由实模式到start_kernel

by zenhumay

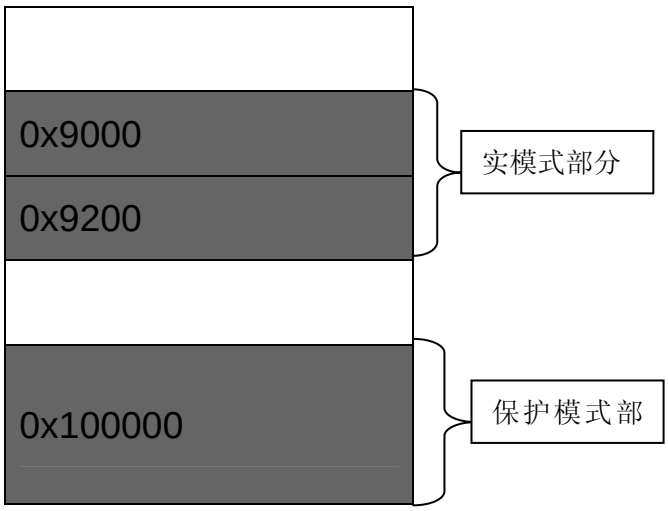
2012-03-20——2012-04-20

目录

1. 概述

本章内容涉及源代码全部来至 linux 内核 2.6.34。

当 linux 的引导程序（LILO,GRUB）将 linux 加载到内核后，引导程序的使命已经完成，然后将控制权转交给 linux 内核。



图表 1 引导程序加载完成后 linux 内核布局图

当引导程序将内核加载到内存后，linux 内核在内存的布局如图 1 所示。

本章主要记录 linux 由实模式跳转到 start_kernel 的过程。

2. 实模式（建立C函数的运行环境）

/arch/x86/boot/header.S

2.1 0x9000-0x9200 代码

```
28 BOOTSEG      = 0x07C0        /* original address of boot-sector */
29 SYSSEG       = 0x1000        /* historical load address >> 4 */
30
31 #ifndef SVGA_MODE
32 #define SVGA_MODE ASK_VGA
33 #endif
34
35 #ifndef RAMDISK
36 #define RAMDISK 0
37 #endif
38
39 #ifndef ROOT_RDONLY
40 #define ROOT_RDONLY 1
41 #endif
42
43      .code16
44      .section ".bstext", "ax"
45
46      .global bootsect_start
47 bootsect_start:
48
49      # Normalize the start address
50      ljmp    $BOOTSEG, $start2
51
52 start2:
53      movw    %cs, %ax
54      movw    %ax, %ds
55      movw    %ax, %es
56      movw    %ax, %ss
```

```
57        xorw    %sp, %sp
58        sti
59        cld
60
61        movw    $bugger_off_msg, %si
62
63 msg_loop:
64        lodsb
65        andb    %al, %a
66      jz    bs_die
67        movb    $0xe, %ah
68        movw    $7, %bx
69        int $0x10
70        jmp msg_loop
71
72 bs_die:
73        # Allow the user to press a key, then reboot
74        xorw    %ax, %ax
75        int $0x16
76        int $0x19
77
78        # int 0x19 should never return.    In case it does anyway,
79        # invoke the BIOS reset code...
80        ljmp    $0xf000,$0xfff0 #重启计算机
81
82        .section ".bsdata", "a"
83 bugger_off_msg:
84        .ascii  "Direct booting from floppy is no longer supported.\r\n"
85        .ascii  "Please use a boot loader program instead.\r\n"
86        .ascii  "\n"
87        .ascii  "Remove disk and press any key to reboot . . .\r\n"
```

```
88          .byte     0
89
90
```

47-80 行的代码处理 bios 直接将内核镜像 vmlinuz 加载到 0x07c0 处的地址，如果从 47 行开始执行，说明 bios 直接加载了内核镜像，这是不允许的。因为 linux 都需要经过一个特定的引导程序（GRUB，LILO 等）加载。

```
91          # Kernel attributes; used by setup.    This is part 1 of the
92          # header, from the old boot sector.
93
94          .section ".header", "a"
95          .globl    hdr
96 hdr:
97 setup_sects:       .byte 0               /* Filled in by build.c */
98 root_flags: .word ROOT_RDONLY
99 syssize:       .long 0               /* Filled in by build.c */
100 ram_size:      .word 0               /* Obsolete */
101 vid_mode:      .word SVGA_MODE
102 root_dev:      .word 0               /* Filled in by build.c */
103 boot_flag:    .word 0xAA55
104
105         # offset 512, entry point


106
107         .globl    _start
108 _start:
109             # Explicitly enter this as bytes, or the assembler
110             # tries to generate a 3-byte jump here, which causes
111             # everything else to push off to the wrong offset.
```

```
112             .byte   0xeb            # short (2-byte) jump
113             .byte   start_of_setup-1f
```

内核的 setup.ld 链接脚本 setub.bin 入口点是_start，grub 加载 vmlinuz 之后将跳转到_start（0x9200）处开始执行。

```
114 1:
115
116         # Part 2 of the header, from the old setup.S
117
118             .ascii  "HdrS"          # header signature
119             .word   0x020a          # header version number (>= 0x0105)
120                                     # or else old loadlin-1.5 will fail)
121             .globl realmode_swtch
122 realmode_swtch: .word   0, 0               # default_switch, SETUPSEG
123 start_sys_seg:  .word   SYSSEG          # obsolete and meaningless, but just
124                                     # in case something decided to "use" it
125             .word   kernel_version-512 # pointing to kernel version string
126                                     # above section of header is compatible
127                                     # with loadlin-1.5 (header v1.5). Don't
128                                     # change it.
129
130 type_of_loader: .byte   0           # 0 means ancient bootloader, newer
131                                     # bootloaders know to change this.
132                                     # See Documentation/i386/boot.txt for
133                                     # assigned ids
134
135 # flags, unused bits must be zero (RFU) bit within loadflags
136 loadflags:
137 LOADED_HIGH = 1                 # If set, the kernel is loaded high
138 CAN_USE_HEAP    = 0x80              # If set, the loader also has set
139                                     # heap_end_ptr to tell how much
```

```
140                             # space behind setup.S can be used for
141                             # heap purposes.
142                             # Only the loader knows what is free
143             .byte   LOADED_HIGH
144
145 setup_move_size: .word  0x8000         # size to move, when setup is not
146                             # loaded at 0x90000. We will move setup
147                             # to 0x90000 then just before jumping
148                             # into the kernel. However, only the
149                             # loader knows how much data behind
150                             # us also needs to be loaded.
151
152 code32_start:                       # here loaders can put a different
153                             # start address for 32-bit code.
154             .long   0x100000        # 0x100000 = default for big kernel
155
156 ramdisk_image:  .long   0           # address of loaded ramdisk image
157                             # Here the loader puts the 32-bit
158                             # address where it loaded the image.
159                             # This only will be read by the kernel.
160
161 ramdisk_size:   .long   0           # its size in bytes
162
163 bootsect_kludge:
164             .long   0           # obsolete
165
166 heap_end_ptr:   .word   _end+STACK_SIZE-512
167                             # (Header version 0x0201 or later)
168                             # space from here (exclusive) down to
169                             # end of setup code can be used by setup
170                             # for local heap purposes.
```

```
171
172 ext_loader_ver:
173             .byte   0           # Extended boot loader version
174 ext_loader_type:
175             .byte   0           # Extended boot loader type
176
177 cmd_line_ptr:   .long   0           # (Header version 0x0202 or later)
178                             # If nonzero, a 32-bit pointer
179                             # to the kernel command line.
180                             # The command line should be
181                             # located between the start of
182                             # setup and the end of low
183                             # memory (0xa0000), or it may
184                             # get overwritten before it
185                             # gets read.  If this field is
186                             # used, there is no longer
187                             # anything magical about the
188                             # 0x90000 segment; the setup
189                             # can be located anywhere in
190                             # low memory 0x10000 or higher.
191
192 ramdisk_max:    .long 0x7fffffff
193                             # (Header version 0x0203 or later)
194                             # The highest safe address for
195                             # the contents of an initrd
196                             # The current kernel allows up to 4 GB,
197                             # but leave it at 2 GB to avoid
198                             # possible bootloader bugs.
199
200 kernel_alignment:  .long CONFIG_PHYSICAL_ALIGN  #physical addr alignment
201                                 #required for protected mode
```

```
202                                     #kernel
203 #ifdef CONFIG_RELOCATABLE
204 relocatable_kernel:       .byte 1
205 #else
206 relocatable_kernel:       .byte 0
207 #endif
208 min_alignment:          .byte MIN_KERNEL_ALIGN_LG2    # minimum alignment
209 pad3:                   .word 0
210
211 cmdline_size:     .long     COMMAND_LINE_SIZE-1         #length of the command line,
212                                                         #added with boot protocol
213                                                         #version 2.06
214
215 hardware_subarch:     .long 0               # subarchitecture, added with 2.07
216                                     # default to 0 for normal x86 PC
217
218 hardware_subarch_data:    .quad 0
219
220 payload_offset:         .long ZO_input_data
221 payload_length:         .long ZO_z_input_len
222
223 setup_data:         .quad 0               # 64-bit physical pointer to
224                                     # single linked list of
225                                     # struct setup_data
226
227 pref_address:           .quad LOAD_PHYSICAL_ADDR       # preferred load addr
228
229 #define ZO_INIT_SIZE       (ZO__end - ZO_startup_32 + ZO_z_extract_offset)
230 #define VO_INIT_SIZE       (VO__end - VO__text)
231 #if ZO_INIT_SIZE > VO_INIT_SIZE
232 #define INIT_SIZE ZO_INIT_SIZE
```

```
233 #else
234 #define INIT_SIZE VO_INIT_SIZE
235 #endif
236 init_size:          .long INIT_SIZE         # kernel initialization size
237
238 # End of setup header #####################################################
239
```

