疯狂内核

带您探索 Linux 核心

掌握计算机底层软件必要知识

内核初始化

陈云松

blog：http://blog.csdn.net/yunsongice
email：yunsong_ice@163.com
qq:85408821

目录

读内核源代码是一件很有意思的事。它像一条线，把操作系统、C 语言、汇编语言、数据结构与算法、系统结构、组成原理、编译原理，等等计算机的基础课程串起来。而分析 linux 的启动很重要，因为牵涉到硬件的初始化和内核各模块初始化环境的搭建，所以我们就针对 x86 体系，对从打开 PC 电源到屏幕上出现 shell 环境，来对整个 Linux 的初始化过程进行一个全方位的分析，希望能有帮助，不足之处，还请各位网友不吝赐教。

1 引子

目前，市面上的大多数计算机系统的内存都是“ 随机性 ”的：一旦关机断电，存储在内存中的信息、连同操作系统本身都会丢失。所以，必须把操作系统（内核）的程序存储在某种永久性的介质中，使得开机加电时有一个从不挥发介质装入操作系统、并转入运行的过程，这个过程就叫做“ 引导 ”(bootload，或 boot),也称为“ 自举 ”。

1.1 上电

我们只关注 x86 体系的这个过程，CPU 所在的主板会有一个特殊的硬件电路在 CPU 的 RESET 引脚上产生一个电平。这时，CPU 处于实地址模式中，并开始自检，自检的最后一个步骤是把一些寄存器（如 cs 和 eip）设置成固定值。我们知道，实模式下的寻址方法是 16 位段寄存器左移 4 位加上 16 位偏移构成具有能寻址 1MB 能力的 20 位地址。所以，刚开始时，复位输入提供一种初始化的硬件手段：标志寄存器设为 0xuuuu0002(u 代表未定义，实模式下 9 位标志可用，这里是奇偶标志为 1)；指令指针 eip 设为 0x0000fff0，CS 寄存器设为 0xf000；DS、SS、ES、FS 和 GS 寄存器都设为 0x0000，指令队列清空。

CPU 在识别出 Reset 信号后把数据总线设在高阻状态，地址线强行设为 1。由于清空中断标志是初始化的一部分，外部中断被禁止。因为代码段寄存器为 0xf000，指令指针为 0x0000fff0，地址线 A20-A31 全部是 1，从而复位后实模式程序从地址 0xffffff0 开始（注意，只用于实模式高地址位忽略，从地址 0xffff0 开始）。该地址处可以包含一条转移指令跳到启动程序处。

那么，这段程序要有多大呢？这就要看具体的设计了。在早期的计算机中这段程序般都很小，例如 2K 字节或者史小，甚至于只有几条指令（我们的胡希明老同志回忆起 70 年代中美建交后进入中国的 NOVA 机，由此而来的国产机名为 DJS-130，操作系统为 RTOS，引导程序只有 13 条指令，当时称初始引导 13 条，亦称手拨 13 条。13 条指令执行结果是通过光电读入机把存放在穿孔纸带上的 RTOS 执行码装入内存）。这是因为早期的 PROM 或 EPROM 的容量都很小，并且其目的和功能也很单一。

我们在“ PC 存储器 ”http://blog.csdn.net/yunsongice/archive/2010/10/08/5927345.aspx 一篇博文中讲到，ROM 和 RAM 扩展成一个整体的内存系统，这个 ROM 的地址是 32 位的，况且开机时 CPU 只运行在实模式下，高 12 位全被置 1 那么线性地址 0xffff0 对应的物理地址就是 0xffffff0，定位到这个 ROM。ROM 中所存放的程序集在 80x86 体系中通常叫做基本输入

/输出系统（Basic Input/Output System, BIOS），因为它包括几个中断驱动的低级过程。比如，执行 INT 10H，就是显示服务，10H 是中断类型号，在运行以下流程：

```
(SP)←(SP) - 2              ；SP - 2
((SP+1),(SP))←(PSW)        ；标志位寄存器内容进栈
IF←0, TF←0                 ；清除中断和陷阱标志
(SP)←(SP) - 2              ；SP - 2
((SP+1),(SP))←(CS)         ；CS 入栈
(SP)←(SP) – 2              ；SP - 2
((SP+1),(SP))←(IP)         ；IP 入栈
(IP)←(TYPE*4)              ；取中断服务程序偏移地址，这里的 TYPE 是 10H
(CS)←(TYPE*4 + 2)          ；取中断服务程序段地址，这里的 TYPE 是 10H
……                         ；开始执行位于 ROM 的 BIOS 中断服务程序
(SP)←(SP) + 2              ；SP + 2
(IP)←((SP-1),(SP))         ；IP 出栈
(SP)←(SP) + 2              ；SP + 2
(CS)←((SP-1),(SP))         ；CS 出栈
IF←1, TF←1                 ；设置中断和陷阱标志
(SP)←(SP) + 2              ；SP + 2
(PSW)←((SP-1),(SP))        ；标志位寄存器内容出栈
```

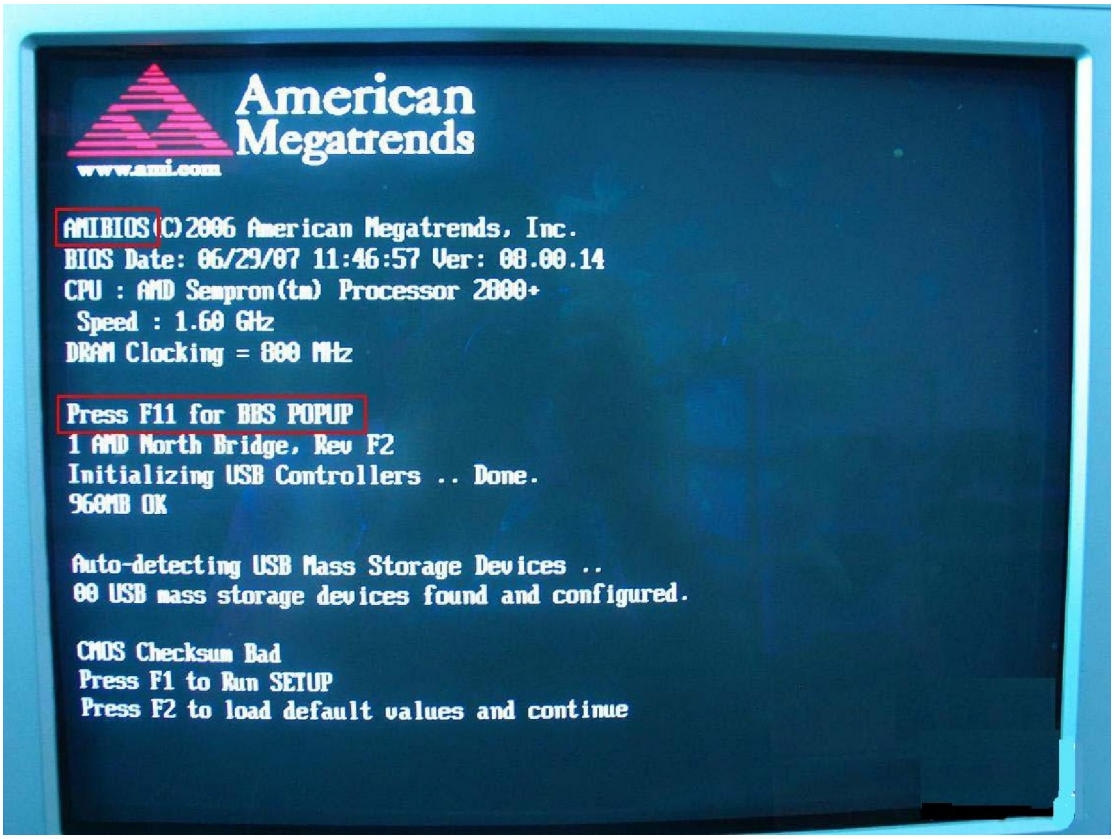
不管 Linux、MS-DOS 还是 Windows，在启动时，都要通过这些过程对计算机硬件设备初始化。Linux 一旦进入了保护模式，就不再使用 BIOS，而是为计算机上的每个硬件设备提供各自的设备驱动程序。

1.2 BIOS 时代

BIOS 使用实模式的地址，一个实模式的地址由一个段基址 seg 和一个偏移量 off 组成，相应的物理地址可以这样计算：seg * 16 + off。所以 CPU 寻址电路根本就不需要全局描述符表、局部描述符表或页表把逻辑地址转换成物理地址。显然，对 GDT、LDT 和页表进行初始化的代码必须在实模式下运行。

Linux 启动阶段进入 BIOS 时代，此时 Linux 最重要的工作就是从磁盘或者其他外部设备获取内核映像。这个过程主要由 BIOS 来完成，实际上执行了以下操作：

1. 对计算机硬件执行一系列的测试，用来检测现在都有什么设备以及这些设备是否正常工作。这个阶段通常称为 POST（Power-On Self-Test，上电自检）。在这个阶段，会显示一些信息，例如 BIOS 版本号。



2. 初始化硬件设备。这个阶段在现代基于 PCI 的体系结构中相当重要，因为它可以保证所有的硬件设备操作不会引起 IRQ 线与 I/O 端口冲突。在这一步以后，会在屏幕闪现一下系统中所安装的所有 PCI 设备的一个列表。

3. 搜索一个操作系统来启动。计算机系统发展到当今这个时代，这个过程实际上是根据 BIOS 的设置顺序，在软盘、硬盘和 CD-ROM 的第一个扇区（引导扇区）进行访问。

4. 只要找到一个有效的设备，就把第一个扇区 MBR 的内容拷贝到 RAM 中从物理地址 0x00007c00 开始的位置，然后跳转到此处，开始执行刚才装载进来的代码。

5. 检查（WORD）0000：7dfe 是否等于 0xaa55。若不等于则转去尝试其他介质；如果没有其他启动介质，则显示 “ No ROM BASIC ”，然后死机；

6. MBR 先将自己复制到 0000：0600 处，然后继续执行；

7. 在主分区表中搜索标志为活动的分区。如果发现没有活动分区或者不止一个活动分区，则停止；

8. 将活动分区的第一个扇区读入内存地址 0000：7c00 处；

9. 检查(WORD)0000 :7dfe 是否等于 0xaa55,若不等于则显示 “ Missing Operating System ”，然后停止；

10. 跳转到 0000：7c00 处继续执行特定系统的 bootloader 程序。

Linux 发展到今天，从第 4 步开始的过程已经和过去有很大的不同了。如果是硬盘启动，这个过程是先从整个硬盘的引导扇区读入一个作为中间步骤的工具性程序（而不是操作系统映象），称为“引导装入程序（bootloader）”，再由引导装入程序装入操作系统映象。

当今 Linux 世界的 bootloader 已经有了长足的发展，面对各种嵌入式产品有不同的 bootloader，而我们 x86 PC 或服务器用得最多的就是 LILO 和 Grub。从概念上说引导装入程序同样也是使用的 BIOS 服务程序，只是体积更大，功能更为复杂。例如 Grub 就可以让用户从磁盘上的多个操作系统映象中有选择地引导。另一方面，由于引导装入程序的功能较强，也有利于减小对操作系统映象在磁盘上存储位置和方式的限制。

由于硬盘容量的迅速发展，实际使用中常常把一个硬盘划分成若干“分区”，从而把一个物理的硬盘划分成若干个逻辑磁盘。这样一来，每个逻辑磁盘的第一个扇区仍然是引导扇区，分别用于相应逻辑磁盘中的操作系统映象（如果有的话）。但是，这些引导扇区在物理上当然不再是整个硬盘的第一个扇区了。在这种情况下，整个硬盘的第一个扇区不属于任何一个逻辑磁盘，或者说已经超脱于所有这些逻辑磁盘之外，上了一个层次。但是 BIOS 还是把它当作整个硬盘的引导扇区，加电时还是从这个扇区“引导”，所以这个扇区称为“主引导记录块（扇区）”MBR。这个概念就是这么来的。

MBA 中含有关于盘区划分的信息（通过 fdisk 设置），还有一段简短的程序，一共 512 字节。不过，MBR 中的程序并不直接引导操作系统，而是根据盘区划分的信息从一个预定的“活跃”逻辑磁盘中读入其引导扇区，再由这个引导扇区自己采取进一步的行动。所以，可以把 MBR 的作用看成是为 BIOS 中的初始引导程序提供了一种类似于间接寻址的手段。不过，现在大家都不这么干了，用来“引导”Grub 的程序（连同有关盘区划分的信息）一起放在了 MBR 中，使整个引导过程少转一道弯。

加载内核引导程序以后，POST 部分的大多数代码会被从内存中清理出来，但仍然会有部分的运行时服务保留在内存之中供目标操作系统使用，后面我们会看到相应的内存布局。

1.3 内核引导程序

前面说了，内核引导程序（boot loader）是由 BIOS 用来把操作系统的内核映像装载到 RAM 中所调用的一个程序。让我们简要地描绘一下引导装入程序在 IBM 的 PC 体系结构中是如何工作的。内核引导程序 bootloader 有很多版本，最流行的是 grub 和 lilo。不管什么 bootloader，其引导过程主要分两个阶段：主引导程序（MBR）；活动分区引导记录中的次引导程序。

主引导程序是一个块（扇区），所以是 512 字节，它包含 446 字节的程序代码和 64 字节的分区表，最后还有 2 个字节，固定是 0xAA55，用于检测 MBR 是否有效。其中的程序代码执行扫描分区表，寻址活动分区，将位于活动分区的引导记录中的次引导程序加载到内存中并执行。

次引导程序负责加载 Linux 内核映像，并将控制权转交给内核。内核被加载到内存中并获得

控制权之后，内核阶段开始工作。

假设我们使用最常用的 grub 作为 bootloader，则/boot/grub/grub.conf 文件作为其配置文件，它的内容如下：

```
[root@localhost grub]# cat grub.conf
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE:    You have a /boot partition.    This means that
#                 all kernel and initrd paths are relative to /boot/, eg.
#                 root (hd0,0)
#                 kernel /vmlinuz-version ro root=/dev/rootVG/root
#                 initrd /initrd-version.img
#boot=/dev/sda
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title Red Hat Enterprise Linux Server (2.6.18-194.el5)
        root (hd0,0)
        kernel /vmlinuz-2.6.18-194.el5 ro root=/dev/rootVG/root elevator=noop rhgb quiet
        initrd /initrd-2.6.18-194.el5.img
```

这个配置文件提供相关信息给 grub 程序，grub 的次引导程序通过它将内核映像加载到内存的某个位置，然后跳转到此处执行该处的代码。倒数第二行 kernel 命令的第一个参数：vmlinuz-2.6.18-194.el5，就是内核映像。

grub 程序引导系统的过程如下：

grub 分为 stage1(stage1.5)和 stage2。stage1 可以看成主引导程序，只有 512 个字节；而 stage2 则是次引导程序。stage2 起到了 grub 的关键作用，包括对特定文件系统的支持(ext2,ext3 等)。

grub 自己有一个 shell 脚本，以及一些内部程序（如 kernel、initrd、root 命令等)。我们看到了，配置文件里用到了 root、kernel 和 root 命令。要让 grub 执行这些命令，就需要在这些命令前足留空格（或 Tab 键），不然 grub 会认为是参数，如 default、timeout 等。

先来看 stage1：stage1 叫做装载基本的引导装载程序(stage1)，其大小 512 字节，对应/boot/grub/stage1 文件，通常位于主引导扇区里面，对于硬盘就是 MBR 了。MBR(0 头 0 道 1 扇区)，前 446 字节是存放 stage1，后面跟的是硬盘分区表信息。前面提到了，BIOS 将 stage1 载入内存的 0x7c00 处并跳转执行；stage1(/stage1/start.S)的任务很单纯，就是利用 BIOS 指令将硬盘 0 头 0 道 2 扇区的读入内存。0 头 0 道 2 扇区的内容是源代码中的/stage2/start.S，编译后 512 字节，是 stage2 或 stage1.5 的入口。

stage2 叫做装载第二引导装载程序，这第二引导装载程序实际上是引出更高级的功能，以允许用户装载入一个特定的操作系统。在 GRUB 中，这步是让用户显示一个菜单或是输入命令。由于 stage2 很大，所以它一般位于文件系统之中(通常是 boot 所在的根分区)。

stage2 对启动系统起关键作用，该部分提供了 GRUB 启动菜单和交互式的 GRUB 的 shell。启动菜单在启动时候通过/boot/grub/grub.conf 文件所定义的内容生成。在启动菜单中选择了 kernel 之后，GRUB 会负责解压和装载 kernel image 并且将 initrd 也解压并装载到内存中。最后 GRUB 初始化 kernel 启动代码。完成之后后续的引导权被移交给 kernel。

上面还提到了 stage1.5 这个文件，它的作用是什么呢？ 到/boot/grub 目录下看看 fat_stage_1.5、e2fs_stage_1.5、xfs_stage_1.5 等等，很容易猜想 stage1.5 和文件系统有关系。有时候基本引导装载程序(stage1)不能识别 stage2 所在的文件系统分区，那么这时候就需要 stage1.5 来连接 stage1 和 stage2 了。因此对于不同的文件系统就会有不同的 stage1.5。但是对于我机器上的 grub 0.97 好像 stage1.5 并不是很重要，因为我尝试在没有 stage1.5 的情况下，我把 stage1 安装在软盘的引导扇区内，然后把 stage2 放在格式化成 ext2 或者 fat 格式的软盘内，启动的时候照常引导，并不需要 e2fs_stage_1.5 或者 fat_stage_1.5。

至于如何读扇区，有专门的 BIOS 汇编指令，感兴趣的可以查阅一下博文“ BIOS 系统服务 —— 直接磁盘服务 ”http://blog.csdn.net/yunsongice/archive/2010/10/08/5927560.aspx。下面是关于 grub 常用的几个指令对应的函数：

```
grub>root (hd0,0)                                  --root 指令为 grub 指定了一个根分区
grub>kernel /vmlinuz-2.6.18-194.el5          --kernel 指令将操作系统内核载入内存
grub>module /vmlinuz-2.6.18-194.el5xen ro root=/dev/sda2       --module 指令加载指定的模块
grub>initrd / initrd-2.6.18-194.el5.img          --指定 initrd 文件
grub>boot                                           --boot 指令调用相应的启动函数启动 OS 内核
```

当执行 grub>boot 指令后，grub 载入内核 vmlinuz 到内存的指定位置（后面会讲到 vmlinuz 加载到内存的物理地址），同时载入 initrd.img 到内存，系统启动的控制权移交给 kernel。kernel 会立即初始化系统中各设备并做相关配置工作，进入了“ 中世纪 ”。

对应内核映像，我还要多说几句。这里的 vmlinuz，"vm"表示 virtual memory，表示虚拟内存，z 表示 zip(压缩)。vmlinuz 的建立有两种方式。一是编译内核时通过“ make zImage ”创建，然后通过:“ cp /usr/src/linux-2.6.34/arch/x86/linux/boot/zImage /boot/vmlinuz ”产生。zImage 适用于小内核的情况，他的存在是为了向后的兼容性。二是内核编译时通过命令 make bzImage 创建，然后通过：“ cp/usr/src/linux-2.6.34/arch/x86/linux/boot/bzImage /boot/vmlinuz ”产生。bzImage 是压缩的内核映像。

需要注意，bzImage 不是用 bzip2 压缩的，bzImage 中的 bz 容易引起误解，bz 表示“ big zImage ”。zImage 和 bzImage 都是用 gzip 等压缩程序压缩的，具体什么压缩程序需要你编译时指定。所以，他们不仅是个压缩文件，而且在这两个文件的开头部分内嵌有解压缩代码。所以你不能用 gunzip 或 gzip ?dc 解压缩 vmlinuz。

注意到，最后一行还有个 initrd /initrd-2.6.18-194.el5.img 参数。initrd 是“ initial ramdisk ”的简写。initrd 一般被用来临时的引导硬件到实际内核 vmlinuz 能够接管并继续引导的状态。

参数中的 initrd-2.6.18-194.el5.img 主要是用于加载 ext3 等文件系统及 scsi 设备的驱动。

initrd 是一种基于内存的文件系统，启动过程中，系统在访问真正的根文件系统时，会先访问 initrd 文件系统。比如，使用的是 scsi 硬盘，而内核 vmlinuz 中并没有这个 scsi 硬件的驱动，那么在装入 scsi 模块之前，内核不能加载根文件系统，但 scsi 模块存储在根文件系统的 /lib/modules 下。为了解决这个问题，能引导一个能够读实际内核的 initrd 内核并用 initrd 修正 scsi 引导问题。initrd-2.6.18-194.el5.img 是用 gzip 压缩的文件，initrd 实现加载一些模块和安装文件系统等功能。

initrd 映象文件是使用 mkinitrd 命令创建的。mkinitrd 实用程式能够创建 initrd 映象文件。这个命令是 RedHat 专有的。其他 Linux 发行版或许有相应的命令。这是个非常方便的实用程序。具体情况请看帮助:man mkinitrd 下面的命令创建 initrd 映象文件。

2 内核映像的形成

前面提到了，grub 载入内核 vmlinuz 并加载到内存指定位置。别看这小小一句话，里边的学问可大着哩，要搞懂这里面的内幕，我们还要从内核编译说起。

我们知道，kernel 主要是由 C 语言和汇编语言交叉编译而形成的，其源代码来自网上，网址为 www.kernel.org。当你随便下一个内核版本并解压后，就可以看到里面的文件，要么是 C 代码（.c 或.h 文件），要么就是汇编代码（.S 文件）。这些文件都是内核代码的源文件，如果只有这些源文件的话，你没法把它拿到机器上去运行，因为我们的 PC 或服务器只认识二进制机器代码，所以，其中的编译过程就很重要的。所以，我们看到，除了这些源文件，内核源文件包里还有些没有后缀名的文件和一些 shell 脚本文件。这些都是辅助编译的文件，我们称之为内核代码辅助文件。

gcc 要把源文件编译成二进制代码，就需要这些辅助文件的帮助。下面我们就来说说整个过程。不过，直接列整个过程的意义并不大，所以我想把我的研究思路列出来，也许我的思路是错误的，也请大家不吝指正。

内核编译的步骤在网上已经出现得很多了，主要是以下五步：

1、make menuconfig：配置编译选项
2、make：编译及链接源代码，主要是些核心程序
3、make modules：编译及链接模块代码，主要是些驱动程序
4、make modules_install：安装模块到
5、make install：制作拷贝内核映像 bzImage 到/boot 目录下，并把名字换成 vmlinuz

2.1 MakeFile 预备知识

我们分析内核的主要任务是学习相关知识，前面我们学习了 BIOS 即系统启动相关的知识，这里学习一下 GUN make 的知识。

在 shell 的提示符号下，若输入“ make xxxx ”，则它会到目前的目录下找寻 Makefile 这个文件。然后依照 Makefile 中所记录的步骤一步一步的来执行。在我们写程序的时候，如果事先就把 compiler 程式所需要的步骤先写在 Makefile 中的话，想要 compiler 程序的时候就只要打入 make 的指令。

在我们的 Linux 内核项目文件中，有成百上千个源文件，每次修改其中的一个都需要全部重新编译是不可想象的事情，但通过编辑 Makefile 文件，利用 make 命令就可以只针对其中修改的源文件进行编译，而不需要全体编译。这就是 make 命令在编译项目文件时体现出来的优势。能做到这点，主要是基于 Makefile 文件的编写，和 make 命令对 Makefile 文件的调用。

Makefile 文件作为 make 命令的默认参数，使一个基于依赖关系编写的结构文件。

2.1.1 Makefile 书写规则

大家经常看到使用 make all、make install、make clean 等命令，包括我们这里配置编译选项所使用的 make menuconfig，而他们处理的目标都是一个 Makefile 文件，那么 all、install、clean 这些跟在 make 后面的参数是如何调用 Makefile 文件的运行呢？在这里，如果向上面的命令如果能够正确运行的话，那么在 Makefile 文件里一定有这样的几行，他们的以 all、install、clean 开始，我们叫做目标（Target）：

```
all:xxxx
        xxxxxxxxxxxx
install:xxxx
        xxxxxxxxxxxx
clean:xxxx
        xxxxxxxxxxxx
```

Makefile 文件的一般以以下方式组成：

```
target ... : prerequisites ...
      command
...
...
```

target 也就是一个目标文件，可以是 Object File，也可以是执行文件。还可以是一个标签（Label），对于标签这种特性，这个就是“伪目标”。

prerequisites 就是，要生成那个 target 所需要的文件或是目标。command 也就是 make 需要执行的命令。（任意的 Shell 命令）

这是一个文件的依赖关系，也就是说，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。**说白一点就是说，prerequisites 中如果有一个以上的文件比 target 文件要新的话，command 所定义的命令就会被执行。这就是 Makefile 的规则。也就是 Makefile 中最核心的内容。**

在默认的方式下，也就是我们只输入 make 命令。那么，

1、 make 会在当前目录下找名字叫“Makefile”或“makefile”的文件。
2、 如果找到，它会找文件中的第一个目标文件（target），并把这个文件作为最终的目标文件。
3、 如果第一个目标文件不存在，或是它所依赖的后面文件的修改时间要比这个文件新，那么，他就会执行后面所定义的命令来生成这个目标文件。
4、 如果第一个目标所依赖的文件也存在，那么 make 会在当前文件中找目标为该文件的依赖性，如果找到则再根据那一个规则生成文件。（这有点像一个堆栈的过程）
5、 当然，你的 C 文件和 H 文件是存在的啦，于是 make 会生成 .o 文件，然后再用 .o 文

件生命 make 的终极任务，也就是第一目标所对应的执行文件了。

Makefile 书写规则：
targets : prerequisites
    command
    ...

或是这样：
targets : prerequisites ; command
    command
    ...

targets 是文件名，以空格分开，可以使用通配符。一般来说，我们的目标基本上是一个文件，但也有可能是多个文件。

command 是命令行，如果其不与“ target:prerequisites ”在一行，那么，必须以[Tab 键]开头，如果和 prerequisites 在一行，那么可以用分号做为分隔。

prerequisites 也就是目标所依赖的文件（或依赖目标）。如果其中的某个文件要比目标文件要新，那么，目标就被认为是“ 过时的 ”，被认为是需要重生成的。这个在前面已经讲过了。

如果命令太长，你可以使用反斜框（‘ \ ’）作为换行符。make 对一行上有多少个字符没有限制。规则告诉 make 两件事，文件的依赖关系和如何成成目标文件。一般来说，make 会以 UNIX 的标准 Shell，也就是/bin/sh 来执行命令。

2.1.2 Makefile 变量

在 Makefile 中定义的变量，就像是 C/C++语言中的宏一样，他代表了一个文本字串，在 Makefile 中执行的时候其会自动原模原样地展开在所使用的地方。其与 C/C++所不同的是，你可以在 Makefile 中改变其值。在 Makefile 中，变量可以使用在“ 目标 ”，“ 依赖目标 ”，“ 命令 ”或是 Makefile 的其它部分中。

变量的命名字可以包含字符、数字，下划线（可以是数字开头），但不应该含有“ : ”、“ # ”、“ = ”或是空字符（空格、回车等）。变量是大小写敏感的，“ foo ”、“ Foo ”和“ FOO ”是三个不同的变量名。传统的 Makefile 的变量名是全大写的命名方式，但我推荐使用大小写搭配的变量名，如：MakeFlags。这样可以避免和系统的变量冲突，而发生意外的事情。

有一些变量是很奇怪字串，如“ $< ”、“ $@ ”等，这些是自动化变量，在内核的 Makefile 里，出现了很多自动化变量。在定义变量的值时，我们可以使用其它变量来构造变量的值，在 Makefile 中有两种方式来在用变量定义变量的值。

先看第一种方式，也就是简单的使用“ = ”号，在“ = ”左侧是变量，右侧是变量的值，右侧变量的值可以定义在文件的任何一处，也就是说，右侧中的变量不一定非要是已定义好的

值，其也可以使用后面定义的值。如：

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?
all:
	echo $(foo)
```

我们执行“ make all ”将会打出变量$(foo)的值是“ Huh? ”（ $(foo)的值是$(bar)，$(bar)的值是$(ugh)，$(ugh)的值是“ Huh? ”）可见，变量是可以使用后面的变量来定义的。

这个功能有好的地方，也有不好的地方，好的地方是，我们可以把变量的真实值推到后面来定义，如：

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

当“ CFLAGS ”在命令中被展开时，会是“ -Ifoo -Ibar -O ”。但这种形式也有不好的地方，那就是递归定义，如：

```
CFLAGS = $(CFLAGS) -O
```

或：

```
A = $(B)
B = $(A)
```

这会让 make 陷入无限的变量展开过程中去，当然，我们的 make 是有能力检测这样的定义，并会报错。还有就是如果在变量中使用函数，那么，这种方式会让我们的 make 运行时非常慢，更糟糕的是，他会使用得两个 make 的函数“ wildcard ”和“ shell ”发生不可预知的错误。因为你不会知道这两个函数会被调用多少次。

为了避免上面的这种方法，我们可以使用 make 中的另一种用变量来定义变量的方法。这种方法使用的是“ := ”操作符，如：

```
x := foo
y := $(x) bar
x := later
```

其等价于：

```
y := foo bar
x := later
```

值得一提的是，这种方法，前面的变量不能使用后面的变量，只能使用前面已定义好了的变量。如果是这样：

```
y := $(x) bar
x := foo
```

那么，y 的值是“ bar ”，而不是“ foo bar ”。

上面都是一些比较简单的变量使用了，让我们来看一个复杂的例子，其中包括了 make 的

函数、条件表达式和一个系统变量“ MAKELEVEL ”的使用：

```
ifeq (0,${MAKELEVEL})
cur-dir := $(shell pwd)
whoami := $(shell whoami)
host-type := $(shell arch)
MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}
endif
```

关于条件表达式和函数，我们在后面再说，对于系统变量“ MAKELEVEL ”，其意思是，如果我们的 make 有一个嵌套执行的动作，那么，这个变量会记录了我们的当前 Makefile 的调用层数。

下面再介绍两个定义变量时我们需要知道的，请先看一个例子，如果我们要定义一个变量，其值是一个空格，那么我们可以这样来：

```
nullstring :=
space := $(nullstring) # end of the line
```

nullstring 是一个 Empty 变量，其中什么也没有，而我们的 space 的值是一个空格。因为在操作符的右边是很难描述一个空格的，这里采用的技术很管用，先用一个 Empty 变量来标明变量的值开始了，而后面采用“ # ”注释符来表示变量定义的终止，这样，我们可以定义出其值是一个空格的变量。请注意这里关于“ # ”的使用，注释符“ # ”的这种特性值得我们注意，如果我们这样定义一个变量：

```
dir := /foo/bar       # directory to put the frobs in
```

dir 这个变量的值是“ /foo/bar ”，后面还跟了 4 个空格，如果我们这样使用这样变量来指定别的目录——“ $(dir)/file ”那么就完蛋了。

还有一个比较有用的操作符是“ ?= ”，先看示例：

```
FOO ?= bar
```

其含义是，如果 FOO 没有被定义过，那么变量 FOO 的值就是“ bar ”，如果 FOO 先前定义过，那么这条语将什么也不做，其等价于：

```
ifeq ($(origin FOO), undefined)
    FOO = bar
endif
```

还有一种设置变量值的方法是使用 define 关键字。使用 define 关键字设置变量的值可以有换行，这有利于定义一系列的命令（前面我们讲过“ 命令包 ”的技术就是利用这个关键字）。

define 指示符后面跟的是变量的名字，而重起一行定义变量的值，定义是以 endef 关键字结束。其工作方式和“ = ”操作符一样。变量的值可以包含函数、命令、文字，或是其它变量。因为命令需要以[Tab]键开头，所以如果你用 define 定义的命令变量中没有以[Tab]键开头，那么 make 就不会把其认为是命令。

下面的这个示例展示了 define 的用法：
define two-lines
echo foo
echo $(bar)
endef

我们还可以使用“ += ”操作符给变量追加值，如：
objects = main.o foo.o bar.o utils.o
objects += another.o

于是，我们的$(objects)值变成：“ main.o foo.o bar.o utils.o another.o ”（another.o 被追加进去了）

如果变量之前没有定义过，那么，“ += ”会自动变成“ = ”，如果前面有变量定义，那么“ += ”会继承于前次操作的赋值符。如果前一次的是“ := ”，那么“ += ”会以“ := ”作为其赋值符，如：
variable := value
variable += more

等价于：
variable := value
variable := $(variable) more

但如果是这种情况：
variable = value
variable += more

由于前次的赋值符是“ = ”，所以“ += ”也会以“ = ”来做为赋值，那么就不会发生变量的递归定义，这是很不好的，所以 make 中尽量不要用“ = ”，而多用“ := ”。

如果有变量是通常 make 的命令行参数设置的，那么 Makefile 中对这个变量的赋值会被忽略。如果你硬要在 Makefile 中设置这类参数的值，那么，你可以使用“ override ”指示符。其语法是：
override <variable> = <value>
override <variable> := <value>

当然，你还可以追加：
override <variable> += <more text>

对于多行的变量定义，我们用 define 指示符，在 define 指示符前，也同样可以使用 ovveride 指示符，如：
override define foo
bar
endef

2.1.3 条件判断

使用条件判断，可以让 make 根据运行时的不同情况选择不同的执行分支。条件表达式可以是比较变量的值，或是变量和常量的值。

下面的例子，判断$(CC)变量是否“ gcc ”，如果是的话，则使用 GNU 函数编译目标。

```
libs_for_gcc = -lgnu
normal_libs =
foo: $(objects)
ifeq ($(CC),gcc)
	$(CC) -o foo $(objects) $(libs_for_gcc)
else
	$(CC) -o foo $(objects) $(normal_libs)
endif
```

可见，在上面示例的这个规则中，目标“ foo ”可以根据变量“ $(CC) ”值来选取不同的函数库来编译程序。

我们可以从上面的示例中看到三个关键字：ifeq、else 和 endif。ifeq 的意思表示条件语句的开始，并指定一个条件表达式，表达式包含两个参数，以逗号分隔，表达式以圆括号括起。else 表示条件表达式为假的情况。endif 表示一个条件语句的结束，任何一个条件表达式都应该以 endif 结束。

当我们的变量$(CC)值是“ gcc ”时，目标 foo 的规则是：

```
foo: $(objects)
	$(CC) -o foo $(objects) $(libs_for_gcc)
```

而当我们的变量$(CC)值不是“ gcc ”时（比如“ cc ”），目标 foo 的规则是：

```
foo: $(objects)
	$(CC) -o foo $(objects) $(normal_libs)
```

当然，我们还可以把上面的那个例子写得更简洁一些：

```
libs_for_gcc = -lgnu
normal_libs =
ifeq ($(CC),gcc)
libs=$(libs_for_gcc)
else
libs=$(normal_libs)
endif
foo: $(objects)
$(CC) -o foo $(objects) $(libs)
```

条件表达式的语法为：
<conditional-directive>
<text-if-true>
endif

以及：
<conditional-directive>
<text-if-true>
else
<text-if-false>
endif

其中<conditional-directive>表示条件关键字，如“ ifeq ”。这个关键字有四种。
第一个是我们前面所见过的“ ifeq ”
ifeq (<arg1>, <arg2>)
ifeq '<arg1>' '<arg2>'
ifeq "<arg1>" "<arg2>"
ifeq "<arg1>" '<arg2>'
ifeq '<arg1>' "<arg2>"

比较参数“ arg1 ”和“ arg2 ”的值是否相同。当然，参数中我们还可以使用 make 的函数。如：
ifeq ($(strip $(foo)),)
<text-if-empty>
endif

这个示例中使用了“ strip ”函数，如果这个函数的返回值是空（Empty），那么<text-if-empty>就生效。

第二个条件关键字是“ ifneq ”。语法是：
ifneq (<arg1>, <arg2>)
ifneq '<arg1>' '<arg2>'
ifneq "<arg1>" "<arg2>"
ifneq "<arg1>" '<arg2>'
ifneq '<arg1>' "<arg2>"

其比较参数“ arg1 ”和“ arg2 ”的值是否相同，如果不同，则为真。和“ ifeq ”类似。第三个条件关键字是“ ifdef ”。语法是：
ifdef <variable-name>

如果变量<variable-name>的值非空，那到表达式为真。否则，表达式为假。当然，<variable-name>同样可以是一个函数的返回值。注意，ifdef 只是测试一个变量是否有值，其并不会把变量扩展到当前位置。还是来看两个例子：
示例一：

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

示例二：

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

第一个例子中，“ $(frobozz) ”值是“ yes ”，第二个则是“ no ”。

第四个条件关键字是“ ifndef ”。其语法是：

```
ifndef <variable-name>
```

这个我就不多说了，和“ ifdef ”是相反的意思。

在<conditional-directive>这一行上，多余的空格是被允许的，但是不能以[Tab]键做为开始（不然就被认为是命令）。而注释符“ # ”同样也是安全的。“ else ”和“ endif ”也一样，只要不是以[Tab]键开始就行了。

特别注意的是，make 是在读取 Makefile 时就计算条件表达式的值，并根据条件表达式的值来选择语句，所以，你最好不要把自动化变量（如“ $@ ”等）放入条件表达式中，因为自动化变量是在运行时才有的。

而且，为了避免混乱，make 不允许把整个条件语句分成两部分放在不同的文件中。

2.1.4 函数

在 Makefile 中可以使用函数来处理变量，从而让我们的命令或是规则更为的灵活和具有智能。make 所支持的函数也不算很多，不过已经足够我们的操作了。函数调用后，函数的返回值可以当做变量来使用。

函数调用，很像变量的使用，也是以“ $ ”来标识的，其语法如下：

```
$(<function> <arguments>)
```

或是

```
${<function> <arguments>}
```

这里，<function>就是函数名，make 支持的函数不多。<arguments>是函数的参数，参数间以逗号“ , ”分隔，而函数名和参数之间以“ 空格 ”分隔。函数调用以“ $ ”开头，以圆括号或花括号把函数名和参数括起。感觉很像一个变量，是不是？函数中的参数可以使用变量，为了风格的统一，函数和变量的括号最好一样，如使用“ $(subst a,b,$(x)) ”这样的形式，而不是“ $(subst a,b,${x}) ”的形式。因为统一会更清楚，也会减少一些不必要的麻烦。

还是来看一个示例：

```
comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
```

在这个示例中，$(comma)的值是一个逗号。$(space)使用了$(empty)定义了一个空格，$(foo)的值是“ a b c ”，$(bar)的定义用，调用了函数“ subst ”，这是一个替换函数，这个函数有三个参数，第一个参数是被替换字串，第二个参数是替换字串，第三个参数是替换操作作用的字串。这个函数也就是把$(foo)中的空格替换成逗号，所以$(bar)的值是“ a,b,c ”。

2.1.5 隐含规则

如果要使用隐含规则生成你需要的目标，你所需要做的就是不要写出这个目标的规则。那么，make 会试图去自动推导产生这个目标的规则和命令，如果 make 可以自动推导生成这个目标的规则和命令，那么这个行为就是隐含规则的自动推导。当然，隐含规则是 make 事先约定好的一些东西。例如，我们有下面的一个 Makefile：

```
foo : foo.o bar.o
    cc –o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

我们可以注意到，这个 Makefile 中并没有写下如何生成 foo.o 和 bar.o 这两目标的规则和命令。因为 make 的“ 隐含规则 ”功能会自动为我们自动去推导这两个目标的依赖目标和生成命令。

make 会在自己的“ 隐含规则 ”库中寻找可以用的规则，如果找到，那么就会使用。如果找不到，那么就会报错。在上面的那个例子中，make 调用的隐含规则是，把[.o]的目标的依赖文件置成[.c]，并使用 C 的编译命令“ cc －c $(CFLAGS) [.c] ”来生成[.o]的目标。也就是说，我们完全没有必要写下下面的两条规则：

```
foo.o : foo.c
    cc –c foo.c $(CFLAGS)
bar.o : bar.c
    cc –c bar.c $(CFLAGS)
```

因为，这已经是“ 约定 ”好了的事了，make 和我们约定好了用 C 编译器“ cc ”生成[.o]文件的规则，这就是隐含规则。

当然，如果我们为[.o]文件书写了自己的规则，那么 make 就不会自动推导并调用隐含规则，它会按照我们写好的规则忠实地执行。

还有，在 make 的“ 隐含规则库 ”中，每一条隐含规则都在库中有其顺序，越靠前的则是越被经常使用的，所以，这会导致我们有些时候即使我们显示地指定了目标依赖，make 也不会管。如下面这条规则（没有命令）：
foo.o : foo.p

依赖文件“ foo.p ”（Pascal 程序的源文件）有可能变得没有意义。如果目录下存在了“ foo.c ”文件，那么我们的隐含规则一样会生效，并会通过“ foo.c ”调用 C 的编译器生成 foo.o 文件。因为，在隐含规则中，Pascal 的规则出现在 C 的规则之后，所以，make 找到可以生成 foo.o 的 C 的规则就不再寻找下一条规则了。如果你确实不希望任何隐含规则推导，那么，你就不要只写出“ 依赖规则 ”，而不写命令。

在隐含规则中的命令中，基本上都是使用了一些预先设置的变量。你可以在你的 makefile 中改变这些变量的值，或是在 make 的命令行中传入这些值，或是在你的环境变量中设置这些值，无论怎么样，只要设置了这些特定的变量，那么其就会对隐含规则起作用。当然，你也可以利用 make 的“ -R ”或“ --no－builtin-variables ”参数来取消你所定义的变量对隐含规则的作用。

例如，第一条隐含规则——编译 C 程序的隐含规则的命令是
“ $(CC) －c $(CFLAGS) $(CPPFLAGS) ”。

Make 默认的编译命令是“ cc ”，如果你把变量“ $(CC) ”重定义成“ gcc ”，把变量“ $(CFLAGS) ”重定义成“ -g ”，那么，隐含规则中的命令全部会以“ gcc －c –g $(CPPFLAGS) ”的样子来执行了。

我们可以把隐含规则中使用的变量分成两种：一种是命令相关的，如“ CC ”；一种是参数相的关，如“ CFLAGS ”。

2.1.6 定义模式规则

你可以使用模式规则来定义一个隐含规则。一个模式规则就好像一个一般的规则，只是在规则中，目标的定义需要有“ % ”字符。“ % ”的意思是表示一个或多个任意字符。在依赖目标中同样可以使用“ % ”，只是依赖目标中的“ % ”的取值，取决于其目标。

有一点需要注意的是，“ % ”的展开发生在变量和函数的展开之后，变量和函数的展开发生在 make 载入 Makefile 时，而模式规则中的“ % ”则发生在运行时。

1、模式规则介绍

模式规则中，至少在规则的目标定义中要包含“ % ”，否则，就是一般的规则。目标中的“ % ”

定义表示对文件名的匹配，“ % ” 表示长度任意的非空字符串。例如：“ %.c ” 表示以 “ .c ” 结尾的文件名（文件名的长度至少为 3），而 “ s.%.c ” 则表示以 “ s. ” 开头，“ .c ” 结尾的文件名（文件名的长度至少为 5）。

如果 “ % ” 定义在目标中，那么，目标中的 “ % ” 的值决定了依赖目标中的 “ % ” 的值，也就是说，目标中的模式的 “ % ” 决定了依赖目标中 “ % ” 的样子。例如有一个模式规则如下：
%.o : %.c ; <command .....>

其含义是，指出了怎么从所有的[.c]文件生成相应的[.o]文件的规则。如果要生成的目标是‘ a.o b.o ”，那么 “ %c ” 就是 “ a.c b.c ”。

一旦依赖目标中的 “ % ” 模式被确定，那么，make 会被要求去匹配当前目录下所有的文件名，一旦找到，make 就会规则下的命令，所以，在模式规则中，目标可能会是多个的，如果有模式匹配出多个目标，make 就会产生所有的模式目标，此时，make 关心的是依赖的文件名和生成目标的命令这两件事。

2、模式规则示例

下面这个例子表示了,把所有的[.c]文件都编译成[.o]文件.
%.o : %.c
$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

其中，“ $@ ” 表示所有的目标的挨个值，“ $< ” 表示了所有依赖目标的挨个值。这些奇怪的变量我们叫 “ 自动化变量 ”，后面会详细讲述。

下面的这个例子中有两个目标是模式的：
%.tab.c %.tab.h: %.y
bison -d $<

这条规则告诉 make 把所有的[.y]文件都以 “ bison －d <n>.y ” 执行，然后生成 “ <n>.tab.c ” 和 “ <n>.tab.h ” 文件。（其中，“ <n> ” 表示一个任意字符串）。如果我们的执行程序 “ foo ” 依赖于文件 “ parse.tab.o ” 和 “ scan.o ”，并且文件 “ scan.o ” 依赖于文件 “ parse.tab.h ”，如果 “ parse.y ” 文件被更新了，那么根据上述的规则，“ bison - d parse.y ” 就会被执行一次，于是，“ parse.tab.o ” 和 “ scan.o ” 的依赖文件就齐了。（假设，“ parse.tab.o ” 由 “ parse.tab.c ” 生成，和 “ scan.o ” 由 “ scan.c ” 生成，而 “ foo ” 由 “ parse.tab.o ” 和 “ scan.o ” 链接生成，而且 foo 和其[.o]文件的依赖关系也写好，那么，所有的目标都会得到满足）

3、自动化变量

在上述的模式规则中，目标和依赖文件都是一系例的文件，那么我们如何书写一个命令来完成从不同的依赖文件生成相应的目标？因为在每一次的对模式规则的解析时，都会是不同的目标和依赖文件。

自动化变量就是完成这个功能的。在前面，我们已经对自动化变量有所提涉，相信你看到这

里已对它有一个感性认识了。所谓自动化变量，就是这种变量会把模式中所定义的一系列的文件自动地挨个取出，直至所有的符合模式的文件都取完了。这种自动化变量只应出现在规则的命令中。

下面是所有的自动化变量及其说明：

$@
表示规则中的目标文件集。在模式规则中，如果有多个目标，那么，“ $@ ”就是匹配于目标中模式定义的集合。

$%
仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是“ foo.a(bar.o) ”，那么，“ $% ”就是“ bar.o ”，“ $@ ”就是“ foo.a ”。如果目标不是函数库文件（Unix 下是[.a]，Windows 下是[.lib]），那么，其值为空。

$<
依赖目标中的第一个目标名字。如果依赖目标是以模式（即“ % ”）定义的，那么“ $< ”将是符合模式的一系列的文件集。注意，它是一个一个取出来的。

$?
所有比目标新的依赖目标的集合。以空格分隔。

$^
所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那个这个变量会去除重复的依赖目标，只保留一份。

$+
这个变量很像“ $^ ”，也是所有依赖目标的集合。只是它不去除重复的依赖目标。

$*
这个变量表示目标模式中“ % ”及其之前的部分。如果目标是“ dir/a.foo.b ”，并且目标的模式是“ a.%.b ”，那么，“ $* ”的值就是“ dir/a.foo ”。这个变量对于构造有关联的文件名是比较有较。如果目标中没有模式的定义，那么“ $* ”也就不能被推导出，但是，如果目标文件的后缀是 make 所识别的，那么“ $* ”就是除了后缀的那一部分。例如：如果目标是“ foo.c ”，因为“ .c ”是 make 所能识别的后缀名，所以，“ $* ”的值就是“ foo ”。

这个特性是 GNU make 的，很有可能不兼容于其它版本的 make，所以，你应该尽量避免使用“ $* ”，除非是在隐含规则或是静态模式中。如果目标中的后缀是 make 所不能识别的，那么“ $* ”就是空值。

当你希望只对更新过的依赖文件进行操作时，“ $? ”在显式规则中很有用，例如，假设有一个函数库文件叫“ lib ”，其由其它几个 object 文件更新。那么把 object 文件打包的比较有效率的 Makefile 规则是：
lib : foo.o bar.o lose.o win.o

ar r lib $?

