Linux操作系统分析

2 基于x86的Linux启动代码分析

陈香兰（xlanchen@ustc.edu.cn）

计算机应用教研室@计算机学院
嵌入式系统实验室@苏州研究院
中国科学技术大学

Fall 2010

Outline

源代码来源

- 内核版本：2.6.26
- ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.26.tar.gz
  - 解压缩后，成功编译一次
  - 建立Source Insight工程
    - Windows+Source Insight
    - Wine+Source Insight
      安装wine：sudo apt-get install wine
      在wine中安装SourceInsight：wine XXX.exe

基于x86的Linux启动代码分析

- 了解linux的源码组织
  - 看目录结构
- 了解linux的内核代码结构
  - 看Makefile
- 了解基于x86的linux的boot image的结构
  - 看Makefile文件和链接描述文件
- 掌握x86的启动流程
  - 阅读启动源码文件

Linux-2.6.26

- 阅读linux目录下的README
  - 关于Linux的介绍（WHAT IS LINUX?）
  - 该版本内核支持的体系结构（ON WHAT HARDWARE DOES IT RUN?）
  - Linux源代码目录中的文档目录（DOCUMENTATION）
  - 如何配置、编译、安装
    - INSTALLING the kernel
    - BUILD directory for the kernel
    - CONFIGURING the kernel
    - COMPILING the kernel
  - 等等

Linux内核源代码中的主要子目录 I

- Documentation 内核方面的相关文档。
- arch 与体系结构相关的代码。
  对应于每个支持的体系结构，有一个相应的目录如x86、arm、alpha等。每个体系结构子目录下包含几个主要的子目录：
  - kernel 与体系结构相关的核心代码
  - mm 与体系结构相关的内存管理代码
  - lib 与体系结构相关的库代码
- include 内核头文件。
  对每种支持的体系结构有相应的子目录，如asm-x86、asm-arm、asm-alpha等。
  符号链接asm，如“asm -> asm-x86”。
  实际上，“#include ‘asm/xxxx.h’”？

Linux内核源代码中的主要子目录 II

- init 内核初始化代码。提供main.c，包含start_kernel函数。
- kernel 内核管理代码。
- mm 内存管理代码。
- ipc 进程间通讯代码。
- net 网络部分代码。
- lib 与体系结构无关的内核库代码。
- drivers 设备驱动代码。每类设备有相应的子目录，如char、block、net等
- fs 文件系统代码。每个支持文件系统有相应的子目录，如ext2、proc等。
- modules 可动态加载的模块。
- Scripts 配置核心的脚本文件。
- 等等

考虑Arch为i386 考察如下目录

- 观察Linux源码的根目录
- 观察arch目录
- 观察arch下的x86目录
  - arch/x86/boot
  - arch/x86/boot/compressed
  - arch/x86/kernel
- 观察Linux的init目录

阅读documentation/i386/boot.txt I

- 由于一些历史的原因，基于x86的Linux的启动比较复杂
- 这个文档（THE LINUX/I386 BOOT PROTOCOL）包含如下内容
  1. Linux/i386的启动协议（若干个）
  2. 内存布局图（MEMORY LAYOUT ）
  3. 实模式下的内核头结构及细节（THE REAL-MODE KERNEL HEADER ）
  4. 内核的命令行（THE KERNEL COMMAND LINE）
  5. MEMORY LAYOUT OF THE REAL-MODE CODE
  6. 启动配置示例（SAMPLE BOOT CONFIGURATION）
  7. 装载Linux的剩余部分（LOADING THE REST OF THE KERNEL）

阅读documentation/i386/boot.txt II

8. 特殊的命令行参数（SPECIAL COMMAND LINE OPTIONS）
9. 运行内核（RUNNING THE KERNEL）
10. 高级启动回调函数（ADVANCED BOOT LOADER HOOKS）
11. 32-bit BOOT PROTOCOL

阅读Linux源码根目录下的Makefile

找到缺省目标all

```
# The all: target is the default when no target is given on the
# command line.
# This allow a user to issue only 'make' to build a kernel including modules
# Defaults vmlinux but it is usually overridden in the arch makefile
all: vmlinux
```

找到vmlinux目标，并阅读

```
# vmlinux image - including updated kernel symbols
vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o $(kallsyms.o) FORCE
ifdef CONFIG_HEADERS_CHECK
        $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
endif
ifdef CONFIG_SAMPLES
        $(Q)$(MAKE) $(build)=samples
endif
        $(call vmlinux-modpost)
        $(call if_changed_rule,vmlinux__)
        $(Q)rm -f .old_version
```

解释：关于

$(call if_changed_rule, vmlinux__)

↓

rule_vmlinux__

```
# Link of vmlinux
# If CONFIG_KALLSYMS is set .version is already updated
# Generate System.map and verify that the content is consistent
# Use + in front of the vmlinux_version rule to silent warning with make -j2
# First command is ':' to allow us to use + in front of the rule
define rule_vmlinux__
        :
        $(if $(CONFIG_KALLSYMS),,+$(call cmd,vmlinux_version))

        $(call cmd,vmlinux__)
        $(Q)echo 'cmd_$@ := $(cmd_vmlinux__)' > $(@D)/.$(@F).cmd

        $(Q)$(if $($(quiet)cmd_sysmap),                                  \
          echo '  $($(quiet)cmd_sysmap)  System.map' &&)                 \
        $(cmd_sysmap) $@ System.map;                                     \
        if [ $$? -ne 0 ]; then                                           \
                rm -f $@;                                                \
                /bin/false;                                              \
        fi;
        $(verify_kallsyms)
endef
```

```
# Rule to link vmlinux - also used during CONFIG_KALLSYMS
# May be overridden by arch/$(ARCH)/Makefile
quiet_cmd_vmlinux__ ?= LD      $@
      cmd_vmlinux__ ?= $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) -o $@ \
      -T $(vmlinux-lds) $(vmlinux-init)                          \
      --start-group $(vmlinux-main) --end-group                  \
      $(filter-out $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o FORCE ,$^)
```