96 行到 239 行初始化内核启动时非常重要的一个结构，setup_header。

该结构在 arch/x86/include/asm/bootparam.h 中定义：

从结构的定义和 header.S 中的代码可以看出，它们之间的字段是一一对应的。

```
24 struct setup_header {
25        __u8    setup_sects;
26        __u16   root_flags;
27        __u32   syssize;
28        __u16   ram_size;
29 #define RAMDISK_IMAGE_START_MASK      0x07FF
30 #define RAMDISK_PROMPT_FLAG        0x8000
31 #define RAMDISK_LOAD_FLAG          0x4000
32        __u16   vid_mode;
33        __u16   root_dev;
34        __u16   boot_flag;
35        __u16   jump;
36        __u32   header;
37        __u16   version;
38        __u32   realmode_swtch;
39        __u16   start_sys;
```

```
    __u16   kernel_version;
    __u8    type_of_loader;
    __u8    loadflags;
#define LOADED_HIGH (1<<0)
#define QUIET_FLAG  (1<<5)
#define KEEP_SEGMENTS   (1<<6)
#define CAN_USE_HEAP    (1<<7)
    __u16   setup_move_size;
    __u32   code32_start;
    __u32   ramdisk_image;
    __u32   ramdisk_size;
    __u32   bootsect_kludge;
    __u16   heap_end_ptr;
    __u8    ext_loader_ver;
    __u8    ext_loader_type;
    __u32   cmd_line_ptr;
    __u32   initrd_addr_max;
    __u32   kernel_alignment;
    __u8    relocatable_kernel;
    __u8    _pad2[3];
    __u32   cmdline_size;
    __u32   hardware_subarch;
    __u64   hardware_subarch_data;
    __u32   payload_offset;
    __u32   payload_length;
    __u64   setup_data;

} __attribute__((packed));
```

引导程序跳转到_start 出，指向 112-113 行的代码，这是一个短跳转，对以的机器码如下，该指令执行后，将跳转到 241 行的代码开始执行。

```
112             .byte   0xeb            # short (2-byte) jump
113             .byte   start_of_setup-1f
```

```
240     .section ".entrytext", "ax"
241 start_of_setup:
242 #ifdef SAFE_RESET_DISK_CONTROLLER
243 # Reset the disk controller.
244     movw    $0x0000, %ax            # Reset disk controller
245     movb    $0x80, %dl          # All disks
246     int $0x13 #重置硬盘驱动器
247 #endif
248
249 # Force %es = %ds
250     movw    %ds, %ax
251     movw    %ax, %es
252     cld
```

250-252 行先强制附加段 es 的内容和 ds 相等。

```
253
254 # Apparently some ancient versions of LILO invoked the kernel with %ss != %ds,
255 # which happened to work by accident for the old code.    Recalculate the stack
256 # pointer if %ss is invalid.    Otherwise leave it alone, LOADLIN sets up the
257 # stack behind its own code, so we can't blindly put it directly past the heap.
258
259     movw    %ss, %dx #ax=ds,dx=ss
260     cmpw    %ax, %dx    # %ds == %ss?
261     movw    %sp, %dx
262     je  2f      # -> assume %sp is reasonably set
```

259-262 行的代码比较 ds 和 ss 的值，设置 dx 为栈顶指针的值。如果不等，就建一个新栈。

```
263
```

```
264         # Invalid %ss, make up a new stack
265         movw    $_end, %dx
266         testb   $CAN_USE_HEAP, loadflags
267         jz  1f（一般不跳转）
268         movw    heap_end_ptr, %dx
269 1:      addw    $STACK_SIZE, %dx
```

%dx 为_end+512

```
270         jnc 2f
271         xorw    %dx, %dx    # Prevent wraparound
272
273 2:      # Now %dx should point to the end of our stack space
274         andw    $~3, %dx    # dword align (might as well...)
275         jnz 3f (跳转到 3f)
276         movw    $0xfffc, %dx    # Make sure we're not zero
277 3:      movw    %ax, %ss
278         movzwl  %dx, %esp   # Clear upper half of %esp
279         sti             # Now we should have a working stack


279         sti             # Now we should have a working stack
280
```

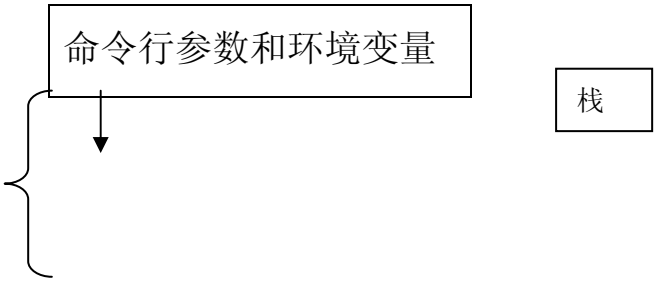
277 行将 ax(保存的 ds)的值，赋值给 ss,

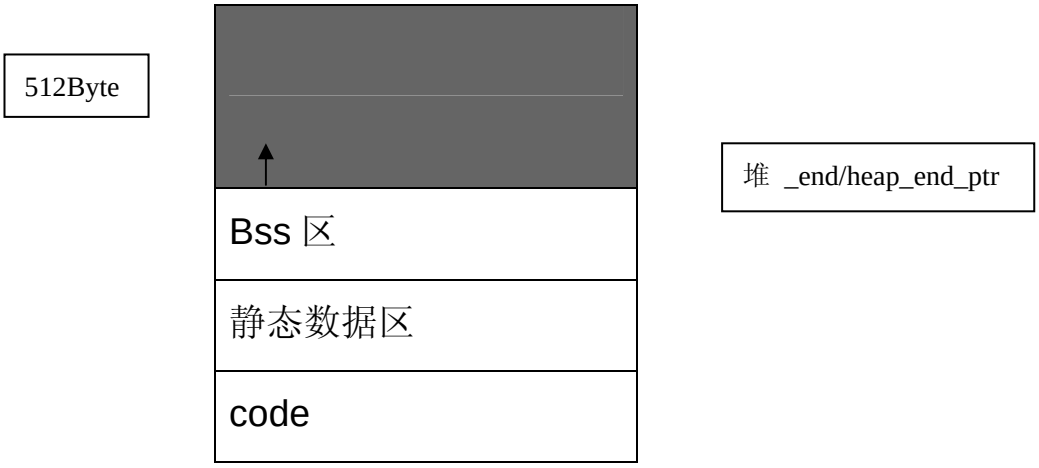
278 行将 dx(_end+512)的值，赋值给 esp，

现在 ss:esp 可以使用啦

上面的代码经过一系列的检测，最终将栈顶指针设置为_end+512

出事化之后的布局图如下所示：






```
281 # We will have entered with %cs = %ds+0x20, normalize %cs so
282 # it is on par with the other segments.
283         pushw   %ds
284         pushw   $6f
285         lretw
```

Grub 跳转到内核代码时，使用的是 ljmp，当时设置的数据段 ds 为 0x9000,跳过来后，cs 被设置为了 0x9020，由于编译时，标号都是相对于 0x90000 的偏移，在 285 行之前执行的都是短跳转没有啥问题，但下面要执行 call main 时（长跳转），如果还是用 cs 原来的值，机 0x9020，

则 call main 将跳转到 0x9020:main 处，而 main 的标号是相对于 0x9000 的，所以现在必须调整 cs 的值。

283-285 行跳转 cs 的值，让其等于 ds，即 0x9000

```
286 6:
```

```
287
288 # Check signature at end of setup
289         cmpl    $0x5a5aaa55, setup_sig
290         jne setup_bad
291
292 # Zero the bss
293         movw    $__bss_start, %di
294         movw    $_end+3, %cx
295         xorl    %eax, %eax
296         subw    %di, %cx
297         shrw    $2, %cx
298         rep; stosl
```

293-298 行初始化 C 中的 bss 区，将其清零。

```
299
300 # Jump to C code (should not return)
301         calll   main
```