在上述所列出来的自动量变量中。四个变量（$@、$<、$%、$*）在扩展时只会有一个文件，而另三个的值是一个文件列表。这七个自动化变量还可以取得文件的目录名或是在当前目录下的符合模式的文件名，只需要搭配上“ D ”或“ F ”字样。这是 GNU make 中老版本的特性，在新版本中，我们使用函数“ dir ”或“ notdir ”就可以做到了。“ D ”的含义就是 Directory，就是目录，“ F ”的含义就是 File，就是文件。

下面是对于上面的七个变量分别加上“ D ”或是“ F ”的含义：

$(@D)
表示“ $@ ”的目录部分（不以斜杠作为结尾），如果“ $@ ”值是“ dir/foo.o ”，那么“ $(@D) ”就是“ dir ”，而如果“ $@ ”中没有包含斜杠的话，其值就是“ . ”（当前目录）。

$(@F)
表示“ $@ ”的文件部分，如果“ $@ ”值是“ dir/foo.o ”，那么“ $(@F) ”就是“ foo.o ”，“ $(@F) ”相当于函数“ $(notdir $@) ”。

“$(*D)”
“$(*F)”
和上面所述的同理，也是取文件的目录部分和文件部分。对于上面的那个例子，“ $(*D) ”返回“ dir ”，而“ $(*F) ”返回“ foo ”。

“$(%D)”
“$(%F)”
分别表示了函数包文件成员的目录部分和文件部分。这对于形同“ archive(member) ”形式的目标中的“ member ”中包含了不同的目录很有用。

“$(<D)”
“$(<F)”
分别表示依赖文件的目录部分和文件部分。

“$(^D)”
“$(^F)”
分别表示所有依赖文件的目录部分和文件部分。（无相同的）

“$(+D)”
“$(+F)”
分别表示所有依赖文件的目录部分和文件部分。（可以有相同的）

“$(?D)”
“$(?F)”
分别表示被更新的依赖文件的目录部分和文件部分。

最后想提醒一下的是，对于“ $< ”，为了避免产生不必要的麻烦，我们最好给$后面的那个特定字符都加上圆括号，比如，“ $(<) ”就要比“ $< ”要好一些。

还得要注意的是，这些变量只使用在规则的命令中，而且一般都是“ 显式规则 ”和“ 静态模式规则 ”（参见前面“ 书写规则 ”一章）。其在隐含规则中并没有意义。

4、模式的匹配

一般来说，一个目标的模式有一个有前缀或是后缀的“ % ”，或是没有前后缀，直接就是一个“ % ”。因为“ % ”代表一个或多个字符，所以在定义好了的模式中，我们把“ % ”所匹配的内容叫做“ 茎 ”，例如“ %.c ”所匹配的文件“ test.c ”中“ test ”就是“ 茎 ”。因为在目标和依赖目标中同时有“ % ”时，依赖目标的“ 茎 ”会传给目标，当做目标中的“ 茎 ”。

当一个模式匹配包含有斜杠（实际也不经常包含）的文件时，那么在进行模式匹配时，目录部分会首先被移开，然后进行匹配，成功后，再把目录加回去。在进行“ 茎 ”的传递时，我们需要知道这个步骤。例如有一个模式“ e%t ”，文件“ src/eat ”匹配于该模式，于是“ src/a ”就是其“ 茎 ”，如果这个模式定义在依赖目标中，而被依赖于这个模式的目标中又有个模式“ c%r ”，那么，目标就是“ src/car ”。（“ 茎 ”被传递）

5、重载内建隐含规则

你可以重载内建的隐含规则（或是定义一个全新的），例如你可以重新构造和内建隐含规则不同的命令，如：

```
%.o : %.c
$(CC) –c $(CPPFLAGS) $(CFLAGS) -D$(date)
```

你可以取消内建的隐含规则，只要不在后面写命令就行。如：

```
%.o : %.s
```

同样，你也可以重新定义一个全新的隐含规则，其在隐含规则中的位置取决于你在哪里写下这个规则。朝前的位置就靠前。

关于 Makefile 的预备知识先说到这里，有了上面的这些知识大家就可以上手了，其他的那些重要的知识主要涉及到一些细节，等我们遇到了再去查 GNU Make 手册（注意，这是一种很重要的学习方法）。

2.1 KBuild 体系

从 Linux 内核 2.6 开始，Linux 内核的编译采用 Kbuild 系统，这和过去的编译系统有很大的不同，尤其对于 Linux 内核模块的编译。在新的系统下，Linux 编译系统会两次扫描 Linux 的 Makefile：首先编译系统会读取 Linux 内核顶层的 Makefile，然后根据读到的内容第二次读取 Kbuild 的 Makefile 来编译 Linux 内核。

Kbuild 是建立在 GUN make 机制上的一种编译体系，理解 KBuild 体系，首先要理解 GUN make，这也就是为什么要在前一节大谈特谈 Makefile 预备知识的原因。

那么 Linux 内核的 KBuild 体系中用到的 Makefile 分为 5 个部分：

（1）顶层 Makefile

Kernel Makefile 位于 Linux 内核源代码的顶层目录，也叫 Top Makefile。它主要用于指定编译 Linux Kernel 目标文件（vmlinux）和模块（module）。这编译内核或模块是，这个文件会被首先读取，并根据读到的内容配置编译环境变量。对于内核或驱动开发人员来说，这个文件几乎不用任何修改。

（2）.config

内核配置文件，当配置完 menuconfig 以后，就会在主目录下生成一个.config 文件，我们再来看看这个文件的部分内容：

……
CONFIG_NEED_PER_CPU_EMBED_FIRST_CHUNK=y
CONFIG_NEED_PER_CPU_PAGE_FIRST_CHUNK=y
CONFIG_HAVE_CPUMASK_OF_CPU_MAP=y
……

（3）arch/$(ARCH)/Makefile

具体 CPU 架构的 Makefile，位于 ARCH/$(ARCH)/Makefile，是系统对应平台的 Makefile。Kernel Top Makefile 会包含这个文件来指定平台相关信息。只有平台开发人员会关心这个文件。

（4）scripts/Makefile.*

Kbuild 使用到通用的规则等，面向所有的 Kbuild Makefiles，包含了所有的定义、规则等信息。这些文件被用来编译基于 kbuild Makefile 的内核。

（5）kbuild Makefiles

内核源代码中大约有上百个这样的文件，一般是每个目录一个 Makefile，同它对应的有一个 Kconfig，其内容是一些默认的编译选项。每一个子目录都有一个 Kbuild Makefile 文件，用来执行从其上层目录传递下来的命令。注意，Kbuild Makefile 并不直接被当做 Makefile 执行，而是从.config 文件中提取信息，生成 Kbuild 完成内核编译所需的文件列表。

2.1.1 内核目标

Kbuild Makefile 的最核心的部分是内核目标的定义。主要是定义需要编译的文件，所有的选项，以及到哪些子目录去执行递归操作。

例如，源代码某目录中有一个 foo.c，要编译成一对象，那么该目录中的 Kbuild Makefile 其中必有一行形式如下：
obj-$(CONFIG_FOO) += foo.o

$(CONFIG_FOO)来自.config 内核配置文件，可以为 y(编译进内核) 或 m(编译成模块)。如果 CONFIG_FOO 不是 y 和 m，那么该文件就不会被编译联接了。

（1）编译进内核

Kbuild Makefile 规定所有编译进内核的目标文件都存在$(obj-y)列表中。而这些列表依赖内核的配置。

Kbuild 编译所有的$(obj-y)文件。然后，调用"$(LD) -r"将它们合并到一个 build-in.o 文件中。稍后，该 build-in.o 会被顶层 Makefile 链接进 vmlinux 中。

注意，$(obj-y)中的文件是有顺序的。列表中有重复项是可以的，因为当第一个文件被联接到 built-in.o 中后，其余文件就被忽略了。

另外，链接也是有顺序的，那是因为有些函数(module_init()/__initcall)将会在启动时按照他们出现的顺序进行调用。所以，记住改变链接的顺序可能改变你系统顺序，从而导致你的硬件损害，这一点千万要注意。

（2）编译成模块

编译成模块和内核的区别不用我多说，因为模块是可装载，通过 inmod 等命令。$(obj-m) 列举出了哪些文件要编译成可装载模块。一个模块可以由一个文件或多个文件编译而成。如果是一个源文件，KbuildMakefile 只需简单的将其加到$(obj-m)中去就可以了。

如果内核模块是由多个源文件编译而成，那你就要采用一个方法去声明你所要编译的模块：
obj-$(CONFIG_FOO) += isdn.o

解释一下这个方法，很简单，Kbuild 需要知道你所编译的模块是基于哪些文件，所以你需要通过变量$(<module_name>-objs)来告诉它：
#drivers/isdn/i4l/Makefile
obj-$(CONFIG_ FOO) += isdn.o
isdn-objs := isdn_net_lib.o isdn_v110.o isdn_common.o

上面的例子中，isdn.o 就是模块名，Kbuild 将编译在$(isdn-objs)中列出的所有文件，然后使

用"$(LD) -r"生成 isdn.o。

Kbuild 能够识别用于组成目标文件的后缀-objs 和后缀-y。这就让 KbuildMakefile 可以通过使用 CONFIG_ 符号来判断该对象是否是用来组合对象的：

```
#fs/ext2/Makefile
obj-$(CONFIG_EXT2_FS) += ext2.o
ext2-y := balloc.o bitmap.o
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o
```

在这个例子中，如果 $(CONFIG_EXT2_FS_XATTR) 是 'y'，xattr.o 将是复合对象 ext2.o 的一部分。

注意：当然，当你要将其编译进内核时，上面的语法同样适用。所以，如果你的 CONFIG__EXT2__FS=y，那 Kbuild 会按你所期望的那样，生成 ext2.o 文件，然后将其联接到 built-in.o 中。

（3）编译成库文件

在 obj-* 中所列文件是用来编译成模块，或者是链接到特定目录中的 built-in.o 以编译进内核。同样，也可以列出一些将被包含在 lib.a 库中的文件。在 lib-y 中所列出的文件用来组成该目录下的一个库文件。

在 obj-y 与 lib-y 中同时列出的文件，因为都是可以访问的，所以该文件是不会被包含在库文件中的。而同样的情况， lib-m 中的文件就要包含在 lib.a 库文件中。

注意，一个 Kbuild makefile 可以同时列出要编译进内核的文件与要编译成库的文件。所以，在一个目录里可以同时存在 built-in.o 与 lib.a 两个文件：

```
#arch/x86/lib/Makefile
lib-y := chechsum.o delay.o
```

这将由 checksum.o 和 delay.o 两个文件创建一个库文件 lib.a。为了让 Kbuild 真正认识到这里要有一个库文件 lib.a 要创建，其所在的目录要加到 libs-y 列表中。lib-y 使用一般限制在 lib/ 和 arch/*/lib 中。

（4）递归向下访问目录

一个 Kbuild Makefile 只对编译所在目录的对象负责。在子目录中的文件的编译要由其所在的子目录的 Makefile 来管理。只要你让 Kbuild 知道它应该递归操作，那么该系统就会在其子目录中自动的调用 make 递归操作。这就是 obj-y 和 obj-m 的作用。

例如，ext2 被放的一个单独的目录下，在 fs 目录下的 Makefile 会告诉 Kbuild 使用下面的赋值进行向下递归操作：

```
#fs/Makefile
obj-$(CONFIG_EXT2_FS) += ext2/
```

如果 CONFIG_EXT2_FS 被设置为 'y'(编译进内核)或是'm'(编译成模块)，相应的 obj- 变量就会被设置，并且 Kbuild 就会递归向下访问 ext2 目录。Kbuild 只是用这些信息来决定它是否需要访问该目录，而具体怎么编译由该目录中的 Makefile 来决定。

注意，将 CONFIG_变量设置成目录名是一个好的编程习惯。这让 Kbuild 在完全忽略那些相应的 CONFIG_值不是'y'和'm'的目录。

2.1.2 主机程序

在内核编译阶段，Kbuild 要先去编译那些将在编译阶段使用的工具文件。为了使用该可执行文件，要将编译分成二个阶段：
第一阶段是告诉 Kbuild 存在哪些可执行文件。这是通过变量 hostprogs-y 来完成的。
第二阶段是添加一个对可执行文件的显性依赖。有两种方法:增加依赖关系到一个规则中，或是利用变量 $(always)。以下是详细叙述：

（1）简单的主机程序

在编译内核时，有时会需要编译并运行一个程序。下面这行就告诉了 kbuild，程序 bin2hex 用来在编译阶段被执行：
hostprogs-y := bin2hex

在上面的例子中，Kbuild 假设 bin2hex 是由一个与其在同一目录下，名为 bin2hex.c 的 C 语言源文件编译而成的。

（2）复合的主机程序

本机程序可以由多个文件编译而成。所使用的语法与内核的相应语法很相似。$(<executeable>-objs) 列出了联接成最后的可执行文件所需的所有目标文件：
#scripts/lxdialog/Makefile
hostprogs-y := lxdialog
lxdialog-objs := checklist.o lxdialog.o

扩展名为.o 的文件是从相应的.c 文件编译而来的。在上面的例子中，checklist.c 编译成了 checklist.o，lxdialog.c 编译成了 lxdialog.o。最后，两个.o 文件联接成了一可执行文件，lxdialog。

注意：语法 <executable>-y 不是只能用来生成本机程序。

（3）定义共享库

扩展名为 so 的文件称为共享库，被编译成位置无关对象。Kbuild 也支持共享库，但共享库的使用很有限。在下面的例子中，libconfig.so 共享库用来联接到可执行文件 conf 中：
#scripts/kconfig/Makefile

```
hostprogs-y := conf
conf-objs := conf.o libkconfig.so
libkcofig-objs := expr.o type.o
```

共享库文件经常要求一个相应的 -objs，在上面的例子中，共享库 libkconfig 是由 expr.o 和 type.o 两个文件组成的。expr.o 和 type.o 将被编译成位置无关码，然后联接成共享库文件 libkconfig.so。注意，KBuild 还支持用 C++写的主机程序，但 C++并不支持共享库。

2.1.3 编译标志

KBuild 体系还有一个重要的概念，那就是编译标志。我们可以在一些 Makefile 中看到如下标志：
EXTRA_CFLAGS、 EXTRA_AFLAGS、 EXTRA_LDFLAGS、EXTRA_ARFLAGS

这些 EXTRA_开头的大写字母变量都是编译标志，所有的 EXTRA_ 变量只在所定义的 Kbuild Makefile 中起作用。EXTRA_ 变量可以在 Kbuild Makefile 中所有命令中使用。

$(EXTRA_CFLAGS) 是用 $(CC) 编译 C 源文件时的选项，例如：

```
# drivers/sound/emu10kl/Makefile
EXTRA_CFLAGS += -I$(obj)
ifdef DEBUG
EXTRA_CFLAGS += -DEMU10KL_DEBUG
endif
```

由于顶层 Makefile 的$(CC)拥有变量 $(CFLAGS) 用来作为整个源代码树的编译选项，所以在这里做这么一个设置就是将$(CFLAGS)替换成 EXTRA_CFLAGS。

$(EXTRA_AFLAGS) 也是一个针对每个目录的选项，只不过它是用来编译汇编源代码的：

```
#arch/x86/kernel/Makefile
EXTRA_AFLAGS := -traditional
```

$(EXTRA_LDFLAGS) 和 $(EXTRA_ARFLAGS)分别与 $(LD)和 $(AR)类似，只不过，他们是针对每个目录的：

```
#arch/m68k/fpsp040/Makefile
EXTRA_LDFLAGS := -x
```

CFLAGS_$@, AFLSGA_$@
CFLAGS_$@ 和 AFLAGS_$@ 只能在当前 Kbuild Makefile 中的命令中使用。
$(CFLAGS_$@) 是 $(CC) 针对每个文件的选项，而不是目录。$@ 表明了具体操作的文件：

```
# drivers/scsi/Makefile
CFLAGS_aha152x.o = -DAHA152X_STAT -DAUTOCONF
CFLAGS_gdth.o = # -DDEBUG_GDTH=2 -D__SERIAL__ -D__COM2__ \
-DGDTH_STATISTICS
```

CFLAGS_seagate.o = -DARBITRATE -DPARITY -DSEAGATE_USE_ASM

以上三行分别设置了 aha152x.o，gdth.o 和 seagate.o 的编辑选项。

$(AFLAGS_$@) 也类似，只不是是针对汇编语言的。
arch/arm/kernel/Makefile
AFLAGS_head-armv.o := -DTEXTADDR=$(TEXTADDR) -traditional
AFLAGS_head-armo.o := -DTEXTADDR=$(TEXTADDR) -traditional

注意，Kbuild 跟踪在以下方面依赖：
1) 所有要参与编译的文件(所有的.c 和.h 文件)
2) 在参与编译文件中所要使用的 CONFIG_ 选项
3) 用于编译目标的命令行

因此，如果你改变了 $(CC) 的编译选项，所有受影响的文件都要重新编译。

2.2 内核编译分析

关于 Kbuild 的预备知识先说到这里，有了上面的这些知识大家就可以上手了，其他的那些重要的知识主要涉及到一些细节，等我们遇到了再去查 GNU Make 手册和 linux-2.6.34.1\Documentation\kbuild 下的那些文档（注意，这是一种很重要的学习方法）。

在递归访问目录之前，顶层 Makefile 要完成设置环境变量以及递归访问的准备工作。顶层 Makefile 包含的公共部分，而 arch/$(ARCH)/Makefile 包含着针对某一特定 CPU 架构的配置信息。所以，要在 arch/$(ARCH)/Makefile 中设置一部分变量，并定义一些目标。

Kbuild 执行的几个步骤(大致)：
1) 根据内核配置生成文件 .config
2) 将内核的版本号存储在 include/linux/version.h
3) 生成指向 include/asm-$(ARCH) 的符号链接
4) 更新所有编译所需的文件： 附加的文件由 arch/$(ARCH)/Makefile 指定。
5) 递归向下访问所有在下列变量中列出的目录： init-* core* drivers-* net-* libs-*，并编译生成目标文件。这些变量的值可以在 arch/$(ARCH)/Makefile 中扩充。
6) 联接所有的目标文件，在源代码树顶层目录中生成 vmlinux。最先联接是在 head-y 中列出的文件，该变量由 arch/$(ARCH)/Makefile 赋值。
7) 最后完成具体架构的特殊要求，并压缩成最终的内核映像。
   -- 包含生成启动指令
   -- 准备 initrd 镜像或类似文件

2.2.1 编译配置

那么针对内核，我们从哪里下手呢？我们是在内核源代码最高目录 linux-2.6.34.1 上运行的 make menuconfig，那么就从 KBuild 的顶层——linux-2.6.34.1/Makefile 文件下手。

在这个文件找“ menuconfig: ”这一行。可是找不到怎么办，难道我们的思路出了问题？大家在学习代码的时候，如果一条路走不通了，可千万别急，有百度呢。查一查，GNU make 手册，Makefile 跟我们的 C 语言一样，可以 include 呢。也就是说，可以把其他 Makefile 嵌入进来。所以，我们再在该文件下搜一搜所以的 include 关键字，有这么几行：

309 include $(srctree)/scripts/Kbuild.include
452 include $(srctree)/arch/$(SRCARCH)/Makefile
486 -include include/config/auto.conf
491 -include include/config/auto.conf.cmd
535 include $(srctree)/arch/$(SRCARCH)/Makefile

注意到 srctree 宏，我们可以猜到就是当前目录，SRCARCH 是 x86，再加上我们只关注 Makefile 文件，所以我们就只关注/usr/src/kernels/linux-2.6.34.1/arch/x86/Makefile。搜一下，还是没有。这下我怀疑真的是我的思路错了。别着急，继续百度。终于在一个论坛上查到了，linux-2.6.34.1/Makefile 文件的 459 行有段这样的代码：

```
459 %config: scripts_basic outputmakefile FORCE
460         $(Q)mkdir -p include/linux include/config
461         $(Q)$(MAKE) $(build)=scripts/kconfig $@
```

“ % ”就是前面讲到的模式规则，这里就用到了。所有以 config 结尾的目标（如：menuconfig xconfig gconfig）都采用这个规则。另外，$(Q)在 286 行定义，它根据 KBUILD_VERBOSE 是否设置而定义，如果设置就是空的，没有设置就是“ @ ”。(Q)(MAKE)这两个宏组合在一起，就是@make，我们知道，就是不把命令详细信息打印出来。所以 make menuconfig 最终会运行@make $(build)=scripts/kconfig menuconfig 命令。根据前面的知识，$@就是指目标 menuconfig。至于后面的$(build)变量是什么？待会儿再讲。

这个规则有三个依赖：scripts_basic、outputmakefile、FORCE。下面看一下这三个依赖：
我们从最简单的开始，首先分析一下这个 FORCE 依赖，它的规则定式义在 1553 行：
FORCE:

这个规则没有命令也没有依赖，只做一个.PHONY: $(PHONY)动作。这里要回顾一下 GUN make 的隐含规则。我们知道，make 的“ 隐含规则 ”功能会自动为我们自动去推导目标的依赖目标和生成命令。如果推导出来的目标（是什么，不知道，make 的那么多隐含规则，谁也说不清楚）与伪目标（如%config）同名，也就是已经存在于系统中了，那么伪目标中的命令就不会被执行。而执行 FORCE 这个依赖的作用就是在执行此规则时，目标 FORCE 总会被认为是最新的。这样当它作为其它规则的依赖时，因为依赖总被认为被更新过的，所以那个%config 目标中定义的命令总会被执行。其实我们也可以从 force 这个单词的中文意思中猜测（分析内核就需要这种连蒙带猜的精神）。

再来看看 scripts_basic 这个依赖的规则在 384 行定义：

```
PHONY += scripts_basic
scripts_basic:
	$(Q)$(MAKE) $(build)=scripts/basic
	$(Q)rm -f .tmp_quiet_recordmcount
```

注意，PHONY 这个变量是我们第一次见到，在内核源码中的 Makefile 随处可见，它的作用在最后一行定义：.PHONY: $(PHONY)，表示 PHONY 变量中所有的目标都是伪目标，而不是具体的文件，所以，这里的 scripts_basic 就是个伪目标。

下面来讲 build 这个变量，它定义在我们刚才 include 的 scripts/kbuild.include 的 114 行：

```
build := -f $(if $(KBUILD_SRC),$(srctree)/)scripts/Makefile.build obj
```

KBUILD_SRC 是当前目录，后面会讲到，所以 srctree 参数也是当前目录，所以省去，上面的规则可写成如下形式：

```
scripts_basic:
		@make –f scripts/Makefile.build obj=scripts/basic
```

这个规则的命令最终会进入 scripts 目录，执行 Makefile.build 文件（传说中的 Kbuild 脚本），并传递参数 obj=scripts/basic。

最后再来看 outputmakefile，参数构建规则在 396 行开始定义：

```
outputmakefile:
ifneq ($(KBUILD_SRC),)
	$(Q)ln -fsn $(srctree) source
	$(Q)$(CONFIG_SHELL) $(srctree)/scripts/mkmakefile \
	    $(srctree) $(objtree) $(VERSION) $(PATCHLEVEL)
endif
```

这个规则的命令运行一个 shell 脚本 scripts/mkmakefile，并传递四个参数。这个脚本主要是在$(objtree)参数指定的目录中生成一个 Makefile 文件。由于这里 KBUILD_SRC 为空，所以这个脚本并不会被执行。

在他的依赖被处理完后，开始执行规则的命令。第一个命令创建了两个目录，第二个命令扩展后为：

```
@make  –f  scripts/Makefile.build  obj=scripts/kconfig  menuconfig
```

这个命令依然是执行 scripts/Makefile.build 这个 makefile 文件。并传递参数 obj=scripts/kconfig 和 menuconfig。

scripts/Makefile.build 脚本的具体代码我就不去分析了，但必须知道上面命令的原理，这是 KBuild 机制的重中之重。前办部分是@make –f scripts/Makefile.build，这整个部分就是 KBuild 的主题命令名称；obj=scripts/kconfig 表示对应目录，KBuild 只关心该目录中的 Makefile 文件；menuconfig 是传进去的参数，意思是执行 Makefile 里面 menuconfig 的目标，我们看到

在 scripts/kconfig/Makefile 的 20 行定义：

```
8 ifdef KBUILD_KCONFIG
9 Kconfig := $(KBUILD_KCONFIG)
10 else
11 Kconfig := arch/$(SRCARCH)/Kconfig
12 endif
20 menuconfig: $(obj)/mconf
21       $< $(Kconfig)
```

从这个命令可以看出，最终会以 arch/x86/Kconfig 为参数运行 scripts/kconfig/mconf 这个脚本，出现配置界面。注意，mconf 是个 C 程序，该程序并不属于内核，而是一个用户态程序。Linux 源代码中这一类程序还有很多。如在 scripts/kconfig/目录下就有 mconf,gconf,qconf 等等。它们用来执行内核的配置工作。

scripts/kconfig/mconf 这个程序采用了 ncurses 类库。这是一个在文本界面下进行画图操作类库。由于要适应不同平台，源代码中的 mconf 不是预编译

好的 elf 可执行文件，而是在使用时才去编译生成。这使用户在运行 make menuconfig 时要依赖 ncurses 的开发包。所以，如果你机器上没有 ncurses 的 rpm 包，建议去下载并安装。

arch/x86/Kconfig，准确地说是各个 Kconfig 文件记录了各个内核配置的选项。我们在 make menuconfig 或者 make xconfig 时显示的菜单项和帮助信息，都是从这个文件中读出来的。我们来看看这个文件的内容：

```
……
config NEED_PER_CPU_EMBED_FIRST_CHUNK
      def_bool y
config NEED_PER_CPU_PAGE_FIRST_CHUNK
      def_bool y
config HAVE_CPUMASK_OF_CPU_MAP
      def_bool I386_SMP
……
```

当年配置完 menuconfig 以后，就会在主目录下生成一个.config 文件，我们再来看看这个文件的部分内容：

```
……
CONFIG_NEED_PER_CPU_EMBED_FIRST_CHUNK=y
CONFIG_NEED_PER_CPU_PAGE_FIRST_CHUNK=y
CONFIG_HAVE_CPUMASK_OF_CPU_MAP=y
……
```

不错，生成一些 CONFIG_XXX 的东西。这些东西作为全局宏变量，由我们的 Makefile 引用。至于如何引用，且听下回分解。

2.2.2 寻找第一个目标

在 make menucofig 之后，即编译内核的第二步，是执行 make 命令（2.6 之前的内核是执行 make bzImage）。我们知道，make 命令没有任何参数的时候，默认去执行当前目录下的 Makefile 文件。我们编译内核的时候是在刚刚从 kernel.org 下载来的 linux-2.6.34.1 源码包，所以执行的就是 linux-2.6.34.1/Makefile 文件，而这个文件的突破口，是找到它的第一个目标。下面，我们就来详细分析这个文件。

首先是几个关于版本信息的宏。

```
1 VERSION = 2
2 PATCHLEVEL = 6
3 SUBLEVEL = 34
4 EXTRAVERSION = .1
5 NAME = Sheep on Meth
```

随后 17 到 23 行设置环境参数，主要是把 LC_COLLATE 和 LC_NUMERIC 两个环境变量的值设置成 C，表示 C 语言。

继续走，44 到 49 行：

```
44ifeq ("$(origin V)", "command line")
45    KBUILD_VERBOSE = $(V)
46endif
47ifndef KBUILD_VERBOSE
48    KBUILD_VERBOSE = 0
49endif
```

我们本着“看内核学知识”的原则，介绍介绍 origin 函数。函数 origin 并不操作变量的值，只是告诉你你的这个变量是哪里来的，其语法是： $(origin <variable>;)。注意，<variable> 是变量的名字，不应该是引用。所以你最好不要在<variable>中使用“$”字符。origin 函数会以其返回值来告诉你这个变量的“出生情况”，origin 函数的返回值有：“undefined”从来没有定义过、“default”是一个默认的定义、“environment”是一个环境变量、“file”这个变量被定义在 Makefile 中、“command line”这个变量是来自命令行的、“override”是被 override 指示符重新定义的、“automatic”是一个命令运行中的自动化变量，应用变量的语法是：$(变量名)。

这些信息对于我们编写 Makefile 是非常有用的，例如，在这里我们 make 命令后跟了一个参数“V”，而我们的环境中也有一个环境变量“V”，此时，我们想判断一下，如果变量来源于参数，那么我们就把之重定义了，如果来源于非命令行或环境变量，那么我们就不重新定义它。

KBUILD_VERBOSE 的值根据在命令行中是否定义了变量 V，当没有定义时，默认为 V＝0，输出为 short version；可以用 make V=1 来输出全部的命令。

好了，我们跳过若干行变量定义，看到 105 行，我们的故事开始了：

```
105 # That's our default target when none is given on the command line
106 PHONY := _all
107 _all:
```

看到 105 行的注释，如果 make 后什么参数都没有，就到这里。在继续往下走之前，我们先来看个前提条件，就是 97 行对 KBUILD_SRC 有个判断，即 KBUILD_SRC 为空才会来到这里。我们看到注释，KBUILD_SRC 当然是空的，所以我们敲击 make 之后，首先来到的是第 107 行。

当然，你可以做个试验，比如在 107 行后面添加一行，然后写一个@echo “hello, world！”命令，看看 make 以后会不会打印出来。试验的结果很令人诧异，什么也没有，看来我们的 GNU make 基本功还不扎实，还需要继续刻苦学习。看到 142 行：

```
142 ifeq ($(KBUILD_EXTMOD),)
143 _all: all
144 else
145 _all: modules
146 endif
```

这里怎么又出现了一个_all 目标呢？这里引出一个非常重要的 GNU make 知识，那就是如果出现了相同的 Target，那么后面的 Target 将会把前面的 Target 覆盖，并且执行后一个 Target 的命令和依赖。所以，当你 make 的时候，虽然会首先来到 107 行的_all 目标，但紧接着，make 会发现后面 143 或 145 还有一个_all，那么后面这一个_all 就会覆盖它，所以我们看到 142 行。说起这个 KBUILD_EXTMOD 变量，就要回到我们的 71 行：

```
71 ifdef SUBDIRS
72      KBUILD_EXTMOD ?= $(SUBDIRS)
73 endif
75 ifeq ("$(origin M)", "command line")
76      KBUILD_EXTMOD := $(M)
77 endif
```

前面讲了操作符“ := ”与操作符“ += ”的功能相同，只是后面的操作符“ := ”用来定义变量（KBUILD_EXTMOD）的变量 M 只能是前面定义好的，也就是命令行中有 M=dir，那么这个 M 就等于 dir 对应的值（SUBDIRS）。如果操作符“ ?= ”前面的变量 KBUILD_EXTMOD 没有定义过，那么就将 SUBDIRS 赋给 KBUILD_EXTMOD；如果定义过，则语句 KBUILD_EXTMOD ?= $(SUBDIRS)什么也不做。

KBUILD_EXTMOD 叫做 Kbuild 扩展模式，如果我们 make M=dir，那么会将变量 KBUILD_EXTMOD 的值设置为 dir，随后在 142 行，会跳到 145 行去执行依赖于 modules 的代码，就相当于执行 make modules。这个 make 方法也是用来提高我们的驱动开发效率的，如果你只想编译某个具体的驱动，只要指定对应的子目录，就可以只编译这个驱动而不去碰其他的内核代码了。

继续往下走之前，还要强调一个知识点。all 这个目标被定义在顶层 Makefile 的 527 行：

527 all: vmlinux

大家看到在我们的 535 行，include 了一个$(srctree)/arch/$(SRCARCH)/Makefile 文件，也就是前面在 KBuild 体系中提到的那个具体 CPU 架构的 Makefile。针对我的系统，这个文件自然而然就是 linux-2.6.34.1/arch/x86/Makefile，我们打开这个文件，可以看到它的 150 行也有一个 all 目标。也就是说，x86 体系的 Makefile 把顶层 Makefile 的 all 目标给覆盖了：

```
150 all: bzImage
151
152 # KBUILD_IMAGE specify target image being built
153 KBUILD_IMAGE := $(boot)/bzImage
154
155 bzImage: vmlinux
```

有意思的是，不管是顶层的 Makefile，还是具体体系的 Makefile，他们的 all 目标都依赖于 vmlinux（当然，arch/x86/Makefile 是间接依赖）。所以我们还是来关注位于顶层 Makefile 的 vmlinux 目标。vmlinux 目标是顶层 Makefile 中最重要的 Target，其目的主要是用来初始化编译期间 KBuild 使用到的内核目标。查找 vmlinux 还真费劲，在后面 849 行：

```
vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o $(kallsyms.o) FORCE
```

注意，这个 vmlinux 后面依赖了 FORCE，说明是伪目标，跟最后生成的 vmlinux 风马牛不相及。vmlinux 又依赖了五个东西，个个都不是省油的灯，非常重要，我们一个一个来看，其中前 3 个目标定义来自来自 699、700、702 行（这也是为什么 vmlinux 要定义在 800 多行的原因）：

```
699 vmlinux-init := $(head-y) $(init-y)
700 vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
701 vmlinux-all    := $(vmlinux-init) $(vmlinux-main)
702 vmlinux-lds    := arch/$(SRCARCH)/kernel/vmlinux.lds
703 export KBUILD_VMLINUX_OBJS := $(vmlinux-all)
```

vmlinux.o 目标来自 869 行：

```
869 vmlinux.o: $(modpost-init) $(vmlinux-main) FORCE
870         $(call if_changed_rule,vmlinux-modpost)
```

kallsyms.o 变量来自 776 行：

```
776 kallsyms.o := .tmp_kallsyms$(last_kallsyms).o
```

vmlinx 是我们的重中之重，要分析它，必须把它的依赖详细展开来看。不过这个工作先放一放，我们这里的目的是寻找第一个目标，所以这部分的展开先略过。vmlinux 的第一个依赖是$(vmlinux-lds)，其值被赋成了 arch/x86/kernel/vmlinux.lds，这个文件是很重要的，后面会详细讲解。而$(vmlinux-lds)跟其他 vmlinux 的依赖一起，又依赖于$(vmlinux-dirs) ，我们看到 874 行：

```
874 $(sort $(vmlinux-init) $(vmlinux-main)) $(vmlinux-lds): $(vmlinux-dirs) ;
```

这条语句是顶层 Makefile 中最为关键的一行。vmlinux-dirs 变量是什么？就是形成 vmlinux

的所有待编译目录的集合，定义于 655 行：

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

filter 函数很简单，过滤掉$(init-y) $(init-m) $(core-y) $(core-m) $(drivers-y) $(drivers-m) $(net-y) $(net-m) $(libs-y) $(libs-m)这么一长串字符串中的%/，那么 vmlinux-dirs 最后的值就是类似 init usr kernel mm fs ipc security crypto block drivers sound firmware net lib……这么一长串的字符串。当然，这串字符串具体是什么，以及后面执行啥命令，这是整个 KBuild 体系的工作核心，后面还会详细分析，现在只是说明一下这里有这么一个过程，一个思路。

这里顺便提一下，659 行 vmlinux-alldirs 变量的赋值过程类似，只不过最顶层是通过 sort 函数对最后的字符串进行一个排序，还要加上一些 init-n 或 init-形式的目录。vmlinux-alldirs 变量基本上用不到。

看到 883 行，$(vmlinux-dirs)又有两个依赖：

```
883 $(vmlinux-dirs): prepare scripts
884         $(Q)$(MAKE) $(build)=$@
885 ifdef CONFIG_MODULES
886         $(Q)$(MAKE) $(modbuiltin)=$@
887 endif
```

他们分别是 prepare 和 scripts，先来看第一个依赖目标，来自 990 行：

```
990 prepare: prepare0
```

不错，它又依赖于 prepare0，看到 985 行：

```
985 prepare0: archprepare FORCE
```

又依赖于 archprepare，来自 983 行：

```
983 archprepare: prepare1 scripts_basic
```

它又依赖于 prepare1 和 scripts_basic，看来寻找第一个目标的任务还得继续啊，来看 prepare1，来自 979 行：

```
979 prepare1: prepare2 include/linux/version.h include/generated/utsrelease.h \
980                         include/config/auto.conf
```

继续走，又依赖 prepare2，来自 977：

```
977 prepare2: prepare3 outputmakefile
```

继续走，又依赖 prepare3，来自 966：

```
prepare3: include/config/kernel.release
```

原本到 966 行，我以为 prepare3 的依赖就是一个文件，只是不存在的文件，都准备把 prepare3 作为第一个目标了，结果眼睛一瞥，949 行：

949 include/config/kernel.release: include/config/auto.conf FORCE

还好我留了一个心眼，果然，include/config/auto.conf 也是一个 Target：
507 include/config/auto.conf:

经过艰苦的努力，终于把顶层 Makefile 的第一个目标找到了。当然本着学习的态度，我们还要稍微深入地讨论一下，那就是整个顶层 Makefile 还有三个重要的变量：mixed-targets、config-targets 和 dot-config，定义在 416 行：

```
416 config-targets := 0
417 mixed-targets   := 0
418 dot-config = 1
```

随后看到 426 行：

```
426 ifeq ($(KBUILD_EXTMOD),)
427            ifneq ($(filter config %config,$(MAKECMDGOALS)),)
428                    config-targets := 1
429                    ifneq ($(filter-out config %config,$(MAKECMDGOALS)),)
430                            mixed-targets := 1
431                    endif
432            endif
433 endif
```

$(MAKECMDGOALS)是环境变量，就是 make 后面跟的参数字符串。注意，这里一个很重要的知识点，make 在执行时设置一个特殊变量“ MAKECMDGOALS ”，记录了命令行参数指定的终极目标列表，没有通过参数指定终极目标时此变量为空。注意：此变量仅用在特殊的场合（比如判断），在 Makeifle 中不要对它进行重新定义！但在编译内核这种特殊环境中，如果在 make menuconfig 之前，make 后没有任何参数， $(MAKECMDGOALS)是有值的：silentoldconfig。而在配置完成后，$(MAKECMDGOALS)是为空。至于这是为什么，我也不知道，我知道结论是通过做实验来得到的：

make menuconfig 前：

```
mixed-targets = 0
config-targets = 1
dot-config = 1
$(MAKECMDGOALS) = silentoldconfig
```

make menuconfig 后：

```
mixed-targets = 0
config-targets = 0
dot-config = 1
$(MAKECMDGOALS) =
```

如果 make 后面跟的参数是 config 或形式为%config，那么 config-targets 就为 1，并且还有其他的参数，那么 mixed-targets 也为 1。但更为重要的是 dot-config 变量，看到 420 行：

```
420 ifneq ($(filter $(no-dot-config-targets), $(MAKECMDGOALS)),)
```

```
421         ifeq ($(filter-out $(no-dot-config-targets), $(MAKECMDGOALS)),)
422                 dot-config := 0
423         endif
424 endif
```

