 0 for userspace and 1 for kernel space. All addresses which are in kernel space and in userspace or in other words which higher `63:48` bits are zeroes or ones are called `canonical` addresses. There is a `non-canonical` area between these memory regions. Together these two memory regions (kernel space and user space) are exactly `2^48` bits wide. We can find the virtual memory map with 4 level page tables in the Documentation/x86/x86_64/mm.txt:

```
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
hole caused by [48:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffc0000000000 (=44 bits) kasan shadow memory (16TB)
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffff80000000 - ffffffffa0000000 (=512 MB)  kernel text mapping, from phys 0
ffffffffa0000000 - ffffffffff5fffff (=1525 MB) module mapping space
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
```

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

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

Previously this guard hole and `__PAGE_OFFSET` was from `0xffff800000000000` to `0xffff80ffffffffff` to prevent access to non-canonical area, but was later extended by 3 bits for the hypervisor.

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

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

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

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

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

After the kernel `.text` region there is the virtual memory region for kernel modules, `vsyscalls` and an unused hole of 2 megabytes.

We've seen how the kernel's virtual memory map is laid out and how a virtual address is translated into a physical one. Let's take for example following address:

```
0xffffffff81000000
```

In binary it will be:

```
1111111111111111 111111111 111111110 000001000 000000000 000000000000
      63:48        47:39     38:30     29:21     20:12      11:0
```

This virtual address is split in parts as described above:

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

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

Conclusion

It's the end of this short part about paging theory. Of course this post doesn't cover every detail of paging, but soon we'll see in practice how the linux kernel builds paging structures and works with them.

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

Links

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

ELF文件格式

ELF (Executable and Linkable Format)是一种为可执行文件，目标文件，共享链接库和内核转储(core dumps)准备的标准文件格式。 Linux和很多类Unix操作系统都使用这个格式。 让我们来看一下64位ELF文件格式的结构以及内核源码中有关于它的一些定义。

一个ELF文件由以下三部分组成：

- ELF头(ELF header) - 描述文件的主要特性：类型，CPU架构，入口地址，现有部分的大小和偏移等等；
- 程序头表(Program header table) - 列举了所有有效的段(segments)和他们的属性。 程序头表需要加载器将文件中的节加载到虚拟内存段中；
- 节头表(Section header table) - 包含对节(sections)的描述。

现在让我们对这些部分有一些更深的了解。

ELF头(ELF header)

ELF头(ELF header)位于文件的开始位置。 它的主要目的是定位文件的其他部分。 文件头主要包含以下字段：

- ELF文件鉴定 - 一个字节数组用来确认文件是否是一个ELF文件，并且提供普通文件特征的信息；
- 文件类型 - 确定文件类型。 这个字段描述文件是一个重定位文件，或可执行文件,或...；
- 目标结构；
- ELF文件格式的版本；
- 程序入口地址；
- 程序头表的文件偏移；
- 节头表的文件偏移；
- ELF头(ELF header)的大小；
- 程序头表的表项大小；
- 其他字段...

你可以在内核源码种找到表示ELF64 header的结构体 `elf64_hdr` ：

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

这个结构体定义在 elf.h

节(sections)

所有的数据都存储在ELF文件的节(sections)中。 我们通过节头表中的索引(index)来确认节(sections)。 节头表表项包含以下字段：

- 节的名字；
- 节的类型；
- 节的属性；
- 内存地址；
- 文件中的偏移；
- 节的大小；
- 到其他节的链接；
- 各种各样的信息；
- 地址对齐；
- 这个表项的大小，如果有的话；

而且，在linux内核中结构体 `elf64_shdr` 如下所示:

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

elf.h

程序头表**(Program header table)**

在可执行文件或者共享链接库中所有的节(sections)都被分为多个段(segments)。 程序头是一个结构的数组，每一个结构都表示一个段(segments)。 它的结构就像这样：

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

在内核源码中。

`elf64_phdr` 定义在相同的 elf.h 文件中.

EFL文件也包含其他的字段或结构。 你可以在 Documentation 中查看。 现在我们来查看一下 `vmlinux` 这个ELF文件。

vmlinux

`vmlinux` 也是一个可重定位的ELF文件。 我们可以使用 `readelf` 工具来查看它。 首先，让我们看一下它的头部：

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

我们可以看出 `vmlinux` 是一个64位可执行文件。 我们可以从 Documentation/x86/x86_64/mm.txt 读到相关信息:

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

之后我们可以在 `vmlinux` ELF文件中查看这个地址：

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

值得注意的是，`startup_64` 例程的地址不是 `ffffffff80000000`，而是 `ffffffff81000000`。现在我们来解释一下。

我们可以在 arch/x86/kernel/vmlinux.lds.S 看见如下的定义：

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

其中，`__START_KERNEL` 定义如下:

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

从这个文档中看出，`__START_KERNEL_map` 的值是 `ffffffff80000000` 以及 `__PHYSICAL_START` 的值是 `0x1000000`。这就是 `startup_64` 的地址是 `ffffffff81000000` 的原因了。

最后我们通过以下命令来得到程序头表的内容：

```
readelf -l vmlinux

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

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000200000 0xffffffff81000000 0x0000000001000000
                 0x0000000000cfd000 0x0000000000cfd000  R E    200000
  LOAD           0x0000000001000000 0xffffffff81e00000 0x0000000001e00000
                 0x0000000000100000 0x0000000000100000  RW     200000
  LOAD           0x0000000001200000 0x0000000000000000 0x0000000001f00000
                 0x0000000000014d98 0x0000000000014d98  RW     200000
  LOAD           0x0000000001315000 0xffffffff81f15000 0x0000000001f15000
                 0x000000000011d000 0x0000000000279000  RWE    200000
  NOTE           0x0000000000b17284 0xffffffff81917284 0x0000000001917284
                 0x0000000000000024 0x0000000000000024         4

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

这里我们可以看出五个包含节(sections)列表的段(segments)。 你可以在生成的链接器脚本 - `arch/x86/kernel/vmlinux.lds` 中找到所有的节(sections)。

就这样吧。 当然，它不是ELF(Executable and Linkable Format)的完整描述，但是如果你想要知道更多，可以参考这个文档 - 这里