经过前面一些列的努力，已经建立好了实模式下的 C 运行环境：栈，堆，bss 区初始化，然后通过 301 行的代码进入 C 运行。

```
302
303 # Setup corrupt somehow...
304 setup_bad:
305         movl    $setup_corrupt, %eax
306         calll   puts
307         # Fall through...
308
309         .globl  die
310         .type   die, @function
311 die:
312         hlt
313         jmp die
```

```
314
315         .size     die, .-die
316
317         .section ".initdata", "a"
318 setup_corrupt:
319         .byte     7
320         .string "No setup signature found...\n"
```

2.2 启动参数准备，内存探测，跳转到保护模式

在 1.1 中，准备好 C 语言的运行环境后，开始执行 arch/x86/boot/main.c 中的代码：

```
128 void main(void)
129 {
130         /* First, copy the boot header into the "zeropage" */
131         copy_boot_params();
132
133         /* End of heap check */
134         init_heap();
135
136         /* Make sure we have all the proper CPU support */
137         if (validate_cpu()) {
138                 puts("Unable to boot - please use a kernel appropriate "
139                      "for your CPU.\n");
140                 die();
141         }
142
143         /* Tell the BIOS what CPU mode we intend to run in. */
144         set_bios_mode();
```

```
145
146         /* Detect memory layout */
147         detect_memory();
148
149         /* Set keyboard repeat rate (why?) */
150         keyboard_set_repeat();
151
152         /* Query MCA information */
153         query_mca();
154
155         /* Query Intel SpeedStep (IST) information */
156         query_ist();
157
158         /* Query APM information */
159 #if defined(CONFIG_APM) || defined(CONFIG_APM_MODULE)
160         query_apm_bios();
161 #endif
162
163         /* Query EDD information */
164 #if defined(CONFIG_EDD) || defined(CONFIG_EDD_MODULE)
165         query_edd();
166 #endif
167
168         /* Set the video mode */
169         set_video();

170
171         /* Parse command line for 'quiet' and pass it to decompressor. */
172         if (cmdline_find_option_bool("quiet"))
173                 boot_params.hdr.loadflags |= QUIET_FLAG;
174
```

```
175        /* Do the last things and invoke protected mode */
176        go_to_protected_mode();
177 }
```

该 main 函数主要执行下列函数

copy_boot_params()：赋值 setup_header 变量

init_heap()：初始化堆

validate_cpu()：检验内核是否支持该 CPU

set_bios_mode()：设置 bios 模式

detect_memory()：探测物理内存布局

query_mca( )：填充系统环境表

query_list( )：填充 ist 信息。

set_video( )：设置视频

go_to_protected_mode：跳转到保护模式

下面只对如下几个比较重要的函数进行分析。

copy_boot_params()：赋值 setup_header 变量

init_heap()：初始化堆

detect_memory()：探测物理内存布局

go_to_protected_mode：跳转到保护模式

2.3 copy_boot_params()

该函数的主要功能是使用 header.S 中的变量 hdr 初始化内核启动参数 boot_params（全局变量）中的 setup_header 变量。

下面看看 boot_params 的定义：

arch/x86/boot/main.c

```
18 struct boot_params boot_params __attribute__((aligned(16)));
```

boot_params 结构体的声明如下：

arch/x86/include/asm/bootparam.h

```
85 struct boot_params {
 86        struct screen_info screen_info;               /* 0x000 */
 87        struct apm_bios_info apm_bios_info;        /* 0x040 */
 88        __u8    _pad2[4];                          /* 0x054 */
 89        __u64    tboot_addr;                       /* 0x058 */
 90        struct ist_info ist_info;                  /* 0x060 */
 91        __u8    _pad3[16];                         /* 0x070 */
 92        __u8    hd0_info[16]; /* obsolete! */      /* 0x080 */
 93        __u8    hd1_info[16]; /* obsolete! */      /* 0x090 */
 94        struct sys_desc_table sys_desc_table;          /* 0x0a0 */
 95        __u8    _pad4[144];                        /* 0x0b0 */
 96        struct edid_info edid_info;                /* 0x140 */
 97        struct efi_info efi_info;                  /* 0x1c0 */
 98        __u32 alt_mem_k;                           /* 0x1e0 */
 99        __u32 scratch;          /* Scratch field! */      /* 0x1e4 */
100        __u8    e820_entries;                      /* 0x1e8 */
101        __u8    eddbuf_entries;                    /* 0x1e9 */
102        __u8    edd_mbr_sig_buf_entries;              /* 0x1ea */
103        __u8    _pad6[6];                          /* 0x1eb */
104        struct setup_header hdr;       /* setup header */    /* 0x1f1 */
105        __u8    _pad7[0x290-0x1f1-sizeof(struct setup_header)];
106        __u32 edd_mbr_sig_buffer[EDD_MBR_SIG_MAX];    /* 0x290 */
107        struct e820entry e820_map[E820MAX];        /* 0x2d0 */
108        __u8    _pad8[48];                         /* 0xcd0 */
109        struct edd_info eddbuf[EDDMAXNR];          /* 0xd00 */
110        __u8    _pad9[276];                        /* 0xeec */
111 } __attribute__((packed));
```

```
29 static void copy_boot_params(void)
30 {
31      struct old_cmdline {
32          u16 cl_magic;
33          u16 cl_offset;
34      };
35      const struct old_cmdline * const oldcmd =
36          (const struct old_cmdline *)OLD_CL_ADDRESS;
37
38      BUILD_BUG_ON(sizeof boot_params != 4096);
39      memcpy(&boot_params.hdr, &hdr, sizeof hdr);
```

39 行：拷贝 hdr 到启动参数中。

```
40
41      if (!boot_params.hdr.cmd_line_ptr &&
42          oldcmd->cl_magic == OLD_CL_MAGIC) {
43          /* Old-style command line protocol. */
44          u16 cmdline_seg;
45
46          /* Figure out if the command line falls in the region
47              of memory that an old kernel would have copied up
48              to 0x90000... */
49          if (oldcmd->cl_offset < boot_params.hdr.setup_move_size)
50              cmdline_seg = ds();
51          else
52              cmdline_seg = 0x9000;
53
54          boot_params.hdr.cmd_line_ptr =
55              (cmdline_seg << 4) + oldcmd->cl_offset;
56      }
```

```
57 }
```

2.4 init_heap

```
109 static void init_heap(void)
110 {
111         char *stack_end;
112
113         if (boot_params.hdr.loadflags & CAN_USE_HEAP) {
114                 asm("leal %P1(%%esp),%0"
115                     : "=r" (stack_end) : "i" (-STACK_SIZE));
116
```

Stack_end = esp-STACK_SIZE=_end

Heap_end = _end + 512

```
117                 heap_end = (char *)
118                     ((size_t)boot_params.hdr.heap_end_ptr + 0x200);
119                 if (heap_end > stack_end)
120                     heap_end = stack_end;
```

然 header.S 中的设置，很显然 119 行的 if 成立，导致 heap_end=stack_end，这是否意味着堆的大小为 0（有待进一步考察）

```
121         } else {
122                 /* Boot protocol 2.00 only, no heap available */
123                 puts("WARNING: Ancient bootloader, some functionality "
124                      "may be limited!\n");
125         }
126 }
```

2.5 detect_memory

该函数由于探测物理内存的布局，这也是内核第一次出现于内存管理相关的代码，这里设置的内容是后续内存管理的基础，内存管理的征程开始啦。

```
122 int detect_memory(void)
123 {
124         int err = -1;
125
126         if (detect_memory_e820() > 0)
127                 err = 0;
128
129         if (!detect_memory_e801())
130                 err = 0;
131
132         if (!detect_memory_88())
133                 err = 0;
134
135         return err;
136 }
```

根据物理内存的类型，探测物理内存的布局。
这里只查看 detect_memory_e820 函数。
20 static int detect_memory_e820(void)

```
 21 {
 22         int count = 0;
 23         struct biosregs ireg, oreg;
 24         struct e820entry *desc = boot_params.e820_map;
 25         static struct e820entry buf; /* static so it is zeroed */
 26
 27         initregs(&ireg);
```

```
28        ireg.ax    = 0xe820;
29        ireg.cx    = sizeof buf;
30        ireg.edx = SMAP;
31        ireg.di    = (size_t)&buf;
32
33        /*
34         * Note: at least one BIOS is known which assumes that the
35         * buffer pointed to by one e820 call is the same one as
36            * the previous call, and only changes modified fields. Therefore,
37         * we use a temporary buffer and copy the results entry by entry.
38         *
39         * This routine deliberately does not try to account for
40         * ACPI 3+ extended attributes.    This is because there are
41         * BIOSes in the field which report zero for the valid bit for
42         * all ranges, and we don't currently make any use of the
43         * other attribute bits.    Revisit this if we see the extended
44         * attribute bits deployed in a meaningful way in the future.
45         */
46
47        do {
48              intcall(0x15, &ireg, &oreg);
49              ireg.ebx = oreg.ebx; /* for next iteration... */
50
51                /* BIOSes which terminate the chain with CF = 1 as opposed
52                   to %ebx = 0 don't always report the SMAP signature on
53                   the final, failing, probe. */
54              if (oreg.eflags & X86_EFLAGS_CF)
55                    break;
56
57              /* Some BIOSes stop returning SMAP in the middle of
```

```
58                  the search loop.    We don't know exactly how the BIOS
59                  screwed up the map at that point, we might have a
60                  partial map, the full map, or complete garbage, so
61                 just return failure. */
62              if (oreg.eax != SMAP) {
63                  count = 0;
64                  break;
65              }
66
67              *desc++ = buf;
68              count++;
69          }
while (ireg.ebx && count < ARRAY_SIZE(boot_params.e820_map));
70
71          return boot_params.e820_entries = count;
72 }
```