所以这里，如果 make 不跟任何参数，那么 config-targets 和 dot-config 都是等于 1 的。在 411 行说明了 no-dot-config-targets 是由这些字符串组成：clean mrproper distclean cscope TAGS tags help %docs check% include/linux/version.h headers_% kernelrelease kernelversion，那么 420 行说明如果 make 后面的参数是以上的形式，那么 421 行再把这些过滤掉，如果为空，则 dot-config 就为 0。大家注意，no-dot-config-targets 所涉及的参数都跟我们的.config 无关，而 dot-config 变量的值就是这个意思。

综上所述，如果在 make menuconfig 之前 make，那么$(MAKECMDGOALS)是 silentoldconfig，因此 config-targets 的值是 1，那么顶层的 Makefile 的从 463 到 1433 行的代码将会被略过。那么第一个目标是 385 行的 scripts_basic。具体的细节请去看上一节编译配置的内容。

如果在 make menuconfig 之后 make 那么$(MAKECMDGOALS)的值是空的，那么编译配置时会生成文件 include/config/auto.conf，大概 46k，其内容如下：

```
CONFIG_USB_SISUSBVGA=m
CONFIG_PCMCIA_FMVJ18X=m
CONFIG_BLK_CPQ_DA=m
CONFIG_BLK_DEV_FD=m
CONFIG_ACPI_AC=m
CONFIG_ACPI_SYSFS_POWER=y
CONFIG_SECURITY_NETWORK=y
CONFIG_OSF_PARTITION=y
CONFIG_USB_LEGOTOWER=m
……
```

所以 949 行其实有个 FORCE，因此虽然 include/config/auto.conf 生成了，但还是要执行 include/config/kernel.release 目标的命令，那么这个目标就是我们寻址的第一个目标。

2.2.3 prepare 和 scripts 目标

上一节我们找到了整个 KBuild 体系的第一个目标，根据是否进行了编译配置而分别是 385 行的 scripts_basic 和 949 行的 include/config/kernel.release。更重要的是，通过第一个目标的寻找，我们把 KBuild 体系的整个顶层 Makefile 文件脉络梳理了一遍，这对我们研究 vmlinux 的形成及其重要。下面我们就从第一个目标开始，对编译配置后 make 的整个过程来过一遍。

949 行，第一个目标：

```
949 include/config/kernel.release: include/config/auto.conf FORCE
950         $(Q)rm -f $@
951         $(Q)echo $(kernelrelease) > $@
```

执行的命令很简单，删除旧的 kernel.release，新建一个 kernel.release 并把 kernelrelease 变量的值转移标准输出，写入该文件：

```
358 KERNELVERSION = $(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)
948 kernelrelease = $(KERNELVERSION)$(localver-full)
```

第一个目标完成后，来到 966 行的 prepare3 目标：

```
966 prepare3: include/config/kernel.release
967 ifneq ($(KBUILD_SRC),)
968         @$(kecho) '    Using $(srctree) as source for kernel'
969         $(Q)if [ -f $(srctree)/.config -o -d $(srctree)/include/config ]; then \
970                 echo "    $(srctree) is not clean, please run 'make mrproper'";\
971                 echo "    in the '$(srctree)' directory.";\
972                 /bin/false; \
973         fi;
974 endif
```

没有问题，不会去执行的，因为$(KBUILD_SRC)是空的。所以来到 977 行：

```
977 prepare2: prepare3 outputmakefile
```

我们看到 prepare2 什么命令也不执行，但它除了 prepare3 还有个依赖 outputmakefile：

```
396 outputmakefile:
397 ifneq ($(KBUILD_SRC),)
398         $(Q)ln -fsn $(srctree) source
399         $(Q)$(CONFIG_SHELL) $(srctree)/scripts/mkmakefile \
400                 $(srctree) $(objtree) $(VERSION) $(PATCHLEVEL)
401 endif
```

同样也不会执行这个命令。那么就来到了 979 行：

```
979 prepare1: prepare2 include/linux/version.h include/generated/utsrelease.h \
980                                 include/config/auto.conf
981         $(cmd_crmodverdir)
```

看到它的依赖：

```
1013 include/linux/version.h: $(srctree)/Makefile FORCE
1014        $(call filechk,version.h)
```

有个 FORCE，所以尽管 include/linux/version.h 是存在的，但也要执行 1014 的命令。我们只是学习学习新知识，1014 行代码，函数 call。call 关键字是唯一一个可以用来创建新的参数化的函数。你可以写一个非常复杂的表达式，这个表达式中，你可以定义许多参数，然后你可以用 call 函数来向这个表达式传递参数。其语法是：

```
$(call <expression>,<parm1>,<parm2>,<parm3>...)
```

当 make 执行这个函数时，<expression>参数中的变量，如$(1)，$(2)，$(3)等，会被参数

<parm1>，<parm2>，<parm3>依次取代。而<expression>的返回值就是 call 函数的返回值。

例如：

```
reverse = $(1) $(2)
foo = $(call reverse,a,b)
```

那么，foo 的值就是“ a b ”。当然，参数的次序是可以自定义的，不一定是顺序的，如：

```
reverse = $(2) $(1)
foo = $(call reverse,a,b)
```

此时的 foo 的值就是“ b a ”。

所以我们猜到，1014 行的代码是让 filechk 变量等于变成另一个字符串。那么，filechk 变量在哪儿定义的呢？顶层 Makefile 找不到，那么在/scripts/Kbuild.include 中找，49 行：

```
define filechk
	$(Q)set -e;                             \
	$(kecho) '   CHK        $@';          \
	mkdir -p $(dir $@);                 \
	$(filechk_$(1)) < $< > $@.tmp;            \
	if [ -r $@ ] && cmp -s $@ $@.tmp; then      \
		rm -f $@.tmp;                 \
	else                                \
		$(kecho) '   UPD        $@';  \
		mv -f $@.tmp $@;          \
	fi
endef
```

调用 call 以后，filechk 变量就会变成：

```
	$(Q)set -e;                             \
	$(kecho) '   CHK        $@';          \
	mkdir -p $(dir $@);                 \
	$(filechk_ version.h) < $< > $@.tmp;          \
	if [ -r $@ ] && cmp -s $@ $@.tmp; then      \
		rm -f $@.tmp;                 \
	else                                \
		$(kecho) '   UPD        $@';  \
		mv -f $@.tmp $@;          \
	fi
```

其中的$@是 include/linux/version.h，所以我们看到 make 之后屏幕输出的第一行肯定是：

```
  CHK         include/linux/version.h
```

第三个依赖，include/generated/utsrelease.h，来自 1016 行：

```
1016 include/generated/utsrelease.h: include/config/kernel.release FORCE
1017       $(call filechk,utsrelease.h)
```

跟第二个依赖一样，所以不去管它了。

最后一个依赖 include/config/auto.conf，make menucofig 生成的，所以就执行 prepare1 的命令了：

```
cmd_crmodverdir = $(Q)mkdir -p $(MODVERDIR) \
                  $(if $(KBUILD_MODULES),; rm -f $(MODVERDIR)/*)
```

这个命令我没看懂是什么，只知道在 373 行有个$(MODVERDIR)变量的定义：

```
export MODVERDIR := \
$(if $(KBUILD_EXTMOD),$(firstword $(KBUILD_EXTMOD))/).tmp_versions
```

不管怎样，这个文件夹终究会被删除的，不去详细分析了。那么 prepare1 也结束了，这时候才来到了 archprepare 的第二个依赖，scripts_basic。有关 scripts_basic 我们在编译配置里已经详细谈过了，其结果是在 scripts/basic/生成三个 c 程序：fixdep、docproc、hash。具体是怎么得到他们的请参照前面讲解的 KBuild 体系去查看 scripts/basic/Makefile 文件。至于他们是干什么的，以后用到了再研究。

scripts_basic 目标结束后，archprepare 因为没有附带任何命令，也结束了。所以来到 985 行的 prepare0：

```
prepare0: archprepare FORCE
	$(Q)$(MAKE) $(build)=.
	$(Q)$(MAKE) $(build)=. missing-syscalls
```

翻译一下：

```
@make -f scripts/Makefile.build obj=.
@make -f scripts/Makefile.build obj=. missing-syscalls
```

这里是将顶层目录，也就是/usr/src/kernels/linux-2.6.34.1 赋值给 obj。当然，顶层 Makefile 本身也是个 KBuild 的参数，所以会在屏幕上打印以下一些信息：

```
  CC          kernel/bounds.s
  GEN         include/generated/bounds.h
  CC          arch/x86/kernel/asm-offsets.s
  GEN         include/generated/asm-offsets.h
  CALL        scripts/checksyscalls.sh
```

至于为啥，我也没弄懂，还要请教广大网友帮我研究研究。prepare0 完毕以后，prepare 也就完成了，$(vmlinux-dirs)的第一个依赖也就搞完了。一路走来还是很艰辛的，这就是内核！来看第二个依赖，scripts，在 473 行：

```
473 scripts: scripts_basic include/config/auto.conf include/config/tristate.conf
474         $(Q)$(MAKE) $(build)=$(@)
```

scripts_basic 目标已经执行过了，include/config/tristate.conf 文件也存在，所以直接执行 474 行的命令：

```
@make -f scripts/Makefile.build obj= scripts
```

我们就去看 scripts 目录下的 Makefile 文件。具体内容我就不列出来了，主要是理清思路：先是 subdir-y，有 subdir-$(CONFIG_MODVERSIONS) += genksyms 等内容，那么就到 genksyms 等等这些子目录去执行他们的 Makefile，执行完后，再根据 hostprogs-y 的内容生成主机程序。

hostprogs-$(CONFIG_XXX)是去根据 CONFIG_XXX 标志主机程序，再去形成这些主机程序，也就是说直接去生成 c 程序而不是形成.o 以制作 built-in.o。具体的原理请去查阅前面有关 KBuild 体系主机程序的内容。

scripts 目标执行完后，整个前期准备工作就算完成了。后面就是更加艰难的形成 vmlinux 工作。

2.2.4 递归编译各对象

当 prepare 和 scripts 目标执行完后，就会去执行$(vmlinux-dirs)目标的命令：

```
883 $(vmlinux-dirs): prepare scripts
884         $(Q)$(MAKE) $(build)=$@
885 ifdef CONFIG_MODULES
886         $(Q)$(MAKE) $(modbuiltin)=$@
887 endif
```

别看这小小的 5 行代码，两条命令，整个 Makefile 最重要最核心的部分就是这里了。不过呢，只要熟悉了前面提到的 KBuild 体系，弄懂这里是易如反掌的。其实这里最难搞清楚的，就是$(vmlinux-dirs)变量的值，也就是形成 vmlinux 所需要哪些目录。下面，我们就来重点考察$(vmlinux-dirs)这个变量。它第一次被定义在 655 行：

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

init-y、core-y 、drivers-y、net-y、libs-y 这五个变量来自 477 到 481 行：

```
477 init-y        := init/
478 drivers-y     := drivers/ sound/ firmware/
479 net-y         := net/
480 libs-y              := lib/
481 core-y              := usr/
```

init-m、core-m 、drivers-m 、net-m 、libs-m 没有定义，为空。那么针对前面那五个，一个一个来看，首先 init-y，除了 477 行定义，还有 664 行：

```
664 init-y        := $(patsubst %/, %/built-in.o, $(init-y))
```

注意，664 行开始的一连串定义是被包含在了 ifeq ($(KBUILD_EXTMOD),)条件中，我们的

make 满足，所以 664 行定义也满足，那么 init-y 的值是什么呢？学知识了，讲讲 patsubst 函数。

模式字符串替换函数——patsubst：
$(patsubst <pattern>,<replacement>,<text>)
它的功能是查找<text>中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式<pattern>，如果匹配的话，则以<replacement>替换。这里，<pattern>可以包括通配符“%”，表示任意长度的字串。如果<replacement>中也包含“%”，那么，<replacement>中的这个“%”将是<pattern>中的那个“%”所代表的字串。（可以用“\”来转义，以“\%”来表示真实含义的“%”字符）。

所以这里，init-y 的值就是 init/built-in.o，这个文件是不存在，Kbuild 将编译所有的$(obj-y)文件，然后，调用"$(LD) -r"将它们合并到当前目录的 build-in.o 文件中。稍后，该 build-in.o 会被顶层 Makefile 链接进 vmlinux 中。

再来看 core-y，481 行第一次定义为 usr/，随后 653 行：
653 core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
也就是说最核心的目录名字符串就是 core-y。完了吗？别忘了我们在 535 行 include 了一个$(srctree)/arch/$(SRCARCH)/Makefile，找到 arch/x86/Makefile，126 行：
126 core-y += arch/x86/
回到顶层 Makefile 然后 665 行，
665 core-y := $(patsubst %/, %/built-in.o, $(core-y))

所以 core-y 最后的值是：usr/built-in.o arch/x86/built-in.o kernel/built-in.o mm/built-in.o fs/built-in.o ipc/built-in.o security/built-in.o crypto/built-in.o block/built-in.o

继续走，drivers-y，478 行第一次定义之后，随后在 666 行：
drivers-y := $(patsubst %/, %/built-in.o, $(drivers-y))

所以最后 drivers-y 的值是 drivers/built-in.o sound/built-in.o firmware/built-in.o。好了，我们再来考察 net-y 变量，479 行定义后，667 行：
667 net-y := $(patsubst %/, %/built-in.o, $(net-y))

net-y 的最终值是 net/built-in.o。libs-y 的分析方法跟他们类似，最后是 lib/lib.a lib/built-in.o。
综上所述，vmlinux-dirs 的值再经过 filter 和 patsubst 两个函数一折腾，就成了这么一串字符串：
init usr arch/x86 kernel mm fs ipc security crypto block drivers sound firmware net lib lib
好了，vmlinux-dirs 变量我们理顺了，883 到 887 行的代码就很好理解了，参照前面讲的 Kbuild 体系的知识，对上面提到的 14 个目录进行编译与链接，生成对应的 built-in.o 文件。我们随便找一个目录，比如说第一个 init，翻译成命令就是：
@make -f scripts/Makefile.build obj=init

首先看它的 Makefile，先编译 obj-y 的那些对象，我们看到第 5 到 11 行：
obj-y := main.o version.o mounts.o

说明该 init 目录下肯定会有三个目标被编译的，他们是 main.o version.o mounts.o；还有一些目标是可能会被编译的，如 noinitramfs.o，取决于编译配置了 CONFIG_BLK_DEV_INITRD 选项；那么，肯定会被编译的东西中，version.o 和 mounts.o 两个东西又很讨厌，前一个需要依赖一个头文件，后一个又需要另外一些对象链接而成。所以经过一系列编译和链接后会在 init 目录中生成 built-in.o 文件。这就是我们在 KBuild 体系中讲过的，编译本目录中所有的 obj-y 所包含文件，然后，调用"$(LD) -r"将它们合并到一个 build-in.o 文件中。

我们再看一个难一点的，看到 fs 文件夹。还是首先看它的 Makefile，8 到 14 行，先编译 obj-y 的那些对象，所以我们看到 open.o read_write.o file_table.o 等这些对象是必被编译的。16 行以后是根据配置编译相应的对象。注意，如果对象是一个目录（以“ / ”结束的），比如：

```
72 obj-$(CONFIG_EXT4_FS)            += ext4/
```

其实就相当于：

```
@make -f scripts/Makefile.build obj= ext4
```

那么，我们就去 ext4 目录中去找他的 Makefile，这个文件很简单，我把全部内容先列出来：

```
obj-$(CONFIG_EXT4_FS) += ext4.o
ext4-y      := balloc.o bitmap.o dir.o file.o fsync.o ialloc.o inode.o \
               ioctl.o namei.o super.o symlink.o hash.o resize.o extents.o \
               ext4_jbd2.o migrate.o mballoc.o block_validity.o move_extent.o
ext4-$(CONFIG_EXT4_FS_XATTR)          += xattr.o xattr_user.o xattr_trusted.o
ext4-$(CONFIG_EXT4_FS_POSIX_ACL)  += acl.o
ext4-$(CONFIG_EXT4_FS_SECURITY)           += xattr_security.o
```

我们看到，根据 CONFIG_EXT4_FS 的值是 y 还是 m，将 ext4.o 对象编译进内核或模块。而 ext4.o 对象又是如何编译得到的呢，后面 ext4-y 或 ext4-m 中的对象就告诉了你答案。这里顺便补充一个知识点，前面我们讲到了编译配置，在这儿更进一步，scripts/kconfig/mconf 这个程序读取的是每个目录中的 Kconfig 文件。这个文件有若干的 item，每个 item 格式如下：

```
config EXT4_FS_CHEN
        bool "Just For Test"
        depends on EXT4_FS_XATTR
        select FS_CHEN
        help
          hello, world!.
```

其中 config、bool、depends、select 和 help 是关键字，后四个关键字前必须用[tab]隔开。config 后表示配置选项名称，比如我们这里，它最后的名称就是 CONFIG_EXT4_FS_CHEN；bool 后面是我们 make menuconfig 后在菜单中看到的 item 名称；depends 表示这个 item 需要依赖的配置选项名；select 表示，如果选择了，那么对应的配置选项 CONFIG_FS_CHEN 也将被设置；最后 help，就是留给内核开发人员的对应配置选项注释信息。

还有一个重要的知识点，当在一个目录中编译完成后，会生成一些以.cmd 结尾的隐藏文件，如.built-in.o.cmd。这个表示生成对象的具体命令。比如，init/.built-in.o.cmd 的内容如下：

```
cmd_init/built-in.o :=    ld -m elf_i386      -r -o init/built-in.o init/main.o init/version.o
init/mounts.o init/initramfs.o init/calibrate.o
```

就是指 built-in.o 这个对象，是通过 ld 命令链接 nit/main.o、init/version.o、init/mounts.o、init/initramfs.o 和 init/calibrate.o 而生成的。我也尝试着看了看.calibrate.o.cmd 文件的内容，很长，说明生成.calibrate.o 的命令很长，也很复杂，对编译感兴趣的朋友可以去分析一下，很有挑战性的哟。

在递归向下访问所有在下列变量中列出的目录： init-* core* drivers-* net-* libs-*，并编译生成目标文件 built-in.o 后，下一步就是在源代码树顶层目录中生成 vmlinux。前面“ 寻找第一个目标 ”讲了，在 849 行：

vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o $(kallsyms.o) FORCE

是 vmlinux 目标的各个依赖，再看到 874 行：

$(sort $(vmlinux-init) $(vmlinux-main)) $(vmlinux-lds): $(vmlinux-dirs) ;

2.2.5 链接 vmlinux

当我们前面$(vmlinux-dirs)目标的工作做完后，也就是形成了各目录中的 built-in.o 文件，那么就会回到$(sort $(vmlinux-init) $(vmlinux-main)) $(vmlinux-lds)所对应的目标中，这个目标是一行空命令，看到 699 行：

```
699 vmlinux-init := $(head-y) $(init-y)
700 vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
701 vmlinux-all    := $(vmlinux-init) $(vmlinux-main)
702 vmlinux-lds    := arch/$(SRCARCH)/kernel/vmlinux.lds
```

我们先来看 vmlinux-lds，其值被赋成了 arch/x86/kernel/vmlinux.lds，这个文件前面提过了，它是用来链接 vmlinux 的，后面会详细讲解。vmlinux-init 的值是由 head-y 和 init-y 变量组成。init-y 前面讲过了，head-y 呢，在我们的 arch/x86/Makefile 的 118 行，具体这里就不再赘述了，我们关注的还是思路。vmlinux-main 的值，我们看到了 core-y、libs-y、drivers-y、net-y，都是些熟面孔，前面讲得很多了，略过。

空命令执行完后，就会来到 vmlinux.o 目标，来自 869 行：

```
869 vmlinux.o: $(modpost-init) $(vmlinux-main) FORCE
870         $(call if_changed_rule,vmlinux-modpost)
```

这个 if_changed_rule 是我们第一次见到，在 scripts/Kbuild.include 的 220 行定义：

```
if_changed_rule = $(if $(strip $(any-prereq) $(arg-check) ),                    \
      @set -e;                                                                  \
      $(rule_$(1)))
```

这个 if_changed_rule 的作用就是既检查依赖的更新也检查命令行参数的更新。所以，call 函数之后，if_changed_rule 的值变为 rule_ vmlinux-modpost，它定义在顶层 Makefile 的 841 行：

```
841 define rule_vmlinux-modpost
842         :
843         +$(call cmd,vmlinux-modpost)
844         $(Q)$(MAKE) -f $(srctree)/scripts/Makefile.modpost $@
```

845 $(Q)echo 'cmd_$@ := $(cmd_vmlinux-modpost)' > $(dot-target).cmd
846 endef

cmd 也来自 scripts/Kbuild.include：
cmd = @$(echo-cmd) $(cmd_$(1))
用来打印命令，后面还有很长一串依赖，我们就不去管它了。844 行，翻译过来就是：
@make –f scripts/Makefile.modpost vmlinux.o

Makefile.modpost 是什么呢？这里补充一个 KBuild 的知识。在 scripts 目录下有如下重要的编译规则文件：
kbuild.include ——共用的定义。会被所有独立的 Makefile 包括,如 Makefile.build 等
Makefile.build ——提供编译 built-in.o, lib.a 等规则的 Makefile
Makefile.lib ——负责归类分析 obj-y obj-m 和其中的目录 subdir-ym
Makefile.host ——主机程序 hostprog 编译规则
Makefile.clean ——clean 规则
Makefile.headerinst ——头文件 install 规则
Makefile.modinst ——模块 install 规则
Makefile.modpost ——模块编译的第二阶段,由.o 和.mod 生成<module>.ko

所以，Makefile.modpost 是代表一种编译规则，负责将相应的.o 对象文件编译成.ko 文件。由于有个 FORCE，所以在这一步，即使之前刚刚生成了 vmlinux.o 文件（我猜想，KBuild 可能规定，最顶层的目录不会生成 built-in.o，而是 vmlinux.o），也会执行这个命令。vmlinux.o 是怎么生成的呢？看看.vmlinux.o.cmd：
cmd_vmlinux.o := ld -m elf_i386 -r -o vmlinux.o arch/x86/kernel/head_64.o arch/x86/kernel/head64.o arch/x86/kernel/head.o arch/x86/kernel/init_task.o init/built-in.o --start-group usr/built-in.o arch/x86/built-in.o kernel/built-in.o mm/built-in.o fs/built-in.o ipc/built-in.o security/built-in.o crypto/built-in.o block/built-in.o lib/lib.a arch/x86/lib/lib.a lib/built-in.o arch/x86/lib/built-in.o drivers/built-in.o sound/built-in.o firmware/built-in.o arch/x86/pci/built-in.o arch/x86/power/built-in.o arch/x86/video/built-in.o net/built-in.o --end-group

几乎涵盖了最重要，最核心的内核部分，所以 make 之后，我机器上最后生成的 vmlinux.o 将近 200MB。继续走，再来看到 kallsyms.o 目标，是在 776 行定义的：
770 ifdef CONFIG_KALLSYMS_EXTRA_PASS
771 last_kallsyms := 3
772 else
773 last_kallsyms := 2
774 endif
776 kallsyms.o := .tmp_kallsyms$(last_kallsyms).o
我们看到前面 make menuconfig 在主目录中生成的.config 的 126 行有：
126 CONFIG_KALLSYMS=y
127 CONFIG_KALLSYMS_ALL=y
128 CONFIG_KALLSYMS_EXTRA_PASS=y

所以，这里 last_kallsyms 的值为 3，kallsyms.o 变量的值就是.tmp_kallsyms3.o，这是一个隐藏文件，也是一个 make 目标。那么，KALLSYMS 究竟是个什么东西呢？百度之。在 2.6 内核中，为了更好地调试内核，引入了 kallsyms 。kallsyms 抽取内核用到的所有函数地址(全局的，静态的)和非栈数据变量地址，生成一个数据小块(data blog)，作为只读数据链接进 kernel image 。相当于内核中存在了一份 System.map .内核可以根据符号名查出函数的地址。当系统发生 oops 时，（不懂 oops？好吧，内核崩溃总懂了吧）kallsyms 的 print_symbol 函数会显示相关信息，以此形式跟踪栈的状态。

原来这是一个内核调试选项，主要回去执行 752 到 833 行的那些依赖目标，我们就不去详细分析它了，只要知道它最后会生成.tmp_vmlinux1、.tmp_vmlinux2 和.tmp_vmlinux3 这三个文件就行了。

走到这里，vmlinux 的所有依赖都完成了，所有的目录也都生成了其对于的 built-in.o 文件。

```
849 vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o $(kallsyms.o) FORCE
850 ifdef CONFIG_HEADERS_CHECK
851         $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
852 endif
853 ifdef CONFIG_SAMPLES
854         $(Q)$(MAKE) $(build)=samples
855 endif
856 ifdef CONFIG_BUILD_DOCSRC
857         $(Q)$(MAKE) $(build)=Documentation
858 endif
859         $(call vmlinux-modpost)
860         $(call if_changed_rule,vmlinux__)
861         $(Q)rm -f .old_version
```

CONFIG_HEADERS_CHECK 在.config 里没有设置；CONFIG_SAMPLES 设置了，所以 854 行回去链接 samples 目录下的 built-in.o；CONFIG_BUILD_DOCSRC 在.config 里也没有设置，所以直接来到 859 行，vmlinux-modpost。找遍了，都没有定义这个东西，所以忽略。860 行，转变一下 if_changed_rule 就是“ rule_vmlinux__ ”，所以看到 734 行：

```
734 define rule_vmlinux__
735         :
736         $(if $(CONFIG_KALLSYMS),,+$(call cmd,vmlinux_version))
738         $(call cmd,vmlinux__)
739         $(Q)echo 'cmd_$@ := $(cmd_vmlinux__)' > $(@D)/.$(@F).cmd
741         $(Q)$(if $($(quiet)cmd_sysmap),                                 \
742           echo '    $($(quiet)cmd_sysmap)    System.map' &&)            \
743         $(cmd_sysmap) $@ System.map;                                        \
744         if [ $$? -ne 0 ]; then                                          \
745                 rm -f $@;                                               \
746                 /bin/false;                                             \
747         fi;
748         $(verify_kallsyms)
```

749 endef

还记得@D 和@F 吧，看来前面的预备知识是很有必要的，定义模式规则里面讲了，一个是目录部分，一个是文件部分。当然这里@D 是空的，@F 自然就是 vmlinux。所以上面这一大段 Makefile 的代码你看不懂没关系，只要知道 739 行的目的就是把制作 vmlinux 的最后一步打印出来，并重定向到顶层目录的.vmlinux.cmd 文件中。我们来看看这个文件的内容：
cmd_vmlinux := ld -m elf_i386 -o vmlinux -T arch/x86/kernel/vmlinux.lds arch/x86/kernel/head_64.o arch/x86/kernel/head64.o arch/x86/kernel/head.o arch/x86/kernel/init_task.o init/built-in.o --start-group usr/built-in.o arch/x86/built-in.o kernel/built-in.o mm/built-in.o fs/built-in.o ipc/built-in.o security/built-in.o crypto/built-in.o block/built-in.o lib/lib.a arch/x86/lib/lib.a lib/built-in.o arch/x86/lib/built-in.o drivers/built-in.o sound/built-in.o firmware/built-in.o arch/x86/pci/built-in.o arch/x86/power/built-in.o arch/x86/video/built-in.o net/built-in.o --end-group .tmp_kallsyms3.o

跟前面制作 vmlinux.o 的内容差不多，但是，很重要一点，多了一个
-T arch/x86/kernel/vmlinux.lds
和
.tmp_kallsyms3.o

.tmp_kallsyms3.o 就是把内核跟踪模块，不去管它了。重点来看第一个，我们看到 arch/x86/kernel/head.o 处于 vmlinux 的最前面，其主要作用就是初始化中断描述符表 IDT ，内存页目录表 GDT ，把系统从 64 位段式寻址的保护模式跳到 64 位页式寻址的保护模式 (Enable paging) .head.S 中首先定义的就是：
.section .text.head，"ax"，@progbits
ENTRY(startup_64)
starup_64 会在稍后的 vmlinux.ld 中看到。

ld 使用了链接脚本：arch/x86/kernel/vmlinux.lds。一般来说，普通程序是不需要指定 linker script 的，也不需要关心各个 section 的具体位置的。当程序执行时，kernel 中的 ELF Loader 会依据 ELF Program header 解析可执行文件的各个 SECTION，并把它们映射到虚拟地址空间。然而，内核启动时，必须首先确定各个 section 的具体位置，这就是 vmlinux.lds 的作用。

注意，vmlinux.lds 是个 C 程序，是在“ 递归编译各对象 ”时，生成的。至于是如何生成的，感兴趣的同志可以去研究研究 arch/x86/kernel/.vmlinux.lds.cmd 文件。生成之后，我们看到它前面大部分是注释，到 404 行开始才是实际内容：
/* 定义了输出格式和平台*/
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
/* 定义 phys_startup_32 为程序入口点*/
ENTRY(phys_startup_32)
/* jiffes 和 jiffies_64 地址相同,也就是说 jiffies_64 的低 32 位就是 jiffies */
jiffies_64 = jiffies;
/* 指定 ELF Program header 可以用 objdump -p 查看（有关可执行文件的结构的相关知识我

们会在相关的博客中补上） */
```
PHDRS {
  text PT_LOAD FLAGS(5); /* R_E */
  data PT_LOAD FLAGS(7); /* RWE */
  note PT_NOTE FLAGS(0); /* ___ */
}
/*  随后定义输入文件的 section 如何组织到输出文件的 section */
SECTIONS
{
/*  起始 VMA（此时线性地址就是虚拟地址-0xC0000000）地址为 0xC0100000, startup_32 也为 0xC0100000 */
            . = 0xC0000000 + ((0x1000000 + (0x400000 - 1)) & ~(0x400000 - 1));
            phys_startup_32 = startup_32 - 0xC0000000;
/* _text 地址为当前物理地址是  0x1000000，head.o 会放到此处*/
.text : AT(ADDR(.text) - 0xC0000000) {
    _text = .;
    /* bootstrapping code */
    *(.head.text)
    . = ALIGN((1 << 12));
    *(.text.page_aligned)
    . = ALIGN(8);
    _stext = .;
    .  =  ALIGN(8);  *(.text.hot)  *(.text)  *(.ref.text)  *(.devinit.text)  *(.devexit.text)  *(.cpuinit.text)
*(.cpuexit.text) *(.text.unlikely)
    . = ALIGN(8); __sched_text_start = .; *(.sched.text) __sched_text_end = .;
    . = ALIGN(8); __lock_text_start = .; *(.spinlock.text) __lock_text_end = .;
    . = ALIGN(8); __kprobes_text_start = .; *(.kprobes.text) __kprobes_text_end = .;

    *(.fixup)
    *(.gnu.warning)
    /* End of text section */
    _etext = .;
  } :text = 0x9090    /*合并 section 中的空隙用 0x9090 填充*/
```

用 nm 命令查看一下：

```
[root@localhost linux-2.6.34.1]# nm vmlinux|grep " _text"
c0100000 T _text
[root@ localhost linux-2.6.34.1]# nm vmlinux|grep startup_32
00100000 A phys_startup_32
c0100000 T startup_32
c01000c0 T startup_32_smp
```

要阅读这个.lds 文件必须对.lds 的语法有一定了解，本着学习知识的原则，还是多多少少地讲一点把。.lds 首先最重要的是定义程序的入口，ENTRY(phys_startup_32)指明程序的入口

点：phys_startup_32 标号；随后.=0xC0100000 指明代码段起始虚拟地址；.text : { ....}表示从该位置开始放置所有目标文件的代码段，里面又分几个小区段。又如：

```
_text = .;
_stext = .;
__sched_text_start = .;
```

这些是代码段.text 中又分的几个小区段，设置_text 这样的符合作为地址常量。

再来，每个小区段中，如_stext 中，*(.text.hot) *(.text) *(.ref.text) *(.devinit.text) *(.devexit.text) *(.cpuinit.text) *(.cpuexit.text) *(.text.unlikely)意思是将所有输入文件的.text.hot、.text、.ref.text 节合并到这里。这些节来自哪里？说白了就是来自前面看到.vmlinux.cmd 文件中的那些 built-in.o 文件中。

再解释一下 ALIGN(8)，表示.(.是 current location counter)必需要对齐，如果没有这行的话会有隐患（我想是总线数据必须以 8 位对齐以增加效率吧）。

我这里只总结一下最顶层有那些段，详细的大家可以去看一下自己生成的文件。.text 段最重要的是_text，表示代码段开始地址；_stext 表示实际代码段（除去前面的 head 部分）起始地址；_etext 表示代码段结束地址。代码段结束后，然后是. notes 段，即摘要段，主要存放一下内核版本信息。

接下来是几个符号地址__ex_table，对应代码里的__ex_table 表它看成异常段的入口地址。当异常发生时，内核的异常响应会从这里开始搜索 search_exception_table(unsigned long addr)看能不能找到相应的异常处理办法。接下来是.rodata 段，只读数据段（read-only data section, rodata），用于存放一些 debug 或 fixup 的数据，程序无法对其内容进行修改。

然后就是千呼万唤始出来的.data 段，内核数据段。跟.text 不一样，它没有_data，直接是_sdata 打头，_edata 结尾。随后是.vsyscall_0、.vsyscall_fn、.vsyscall_gtod_data、.vsyscall_clock、.vsyscall_1、.vsyscall_2、.vgetcpu_mode、.jiffies、.vsyscall_3 这些段。他们都是一些全局变量，我们最熟悉的莫过于 jiffies 了，其实 vsyscall 也很重要，用于快速加载系统调用号到 eax 寄存器。其他的段我们就不去讨论他们了，感兴趣的可以深入研究一下。

接下来是内核初始化相关的段，主要有.init.begin、.init.text、.init.data 等段，最后以.init.end 段的__init_end 符号结束。接下来还有两个重要的段，.bss 段和.brk 段，分别以__bss_start 开始__bss_stop 结束和以__brk_base 开始__brk_limit 结束。BSS 段包含了内核中未初始化全局变量，在内存中 bss 段全部置零。BRK 段是保留给用户进程通过系统调用 brk() 向内核申请的内存空间。最后，以符号_end 结束了整个 SECTIONS 的工作，vmlinux 的链接也就结束了。

2.2.6 制作 bzImage

回到顶层 Makefile 的 849 行，随着 vmlinux 的链接工作结束，在主目录下生成了 vmlinux 文件，vmlinux 目标也就结束了。那么就会来到 arch/x86/Makefile 的 155 行 bzImage 目标。看到这个目标，就知道后面的工作就算打包压缩 vmlinux 并制作 bzImage 了。看到 156 行，因

为 CONFIG_X86_DECODER_SELFTEST 没有设置，所以会直接执行 159 行以后的命令：

```
159     $(Q)$(MAKE) $(build)=$(boot) $(KBUILD_IMAGE)
160     $(Q)mkdir -p $(objtree)/arch/$(UTS_MACHINE)/boot
161     $(Q)ln -fsn ../../x86/boot/bzImage $(objtree)/arch/$(UTS_MACHINE)/boot/$@
```

160 行是建立一个 arch/i386/boot 目录，161 行是建立一个符号链接，重中之重是 159 行，其中$(boot)宏来自 143 行，$(KBUILD_IMAGE)宏来自 153 行，所以翻译 159 行：

```
@make  -f  scripts/Makefile.build  obj=arch/x86/boot  arch/x86/boot/bzImage
```

前面提到 vmlinux 的入口地址为 phys_startup_32，该函数是工作在 32-bit 段寻址的保护模式，但是问题是系统自加电那一刻起，就运行于 16-bit 实模式。所以我们需要一些辅助程序从 16-bit 实模式转到 32-bit 或 64-bit 保护模式，设置好必须的参数后才能开启分页模式转到 32-bit 或 64-bit 分页保护模式。前半部分是由 arch/x86/boot/setup.bin 实现的，后半部分则是由 arch/x86/kernel/head.o 实现的。setup.bin 除了向保护模式转换外，还有很多其它的事情要做。

从 vmlinux 的链接过程来看 vmlinux 是一个已编好的 kernel，和普通的 ELF 可执行文件没有什么区别：

```
[root@localhost linux-2.6.34.1]# file ./vmlinux
vmlinux: ELF 32-bit LSB executable, Intel i386, version 1 (SYSV), statically linked, not stripped
```

我们知道，C 和汇编源文件被编译成 ELF 文件后，通常会包含有连接器（Linker）或加载器(Loader)所需要的 ELF header，Program header table 或 Section header table。这些 ELF header 和一些 section 的作用是告诉内核 ELF Loader 如何载入 ELF 可执行文件。但是， Linux 内核作为一种特殊的 ELF 文件，没有 ELF Loader 的帮助，需要特殊辅助程序去装载它。它的装载地址是固定的，前面讲了的，0xC0100000。

这时，为了保证通用性而存在的 ELF header 和一些 section 对内核的装载就没有意义了。为了使内核尽可能小，可以把这些信息去掉。所以通常采用 objcopy 命令来去掉原来 ELF 文件中的 header 和 section，同时转化为 raw binary 格式的文件。即便如此，通过 objcopy 处理过的 vmlinux 一般需要压缩以后再重新链接，当然必须把解压缩的程序也同时链接进来。

这个压缩的过程着实奇怪：压缩后，然后再解压缩，岂不是浪费启动时间？这还是因为当 x86 系列处理器启动初期，处于实模式状态，可以寻得的地址空间十分有限，仅仅是 1MB，如果内核过大，就无法加载。更新的内核（从 2.6.30 开始）甚至提供了更高压缩率的格式：bzip 或 LZMA。

由上面的分析可以看出， bzImage 至少包含 kernel booting 辅助程序和压缩的 vmlinux。这可以从 bzImage 的规则链得到验证：

刚才看到了，arch/x86/Makefile 中关于 bzImage 的规则调用命令为：

```
make -f scripts/Makefile.build obj=arch/x86/boot arch/x86/boot/bzImage
```

注意：这个命令为 Makefile.build 指定了目标 arch/x86/boot/bzImage ，而不是使用缺省的 __build。

arch/x86/boot/Makefile 中 bzImage 的规则如下：

```
81 $(obj)/bzImage: $(obj)/setup.bin $(obj)/vmlinux.bin $(obj)/tools/build FORCE
82         $(call if_changed,image)
83         @echo 'Kernel: $@ is ready' ' (#'`cat .version`')'
```

其中 obj= arch/x86/boot，所以 arch/x86/boot/bzImage 依赖的目标是：arch/x86/boot/setup.bin、arch/x86/boot/vmlinux.bin、arch/x86/boot/tools/build。tools/build 已加到 hostprogs-y 中去了，会生成相应的主机程序。tools/build 主机程序的主要工作是将 arch/x86/boot/setup.bin 和 arch/x86/boot/vmlinux.bin 拼接在一起，后面会看到。下面我们将分别讲述 setup.bin 和 vmlinux.bin 的生成过程。

先来梳理一下前面讲的系统启动流程。BIOS 从启动设备(如 Hard Disk 的 MBR)中装载 grub：stage1 是简单的 bootsector，只占一个扇区；然后是 stage2，有很多扇区，要分好几个阶段才能完全装入。

bootsector，装载 setup.bin，然后跳入 setup.bin 的入口函数_start(arch/x86/boot/header.S)。setup.bin 主要作用就是完成一些系统检测和环境准备的工作，其函数调用顺序为：

```
_start
->start_of_setup, 这两个函数都定义在 arch/x86/boot/header.S
->main(arch/x86/boot/main.c)
->goto_protect_mode(arch/x86/boot/pm.c)
->protect_mode_jump(arch/x86/pmjump.S)
```

main.c 主要功能为检测系统参数如: Detect memory layout, set video mode 等，后面会详细分析，最后调用 goto_protect_mode，设置 32-bit 或 64-bit 保护段式寻址模式。注意: main.o 编译为 16 位实模式程序。goto_protect_modea 则会调用 protected_mode_jump。

而在 arch/x86/pmjump.S 中，protected_mode_jump 最后几行就相当于 ljmpl __BOOT_CS:0，实际地址为 0x00000000，也就是 vmlinux.bin 中的 startup_32 函数。注意，整个内核中有两个 startup_32 函数，一个在 arch/x86/boot/compressed/head_32.S，另一个呢，是在 arch/x86/kernel/head_32.S。那么 vmlinux.bin 的中的是哪个呢？别忘了，之前递归编译各对象，并且链接 vmlinux 时，没有用到 arch/x86/boot 对象，说明 arch/x86/kernel/head_32.S 的 startup_32 函数是被链接进了 vmlinux，随后，vmlinux 将会被压缩，所以，这一个 startup_32 就不存在了。

看到 arch/x86/boot/compressed/head_32.S 的入口函数 startup_32，startup_32 会解压缩 kernel：

```
 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
139             addl    $20, %esp
167/*
168 * Jump to the decompressed kernel.
169 */
170             xorl    %ebx, %ebx
171             jmp     *%ebp
```

看到 138 行，这段汇编语句调用 decompress_kernel 函数对内核进行解压缩，前面 129~137 行为该函数准备参数，这个函数及其参数的具体分析后边会介绍，这里只是说明有这么一个过程。最后两条命令会跳入解压缩后 kernel 入口函数：startup_32 (arch/x86/kernel/head_32.S)。

那么 arch/x86/boot/compressed/head_32.S 是如何加入到 vmlinux.bin 中的呢？马上谈到，现在先继续看到 arch/x86/boot/Makefile，setup.bin 的目标生成规则链为：

```
113 LDFLAGS_setup.elf       := -T
114 $(obj)/setup.elf: $(src)/setup.ld $(SETUP_OBJS) FORCE
115         $(call if_changed,ld)

117 OBJCOPYFLAGS_setup.bin  := -O binary
118 $(obj)/setup.bin: $(obj)/setup.elf FORCE
119         $(call if_changed,objcopy)
```

if_changed 跟我们的 if_changed_rule 一样，都是定义于 Makefile.lib 中，所以我们还是去查看.setup.elf.cmd，看看其最终链接命令为：

```
cmd_arch/x86/boot/setup.elf := ld -m elf_i386    -T arch/x86/boot/setup.ld arch/x86/boot/a20.o arch/x86/boot/bioscall.o arch/x86/boot/cmdline.o arch/x86/boot/copy.o arch/x86/boot/cpu.o arch/x86/boot/cpucheck.o arch/x86/boot/edd.o arch/x86/boot/header.o arch/x86/boot/main.o arch/x86/boot/mca.o arch/x86/boot/memory.o arch/x86/boot/pm.o arch/x86/boot/pmjump.o arch/x86/boot/printf.o arch/x86/boot/regs.o arch/x86/boot/string.o arch/x86/boot/tty.o arch/x86/boot/video.o arch/x86/boot/video-mode.o arch/x86/boot/version.o arch/x86/boot/video-vga.o arch/x86/boot/video-vesa.o arch/x86/boot/video-bios.o -o arch/x86/boot/setup.elf
```

.setup.bin.cmd 的内容：

```
cmd_arch/x86/boot/setup.bin := objcopy -O binary arch/x86/boot/setup.elf arch/x86/boot/setup.bin
```