- 链接描述文件？
  - 链接顺序：
    - vmlinux-init
    - vmlinux-main

```
vmlinux
  ^
  |
  +-< $(vmlinux-init)
  |   +--< init/version.o + more
  |
  +--< $(vmlinux-main)
  |    +--< driver/built-in.o mm/built-in.o + more
  |
  +-< kallsyms.o (see description in CONFIG_KALLSYMS section)
```

- 参见“Documentation/kbuild/makefiles.txt”

```
--- 6.7 Custom kbuild commands

    When kbuild is executing with KBUILD_VERBOSE=0 then only a shorthand
    of a command is normally displayed.
    To enable this behaviour for custom commands kbuild requires
    two variables to be set:
    quiet_cmd_<command>     - what shall be echoed
          cmd_<command>     - the command to execute

    Example:
            #
            quiet_cmd_image = BUILD   $@
                  cmd_image = $(obj)/tools/build $(BUILDFLAGS) \
                                                 $(obj)/vmlinux.bin > $@

            targets += bzImage
            $(obj)/bzImage: $(obj)/vmlinux.bin $(obj)/tools/build FORCE
                    $(call if_changed,image)
                    @echo 'Kernel: $@ is ready'

    When updating the $(obj)/bzImage target the line:

    BUILD    arch/i386/boot/bzImage

    will be displayed with "make KBUILD_VERBOSE=0".
```

- 注意：
  - vmlinux-init
  - vmlinux-main

```
vmlinux-init := $(head-y) $(init-y)
vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
vmlinux-all  := $(vmlinux-init) $(vmlinux-main)
```

  - vmlinux-dirs

```
vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
                     $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
                     $(net-y) $(net-m) $(libs-y) $(libs-m)))
```

主要目标文件的编译

- vmlinux

↓

```
# The actual objects are generated when descending,
# make sure no implicit rule kicks in
$(sort $(vmlinux-init) $(vmlinux-main)) $(vmlinux-lds): $(vmlinux-dirs) ;
```

↓

```
# Handle descending into subdirectories listed in $(vmlinux-dirs)
# Preset locale variables to speed up the build process. Limit locale
# tweaks to this spot to avoid wrong language settings when running
# make menuconfig etc.
# Error messages still appears in the original language

PHONY += $(vmlinux-dirs)
$(vmlinux-dirs): prepare scripts
        $(Q)$(MAKE) $(build)=$@
```

↓

```
vmlinux-dirs    := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
                     $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
                     $(net-y) $(net-m) $(libs-y) $(libs-m)))
```

- 不妨以**core-y**为例，观察体系相关和体系无关部分的代码是如何被包含进来的

x86的启动文件

- 根据vmlinux-init找到head-y, init-y
  - 关于arch/x86/Makefile的引入
    根Makefile中以include的方式包含了X86体系结构相关部分的Makefile

```
# Read arch specific Makefile to set KBUILD_DEFCONFIG as needed.
# KBUILD_DEFCONFIG may point out an alternative default configuration
# used for 'make defconfig'
include $(srctree)/arch/$(SRCARCH)/Makefile
export KBUILD_DEFCONFIG
```

  - 在这个Makefile中

```
161 head-y := arch/x86/kernel/head_$(BITS).o
162 head-y += arch/x86/kernel/head$(BITS).o
163 head-y += arch/x86/kernel/init_task.o
```

    - 其中，变量BITS为32或者64，我们只考虑32位的情况
- 在根Makefile中

```
init-y          := init/
```

- 可以看到vmlinux包含如下内容
  - i386/kernel/head_32.S等 +
  - init/main.c + init/version.o +
  - CORE_FILES + DRIVERS +
  - NETWORKS + LIBS

为便于阅读，了解关于命令输出的相关内容

```
# Beautify output
# ---------------------------------------------------------------------------
#
# Normally, we echo the whole command before executing it. By making
# that echo $($(quiet)$(cmd)), we now have the possibility to set
# $(quiet) to choose other forms of output instead, e.g.
#
#         quiet_cmd_cc_o_c = Compiling $(RELDIR)/$@
#         cmd_cc_o_c       = $(CC) $(c_flags) -c -o $@ $<
#
# If $(quiet) is empty, the whole command will be printed.
# If it is set to "quiet_", only the short version will be printed.
# If it is set to "silent_", nothing wil be printed at all, since
# the variable $(silent_cmd_cc_o_c) doesn't exist.
#
# A simple variant is to prefix commands with $(Q) - that's useful
# for commands that shall be hidden in non-verbose mode.
#
#       $(Q)ln $@ :<
#
# If KBUILD_VERBOSE equals 0 then the above command will be hidden.
# If KBUILD_VERBOSE equals 1 then the above command is displayed.

ifeq ($(KBUILD_VERBOSE),1)
  quiet =
  Q =
else
  quiet=quiet_
  Q = @
endif
```

make XXX

- 若make install
  - 在x86的Makefile中有install规则
- 若make bzImage/zImage等，则要找到对应的目标然后进行
  - bzImage/zImage可在arch/x86的Makefile中找到相应规则
  - 其他的zXXX/bzXXX也都依赖于boot下的zImage/bzImage
- 它们最终都找到i386/boot/Makefile

考虑boot bzImage I

- Make bzImage ···
- 在arch/X86/Makefile中

```
215 zImage bzImage: vmlinux
216 »       $(Q)$(MAKE) $(build)=$(boot) $(KBUILD_IMAGE)
217 »       $(Q)mkdir -p $(objtree)/arch/$(UTS_MACHINE)/boot
218 »       $(Q)ln -fsn ../../x86/boot/bzImage $(objtree)/arch/$(UTS_MACHINE)/boot/bzImage
```

  - z代表压缩；b代表大内核
- 到boot目录下的Makefile
  - 观察boot目录和boot下的Makefile
  - 观察compressed目录及该目录下的Makefile

考虑boot bzImage II

最后:

- 在Linux内核源代码顶层目录下生成一个vmlinux
  - arch/x86/kernel/head_32.S···+init/main.c+···
- compressed下的vmlinux为
  - compressed/head_32.S + 压缩后的顶层目录下的vmlinux ···
- boot下的bzImage为
  - boot下header.S等（即setup.bin）+compressed/vmlinux

x86的启动（小结）

- boot/header.S等
- compressed/head_32.S等
- kernel/head_32.S等
- init/main.c

I386机器的启动层次

1. BIOS (Basic I/O System)
2. Bootloader
   - 软盘启动
   - 硬盘启动
   - 嵌套boot loader
   - 例如：grub、lilo、. . . . . .
3. Linux kernel

Bootloader必须完成内核代码的加载，然后跳转到入口处运行

BIOS I

- 加电，RESET引脚

CPU加电后，将会初始化程序指针到某个约定好的地址上取指令运行，在这个地指处，往往安排了启动相关的代码，例如BIOS或者reset向量处理入口

- 初始化寄存器；CS:IP = 0xffffffff0, in ROM
- ROM←BIOS
- BIOS启动内容
  - POST（上电自检）
  - 初始化硬件设备
  - 搜索一个操作系统来启动
    根据配置，操作系统可以在软盘/硬盘/CD_ROM上
  - 把对应设备的第一个扇区的内容（bootloader或部分）拷贝到RAM(0x7c00)处
  - 跳转到0x7c00处执行

Bootloader（引导装载程序）

- BIOS调用Bootloader把操作系统内核映像装载到RAM中

考虑IBM PC的启动

软盘启动：

BIOS拷贝第一个扇区的内容（bootsect）到RAM（0x7c00）中

硬盘启动：

硬盘的第一个扇区：主引导记录MBR, Master Boot Record

- MBR存储该硬盘的分区表+ 一小段引导程序
- 这个引导程序用来装载OS所在分区的第一个扇区（boot loader）的内容到RAM中
- 这个引导程序也可以被替换

Linux的BootLoader

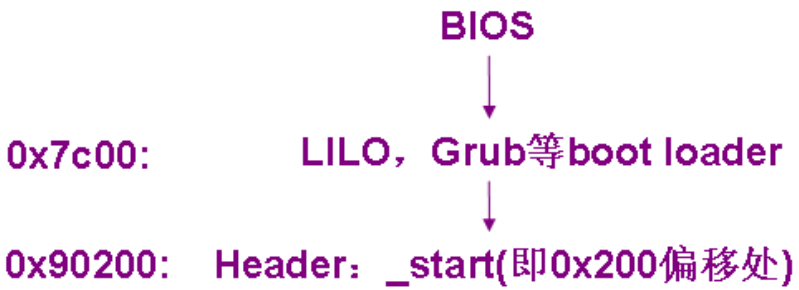
- 典型的有：LILO和Grub
- LILO（Linux Loader）
  - 可以被安装在OS分区的第一个扇区（启动扇区）
  - 也可以代替MBR中的引导程序
- 事实上，LILO的代码尺寸大于一个扇区，因此被分成两个部分
  - MBR或启动扇区部分
  - 剩余部分
- 第一部分也被BIOS装载到RAM中0x7c00的位置
- 第一部分在运行时将自己完整的装载到RAM中
- 通常LILO或GRUB会显示一个已安装操作系统的列表
  - 按照用户的选择（或者按照缺省项）装载目标操作系统运行
  - 可能装载操作系统指定的启动代码运行（嵌套的情况）
  - 可能直接装载操作系统内核来运行

LILO的OS启动过程

- 显示“Loading…”
- header内容被装载到RAM的0x90000
- 操作系统内核的其他内容被装载到
  - 对于小内核：0x10000（即64K处），称为低装载
  - 对于大内核：0x100000（即1M处），称为高装载
- 跳转到0x90200处运行