在 do while 循环中探测内存区域，用以初始化 boot_params.e820_map 变量。在后面和内存相关的内核代码中，会多次用到该变量。

用以表示一个内存段的数据结构是：struct e820entry，具体定义在 arch/x86/include/asm/e820.h 中。

```
42 #define E820_RAM       1
43 #define E820_RESERVED     2
44 #define E820_ACPI     3
45 #define E820_NVS       4
46 #define E820_UNUSABLE     5
47
48 /* reserved RAM used by kernel itself */
49 #define E820_RESERVED_KERN              128
50
51 #ifndef __ASSEMBLY__
```

```
52 #include <linux/types.h>
53 struct e820entry {
54         __u64 addr; /* start of memory segment */
55         __u64 size; /* size of memory segment */
56         __u32 type; /* type of memory segment */
57 } __attribute__((packed));
58
59 struct e820map {
60         __u32 nr_map;
61         struct e820entry map[E820_X_MAX];
62 }
```

其中 addr 表示该段内存的起始物理地址

size 表示该段内存的大小

type 表示该段内存的内型

2.6 go_to_protected_mode

做好进入保护模式的准备后，在该函数中将进入实模式。

函数在 arch/x86/boot/pm.c 中。

```
104 void go_to_protected_mode(void)
105 {
106         /* Hook before leaving real mode, also disables interrupts */
107         realmode_switch_hook();
108
109         /* Enable the A20 gate */
110         if (enable_a20()) {
111                 puts("A20 gate not responding, unable to boot...\n");
112                 die();
113         }
```

```
114
115         /* Reset coprocessor (IGNNE#) */
116         reset_coprocessor();
117
118         /* Mask all interrupts in the PIC */
119         mask_all_interrupts();
120
121         /* Actual transition to protected mode... */
122         setup_idt();
123         setup_gdt();
124         protected_mode_jump(boot_params.hdr.code32_start,
125                             (u32)&boot_params + (ds() << 4));
126 }
```

该函数调用如下函数：

realmode_switch_hook：禁止可屏蔽和不可屏蔽中断。

enable_a20()：打开 A20 地址线

reset_coprocessor()：对浮点运算协处理器 FPU 初始化

mask_all_interrupt s （）：对 PIC 端口进行设置。

Setup_idt：设置中断向量表

Setup_gdt：设置全局目录表指针

protected_mode_jump：跳转到包含模式

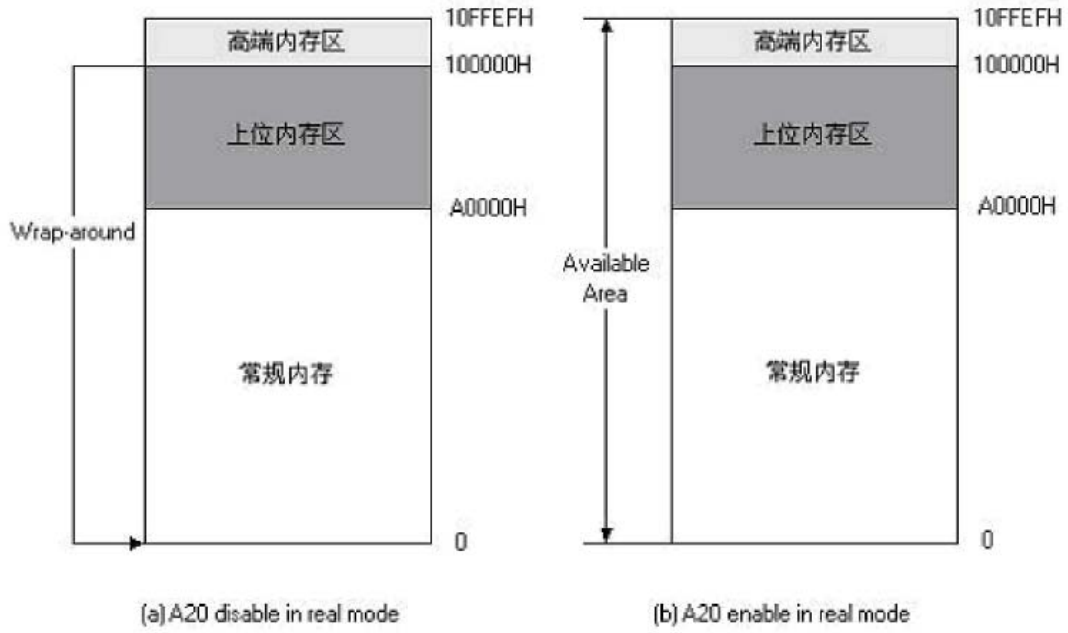
下面主要分析 enable_a20, Setup_idt ,Setup_gdt, protected_mode_jump 函数。

2.6.1 enable_a20

PC 及其兼容机的第 21 根地址线（A20）比较特殊。到了 80286，系统的地址总线有原来的 20 根发展为 24 根，这样能够访问的内存可以达到 2^24=16M。Intel 在设计 80286 时提出的目标是向下兼容。所以，在实模式下，系统所表现的行为应该和 8086/8088 所表现的

完全一样，也就是说，在实模式下，80286 以及后续系列，应该和 8086/8088 完全兼容。

但最终，80286 芯片却存在一个 BUG：因为有了 80286 有 A20 线，如果程序员访问 100000H-10FFEFH 之间的内存，如果程序员访问 100000H-10FFEFH 之间的内存，系统将实际访问这块内存，而不是象 8086/8088 一样从 0 开始。我们来看一副图：



为了解决上述兼容性问题，IBM 使用键盘控制器上剩余的一些输出线来管理第 21 根地址线（从 0 开始数是第 20 根） 的有效性，被称为 A20 Gate：

1 如果 A20 Gate 被打开，则当程序员给出 100000H-10FFEFH 之间的地址的时候，系统将真正访问这块内存区域；

2 如果 A20 Gate 被禁止，则当程序员给出 100000H-10FFEFH 之间的地址的时候，系统仍然使用 8086/8088 的方式即取模方式（8086

仿真）。绝大多数 IBM PC 兼容机默认的 A20 Gate 是被禁止的。现在许多新型 PC 上存在直接通过 BIOS 功能调用来控制 A20 Gate 的功能。

上面所述的内存访问模式都是实模式，在 80286 以及更高系列的 PC 中，即使 A20 Gate 被打开，在实模式下所能够访问的内存最大也只能为 10FFEFH，尽管它们的地址总线所能够访问的能力都大大超过这个限制。为了能够访问 10FFEFH 以上的内存，则必须进入保护模式。

也就是说，打开 A20 的直接原因是：

为程序进入保护模式访问 100000H-10FFEFH 地址做好准备。

下面的函数就是基于上面的原理，来打开 a20。

```
130 int enable_a20(void)
131 {
132             int loops = A20_ENABLE_LOOPS;
133             int kbc_err;
134
135             while (loops--) {
136                     /* First, check to see if A20 is already enabled
137                    (legacy free, etc.) */
138                     if (a20_test_short())
139                             return 0;
140
141                     /* Next, try the BIOS (INT 0x15, AX=0x2401) */
142                     enable_a20_bios();
143                     if (a20_test_short())
```

```
144                     return 0;
145
146             /* Try enabling A20 through the keyboard controller */
147             kbc_err = empty_8042();
148
149             if (a20_test_short())
150                     return 0; /* BIOS worked, but with delayed reaction
*/
151
152             if (!kbc_err) {
153                     enable_a20_kbc();
154                     if (a20_test_long())
155                             return 0;
156             }
157
158             /* Finally, try enabling the "fast A20 gate" */
159             enable_a20_fast();
160             if (a20_test_long())
161                     return 0;
162     }
163
164     return -1;
165 }
```

2.6.2 Setup_idt

用该函数来设置中断描述符表。

```
61 struct gdt_ptr {
 62     u16 len;
 63     u32 ptr;
 64 } __attribute__((packed));
```

```
95 static void setup_idt(void)
 96 {
 97         static const struct gdt_ptr null_idt = {0, 0};
 98         asm volatile("lidtl %0" : : "m" (null_idt));
 99 }
```