值得一提的是形成 setup.elf 的链接脚本 arch/i386/boot/setup.ld：

```
1/*
2 * setup.ld
```

```
 3 *
 4 * Linker script for the i386 setup code
 5 */
 6OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
 7OUTPUT_ARCH(i386)
 8ENTRY(_start)
 9
10SECTIONS
11{
12              . = 0;
13              .bstext                 : { *(.bstext) }
14              .bsdata                 : { *(.bsdata) }
15
16              . = 497;
17              .header                 : { *(.header) }
18              .entrytext              : { *(.entrytext) }
19              .inittext               : { *(.inittext) }
20              .initdata               : { *(.initdata) }
21              __end_init = .;
22
23              .text                   : { *(.text) }
24              .text32                 : { *(.text32) }
25
26              . = ALIGN(16);
27              .rodata                 : { *(.rodata*) }
28
29              .videocards             : {
30                      video_cards = .;
31                      *(.videocards)
32                      video_cards_end = .;
33              }
34
35              . = ALIGN(16);
36              .data                   : { *(.data*) }
37
38              .signature              : {
39                      setup_sig = .;
40                      LONG(0x5a5aaa55)
41              }
42
43
44              . = ALIGN(16);
45              .bss                    :
46              {
```

```
47                      __bss_start = .;
48                      *(.bss)
49                      __bss_end = .;
50              }
51              . = ALIGN(16);
52              _end = .;
53
54              /DISCARD/ : { *(.note*) }
55
56              /*
57                * The ASSERT() sink to . is intentional, for binutils 2.14 compatibility:
58                */
59              . = ASSERT(_end <= 0x8000, "Setup too big!");
60              . = ASSERT(hdr == 0x1f1, "The setup header has the wrong offset!");
61              /* Necessary for the very-old-loader check to work... */
62              . = ASSERT(__end_init <= 5*512, "init sections too big!");
63
64}
```

setup.ld 定义了输出文件的 section 组织顺序：由于 header.o 位于 setup.elf 最前部，只有 arch/x86/boot/header.o 定义了这些 section：
.bstext、.bsdata、.header、.inittext、.initdata

后面会提到 header 是整个实模式阶段的开端，0-496 字节为.bstext 和.bsdata 段。497 开始为.header，.inittext，.initdata，.text 段的数据。497 - 511 的 15 个字节为缺省参数， tools/build 在生成 bzImage 的过程中会改变某些缺省值。

_start 函数进入点正好在 512 偏移地址处：

```
[root@localhost boot]# nm setup.elf |grep " _start"
0000000000000200 R _start
```

arch/x86/boot/video-*.c 都定义有__videocard，其最终定义为：

```
#define __videocard struct card_info __attribute__((section(".videocards")))
```

所有以__videocard 属性修饰的数据都会放入.videocards 段。

再看到 45 行，.bss 段并不会占用实际存储空间，setup.bin 的.signature 位于文件末尾：

```
[root@localhost boot]# hexdump -C setup.bin | tail -2
00003730    28 36 00 00 55 aa 5a 5a                                       |(6..U.ZZ|
00003738
```

最后 59 到 62 行有三个很重要的 ASSERT 关键字，对连接后的 setup.bin 这么一个 c 程序进行检查，如果_end 地址大于 0x8000、hdr 不等于 0x1f1 或者__end_init 大于 5*512，就会发出一些警告信息。注意这个 hdr，叫做安装头（setup head），是由 bootloader 设置的一些系统主要安装信息。这个东西很重要，初始化很大一段程序会围绕着它工作。

下面再来看看 vmlinux.bin，vmlinux.bin 的目标生成规则链为：

```
86 $(obj)/vmlinux.bin: $(obj)/compressed/vmlinux FORCE
87         $(call if_changed,objcopy)
121 $(obj)/compressed/vmlinux: FORCE
122        $(Q)$(MAKE) $(build)=$(obj)/compressed $@
```

而 arch/x86/boot/compressed/Makefile 中 vmlinux 规则为：

```
26$(obj)/vmlinux: $(obj)/vmlinux.lds $(obj)/head_$(BITS).o $(obj)/misc.o $(obj)/piggy.o FORCE
27         $(call if_changed,ld)
28         @:
```

compressed 目录下 Makefile 的编译过程要复杂一些，我们看到：

```
targets := vmlinux.lds vmlinux vmlinux.bin vmlinux.bin.gz vmlinux.bin.bz2 vmlinux.bin.lzma
vmlinux.bin.lzo head_$(BITS).o misc.o piggy.o
```

所以大致分为几步：

1. 通过编译 vmlinux.lds.S 生成链接脚本 vmlinux.lds。
2. 使用 objcopy 命令从顶层目录拷贝刚刚生成的 vmlinux 到 arch/x86/boot/compressed/目录中，并删除其中的.comment 段，也就是把注释给删掉。
3. 根据编译选项，选择一个压缩程序，对上一步的 vmlinux.bin 进行压缩，由于我使用的默认配置是 CONFIG_KERNEL_GZIP，所以生成的是 vmlinux.bin.gz 文件。
4. 编译 head_32.S 生成 head_32.o。
5. 编译 misc.c，生成 misc.o。
6. 编译 piggy.S 汇编源文件，生成 piggy.o。
7. 使用 arch/x86/boot/compressed/vmlinux.lds 链接脚本将 head_32.o，misc.o，piggy.o 链接生成 arch/x86/boot/compressed/vmlinux。

当 arch/x86/boot/compressed/目录中的 Makefile 执行完毕后，会在其目录下生成一个新的 vmlinux。随后 arch/x86/boot/目录的 Makefile 使用 objcopy 命令拷贝刚刚生成的 vmlinux 除掉 ELF header 和.note 段等无用信息后便在 arch/x86/boot/目录下生成另一个二进制格式的 vmlinux.bin。

一定要注意，此时就有了两个 vmlinux，一个在顶层目录，一个在 arch/x86/boot/compressed/目录下；还有两个 vmlinux.bin，一个在 arch/x86/boot/compressed/，另一个在 arch/x86/boot/目录下。而最终，我们 bzImage 需要的 vmlinux.bin 是 arch/x86/boot 下的那个。

vmlinux.bin 和 setup.bin 工作完成后，就开始链接 bzImage 了。arch/x86/boot/tools/build 是用于构建最终 bzimage 的实用程序，他的作用就是把 setup.bin 和 vmlinux.bin 连接到一起：setup.bin 按照 512 字节对齐，同时负责把 rootdev，内核 crc，以及 setup 和 kernel 的大小，patch 到 setup.bin 开头的 arch/x86/boot/head_32.S 中。我们来看看.bzImage.cmd 的内容：

```
cmd_arch/x86/boot/bzImage := arch/x86/boot/tools/build arch/x86/boot/setup.bin
arch/x86/boot/vmlinux.bin CURRENT > arch/x86/boot/bzImage
```

很简单的拼接成功后，就会在 arch/x86/boot/目录下生成 bzImage 文件。至此，需要告诉你的

内核映像生成过程就结束了，我们总结一下：最终 bzImage 由两部分顺序拼接而成：setup.bin 和 vmlinux.bin。其中 setup.bin 通过 setup.ld 链接脚本涵盖了 arch/x86/boot 目录下几乎所有的代码，其重中之重是该目录下的 header 文件，它是整个内核部分的最先被执行的实模式代码；而 vmlinux.bin 不仅包含了 vmlinux 的压缩代码和一个对其解压缩的程序，还包含了 arch/x86/boot/compressed 目录下的 head_32.S 代码，该代码最重要的是包含了保护模式的入口函数 startup_32。

而至于 make 随后的三个命令：make modules、make modules_install 和 make install 的执行过程，我们就不详细分析了，这里只简单提一下 make install，是执行这么一个命令：

```
sh /usr/src/kernels/linux-2.6.34.1/arch/x86/boot/install.sh 2.6.34.1 arch/x86/boot/bzImage \
        System.map "/boot"
```

我们来看看 arch/x86/boot/install.sh 脚本的内容：

```
20verify () {
21        if [ ! -f "$1" ]; then
22                echo ""                                                   1>&2
23                echo " *** Missing file: $1"                              1>&2
24                echo ' *** You need to run "make" before "make install".' 1>&2
25                echo ""                                                   1>&2
26                exit 1
27        fi
28}
29
30# Make sure the files actually exist
31verify "$2"
32verify "$3"
33
34# User may have a custom install script
35
36if [ -x ~/bin/${INSTALLKERNEL} ]; then exec ~/bin/${INSTALLKERNEL} "$@"; fi
37if [ -x /sbin/${INSTALLKERNEL} ]; then exec /sbin/${INSTALLKERNEL} "$@"; fi
38
39# Default install - same as make zlilo
40
41if [ -f $4/vmlinuz ]; then
42        mv $4/vmlinuz $4/vmlinuz.old
43fi
44
45if [ -f $4/System.map ]; then
46        mv $4/System.map $4/System.old
47fi
48
49cat $2 > $4/vmlinuz
50cp $3 $4/System.map
```

```
51
52if [ -x /sbin/lilo ]; then
53          /sbin/lilo
54elif [ -x /etc/lilo/install ]; then
55          /etc/lilo/install
56else
57          sync
58          echo "Cannot find LILO."
59fi
```

很简单的一个安装脚本，主要做了以下三个工作：

1. 我们把刚才压缩出来的内核映像 bzImage，拷入到/boot 目录，名字改成 vmlinuz-2.6.34.1；
2. 调用 mkinitrd 程序来创建 imitrd-xxx.img 文件，其作用我们前边已经讲过了。其中 xxx 为内核的版本号，是通过查看 /lib/modules 来版本来对应的，我们是编译出来的是 2.6.34.1，所以就运行上面的命令创建，创建的出来的是 initrd-2.6.34.1.img。不创建这个文件，有时是启动不起来的，比如提示 VFS 错误等；
3. 拷贝 System.map 符号映射表到/boot 目录下。这个表是把内核所有的全局变量和它对应的虚拟地址全部列出来，感兴趣的同学可以去 cat 一下。

好了，通过对内核映像的编译，我们了解了它是怎么形成的，以及一些重要的编译、链接知识。这些东西对我们后面研究系统如何初始化，乃至整个系统的工作模式都很重要，希望给大家的 Linux 知识的学习带来帮助。

3 实模式下的内核代码

通过上一章节，大家都看到了，最终每个 Linux 系统都有一个类似/boot/vmlinuz-2.6.x 的文件，就是所谓的内核映像。而大家对从源代码到内核映像的形成有了一个较完整的认识。前面也提到了，系统启动的时候，grub 会把内核映像加载到内存，然后执行它。总结一下 Grub 的这一个过程，就是：

1. 调用一个 BIOS 过程显示“ Loading ”信息。
2. 调用一个 BIOS 过程从磁盘装入内核映像的初始部分，即将内核映像的第一个 512 字节加载到物理地址 0x00090000 开始的内存中，而将 setup 程序的代码（参见后面的内存布局）从地址 0x00090200 开始存入 RAM 中。
3. 调用一个 BIOS 过程从磁盘中装载其余的内核映像，并把内核映像放入从低地址 0x00010000（适用于使用 make zImage 编译的小内核映像）或者从高地址 0x00100000（适用于使用 make bzImage 编译的大内核映像，也就是我们现在的情况）开始的 RAM 中。大内核映像的支持虽然本质上与其他启动模式相同，但是它却把数据放在不同的物理内存地址，以避免 ISA 黑洞问题。
4. 跳转到 arch/x86/boot/header.S 的_start 处开始执行。

注意 grub 的 stage2 会临时地打开保护模式，完成上述的第 3 步，把内核映像 vmlinuz-2.6.x 除 setup 的其余部分从磁盘整体加载到内存，然后再回到实模式下。所以此时，在内存中的内核映像分成了两个部分，实模式部分和保护模式部分，其分界线就是物理地址 0x100000。而实模式部分也分成了两部分，bootsect 部分和 setup 部分。

3.1 内核映像内存布局

针对多种体系结构，linux 使用不同的 bootloader，因为我看的代码是 x86 下的，所以我们全文的内核引导程序都指的是 grub。其实每个 bootloader 都很复杂，比如 grub 如何读，把它加载到哪，怎么执行内核。

第一个问题，grub 是怎么读内核映像的？好吧，再说一次，BIOS 启动后，grub 首先根据系统盘的第一个块，即 512 字节中的数据（bootsect）来判断识别文件系统的代码（512 字节显然不能识别文件系统）在磁盘中的什么块，这些数据是 grub 安装的时候记录的，然后执行下一个步骤，即识别系统盘的文件系统。

识别系统盘的文件系统主要是通过 initrd 程序来完成，前面也提到了，initrd 就像个小操作系统，但它的作用仅仅是让 grub 能够获得磁盘驱动程序以识别内核映像所在的文件系统，所以 initrd.img 压缩包并不大，只有 3.5MB 左右。initrd 程序还是很复杂的，只不过不是我们的研究重点，有兴趣的同志可以尝试解压缩 initrd-2.6.18-194.el5.img 文件（先要改名成.gz 文件，然后用 gunzip 程序解压缩），并将其 mount 到一个指定位置。这时你就可以看到，这个内存文件系统提供若干命令和脚本。其中最重要的是提供系统初始化的 linuxrc 脚本。必

须注意的是这里使用的 shell 是 nash 而不是 bash，nash 是专门为 linuxrc 可执行脚本设计的，因此你也有必要看一看 nash 的 man 文档。

现在就假设 grub 已经能识别文件系统了。接下来，首先 grub 通过该文件系统得到内核映像 /boot/vmlinuz-2.6.34.1（以后就简称 vmlinuz 了）文件。怎么处理它呢，这就要协议，(协议的详细信息参考 linux-2.6.34.1/Documentation/x86/boot.txt)。总的来说 vmlinuz 由三部分构成（我机器上的 vmlinuz 文件的大小约为 1.2MB）：

- 第一个是 512 字节的 bootsect（第一个块）
- 第二个是 setup 代码，若干不多个 512 字节（一会再说它多大）
- 保护模式下的内核代码

第一个 512 就是通常的启动扇区，对应于 ULK3 的远古时代（但是它有点特殊，因为它现在并不用作为启动扇区了，一会儿会看到)，以前是在 arch/x86/boot/bootsect.S 中，但是现在这个文件消失了，它到哪儿去了呢？

第二部分是实模式下 setup.bin 中的代码，对于 ULK3 中说的中世纪。也就是说，vmlinuz 的第二部分，以前是 arch/x86/boot/setup.S 代码，但是这个文件也失踪了。OK，揭开谜底了。第一段 bootsect.S 和第二段代码 setup.S(的部分)合并到 arch/x86/boot/header.S 中了，除此之外，后面还会看到，还有部分 c 代码，主要是用来建立保护模式的。

在链接脚本 setup.ld 中我们看到，setup.bin 的入口程序是_start，即跳过了它的第一个 512 字节。而 setup.bin 程序就是我们前面所说的 setup 程序，它内容来自 boot 目录下的其它一些源文件（参考 boot/Makefile)。这段代码的大小，在通过 setup.ls 链接脚本链接 setup.elf 的时候会得到，然后这个数据会写入第一个 512 字节中的偏移（也是整个 vmlinuz 的偏移）位置为 0x01F1 处的一个字节：

```
[root@localhost ~]# od -j 0x01F1 -N1 /boot/vmlinuz-2.6.34.1 -D
0000761             21
0000762
```

解释一下，od 命令的目的是输出文件内容。我们看到 vmlinuz 文件的 0x01F1 处的内容为 21，这个值表示第一部分和第二部分总共占据的块的个数。由于一个块是 512 字节，所有，表示 21*512 = (1+20)*512，就说明这个位于实模式下的 setup 程序的长度为 20*512 = 10K。

注意！回忆我们在制作 bzImage 一节中讲到连接 setup.bin 的链接脚本，hdr 的位置正好就是 0x1f1。整个初始化阶段，最重要的东西，就是这个位于 vmlinuz 第一个 512 字节的 497 偏移的 hdr。我们可以来看看 arch/x86/boot/header.S 下，从 96 行开始：

```
hdr:
setup_sects:    .byte 0                 /* Filled in by build.c */
root_flags:     .word ROOT_RDONLY
syssize:    .long 0                 /* Filled in by build.c */
ram_size: .word 0                   /* Obsolete */
……
```

这个 hdr 的第一个变量就是 setup_sects，占一个字节，用来存放整个一二部分的大小。其值再编译的时候由 build 程序写入。回忆一下，最后制作 bzImage 是就是利用的该程序。

第三部分就是内核映像的除 setup 后的其它部分，其入口是 arch/i386/boot/compressed 目录下 head_32.S 中的 startup_32。因为第三部分代码是进入保护模式后执行的，所以被称为保护模式内核代码。

好了，在对内核文件的构成有个了解之后来看看加载过程，
先看一段文字(Documentation/i386/boot.txt)：

```
For a modern bzImage kernel with boot protocol version >= 2.02, a
memory layout like the following is suggested:

         ~                        ~
         |  Protected-mode kernel |
         #这是上面说的保护模式代码, grub会把它放到这里
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.
         #上面这两个就是vmlinuz的前面两部分的代码和数据
X        +------------------------+
         |  Boot loader           |        <- Boot sector entry point 0000:7C00
001000   +------------------------+
         |  Reserved for MBR/BIOS |
000800   +------------------------+
         |  Typically used by MBR |
000600   +------------------------+
         |  BIOS use only         |
000000   +------------------------+

... where the address X is as low as the design of the boot loader
permits.
```

我们可以看到，上面是内核提供的文档对 grub 加载 vmlinuz 之后的物理内存布局的描述。按照协议，bootloader 被加载到了 0x1000 处，由于每种 bootloader 的大小不同，协议中设为 X，而 grub 设置的就是 0x90000，前面是提到过的。紧接着 vmlinuz 的前两部分在一起，也就是 setup.bin 的实模式部分代码。而第三部分在 0x100000 开始。

注意，grub 加载 vmlinuz 后的内存布局非常重要，主要分为实模式部分和保护模式部分。理解了这部分，大家才会看懂我后面的讲解。

3.2 实模式汇编代码 header.S

如上所述，第一和第二部分是实模式代码，这些代码来自于汇编程序 arch/x86/boot/header.S 和 c 程序 arch/x86/boot/main.c。

再从加载开始说，如上所述，vmlinuz 保护模式的代码加载到 0x100000 开始的位置。而实模

式的代码，因为被加载的位置不要求是固定的，也就是上面文档中看到的：

Kernel setup	|	The kernel real-mode code.
Kernel boot sector		

它们的位置是 X，其不确定性是受 Boot loader，也就是 grub 大小的影响。

3.2.1 无用的 bootsect 代码

所以，我们先来看 header.S 的代码，第 46 行：

```
46              .global bootsect_start
47 bootsect_start:
48
49              # Normalize the start address
50              ljmp    $BOOTSEG, $start2
```

这是 bootsect 开始代码，也就是 vmlinuz 第一个 512 字节的源代码。

我们看到$BOOTSEG 是在源代码的第 28 行定义的 BOOTSEG=0x07C0。没错，这个 ljmp 的意思就是说的是跳转到 07C0 的偏移 start2 处。还记得我们 BIOS 一节说过，BIOS 不管你是内核还是 bootloader，总会一来把第一个块加载到内存然后执行 0x07c0 处的代码。那么如果从这里开始执行，就说明 vmlinuz 是被 BIOS 直接加载过来的，这是不允许的，因为现在 linux 都需要经过一个特定的 bootloader，如前面大篇幅介绍的 GRUB。这也就是前面说的 bootsect 有点特殊的地方，就是说它并没打算用来执行，是个死代码。所以万一它被作为 bootsect 由 BIOS 直接执行，那么就直接提示 reboot。可以拿 vmware 实验一下：

```
dd if=/boot/vmlinuz-2.6.34.1 of=vm.img bs=512 count=1
```

然后用 vm.img 作为软盘启动。程序就会跳到$start2 处：

```
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
```

我们看到，start2 首先将将 ds,es,ss 全设置为 cs，其内容为 0x07C0；然后将 sp 设置成 0、开中断 sti、cld 清方向标志最后把下面 bugger_off_msg 符号的偏移地址放在 si 寄存器中。继续走：

```
63 msg_loop:
64              lodsb
65              andb        %al, %al
66              jz          bs_die
67              movb        $0xe, %ah
68              movw        $7, %bx
69              int         $0x10
70              jmp         msg_loop
71
```

63 行的 msg_loop 的 lodsb 指令把 si 指向的源串的内容逐步装入 al 中（另一个指令 stosb 是将 al 中数据装入 di 指向的地址中）。64 行，当取完的时候，al=0，此时操作 andb 后方向为 0,执行下句跳转语句到 bs_die。当然，al 不为 0 的时候，执行 69 行 Video 中断服务指令，其中 AH = 0Eh，AL = 对应字符（8 位），BL = Color (only in graphic mode)。

```
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
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
91              # Kernel attributes; used by setup.    This is part 1 of the
92              # header, from the old boot sector.
93
```

上面这些代码的作用就是打印消息并等待重启，就不多说了。在这里顺便说一下，grub 的 boot_func 中的 big_linux_boot 里，描述了实际上 grub 的 stage2 将内核的 bootsect 和 setup 实模式代码载入到地址 0x90000 后。也就是说，这里就是对应 Grub 是 0x90000 的证据。由于我们 setup.ld 链接脚本里规定的 setup.bin 入口点是_start，那么 grub 加载 vmlinuz 后将跳过前 512 个字节，即 0x200 个字节的，直接跳转到地址 0x90200 处执行的。我们还是来关注真正的 setup 代码了，从偏移内核映像 vmlinuz 的 0x0200 开始。

那么 grub 怎么执行这条代码呢？grub 会执行 jmp_far(0x20, 0)；也就是跳过 vmlinuz 的 0x200 个字节，也就是跳到 vmlinuz 实模式代码_start 执行了，因为这样就略过了前 512 字节的 bootsect。这也就是一开始我们说 bootsect.S 消失了的秘密。

实际上，jmp_far 就是 grub 中一条指令，ljmp。再看看第 105 行的注释，这就是 ljmp 跳到的位置，即_start，也就是进入“ 中世纪时代 ”。

3.2.2 初始化头变量 hdr

在讲解“ 中世纪时代 ”的代码之前，先详细介绍一个在初始化当中非常重要的内容，hdr，全称叫做“ setup_header ”。这个头变量存放着所有初始化期间使用到的数据，在编译 setup.bin 的时候存放在.header 段中，其代码我们看，从 arch/x86/boot/header.S 的第 96 行开始：

```
 94              .section ".header", "a"
 95              .globl  hdr
 96hdr:
 97setup_sects:       .byte 0                         /* Filled in by build.c */
 98root_flags:        .word ROOT_RDONLY
 99syssize:            .long 0                         /* Filled in by build.c */
100ram_size:           .word 0                         /* Obsolete */
101vid_mode:            .word SVGA_MODE
102root_dev:           .word 0                         /* Filled in by build.c */
103boot_flag:          .word 0xAA55
104
105             # offset 512, entry point
106
107             .globl  _start
108_start:
112                     .byte   0xeb
113                     .byte   start_of_setup-1f
1141:
118                     .ascii  "HdrS"
119                     .word   0x020a
121                     .globl realmode_swtch
122realmode_swtch: .word    0, 0
123start_sys_seg:  .word    SYSSEG
125                     .word   kernel_version-512
130type_of_loader: .byte    0
136loadflags:
137LOADED_HIGH        = 1
138CAN_USE_HEAP       = 0x80
143                     .byte   LOADED_HIGH
```

```
145setup_move_size: .word    0x8000
152code32_start:
154                          .long      0x100000
156ramdisk_image:    .long      0
161ramdisk_size:     .long      0
163bootsect_kludge:
164                          .long      0
166heap_end_ptr:      .word      _end+STACK_SIZE-512
172ext_loader_ver:
173                          .byte      0
174ext_loader_type:
175                          .byte      0
177cmd_line_ptr:       .long      0
192ramdisk_max:        .long 0x7fffffff
200kernel_alignment:    .long CONFIG_PHYSICAL_ALIGN
203#ifdef CONFIG_RELOCATABLE
204relocatable_kernel:        .byte 1
205#else
206relocatable_kernel:        .byte 0
207#endif
208min_alignment:              .byte      MIN_KERNEL_ALIGN_LG2
209pad3:                              .word 0
211cmdline_size:      .long      COMMAND_LINE_SIZE-1
215hardware_subarch:             .long 0
218hardware_subarch_data:    .quad 0
220payload_offset:                .long ZO_input_data
221payload_length:                 .long ZO_z_input_len
223setup_data:                     .quad 0
```

浩浩荡荡 100 多行的代码，去掉冗余的注释之后，就所剩无几了。不过，从注释的规模看了，这段代码实在是太重要了，这里只大概的介绍一下。上一节虽然说，内核本身的 bootsect 没有什么用处，不过 hdr 的起始位置确是在这个 bootsect 当中，从 0x1f1 处开始的。所以话又说回来了，这个 bootsect 并不是一无是处。

hdr 位于数据段，其中部分内容在形成 bzImage 时，由 arch/x86/boot/tools/build 程序赋值，比如前面提到的 setup_sects，表明了前两部分实模式下内核映像的代码。其他的值我们以后碰到了再去详细的分析它。

3.2.3 准备实模式下 C 语言环境

好了，执行到 setup 代码了，header.S 几乎所有的代码都在准备实模式下的 C 语言环境。在讲解这部分代码之前先回顾一下 C 语言程序关于程序中.text，.data，.bss 等段的说明。由于历史原因，C 程序一直由下列几部分组成：

1、 正文段。这是由 CPU 执行的机器指令部分。通常，正文段是可共享的，所以即使是经常执行的程序(如文本编辑程序、C 编译程序、shell 等)在存储器中也只需有一个副本，另外，正文段常常是只读的，以防止程序由于意外事故而修改其自身的指令。

2、 初始化数据段。通常将此段称为数据段，它包含了程序中需赋初值的变量。例如， C 程序中任何函数之外的说明：
int maxcount = 99; 使此变量以初值存放在初始化数据段中。

3、 非初始化数据段。通常将此段称为 bss 段，在程序开始执行之前，内核将此段初始化为 0。函数外的说明：
long sum[1000] ; 使此变量存放在非初始化数据段中。

4、 栈。自动变量以及每次函数调用时所需保存的信息都存放在此段中。每次函数调用时，其返回地址、以及调用者的环境信息（例如某些机器寄存器）都存放在栈中。然后，新被调用的函数在栈上为其自动和临时变量分配存储空间。通过以这种方式使用栈， C 函数可以递归调用。

5、 堆。通常在堆中进行动态存储分配。由于历史上形成的惯例，堆位于非初始化数据段顶和栈底之间。

现在来看看 vmlinuz 第二个 512 字节处，也就是_start：

```
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                    # 0200 处的第一个字节是跳转指令
113                                                                 #（其中，0xeb 是该指令代码）。
114                             .byte               start_of_setup-1f       #  0201  是跳转距离，
```

(start_of_setup-1f)其实就是 setup 中的头部长度。

第一条指令，这是一条短跳转，跳过了一些数据（从标号 1 到 start_of_setup），由于以后会再提及这些数据，暂时不管他。注意这里，从_start 到 start_of_setup 的代码是些伪指令，即 setup 的初始化头变量 hdr，涉及的是内存布局，并不涉及具体的指令执行。此时，cs:eip 指向的是 start_of_setup 处的代码，所以现在接着看 start_of_setup：

```
240             .section ".entrytext", "ax"
241start_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
247#endif
```

上面代码用 13 号中断重设系统盘的磁盘控制器 ax=0x0 ，dl=0x80，不明白的请查看博文“ BIOS 系统服务 —— 直接磁盘服务 ”
http://blog.csdn.net/yunsongice/archive/2010/10/08/5927560.aspx

```
249# Force %es = %ds
250             movw    %ds, %ax
251             movw    %ax, %es
252             cld
```

先强制让附加数据段 es 的内容等于数据段 ds 的内容。接下来要为即将开启的 C 程序建立堆栈段了（其实 C 程序的运行条件很简单，只要你给我提供一个堆、一个栈，没堆栈不行 !）。

```
259             movw    %ss, %dx
260             cmpw    %ax, %dx            # %ds == %ss?
261             movw    %sp, %dx
262             je      2f                  # -> assume %sp is reasonably set
```

上面的代码比较 ds 和 ss 是否相等，注意 ax 里的还是 ds 的值，现在就是比较 ds 和 ss 的值，再把 dx 设置为栈顶指针 sp 值。随后判断，如果 ds=ss，则跳转到前面（f 表前，b 表后）的标号 2 处，注意， C 语言中把数据段当做堆栈段使用；不相等就新建一个栈（一般情况下是不会相等的，因为肯定是会有数据的）：

```
264             # Invalid %ss, make up a new stack
265             movw    $_end, %dx
266             testb   $CAN_USE_HEAP, loadflags
267             jz      1f
268             movw    heap_end_ptr, %dx
269 1:          addw    $STACK_SIZE, %dx
270             jnc     2f
271             xorw    %dx, %dx            # Prevent wraparound
```

上面首先把栈底设置_ end 给 dx。还记得制作 bzImage 时，链接 setup.elf 的那个脚本 setup.ld 吗？当时说它重要，这里就体现出来了，这个_end 就来自那里，是整个 setup.bin 的结尾。

所以，此时此刻这个_end 就应该作为 setup 阶段的 C 语言使用堆栈的临时栈顶。千万要注意，我们在链接 vmlinux 的时候使用的那个链接脚本也有一个_end 标号，但是它随着 vmlinux 被压缩进了 vmlinuz，所以不存在与 setup.bin 中的_end 相冲突的情况。

不过 C 语言有可能还要使用堆，所以随后一个 testb 命令，让两操作数（byte）做与运算，

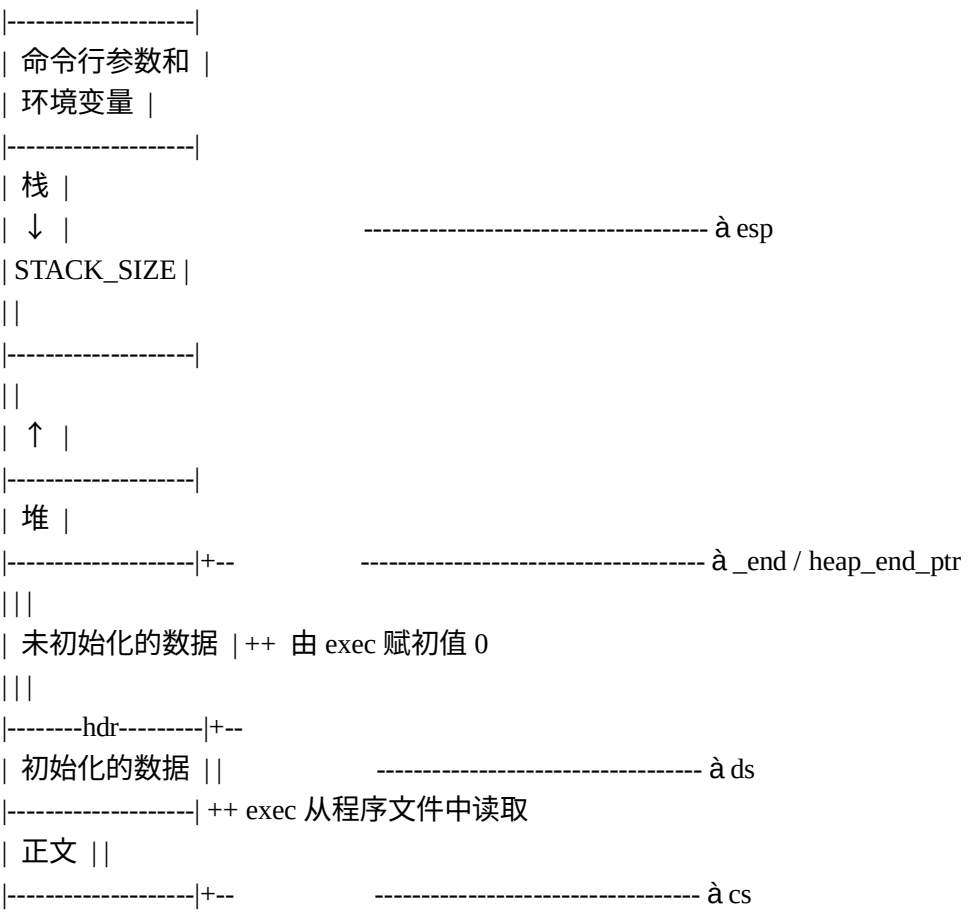
只修改标志位，看内核头参数中有没有设置 CAN_USE_HEAP 位。如果设置了 CAN_USE_HEAP 位，表示 C 语言可以使用堆，如果与运算结果等于 0，即没有设置这一位则跳到上面标号 1 处。当然，我们看到前面第 138 行 CAN_USE_HEAP 的值为 0x80，而在 arch/x86/include/asm/Bootparam.h 的 46 行定义的：
#define CAN_USE_HEAP (1<<7)

所以这一位是肯定有的，所以会用堆的顶部，即 heap_end_ptr 变量加到栈顶，随后将其值存放在 dx 寄存器中。heap_end_ptr 来自 166 行：
heap_end_ptr: .word _end+STACK_SIZE-512

我们看到 heap_end_ptr 的值就是_end 的地址，因为 STACK_SIZE 在 arch/x86/boot/Boot.h 中定义的值是 512：
/arch/x86/boot/Boot.h 中：
#define STACK_SIZE 512

顺便说一句,如何使用堆？其实就是通过 c 语言的 malloc 和 alloc 等函数人工分配数据空间。

继续走，到标号 1，栈顶还要加上一个 STACK_SIZE。为什么还要加一个 STACK_SIZE 呢？我们知道，x86 系列处理器的栈是向下递增的，也就是说，压栈动作，ss:esp 向低地址移动；出栈动作，ss:esp 向高地址移动。所以我结合前面回忆的 C 语言程序的特点，再根据刚才的内核映像 vmlinuz 内存布局简单地对实模式下的部分画了一个图：



从图中可以看到 STACK_SIZE 就是栈的大小，对于实模式下 c 语言环境，512 字节的栈大小足够了。同时，由于_end 和 heap_end_ptr 相等，所以，堆的起始地址就是_end。

好了，我们接着往下看：

```
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
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
```

如果无进位，就转移到标号 2 处。标号 2 代码的作用是将 dx 中的栈顶地址按双字对齐，即将最低两位清零。到标号 3 处，这时候，ax 是数据段寄存器 ds 的值，把它赋给堆栈段 ss 寄存器；dx 是经过刚才一系列过程以后的栈顶地址，赋给 esp 寄存器。此时此刻，ss:esp 开始工作！说明经过编译器编译的那些 C 代码可以工作了（主要就是那些 C 语言的函数使用堆栈段）。

从这里我们也可以看出，Linux 中，内核数据段与堆栈段其实都来自于内核数据段基址，只是偏移不同，esp 跨过了一部分内核数据段和堆，并向下、向低地址发展（如果有的话，即 DS≠SS）。我们继续往下看：

```
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
```

根据上面说的，grub 是以 ljmp 指令过来的，跳过来的时候数据段被设置为 X，即 0x90000。所以跳过来这后，cs 寄存器的值为 0x90020。这里做一个修正，因为堆栈已经准备好，那么上面三条语句利用堆栈指令将$6f 压栈，这样随后执行返回指令的话，就可以修正前面的偏差了，并且还可以测试一下刚刚建立的堆栈是否可靠，真是一举双得啊。

```
286 6:
287
288# Check signature at end of setup
289         cmpl    $0x5a5aaa55, setup_sig
290         jne     setup_bad
```

上面这段指令设置堆栈，对于正确的 setup ，cmpl 指令是总是对的，如果不对，就跳到 setup_bad。

```
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
```

上面这段代码清空 setup 的 bss 段。注意和数据段的区别，BSS 存放的是未初始化的全局变量和静态变量，数据段存放的是初始化后的全局变量和静态变量。

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

好啦，至此，代码奔向到 C 语言了，这 main 就是在 boot/main.c 中的 main 函数。后面的代码是一些错误处理，我们就不详细分析了。这里总结一下，汇编代码 header.S 从开始到 # offset 512, entry point 功能和以前的 bootsect 一样。后面的功能和 setup.S 的一部分类似，主要的工作是设置 setup header 参数部分；设置堆栈；检查 setup 中的标签；清除 BSS 段；调用 C 入口 main。此时 cs:eip 指向的是 main，ss:esp 指向的是堆栈栈顶，ds:edi 指向哪儿不知道，也不重要。

```
303# Setup corrupt somehow...
304setup_bad:
305             movl    $setup_corrupt, %eax
306             calll   puts
307             # Fall through...
308
309             .globl  die
310             .type   die, @function
311die:
312             hlt
313             jmp     die
314
315             .size   die, .-die
316
317             .section ".initdata", "a"
318setup_corrupt:
319             .byte   7
320             .string "No setup signature found...\n"
```

3.3 实模式代码 main 函数

前面 header.S 中最后跳到 main 函数，在 boot/main.c，void main()。再次强调一下，这一段 C

代码还是系统处于实模式下所执行的代码：

```
128void 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}
```

这个 main 函数里面的代码大部分对应 ULK3 中附录 A 讲 setup 函数的那一小节，主要是在实模式下做些前期工作。涉及初始化计算机中的硬件设备，并为内核程序的执行建立环境。虽然前面看到 BIOS 已经初始化了大部分硬件设备，但是 Linux 并不依赖于 BIOS，而是以自己的方式重新初始化设备以增强可移植性和健壮性。

这里只强调一个事情，接下来大部分工作都是在为进入保护模式而做准备。虽然现在仅仅工作在实模式中，A20 Gate 暂时关闭（后面会详谈），程序无法访问内存地址 0x100000 以后的内容，但并不意味着，0x100000 以后就没有东西。所以，grub 程序的 stage2 有个打开和关闭保护模式的过程，即将内核映像从磁拷贝到内存时，必须使用保护模式。拷贝完成后，关闭 A20 Gate 并且回到实模式下，但并不意味着高于 0x100000 的内存单元就没有内容了，而是保护起来了，不然怎么叫“保护模式”呢。

3.3.1 复制初始化头变量

131 行是第一个函数，copy_boot_params()，顾名思义用来拷贝启动参数到内存的 0 号页面中：

```
29static 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);
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}
```

第 35 行，const struct old_cmdline * const oldcmd，这里用到 const 的的概念了。首先，oldcmd 是个指针，指向 old_cmdline 结构。前面加上 const，说明是个指针常量。其*前面的 const，有说明这个 oldcmd 指向的内容也是一个常量。好了，这里就是两个常量。

接下来，39 行，将 hdr 的内容拷贝到全局变量 boot_params 的 hdr 字段中。注意哈，这两个 hdr 不是同一个东西一个是 header.S 文件中定义的数据段中的内容，一个是 boot_params 数据结构中的一个字段，只是名字叫 hdr。在这里第一次见到 boot_params，它是在 arch/x86/include/asm/Bootparam.h 文件中定义。这里第一次出现，接下来的大部分工作都是填充这个 boot_params。这个 boot_params 来头可不小，它刚好是一个页面的大小。在 arch/x86/boot/Main.c 的第 18 行：

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

注意，这个变量 boot_params 是 Main.c 的全局变量，且未被初始化，所以位于 BSS 段。grub 把 vmlinuz 加载进内存后，boot_params 就位于_bss_start 的开始位置。而后面当启动保护模式的分页功能后，第一个页面就是从它开始的（注意，不是从 0x0 开始的喔）。所以内核注释它为“ zeropage ”，即所谓的 0 号页面，足见这个 boot_params 的重要性：

```
84/* The so-called "zeropage" */
85struct 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));
```

就这个东西，把前面的初始化变量 hdr 拷贝到 boot_params 的 hdr 字段中。它的 hdr 字段是个 setup_header 结构，定义在同一个头文件中：

```
 24struct 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;
 40              __u16   kernel_version;
 41              __u8    type_of_loader;
 42              __u8    loadflags;
 43#define LOADED_HIGH     (1<<0)
 44#define QUIET_FLAG      (1<<5)
 45#define KEEP_SEGMENTS   (1<<6)
 46#define CAN_USE_HEAP    (1<<7)
 47              __u16   setup_move_size;
 48              __u32   code32_start;
 49              __u32   ramdisk_image;
```

```
50          __u32   ramdisk_size;
51          __u32   bootsect_kludge;
52          __u16   heap_end_ptr;
53          __u8    ext_loader_ver;
54          __u8    ext_loader_type;
55          __u32   cmd_line_ptr;
56          __u32   initrd_addr_max;
57          __u32   kernel_alignment;
58          __u8    relocatable_kernel;
59          __u8    _pad2[3];
60          __u32   cmdline_size;
61          __u32   hardware_subarch;
62          __u64   hardware_subarch_data;
63          __u32   payload_offset;
64          __u32   payload_length;
65          __u64   setup_data;
66} __attribute__((packed));
```

细心的朋友可能已经发现了，这些字段跟我们前面在 header.S 中看到的初始化头变量 hdr 中的字段是一一对应的，所以在 c 代码里面才能通过 memcpy 的方式拷贝过来。

随后的 if 语句针对老内核，对 boot_params.hdr 的 cmd_line_ptr 字段做些调整，我们就不管他了。

3.3.2 初始化堆

回到 main 函数，第二个函数，init_heap()，用来检查内核初始化阶段使用的堆：

```
109static 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
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;
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}
```

这个函数也简单，跟我们在“准备实模式下 c 语言环境”中一样，hdr 字段的 loadflags 标记如果 CAN_USE_HEAP 被置位，则说明本体系支持堆。我们这里用的 x86 最新的协议时支持的，所以进入 if 分支。If 分支内检查堆的大小，不能益出，如果发现溢出，就调整到 stack_end。

3.3.3 确保支持当前运行的 CPU

看过了 init_heap 函数，我们继续看第三个函数，validate_cpu()：

```
35int validate_cpu(void)
36{
37              u32 *err_flags;
38              int cpu_level, req_level;
39              const unsigned char *msg_strs;
40
41              check_cpu(&cpu_level, &req_level, &err_flags);
42
43              if (cpu_level < req_level) {
44                      printf("This kernel requires an %s CPU, ",
45                              cpu_name(req_level));
46                      printf("but only detected an %s CPU.\n",
47                              cpu_name(cpu_level));
48                      return -1;
49              }
50
51              if (err_flags) {
52                      int i, j;
53                      puts("This kernel requires the following features "
54                              "not present on the CPU:\n");
55
56                      msg_strs = (const unsigned char *)x86_cap_strs;
57
58                      for (i = 0; i < NCAPINTS; i++) {
59                              u32 e = err_flags[i];
60
61                              for (j = 0; j < 32; j++) {
```

```
62                                      if (msg_strs[0] < i ||
63                                          (msg_strs[0] == i && msg_strs[1] < j)) {
64                                              /* Skip to the next string */
65                                              msg_strs += 2;
66                                              while (*msg_strs++)
67                                                      ;
68                                      }
69                                      if (e & 1) {
70                                              if (msg_strs[0] == i &&
71                                                  msg_strs[1] == j &&
72                                                  msg_strs[2])
73                                                      printf("%s ", msg_strs+2);
74                                              else
75                                                      printf("%d:%d ", i, j);
76                                      }
77                                      e >>= 1;
78                              }
79                      }
80                      putchar('\n');
81                      return -1;
82              } else {
83                      return 0;
84              }
85}
```

这个函数也比较简单，注意第 41 行，check_cpu，通过指令读取 cpu 的信息，存放在 cpu_level、req_level 和 err_flags 三个内部变量中。req_level 表示内核需要运行的 CPU 最低版本，cpu_level 表示系统实际的 CPU 版本，当然，如果 cpu_level < req_level，那自然就报错啦。至于具体用了哪些汇编指令，自己去/arch/x86/boot/Cpucheck.c 中看，这里不在话下。