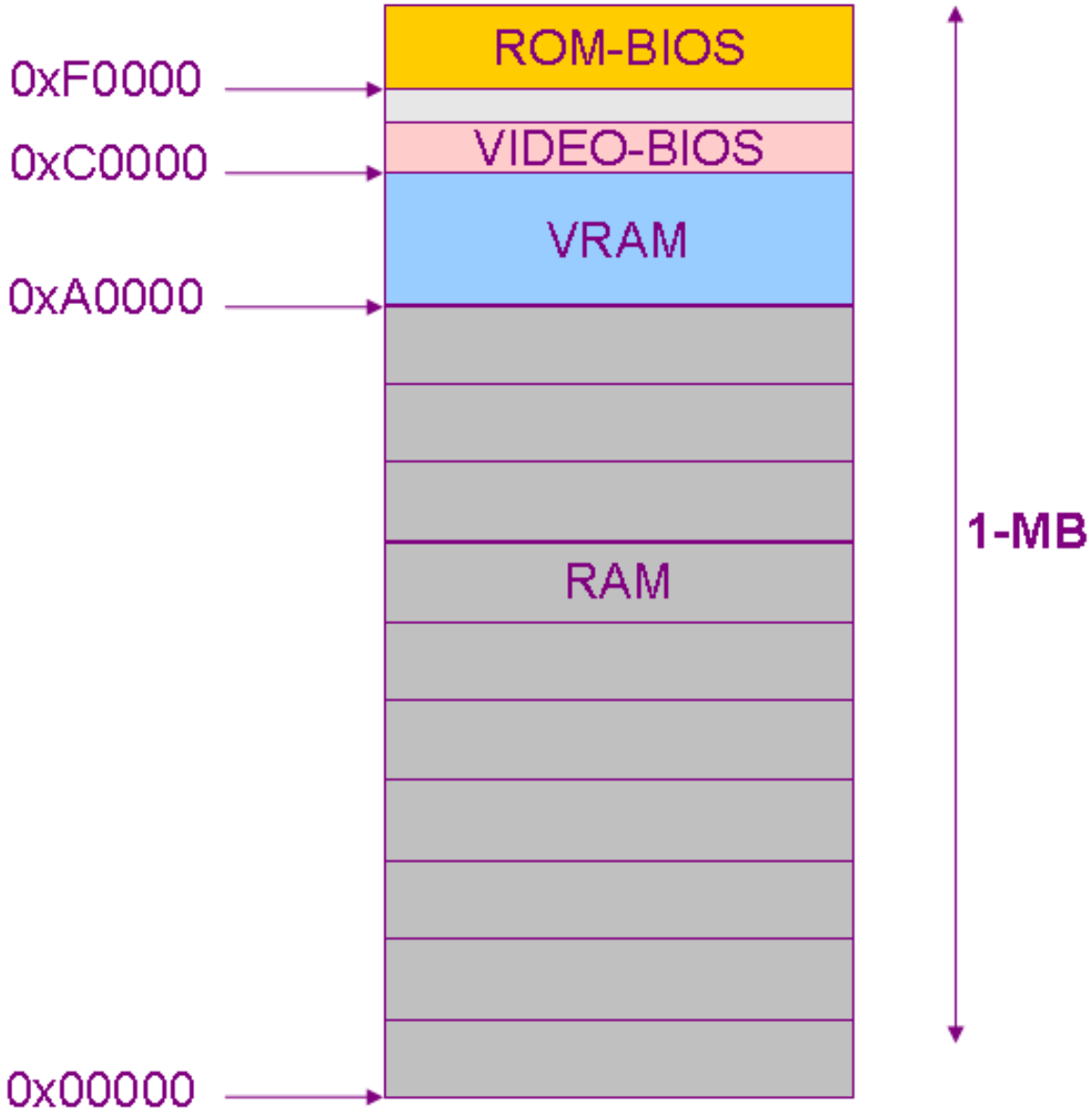
I386内核的启动

- 启动方式
  - 软盘启动：Linux2.6.26不支持
  - 硬盘启动：从header的_start开始运行



在进入源代码讲解之前，先看一下加载i386内核的内存布局图

硬件角度：I386实模式下的内存布局图



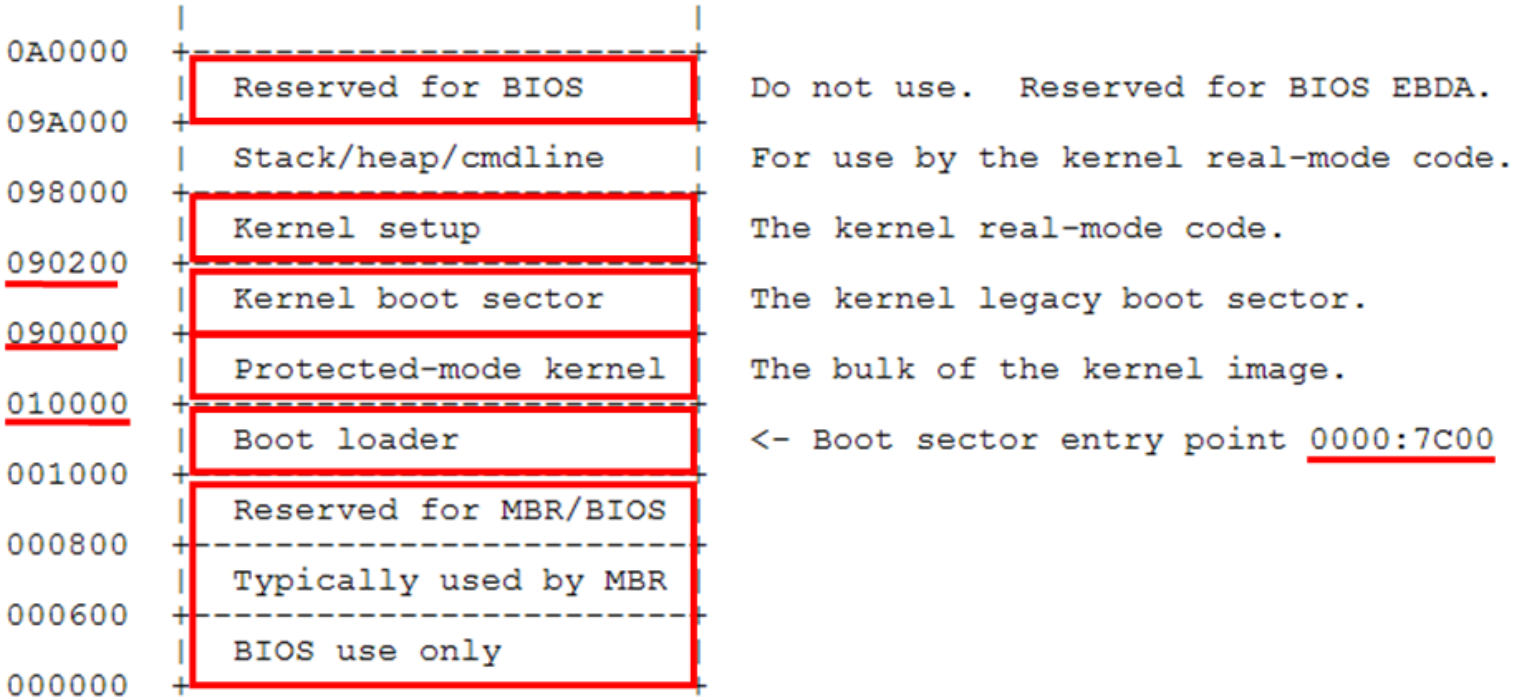
I386内核从实模式开始启动运行

什么是实模式?