Null_idt 初始化一个默认的中断描述符表，len 为 0，ptr 为 NULL，然后将此赋值给 idtl 寄存器。

2.6.3 第一次初始化gdt （setup_gdt）

```
  6 #define GDT_ENTRY(flags, base, limit)                   \
  7         ((((base)  & 0xff000000ULL) << (56-24)) |     \
  8          (((flags) & 0x0000f0ffULL) << 40) |             \
  9          (((limit) & 0x000f0000ULL) << (48-16)) |     \
 10          (((base)  & 0x00ffffffULL) << 16) |             \
 11          (((limit) & 0x0000ffffULL)))
 12
 13 /* Simple and small GDT entries for booting only */
 14
 15 #define GDT_ENTRY_BOOT_CS     2
 16 #define __BOOT_CS           (GDT_ENTRY_BOOT_CS * 8)
 17
 18 #define GDT_ENTRY_BOOT_DS     (GDT_ENTRY_BOOT_CS + 1)
 19 #define __BOOT_DS           (GDT_ENTRY_BOOT_DS * 8)
 20
 21 #define GDT_ENTRY_BOOT_TSS    (GDT_ENTRY_BOOT_CS + 2)
 22 #define __BOOT_TSS          (GDT_ENTRY_BOOT_TSS * 8)
```

```
66 static void setup_gdt(void)
```

```
67 {
68      /* There are machines which are known to not boot with the GDT
69           being 8-byte unaligned.  Intel recommends 16 byte
alignment. */
70      static const u64 boot_gdt[] __attribute__((aligned(16))) = {
71          /* CS: code, read/execute, 4 GB, base 0 */
72              [GDT_ENTRY_BOOT_CS] = GDT_ENTRY(0xc09b, 0,
0xfffff),
73          /* DS: data, read/write, 4 GB, base 0 */
74              [GDT_ENTRY_BOOT_DS] = GDT_ENTRY(0xc093, 0,
0xfffff),
75          /* TSS: 32-bit tss, 104 bytes, base 4096 */
76          /* We only have a TSS here to keep Intel VT happy;
77              we don't actually use it for anything. */
78          [GDT_ENTRY_BOOT_TSS] = GDT_ENTRY(0x0089, 4096,
103),
79      };
80      /* Xen HVM incorrectly stores a pointer to the gdt_ptr, instead
81          of the gdt_ptr contents.  Thus, make it static so it will
82          stay in memory, at least long enough that we switch to the
83          proper kernel GDT. */
84      static struct gdt_ptr gdt;
85
86      gdt.len = sizeof(boot_gdt)-1;
87      gdt.ptr = (u32)&boot_gdt + (ds() << 4);
88
89      asm volatile("lgdtl %0" : : "m" (gdt));
90 }
```

2.6.4 第一次启动保护模式(protected_mode_jump)

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

该函数接受两个变量，一个是保护模式的起始地址，一个是 boot_params 的线性地址（之所以要是线性地址，因为保护模式下的寻址方式已经发生改变）。

该函数所在的文件是 arch/x86/boot/pmjump.S

```
20      .text
21      .code16
22
23 /*
24   * void protected_mode_jump(u32 entrypoint, u32 bootparams);
25   */
26 GLOBAL(protected_mode_jump)
27      movl    %edx, %esi      # Pointer to boot_params table
```

27 行：将启动参数的地址保存到 esi 中。

```
28
29      xorl    %ebx, %ebx
30      movw    %cs, %bx
31      shll    $4, %ebx
32      addl    %ebx, 2f
```

29-32 行：将 2f 出的线性地址保存到标号 2 出的变量 in_pm32 中。

```
33      jmp 1f              # Short jump to serialize on 386/486
34 1:
35
36      movw    $__BOOT_DS, %cx
37      movw    $__BOOT_TSS, %di
38
39      movl    %cr0, %edx
40      orb $X86_CR0_PE, %dl    # Protected mode
```

```
41        movl      %edx, %cr0
```

开启 cr0 中的 PE 位，启动保护模式。从 41 行开始之后，就进入保护模式啦。

```
42
43        # Transition to 32-bit mode
44        .byte     0x66, 0xea          # ljmpl opcode
45 2:    .long     in_pm32             # offset
46        .word     __BOOT_CS           # segment
```

在保护模式下，跳转到 in_pm32 下执行。

```
47 ENDPROC(protected_mode_jump)
48
49        .code32
50        .section ".text32","ax"
51 GLOBAL(in_pm32)
52        # Set up data segments for flat 32-bit mode
53        movl      %ecx, %ds
54        movl      %ecx, %es
55        movl      %ecx, %fs
56        movl      %ecx, %gs
57        movl      %ecx, %ss
```

%ecx 为__BOOT_DS,初始化 ds,es,fs,gs,ss 为__BOOT_DS

```
58        # The 32-bit code sets up its own stack, but this way we do have
59        # a valid stack if some debugging hack wants to use it.
60        addl      %ebx, %esp
61
62        # Set up TR to make Intel VT happy
63        ltr %di
```

设这 tss 相关的 trr 寄存器。

```
64
65      # Clear registers to allow for future extensions to the
66      # 32-bit boot protocol
67      xorl    %ecx, %ecx
68      xorl    %edx, %edx
69      xorl    %ebx, %ebx
70      xorl    %ebp, %ebp
71      xorl    %edi, %edi
72
73      # Set up LDTR to make Intel VT happy
74      lldt    %cx
```

设置 ldt 寄存器

```
75
76      jmpl    *%eax           # Jump to the 32-bit entrypoint
```

跳转到保护模式地址（0x100000）

```
77 ENDPROC(in_pm32)
```

3. 保护模式（加载基址为 0x100000）

保护模式下最先执行的代码在 arch/x86/boot/compressed/head_32.S，该模块主要的功能是解压缩内核，然后将控制权交给解压缩后的内核。

```
33      __HEAD
 34 ENTRY(startup_32)
 35      cld
```

```
36      /*
37       * Test KEEP_SEGMENTS flag to see if the bootloader is asking
38       * us to not reload segments
39       */
40      testb   $(1<<6), BP_loadflags(%esi)
41      jnz 1f
42
43      cli
44      movl    $__BOOT_DS, %eax
45      movl    %eax, %ds
46      movl    %eax, %es
47      movl    %eax, %fs
48      movl    %eax, %gs
49      movl    %eax, %ss
```

初始化段寄存器

```
50 1:
51
52 /*
53  * Calculate the delta between where we were compiled to run
54  * at and where we were actually loaded at.  This can only be done
55  * with a short local call on x86.  Nothing  else will tell us what
56  * address we are running at.  The reserved chunk of the real-mode
57  * data at 0x1e4 (defined as a scratch field) are used as the stack
58  * for this calculation. Only 4 bytes are needed.
59  */
60      leal    (BP_scratch+4)(%esi), %esp
```

60 行：esi 中存放的是 boot_params 的首地址，60 行的作用是将 BP_scratch 字段的末地址放入到 esp 中。

```
61      call    1f # push eip,jmp 1f
```

```
62 1:   popl    %ebp
63      subl    $1b, %ebp
```

62 行：将 1 出的地址存放到 ebp 中

63 行：用 ebp 减去标号 1 处的偏移，得到内核在内存中的起始地址。

```
64
65 /*
66  * %ebp contains the address we are loaded at by the boot loader and %ebx
67  * contains the address where we should move the kernel image temporarily
68  * for safe in-place decompression.
69  */
70
71 #ifdef CONFIG_RELOCATABLE
72      movl    %ebp, %ebx
73      movl    BP_kernel_alignment(%esi), %eax
```

Boot_params 偏移为 BP_kernel_alignment 出的值 0x1000000（16M）赋值给 eax

```
74      decl    %eax
75      addl    %eax, %ebx
76      notl    %eax
77      andl    %eax, %ebx
78 #else
79      movl    $LOAD_PHYSICAL_ADDR, %ebx
80 #endif
```

经过 74-80 行的代码，ebx 中保存的就是内核需要被拷贝的地址。

```
81
82      /* Target address to relocate to for decompression */
```

```
 83        addl    $z_extract_offset, %ebx
 84
 85        /* Set up the stack */
 86        leal    boot_stack_end(%ebx), %esp
 87
 88        /* Zero EFLAGS */
 89        pushl   $0
 90        popfl
 91
 92 /*
 93    * Copy the compressed kernel to the end of our buffer
 94    * where decompression in place becomes safe.
 95    */
 96        pushl   %esi
 97        leal    (_bss-4)(%ebp), %esi
 98        leal    (_bss-4)(%ebx), %edi
 99        movl    $(_bss - startup_32), %ecx
100        shrl    $2, %ecx
101        std
102        rep movsl
103        cld
104        popl    %esi
```

第 96-104 行：将内核拷贝到 ebx 所指向的地址。

```
105
106 /*
107    * Jump to the relocated address.
108    */
109        leal    relocated(%ebx), %eax
110        jmp *%eax
```

109-110 行：跳转到拷贝后内核处的 relocated。

```
111 ENDPROC(startup_32)
112
113         .text
114 relocated:
115
116 /*
117  * Clear BSS (stack is currently empty)
118  */
119         xorl    %eax, %eax
120         leal    _bss(%ebx), %edi
121         leal    _ebss(%ebx), %ecx
122         subl    %edi, %ecx
123         shrl    $2, %ecx
124         rep stosl
```