3.3.4 设置 BIOS 的 x86 模式

继续走，下一个函数，set_bios_mode()，跟 main 来自同一个文件：

```
 94/*
 95 * Tell the BIOS what CPU mode we intend to run in.
 96 */
 97static void set_bios_mode(void)
 98{
 99#ifdef CONFIG_I386
100             struct biosregs ireg;
101
```

```
102             initregs(&ireg);
103             ireg.ax = 0xec00;
104             ireg.bx = 2;
105             intcall(0x15, &ireg, NULL);
106#endif
107}
```

当然，set_bios_mode 函数只针对 i386，用于告诉 BIOS 什么 CPU 我们将要去运行。对于 x86_64 来说，是个空函数。这个过程如何处理呢？看到 105 行，这个 intcall 是初始化阶段的中断处理函数（牢记，初始化阶段至此，仍然处于实模式阶段，Linux 内核的中断系统还没有被初始化，这个函数是内核编译后临时生成的一个 bios 服务程序，跟前面 header.S 中的那些 int 指令效果是一样的），就是将 BIOS 的寄存器设置成 ireg 变量。该变量分别针对 32 位、16 位和 8 位而声明称以下一些联合体：（arch/x86/boot/Boot.h）

```
struct biosregs {
    union {
        struct {
            u32 edi;
            u32 esi;
            u32 ebp;
            u32 _esp;
            u32 ebx;
            u32 edx;
            u32 ecx;
            u32 eax;
            u32 _fsgs;
            u32 _dses;
            u32 eflags;
        };
        struct {
            u16 di, hdi;
            u16 si, hsi;
            u16 bp, hbp;
            u16 _sp, _hsp;
            u16 bx, hbx;
            u16 dx, hdx;
            u16 cx, hcx;
            u16 ax, hax;
            u16 gs, fs;
            u16 es, ds;
            u16 flags, hflags;
        };
        struct {
            u8 dil, dih, edi2, edi3;
```

```
                u8 sil, sih, esi2, esi3;
                u8 bpl, bph, ebp2, ebp3;
                u8 _spl, _sph, _esp2, _esp3;
                u8 bl, bh, ebx2, ebx3;
                u8 dl, dh, edx2, edx3;
                u8 cl, ch, ecx2, ecx3;
                u8 al, ah, eax2, eax3;
            };
        };
};
```

当然，102 行的 initregs()函数是用来做一般初始化的：

```
void initregs(struct biosregs *reg)
{
        memset(reg, 0, sizeof *reg);
        reg->eflags |= X86_EFLAGS_CF;
        reg->ds = ds();
        reg->es = ds();
        reg->fs = fs();
        reg->gs = gs();
}
```

我把它贴出来的目的主要是为了说明一下 ds()等函数，用来获取段寄存器：

```
static inline u16 ds(void)
{
        u16 seg;
        asm("movw %%ds,%0" : "=rm" (seg));
        return seg;
}
```

3.3.5 内存的检测

好啦，回到 main 函数继续我们的旅程，147 行 detect_memory()，用于探测物理内存的布局。
注意哈，这里是第一次出现与内存管理相关的代码。

```
122int 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}
```

detect_memory()函数主要根据物理内存的类型探测内存布局，代码非常简单，由上面可知，linux 内核会分别尝试调用 detect_memory_e820()、detcct_memory_e801()、detect_memory_88()获得系统物理内存布局，这 3 个函数内部其实都会以内联汇编的形式调用 bios 中断以取得内存信息。前面已经看到了该中断调用形式为 int 0x15，同时调用前分别把 AX 寄存器设置为 0xe820h、0xe801h、0x88h，关于 0x15 号中断的具体作用我这里就不再说明，有兴趣的可以去查询相关手册。下面我仅分析下 detect_memory_e820()的代码，因为它是当今 x86 体系的一般情况，其它代码也基本一样：

```
20static 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}
```

第 23 行的 biosregs 数据结构前面已经看到过了，现在主要看 24 行用到的两个数据结构，来自/arch/x86/include/asm/e820.h：

```
53struct 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));
```

由于历史原因，一些 I/O 设备除了占据 I/O 端口，也会占据一部分内存物理地址空间。因此当内核初始化程序第一次跟内存管理打交道时，其可以使用的物理内存空间是不连续的。此时的内存被分成了很多段，每个段的属性也是不一样的。

int 0x15 查询物理内存时每次返回一个内存段的信息，因此要想返回系统中所有的物理内存，我们必须以循环的方式去查询，这就是为什么第 47 行开始，有个 do-while 循环。

detect_memory_e820()函数把 int 0x15 放到一个 do-while 循环里，每次得到的一个内存段放到 struct e820entry 里，而 struct e820entry 的结构正是 e820 返回结果的结构！而像其它启动时获得的结果一样，最终都会被放到 boot_params 里，e820 被放到了 boot_params.e820_map。

有了这个概念后，我们来看 detect_memory_e820 的代码流程，22 行，count 用于记录已检测到的物理内存段数目。23 行，函数体内部 biosregs 结构变量 ireg，oreg。24 行，另一个内部变量 desc，指向前面拷贝进来的启动参数 boot_params 的.e820_map 数组的头。注意，此时的启动参数的 e820_map 字段还是一个空数组，本函数的目的就是给这个数组初始化。25 行，我们看到注释写得也很清楚，static 关键字初始化的一个 e820entry 结构类型的变量，buf，其值全被清零了，它的作用是在函数体内作为一个临时缓存存放探测到的 e820entry 数据。

从 27 行开始，执行 initregs()，前面我们已经看过了，其目的在于把内存中的 biosregs 结构的 ireg 的 ds、es、fs、gs 设置成当前值，并把 eflags 标志寄存器 CF 置位（X86_EFLAGS_CF=0x00000001）。28 到 31 行把 ireg 的 ax、cx、edx 和 di 字段初始化了。注意第 30 行‘ SMAP ’表明中断调用正确；第 31 行，di 字段存放我们临时缓存的地址值。

好了，进入循环体。48 行首先执行一个 intcall，该函数前面已经遇到过，只是没有去说它，现在就来好好谈谈。在源代码中其只是在 arch/x86/boot/Boot.h 中声明了一下：
void intcall(u8 int_no, const struct biosregs *ireg, struct biosregs *oreg);

这个初始化的期间调用的函数是编译器在实模式环境下，才会编译成以下汇编过程：

首先调用 0x15 号 bios 中断：int 0x15。这条指令需要的输入参数就是上面 c 语言代码种提供 ireg 参数，它执行完后会对输出参数 oreg 设置以下一些寄存器的：调用成功则清除 CF 标志否则设置 CF 标志、调用成功设置 eax 为‘ SMAP ’否则设置 eax 为一个出错码、将返回的内存段描述符填入 edi 指向的内存空间（临时缓存）、ecx 填入返回的字节数、ebx 填入下一次迭代获取内存段描述符需要的一个值。

这些指令执行完毕后，我们的 oreg 变量就获得了一些值，后面就可以根据 oreg 变量的值来判断获取内存段描述符那条指令是否成功。所以，49 行 ireg.ebx = oreg.ebx 准备好下次循环，然后第 54 行判断 oreg 的 eflags 是否置位，如果是则说明失败，跳出循环。第 62 行，如果 eax 不为‘ SMAP ’表明中断都执行失败了，那么需要把 count 清零然后退出循环。如果上面两个条件都满足，那么好，把找到的内存信息复制到启动参数的 e820_map 数组的对应元素中。

OK，69 行，如果 ireg.ebx 为 0 了，或者 count 超过数组的最大下标，跳出循环。这里提一下，boot_params.e820_map 数组的最大下标是前面我们看到的那个 E820_X_MAX 宏，其值是 128。

最后设置 boot_params 中 e820_entries 的值为物理内存段的个数并返回物理内存段个数给 main 函数。

启动程序走到此，我们来回顾一下 boot_param 结构已经填充了哪些内容了，只有两个：104 行的启动头有了，在第一个函数 copy_boot_params()就把位于 header.S 中的那个 hdr 段拷贝

了进去；107 行也有了，就是刚刚 detect_memory()做的事情。

3.3.6 设置键盘属性

所以回到 main 函数中，继续走，下一个该执行 keyboard_set_repeat()函数了。

```
59/*
60 * Set the keyboard repeat rate to maximum.    Unclear why this
61 * is done here; this might be possible to kill off as stale code.
62 */
63static void keyboard_set_repeat(void)
64{
65              struct biosregs ireg;
66              initregs(&ireg);
67              ireg.ax = 0x0305;
68              intcall(0x16, &ireg, NULL);
69}
70
```

这个函数不需要多说，就是触发 bios 的第 16 号中断。这个中断的参数来自于 ireg 的 ax 寄存器，该参数的值为 0x0305，其作用为设置键盘重复延时和速率。从注释我们可以看出，写代码的人也不明白为什么这个时候要触发这么一个中断，估计是为了与老代码兼容吧。

3.3.7 填充系统环境配置表

继续走，来看 main 中的 query_mca()函数。这个函数来自 arc/x86/boot/Mca.c：

```
18int query_mca(void)
19{
20              struct biosregs ireg, oreg;
21              u16 len;
22
23              initregs(&ireg);
24              ireg.ah = 0xc0;
25              intcall(0x15, &ireg, &oreg);
26
27              if (oreg.eflags & X86_EFLAGS_CF)
28                              return -1;          /* No MCA present */
29
30              set_fs(oreg.es);
31              len = rdfs16(oreg.bx);
32
```

```
33              if (len > sizeof(boot_params.sys_desc_table))
34                      len = sizeof(boot_params.sys_desc_table);
35
36              copy_from_fs(&boot_params.sys_desc_table, oreg.bx, len);
37              return 0;
38}
```

在“BIOS 系统服务 —— 杂项系统服务”一篇博文中
http://blog.csdn.net/yunsongice/archive/2010/10/08/5927585.aspx
我们知道，BIOS 的 15 号服务程序用于读取系统环境，这里的 query_mca()函数中执行 25 行代码后，oreg 中的寄存器就有了博文中“配置表的地址（ES:BX）”，再通过 30 行将 fs 寄存器的值设为 es，即配置表地址的基地址，内部变量 len 为配置表地址的偏移量。最后将系统环境配置表的内容拷贝到 boot_params 的 sys_desc_table 中。

3.3.8 填充 IST 信息

在 main 函数里接下来一个过程是 query_ist 函数，位于 main 函数同一个文件中：

```
71/*
72 * Get Intel SpeedStep (IST) information.
73 */
74static void query_ist(void)
75{
76              struct biosregs ireg, oreg;
77
78              /* Some older BIOSes apparently crash on this call, so filter
79                   it from machines too old to have SpeedStep at all. */
80              if (cpu.level < 6)
81                      return;
82
83              initregs(&ireg);
84              ireg.ax    = 0xe980;            /* IST Support */
85              ireg.edx = 0x47534943;     /* Request value */
86              intcall(0x15, &ireg, &oreg);
87
88              boot_params.ist_info.signature    = oreg.eax;
89              boot_params.ist_info.command      = oreg.ebx;
90              boot_params.ist_info.event        = oreg.ecx;
91              boot_params.ist_info.perf_level = oreg.edx;
92}
```

该函数同样是利用第 15 号 BIOS 服务程序，得到 Intel 的 IST 信息。分别保存在 boot_params 的 ist_info 的四个字段中。main.c 的随后两个函数 query_apm_bios()和 query_edd()涉及到 APM

和 EDD 的内容，执行过程也与 query_ist 差不多，分别填充了的 boot_params 的 apm_bios_info 字段、edd_mbr_sig_buf_entries 字段和 eddbuf_entries 字段，所以我们也就不细讲了。

3.3.9 设置 Video 模式

接下来 main 将执行进入保护模式前最重要的一个步骤，设置视频模式，调用 set_video()函数，来自 linux/arch/x86/boot/video.c：

```
315void set_video(void)
316{
317             u16 mode = boot_params.hdr.vid_mode;
318
319             RESET_HEAP();
320
321             store_mode_params();
322             save_screen();
323             probe_cards(0);
324
325             for (;;) {
326                     if (mode == ASK_VGA)
327                             mode = mode_menu();
328
329                     if (!set_mode(mode))
330                             break;
331
332                     printf("Undefined video mode number: %x\n", mode);
333                     mode = ASK_VGA;
334             }
335             boot_params.hdr.vid_mode = mode;
336             vesa_store_edid();
337             store_mode_params();
338
339             if (do_restore)
340                     restore_screen();
341}
```

317 行，根据传过来的 hdr 参数得到视频模式，存储到内部变量 mode 中。我们看到 header.S 中的 vid_mode 值是 SVGA_MODE。这时候要从新设置一下堆的位置，把它设定到_end 处。

为什么要调整一下堆的位置，我也搞不清楚，也许是跟视频配置有着千丝万缕的关系吧，有兴趣的同志可以去研究一下。随后，调用 store_mode_params()来设置 boot_params 的 screen_info 字段：

```
53/*
54 * Store the video mode parameters for later usage by the kernel.
55 * This is done by asking the BIOS except for the rows/columns
56 * parameters in the default 80x25 mode -- these are set directly,
57 * because some very obscure BIOSes supply insane values.
58 */
59static void store_mode_params(void)
60{
61          u16 font_size;
62          int x, y;
63
64          /* For graphics mode, it is up to the mode-setting driver
65               (currently only video-vesa.c) to store the parameters */
66          if (graphic_mode)
67                    return;
68
69          store_cursor_position();
70          store_video_mode();
71
72          if (boot_params.screen_info.orig_video_mode == 0x07) {
73                    /* MDA, HGC, or VGA in monochrome mode */
74                    video_segment = 0xb000;
75          } else {
76                    /* CGA, EGA, VGA and so forth */
77                    video_segment = 0xb800;
78          }
79
80          set_fs(0);
81          font_size = rdfs16(0x485); /* Font size, BIOS area */
82          boot_params.screen_info.orig_video_points = font_size;
83
84          x = rdfs16(0x44a);
85          y = (adapter == ADAPTER_CGA) ? 25 : rdfs8(0x484)+1;
86
87          if (force_x)
88                    x = force_x;
89          if (force_y)
90                    y = force_y;
91
92          boot_params.screen_info.orig_video_cols  = x;
93          boot_params.screen_info.orig_video_lines = y;
94}
```

store_mode_params()函数主要是利用 BIOS 显示服务程序对视频显示进行设置。有关 BIOS

显示服务程序请参考有关博文“ BIOS 系统服务 —— 显示服务 ”
http://blog.csdn.net/yunsongice/archive/2010/10/08/5927536.aspx

首先，我们看到 69 行 store_cursor_position()函数：

```
20static void store_cursor_position(void)
21{
22          struct biosregs ireg, oreg;
23
24          initregs(&ireg);
25          ireg.ah = 0x03;
26          intcall(0x10, &ireg, &oreg);
27
28          boot_params.screen_info.orig_x = oreg.dl;
29          boot_params.screen_info.orig_y = oreg.dh;
30
31          if (oreg.ch & 0x20)
32                    boot_params.screen_info.flags |= VIDEO_FLAGS_NOCURSOR;
33
34          if ((oreg.ch & 0x1f) > (oreg.cl & 0x1f))
35                    boot_params.screen_info.flags |= VIDEO_FLAGS_NOCURSOR;
36}
```

第 25 行，AH 为 03 表示在文本坐标下，读取光标各种信息功能。出口参数 oreg 为：CH＝光标的起始行；CL＝光标的终止行；DH＝行(Y 坐标)；DL＝列(X 坐标)。那么将 DL 和 DH 的值分别存储到 boot_params 参数的 screen_info.orig_x 字段和 screen_info.orig_y 中，表示当前光标的位置，并根据结果修改 screen_info 的 flags。

回到 store_mode_params()函数中，接下来执行 store_video_mode()函数

```
38static void store_video_mode(void)
39{
40          struct biosregs ireg, oreg;
41
42          /* N.B.: the saving of the video page here is a bit silly,
43               since we pretty much assume page 0 everywhere. */
44          initregs(&ireg);
45          ireg.ah = 0x0f;
46          intcall(0x10, &ireg, &oreg);
47
48          /* Not all BIOSes are clean with respect to the top bit */
49          boot_params.screen_info.orig_video_mode = oreg.al & 0x7f;
50          boot_params.screen_info.orig_video_page = oreg.bh;
51}
```

第 45 行，AH 为 0f 表示读取显示器模式功能，出口参数 oreg 中，ah 为屏幕字符的列数；al 为显示模式；bh＝页码。随后将这些信息分别存储到 screen_info 的 orig_video_mode 和 orig_video_page 中，表示当前的显示属性。

回到 store_mode_params()函数中，第 72 到 78 行，没有问题，根据刚刚得到的显示模式，设置全局变量 video_segment 的值。同样，这个全局变量跟_end 一样，是编译的时候生成的。第 80、81 行，从 BIOS 中获取字体大小，保存到内部变量 font_size 中。注意，为什么说是从 BIOS 中，因为调用 rdfs16 内联函数：

```
static inline u16 rdfs16(addr_t addr)
{
    u16 v;
    asm volatile("movw %%fs:%1,%0" : "=r" (v) : "m" (*(u16 *)addr));
    return v;
}
```

其参数 addr 的值为 0x485，对照内核映像解压缩后的内存布局，可以看到这个地址是在 BIOS 的区域内。82 行的代码就保留这个值。

84 到 93 行，仍然是通过直接读 BIOS 的方式得到显示属性的行数和列数，并填充 screen_info 的 orig_video_cols 和 orig_video_lines 字段。注意这里有个编译选项，如果在编译的时候设定了 force_x 或 force_y，就把显示属性的行数和列数强制设为这个值。

store_mode_params()函数结束了，基本的显示参数都存储到 boot_params 的 screen_info 字段中了，回到 set_video 函数中，第 322 行，save_screen()函数，用于将当前屏幕的内容存储到指定的内存空间中：

```
230/* Save screen content to the heap */
231static struct saved_screen {
232             int x, y;
233             int curx, cury;
234             u16 *data;
235} saved;
236
237static void save_screen(void)
238{
239             /* Should be called after store_mode_params() */
240             saved.x = boot_params.screen_info.orig_video_cols;
241             saved.y = boot_params.screen_info.orig_video_lines;
242             saved.curx = boot_params.screen_info.orig_x;
243             saved.cury = boot_params.screen_info.orig_y;
244
245             if (!heap_free(saved.x*saved.y*sizeof(u16)+512))
246                          return;             /* Not enough heap to save the screen */
```

```
247
248             saved.data = GET_HEAP(u16, saved.x*saved.y);
249
250             set_fs(video_segment);
251             copy_from_fs(saved.data, 0, saved.x*saved.y*sizeof(u16));
252}
```

一路走来的朋友们对这种函数应该熟悉了吧，240 行道 248 行设置全局变量 saved。有个地方不看不懂，就是 GET_HEAP：

```
#define RESET_HEAP() ((void *)( HEAP = _end ))

static inline char *__get_heap(size_t s, size_t a, size_t n)
{
        char *tmp;

        HEAP = (char *)(((size_t)HEAP+(a-1)) & ~(a-1));
        tmp = HEAP;
        HEAP += s*n;
        return tmp;
}
#define GET_HEAP(type, n) \
        ((type *)__get_heap(sizeof(type),__alignof__(type),(n)))
```

像这么复杂的代码在 Linux 中比比皆是。虽然我们看不懂，但是可以猜。不过就像当年我们学英语做阅读题一样，必须有根据地猜，不能瞎猜。首先，还记得先前 set_video 的 319 行有个 RESET_HEAP，就是把 HEAP 全局变量设置成了_end。_end 这个东西我们熟悉，是 vmlinuz 实模式部分的结束位置，也是堆栈段的开始位置。调用 GET_HEAP 宏之前，HEAP 指向的_end。而调用 GET_HEAP(u16, saved.x*saved.y)，对应__get_heap 返回后，HEAP 大约就指向了_end 后再经过 2*saved.x*saved.y*2 字节偏移的位置（我们假设 sizeof(u16)为 2）。

因此，我猜测，屏幕假设有 saved.x*saved.y 个像素，那么每个像素需要 2*2 即 4 个字节的内存来存储，248 行使用 GET_HEAP 宏就是分配这么个堆来存储这些东西，并且 saved 得 data 字段指向这个堆的首地址。顺便提一下，学过操作系统的都知道，堆跟栈是差不多的概念，唯一不同的是栈是向上增长的，堆跟栈正好相反，向下增长。所以这个堆用在这里，给这个多分配了这么多空间，就不会与其他内存中的内容发生冲突。

save_screen()函数的最后一行，调用一个 copy_from_fs 函数将当前屏幕的每个像素的内容拷贝到这个堆中。注意，video_segment 是 grub 传递过来的参数，记录着内存中显示器设备的内容。

回到 set_video 中，接下来 315 行调用 probe_cards(0)，用来扫描显卡，来自 linux/arch/x86/boot/video-mode.c：

```
/* Probe the video drivers and have them generate their mode lists. */
  33void probe_cards(int unsafe)
  34{
  35            struct card_info *card;
  36            static u8 probed[2];
  37
  38            if (probed[unsafe])
  39                        return;
  40
  41            probed[unsafe] = 1;
  42
  43            for (card = video_cards; card < video_cards_end; card++) {
  44                        if (card->unsafe == unsafe) {
  45                                    if (card->probe)
  46                                                card->nmodes = card->probe();
  47                                    else
  48                                                card->nmodes = 0;
  49                        }
  50            }
  51}
```

传递给 probe_cards 的参数为 0，所以 unsafe=0。随后扫描整个显卡列表。video_cards 和 video_cards_end 都是 grub 传递过来的显卡列表。

回到 set_video 中，325 行进入一个循环。根据 bootloader 传进来的 hdr 的 vid_mode，如果 mode 为 ASK_VGA，就进行一些交互式的工作。grub 传进来的 vid_mode 不是 ASK_VGA，所以我们不去分析 mode_menu()函数了，感兴趣的，或者想了解 BIOS 编程的可以去分析一下它，这个函数是交互式 BIOS 程序的编程典范。

直接跳出循环，来到 336 行。vesa_store_edid()函数，是对 EDID 的设置。EDID 是一种 VESA 标准数据格式，其中包含有关监视器及其性能的参数，包括供应商信息、最大图像大小、颜色设置、厂商预设置、频率范围的限制以及显示器名和序列号的字符串。我们不去分析它了。

337 行再次执行 store_mode_params()来设置 boot_params 的 screen_info 字段，最后根据是否进入了 mode_menu 设置了 do_restore 来恢复刚刚被保存的 screen_info 信息，它跟我们刚刚介绍过的 save_screen 正好相反：

```
 254static void restore_screen(void)
 255{
 256            /* Should be called after store_mode_params() */
 257            int xs = boot_params.screen_info.orig_video_cols;
 258            int ys = boot_params.screen_info.orig_video_lines;
 259            int y;
 260            addr_t dst = 0;
```

```
261         u16 *src = saved.data;
262         struct biosregs ireg;
263
264         if (graphic_mode)
265                 return;         /* Can't restore onto a graphic mode */
266
267         if (!src)
268                 return;         /* No saved screen contents */
269
270         /* Restore screen contents */
271
272         set_fs(video_segment);
273         for (y = 0; y < ys; y++) {
274                 int npad;
275
276                 if (y < saved.y) {
277                         int copy = (xs < saved.x) ? xs : saved.x;
278                         copy_to_fs(dst, src, copy*sizeof(u16));
279                         dst += copy*sizeof(u16);
280                         src += saved.x;
281                         npad = (xs < saved.x) ? 0 : xs-saved.x;
282                 } else {
283                         npad = xs;
284                 }
285
286                 /* Writes "npad" blank characters to
287                    video_segment:dst and advances dst */
288                 asm volatile("pushw %%es ; "
289                              "movw %2,%%es ; "
290                              "shrw %%cx ; "
291                              "jnc 1f ; "
292                              "stosw \n\t"
293                              "1: rep;stosl ; "
294                              "popw %%es"
295                              : "+D" (dst), "+c" (npad)
296                              : "bdS" (video_segment),
297                                "a" (0x07200720));
298         }
299
300         /* Restore cursor position */
301         if (saved.curx >= xs)
302                 saved.curx = xs-1;
303         if (saved.cury >= ys)
304                 saved.cury = ys-1;
```

```
305
306             initregs(&ireg);
307             ireg.ah = 0x02;             /* Set cursor position */
308             ireg.dh = saved.cury;
309             ireg.dl = saved.curx;
310             intcall(0x10, &ireg, NULL);
311
312             store_cursor_position();
313}
```

当然，针对我们的 grub，也是不会走这一步的，所以上面的代码也就只是摆设而已了。

3.4 实模式代码 go_to_proteced_mode 函数

main 函数走到 go_to_protected_mode，就是将告别实模式环境，进入保护模式了。保护模式（Protected Mode，或有时简写为 pmode）是一种 80286 系列和之后的 x86 兼容 CPU 操作模式。保护模式有一些新的特色，设计用来增强多功能和系统稳定度，像是内存保护，分页系统，以及硬件支援的虚拟内存。大部分的现今 x86 操作系统都在保护模式下运行，包含 Linux、FreeBSD、以及微软 Windows 2.0 和之后版本。

注意，这里还是在实模式下，只不过是进入保护模式相关的代码。有关保护模式编程的详细介绍，请参考博客“保护模式编程”：
http://blog.csdn.net/yunsongice/archive/2010/10/06/5924409.aspx

看到 main 函数中的最后一行代码，注释上写的是激活保护模式，调用 go_to_protected_mode()函数，在 linux/arch/x86/boot/pm.c 中：

```
101/*
102 * Actual invocation sequence
103 */
104void 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}
```

3.4.1 禁止可屏蔽和不可屏蔽中断

107 行，调用 realmode_switch_hook()函数。这个函数位于同一个文件中，内容如下：

```
 18/*
 19 * Invoke the realmode switch hook if present; otherwise
 20 * disable all interrupts.
 21 */
 22static void realmode_switch_hook(void)
 23{
 24             if (boot_params.hdr.realmode_swtch) {
 25                          asm volatile("lcallw *%0"
 26                                       : : "m" (boot_params.hdr.realmode_swtch)
 27                                       : "eax", "ebx", "ecx", "edx");
 28             } else {
 29                          asm volatile("cli");
 30                          outb(0x80, 0x70); /* Disable NMI */
 31                          io_delay();
 32             }
 33}
```

注释中写得很清楚了，这个函数的目的在于如果 hdr 参数的 realmode_swtch 字段被设置，则直接跳到对应的子程序中，一些 bootloader 程序用该子程序来抓住实模式下最后的执行代码的机会。否则关中断。我们看到 header.S 的第 122 行，realmode_swtch 字段为 0，即没有设置，所以函数直接到 29 行，执行 cli 指令关中断。有关 x86 相关的指令请参考博客“ Intel 8086/8088 指令系统 ”
http://blog.csdn.net/yunsongice/archive/2010/10/04/5920438.aspx

30 行是全内核代码中第一次出现跟 I/O 低级函数，在博文“ Linux I/O 体系结构 ”有这些低级函数的介绍，这里讲这些函数的原型：

```
static inline void outb(u8 v, u16 port)
{
	asm volatile("outb %0,%1" : : "a" (v), "dN" (port));
}
```

其实这些低级 I/O 函数都在 arch/x86/boot/Boot.h 中定义，以后全代码的所有对这些函数的扩展，最后也都会走到这里，我们就只看 outb 的源代码，其他的类似。outb()函数就是对 outb 指令进行封装，将参数 v 的内容传输到参数 port 对应的端口中。那么 30 行的 outb(0x80, 0x70)意思是将 0x70 端口对应接口内部寄存器的值置为 0x80。

注意，Linux 启动程序运行到此，还没有建立起内核 I/O 子系统，所有这里的 I/O 端口跟我们平常在/proc/ioports 文件中查的 I/O 端口是不一样的。系统上电以后，BIOS 程序把 0x70 号端口置为的不可屏蔽中断寄存器，NMI。

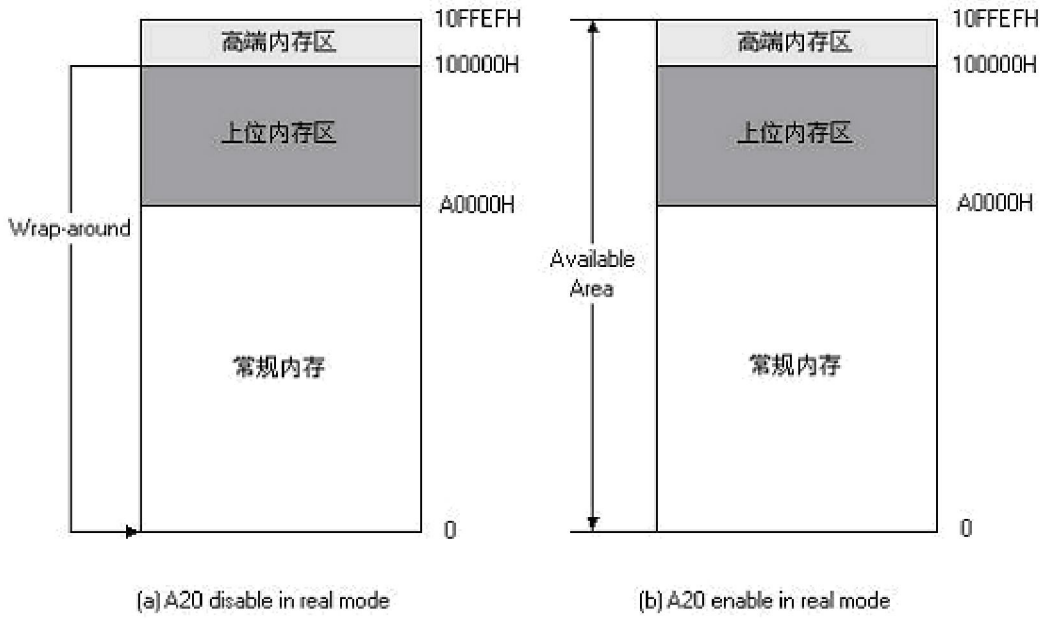
NMI (Non Maskable Interrupt)——不可屏蔽中断(即 CPU 不能屏蔽)，无论状态寄存器中 IF 位的状态如何，CPU 收到有效的 NMI 必须进行响应。NMI 中断类型号固定为 2，它在被响应时无中断响应周期。不可屏蔽中断通常用于故障处理（如：协处理器运算出错，存储器校验出错，I/O 通道校验出错等）。这里执行 30 行代码就是紧接着把这个 NMI 也给禁止掉。随后 31 行 io_delay()函数，是对 outb()函数的扩展：

```
static inline void io_delay(void)
{
	const u16 DELAY_PORT = 0x80;
	asm volatile("outb %%al,%0" : : "dN" (DELAY_PORT));
}
```

是把 0x80 端口对应接口内部寄存器的值也置为 0x80。我们知道，0x80 其实是个空端口，所以这个 io_delay 函数的意义仅仅是对 I/O 端口执行一个空操作，以达到对 CPU 的 I/O 访问周期的延时工作。

3.4.2 打开 A20 地址线

回到 go_to_protected_mode()的 110 行，调用一个 enable_a20()函数。这里又用到“ PC 汇编及 BIOS 编程 ”的知识了。PC 及其兼容机的第 21 根地址线(A20)较特殊，这就是“ Intel 80286 工作模式 ” http://blog.csdn.net/yunsongice/archive/2010/10/04/5920447.aspx 提到的 PC 中安排的一个“ 门 ”控制该地址线是否有效。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，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。

那么，如何打开这个“门”呢？这就是 enable_a20()函数干的事情，来自 linux/arch/x86/boot/a20.c：

```
/*
 125 * Actual routine to enable A20; return 0 on ok, -1 on failure
 126 */
 127
 128#define A20_ENABLE_LOOPS 255        /* Number of times to try */
 129
 130int 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}
```

enable_a20()函数返回 0，则直接到 go_to_protected_mode 的 116 行，否则重启。下面重点来分析一下 enable_a20()函数。函数首先进行一个最大循环为 255 次的有限循环中。然后不断调用探测函数 a20_test_short()和 a20_test_long()：

```
/* Returns nonzero if the A20 line is enabled.    The memory address
   50      used as a test is the int $0x80 vector, which should be safe. */
   51
   52#define A20_TEST_ADDR      (4*0x80)
   53#define A20_TEST_SHORT     32
   54#define A20_TEST_LONG      2097152 /* 2^21 */
```

```
55
56static int a20_test(int loops)
57{
58              int ok = 0;
59              int saved, ctr;
60
61              set_fs(0x0000);
62              set_gs(0xffff);
63
64              saved = ctr = rdfs32(A20_TEST_ADDR);
65
66              while (loops--) {
67                              wrfs32(++ctr, A20_TEST_ADDR);
68                              io_delay();         /* Serialize and make delay constant */
69                              ok = rdgs32(A20_TEST_ADDR+0x10) ^ ctr;
70                              if (ok)
71                                              break;
72              }
73
74              wrfs32(saved, A20_TEST_ADDR);
75              return ok;
76}
77
78/* Quick test to see if A20 is already enabled */
79static int a20_test_short(void)
80{
81              return a20_test(A20_TEST_SHORT);
82}
83
84/* Longer test that actually waits for A20 to come on line; this
85     is useful when dealing with the KBC or other slow external circuitry. */
86static int a20_test_long(void)
87{
88              return a20_test(A20_TEST_LONG);
89}
```

一路走来的同志们看这些代码应该不难，就是用 fs 配合 gs 来测试 A20 Gate 是否被打开。我们在之前已经提到，如果 A20 Gate 被打开了，则在实模式下，程序员可以直接访问 100000H~10FFEFH 之间的内存，如果 A20 Gate 被禁止，则在实模式下，若程序员访问 100000H~10FFEFH 之间的内存，则会被硬件自动转换为 0H~0FFEFH 之间的内存，所以我 a20_test 函数就是利用这个差异来检测 A20 Gate 是否被打开。

首先，61、62 行，把 fs 和 gs 的值分别设置为 0x0000 和 0xffff。然后 64 行调用 rdfs32 函数得到 0000: 4*0x80 的内容并存放到 32 位的临时变量 saved 和 ctr 中。4*0x80=0x200，所以

saved 和 ctr 中存放的是内存地址 0x200 处的值。对应内核映像解压缩后的内存布局，我们知道这个地址属于 BIOS 的一些数据存放的区域，虽然我不知道这个地址到底是存的什么数据，但是可以肯定的是，这个数据可以随意修改，来做我们的 A20 测试的。

67 行进入循环，使 ctr 加 1，具体等于多少不知道，把这个 32 位的值写入 0000: 4*0x80 对应的内存单元中。69 行，“^”是按位异或运算符，即如果 ffff: 4*0x80+0x10 内存单元存放的值与 ctr 相等，则说明有可能刚刚写入的 ctr 其实是写入的 0000: 4*0x80 内存单元。注意，ffff: 4*0x80+0x10 换算实模式地址就是 ffff0+4*0x80+0x10=0x100200，不过也不排除偶然的情况，这两个内存单元相等。所以继续循环，修改一下 ctr 的值，多试几次。

74 行，跳出循环后，你刚才给人家 0x200 的地址对应的内存修改了数据，得给人家改回去啊，最后返回 ok。如果执行了 loops 次这个 ok 都是 0，就说明 A20 肯定没有打开。

回到 enable_a20 函数，如果 bootloader 没有已经关闭了 A20 的话（grub 是肯定关闭了的），也就是第一个 a20_test_short()没有成功，试试 142 行 enable_a20_bios()：

```
91static void enable_a20_bios(void)
92{
93              struct biosregs ireg;
94
95              initregs(&ireg);
96              ireg.ax = 0x2401;
97              intcall(0x15, &ireg, NULL);
98}
```

嗯，没问题，调用 BIOS 的 15 号服务程序，打开 A20。如果没成功，执行 147 行代码：

```
18#define MAX_8042_LOOPS    100000
19#define MAX_8042_FF         32
20
21static int empty_8042(void)
22{
23              u8 status;
24              int loops = MAX_8042_LOOPS;
25              int ffs      = MAX_8042_FF;
26
27              while (loops--) {
28                              io_delay();
29
30                              status = inb(0x64);
31                              if (status == 0xff) {
32                                              /* FF is a plausible, but very unlikely status */
33                                              if (!--ffs)
34                                                              return -1; /* Assume no KBC present */
```

```
35                      }
36                      if (status & 1) {
37                              /* Read and discard input data */
38                              io_delay();
39                              (void)inb(0x60);
40                      } else if (!(status & 2)) {
41                              /* Buffers empty, finished! */
42                              return 0;
43                      }
44              }
45
46              return -1;
47}
```

0x64 号端口号对应的是键盘控制器（keyboard controller, KBC）的状态寄存器，首先 30 行获得该控制器的状态值，保存到内部变量 status 中。status 不能为 0xff，否则出错返回。其次，status 的最低位 D0 如果被设置，则说明键盘缓存中还有数据，则通过 inb(0x60)从键盘缓存对应的端口读出来，并且置空。

介绍一下键盘控制器：主板的一个芯片。通过 LPC 总线和南桥相连。一般作用：键盘控制、LCD 明暗度的调节、低级电源的管理、风扇、蓝牙等一些小功能。

注意这个逻辑关系，如果键盘缓存中没有内容，并且 status 的次低位 D1 不为 1，则说明键盘缓存是空的，这时候 empty_8042 返回 0，enable_a20 再一次检测一下 BIOS 然后来到了 153 行，调用 enable_a20_kbc()函数，利用键盘控制器来尝试打开 A20：

```
100static void enable_a20_kbc(void)
 101{
 102            empty_8042();
 103
 104            outb(0xd1, 0x64);           /* Command write */
 105            empty_8042();
 106
 107            outb(0xdf, 0x60);           /* A20 on */
 108            empty_8042();
 109
 110            outb(0xff, 0x64);           /* Null command, but UHCI wants it */
 111            empty_8042();
 112}
```

如果还打不开 A20，没辙了，就来到 enable_a20()的 159 行进行最后的努力，调用 enable_a20_fast()函数。

```
 114static void enable_a20_fast(void)
```

```
115{
116             u8 port_a;
117
118             port_a = inb(0x92);         /* Configuration port A */
119             port_a |=    0x02;            /* Enable A20 */
120             port_a &= ~0x01;              /* Do not reset machine */
121             outb(port_a, 0x92);
122}
```

这个函数是个熟面孔，在我们的博文“保护模式编程实例”
http://blog.csdn.net/yunsongice/archive/2010/10/08/5927137.aspx
中，那个 EnableA20 汇编宏就是干的这么一个事情，将 0x92 端口的 D1 位置位。

注意这个 enable_a20_fast，基于 x86 的主板都提供 0x92 端口来作为一个主板控制寄存器，所以，如果这个函数执行后都还是打不开 A20 的话，那也就没辙了，只好放弃。

3.4.3 安装临时全局描述符表

回到 go_to_protected_mode()函数中，接下来 116 和 119 行的两个函数比较简单，我们快速过一下：

```
35/*
36 * Disable all interrupts at the legacy PIC.
37 */
38static void mask_all_interrupts(void)
39{
40             outb(0xff, 0xa1);           /* Mask all interrupts on the secondary PIC */
41             io_delay();
42             outb(0xfb, 0x21);            /* Mask all but cascade on the primary PIC */
43             io_delay();
44}
45
46/*
47 * Reset IGNNE# if asserted in the FPU.
48 */
49static void reset_coprocessor(void)
50{
51             outb(0, 0xf0);
52             io_delay();
53             outb(0, 0xf1);
54             io_delay();
55}
```

这两个函数很简单，就是对浮点运算协处理器 FPU 和 PIC 的端口进行清零和复位工作。

回到 go_to_protected_mode()函数中继续走，122 行 setup_idt()函数，

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

就是对中断描述符表进行初始化。初始化阶段的中断描述符初始化过程是非常简单的，设置一个全局的空变量 null_idt，它的类型是 gdt_ptr：

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

然后再执行 lidtl 指令，将中断描述符表首地址存入 IDTR 寄存器中。

好啦，到 123 行 setup_gdt()函数，就个函数就重要了：

```
66static 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}
```

第 70 行，建立一个 boot_gdt[]数组，每个成员是 64 位，符合我们对全局描述符的定义。对全局描述符的相信介绍，请查看博客“ Intel 80386 的保护模式 ”
http://blog.csdn.net/yunsongice/archive/2010/10/04/5921882.aspx

```
#define GDT_ENTRY_BOOT_CS       2
#define GDT_ENTRY_BOOT_DS       (GDT_ENTRY_BOOT_CS + 1)
#define GDT_ENTRY_BOOT_TSS      (GDT_ENTRY_BOOT_CS + 2)
```

这三个宏表示 boot_gdt[]数组的下标，即是从 2 开始的。为什么从 2 开始，注释里说了，Intel 要求按 16 字节对齐，即 2 个 u64。GDT_ENTRY 宏定义如下：

```
#define GDT_ENTRY(flags, base, limit)                   \
        ((((base)  & 0xff000000ULL) << (56-24)) | \
         (((flags) & 0x0000f0ffULL) << 40) |         \
         (((limit) & 0x000f0000ULL) << (48-16)) | \
         (((base)  & 0x00ffffffULL) << 16) |         \
         (((limit) & 0x0000ffffULL)))
```

经过宏 GDT_ENTRY 运算后，flags、base 和 limit 的位置就如“ Intel 80386 的保护模式 ”第一个图那样，base 是基地址，limit 是段限，flags 是除基地址和段限后的内容。

我们看到，boot_gdt[]数组下标 2 是代表的内核代码段，下标 3 代表内核数据段，下标 4 代表任务状态段。注意，这三个段都只是在初始化阶段，vmlinuz 被 grub 加载后，为了访问其 1MB 以后的代码所需要保护模式的段。CS 和 DS 的基地址都是 0，段限都是 0xfffff，说明都能访问到 4GB 的空间。TSS 的基地址是 4096，段限是 103，注释里说得很清楚，整个系统初始化阶段都不会使用 TSS 段。这是显然的，没有人会在初始化的时候做任务切换。

第 84 行，同样也是建立一个 gdt_ptr 的静态变量，存放刚刚建好的全局描述符表 boot_gdt[]的长度和地址。随后通过 89 行的 lgdtl 指令将 gdt 的值装入 GDTR 寄存器中。

3.4.4 第一次启动保护模式

回到 go_to_protected_mode()函数中，最后一行调用：
protected_mode_jump(boot_params.hdr.code32_start,(u32)&boot_params + (ds() << 4));

这个函数接受两个参数，第一个参数是 grub 传过来的保护模式的第一条代码。这个值就是前面说到了的，0x100000。后面这个就是给内核传递的参数，由于切换到了保护模式，所以要给出参数的线性地址，而不是有效地址，ds()函数就是 ds 寄存器的值。

来看看这个函数，它是由汇编语言实现的，代码在 arch/x86/boot/pmjump.S 中：

```
23/*
24 * void protected_mode_jump(u32 entrypoint, u32 bootparams);
25 */
26GLOBAL(protected_mode_jump)
27              movl    %edx, %esi              # Pointer to boot_params table
28
29              xorl    %ebx, %ebx
30              movw    %cs, %bx
31              shll    $4, %ebx
32              addl    %ebx, 2f
33              jmp     1f                      # Short jump to serialize on 386/486
341:
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
42
43              # Transition to 32-bit mode
44              .byte   0x66, 0xea              # ljmpl opcode
452:            .long   in_pm32                 # offset
46              .word   __BOOT_CS               # segment
47ENDPROC(protected_mode_jump)
```

要看懂上面的代码，需要用到一个 C 语言背景知识。在 C 语言中，假设咱们有这样的一个函数：

```
int function(int a, int b){
    return a+b;
}
```