- 实模式是为了兼容早期的CPU而设置的
- i386系统总是始于实模式
- 实模式下
  - 地址总线：20位
  - 内存范围：0~1MB
  - 逻辑地址 = 段地址 + 段内偏移
  - 段地址 = 段寄存器中的值*16 （或左移4位）
  - 段寄存器： cs/ds/es/fs/gs
  - 段寄存器长度：16bit
  - 段长：16位偏移=64KB

加载I386内核的内存布局图

- zImage/Image的内核加载器所使用的经典的内存布局（1M=0x100000）
- 参见boot.txt



硬盘启动，两阶段引导

- 装载LILO（LInuxLOader）
  - 第一个扇区
  - …
- 装载LINUX
  - header.S等→0x90000，其中_start在0x90200处
  - 系统
    - 0x10000
    - 0x100000
- 跳转到_start

启动第一步，小结

- 总之，在跳转到header.S的_start的时候，内存里面的代码布局为
  - 0x90000： header.S 前512字节内容
  - 0x90200： header.S 的_start及其后
  - 低装载：
    - 0x10000：带解压的vmlinux
  - 高装载：
    - 0x100000：带解压的vmlinux
- 实模式下的内核头结构
  - 包括512字节的最后和_start 之后的一些位置
  - 从偏移0x1F1开始，具体描述参见documentation/i386/boot.txt

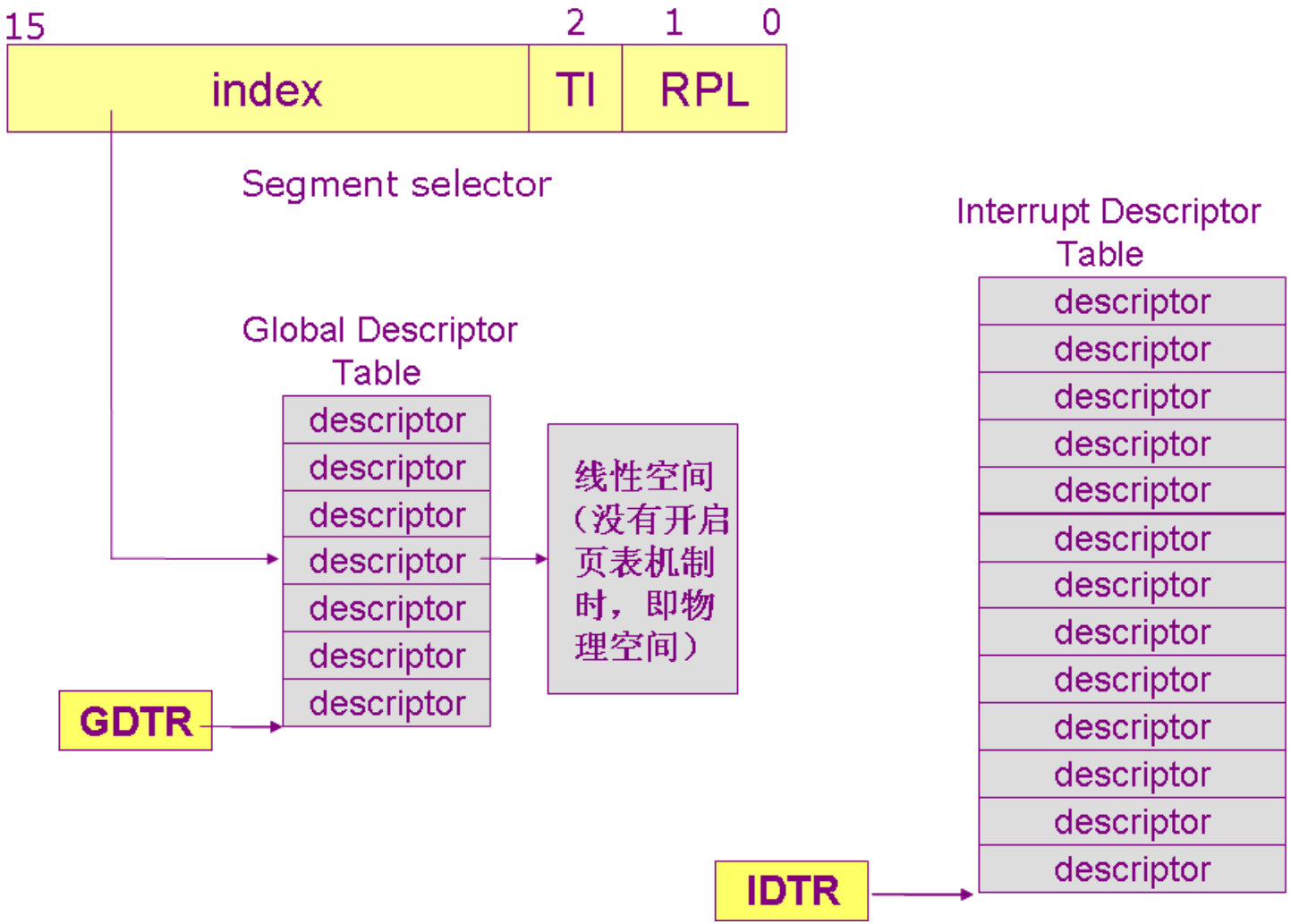
Header.S之_start：0x90200

- __start：
  - 跳转到start_of_setup
  - 检查setup的signature
  - 清除BSS段
  - 跳转到main执行
- 其中，Main用来初始化硬件设备并为内核程序的执行建立环境
  - 内存检测、键盘、视频、…
  - go_to_protected_mode

关于保护模式

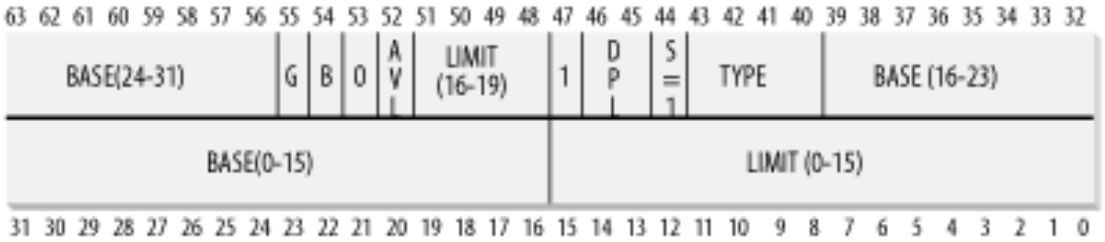
- 在setup.bin中，从实模式→保护模式
- 保护模式下，地址总线32位，访存范围为4GB
- 原来的段寄存器现在被称作段选择子，与GDT表配合使用
  - GDT表由gdtr指示其位置和长度
  - 使用特殊的指令进行操作：sgdt/lgdt