119-124：清空 bss 段

```
125
126 /*
127  * Do the decompression, and jump to the new kernel..
128  */
129         leal    z_extract_offset_negative(%ebx), %ebp
130                                 /* push arguments for decompress_kernel: */
131         pushl   %ebp            /* output address */
132         pushl   $z_input_len    /* input_len */
133         leal    input_data(%ebx), %eax
134         pushl   %eax            /* input_data */
135         leal    boot_heap(%ebx), %eax
136         pushl   %eax            /* heap area */
137         pushl   %esi            /* real mode pointer */
138         call    decompress_kernel
```

129-138 行：将 decompress_kernel 所需要的参数入栈，然后调

用 decompress_kernel 解压内核。

decompress_kernel 函数就不深入进去了，其作用是将内核解压到 ebp 所在的内存区。

```
139         addl       $20, %esp
140
141 #if CONFIG_RELOCATABLE
142 /*
143    * Find the address of the relocations.
144    */
145         leal       z_output_len(%ebp), %edi
146
147 /*
148    * Calculate the delta between where vmlinux was compiled to run
149    * and where it was actually loaded.
150    */
151         movl       %ebp, %ebx
152         subl       $LOAD_PHYSICAL_ADDR, %ebx
153         jz    2f    /* Nothing to be done if loaded at compiled addr. */
154 /*
155    * Process relocations.
156    */
157
158 1:    subl       $4, %edi
159         movl       (%edi), %ecx
160         testl     %ecx, %ecx
161         jz    2f
162         addl       %ebx, -__PAGE_OFFSET(%ebx, %ecx)
163         jmp 1b
164 2:
165 #endif
```

```
166
167 /*
168   * Jump to the decompressed kernel.
169   */
170       xorl    %ebx, %ebx
171       jmp *%ebp
```

170-171 行：开始解压缩后的第一条代码。

```
172
173 /*
174   * Stack and heap for uncompression
175   */
176       .bss
177       .balign 4
178 boot_heap:
179       .fill BOOT_HEAP_SIZE, 1, 0
180 boot_stack:
181       .fill BOOT_STACK_SIZE, 1, 0
182 boot_stack_end:
```

4. 保护模式(加载基址 0xc0100000）

这是内核解压后的保护模式，开始执行的代码在文件 arch/x86/kernel/head_32.S。

要注意，解压缩后的内核在编译时的链接基地址为 0xc0100000，这就导致在实模式中很多与保护模式同名的变量必须重新初始化。

4.1 第二次启动保护模式

```
85 ENTRY(startup_32)
86      /* test KEEP_SEGMENTS flag to see if the bootloader is asking
87           us to not reload segments */
88      testb $(1<<6), BP_loadflags(%esi)
89      jnz 2f
90
91 /*
92  * Set segments to known values.
93  */
94      lgdt pa(boot_gdt_descr)
```

重新初始化全局段描述符

```
95      movl $(__BOOT_DS),%eax
96      movl %eax,%ds
97      movl %eax,%es
98      movl %eax,%fs
99      movl %eax,%gs
```

初始化段寄存器

```
100 2:
101
102 /*
103  * Clear BSS first so that there are no surprises...
104  */
105     cld
106     xorl %eax,%eax
107     movl $pa(__bss_start),%edi
108     movl $pa(__bss_stop),%ecx
109     subl %edi,%ecx
110     shrl $2,%ecx
```

```
111         rep ; stosl
```

初始化内核 bss 段

```
112 /*
113  * Copy bootup parameters out of the way.
114  * Note: %esi still has the pointer to the real-mode data.
115  * With the kexec as boot loader, parameter segment might be loaded beyond
116  * kernel image and might not even be addressable by early boot page tables.
117  * (kexec on panic case). Hence copy out the parameters before initializing
118  * page tables.
119  */
120         movl $pa(boot_params),%edi
121         movl $(PARAM_SIZE/4),%ecx
122         cld
123         rep
```

esi 指向实模式下的 boot_params,

edi 指向保护模式下的 boot_params,

120-123 处将实模式下的 boot_params 拷贝到保护模式下，为保护模式下通过 boot_params 变量访问启动参数做好准备。

需要注意的是，虽然实模式下和保护模式下启动参数都叫 boot_params，但是它们是两个不同的变量，在实模式下的定义在 arch/x86/boot/main.c 中

```
18 struct boot_params boot_params __attribute__((aligned(16)));
```

在保护模式下的定义在 arch/x86/kernel/setup.c 中。

```
144 #ifndef CONFIG_DEBUG_BOOT_PARAMS
```

```
145 struct boot_params __initdata boot_params;
146 #else
147 struct boot_params boot_params;
148 #endif
```

```
124         movsl
125         movl pa(boot_params) + NEW_CL_POINTER,%esi
126         andl %esi,%esi
127         jz 1f                   # No comand line
128         movl $pa(boot_command_line),%edi
129         movl $(COMMAND_LINE_SIZE/4),%ecx
130         rep
131         movsl
```

4.2 第一次启动分页管理

```
227 page_pde_offset = (__PAGE_OFFSET >> 20);
228
229         movl $pa(__brk_base), %edi
230         movl $pa(swapper_pg_dir), %edx
231         movl $PTE_IDENT_ATTR, %eax //eax = 0x003
232 10:
233         leal PDE_IDENT_ATTR(%edi),%ecx          /* Create PDE entry
*/
234         movl %ecx,(%edx)                    /* Store identity PDE entry */
235         movl  %ecx,page_pde_offset(%edx)        /*  Store  kernel  PDE
entry */
```

page_pde_offset 为 768，_brk_base 作为 BRK 段的起始地址。

229 行将其物理地址复制给 edi。

swapper_pg_dir 指向页目录表的起始地址。其定义如下：

```
617 #ifdef CONFIG_X86_PAE
618 swapper_pg_pmd:
619         .fill 1024*KPMDS,4,0
620 #else
621 ENTRY(swapper_pg_dir)
622         .fill 1024,4,0
623 #endif
```

230 行将其地址赋值给 edx。

234 行将__brk_base 的起始地址保存到 swapper_pg_dir 所指向的页目录中（用户页目录项的第一项）第一项。

235 行将__brk_base 的起始地址保存到 swapper_pg_dir 所指向的页目录中的第 768 项（内核页目录项的第一项）

内核启动时运行在内核态，这里为啥要在用户态的目录项中设置与内核态对应的项呢？答案是为了内核由用户空间向系统空间的平稳过渡。

```
236         addl $4,%edx
237         movl $1024, %ecx
238 11:
239         stosl //将 eax 中的值保存到 es:edi 指向的内存中，eax 的初始值 0x003
240         addl $0x1000,%eax //eax 增加 4k
241         loop 11b
```

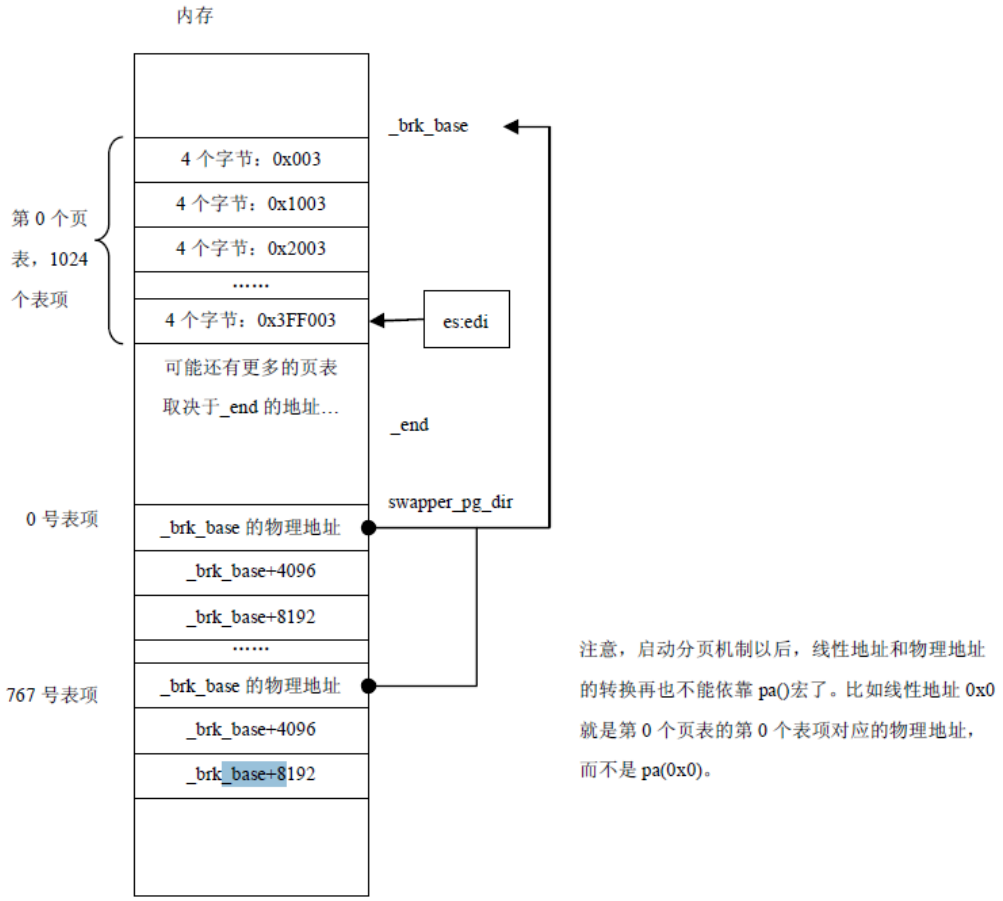
239-241 行：初始化内核的第一个页表。

```
242     /*
243          * End condition: we must map up to the end + MAPPING_BEYOND_END.
244      */
245          movl $pa(_end) + MAPPING_BEYOND_END + PTE_IDENT_ATTR, %ebp
246     cmpl %ebp,%eax
247     jb 10b
```