调用时只要用 result = function(1, 2)那样的方法就能够应用那个参数。但是，当高级语言被编译成电脑能够识别的机器码时，有一个疑问目就出现了：在 CPU 中，计算机没有办法清楚一个参数调用需求多少个、什么样的参数，也没有硬件能够保存这一些参数。也就是说，编译器不清楚怎么给那个参数传递参数，传递参数的任务必需由参数调用者和参数本身来协调。为此，使用栈来支持参数传递。

学过《数据结构》这么课程的童鞋都知道，栈是一种先进后出的 Data 框架，栈有一个存储区、一个栈顶指针。参数调用时，调用者依次把参数压栈，然后调用参数，参数被调用以后，在堆栈中取得 Data，并停止计算。参数计算结束以后，或者调用者、或者参数本身改正堆栈，使堆栈还原原装。

在参数传递中，有两个很重要的东西目必需得到明确说明：

- 当参数个数多于 1 个时，按照什么顺序把参数压入堆栈
- 参数调用后，由谁来把堆栈还原原装

在编译器中，通过参数调用约定来说明这两个疑问，大多数编译器日常的调用约定有：stdcall、cdecl、fastcall、thiscall、naked call。

过去，stdcall 是 gcc 的默认方式，不过如今要显式地约定声明了：

```
int __stdcall function(int a, int b)
```

stdcall 的调用约定意味着：1）参数从右向左压入堆栈，2）参数自身改正堆栈 3）参数名自动加前导的下划线，后面紧跟一个@符号，其后紧跟着参数的尺寸。

以上述那个参数为例，参数 a 首先被压栈，然后是参数 b，参数调用 function(1, 2)。调用处汇编语言将变成：

```
push 2  第二个参数入栈
push 1  第一个参数入栈
call function  调用参数，留意此时自动把 cs：eip 入栈
```

被调用参数_function 处：

```
push ebp  保存 ebp 寄存器，该寄存器将用来保存堆栈的栈顶指针，能够在出参数时还原
mov ebp,esp  保存堆栈指针
mov eax,[ebp+8H]  堆栈中 ebp 指向位置之前依次保存有 ebp、cs:eip、a、b，
                  所以 ebp +8 指向 a
add eax,[ebp + 0CH]  堆栈中 ebp + 12 处保存了 b
mov esp,ebp  还原 esp
pop ebp
ret 8
```

而在编译时，那个参数的姓名被中英对译成_function@8。从参数调用看，2 和 1 依次被 push 进堆栈，而在参数中又经过相对于 ebp（即刚进参数时的堆栈指针）的偏移量存取参数。参数结束后，ret 8 意思是清理 8 个字节的堆栈，参数本人还原了堆栈。

fastcall 调用约定和 stdcall 类似，它意味着：

参数的第一个和第二个 DWORD 参数（或者尺寸更小的）分别经过 eax 和 edx 传递，更多参数则经过从右向左的顺序压栈。过去的内核还有一个 Gcc 属性宏定义：

```
#define FASTCALL __attribute__((regparm(3)))
```

不过目前我看的 2.6.34.1 版本的内核的 x86 体系以及没有这个宏定义了，由此我推断，新的内核依赖的 gcc 新版本为了提升针对 x86 的性能，已经不再使用 stdcall 或 cdecl 方案了，所有函数都是 fastcall。为什么能提升了性能呢，你想想啊，x86 中 eax 和 edx 用于存放传入参数，作为高水平内核级程序员，函数的参数不应该超过 2 个。那么有人就会反过来问：超过 2 个的那些参数又何去何从？我的回答是：传入参数如果超过 2 个，多余的参数还是被存放到局部栈中，表面上没什么不好，但是不管是系统调用，还是系统异常都会引起用户空间陷

入内核空间。我们知道，系统空间的权限级是 0，用户空间的权限级为 3，系统调用从权限级为 3 的用户空间陷到权限级为 0 的内核空间，必然引起堆栈切换，linux 系统将从全局任务状态栈 TSS 中找到一个合适的内核栈信息保存覆盖当前 SP、SS 两个寄存器的内容，以完成堆栈切换，此时处于内核空间所看到的栈已不是用户空间那个栈，所以在调用的时候压入用户栈的数据就在陷入内核的那个瞬间，被滞留在用户空间栈，内核根本不知道它的存在了，所以作为安全考虑或者作为高水平程序员的切身修养出发，都不应该向系统调用级函数传入过多的参数。

一边分析内核，一边学习大量的计算机与编程知识，这就是内核给我们带来的巨大乐趣，何乐而不为呢？我们继续。刚才介绍了带参数函数调用的参数传递的 C 语言方面的知识，现在我们知道了，来到 protected_mode_jump 后 eax 存放的是 code32_start，其值是 header.S 中的 152 行定义的 hdr 传递过来的 0x100000；edx 存放的是 bootparams 参数 32 位的地址。

27 行：

```
movl    %edx, %esi          # Pointer to boot_params table
```

执行此指令后 esi 寄存器存放着 boot_params 结构的首地址，随后 29~32 行：

```
xorl  %ebx, %ebx
movw    %cs, %bx
shll  $4, %ebx
addl %ebx, 2f
jmp  1f                 # Short jump to serialize on 386/486
```

执行上述指令后，ebx 寄存器存放的就是 2f 程序段对应的地址值。为了学习，这里不得不多说几句。cs 是 16 位的，而我们马上要进入保护模式了，不管是代码寻址还是数据寻址，都是 32 位的了，此时此刻的这个 cs 所指向的段没有什么意义了，所以上面的动作就是将进入保护模式代码段了之前的 cs:eip 的值保存在 ebx 中，保存来干嘛呢，马上会用到。

继续走：34~44 行：

```
1:

    movw    $__BOOT_DS, %cx
    movw    $__BOOT_TSS, %di

    movl    %cr0, %edx
    orb   $X86_CR0_PE, %dl      # Protected mode
    movl    %edx, %cr0

    # Transition to 32-bit mode
    .byte0x66, 0xea         # ljmpl opcode
```

这段代码就是 protected_mode_jump 的核心，即打开 cr0 的 PE 位，打开保护模式。由于：

```
#define GDT_ENTRY_BOOT_CS       2
#define __BOOT_CS           (GDT_ENTRY_BOOT_CS * 8)
```

```
#define GDT_ENTRY_BOOT_DS       (GDT_ENTRY_BOOT_CS + 1)
#define __BOOT_DS           (GDT_ENTRY_BOOT_DS * 8)

#define GDT_ENTRY_BOOT_TSS      (GDT_ENTRY_BOOT_CS + 2)
#define __BOOT_TSS          (GDT_ENTRY_BOOT_TSS * 8)
```

所以 16 位 cx 的值是（2*8+1）*8 即 136，16 进制为 0x88；di 为（2*8+2）*8 即 144，16 进制为 144；eax 存放的 0x100000，edx 和 esi 存放了 boot_params 的首址；还有一个 ebx 存放这向保护模式切换前 cs:eip 指令地址。所以即将进入保护模式前夕，这几个寄存器的值就是这个样子滴。

在执行完第 40 行代码后，内核从此告别实模式，开始了 x86 的保护模式之旅。注意，从系统启动到此，并不是第一次进入保护模式，前面 bootloader 阶段，grub 已经执行过一下保护模式的命令了，不然怎么会把 vmlinuz 第三部分的代码拷贝到内存 0x100000 之后呢。随后立即跳到 45 行标号 2，开始执行保护模式下的代码：

```
2:  .long   in_pm32         # offset
    .word   __BOOT_CS       # segment
```

在函数 main 的最后，实际上就是已关中断，准备进入保护模式，设置最初始的 gdt,idt 等。当前面执行.byte 0x66, 0xea 指令时，cs 寄存器的值被设置为__BOOT_CS，即代码段选择子，偏移量就是子程序名 in_pm32。至于如何的到对应的物理地址，请查看“ 保护模式编程 ”
http://blog.csdn.net/yunsongice/archive/2010/10/04/5920447.aspx
来看子程序 in_pm32：

```
49          .code32
50          .section ".text32","ax"
51GLOBAL(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
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
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
75
76          jmpl    *%eax                   # Jump to the 32-bit entrypoint
77ENDPROC(in_pm32)
```

53~57 行首先把 ds、es、fs、gs、ss 设置成同样的值，即__BOOT_DS，其值为 0x88。为什么是 0x88 呢？注意，我们分析代码的目的是学习，是研究，不是其他的，所以来研究一下。我们知道，进入保护模式后，段寄存器的值就不再是存放段基址了，而是存放段选择子。

选择子的格式如下：（来自博客“ Intel 80286 工作模式 ”
http://blog.csdn.net/yunsongice/archive/2010/10/04/5920447.aspx）

0x88 正好就是二进制的 10001000，其中 10001 对应选择子的偏移量 D15～D3，即所要访问的描述子在描述子表中的偏移量。RPL 为 00，对应最高特权级（初始化阶段，特权级当然最高），TI 为 0，表示全局描述符。

好，继续研究。那么偏移量为啥要设置成 10001 呢？回顾一下，我们前面是如何安装全局描述符表的。setup_gdt()函数中，我们建立了一个全局描述符表 boot_gdt，然后把这个表的首地址和长度加载到 GDTR 寄存器中。这个表的每个成员是 64 位，即 8 个字节。10001 换算成十进制就是 17，也就是经历了 2 个 8 字节后的偏移位置。我为什么说是 2 个 8 字节，不错，看看那个表的定义，第 3 个 8 字节，正是从 GDT_ENTRY_BOOT_DS 下标开始的。所以，经过这样的一系列指令后，ds、es、fs、gs、ss 这五个寄存器的内容都是指向全局描述符表 boot_gdt 的数据段了。

继续走，60 行对进入保护模式后的堆栈进行调整，将原来的栈顶指针 esp 加上刚才保存在 ebx 的代码段偏移。注意，这个地方注释上写得很清楚了，32 位保护模式有自己的堆栈，这里调整堆栈的目的是供某些黑客娱乐的。

63 行，执行 ltr 指令，目的在于设置 TSS 相关的 TR 寄存器。至于想学习 TSS 内容的朋友，请参考博客“ Intel 80286 工作模式 ”
http://blog.csdn.net/yunsongice/archive/2010/10/04/5920447.aspx。67 到 71 行清空刚才使用的那些通用寄存器。最后 74 行加载 LDTR 寄存器设置局部描述符表。此时 ecx 的值为空，所以这步操作是一个没有意义的操作。

最后一行，76 行执行 jmpl *%eax，开始执行由函数参数方式传递进来的 code32_start，即

0x100000 处的代码，我们著名的“ 文艺复兴时期 ”。前面讲过，在解压缩 vmlinuz 之前，这代码在 arch/x86/boot/compressed/head_32.S 中，入口是 ENTRY(startup_32)。

由于段的基地址为 0（可以参考 go_to_protected_mode()中的 setup_gdt()函数），所以线性地址等于有效地址，因为目前还没有分页，所以线性地址也其实就是物理地址，物理地址 1M 后正是保护模式代码所在地）

4 保护模式下的内核代码

4.1 32 位 x86 保护模式代码

到保护模式的代码了，最先执行的代码就是 arch/x86/boot/compressed/head_32.S 中的 startup_32，对于 bzImage，grub 把它加载到 0x100000 的位置。

首先，来到 34 行 ENTRY(startup_32)，在这里，第一条保护模式的指令开始了。注意，前面的 set_gdt 仅仅是为了进入保护模式后的 ljmp，意义并不大。

```
34ENTRY(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
501:
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
61              call      1f
```

4.1.1 内核解压缩的前期工作

35 行，第一条进入 32 位保护模式后的指令，清除方向标志指令。随后根据 hdr 的 loadflags 的第 7 位是否被设置，我们看到根本没有设置，所以经过 44~49 行设置各个段寄存器。注意 40 行有一个%esi 寄存器，我们在 protected_mode_jump 汇编代码中看到了，它存放着刚才 c 语言代码中设置好了的 boot_params 变量的地址。这个地址在传入 protected_mode_jump 之后变成了 32 位：(u32)&boot_params + (ds() << 4)。这个东西就是所谓 0 号页面的内容。

再一个就是 BP_loadflags，后面会看到很多 BP 开头的常量。这些常量都是编译的时候设置的某些数据结构中一些字段地址偏移，大家可以去看看 arch/x86/kernel/asm-offsets_32.c 的最后几行，BP 开头的常量就是 boot_params 结构的某字段的偏移。BP_loadflags 就是 hdr.loadflags 相对于 boot_params 结构首部的偏移，于是 BP_loadflags+%esi 就是 loadflags 的地址。

再看到 60 行，BP_scratch 是 scratch 字段相对于 boot_params 结构首部的偏移，其实就是让栈顶指向(BP_scratch+4)(%esi)地址，根据注释来讲这是一个“刮刮卡”区域。这个“刮刮卡”是用来干什么的呢？前面 main.c 中没有对它赋值啊。原来，它是用来计算当期内核映像位置的。60 行 lea 指令得到 boot_params.scratch 的 32 位物理地址，存放到 esp 寄存器中。虽然前面说了的那个 X，grub 加载 linuz 的时候把这个值设置成 0x90000，但为了对所有 boot loader 兼容，所有代码在这里做了一个通用化处理，现场计算这个值。但请注意，这里并不是 0x90000，而是 boot_params.scratch 的 32 位物理地址，这样可以把前面已经执行过的代码略过，减少待会儿的拷贝量。然后到 62 行：

```
621:            popl    %ebp
63              subl    $1b, %ebp
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
74              decl    %eax
75              addl    %eax, %ebx
76              notl    %eax
77              andl    %eax, %ebx
78#else
79              movl    $LOAD_PHYSICAL_ADDR, %ebx
80#endif
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
```

62 行出栈，于是 ebp 寄存器就存放 boot_params.scratch 开始的 32 位的地址，随后再减 4，把 scratch 再略过。再来，由于我们在顶层.config 文件中看见 CONFIG_RELOCATABLE=y，所以进入 72 行，经过 72 到 77 行的处理，ebx 就指向了 BP_kernel_alignment 偏移的位置，按照注释讲是将要存放内核映像位置的 32 位首地址。因为 BP_kernel_alignment 正是 boot_params.hdr.kernel_alignment，在我们的 header.S 的第 200 行把它赋值成了 CONFIG_PHYSICAL_ALIGN，查看我们的.config 文件，这个值是 0x1000000。现在保护模式打开了，这个 32 位地址是在 1MB 以后的。所有后面的拷贝主要是把内核映像其余部分拷贝到 1MB 以后。

随后，对 ebx 进行调整，增加一个 z_extract_offset 的值，这个值我不太清楚是怎样来的，好像是编译的时候产生的。然后再把 ebx 偏移 boot_stack_end 的地址赋值给 esp，这样就可以使用栈了。

```
 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
```

然后进入 96 行代码，先把 boot_params 首地址压栈保存起来，然后将内核映像从 ebp 偏移 _bss-4 开始的内容拷贝到 ebx 偏移_bss-4 开始对应的内存单元中，拷贝的长度是_bss-startup_32。注意，_bss 是待会看到的解压缩程序的 BSS 段，_bss - startup_32 涵盖了 linuz 从 startup_32 起整个以后的代码长度，包括整个待解压内核，然后再右移 2 位，即除以 4，这样 102 行 rep 的时候按 4 个字节 32 位一组地进行拷贝，方便快捷。

注意第 101 行，设置了方向标志置位，所有 rep 的时候是由高地址向低地址进行，从_bss 到现在我们正在进行的 startup_32。

做这么一个拷贝的目的是把内核映像拷贝到 0x1000000 以后的内存单元中，进入保护模式后，下一步主要是解压缩内核映像，而这样就可以很安全的执行解压缩的 c 代码了。

```
105
106/*
107 * Jump to the relocated address.
108 */
109             leal    relocated(%ebx), %eax
110             jmp     *%eax
111ENDPROC(startup_32)
```

拷贝完成后，加载存在于 0x1000000 以后的内存单元中新的 relocated 位置。所用的是%ebx 对应的地址，而不是直接 jmp relocated。所以，看到 114 行的 relocated，执行到此时此刻，已经离开了刚才 grub 加载 linuz 的位置了，来到了新的保护模式位置。

```
113             .text
114relocated:
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
125
```

relocated 一来首先清洗 BSS 段，使用的 rep 指令，很简单因为刚才没有拷贝 BSS 段，而在这里必须建立一个新的，继续走：

```
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
139             addl    $20, %esp
140
```

这段代码全是 pushl，即压栈操作，为将进行的解压缩程序 decompress_kernel 函数准备参数，我们将在下一节予以详细分析。

4.1.2 解压缩内核

解压缩内核使用的是 decompress_kernel 函数，来自 arch/x86/boot/compressed/misc.c：

```
301asmlinkage void decompress_kernel(void *rmode, memptr heap,
302                                  unsigned char *input_data,
303                                  unsigned long input_len,
304                                  unsigned char *output)
305{
306             real_mode = rmode;
307
308             if (real_mode->hdr.loadflags & QUIET_FLAG)
309                     quiet = 1;
310
311             if (real_mode->screen_info.orig_video_mode == 7) {
312                     vidmem = (char *) 0xb0000;
313                     vidport = 0x3b4;
314             } else {
315                     vidmem = (char *) 0xb8000;
316                     vidport = 0x3d4;
317             }
318
319             lines = real_mode->screen_info.orig_video_lines;
320             cols = real_mode->screen_info.orig_video_cols;
321
322             free_mem_ptr     = heap;        /* Heap */
323             free_mem_end_ptr = heap + BOOT_HEAP_SIZE;
324
325             if ((unsigned long)output & (MIN_KERNEL_ALIGN - 1))
326                     error("Destination address inappropriately aligned");
327#ifdef CONFIG_X86_64
328             if (heap > 0x3fffffffffUL)
329                     error("Destination address too large");
330#else
331             if (heap > ((-__PAGE_OFFSET-(512<<20)-1) & 0x7fffffff))
```

```
332                     error("Destination address too large");
333#endif
334#ifndef CONFIG_RELOCATABLE
335             if ((unsigned long)output != LOAD_PHYSICAL_ADDR)
336                     error("Wrong destination address");
337#endif
338
339             if (!quiet)
340                     putstr("\nDecompressing Linux... ");
341             decompress(input_data, input_len, NULL, NULL, output, NULL, error);
342             parse_elf(output);
343             if (!quiet)
344                     putstr("done.\nBooting the kernel.\n");
345             return;
346}
```

看到他的参数，共有 5 个，是刚才依次压栈而得到的。第一个是压入的 output（还记得吧，倒着来的），在刚才的 131 行看到的，来自 z_extract_offset_negative，用于作为解压缩的缓存首地址；第二个，input_len，其值等于 z_input_len，表示待压缩内核大小；第三个，input_data，来自 input_data，表示待压缩内核地址；第四个参数，heap，32 位下是 32 位，来自 boot_heap，表示解压缩阶段的堆；最后一个参数，rmode，来自我们刚才用到的 esi 寄存器，表示刚才拷贝之前内核映像地址。

假设我配置的是 CONFIG_KERNEL_BZIP2，那么会调用顶层 lib/decompress_bunzip2.c 中的 decompress 函数，最终会调用位于同一文件的 bunzip2 函数（注意，在我们制作 bzImage 的时候使用的压缩程序只有一个，如果指定的 bunzip2，那么都在 decompress_bunzip2.c 里）：

```
674/* Example usage: decompress src_fd to dst_fd.  (Stops at end of bzip2 data,
675    not end of file.) */
676STATIC int INIT bunzip2(unsigned char *buf, int len,
677                         int(*fill)(void*, unsigned int),
678                         int(*flush)(void*, unsigned int),
679                         unsigned char *outbuf,
680                         int *pos,
681                         void(*error_fn)(char *x))
682{
683             struct bunzip_data *bd;
684             int i = -1;
685             unsigned char *inbuf;
686
687             set_error_fn(error_fn);
688             if (flush)
689                     outbuf = malloc(BZIP2_IOBUF_SIZE);
690
```

```
691         if (!outbuf) {
692                 error("Could not allocate output bufer");
693                 return RETVAL_OUT_OF_MEMORY;
694         }
695         if (buf)
696                 inbuf = buf;
697         else
698                 inbuf = malloc(BZIP2_IOBUF_SIZE);
699         if (!inbuf) {
700                 error("Could not allocate input bufer");
701                 i = RETVAL_OUT_OF_MEMORY;
702                 goto exit_0;
703         }
704         i = start_bunzip(&bd, inbuf, len, fill);
705         if (!i) {
706                 for (;;) {
707                         i = read_bunzip(bd, outbuf, BZIP2_IOBUF_SIZE);
708                         if (i <= 0)
709                                 break;
710                         if (!flush)
711                                 outbuf += i;
712                         else
713                                 if (i != flush(outbuf, i)) {
714                                     i = RETVAL_UNEXPECTED_OUTPUT_EOF;
715                                     break;
716                                 }
717                 }
718         }
719         /* Check CRC and release memory */
720         if (i == RETVAL_LAST_BLOCK) {
721                 if (bd->headerCRC != bd->totalCRC)
722                         error("Data integrity error when decompressing.");
723                 else
724                         i = RETVAL_OK;
725         } else if (i == RETVAL_UNEXPECTED_OUTPUT_EOF) {
726                 error("Compressed file ends unexpectedly");
727         }
728         if (!bd)
729                 goto exit_1;
730         if (bd->dbuf)
731                 large_free(bd->dbuf);
732         if (pos)
733                 *pos = bd->inbufPos;
734         free(bd);
```

```
735exit_1:
736             if (!buf)
737                     free(inbuf);
738exit_0:
739             if (flush)
740                     free(outbuf);
741             return i;
742}
```

bunzip2 函数的第一、二个参数是待压缩内核的首地址和长度；第三、四是两个函数参数，传进来的时候为空；第五个参数是解压缩后的地址，也是刚才传进来的。683 行首先初始化一个指向 bunzip_data 数据结构的指针 bd，待会再去说他。我们没有定义 flush 函数，所以定义一个局部指针 inbuf 指向刚才传递进来的待压缩内核首地址 buf，随后 704 行调用函数 start_bunzip：

```
626static int INIT start_bunzip(struct bunzip_data **bdp, void *inbuf, int len,
627                                     int (*fill)(void*, unsigned int))
628{
629             struct bunzip_data *bd;
630             unsigned int i, j, c;
631             const unsigned int BZh0 =
632                     (((unsigned int)'B') << 24)+(((unsigned int)'Z') << 16)
633                     +(((unsigned int)'h') << 8)+(unsigned int)'0';
634
635             /* Figure out how much data to allocate */
636             i = sizeof(struct bunzip_data);
637
638             /* Allocate bunzip_data.    Most fields initialize to zero. */
639             bd = *bdp = malloc(i);
640             if (!bd)
641                     return RETVAL_OUT_OF_MEMORY;
642             memset(bd, 0, sizeof(struct bunzip_data));
643             /* Setup input buffer */
644             bd->inbuf = inbuf;
645             bd->inbufCount = len;
646             if (fill != NULL)
647                     bd->fill = fill;
648             else
649                     bd->fill = nofill;
650
651             /* Init the CRC32 table (big endian) */
652             for (i = 0; i < 256; i++) {
653                     c = i << 24;
654                     for (j = 8; j; j--)
```

```
655                             c = c&0x80000000 ? (c << 1)^0x04c11db7 : (c << 1);
656                     bd->crc32Table[i] = c;
657             }
658
659             /* Ensure that file starts with "BZh['1'-'9']." */
660             i = get_bits(bd, 32);
661             if (((unsigned int)(i-BZh0-1)) >= 9)
662                     return RETVAL_NOT_BZIP_DATA;
663
664             /* Fourth byte (ascii '1'-'9'), indicates block size in units of 100k of
665                uncompressed data.    Allocate intermediate buffer for block. */
666             bd->dbufSize = 100000*(i-BZh0);
667
668             bd->dbuf = large_malloc(bd->dbufSize * sizeof(int));
669             if (!bd->dbuf)
670                     return RETVAL_OUT_OF_MEMORY;
671             return RETVAL_OK;
672}
```

由于 bdp 和 fill 都是空的，所以传入这个函数的有效的仅仅是待压缩内核首地址及其长度，内部变量 i 是数据结构 bunzip_data 的长度。而 631 行 BZh0 常量很重要，就是著名的 bz 压缩常量。我们看到这段代码：

```
const unsigned int BZh0 =
        (((unsigned int)'B') << 24)+(((unsigned int)'Z') << 16)
        +(((unsigned int)'h') << 8)+(unsigned int)'0';
```

就是压缩常量的值，'B'、'Z'、'h'、'0'是 ASCII 码，计数出来就是 0x48<<24 +0x5A<<16+0x68 最后等于 0x48166800。

继续走，分配 i 个内存单元，由指针 bd 指上，同时 bunzip2 函数的内部变量 bd 也指到它了，然后 642 行将该结构清零（这些都是规定动作，值得我们程序员学习）。看了我们不得不把数据结构 bunzip_data 列出来了：

```
 91/* Structure holding all the housekeeping data, including IO buffers and
 92    memory that persists between calls to bunzip */
 93struct bunzip_data {
 94             /* State for interrupting output loop */
 95             int writeCopies, writePos, writeRunCountdown, writeCount, writeCurrent;
 96             /* I/O tracking data (file handles, buffers, positions, etc.) */
 97             int (*fill)(void*, unsigned int);
 98             int inbufCount, inbufPos /*, outbufPos*/;
 99             unsigned char *inbuf /*,*outbuf*/;
100             unsigned int inbufBitCount, inbufBits;
101             /* The CRC values stored in the block header and calculated from the
102             data */
```

```
103         unsigned int crc32Table[256], headerCRC, totalCRC, writeCRC;
104         /* Intermediate buffer and its size (in bytes) */
105         unsigned int *dbuf, dbufSize;
106         /* These things are a bit too big to go on the stack */
107         unsigned char selectors[32768];          /* nSelectors = 15 bits */
108         struct group_data groups[MAX_GROUPS];    /* Huffman coding tables */
109         int io_error;                            /* non-zero if we have IO error */
110};
```

644 和 645 行初始化 bunzip_data 的 inbuf 和 inbufCount 字段。随后给 crc32Table 字段赋值，根据注释说，这个字段是存储块的头，随后调用 get_bits 函数：

```
113/* Return the next nnn bits of input.    All reads from the compressed input
114     are done through this function.    All reads are big endian */
115static unsigned int INIT get_bits(struct bunzip_data *bd, char bits_wanted)
116{
117         unsigned int bits = 0;
118
119         /* If we need to get more data from the byte buffer, do so.
120              (Loop getting one byte at a time to enforce endianness and avoid
121              unaligned access.) */
122         while (bd->inbufBitCount < bits_wanted) {
123                   /* If we need to read more data from file into byte buffer, do
124                        so */
125                   if (bd->inbufPos == bd->inbufCount) {
126                             if (bd->io_error)
127                                       return 0;
128                             bd->inbufCount = bd->fill(bd->inbuf, BZIP2_IOBUF_SIZE);
129                             if (bd->inbufCount <= 0) {
130                                  bd->io_error = RETVAL_UNEXPECTED_INPUT_EOF;
131                                  return 0;
132                             }
133                             bd->inbufPos = 0;
134                   }
135                   /* Avoid 32-bit overflow (dump bit buffer to top of output) */
136                   if (bd->inbufBitCount >= 24) {
137                             bits = bd->inbufBits&((1 << bd->inbufBitCount)-1);
138                             bits_wanted -= bd->inbufBitCount;
139                             bits <<= bits_wanted;
140                             bd->inbufBitCount = 0;
141                   }
142                   /* Grab next 8 bits of input from buffer. */
143                   bd->inbufBits = (bd->inbufBits << 8)|bd->inbuf[bd->inbufPos++];
144                   bd->inbufBitCount += 8;
```

```
145             }
146             /* Calculate result */
147             bd->inbufBitCount -= bits_wanted;
148             bits |= (bd->inbufBits >> bd->inbufBitCount)&((1 << bits_wanted)-1);
149
150             return bits;
151}
```

这个函数比较简单，一开始 inbufBitCount 肯定小于 32，因为被清零了。由于是解压缩阶段，bd->inbufPos 也为 0，不可能等于 inbufCount。所以第一次循环直接来到 143 行，第一次循环，bd->inbufBits 也就是 bd->inbuf[bd->inbufPos++]的值，bd->inbufPos 加加之后就变成 1 了，所以 bd->inbufBits = bd->inbuf[1]，144 行 bd->inbufBitCount 为 8 了；

第二次循环，bd->inbufBitCount 为 8 还是小于 32，也小于 24，所以又赋值 bd->inbufBits 为 bd->inbuf[1] << 8|bd->inbuf[2]，bd->inbufPos 加加之后就变成 2 了，bd->inbufBitCount 成了 16。

第三次循环，bd->inbufBitCount 为 16 还是小于 32，也小于 24，所以又赋值 bd->inbufBits 为 bd->inbuf[1]<<8|bd->inbuf[2]<<8|bd->inbuf[3]，bd->inbufPos 加加之后就变成 3 了，bd->inbufBitCount 成了 24。

第四次循环，bd->inbufBitCount 为 24 了，还是小于 32，但是要进入 136 行的条件块。只不过得到了一个 bits 值，这个值是：
bd->inbuf[1]<<8|bd->inbuf[2]<<8|bd->inbuf[3]&(1 << bd->inbufBitCount)-1)
换算过来就是 bd->inbuf[1]<<8|bd->inbuf[2]<<8|bd->inbuf[3]&0xfffff。最后 139 行再把这么长串左移 8 位（此时 bits_wanted 在 138 行做了个运算：32-24=8）。最后转了这么久，bd->inbufBitCount 又为 0 了。是不是要进入一个死循环了呢？没有，143 行 bd->inbufBits 继续走，bd->inbuf[1]<<8|bd->inbuf[2]<<8|bd->inbuf[3]|bd->inbuf[4]，但是 bits_wanted 却是变成 8 了，所以跳出循环了。现在感受到内核代码的变态了吧，下面的更变态：bd->inbufBitCount 被清零了，bits，也就是刚才那么长一串还要继续去|= (bd->inbufBits >> 8)&((1 << 8)-1)。

最后返回这个 bits 值。这个值是多少，我实在是不知道，只有机器知道。但根据注释我们知道这个值的意思是待压缩内核压缩组合块的数量，压缩程序是按 4 个字节一组进行压缩的，每个字节按位移动 8 位，4 个字节压缩成一个块，称为压缩组合块。以我的水平只能分析到这种程度了，请吃透了这里代码的同志联系我啊，一定会请你喝酒的。

回到 start_bunzip 中，刚才 bits 返回给了 i，660 行 i-BZh0-1 就是 0-0x48166800-1 转换成无符号整形，就是 0x48166801，最后 i-0x48166801 肯定是小于 9 的，不然就不可能继续走了，我们假设这个值为 8。最后 666 行和 668 行给 bd->dbufSize 和 bd->dbuf 赋值，其长度就是 800k。

最后返回到 bunzip2 中，start_bunzip 最后返回的 RETVAL_OK 为 0，所以进入 705 这个条件语句中，一来就进入 read_bunzip 函数：

```
/* Undo burrows-wheeler transform on intermediate buffer to produce output.
   If start_bunzip was initialized with out_fd =-1, then up to len bytes of
   data are written to outbuf.  Return value is number of bytes written or
   error (all errors are negative numbers).  If out_fd!=-1, outbuf and len
   are ignored, data is written to out_fd and return is RETVAL_OK or error.
*/

static int INIT read_bunzip(struct bunzip_data *bd, char *outbuf, int len)
{
	const unsigned int *dbuf;
	int pos, xcurrent, previous, gotcount;

	/* If last read was short due to end of file, return last block now */
	if (bd->writeCount < 0)
		return bd->writeCount;

	gotcount = 0;
	dbuf = bd->dbuf;
	pos = bd->writePos;
	xcurrent = bd->writeCurrent;

	/* We will always have pending decoded data to write into the output
	   buffer unless this is the very first call (in which case we haven't
	   Huffman-decoded a block into the intermediate buffer yet). */

	if (bd->writeCopies) {
		/* Inside the loop, writeCopies means extra copies (beyond 1) */
		--bd->writeCopies;
		/* Loop outputting bytes */
		for (;;) {
			/* If the output buffer is full, snapshot
			 * state and return */
			if (gotcount >= len) {
				bd->writePos = pos;
				bd->writeCurrent = xcurrent;
				bd->writeCopies++;
				return len;
			}
			/* Write next byte into output buffer, updating CRC */
			outbuf[gotcount++] = xcurrent;
			bd->writeCRC = (((bd->writeCRC) << 8)
				^bd->crc32Table[((bd->writeCRC) >> 24)
				^xcurrent]);
			/* Loop now if we're outputting multiple
```

```
557                             * copies of this byte */
558                             if (bd->writeCopies) {
559                                     --bd->writeCopies;
560                                     continue;
561                             }
562decode_next_byte:
563                             if (!bd->writeCount--)
564                                     break;
565                             /* Follow sequence vector to undo
566                              * Burrows-Wheeler transform */
567                             previous = xcurrent;
568                             pos = dbuf[pos];
569                             xcurrent = pos&0xff;
570                             pos >>= 8;
571                             /* After 3 consecutive copies of the same
572                                byte, the 4th is a repeat count.  We count
573                                down from 4 instead *of counting up because
574                                testing for non-zero is faster */
575                             if (--bd->writeRunCountdown) {
576                                     if (xcurrent != previous)
577                                             bd->writeRunCountdown = 4;
578                             } else {
579                                     /* We have a repeated run, this byte
580                                      * indicates the count */
581                                     bd->writeCopies = xcurrent;
582                                     xcurrent = previous;
583                                     bd->writeRunCountdown = 5;
584                                     /* Sometimes there are just 3 bytes
585                                      * (run length 0) */
586                                     if (!bd->writeCopies)
587                                             goto decode_next_byte;
588                                     /* Subtract the 1 copy we'd output
589                                      * anyway to get extras */
590                                     --bd->writeCopies;
591                             }
592                     }
593                     /* Decompression of this block completed successfully */
594                     bd->writeCRC = ~bd->writeCRC;
595                     bd->totalCRC = ((bd->totalCRC << 1) |
596                                     (bd->totalCRC >> 31)) ^ bd->writeCRC;
597                     /* If this block had a CRC error, force file level CRC error. */
598                     if (bd->writeCRC != bd->headerCRC) {
599                             bd->totalCRC = bd->headerCRC+1;
600                             return RETVAL_LAST_BLOCK;
```

```
601                     }
602             }
603
604             /* Refill the intermediate buffer by Huffman-decoding next
605               * block of input */
606             /* (previous is just a convenient unused temp variable here) */
607             previous = get_next_block(bd);
608             if (previous) {
609                     bd->writeCount = previous;
610                     return (previous != RETVAL_LAST_BLOCK) ? previous : gotcount;
611             }
612             bd->writeCRC = 0xffffffffUL;
613             pos = bd->writePos;
614             xcurrent = bd->writeCurrent;
615             goto decode_next_byte;
616}
```

看到这个函数，我都冒冷汗了。具体的我实在没能力去分析他了，感兴趣的同志可以帮帮我，谢谢。最后解压后的程序会存放到 outbuf 开始的内存中。这个函数结束后，bunzip2 也就带着结束了 RETVAL_OK 结束了，decompress 也就结束了。回到 boot/compressed/head_32.S 的代码中：

```
141#if CONFIG_RELOCATABLE
……
165#endif
```

由于我们.config 没有 CONFIG_RELOCATABLE，所以不去详细分析 141～165 行的代码，直接来到 170 行：

```
170             xorl    %ebx, %ebx
171             jmp     *%ebp
```

开始执行解压缩后的第一条代码，即第二个 startup_32()函数。这个函数主要是为第一个 Linux 进程（进程 0）建立执行环境。该函数主要执行以下操作：

1. 把段寄存器初始化为最终值。
2. 把内核的 bss 段填充为 0。
3. 初始化包含在 swapper_pg_dir 的临时内核页表，并初始化 pg0，以使线性地址一致地映射同一物理地址。
4. 把页全局目录的地址存放在 cr3 寄存器中，并通过设置 cr0 寄存器的 PG 位启用分页。
5. 把从 BIOS 中获得的系统参数和传递给操作系统的参数 boot_params 放入第一个页框中。
6. 为进程 0 建立内核态堆栈。
7. 该函数再一次清零 eflags 寄存器的所有位。
8. 调用 setup_idt 用空的中断处理程序填充中断描述符表 IDT。
9. 识别处理器的型号。

10. 用编译好的 GDT 和 IDT 表的地址来填充 gdtr 和 idtr 寄存器。
11. 初始化虚拟机监视器 Xen。
12. 向 start_kernel()函数进发。

后面咱们会详细分析以上步骤。

4.1.3 第二次启动保护模式

我们在链接 vmlinux 一节中看到，定义 phys_startup_32 为程序入口点，这个入口点才是解压缩后内核的真正开始的地方，而在链接脚本中，phys_startup_32 是逻辑地址 startup_32 的物理地址。从进入保护模式那一刻起，程序就是用逻辑地址了，不过在启动分页机制之前，逻辑地址向物理地址的转换很简单，仅仅是 va 和 pa 操作，即物理地址转成逻辑地址和逻辑地址转成物理地址。

第二个 startup_32 在 arch/x86/kernel/head_32.S 的 85 行，我们就开始分析它了：

```
85ENTRY(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)
95        movl $(__BOOT_DS),%eax
96        movl %eax,%ds
97        movl %eax,%es
98        movl %eax,%fs
99        movl %eax,%gs
```

跟前面一样，BP_loadflags 的第七位没有设置，所以重新设置一下保护模式的环境。我们前面在 setup_gdt()函数中 中看到过通过 C 语言设置保护模式的方法，这里再通过汇编代码复习一下。94 行，加载全局描述符表寄存器，将 boot_gdt_descr 的物理地址所对应的内容加载到 GDTR 寄存器以作为全局描述符表的基地址。

看到 94 行，著名的 pa 宏第一次出现了，来自同一个文件的第 26 行：

```
#define pa(X) ((X) - __PAGE_OFFSET)
```

最重要的__PAGE_OFFSET 出现了，下面好几个地方都有__PAGE_OFFSET，这是因为要引用某个变量所在的地址，那么必须找到物理地址，这就是 pa 宏的作用。__PAGE_OFFSET 被定义为 CONFIG_PAGE_OFFSET，在 32 位 x86 保护模式下，默认为 0xC0000000。

而此刻因为没有分页，所以实际上变量的偏移的值都是实际的 n 再加上 0xC0000000（因为内核最终要分页，所以链接的时候都是相对这个偏移，一会再说说），所以如果不减去__PAGE_OFFSET，那么比如上面 boot_gdt_descr 的值就是 0xC0100000+n。由于 n 是个不大的值，是 vmlinux 中 boot_gdt_desc 相对保护模式开始的偏移。这样，boot_gdt_descr - __PAGE_OFFSET 之后就是 n，这正是 boot_gdt_descr 所在的物理地址。

全局描述符表在哪儿？看到 696 行：

```
696boot_gdt_descr:
697             .word __BOOT_DS+7
698             .long boot_gdt - __PAGE_OFFSET
699
700             .word 0                                 # 32-bit align idt_desc.address
```

由于 GDTR 是一个长度为 48bit 的寄存器，内容为一个 32 位的基地址和一个 16 位的段限。其中 32 位的基址是指 GDT 在内存中的地址。被 GDTR 通过 94 行指令加载的内容就是上面的内容，先是 697 行的段限。__BOOT_DS 还记得吧，初始化阶段的数据段选择子，其值为 0x88，加上 7 就是 0x8f，这个值占据了 2 个字节的空间，16 位，作为段限。

而 GDTR 的 32 位基地址是初始化阶段的全局描述符表现性地址 boot_gdt 的物理地址，这个地址定位到 716 行：

```
716ENTRY(boot_gdt)
717             .fill GDT_ENTRY_BOOT_CS,8,0
718             .quad 0x00cf9a000000ffff            /* kernel 4GB code at 0x00000000 */
719             .quad 0x00cf92000000ffff            /* kernel 4GB data at 0x00000000 */
```

这个四个字节作为段基地址；最后 2 个字节是段描述符段限的其他部分。这里我们又学个新知识，看上面有个.fill。GNU 汇编程序提供了很多这样的指令，这种指令都是以句点（.）为开头，后跟指令名（小写字母），.fill 的形式是 .fill repeat , size , value

其中，repeat、size 和 value 都是常量表达式。Fill 的含义是反复拷贝 size 个字节。Repeat 可以大于等于 0。size 也可以大于等于 0，但不能超过 8，如果超过 8，也只取 8。把 repeat 个字节以 8 个为一组，每组的最高 4 个字节内容为 0，最低 4 字节内容置为 value。所以我们看到由于 GDT_ENTRY_BOOT_CS 定义为 2，所以 717 行代码意思是申请两个 8 字节的空间，其内容为 0。

.quad 也是个新知识，表示零个或多个 bignums（用逗号分隔），对于每个 bignum，其缺省值是 8 字节整数。如果 bignum 超过 8 字节，则打印一个警告信息；并只取 bignum 最低 8 字节。例如，718 行对全局描述符表的填充就用到这个指令，第一个 8 字节大数是 0x00cf9a000000ffff，表示内核 4GB 代码段的描述符内容，起始地址为 0x00000000；第二个 8 字节大数是 0x00cf92000000ffff，表示内核 4GB 数据段的描述符内容，起始地址同样也为 0x00000000。注意，这里还处于初始化阶段，不存在用户代码段和数据段。

有关其他有用的 GNU 汇编程序指令请参考我的博客“ Linux 中的汇编语言 ”
http://blog.csdn.net/yunsongice/archive/2010/10/08/5927895.aspx

boot_gdt 就是初始化阶段描述符表的内容，698 行的运算就是得到它的物理地址。如果对描述符、选择子这些概念还不熟悉的同学，请查阅博客“ Intel 80286 工作模式 ”
http://blog.csdn.net/yunsongice/archive/2010/10/04/5920447.aspx

注意，在进入保护模式后，Linux 进行了第二次段寻址的设置，也就是第二次启动保护模式，这一次设置的原因是在之前的处理过程中，指令地址是从物理地址 0x100000 开始的，而此时整个 vmlinux 的编译链接地址是从虚拟地址 0xC0000000 开始的，所以需要在这里重新设置 boot_gdt 的位置。

随后 95~99 行就是设置 4 个数据段寄存器的值，存放数据段选择子__BOOT_DS。

```
101 2:
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
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
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
```

105 行，没有问题，清除方向标志，使得 SI->DI。107、108 行，将__bss_start 和__bss_stop

的物理地址分别计算出来并传递给 edi 和 ecx 寄存器。然后再讲这两个值一相减，得到 BSS 内核未初始化数据段的长度。当然，为了执行 rep 指令，还要右移两位，就得到了 32 位一个传输单元的 BSS 段长度。随后执行 rep 指令，还记得这个汇编指令把，把 DS:SI 指向的内存单元传输到 ES:DI 执行的内存单元，长度是 ecx。

第 120~123 行，再把 edi 指向 boot_params 的保护模式下物理地址，ecx 设置成 boot_params 的长度，然后 rep 拷贝。注意这个 PARAM_SIZE，正好 1 页大小，在 arch/x86/include/asm/Setup.h 中定义：

```
#define PARAM_SIZE 4096         /* sizeof(struct boot_params) */
```