图示

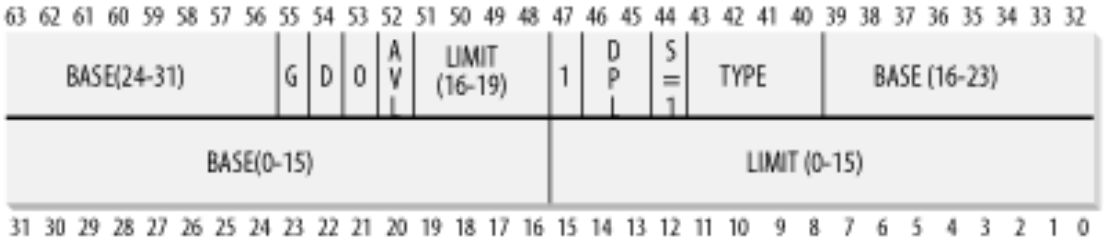

15
2
1
0
index
TI
RPL
Segment selector
Global Descriptor Table
descriptor
descriptor
descriptor
descriptor
descriptor
descriptor
descriptor
GDTR
线性空间（没有开启页表机制时，即物理空间）
Interrupt Descriptor Table
descriptor
descriptor
descriptor
descriptor
descriptor
descriptor
descriptor
descriptor
descriptor
descriptor
descriptor
descriptor
descriptor
IDTR

段描述符的格式

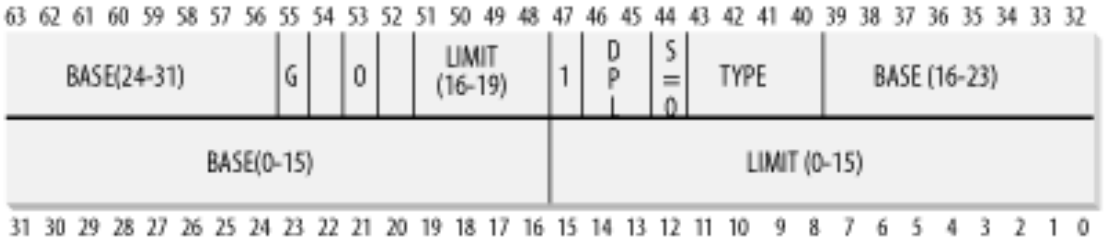
Data Segment Descriptor



Code Segment Descriptor



System Segment Descriptor



- 一般装载gdt和idt之后，要重新装载段寄存器
  - cs、ds、es、fs、gs
  - cs通常通过一条长跳转指令装载
  - 其他数据段寄存器直接设置
  - 如在arch/x86/boot/header.S

```
54          ljmp    $BOOTSEG, $start2
55
56 start2:
57          movw    %cs, %ax
58          movw    %ax, %ds
59          movw    %ax, %es
60          movw    %ax, %ss
61          xorw    %sp, %sp
62          sti
63          cld
64
65          movw    $bugger_off_msg, %si
66
```

go_to_protected_mode I

- arch/x86/boot/main.c中，main函数最后调用go_to_protected_mode函数
  - arch/x86/boot/pm.c中

```
void go_to_protected_mode(void)
{
        /* Hook before leaving real mode, also disables interrupts */
        realmode_switch_hook();

        /* Move the kernel/setup to their final resting places */
        move_kernel_around();

        /* Enable the A20 gate */
        if (enable_a20()) {
                puts("A20 gate not responding, unable to boot...\n");
                die();
        }

        /* Reset coprocessor (IGNNE#) */
        reset_coprocessor();

        /* Mask all interrupts in the PIC */
        mask_all_interrupts();

        /* Actual transition to protected mode... */
        setup_idt();
        setup_gdt();
        protected_mode_jump(boot_params.hdr.code32_start,
                            (u32)&boot_params + (ds() << 4));
}
```

go_to_protected_mode II

- 这里是code32_start的定义在

```
        # Kernel attributes; used by setup.  This is part 1 of the
        # header, from the old boot sector.

        .section ".header", "a"
        .globl  hdr
hdr:
setup_sects:    .byte SETUPSECTS
root_flags:     .word ROOT_RDONLY
syssize:        .long SYSSIZE
ram_size:       .word RAMDISK
vid_mode:       .word SVGA_MODE
root_dev:       .word ROOT_DEV
boot_flag:      .word 0xAA55

        # offset 512, entry point

        .globl  _start
_start:
```

.....

```
code32_start:                           # here loaders can put a different
                                        # start address for 32-bit code.
#ifndef __BIG_KERNEL__
                .long   0x1000          #   0x1000 = default for zImage
#else
                .long   0x100000        # 0x100000 = default for big kernel
#endif
```

go_to_protected_mode III

- main函数在一开始就调用了copy_boot_params

```
static void copy_boot_params(void)
{
        struct old_cmdline {
                u16 cl_magic;
                u16 cl_offset;
        };
        const struct old_cmdline * const oldcmd =
                (const struct old_cmdline *)OLD_CL_ADDRESS;

        BUILD_BUG_ON(sizeof boot_params != 4096);
        memcpy(&boot_params.hdr, &hdr, sizeof hdr);

        if (!boot_params.hdr.cmd_line_ptr &&
            oldcmd->cl_magic == OLD_CL_MAGIC) {
                /* Old-style command line protocol. */
                u16 cmdline_seg;

                /* Figure out if the command line falls in the region
                   of memory that an old kernel would have copied up
                   to 0x90000... */
                if (oldcmd->cl_offset < boot_params.hdr.setup_move_size)
                        cmdline_seg = ds();
                else
                        cmdline_seg = 0x9000;

                boot_params.hdr.cmd_line_ptr =
                        (cmdline_seg << 4) + oldcmd->cl_offset;
        }
}
```

arch/boot/compressed

- 具有自解压功能的vmlinux.bin
  - zImage中，在0x1000处
    - 关于100000，10000，1000（参见move_kernel_arround）
  - bzImage中，在0x100000处
- compressed/head_32.S的startup_32
  - 初始化段寄存器和一个临时堆栈
  - 初始化BSS段
  - 解压缩decompress_kernel
    - 无论高装载或低装载→解压缩后，都在物理地址0x100000（1MB）处
  - 跳转到0x100000处