将内核代码终止地址保存到 ebp 中。比较 eax 是否小于 ebp，如果 eax 小于 ebp，说明内核代码还没有全部映射到页表项中，跳转到标号 10 处，继续循环。

```
248     addl $__PAGE_OFFSET, %edi
249     movl %edi, pa(_brk_end) //从新保存_brk_end 的值
250     shrl $12, %eax
251     movl  %eax,  pa(max_pfn_mapped)  //设置内核最大映射页面数量
252
253     /* Do early initialization of the fixmap area */
254     movl $pa(swapper_pg_fixmap)+PDE_IDENT_ATTR,%eax
255     movl %eax,pa(swapper_pg_dir+0xffc)
256 #endif
257     jmp 3f
```

上面代码运行完后，页表项对应的关系见下图：




```
329   * Enable paging
330   */
331         movl $pa(swapper_pg_dir),%eax
332         movl %eax,%cr3          /* set the page table pointer.. */
333         movl %cr0,%eax
334         orl    $X86_CR0_PG,%eax
335         movl %eax,%cr0          /* ..and set paging (PG) bit */
```

331-332 行将页目录表的物理地址保存到 cr3 中。

333-335 行设置 cr0 寄存器中的分页位。

从 336 行开始这进入到了分页时代。

```
336         ljmp $__BOOT_CS,$1f /* Clear prefetch and normalize %eip */
```

336 行是进入保护模式的关键一行，（在执行 336 行时，内核寻址到 336 行的代码的 eip 仍然是在用户空间寻址到的，所以需要在用户空间的目录项中设置与内核相同的表项，要不 335 行运行完毕，分页机制已启动，而此时 eip 是在运行 335 行的 eip 的基础上加上了一个很小的偏移，很显然，此时 eip 仍在用户空间，用用户空间的 eip 去分页机制中转换地址，如果用户空间的页目录项没有设置，很显然就无法寻址到 336 行的物理地址，所以 335 行到 336 行的执行是一个临界点）。启动分页机制之前（336 行之前），eip 中保存的线性地址就是物理地址(在用户空间),也就是说其地址小于 0xc0000000。

336 行执行时，__BOOT_CS => cs,$1f 的标号地址（该地址已在系统空间）=>eip，此时 eip 的地址由用户空间过渡到了系统空间。

4.3 初始化 0 号进程

```
337 1:
338         /* Set up the stack pointer */
339         lss stack_start,%esp
```

339 行加载堆栈段。

```
657 ENTRY(stack_start)
658         .long init_thread_union+THREAD_SIZE
659         .long __BOOT_DS
```

下面看看一个内核线程的核心数据结构 thread_info：

```
26 struct thread_info {
 27         struct task_struct    *task;          /* main task structure */
 28         struct exec_domain    *exec_domain;     /* execution domain */
 29         __u32                 flags;          /* low level flags */
```

```
30      __u32                   status;         /* thread synchronous flags */
31      __u32                   cpu;            /* current CPU */
32      int             preempt_count;  /* 0 => preemptable,
33                                          <0 => BUG */
34      mm_segment_t            addr_limit;
35      struct restart_block    restart_block;
36      void __user         *sysenter_return;
37 #ifdef CONFIG_X86_32
38          unsigned long                   previous_esp;   /* ESP of the
previous stack in
39                                          case of nested (IRQ) stacks
40                                      */
41      __u8                    supervisor_stack[0];
42 #endif
43      int             uaccess_err;
44 };
```

Thread_info 中有一个指针 task，指向表示进程的 task_struct.

init_thread_union 的定义在

arch/x86/kernel/init_task.c 中：

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

INIT_THREAD_INFO 定义在 arch/x86/include/asm/thread_info.h 中：

```
46 #define INIT_THREAD_INFO(tsk)                   \
 47 {                                      \
 48      .task           = &tsk,                 \
 49      .exec_domain    = &default_exec_domain, \
```

```
50      .flags          = 0,                    \
51      .cpu            = 0,                    \
52      .preempt_count  = INIT_PREEMPT_COUNT,   \
53      .addr_limit = KERNEL_DS,                \
54      .restart_block = {                      \
55              .fn = do_no_restart_syscall,    \
56      },                                      \
57 }
```

0 号进程的 thread_info 和 task_struct 在编译时一部分初始化，esp 指向 thread_info 起始地方偏移为 8K 的地址。

4.4 初始化中断描述符表

```
357  * start system 32-bit setup. We need to re-do some of the things done
358  * in 16-bit mode for the "real" operations.
359  */
360     call setup_idt
```

360 行调用 setup_idt，初始化中断描述表

```
502 setup_idt:
503     lea ignore_int,%edx
504     movl $(__KERNEL_CS << 16),%eax
505     movw %dx,%ax            /* selector = 0x0010 = cs */
506     movw $0x8E00,%dx        /* interrupt gate - dpl=0, present */
```

503：将 ignore_int 的地址赋值给 edx。

504：将 0x100000 赋值给 eax

505：将 ignore_int 的低 16 为赋值给 ax，此时 eax 中的高 16 为 0x0010，低 16 为 ignore_int 地址的低 16 位。

```
575 ignore_int:
576         cld
577 #ifdef CONFIG_PRINTK
578         pushl %eax
579         pushl %ecx
580         pushl %edx
581         pushl %es
582         pushl %ds
583         movl $(__KERNEL_DS),%eax
584         movl %eax,%ds
585         movl %eax,%es
586         cmpl $2,early_recursion_flag
587         je hlt_loop
588         incl early_recursion_flag
589         pushl 16(%esp)
590         pushl 24(%esp)
591         pushl 32(%esp)
592         pushl 40(%esp)
593         pushl $int_msg
594         call printk
595
596         call dump_stack
597
598         addl $(5*4),%esp
599         popl %ds
600         popl %es
601         popl %edx
602         popl %ecx
603         popl %eax
604 #endif
605         iret
```

ignore_int 函数很简单，就是在 early_recursion_flag 不为 2 的情况下，调用 printk 内核打印函数，打印 int_msg 信息：

```
666int_msg:
667 .asciz "Unknown interrupt or fault at: %p %p %p\n"
```

```
508     lea idt_table,%edi
509     mov $256,%ecx
```

508 行将 idt_table 的地址保存到 edi 中。

509 行将 ecx 赋值为 256。

Idt_table 在 arch/x86/include/asm/desc.h 中定义：

Idt_table 表示一个数组，数组的每一项表示一个门描述符，具体定义将 desc_struct。

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

```
typedef gate_desc desc_struct
22 struct desc_struct {
 23     union {
 24         struct {
 25             unsigned int a;
 26             unsigned int b;
 27         };
 28         struct {
 29             u16 limit0;
 30             u16 base0;
 31             unsigned base1: 8, type: 4, s: 1, dpl: 2, p: 1;
 32             unsigned limit: 4, avl: 1, l: 1, d: 1, g: 1, base2: 8;
```

```
33              };
34      };
35 } __attribute__((packed));
```

```
510 rp_sidt:
511     movl %eax,(%edi)
512     movl %edx,4(%edi)
513     addl $8,%edi
514     dec %ecx
515     jne rp_sidt
```

511,512 行，中断门描述符表的第一项前 2 个字节设置为 ignore_int 地址的低 16 位，中间 2 字节被设置为段选择在 0x0010, 后四个字节低端两个字节被设置为$0x8e00，高端两个字节被设置为 ingore_int 的高 16 为地址。

513 到 515 行循环设置 256 个这样的门描述符，每个内容都一样，都是调用 ignore_int 作为中断服务程序。

```
517 .macro    set_early_handler handler,trapno
```

```
518         lea \handler,%edx
519         movl $(__KERNEL_CS << 16),%eax
520         movw %dx,%ax
521         movw $0x8E00,%dx       /* interrupt gate - dpl=0, present */
522         lea idt_table,%edi
523         movl %eax,8*\trapno(%edi)
524         movl %edx,8*\trapno+4(%edi)
525 .endm
526
527         set_early_handler handler=early_divide_err,trapno=0
528         set_early_handler handler=early_illegal_opcode,trapno=6
529         set_early_handler handler=early_protection_fault,trapno=13
530         set_early_handler handler=early_page_fault,trapno=14
531
532         ret
```