我们看到 protected_mode_jump 函数把 setup 中的 boot_params 的参数放到%esi 中了复制一份，这类却再一次。为什么复制两次呢？你想想啊，解压缩以后跳到了解压缩后的代码，再经过上述重置保护模式环境的代码后，GDTR 和数据段寄存器中存放的内容已经变了，如果不做这么一个复制，那么后面的代码永远也找不到这个 boot_params 了。

注意，注释里写得很清楚，为什么这里要进行这么一个拷贝，因为%esi 指向的内存区还是实模式下存放的那个 boot_params，经历了这么久以后一直都没变过，而且最关键的是保护模式的环境已经改变了，vmlinux 的编译链接地址已经是从虚拟地址 0xC0000000 开始了，所以这样的拷贝是必须地。

最后 125~131 行的代码是复制启动参数的 NEW_CL_POINTER 部分到 boot_command_line，这个参数负责命令行，执行的过程跟前面差不多，我们就不细说了。

CONFIG_PARAVIRT 主要用来针对 bootloader 损坏的情况，没有在我们的.config 里面，所以略过 134~165 行代码。而又由于我们没有启动强大的 x83 内存扩展，所以 CONFIG_X86_PAE 也不用考虑，再略过 177~225 行的代码，直接来到 227 行。

4.1.4 第一次启动分页管理

从 227 行开始，著名的分页机制就粉墨登场了：

```
 227page_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
 23210:
 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 */
```

227 行设置一个常量 page_pde_offset，其值 0xc0000000 >> 20 = 0xc00，其作用待会再说。229 行，__brk_base，这个东西咱们见过，在链接 vmlinux 时，__brk_base 作为 BRK 段的开始地址。BRK 段是保留给用户进程通过系统调用 brk() 向内核申请的内存空间。

对这个 brk 我还要多说几句，传统 UNIX 系统有两个系统调用 brk 和 sbrk，主要的工作是实现虚拟内存到实际内存的映射。在 GNU C 程序中，内存分配是这样的：

每个进程可访问的虚拟内存空间为 3G，但在程序编译时，不可能也没必要为程序分配这么大的空间，只分配并不大的数据段空间，程序中动态分配的空间就是从这一块分配的。如果这块空间不够，malloc 函数族（realloc，calloc 等）就调用 brk 系统调用将数据段的下界移动，brk 函数在内核的管理下将虚拟地址空间映射到内存，供 malloc 函数使用。

所以，这个__brk_base 对应的空间就是编译链接 vmlinux 时的数据段下界，229 行就是把这个下界的物理地址赋给了 edi 寄存器。

继续走，230 行，swapper_pg_dir。这个东西就更重要了，我们操作系统的分页单元就全靠它了。它的定义在哪儿呢？同一文件 621 行：

```
621ENTRY(swapper_pg_dir)
622             .fill 1024,4,0
623#endif
```

我们看到.fill 1024,4,0 的意思是重复 1024 次，每组 4 个字节，内容为 0。这正好是一个页面的大小，4k 字节。这个页面的物理地址是$pa(swapper_pg_dir)，内核把它作为第一个页表，把它的物理地址赋给 edx 寄存器。

再来，231 行$PTE_IDENT_ATTR 常量，来自 arch/x86/include/asm/pgtable_types.h：

```
#define PTE_IDENT_ATTR   0x003         /* PRESENT+RW */
#define PDE_IDENT_ATTR   0x067         /* PRESENT+RW+USER+DIRTY+ACCESSED */
#define PGD_IDENT_ATTR   0x001         /* PRESENT (no other attributes) */
```

把 0x003 保存到 eax 寄存器中。随后 233 行 lea 指令，保存地址值。不明白 lea 指令的注意了，这个指令太重要了。lea 指令有两个操作数，其目的操作数，表示操作结果保存在此，该指令目的操作数只能是 8 个通用寄存器之一。而源操作数只能是一个存储单元，表达存储单元有多种寻址方式。

lea 指令的功能是将源操作数、即存储单元的有效地址（偏移地址）传送到目的操作数。代码指令中 PDE_IDENT_ATTR(%edi)采用间接寻址方式表达存储单元，它表示的存储单元的有效地址是：__brk_base 的物理地址再加上 0x67，然后被传送到 ecx 中。

这样做的目的是什么呢？我们看到，__brk_base 的物理地址是 32 位的，而一个页面是 4KB，于是乎最低 12 位必须为 0，不然这个页面就没法对其了。所以，这里__brk_base 物理地址的最低 12 位是无效的，内核另作他用，这里把最低 12 位中的 0x67 置位，就是说明把 PRESENT、RW、USER、DIRTY 和 ACCESSED 置位，然后把整个值传递到 ecx 中。然后 234 行就把这个地址放到 swapper_pg_dir 的第一个 4 字节的单元，也就是 edx 寄存器对应的那个存储单元开始的四个存储单元，作为页全局目录的第 1 项。

继续走，235 行，page_pde_offset，其值 0xc00，我们刚才看到了。这里再把 ecx 中存放的

__brk_base 物理地址存放到 swapper_pg_dir 偏移 0xc00 的内存单元中。0xc00 换算成十进制就是：3072，由于每个表项占 4 个内存单元，所以这个偏移就是 3072/4=768，也就是以 swapper_pg_dir 开始的全局页目录的第 768 个表项。这样我们就得出了一个重要的结论，页全局目录为 swapper_pg_dir 开始的 4096 个内存单元，正好占用 1 个页面，包含 1024 个表项，但是同时从第 1 个和第 768 个表项开始分配，而且这些个表项的 PRESENT、RW、USER、DIRTY 和 ACCESSED 被置位。没有用到表项都为空。

```
236             addl $4,%edx
237             movl $1024, %ecx
23811:
239             stosl
240             addl $0x1000,%eax
241             loop 11b
```

236 行，edx 加上 4 个字节，正好对应页全局目录的第 2 个表项。237 行，ecx 的值设为 1024。看到这里，大家马上会想到肯定又要进入循环了，不错 241 行那个 loop 正是这个循环的标志。

239 行存储字符串数据(Store String Data)，将累加器 ecx 内容存储到由 es:edi 寻址的内存地址。如果使用 stos 指令，必须指定目的操作数，stosl 拷贝 eax 到 es:edi 所寻址的内存中。所以 239 行将 eax 存放的 4 个字节，32 位的数值 0x3 存放到 es:edi 对应的内存单元，也就是 __brk_base 对应的那个内存单元。

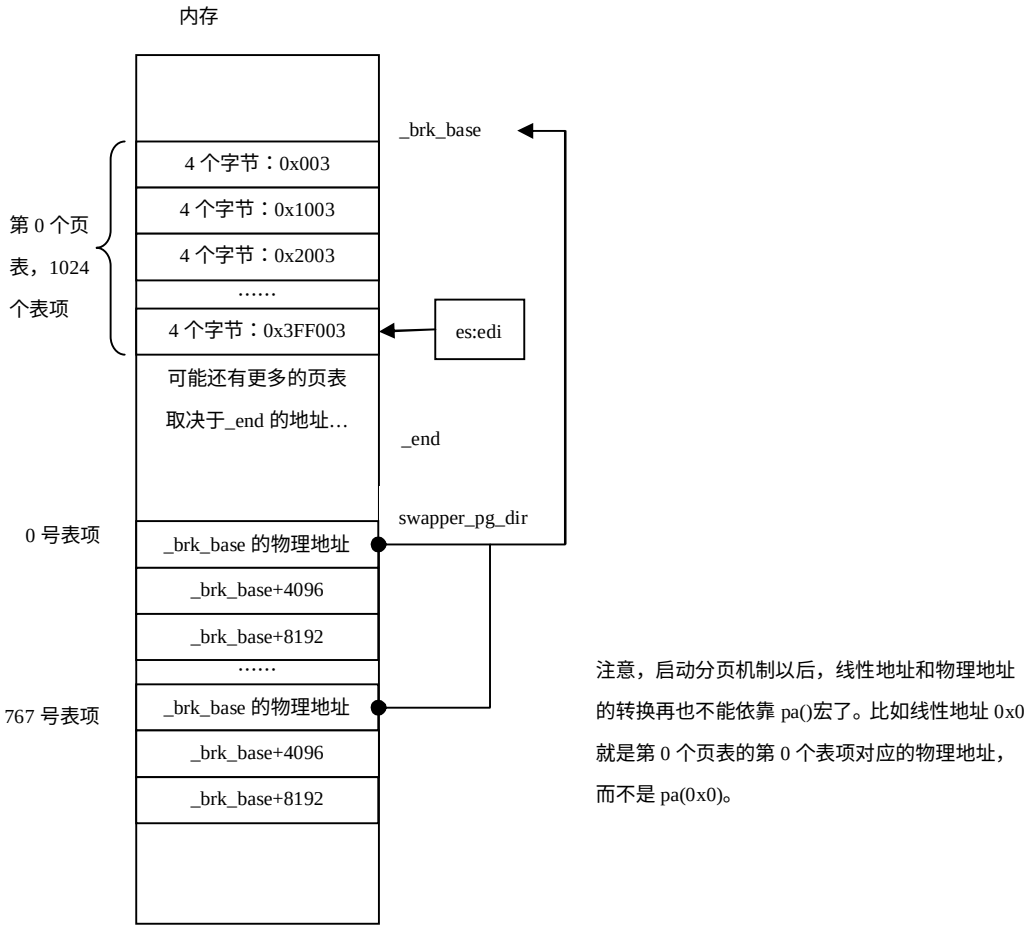
240 行，eax 再加 0x1000，等于 0x1003，这样进入循环，循环 1024 次后，__brk_base 对应的那个内存单元就是首地址为 swapper_pg_dir，共有 1024 个表项的第一个页表。第 1 个表项的内容为 0x3，第 2 个为 0x1003，第 3 个为 0x2003，第 4 个为 0x3003，……，第 1024 个表项为 0x3FF003。很多人分析代码时都不太注意这些细节，其实内核代码主要考验大家的就是掌握计算机知识的基本功，都是我们在学校学过的，没有什么高深的东西。人家 Torvalds 当年发明 Linux 的时候不也就是个小国家的本科生嘛。

Intel 指令集中有 5 组处理字节，字和双字数组的指令，称为基本字符串指令，对于他们的这些基本概念请查阅博客“ Intel 8086/8088 指令系统（四）”
http://blog.csdn.net/yunsongice/archive/2010/10/04/5920424.aspx

```
245             movl $pa(_end) + MAPPING_BEYOND_END + PTE_IDENT_ATTR, %ebp
246             cmpl %ebp,%eax
247             jb 10b
248             addl $__PAGE_OFFSET, %edi
249             movl %edi, pa(_brk_end)
250             shrl $12, %eax
251             movl %eax, pa(max_pfn_mapped)
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
```

由于我们的页表映射关系是以 end + MAPPING_BEYOND_END 结束，所以还有些尾巴摇处理了，245 到 255 行就是处理这个，首先比较一下 end + MAPPING_BEYOND_END 是否超过了 1 页，如果超过了，就还要跳回去再申请若干页全局目录的表项，和若干个作为页表的页面。最后还要申请些空间来处理 fixmap 区。



如上图所示，最后以_brk_base 开始的内存单元存放了若干个页表，每个页表占据一个页面。页表项的内容从 0x003 开始，到$pa(_end) + MAPPING_BEYOND_END + PTE_IDENT_ATTR 取决于_end 的物理地址。所以在这里页全局目录和页表建立完成后，就完成了从物理地址到链接 vmlinux 后的结束地址_end 的所有代码的映射，包括__brk_base 到__brk_limit 这段 BRK 段本身的代码。

略过 SMP 和 MMU 的相关初始化代码（其实这些代码也很重要，只不过太偏了，感兴趣的同学可以自行研究），来到 331 行：

```
328/*
```

```
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 */
336             ljmp $__BOOT_CS,$1f         /* Clear prefetch and normalize %eip */
3371:
338             /* Set up the stack pointer */
339             lss stack_start,%esp
```

当页全局目录和页表建立好一会，就可以启动分页机制了，其步骤很简单，就是把页全局目录的 32 位物理地址传递给 cr3 寄存器，然后启动 cr0 的分页装置（$X86_CR0_PG 项字段）。综上所述，我们看到在 32 位内核的分页环境是启用的三级机制，即 1 个页全局目录，若干页表和页面。

4.1.5 初始化 0 号进程

336 行，进入分页后的内核代码段，执行 lss stack_start,%esp 指令，立即为进程 0 建立内核态堆栈。stack_start 定义在 657 行：

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

我们看到内核态堆栈由 init_thread_union 表示，其在 include/linux/sched.h 中被定义成一个全局变量：

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

#define THREAD_SIZE         (PAGE_SIZE << THREAD_ORDER)
```

由于 PAGE_SIZE 是 4096，THREAD_ORDER 在 32 位 x86 体系中是 1，所以 THREAD_SIZE 的值为 8k。所以这个 thread_union 的大小也为 8k。关于内核栈的详细内容请参考博客“进程相关的数据结构” http://blog.csdn.net/yunsongice/archive/2010/04/18/5499603.aspx

那么，对于一个进程，最重要的是什么呢？两个东西，一个是 thread_info，另一个就是著名的 task_struct。这个 0 号进程的 thread_info，我们已经看到了，是在 init_thread_union 中。那么这个 task_struct 又在哪儿呢？看到 arch/x86/kernel/init_task.c：

```
23 union thread_union init_thread_union __init_task_data =
```

```
24              { INIT_THREAD_INFO(init_task) };
```

扎眼一看，很奇怪，怎么多了个__init_task_data？别紧张，其实，它只是一个宏，不是什么变了，来自 include/linux/init_task.h 的 186 行：

```
#define __init_task_data __attribute__((__section__(".data.init_task")))
```

那么，上面的内容就展开成了：

```
union thread_union init_thread_union __attribute__((__section__(".data.init_task"))) =
{ INIT_THREAD_INFO(init_task) };
```

这是一条赋值语句，对联合体 init_thread_union 赋初始值。晕了吧，c 语言的功底是多么重要啊，通过内核的学习，你也学会了如何对联合体赋初值并且把数据放进指定的数据段.data.init_task 了吧。

继续，具体到如何赋值呢？先讲讲 init_task 这个全局变量，定义在 arch/x86/kernel/init_task.c 的第 31 行：

```
struct task_struct init_task = INIT_TASK(init_task);
```

这不，0 号进程的 task_struct 出现了，就是 init_task。在对 init_thread_union 赋初值的时候，同时也通过调用 INIT_TASK 宏对 init_task 赋初值：

```
110#define INIT_TASK(tsk)  \
111{                                                                       \
112             .state          = 0,                                    \
113             .stack          = &init_thread_info,                    \
114             .usage          = ATOMIC_INIT(2),                       \
115             .flags          = PF_KTHREAD,                           \
116             .lock_depth     = -1,                                   \
117             .prio           = MAX_PRIO-20,                          \
118             .static_prio    = MAX_PRIO-20,                          \
119             .normal_prio    = MAX_PRIO-20,                          \
120             .policy         = SCHED_NORMAL,                         \
121             .cpus_allowed   = CPU_MASK_ALL,                         \
122             .mm             = NULL,                                 \
123             .active_mm      = &init_mm,                             \
124             .se             = {                                     \
125                     .group_node     = LIST_HEAD_INIT(tsk.se.group_node),    \
126             },                                                      \
127             .rt             = {                                     \
128                     .run_list       = LIST_HEAD_INIT(tsk.rt.run_list),      \
129                     .time_slice     = HZ,                           \
130                     .nr_cpus_allowed = NR_CPUS,                     \
131             },                                                      \
132             .tasks          = LIST_HEAD_INIT(tsk.tasks),            \
```

```
133         .pushable_tasks = PLIST_NODE_INIT(tsk.pushable_tasks, MAX_PRIO), \
134         .ptraced        = LIST_HEAD_INIT(tsk.ptraced),                   \
135         .ptrace_entry   = LIST_HEAD_INIT(tsk.ptrace_entry),              \
136         .real_parent    = &tsk,                                          \
137         .parent         = &tsk,                                          \
138         .children       = LIST_HEAD_INIT(tsk.children),                  \
139         .sibling        = LIST_HEAD_INIT(tsk.sibling),                   \
140         .group_leader   = &tsk,                                          \
141         .real_cred      = &init_cred,                                    \
142         .cred           = &init_cred,                                    \
143         .cred_guard_mutex =                                              \
144                 __MUTEX_INITIALIZER(tsk.cred_guard_mutex),               \
145         .comm           = "swapper",                                     \
146         .thread         = INIT_THREAD,                                   \
147         .fs             = &init_fs,                                      \
148         .files          = &init_files,                                   \
149         .signal         = &init_signals,                                 \
150         .sighand        = &init_sighand,                                 \
151         .nsproxy        = &init_nsproxy,                                 \
152         .pending        = {                                              \
153                 .list = LIST_HEAD_INIT(tsk.pending.list),                \
154                 .signal = {{0}}},                                        \
155         .blocked        = {{0}},                                         \
156         .alloc_lock     = __SPIN_LOCK_UNLOCKED(tsk.alloc_lock),          \
157         .journal_info   = NULL,                                          \
158         .cpu_timers     = INIT_CPU_TIMERS(tsk.cpu_timers),               \
159         .fs_excl        = ATOMIC_INIT(0),                                \
160         .pi_lock        = __RAW_SPIN_LOCK_UNLOCKED(tsk.pi_lock),         \
161         .timer_slack_ns = 50000, /* 50 usec default slack */             \
162         .pids = {                                                        \
163                 [PIDTYPE_PID]  = INIT_PID_LINK(PIDTYPE_PID),             \
164                 [PIDTYPE_PGID] = INIT_PID_LINK(PIDTYPE_PGID),            \
165                 [PIDTYPE_SID]  = INIT_PID_LINK(PIDTYPE_SID),             \
166         },                                                               \
167         .dirties = INIT_PROP_LOCAL_SINGLE(dirties),                      \
168         INIT_IDS                                                         \
169         INIT_PERF_EVENTS(tsk)                                            \
170         INIT_TRACE_IRQFLAGS                                              \
171         INIT_LOCKDEP                                                     \
172         INIT_FTRACE_GRAPH                                                \
173         INIT_TRACE_RECURSION                                             \
174         INIT_TASK_RCU_PREEMPT(tsk)                                       \
175}
```

至于 task_struct 的详细介绍，请查看博客“进程相关的数据结构”，这里只挑几个重要的来说说。0 号进程的 stack 就是刚才 init_thread_info 的地址；parent 是他自己；thread 是 INIT_THREAD。还有一些是常量，如 init_fs。等我们用到了再回过来看它。

现在，0 号进程 task_struct 有了，下面就该来讲讲 INIT_THREAD_INFO 宏了，在 arch/x86/include/asm/thread_info.h：

```
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}
```

执行完这个宏以后，init_thread_union 就被初始化成以上内容了，至此，0 号进程的 task_struct 和 thread_info 就初始化完毕了。注意，这个初始化不是在 head_32.S 中初始化的。大家也看到了，上面的内容都是在编译的时候作为一个数据全局数据被初始化的，并且把他们放在.data.init_task 数据段中。head_32.S 中只是将 esp 指针指向这个已经被初始化了的 init_thread_union 以上 8k 作为栈顶，并向下移动（回忆一下 c 程序的特点，并参考“进程相关的数据结构 http://blog.csdn.net/yunsongice/archive/2010/04/18/5499603.aspx ”中那个图）。

4.2 向 start_kernel 进发

分段、分页单元完成，并且我们的 0 号进程的运行环境初步搭建起来以后，那么就要“走向现代”，去调用 start_kernel 了，调用它之前还有几个重要的步骤要走。

4.2.1 初始化中断描述符表

在完成了最重要和最核心的分段、分页环境的初始化后，还有一个就是初始化中断描述符 idt 的工作。

```
341/*
342 * Initialize eflags.  Some BIOS's leave bits like NT set.  This would
343 * confuse the debugger if this code is traced.
```

```
344 * XXX - best to initialize before switching to protected mode.
345 */
346             pushl $0
347             popfl
348
349#ifdef CONFIG_SMP
350             cmpb $0, ready
351             jz    1f                                /* Initial CPU cleans BSS */
352             jmp checkCPUtype
3531:
354#endif /* CONFIG_SMP */
```

346、347 行初始化 CPU 的 eflags 寄存器，对 popf 指令不熟的人可以重新学学汇编了。继续走：

```
355
356/*
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
361
```

360 行，call setup_idt 这里为每一个中断准备最早的处理函数，跳过 checkCPUtype 的若干行相关代码，我们来到 502 行：

```
490/*
491 *    setup_idt
492 *
493 *    sets up a idt with 256 entries pointing to
494 *    ignore_int, interrupt gates. It doesn't actually load
495 *    idt - that can be done only after paging has been enabled
496 *    and the kernel moved to PAGE_OFFSET. Interrupts
497 *    are enabled elsewhere, when we can be relatively
498 *    sure everything is ok.
499 *
500 *    Warning: %esi is live across this function.
501 */
502setup_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 行，获得 ignore_int 的地址，ignore_int 在 575 行定义：

```
575ignore_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
```

以上 575~605 这么一堆代码，就是 ignore_int 函数，把函数的入口地址赋给 edx 寄存器。函数的内容我们不详细讲解了，这里只简单说说，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"
```

回到 setup_idt 中，504 行，重要的__KERNEL_CS 登场了。这个宏是内核代码段的选择子，其定义在 arch/x86/include/asm/segment.h 的 185 行：

```
185#define __KERNEL_CS         (GDT_ENTRY_KERNEL_CS * 8)
186#define __KERNEL_DS         (GDT_ENTRY_KERNEL_DS * 8)
```

```
187#define __USER_DS       (GDT_ENTRY_DEFAULT_USER_DS* 8 + 3)
188#define __USER_CS       (GDT_ENTRY_DEFAULT_USER_CS* 8 + 3)
```

GDT_ENTRY_KERNEL_CS 定义在同一个文件的 149 行，其值为 2，那么__KERNEL_CS 的值就是 0x10，那么 setup_idt 中 movl $(__KERNEL_CS << 16),%eax 之后，eax 寄存器中的值就是 0x100000。150 行，将 ignore_int 地址的低 16 位赋给 ax，这样，eax 寄存器中的低 16 就是 ignore_int 地址的低 16 位，高 16 位就是 0x0010。

继续走，%dx 的值又被赋成了$0x8E00，暂不管他。

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

508 行，edi 寄存器中存放 idt_table 表的地址。这个表就是中断门描述符表，其在我们的 arch/x86/include/asm/desc.h 文件中定义：

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

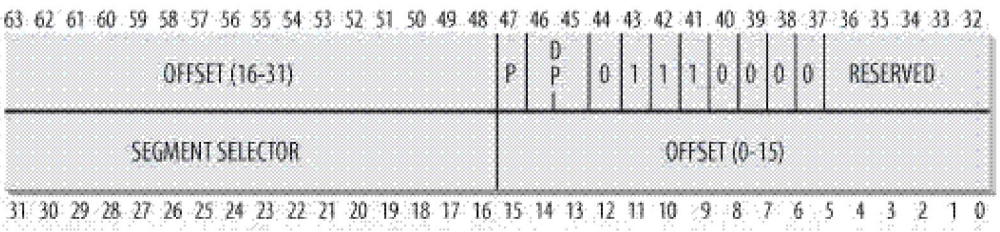
中断门描述符表是一个数组，每一个元素都是一个中断门描述符 gate_desc，其在文件 arch/x86/include/asm/desc_defs.h 中被定义成：

```
81 typedef struct gate_struct64 gate_desc;
```

从名字上就能看出，这个门描述符占据 8 个字节，64 位，在同一个文件中定义：

```
51struct gate_struct64 {
52              u16 offset_low;
53              u16 segment;
54              unsigned ist : 3, zero0 : 5, type : 5, dpl : 2, p : 1;
55              u16 offset_middle;
56              u32 offset_high;
57              u32 zero1;
58} __attribute__((packed));
```

中断门描述符



这里面 segment 是段选择子，其他字段如上图所示。idt_table 就是包含若干这样的门描述符的数组表，每个表项 8 个字节，表头被 edi 寄存器指向。

```
510rp_sidt:
```

```
511             movl %eax,(%edi)
512             movl %edx,4(%edi)
513             addl $8,%edi
514             dec %ecx
515             jne rp_sidt
516
```

随后，511、512 行，中断门描述符表的第一项前 2 个字节被设置成 ignore_int 地址的低 16 位作为偏移，中间 2 个字节被设置成段选择子 0x0010，后 4 个字节被设置成$0x8E00。随后 513~515 行设置 256 个这样的门描述符，每个门描述符内容一样，都是调用函数 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
```

上面代码改变几个中断/异常处理函数，把 0/6/13/14 号中断描述符表项设置成相应的处理函数，而 527~530 就是这些函数：early_divide_err、early_illegal_opcode、early_protection_fault、early_page_fault。这些函数又都是调用的 early_fault 函数：

```
552early_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
569hlt_loop:
570             hlt
571             jmp hlt_loop
572
573/* This is the default interrupt "handler" :-) */
574             ALIGN
```

主要是打印 fault_msg 信息：

```
669fault_msg:
670/* fault info: */
671             .ascii "BUG: Int %d: CR2 %p\n"
672/* pusha regs: */
673             .ascii "        EDI %p    ESI %p    EBP %p    ESP %p\n"
674             .ascii "        EBX %p    EDX %p    ECX %p    EAX %p\n"
675/* fault frame: */
676             .ascii "        err %p    EIP %p      CS %p    flg %p\n"
677             .ascii "Stack: %p %p %p %p %p %p %p %p\n"
678             .ascii "            %p %p %p %p %p %p %p %p\n"
679             .asciz "            %p %p %p %p %p %p %p %p\n"
```

当然，如果没有配置打印选项 CONFIG_PRINTK，就关机（当然不可能没有设置）。

4.2.2 第三次启动保护模式

很多人就有疑问， 刚才我们设置好了中断描述符表，那内核怎么用它呢？不好意思，刚才在略过了的 checkCPUtype 过程中有一步非常重要的步骤，现在把它补上：

```
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
426         lidt idt_descr
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
```

425 行重新加载一个全局描述符表，early_gdt_descr：

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

同样也是包含段限和全局描述符表地址，gdt_page 在 arch/x86/include/asm/desc.h 定义：

```
 35
 36struct gdt_page {
 37          struct desc_struct gdt[GDT_ENTRIES];
 38} __attribute__((aligned(PAGE_SIZE)));
```

为什么需要重新加载一个全局描述符表？很简单，因为现在起到分页了，所以，要重新定位全局描述符表的位置，GDT 的第三次段设置是在开启并设置了了页面寻址之后进行的。设置本身很简单，也是最终的设置，而且还同时设置了 IDT。而由于 aligned(PAGE_SIZE)，所以 gdt_page 正好是一个页面的开始位置。GDT_ENTRIES 的值是 32，定义在 arch/x86/include/asm/segment.h：

```
110#define GDT_ENTRIES 32
```

所以全局描述符表式每个表项为 desc_struct 结构的，32 个表项的数组，它的每个表项是：

```
 22struct 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));
```

正好看到第二个联合体，熟悉吧，描述符结构。哦，你不熟悉，那就看看博客“ Intel 80286 工作模式” http://blog.csdn.net/yunsongice/archive/2010/10/04/5920447.aspx。8 个字节，所以 gdt 表共占 8*32=256 个字节，相对于一个页面的 4096 字节，仅仅占了一个页面前 1/16 的空间。

回到 426 行，把我们刚才设置好的中断门描述符表 idt_descr 通过 lidt 加载上，然后 427~436 行分别将段选择子__KERNEL_CS、__KERNEL_DS、__USER_DS、__KERNEL_PERCPU 加载到寄存器 cs、ss、ds/es、fs。注意这里一个细节，加载 cs 寄存器重来都不使用 mov 指令，而是通过 ljmp $(__KERNEL_CS),$1f 来设置。还有，这时候，gdt 是早已初始化了的。

那么，这个 gdt 是在哪里初始化的呢，请看 arch/x86/kernel/cpu/common.c 文件：

```
86DEFINE_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} };
```

首先讲讲 DEFINE_PER_CPU_PAGE_ALIGNED 宏，在 include/linux/percpu-defs.h 中：

```
#define DEFINE_PER_CPU_PAGE_ALIGNED(type, name)                         \
    DEFINE_PER_CPU_SECTION(type, name, ".page_aligned")          \
    __aligned(PAGE_SIZE)

#define DECLARE_PER_CPU_SECTION(type, name, sec)                \
    extern __PCPU_ATTRS(sec) __typeof__(type) name

#define __PCPU_ATTRS(sec)                                       \
    __percpu __attribute__((section(PER_CPU_BASE_SECTION sec))) \
    PER_CPU_ATTRIBUTES
```

展开就是：

```
    extern __percpu __attribute__((section(PER_CPU_BASE_SECTION .page_aligned)))    \
    PER_CPU_ATTRIBUTES __typeof__(struct gdt_page) gdt_page __aligned(PAGE_SIZE)
```

看到了吗？在编译阶段 gdt[GDT_ENTRIES]就被编译成了一个常量，在 32 位 x86 体系中其值为 103~135 行的代码，将这个常量最后链接进 PER_CPU_BASE_SECTION 中。

另外，GDT_ENTRY_INIT 宏，在 arch/x86/include/asm/desc_defs.h 中定义：

```
#define GDT_ENTRY_INIT(flags, base, limit) { { { \
        .a = ((limit) & 0xffff) | (((base) & 0xffff) << 16), \
```

```
.b = (((base) & 0xff0000) >> 16) | (((flags) & 0xf0ff) << 8) | \
    ((limit) & 0xf0000) | ((base) & 0xff000000), \
```

所以经过编译链接以后，gdt 就包含 14 个全局描述符，其中：

#define GDT_ENTRY_KERNEL_BASE 12

#define GDT_ENTRY_KERNEL_CS (GDT_ENTRY_KERNEL_BASE + 0)

表示内核代码段的索引，即 gdt[12]为代码段描述符。

#define GDT_ENTRY_KERNEL_DS (GDT_ENTRY_KERNEL_BASE + 1)

表示内核数据段的索引，即 gdt[13]为数据段描述符。

#define GDT_ENTRY_DEFAULT_USER_CS 14

表示用户代码段的索引，即 gdt[14]为用户代码段描述符。

#define GDT_ENTRY_DEFAULT_USER_CS 14

表示用户数据段的索引，即 gdt[15]为用户数据段描述符。

#define GDT_ENTRY_PNPBIOS_BASE (GDT_ENTRY_KERNEL_BASE + 6)

#define GDT_ENTRY_PNPBIOS_CS32 (GDT_ENTRY_PNPBIOS_BASE + 0)

#define GDT_ENTRY_PNPBIOS_CS16 (GDT_ENTRY_PNPBIOS_BASE + 1)

#define GDT_ENTRY_PNPBIOS_DS (GDT_ENTRY_PNPBIOS_BASE + 2)

#define GDT_ENTRY_PNPBIOS_TS1 (GDT_ENTRY_PNPBIOS_BASE + 3)

#define GDT_ENTRY_PNPBIOS_TS2 (GDT_ENTRY_PNPBIOS_BASE + 4)

gdt[18]、gdt[19]、gdt[20]、gdt[21]、gdt[22]分别表示 PNP BIOS 的 32 位代码段、16 位代码段、数据段、两个任务段。

#define GDT_ENTRY_APMBIOS_BASE (GDT_ENTRY_KERNEL_BASE + 11)

gdt[23]、gdt[24]、gdt[25]分别表示高级电源管理 APM 的 32 位代码段、16 位代码段和数据段。

#define GDT_ENTRY_ESPFIX_SS (GDT_ENTRY_KERNEL_BASE + 14)

gdt[26]表示堆栈修复段的段描述符。

#define GDT_ENTRY_PERCPU (GDT_ENTRY_KERNEL_BASE + 15)

gdt[26]表示每 CPU 段的段描述符。

最后由于我们的.config 没有定义 CONFIG_CC_STACKPROTECTOR，所以 GDT_STACK_CANARY_INIT 就为空。所以，Linux 启动以来自此加载的 gdt 已有以上若干个段的描述符在编译 vmlinux 的时候初始化了，其他没被初始化的地方暂时保留。

我们知道 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。

4.2.3 启动 x86 虚拟机

看到 arch/x86/kernel/head_32.S 的 681 行：
681 #include "../../x86/xen/xen-head.S"

这时候执行 arch/x86/xen/xen-head.S 下的代码：

```
19#ifdef CONFIG_X86_32
20          mov %esi,xen_start_info
21          mov $init_thread_union+THREAD_SIZE,%esp
22#else
23          mov %rsi,xen_start_info
24          mov $init_thread_union+THREAD_SIZE,%rsp
25#endif
26          jmp xen_start_kernel
```

该文件中有用的就上面几行代码，也就是执行一个 xen_start_kernel 函数，这个函数的局部变量存放的位置通过重置栈顶指针来设定，更重要的是设计了一个全局变量 xen_start_info，其值就是 esi 的内容。esi 可一直没有变过哈，是前面我们研究了很久的 boot_params 的地址。所以 xen_start_info 就是执行它的指针。

```
1057/* First C function to be called on Xen boot */
1058asmlinkage void __init xen_start_kernel(void)
1059{
1060          pgd_t *pgd;
1061
……
1201
1202          /* Start the world */
1203#ifdef CONFIG_X86_32
1204          i386_start_kernel();
1205#else
1206          x86_64_start_reservations((char *)__pa_symbol(&boot_params));
1207#endif
1208}
```

整个 xen_start_kernel 代码是针对 Xen 的启动代码。这里简单介绍一下 Xen，Xen 是一个开源的 x86 体系的 para-virtualizing 虚拟机监控器（VMM）或者“系统管理程序(hypervisor)”。我们熟悉的 Vmware 就是以它为基础的。Xen 可以安全地在一个物理系统上以接近于本地硬件的性能运行多个虚拟机。Xen 具有方便应用的企业级功能，包括：

- 虚拟机的性能接近于本地硬件.
- 在物理机器之间活动迁移正在运行的虚拟机.
- 在每个客户虚拟机支持到 32 个虚拟 CPU，虚拟 CPU(VCPU)热插拔.

- x86/32, 物理地址扩展(PAE)的 x86/32 , 和 x86/64 平台支持.
- 不用修改客户操作系统(包括微软的 Windows)就能支持因特尔虚拟化技术(VT-x).
- 良好的硬件支持(支持几乎所有的 Linux 设备驱动).

所以，xen_start_kernel 函数完成了 Xen 的初始化后，就跳到 1204 行，执行 i386_start_kernel 函数。该函数来自 arch/x86/kernel/head32.c：

```
  31void __init i386_start_kernel(void)
  32{
  33#ifdef CONFIG_X86_TRAMPOLINE
  34            /*
  35             * But first pinch a few for the stack/trampoline stuff
  36             * FIXME: Don't need the extra page at 4K, but need to fix
  37             * trampoline before removing it. (see the GDT stuff)
  38             */
  39            reserve_early_overlap_ok(PAGE_SIZE, PAGE_SIZE + PAGE_SIZE,
  40                                                    "EX TRAMPOLINE");
  41#endif
  42
  43            reserve_early(__pa_symbol(&_text), __pa_symbol(&__bss_stop), "TEXT DATA
BSS");
  44
  45#ifdef CONFIG_BLK_DEV_INITRD
  46            /* Reserve INITRD */
  47            if (boot_params.hdr.type_of_loader && boot_params.hdr.ramdisk_image) {
  48                        /* Assume only end is not page aligned */
  49                        u64 ramdisk_image = boot_params.hdr.ramdisk_image;
  50                        u64 ramdisk_size    = boot_params.hdr.ramdisk_size;
  51                        u64 ramdisk_end     = PAGE_ALIGN(ramdisk_image + ramdisk_size);
  52                        reserve_early(ramdisk_image, ramdisk_end, "RAMDISK");
  53            }
  54#endif
  55
  56            /* Call the subarch specific early setup function */
  57            switch (boot_params.hdr.hardware_subarch) {
  58            case X86_SUBARCH_MRST:
  59                        x86_mrst_early_setup();
  60                        break;
  61            default:
  62                        i386_default_early_setup();
  63                        break;
  64            }
  65
  66            /*
```

```
67              * At this point everything still needed from the boot loader
68              * or BIOS or kernel text should be early reserved or marked not
69              * RAM in e820. All other memory is free game.
70              */
71
72             start_kernel();
73}
```

由于.config 中配置了 CONFIG_X86_TRAMPOLINE，所以首先调用 reserve_early_overlap_ok 函数。这个函数和下面 reserve_early 一起用于保留一些重叠的空间，他们都会调用 __reserve_early。具体的我就不去分析了，感兴趣的同学自己去研究。

47~53 行，也是调用__reserve_early 函数去保留初始化头中的 ramdisk 的相关信息。57 行，我们可以回头看看 header.S 的 hdr 中的 hardware_subarch 是 0，所以直接去调用位于同一文件的 i386_default_early_setup 函数：

```
static void __init i386_default_early_setup(void)
{
    /* Initilize 32bit specific setup functions */
    x86_init.resources.probe_roms = probe_roms;
    x86_init.resources.reserve_resources = i386_reserve_resources;
    x86_init.mpparse.setup_ioapic_ids = setup_ioapic_ids_from_mpc;

    reserve_ebda_region();
}
```

这个函数就是把虚拟机的 x86_init 全局变量的一些安装函数设置好，不在话下。以上的函数完成以后，Xen 的初始化工作就完了，在 i386_start_kernel 的最后一行调用 start_kernel()函数，终于到了 Modern Age。

5 走向现代：start_kernel 函数

这是内核初始化至此，终于跑出 arch/x86 目录了，它在 linux/init/main.c，以后的代码除了少数特例，其余的都在 linux/init/目录中。

```
528asmlinkage void __init start_kernel(void)
529{
530             char * command_line;
531             extern struct kernel_param __start___param[], __stop___param[];
532
533             smp_setup_processor_id();
534
535             /*
536              * Need to run as early as possible, to initialize the
537              * lockdep hash:
538              */
539             lockdep_init();
540             debug_objects_early_init();
541
542             /*
543              * Set up the the initial canary ASAP:
544              */
545             boot_init_stack_canary();
546
547             cgroup_init_early();
548
549             local_irq_disable();
550             early_boot_irqs_off();
551             early_init_irq_lock_class();
552
553/*
554 * Interrupts are still disabled. Do necessary setups, then
555 * enable them
556 */
557             lock_kernel();
558             tick_init();
559             boot_cpu_init();
560             page_address_init();
561             printk(KERN_NOTICE "%s", linux_banner);
562             setup_arch(&command_line);
563             mm_init_owner(&init_mm, &init_task);
```

```
564         setup_command_line(command_line);
565         setup_nr_cpu_ids();
566         setup_per_cpu_areas();
567         smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */
568
569         build_all_zonelists();
570         page_alloc_init();
571
572         printk(KERN_NOTICE "Kernel command line: %s\n", boot_command_line);
573         parse_early_param();
574         parse_args("Booting kernel", static_command_line, __start___param,
575                    __stop___param - __start___param,
576                    &unknown_bootoption);
577         /*
578          * These use large bootmem allocations and must precede
579          * kmem_cache_init()
580          */
581         pidhash_init();
582         vfs_caches_init_early();
583         sort_main_extable();
584         trap_init();
585         mm_init();
586         /*
587          * Set up the scheduler prior starting any interrupts (such as the
588          * timer interrupt). Full topology setup happens at smp_init()
589          * time - but meanwhile we still have a functioning scheduler.
590          */
591         sched_init();
592         /*
593          * Disable preemption - early bootup scheduling is extremely
594          * fragile until we cpu_idle() for the first time.
595          */
596         preempt_disable();
597         if (!irqs_disabled()) {
598                 printk(KERN_WARNING "start_kernel(): bug: interrupts were "
599                                 "enabled *very* early, fixing it\n");
600                 local_irq_disable();
601         }
602         rcu_init();
603         radix_tree_init();
604         /* init some links before init_ISA_irqs() */
605         early_irq_init();
606         init_IRQ();
607         prio_tree_init();
```

```
608         init_timers();
609         hrtimers_init();
610         softirq_init();
611         timekeeping_init();
612         time_init();
613         profile_init();
614         if (!irqs_disabled())
615                 printk(KERN_CRIT "start_kernel(): bug: interrupts were "
616                                 "enabled early\n");
617         early_boot_irqs_on();
618         local_irq_enable();
619
620         /* Interrupts are enabled now so all GFP allocations are safe. */
621         gfp_allowed_mask = __GFP_BITS_MASK;
622
623         kmem_cache_init_late();
624
625         /*
626          * HACK ALERT! This is early. We're enabling the console before
627          * we've done PCI setups etc, and console_init() must be aware of
628          * this. But we do want output early, in case something goes wrong.
629          */
630         console_init();
631         if (panic_later)
632                 panic(panic_later, panic_param);
633
634         lockdep_info();
635
636         /*
637          * Need to run this when irqs are enabled, because it wants
638          * to self-test [hard/soft]-irqs on/off lock inversion bugs
639          * too:
640          */
641         locking_selftest();
642
643#ifdef CONFIG_BLK_DEV_INITRD
644         if (initrd_start && !initrd_below_start_ok &&
645             page_to_pfn(virt_to_page((void *)initrd_start)) < min_low_pfn) {
646                 printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
647                         "disabling it.\n",
648                         page_to_pfn(virt_to_page((void *)initrd_start)),
649                         min_low_pfn);
650                 initrd_start = 0;
651         }
```

```
652#endif
653             page_cgroup_init();
654             enable_debug_pagealloc();
655             kmemtrace_init();
656             kmemleak_init();
657             debug_objects_mem_init();
658             idr_init_cache();
659             setup_per_cpu_pageset();
660             numa_policy_init();
661             if (late_time_init)
662                     late_time_init();
663             sched_clock_init();
664             calibrate_delay();
665             pidmap_init();
666             anon_vma_init();
667#ifdef CONFIG_X86
668             if (efi_enabled)
669                     efi_enter_virtual_mode();
670#endif
671             thread_info_cache_init();
672             cred_init();
673             fork_init(totalram_pages);
674             proc_caches_init();
675             buffer_init();
676             key_init();
677             security_init();
678             vfs_caches_init(totalram_pages);
679             signals_init();
680             /* rootfs populating might need page-writeback */
681             page_writeback_init();
682#ifdef CONFIG_PROC_FS
683             proc_root_init();
684#endif
685             cgroup_init();
686             cpuset_init();
687             taskstats_init_early();
688             delayacct_init();
689
690             check_bugs();
691
692             acpi_early_init(); /* before LAPIC and SMP init */
693             sfi_init_late();
694
695             ftrace_init();
```

```
696
697             /* Do the rest non-__init'ed, we're now alive */
698             rest_init();
699}
```

首先，533 行，执行 smp_setup_processor_id()函数。对于我们 x86 体系，这个函数是个空函数。继续走，539 行，调用 lockdep_init()函数，这个函数来自 kernel/Lockdep.c：

5.1 初始化同步与互斥环境

要了解本节的内容，需要补充一下“ 疯狂内核之同步与互斥 ”的预备知识：
http://blog.csdn.net/yunsongice/category/661010.aspx