- linux-2.6.26\arch\x86\configs\i386_defconfig中定义CONFIG_PHYSICAL_START=0x100000

- 解压缩后，vmlinux在0x100000处
  - 根据vmlinux.lds，vmlinux的地址被链接为0xc0000000+0x100000处
  - 如何正确运行呢?
- 最初是实模式，然后进入保护模式，还没有分页、映射好
- 期间，
  - 没有长跳转，只使用采用相对地址的近距离跳转
  - 不使用符号名

Kernel/Head_32.S

- Startup_32
  - 初始化段寄存器
  - 设置页目录和页表，分页
  - 建立进程0的内核堆栈
  - Setup_idt
  - 拷贝系统参数
  - 识别处理器
  - GDT、IDT
  - i386_start_kernel

如何进入start_kernel?

??? 如何从0～8M的空间中转入3G以上的地址空间中运行的 ???

```
void __init i386_start_kernel(void)
{
        start_kernel();
}
```

关于页目录和页表的定义及初始化 I

arch/x86/kernel/head_32.S中定义了swap_pg_dir

```
/*
 * BSS section
 */
.section ".bss.page_aligned","wa"
        .align PAGE_SIZE_asm
#ifdef CONFIG_X86_PAE
swapper_pg_pmd:
        .fill 1024*KPMDS,4,0
#else
ENTRY(swapper_pg_dir)
        .fill 1024,4,0
#endif
swapper_pg_fixmap:
        .fill 1024,4,0
ENTRY(empty_zero_page)
        .fill 4096,1,0
```

arch/x86/kernel/vmlinux_32.lds.S文件中

```
  .bss : AT(ADDR(.bss) - LOAD_OFFSET) {
        __init_end = .;
        __bss_start = .;                /* BSS */
        *(.bss.page_aligned)
        *(.bss)
        . = ALIGN(4);
        __bss_stop = .;
        _end = . ;
        /* This is where the kernel creates the early boot page tables */
        . = ALIGN(PAGE_SIZE);
        pg0 = . ;
  }
```

关于页目录和页表的定义及初始化 II

分页使能

```
/*
 * Enable paging
 */
        movl $pa(swapper_pg_dir),%eax
        movl %eax,%cr3          /* set the page table pointer.. */
        movl %cr0,%eax
        orl  $X86_CR0_PG,%eax
        movl %eax,%cr0          /* ..and set paging (PG) bit */
        ljmp $__BOOT_CS,$1f     /* Clear prefetch and normalize %eip */
1:
```

在使用页目录和页表之前，首先要进行页目录和页表的初始化（部分）

```
/*
 * This is how much memory *in addition to the memory covered up to
 * and including _end* we need mapped initially.
 * We need:
 *  - one bit for each possible page, but only in low memory, which means
 *     2^32/4096/8 = 128K worst case (4G/4G split.)
 *  - enough space to map all low memory, which means
 *     (2^32/4096) / 1024 pages (worst case, non PAE)
 *     (2^32/4096) / 512 + 4 pages (worst case for PAE)
 *  - a few pages for allocator use before the kernel pagetable has
 *     been set up
 *
 * Modulo rounding, each megabyte assigned here requires a kilobyte of
 * memory, which is currently unreclaimed.
 *
 * This should be a multiple of a page.
 */
LOW_PAGES = 1<<(32-PAGE_SHIFT_asm)
```

关于页目录和页表的定义及初始化 III

```
/*
 * To preserve the DMA pool in PAGEALLOC kernels, we'll allocate
 * pagetables from above the 16MB DMA limit, so we'll have to set
 * up pagetables 16MB more (worst-case):
 */
#ifdef CONFIG_DEBUG_PAGEALLOC
LOW_PAGES = LOW_PAGES + 0x1000000
#endif

#if PTRS_PER_PMD > 1
PAGE_TABLE_SIZE = (LOW_PAGES / PTRS_PER_PMD) + PTRS_PER_PGD
#else
PAGE_TABLE_SIZE = (LOW_PAGES / PTRS_PER_PGD)
#endif
BOOTBITMAP_SIZE = LOW_PAGES / 8
ALLOCATOR_SLOP = 4

INIT_MAP_BEYOND_END = BOOTBITMAP_SIZE + (PAGE_TABLE_SIZE + ALLOCATOR_SLOP)*PAGE_SIZE_asm
```

```
/*
 * Initialize page tables.  This creates a PDE and a set of page
 * tables, which are located immediately beyond _end.  The variable
 * init_pg_tables_end is set up to point to the first "safe" location.
 * Mappings are created both at virtual address 0 (identity mapping)
 * and PAGE_OFFSET for up to _end+sizeof(page tables)+INIT_MAP_BEYOND_END.
 *
 * Note that the stack is not yet set up!
 */
#define PTE_ATTR        0x007           /* PRESENT+RW+USER */
#define PDE_ATTR        0x067           /* PRESENT+RW+USER+DIRTY+ACCESSED */
#define PGD_ATTR        0x001           /* PRESENT (no other attributes) */
```

关于页目录和页表的定义及初始化 IV

```
page_pde_offset = (__PAGE_OFFSET >> 20);

        movl $pa(pg0), %edi
        movl $pa(swapper_pg_dir), %edx
        movl $PTE_ATTR, %eax
10:
        leal PDE_ATTR(%edi),%ecx                /* Create PDE entry */
        movl %ecx,(%edx)                        /* Store identity PDE entry */
        movl %ecx,page_pde_offset(%edx)         /* Store kernel PDE entry */
        addl $4,%edx
        movl $1024, %ecx
11:
        stosl
        addl $0x1000,%eax
        loop 11b
        /*
         * End condition: we must map up to and including INIT_MAP_BEYOND_END
         * bytes beyond the end of our own page tables; the +0x007 is
         * the attribute bits
         */
        leal (INIT_MAP_BEYOND_END+PTE_ATTR)(%edi),%ebp
        cmpl %ebp,%eax
        jb 10b
        movl %edi,pa(init_pg_tables_end)

        /* Do early initialization of the fixmap area */
        movl $pa(swapper_pg_fixmap)+PDE_ATTR,%eax
        movl %eax,pa(swapper_pg_dir+0xffc)
#endif
        jmp 3f
```