517 到 532 行的代码改变中断/异常处理函数，将 0/6/13/14 号中断描述表设置成相应的处理函数，这些函数分别是

early_divide_err

early_illegal_opcode

early_protection_fault

early_page_fault

上述四个函数又调用 early_fault 函数。

```
534 early_divide_err:
535         xor %edx,%edx
536         pushl $0       /* fake errcode */
537         jmp early_fault
538
539 early_illegal_opcode:
540         movl $6,%edx
541         pushl $0       /* fake errcode */
```

```
542         jmp early_fault
543
544 early_protection_fault:
545         movl $13,%edx
546         jmp early_fault
547
548 early_page_fault:
549         movl $14,%edx
550         jmp early_fault
551
552 early_fault:
553         cld
554 #ifdef CONFIG_PRINTK
555         pusha
556         movl $(__KERNEL_DS),%eax
557         movl %eax,%ds
558         movl %eax,%es
559         cmpl $2,early_recursion_flag
560         je hlt_loop
561         incl early_recursion_flag
562         movl %cr2,%eax
563         pushl %eax
564         pushl %edx          /* trapno */
565         pushl $fault_msg
566         call printk
567 #endif
568         call dump_stack
569 hlt_loop:
570         hlt
571         jmp hlt_loop
```

4.5 第三次启动保护模式

设置好中断向量表后，内核必须从新加载 idt 寄存器，才能使用它。

```
418 is386:    movl $2,%ecx            # set MP
419 2:    movl %cr0,%eax
420        andl $0x80000011,%eax     # Save PG,PE,ET
421        orl %ecx,%eax
422        movl %eax,%cr0
423
424        call check_x87
425        lgdt early_gdt_descr
```

重新加载一个全局描述符表

```
#define GDT_ENTRIES 32
707 ENTRY(early_gdt_descr)
708        .word GDT_ENTRIES*8-1
709        .long gdt_page              /* Overwritten for secondary CPUs */
```

```
36 struct gdt_page {
 37        struct desc_struct gdt[GDT_ENTRIES];
 38 } __attribute__((aligned(PAGE_SIZE)));
```

```
 22 struct desc_struct {
  23        union {
  24                struct {
  25                        unsigned int a;
  26                        unsigned int b;
  27                };
```

```
28              struct {
29                      u16 limit0;
30                      u16 base0;
31                      unsigned base1: 8, type: 4, s: 1, dpl: 2, p: 1;
32                      unsigned limit: 4, avl: 1, l: 1, d: 1, g: 1, base2: 8;
33              };
34      };
35 } __attribute__((packed));
```

重新加载一个全局描述符表？因为现在起到分页了，要从新定位全局描述符表的位置。GDT 的第三次设置是在开启并设置了页面寻址之后进行的，所以本身很简单，也是最终的设置。

GDT 的初始化在 arch/x86/kernel/cpu/common.c 中。

```
86 DEFINE_PER_CPU_PAGE_ALIGNED(struct gdt_page, gdt_page) = { .gdt = {
87 #ifdef CONFIG_X86_64
88         /*
89          * We need valid kernel segments for data and code in long mode too
90          * IRET will check the segment types  kkeil 2000/10/28
91          * Also sysret mandates a special GDT layout
92          *
93          * TLS descriptors are currently at a different place compared to i386.
94          * Hopefully nobody expects them at a fixed place (Wine?)
95          */
96         [GDT_ENTRY_KERNEL32_CS]         = GDT_ENTRY_INIT(0xc09b, 0, 0xfffff),
97         [GDT_ENTRY_KERNEL_CS]           = GDT_ENTRY_INIT(0xa09b, 0, 0xfffff),
98         [GDT_ENTRY_KERNEL_DS]           = GDT_ENTRY_INIT(0xc093, 0, 0xfffff),
99         [GDT_ENTRY_DEFAULT_USER32_CS]     = GDT_ENTRY_INIT(0xc0fb, 0, 0xfffff),
100        [GDT_ENTRY_DEFAULT_USER_DS] = GDT_ENTRY_INIT(0xc0f3, 0, 0xfffff),
101        [GDT_ENTRY_DEFAULT_USER_CS] = GDT_ENTRY_INIT(0xa0fb, 0, 0xfffff),
102 #else
```

```
103     [GDT_ENTRY_KERNEL_CS]           = GDT_ENTRY_INIT(0xc09a, 0, 0xfffff),
104     [GDT_ENTRY_KERNEL_DS]           = GDT_ENTRY_INIT(0xc092, 0, 0xfffff),
105     [GDT_ENTRY_DEFAULT_USER_CS] = GDT_ENTRY_INIT(0xc0fa, 0, 0xfffff),
106     [GDT_ENTRY_DEFAULT_USER_DS] = GDT_ENTRY_INIT(0xc0f2, 0, 0xfffff),
107     /*
108      * Segments used for calling PnP BIOS have byte granularity.
109      * They code segments and data segments have fixed 64k limits,
110      * the transfer segment sizes are set at run time.
111      */
112     /* 32-bit code */
113     [GDT_ENTRY_PNPBIOS_CS32]       = GDT_ENTRY_INIT(0x409a, 0, 0xffff),
114     /* 16-bit code */
115     [GDT_ENTRY_PNPBIOS_CS16]       = GDT_ENTRY_INIT(0x009a, 0, 0xffff),
116     /* 16-bit data */
117     [GDT_ENTRY_PNPBIOS_DS]         = GDT_ENTRY_INIT(0x0092, 0, 0xffff),
118     /* 16-bit data */
119     [GDT_ENTRY_PNPBIOS_TS1]        = GDT_ENTRY_INIT(0x0092, 0, 0),
120     /* 16-bit data */
121     [GDT_ENTRY_PNPBIOS_TS2]        = GDT_ENTRY_INIT(0x0092, 0, 0),
122     /*
123      * The APM segments have byte granularity and their bases
124      * are set at run time.    All have 64k limits.
125      */
126    /* 32-bit code */
127     [GDT_ENTRY_APMBIOS_BASE]       = GDT_ENTRY_INIT(0x409a, 0, 0xffff),
128     /* 16-bit code */
129     [GDT_ENTRY_APMBIOS_BASE+1]    = GDT_ENTRY_INIT(0x009a, 0, 0xffff),
130     /* data */
131     [GDT_ENTRY_APMBIOS_BASE+2]    = GDT_ENTRY_INIT(0x4092, 0, 0xffff),
132
133     [GDT_ENTRY_ESPFIX_SS]           = GDT_ENTRY_INIT(0xc092, 0, 0xfffff),
```

```
134     [GDT_ENTRY_PERCPU]      = GDT_ENTRY_INIT(0xc092, 0, 0xfffff),
135     GDT_STACK_CANARY_INIT
136 #endif
137 } };
```

```
426     lidt idt_descr
```

重现加载 idt。

```
427     ljmp $(__KERNEL_CS),$1f
428 1:  movl $(__KERNEL_DS),%eax      # reload all the segment registers
429     movl %eax,%ss                 # after changing gdt.
430
431     movl $(__USER_DS),%eax        # DS/ES contains default USER segment
432     movl %eax,%ds
433     movl %eax,%es
434
435     movl $(__KERNEL_PERCPU), %eax
436     movl %eax,%fs                 # set this cpu's percpu
```

在 gdt 改变后，重现加载所有的段寄存器。

知道 Linux x86 的分段管理是通过 GDTR 来实现的，那么现在就来总结一下 Linux 启动以来到现在，共设置了几次 GDTR：

1．第一次还是 cpu 处于实模式的时候，运行 arch\x86\boot\pm.c 下 setup_gdt()函数的代码。该函数，设置了两个 GDT 项，一个是代码段可读/执行的，另一个是数据段可读写的，都是从 0-4G 直接映射到 0-4G，也就是虚拟地址和线性地址相等。

2．第二次是在内核解压缩以后，用解压缩后的内核代码 arch\x86\kernel\head_32.S 再次对 gdt 进行设置，这一次的设置效

果和上一次是一样的。

3．第三次同样是在 arch\x86\kernel\head_32.S 中，只不过是在开启了页面寻址之后，通过分页寻址得到编译好的全局描述符表 gdt 的地址。这一次效果就跟前两次不一样了，为内核最终使用的全局描述符表，同时也设置了 IDT。

4.6 跳转到start_kernel

```
469         jmp *(initial_code)
```

```
609 ENTRY(initial_code)
610         .long i386_start_kernel
```

469 行跳转到 start_kernel 函数。

5. 参考书籍

[1] linux 内核 2.6.34 源码

[2] 深入了解 linux 内核 （第三版）

[3] linxu 内核源代码情景分析 毛德超

[4] 深入 linxu 内核架构

[5] http://blog.csdn.net/yunsongice(网络资源)