关于kernel/head_32.S的内核堆栈

```
/* Set up the stack pointer */
lss stack_start,%esp
```

```
.data
ENTRY(stack_start)
        .long init_thread_union+THREAD_SIZE
        .long __BOOT_DS
```

```
/*
 * Initial thread structure.
 *
 * We need to make sure that this is THREAD_SIZE aligned due to the
 * way process stacks are handled. This is done by having a special
 * "init_task" linker map entry..
 */
union thread_union init_thread_union
        __attribute__((__section__(".data.init_task"))) =
                { INIT_THREAD_INFO(init_task) };
```

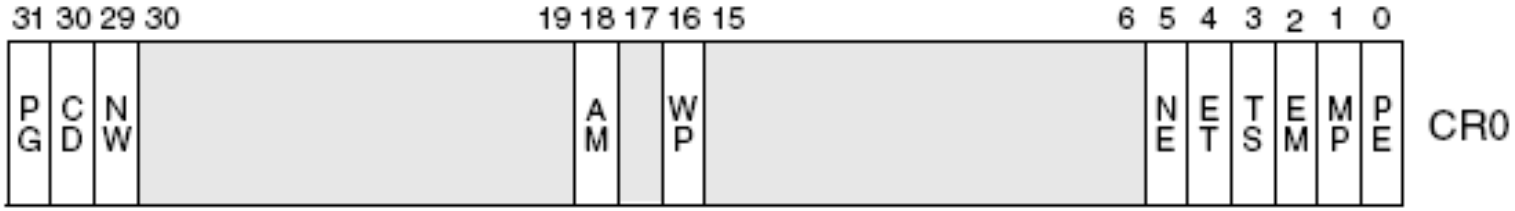
控制寄存器（Control Registers）

- CR0
- CR1
- CR2
- CR3

大多与内存相关

CR0 I

- CR0, MSW register (Machine Status Word, 32-bit version)
  - 包含系统控制位，用于控制操作模式和状态



- Instruction: lmsw
- To turn on the PE-bit (enables protected-mode)
  - LINUX’ setup.S （旧版本）:

```
movw $1, %ax
lmsw %ax
jmp flush_instr // why?
flush_instr:
```

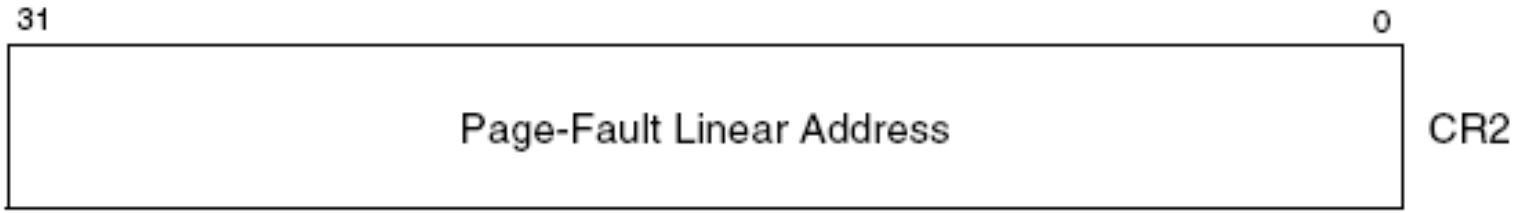
CR0 II

- in Linux-2.6.26/arch/x86/boot/pmjump.S

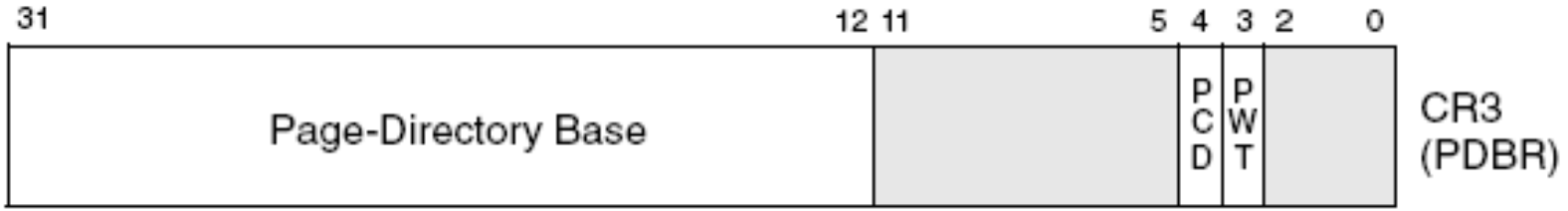
```
        movl    %cr0, %edx
        orb     $X86_CR0_PE, %dl        # Protected mode
        movl    %edx, %cr0
        jmp     1f                      # Short jump to serialize on 386/486
1:

        # Transition to 32-bit mode
        .byte   0x66, 0xea              # ljmpl opcode
2:      .long   in_pm32                 # offset
```

CR1、CR2、CR3

- CR1：保留
- CR2：在缺页异常的时候，记录缺页地址



- CR3：记录页目录所在的物理地址和两个标记(PCD & PWT)



小结

作业2

- i386实模式下是如何解决20位地址空间和16位段寄存器之间的不匹配问题的？
- i386保护模式下的段寄存器的内容与实模式下段寄存器的内容一样么？如何解释？

Project2

- 基于x86的linux-2.6.26的启动分析
  - 首先进行Makefile的分析，了解bzImage的代码结构
  - 考虑grub为启动引导程序，分析代码的启动过程。
- 提供详细分析报告

Thanks !

The end.