5.1.1 屏蔽中断

```
void lockdep_init(void)
{
        int i;

        /*
          * Some architectures have their own start_kernel()
          * code which calls lockdep_init(), while we also
          * call lockdep_init() from the start_kernel() itself,
          * and we want to initialize the hashes only once:
          */
        if (lockdep_initialized)
                return;

        for (i = 0; i < CLASSHASH_SIZE; i++)
                INIT_LIST_HEAD(classhash_table + i);

        for (i = 0; i < CHAINHASH_SIZE; i++)
                INIT_LIST_HEAD(chainhash_table + i);

        lockdep_initialized = 1;
}
```

该函数用于启动 Lock dependency validator，本质上是建立两个散列表 classhash_table 和 chainhash_table，并初始化全局变量 lockdep_initialized。对 Linux 内核的链表和散列表不熟悉的同志可以看看博客“ Linux 内核入门（三）—— C 语言基本功 ”

http://blog.csdn.net/yunsongice/archive/2010/04/10/5471096.aspx
的相关内容。

好了，继续跟进，540 行，debug_objects_early_init()函数，用于内核的对象调试。由于我们.config 文件中没有配置 CONFIG_DEBUG_OBJECTS，所以这个函数为空函数。545 行，调用 boot_init_stack_canary() 函数，同样在也.config 文件中没有配置 CONFIG_CC_STACKPROTECTOR，所以这个函数也为空。547 行，CONFIG_CGROUPS 也没有配置，所以 cgroup_init_early()函数也为空。

549 行，local_irq_disable()，终于不为空了，在 include/linux/irqflags.h 的 61 行定义：

```
#define local_irq_disable() \
	do { raw_local_irq_disable(); trace_hardirqs_off(); } while (0)
```

注意，这个定义必须是 CONFIG_TRACE_IRQFLAGS_SUPPORT 编译选项存在于.config 文件，该宏才有效。我查了查.config，是存在的，所以又定位到 raw_local_irq_disable 和 trace_hardirqs_off 两个宏。先看第一个，raw_local_irq_disable，全内核四个地方有定义，取决于一个重要的 CONFIG_PARAVIRT 编译选项是否被配置。我们看到.config 中根本不存在这个编译选项，所以根本不用考虑 arch/x86/include/asm/paravirt.h 里面的内容。而剩下的定义来自哪儿，取决于宏__ASSEMBLY__。

是该讲讲__ASSEMBLY__宏了，这个宏的作用是避免 gcc 在进行汇编编译的时候，不定义后续相关内容。什么意思？我们看到顶层 Makefile 的 354 行有一个：
354 KBUILD_AFLAGS	:= -D__ASSEMBLY__

这个宏变量实际上是在编译汇编代码的时候，由编译器使用-D 这样的参数加进去的，KBUILD_AFLAGS 这个变量也定义了这个变量，gcc 会把这个宏定义为 1。我们知道，gcc 编译一个 c 程序大致分为三步：编译 c 文件生成临时汇编程序；编译汇编程序生成.o 对象文件；链接对象文件生成 c 程序。而 KBUILD_AFLAGS 指定__ASSEMBLY__，是为了编译汇编程序时，不会用到类似于“ # define ”定义的属性（注意！不是宏定义#define，#后面有一个空格，Linux 内核中定义了很多这样的定义，在编译 c 文件时，就相当于#define），因为这样的属性是在进行 C 语言编译的时候加的，通过__ASSEMBLY__来避免在 C 语言编译成汇编程序后，不在编译这些汇编代码时候的引用，从而避免不必要的宏或函数在编译汇编代码时候的引用。

虽然我们顶层 Makefile 设置了__ASSEMBLY__，但是，我们现在分析的是 c 语言下的代码，__ASSEMBLY__暂没定义，所以来到 arch/x86/include/asm/irqflags.h 中 74 行：

```
static inline void raw_local_irq_disable(void)
{
	native_irq_disable();
}
```

native_irq_disable 来自哪儿？同一文件的 37 行：

```
static inline void native_irq_disable(void)
{
	asm volatile("cli": : :"memory");
}
```

调用 cli 指令取消 eflags 的 IF 标志位，屏蔽可屏蔽中断。有关可屏蔽中断、非屏蔽中断和异常的基本概念请参考博文“ 中断的分类 ”
http://blog.csdn.net/yunsongice/archive/2010/02/11/5306134.aspx

后面由于我们没有设置 CONFIG_TRACE_IRQFLAGS 和 CONFIG_IRQSOFF_TRACER 编译选项，所以 trace_hardirqs_off()函数为空，回到 start_kernel 中。

上面说了，没有配置 CONFIG_TRACE_IRQFLAGS，所以 550 行的 early_boot_irqs_off()函数也为空。而 CONFIG_GENERIC_HARDIRQS 是配置了的，所以 551 行 early_init_irq_lock_class 函数就要被调用了，来自 kernel/irq/handle.c 的 543 行：

```
void early_init_irq_lock_class(void)
{
	struct irq_desc *desc;
	int i;

	for_each_irq_desc(i, desc) {
		lockdep_set_class(&desc->lock, &irq_desc_lock_class);
	}
}
```

for_each_irq_desc 是一个宏，我们设置了 CONFIG_GENERIC_HARDIRQS，所以来自 include/linux/irqnr.h 的 29 行：

```
# define for_each_irq_desc(irq, desc)                    \
	for (irq = 0, desc = irq_to_desc(irq); irq < nr_irqs;          \
	     irq++, desc = irq_to_desc(irq))                   \
		if (!desc)                                 \
			;                                  \
		else
```

我们没有配置 CONFIG_SPARSE_IRQ，所以来到 kernel/irq/handle.c 的 275 行：

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

好了，至此，最著名的 NR_IRQS 变量就出现了。要说清楚这个问题，就要大致学学 APIC 的知识了。APIC 称为高级可编程中断控制器（Advanced Programmable Interrupt Controller，APIC），是通过扩充组合的方式来为多处理器系统驱动中断控制器。在目前的 x86 体系中，系统的每一个部份都是经由 APIC 总线连接的。“ 本地 APIC ”为系统的一部份，负责传递 Interrupt 至指定的处理器。

举例来说，当一台机器上有一个四核处理器，则它必须相对的要有四个本地 APIC。自 1994 年的 Pentium P54c 开始 Intel 已经将本地 APIC 内嵌在它们的处理器中。我们当前的 Intel 处理器的电脑就已经包含了 APIC 系统的部份。

系统中另一个重要的部份为 I/O APIC。系统中最多可拥有 8 个 I/O APIC。它们会收集来自 I/O 装置的 Interrupt 讯号且在当那些装置需要 interrupt 时传送讯息至本地 APIC。每个 I/O APIC 有一个专有的 interrupt 输入号码，就是我们所谓的 IRQ 号。Intel 过去与目前的 I/O APIC 通常有 NR_IRQS 个输入，具体多少个我们稍后分析。而且，有些机器拥有数个 I/O APIC，每一个分别有自己的输入号码，加起来一台机器上会有上百个 IRQ 可供装置 Interrupt 使用。

然而，系统中若没有 I/O APIC，那本机 APIC 就没有用处。像这样的状况下，编译配置中的 CONFIG_X86_IO_APIC 编译选项就不会设置，还会还原使用 8259 PIC，仅仅拥有 16 个 IRQ，其中一个还用来级联。

好了，有关 APIC 的知识我们就暂时介绍到这里，想深入了解的请查看博客“ 中断处理 ” http://blog.csdn.net/yunsongice/archive/2010/03/07/5353715.aspx。由于我们配置了 IO APIC 但没有配置 CONFIG_SPARSE_IRQ 编译选项，所以 NR_IRQS 就定义在 arch/x86/include/asm/irq_vectors.h 的 166 行：

```
#   define NR_IRQS                         \
     (CPU_VECTOR_LIMIT < IO_APIC_VECTOR_LIMIT ? \
          (NR_VECTORS + CPU_VECTOR_LIMIT)  : \
          (NR_VECTORS + IO_APIC_VECTOR_LIMIT))
```

NR_VECTORS 的值是 256，CPU_VECTOR_LIMIT 的值等于 CONFIG_NR_CPUS，我们配置文件中的值是 255，IO_APIC_VECTOR_LIMIT 被定义为(32 * MAX_IO_APICS)，表示 Linux 支持的追到 IO APIC 的数量。MAX_IO_APICS 在 32 位 x86 体系中被定义为 64。因此 CPU_VECTOR_LIMIT 超过了最大限额，NR_IRQS 的值就是 256+64=320。

所以，for_each_irq_desc 宏的意思就是遍历每个 IO APICS 的 IRQ，并调用 lockdep_set_class 宏。由于我们没有 CONFIG_LOCKDEP，所以这个宏是个空宏，没有任何内容。是不是觉得前面说了这么多，但没有实际的代码，有点灰心？没关系，毕竟我们学到新东西了。如果配置了 CONFIG_LOCKDEP，那么 lockdep_set_class 宏定义如下：

```
#define lockdep_set_class(lock, key) \
        lockdep_init_map(&(lock)->dep_map, #key, key, 0)
```

那么上面遍历了 IO APICS 每个中断的中段描述符，并设置中段描述符的锁。

5.1.2 启动大内核锁

回到 start_kernel，557 行，lock_kernel()，实际的代码来了，大内核锁。我们在.config 文件中配置了 CONFIG_LOCK_KERNEL 的，所以这个函数是 start_kernel 中继 local_irq_disable 之后实际执行到的第二个函数。

有关大内核的知识，这里又简单的介绍一下。在早期的 Linux 内核版本中，大内核锁（big kernel block，也叫全局内核锁或 BKL）被广泛使用。在 2.0 版本中，这个锁是相对粗粒度的自旋锁，确保每次只有一个进程能运行在内核态。2.2 和 2.4 内核具有极大的灵活性，不再依赖一个单独的自旋锁，而是由许多不同的自旋锁保护大量的内核数据结构。在 Linux 2.6 版本的内核中，用大内核锁来裸护旧的代码（绝大多数主要是与 VFS 和几个文件系统相关的函数）。

从内核版本 2.6.11 开始，用一个叫做 kernel_sem 的信号量来实现大内核锁（在较早的 2.6 版本中，大内核锁是通过自旋锁来实现的）。但是，大内核锁比简单的信号量要复杂一些。

每个进程描述符 task_struct 都含有 lock_depth 字段，其就是一个整形变量，这个字段允许同一个进程几次获取大内核锁。因此，对大内核锁两次连续的请求不挂起处理器（相对于普通自旋锁）。如果进程未获的过锁，则这个字段的值为-1；否则，这个字段的值加 1，表示已经请求了多少次锁。lock_depth 字段对中断处理程序、异常处理程序及可延迟函数获取大内核锁都是至关重要的。如果没有这个字段，那么，在当前进程已经拥有大内核锁的情况下，任何试图获得这个锁的异步函数都可能产生死锁。

lock_kernel ()和 unlock_kernel()内核函数用来获得和释放大内核锁。前一个函数等价于：

```
depth = current->lock_depth + 1;
if (depth == 0)
    down(&kernel_sem);
current->lock_depth = depth;
```

而后者等价于

```
if (--current->lock_depth < 0)
    up(&kernel_sem);
```

注意，lock_kernel()和 unlock_kernel()函数的 if 语句不需要原子地执行，因为 lock_depth 不是全局变量——这是每个 CPU 在自己当前进程描述符中访问的一个字段。在 if 语句确本地中断也不会引起竞争条件。即使新内核控制路径调用了 lock_kernel()，它在终止前也必须释放大内核锁。

对内核同步与互斥机制还不熟悉的同学，请查看博客“同步与互斥的基本原理”
http://blog.csdn.net/yunsongice/archive/2010/03/11/5370732.aspx

于是乎，后面肯定有会一个 unlock_kernel 函数跟 557 行的 lock_kernel 配对，咱们拭目以待。

5.1.3 注册时钟通知链

由于我们在.config 文件中设置了 CONFIG_GENERIC_CLOCKEVENTS，所以，start_kernel 函数的第 558 行调用 kernel/time/tick-common.c 中的 tick_init 函数，来注册一个时钟事件监听器。tick_init 只调用一个 clockevents_register_notifier 函数：

```
static struct notifier_block tick_notifier = {
	.notifier_call = tick_notify,
};

void __init tick_init(void)
{
	clockevents_register_notifier(&tick_notifier);
}

int clockevents_register_notifier(struct notifier_block *nb)
{
	unsigned long flags;
	int ret;

	raw_spin_lock_irqsave(&clockevents_lock, flags);
	ret = raw_notifier_chain_register(&clockevents_chain, nb);
	raw_spin_unlock_irqrestore(&clockevents_lock, flags);

	return ret;
}
```

这个函数很简单，只要先明白一个 notifier_block 结构就行了：

```
struct notifier_block {
	int (*notifier_call)(struct notifier_block *, unsigned long, void *);
	struct notifier_block *next;
	int priority;
};
```

tick_notifier 全局变量在编译 vmlinux 的时候其 notifier_call 被设置成了 tick_notify 函数：

```
static int tick_notify(struct notifier_block *nb, unsigned long reason,
                       void *dev)
{
	switch (reason) {
```

```
        case CLOCK_EVT_NOTIFY_ADD:
                return tick_check_new_device(dev);

        case CLOCK_EVT_NOTIFY_BROADCAST_ON:
        case CLOCK_EVT_NOTIFY_BROADCAST_OFF:
        case CLOCK_EVT_NOTIFY_BROADCAST_FORCE:
                tick_broadcast_on_off(reason, dev);
                break;

        case CLOCK_EVT_NOTIFY_BROADCAST_ENTER:
        case CLOCK_EVT_NOTIFY_BROADCAST_EXIT:
                tick_broadcast_oneshot_control(reason);
                break;

        case CLOCK_EVT_NOTIFY_CPU_DYING:
                tick_handover_do_timer(dev);
                break;

        case CLOCK_EVT_NOTIFY_CPU_DEAD:
                tick_shutdown_broadcast_oneshot(dev);
                tick_shutdown_broadcast(dev);
                tick_shutdown(dev);
                break;

        case CLOCK_EVT_NOTIFY_SUSPEND:
                tick_suspend();
                tick_suspend_broadcast();
                break;

        case CLOCK_EVT_NOTIFY_RESUME:
                tick_resume();
                break;

        default:
                break;
        }

        return NOTIFY_OK;
}
```

这上面一串函数是什么意思？这里要学一个新知识——通知链技术。我们知道内核一些驱动或文件系统是被编译成模块了的，当某个模块的某些事件对其模块是很有用的，则本模块应该定义一个通知链，并导出接口。在 Linux 内核中，各个模块之间有很强的相互关系，一些被一个子系统生成或者被探测到的事件，很可能是另一个或者多个子系统感兴趣的，也就

是说这个事件的获取者必须能够通知所有对该事件感兴趣的子系统，并且还需要这种通知机制具有一定的通用性。基于这些， Linux 内核引入了“ 通知链 ”技术。

通知链有四种类型，

原子通知链（ Atomic notifier chains ）：通知链元素的回调函数（当事件发生时要执行的函数）只能在中断上下文中运行，不允许阻塞
可阻塞通知链（ Blocking notifier chains ）：通知链元素的回调函数在进程上下文中运行，允许阻塞
原始通知链（ Raw notifier chains ）：对通知链元素的回调函数没有任何限制，所有锁和保护机制都由调用者维护
SRCU 通知链（ SRCU notifier chains ）：可阻塞通知链的一种变体所以对应了四种通知链头结构：
struct atomic_notifier_head ：原子通知链的链头
struct blocking_notifier_head ：可阻塞通知链的链头
struct raw_notifier_head ：原始通知链的链头
struct srcu_notifier_head ： SRCU 通知链的链头

通知链元素的类型：
struct notifier_block ：通知链中的元素，记录了当发出通知时，应该执行的操作（即回调函数）
链头中保存着指向元素链表的指针。通知链元素结构则保存着回调函数的类型以及优先级，参见 notifier.h 文件。

那么其它模块对其事件感兴趣时，可以调用这些接口注册（注销）当事件发生时的行为。而本模块在事件发生时，调用通知其链上的所有订阅者（调用其函数）。

而这个 notifier_block 就是通知链中的元素，记录了当发出通知时，应该执行的操作（即回调函数）。这里 tick_notifier 是一个时钟事件监听器模块，当发出通知是，就执行 tick_notify 函数。

所以，clockevents_register_notifier 函数就调用 raw_notifier_chain_register 来注册，其实就是把三步：获得自旋锁、把 tick_notifier 加入 clockevents_chain、释放自旋锁。

5.1.4 激活第一个 CPU

回到 start_kernel，559 行，boot_cpu_init 函数，跟 start_kernel 位于同一文件：

```
494static void __init boot_cpu_init(void)
495{
496         int cpu = smp_processor_id();
497         /* Mark the boot cpu "present", "online" etc for SMP and UP case */
498         set_cpu_online(cpu, true);
```

```
499         set_cpu_active(cpu, true);
500         set_cpu_present(cpu, true);
501         set_cpu_possible(cpu, true);
502}
```

496 行，第一次见到 smp_processor_id 宏。由于没有配置 CONFIG_DEBUG_PREEMPT，所以其等价于调用宏 raw_smp_processor_id，其意义在于 SMP 的情况下，获得当前 CPU 的 ID。如果不是 SMP，那么就返回 0。那么在 CONFIG_X86_32_SMP 的情况下：

```
#define raw_smp_processor_id() (percpu_read(cpu_number))
```

千万要注意，这里 cpu_number 来自 arch/x86/kernel/setup_percpu.c 的 30 行：

```
30DEFINE_PER_CPU(int, cpu_number);
31EXPORT_PER_CPU_SYMBOL(cpu_number);
```

这个东西不像是 c 语言全局变量，而是通过两个宏来定义的。要读懂这两个宏，必须对每 CPU 变量这个概念非常了解，如果还不是很清楚的同学请查阅一下博客“每 CPU 变量”http://blog.csdn.net/yunsongice/archive/2010/05/18/5605239.aspx。

其中 DEFINE_PER_CPU 来自文件 include/linux/percpu-defs.h：

```
#define DEFINE_PER_CPU(type, name)                      \
    DEFINE_PER_CPU_SECTION(type, name, "")
```

该宏静态分配一个每 CPU 数组，数组名为 name，结构类型为 type。由于我们没有设置 CONFIG_DEBUG_FORCE_WEAK_PER_CPU 编译选项，所以 DEFINE_PER_CPU_SECTION 又被定义为：

```
#define DEFINE_PER_CPU_SECTION(type, name, sec)                 \
    __PCPU_ATTRS(sec) PER_CPU_DEF_ATTRIBUTES                    \
    __typeof__(type) name
```

其中，__PCPU_ATTRS(sec)在 include/linux/percpu-defs.h 中定义：

```
#define __PCPU_ATTRS(sec)                               \
    __percpu __attribute__((section(PER_CPU_BASE_SECTION sec))) \
    PER_CPU_ATTRIBUTES
```

__percpu 是个编译扩展类型，大家可以去看看 include/linux/compile.h 这个文件，里面的 __percpu 是空的。而传进来的 sec 也是空的，PER_CPU_ATTRIBUTES 也是空的，而上面 PER_CPU_DEF_ATTRIBUTES 还是空代码，可能都是留给将来内核代码扩展之用的吧，所以 DEFINE_PER_CPU(int, cpu_number)展开就是：

```
__attribute__((section(PER_CPU_BASE_SECTION)))          \
__typeof__(int) cpu_number
```

所以现在只关注 PER_CPU_BASE_SECTION，来自 include/asm-generic/percpu.h

```
#ifdef CONFIG_SMP
#define PER_CPU_BASE_SECTION ".data.percpu"
#else
```

#define PER_CPU_BASE_SECTION ".data"
#endif

好了，介绍一下 GCC 的一些扩展，首先如何使用 typeof 来支持一些“ genericprogramming ”。gcc 支持一种叫做类型识别的技术，通过 typeof(x)关键字，获得 x 的数据类型。而如果是在一个要被一些 c 文件包含的头文件中获得变量的数据类型，就需要用__typeof__而不是 typeof 关键字了，比如说我们这里。最后，这里就是声明一个 int 类型的 cpu_number 指针，编译的时候把他指向.data.percpu 段的开始位置。

当然，我们自己写程序的时候没有这么地 generic programming，所以没必要这么做，不过想要成为一个内核达人，掌握这些技术还是很有必要的。DEFINE_PER_CPU_SECTION 宏又是什么意思呢？定义在 arch/x86/kernel/percpu-defs.h 的 147 行：
#define EXPORT_PER_CPU_SYMBOL(var) EXPORT_SYMBOL(var)

EXPORT_SYMBOL 是什么东西啊？还记得我们在编译内核时有个 kallsyms 目标么？这里就用到了。所以我说过，内核的编译过程很重要，请大家务必掌握！这个对象对应于“ /proc/kallsyms ”文件，该文件对应着内核符号表，记录了符号以及符号所在的内存地址。模块可以使用如下宏导出符号到内核符号表：
EXPORT_SYMBOL(符号名);
EXPORT_SYMBOL_GPL(符号名)

导出的符号可以被其他模块使用，不过使用之前一定要声明一下。EXPORT_SYMBOL_GPL()只适用于包含 GPL 许可权的模块。所以，EXPORT_SYMBOL(cpu_number)就是把 cpu_number 变量声明出去。接下来重点介绍一下 percpu_read 这个宏，来自 arch/x86/include/asm/percpu.h：
#define percpu_read(var) percpu_from_op("mov", var, "m" (var))

看到 mov，我们就知道可能要调用汇编语言了，所以这个宏调用 percpu_from_op("mov", cpu_number, "m" (cpu_number))，这个宏位于同一个文件：

```
164#define percpu_from_op(op, var, constraint)      \
165({                                                 \
166        typeof(var) pfo_ret__;                    \
167        switch (sizeof(var)) {                    \
168        case 1:                                   \
169                asm(op "b "__percpu_arg(1)",%0"   \
170                    : "=q" (pfo_ret__)            \
171                    : constraint);                \
172                break;                            \
173        case 2:                                   \
174                asm(op "w "__percpu_arg(1)",%0"   \
175                    : "=r" (pfo_ret__)            \
176                    : constraint);                \
177                break;                            \
178        case 4:                                   \
```

```
179                     asm(op "l "__percpu_arg(1)",%0"             \
180                         : "=r" (pfo_ret__)                      \
181                         : constraint);                          \
182                     break;                                          \
183             case 8:                                                 \
184                     asm(op "q "__percpu_arg(1)",%0"             \
185                         : "=r" (pfo_ret__)                      \
186                         : constraint);                          \
187                     break;                                          \
188             default: __bad_percpu_size();                       \
189             }                                                       \
190             pfo_ret__;                                          \
191})
```

当然，我们为了获得 CPU 号，cpu_number 用一个字节就够了，所以上面代码翻译过来就是：

```
asm("movb "__percpu_arg(1)",%0"             \
        : "=q" (pfo_ret__)                  \
        : constraint);
```

其中，__percpu_arg(1)是这样定义的：

```
#ifdef CONFIG_SMP
#define __percpu_arg(x)        "%%"__stringify(__percpu_seg)":%P" #x

#ifdef CONFIG_X86_64
#define __percpu_seg           gs
#define __percpu_mov_op            movq
#else
#define __percpu_seg           fs
#define __percpu_mov_op            movl
#endif
```

所以__percpu_arg(x)翻译过来就是：

```
"%%"__stringify(fs)":%P" #x
```

请注意，这是大家第一次遇到这样的定义方式，跟我们传统的 C 语言宏不一样了。不错，这里要学习新知识了，首先讲讲关于#和##的知识。

在 C 语言的宏中，#的功能是将其后面的宏参数进行字符串化操作（Stringfication），简单说就是在对它所引用的宏变量通过替换后在其左右各加上一个双引号。比如__percpu_arg 宏：

```
#define __percpu_arg(x)        "%%"__stringify(__percpu_seg)":%P" #x
```

而 x，我们传入的是 1；__percpu_seg，我们传入的是 fs，所以__percpu_arg 展开：

```
%%"__stringify(fs)":%P" "1"
```

而##，虽然这里没有用到，但是我还是一并讲讲，供大家扩充 c 语言知识。##被称为连接符

（concatenator），用来将两个 Token 连接为一个 Token。注意这里连接的对象是 Token 就行，而不一定是宏的变量。比如：

```
#define LINK_MULTIPLE(a,b,c,d) a##_##b##_##c##_##d
typedef struct record_type LINK_MULTIPLE(name,company,position,salary);
```

这里这个语句将展开为：

```
typedef struct record_type name_company_position_salary;
```

再来一个新知识，可变参数宏，于 1999 年的 ISO C 标准中定义，也就十年前而已。我们看到这里用到了__stringify 宏，其定义是这样的：

```
#define __stringify(x...)  __stringify_1(x)
```

其中…为可变参数，什么意思？就是这个宏除 x 以外，还可以有额外的多个参数。另外，如果是可变参数宏，那么可变参数 x 可能会展开为多个参数，那么比如上面也可定义为：

```
#define __stringify(...)     my__stringify(y,z)    0
```

不过如果指定了参数名 x，则后者必须包含一个 x：

```
#define __stringify(x...)    my__stringify(x,y,z)
```

那么我要问了，如果我定义成上面三个参数的形式，但是给的却又只有两个参数怎么办？比如执行一个__stringify(a,b)语句，会不会出错？当然会！因为既然是可变参数的宏，那么传递进去一个空参数也是对的啊，只不过你得有多余的逗号啊，也就是__stringify(,a,b)这样。

不过 GCC 还有一个东西可以避免这样的“ bug ”，那就是##符号。如果我们定义成：

```
#define __stringify(x...)    my__stringify(##x,y,z)
```

这时，##这个连接符号充当的作用就是当 x 为空的时候，消除前面的那个逗号，这样就不会有上面的 bug 了。

所以宏：

```
#define __stringify_1(x...)     #x
```

我们就可以看懂了吧，

所以__percpu_arg(1)最终翻译过来就是：

```
"%%" "fs:%P" "1"
```

因此上边的汇编代码翻译过来就是：

```
asm("movb %%fs:%P1, %0"                \
        : "=q" (pfo_ret__)                        \
        : "m" (var));
```

其中 pfo_ret__是输出部%0，q，表示寄存器 eax、ebx、ecx 或 edx 中的一个，并且变量 pfo_ret__存放在这个寄存器中。var 就是刚才我们建立的那个临时的汇编变量 cpu_number，作为输入部%1。

还记得“ 加载全局/中断描述符表 ”中把__KERNEL_PERCPU 段选择子赋给了 fs 了吗，不错，fs: cpu_number 就获得了当前存放在__KERNEL_PERCPU 段中 cpu_number 偏移的内存中，最后把结果返回给 pfo_ret__。所以这个宏最后的结果就是 pfo_ret__的值，其返回的是 CPU 的编号，把它赋给 boot_cpu_init 函数的内部变量 cpu。

那么这个偏移 cpu_number 到底是多少呢？众里寻他千百度，猛回首，这个偏移在生成的 vmlinux.lds 文件的 504 行被我发现了：
__per_cpu_load = .; .data.percpu 0

不错，刚才.data.percpu 处被编译成 0，所以内部 cpu 的值就是 0。我为什么把这一部分讲得这么详细呢，因为这部分内容包含了很多内核代码的细节，以后遇到同样的问题我们就照这样的方法来分析，举一反三。随后的四个函数 set_cpu_online、set_cpu_active、set_cpu_present 和 set_cpu_possible 我不想多说了，就是激活当前 CPU 的 cpu_present_bits 中四个标志位 online、active、present 和 possible，感兴趣的同学可以通过上面学到的方法去详细分析。

5.1.5 初始化地址散列表

560 行，page_address_init()函数，来自 mm/highmem.c：

```
409void __init page_address_init(void)
410{
411             int i;
412
413             INIT_LIST_HEAD(&page_address_pool);
414             for (i = 0; i < ARRAY_SIZE(page_address_maps); i++)
415                         list_add(&page_address_maps[i].list, &page_address_pool);
416             for (i = 0; i < ARRAY_SIZE(page_address_htable); i++) {
417                         INIT_LIST_HEAD(&page_address_htable[i].lh);
418                         spin_lock_init(&page_address_htable[i].lock);
419             }
420             spin_lock_init(&pool_lock);
421}
```

这个函数很简单，首先 413 初始化一个链表 page_address_pool，这个链表在编译的时候被定义成一个全局变量在同一个文件的 309 行：
static struct list_head page_address_pool

414 行，page_address_maps 也是个全局变量，来自同一个文件：
static struct page_address_map page_address_maps[LAST_PKMAP];
其每一个元素是一个 page_address_map 结构：

```
struct page_address_map {
        struct page *page;
        void *virtual;
        struct list_head list;
};
```

OK，第一次见到我们熟悉的 page 数据结构了。一个 page 代表一个物理页面，想要深入了

解这方面知识的请访问博客“Linux 页框管理”。
http://blog.csdn.net/yunsongice/archive/2010/01/22/5224494.aspx
ARRAY_SIZE 获得数组的元素个数，这里肯定就是 LAST_PKMAP 了，由于我们没有启动 PAE，这个值等于 1024，也就是说 page_address_maps 共有 1024 个元素。随后 415 行，1024 个循环以后，page_address_pool 通过每个 page_address_maps 的元素的 list 字段将他们全部链接起来形成个双向循环链表，又 page_address_pool 作为 head。

416 行，page_address_htable，也定义于同一文件：

```
static struct page_address_slot {
        struct list_head lh;            /* List of page_address_maps */
        spinlock_t lock;                /* Protect this bucket's list */
} ____cacheline_aligned_in_smp page_address_htable[1<<PA_HASH_ORDER];
```

____cacheline_aligned_in_smp 是一个编译优化选项，用于 SMP 方式的缓存优化。PA_HASH_ORDER 被定义为 7，所以 2^7=128。所以 ARRAY_SIZE(page_address_htable)为 128，那么经过 128 个循环以后，每个 page_address_htable 元素的 lh 都被初始化成一个链表头了，而对于的互斥锁 lock 也通过 spin_lock_init 函数进行初始化了。具体的初始化过程请查阅博客“Linux 内核入门（三）—— C 语言基本功”
http://blog.csdn.net/yunsongice/archive/2010/04/10/5471096.aspx
以及“自旋锁”
http://blog.csdn.net/yunsongice/archive/2010/05/18/5605264.aspx

最后 page_address_pool 的自旋锁 pool_lock 初始化之后，函数就结束了。简单归简单，但是大家必须知道这个函数的意义。我们知道一个 page 结构代表一个 4k 的页面，我们内核初始化这么一个全局的 page_address_maps 和 page_address_htable 这么两个结构，作为页面的地址映射。

561 行，执行一个 printk 函数。start_kernel()开始执行至此，会显示“Linux version 2.6.34…”信息。

5.1.6 打印版本信息

我们来简单的看看在内核中，在屏幕上打印一条信息的原理是怎么实现的。由于我们配置了 CONFIG_PRINTK 编译选项，所以调用位于 kernel/printk.c 中的 printk 函数：

```
584 asmlinkage int printk(const char *fmt, ...)
585 {
586         va_list args;
587         int r;
588
589         va_start(args, fmt);
590         r = vprintk(fmt, args);
591         va_end(args);
```

```
592
593         return r;
594}
```

又看到了“ … ”，不错，跟刚才讲的可变参数宏差不多，只不过这里是可变参函数。那么 start_kernel 中调用的 printk(KERN_NOTICE "%s", linux_banner)，其中 linux_banner 是个全局字符串常量，定义在 init/version.c：

```
const char linux_banner[] =
        "Linux version " UTS_RELEASE " (" LINUX_COMPILE_BY "@"
        LINUX_COMPILE_HOST ") (" LINUX_COMPILER ") " UTS_VERSION "\n";
```

上面那些大写的东西都在我们顶层 Makefile 中定义过了，就是一些版本信息，整个函数的功能没有问题，我们主要关注本质。所以来看看 va_start、vprintk 和 va_end 是怎么协作的。

首先，va_list 并不是一个什么复杂的数据结构，而是：

```
typedef char *va_list;
```

仅仅就是一个 char*。所以，根据 va_start 的定义：

```
#define _bnd(X, bnd)                    (((sizeof (X)) + (bnd)) & (~(bnd)))
#define va_arg(ap, T)                   \
   (*(T *)(((ap) += (_bnd (T, _AUPBND))) - (_bnd (T,_ADNBND))))
```

这就是一个可变参数的调整。调整好以后，我们就可以使用它了。

```
665asmlinkage int vprintk(const char *fmt, va_list args)
666{
667             int printed_len = 0;
668             int current_log_level = default_message_loglevel;
669             unsigned long flags;
670             int this_cpu;
671             char *p;
672
673             boot_delay_msec();
674             printk_delay();
675
676             preempt_disable();
677             /* This stops the holder of console_sem just where we want him */
678             raw_local_irq_save(flags);
679             this_cpu = smp_processor_id();
680
681             /*
682               * Ouch, printk recursed into itself!
683               */
684             if (unlikely(printk_cpu == this_cpu)) {
685                             /*
686                               * If a crash is occurring during printk() on this CPU,
```

```
687                      * then try to get the crash message out but make sure
688                      * we can't deadlock. Otherwise just return to avoid the
689                      * recursion and return - but flag the recursion so that
690                      * it can be printed at the next appropriate moment:
691                      */
692                     if (!oops_in_progress) {
693                             recursion_bug = 1;
694                             goto out_restore_irqs;
695                     }
696                     zap_locks();
697             }
698
699             lockdep_off();
700             spin_lock(&logbuf_lock);
701             printk_cpu = this_cpu;
702
703             if (recursion_bug) {
704                     recursion_bug = 0;
705                     strcpy(printk_buf, recursion_bug_msg);
706                     printed_len = strlen(recursion_bug_msg);
707             }
708             /* Emit the output into the temporary buffer */
709             printed_len += vscnprintf(printk_buf + printed_len,
710                                       sizeof(printk_buf) - printed_len, fmt, args);
711
712
713             p = printk_buf;
714
715             /* Do we have a loglevel in the string? */
716             if (p[0] == '<') {
717                     unsigned char c = p[1];
718                     if (c && p[2] == '>') {
719                             switch (c) {
720                             case '0' ... '7': /* loglevel */
721                                     current_log_level = c - '0';
722                             /* Fallthrough - make sure we're on a new line */
723                             case 'd': /* KERN_DEFAULT */
724                                     if (!new_text_line) {
725                                             emit_log_char('\n');
726                                             new_text_line = 1;
727                                     }
728                             /* Fallthrough - skip the loglevel */
729                             case 'c': /* KERN_CONT */
730                                     p += 3;
```

```
731                                     break;
732                             }
733                     }
734             }
735
736             /*
737              * Copy the output into log_buf.    If the caller didn't provide
738              * appropriate log level tags, we insert them here
739              */
740             for ( ; *p; p++) {
741                     if (new_text_line) {
742                             /* Always output the token */
743                             emit_log_char('<');
744                             emit_log_char(current_log_level + '0');
745                             emit_log_char('>');
746                             printed_len += 3;
747                             new_text_line = 0;
748
749                             if (printk_time) {
750                                     /* Follow the token with the time */
751                                     char tbuf[50], *tp;
752                                     unsigned tlen;
753                                     unsigned long long t;
754                                     unsigned long nanosec_rem;
755
756                                     t = cpu_clock(printk_cpu);
757                                     nanosec_rem = do_div(t, 1000000000);
758                                     tlen = sprintf(tbuf, "[%5lu.%06lu] ",
759                                                     (unsigned long) t,
760                                                     nanosec_rem / 1000);
761
762                                     for (tp = tbuf; tp < tbuf + tlen; tp++)
763                                             emit_log_char(*tp);
764                                     printed_len += tlen;
765                             }
766
767                             if (!*p)
768                                     break;
769                     }
770
771                     emit_log_char(*p);
772                     if (*p == '\n')
773                             new_text_line = 1;
774             }
```

```
775
776             /*
777               * Try to acquire and then immediately release the
778               * console semaphore. The release will do all the
779               * actual magic (print out buffers, wake up klogd,
780               * etc).
781               *
782               * The acquire_console_semaphore_for_printk() function
783               * will release 'logbuf_lock' regardless of whether it
784               * actually gets the semaphore or not.
785               */
786             if (acquire_console_semaphore_for_printk(this_cpu))
787                             release_console_sem();
788
789             lockdep_on();
790out_restore_irqs:
791             raw_local_irq_restore(flags);
792
793             preempt_enable();
794             return printed_len;
795}
```

vprintk 是个大家伙！该函数负责解析并组成打印格式。673~707 行是几个必要的操作，这几个函数都是跟同步互斥以及调试相关。709 行，调用 vscnprintf 将打印信息拷贝到 printk_buf 缓存中。比如 printk("%d, %s", 5, "hello")，printk_buf 中将是 5，hello 格式化后的数据不带'\0'。

随后，p 指向这个缓存，存放着 linux_banner 对应的字符串。716 行，由于 linux_banner 的第一个字符不是‘ < ”,所以不存在 log 级别，直接到 740 行进入一个循环。743 行 emit_log_char 函数是将打印信息已经保存到 log_buf 中，这个 log_buf 有什么用，待会再说。

749 行，printk_time，由于没有定义 CONFIG_PRINTK_TIME 宏 printk_time = 0，所以忽略 749 到 765 行代码。好了，771 行，打印信息已经保存到 log_buf 中了，最后 787 行调用 release_console_sem()函数启动 console 输出。

注意，我们这里分析的 printk 和 printf 不同。回顾一下，他首先输出到系统的一个缓冲区内，大约 4k，如果登记了 console，则调用 console->wirte 函数输出，否则就一直在 buffer 里呆着。所以，用 printk 输出的信息，如果超出了 4k，会冲掉前面的。在系统引导起来后，用 dmesg 看的也就是这个 buffer 中的东西。

此时的信息还在 buf 中躺着呢，因为 console 还没有初始化。

5.2 执行 setup_arch()函数

回到 start_kernel 当中，562 行，调用 setup_arch 函数，传给他的参数是那个未被初始化的内部变量 command_line。这个 setup_arch()函数是 start_kernel 阶段最重要的一个函数，每个体系都有自己的 setup_arch()函数，主要用于根据处理器硬件平台设置系统；解析 linux 命令行参数；设置 0 号进程的内存描述结构 init_mm；系统内存管理初始化；统计并注册系统各种资源；以及其它项目的初始化等。具体编译哪个体系的 setup_arch()函数，由顶层 Makefile 中的 ARCH 变量决定：

```
 724void __init setup_arch(char **cmdline_p)
 725{
 726            int acpi = 0;
 727            int k8 = 0;
 728
……
 730            memcpy(&boot_cpu_data, &new_cpu_data, sizeof(new_cpu_data));
 731            visws_early_detect();
……
 736            /* VMI may relocate the fixmap; do this before touching ioremap area */
 737            vmi_init();
 738
 739            early_cpu_init();
 740            early_ioremap_init();
 741
 742            ROOT_DEV = old_decode_dev(boot_params.hdr.root_dev);
 743            screen_info = boot_params.screen_info;
 744            edid_info = boot_params.edid_info;
 ……
 746            apm_info.bios = boot_params.apm_bios_info;
 747            ist_info = boot_params.ist_info;
 748            if (boot_params.sys_desc_table.length != 0) {
 749                        set_mca_bus(boot_params.sys_desc_table.table[3] & 0x2);
 750                        machine_id = boot_params.sys_desc_table.table[0];
 751                        machine_submodel_id = boot_params.sys_desc_table.table[1];
 752                        BIOS_revision = boot_params.sys_desc_table.table[2];
 753            }
 ……
 755            saved_video_mode = boot_params.hdr.vid_mode;
 756            bootloader_type = boot_params.hdr.type_of_loader;
 757            if ((bootloader_type >> 4) == 0xe) {
 758                        bootloader_type &= 0xf;
 759                        bootloader_type |= (boot_params.hdr.ext_loader_type+0x10) << 4;
 760            }
```

```
761         bootloader_version    = bootloader_type & 0xf;
762         bootloader_version |= boot_params.hdr.ext_loader_ver << 4;
……
782         x86_init.oem.arch_setup();
783
784         setup_memory_map();
785         parse_setup_data();
786         /* update the e820_saved too */
787         e820_reserve_setup_data();
788
789         copy_edd();
790
791         if (!boot_params.hdr.root_flags)
792                 root_mountflags &= ~MS_RDONLY;
793         init_mm.start_code = (unsigned long) _text;
794         init_mm.end_code = (unsigned long) _etext;
795         init_mm.end_data = (unsigned long) _edata;
796         init_mm.brk = _brk_end;
797
798         code_resource.start = virt_to_phys(_text);
799         code_resource.end = virt_to_phys(_etext)-1;
800         data_resource.start = virt_to_phys(_etext);
801         data_resource.end = virt_to_phys(_edata)-1;
802         bss_resource.start = virt_to_phys(&__bss_start);
803         bss_resource.end = virt_to_phys(&__bss_stop)-1;
……
817
818         strlcpy(command_line, boot_command_line, COMMAND_LINE_SIZE);
819         *cmdline_p = command_line;
……
828         x86_configure_nx();
829
830         parse_early_param();
831
832         x86_report_nx();
833
834         /* Must be before kernel pagetables are setup */
835         vmi_activate();
836
837         /* after early param, so could get panic from serial */
838         reserve_early_setup_data();
839
840         if (acpi_mps_check()) {
841#ifdef CONFIG_X86_LOCAL_APIC
```

```
 842                     disable_apic = 1;
 843#endif
 844                     setup_clear_cpu_cap(X86_FEATURE_APIC);
 845             }
 846
 847#ifdef CONFIG_PCI
 848             if (pci_early_dump_regs)
 849                     early_dump_pci_devices();
 850#endif
 851
 852             finish_e820_parsing();
 853
 854             if (efi_enabled)
 855                     efi_init();
 856
 857             dmi_scan_machine();
 858
 859             dmi_check_system(bad_bios_dmi_table);
 860
……
 865             init_hypervisor_platform();
 866
 867             x86_init.resources.probe_roms();
 868
 869             /* after parse_early_param, so could debug it */
 870             insert_resource(&iomem_resource, &code_resource);
 871             insert_resource(&iomem_resource, &data_resource);
 872             insert_resource(&iomem_resource, &bss_resource);
 873
 874             trim_bios_range();
……
 891             max_pfn = e820_end_of_ram_pfn();
 892
 893             /* preallocate 4k for mptable mpc */
 894             early_reserve_e820_mpc_new();
 895             /* update e820 for memory not covered by WB MTRRs */
 896             mtrr_bp_init();
 897             if (mtrr_trim_uncached_memory(max_pfn))
 898                     max_pfn = e820_end_of_ram_pfn();
 899
……
 901             /* max_low_pfn get updated here */
 902             find_low_pfn_range();
……
```

```
 923             printk(KERN_DEBUG "initial memory mapped : 0 - %08lx\n",
 924                                      max_pfn_mapped<<PAGE_SHIFT);
 925
 926             reserve_brk();
 927
 928             /*
 929               * Find and reserve possible boot-time SMP configuration:
 930               */
 931             find_smp_config();
 932
 933             reserve_ibft_region();
 934
 935             reserve_trampoline_memory();
 ……
 944             init_gbpages();
 945
 946            /* max_pfn_mapped is updated here */
 947            max_low_pfn_mapped = init_memory_mapping(0, max_low_pfn<<PAGE_SHIFT);
 948            max_pfn_mapped = max_low_pfn_mapped;
 949
……
 968             reserve_initrd();
 969
 970             reserve_crashkernel();
 971
 972             vsmp_init();
 973
 974             io_delay_init();
 975
 976             /*
 977               * Parse the ACPI tables for possible boot-time SMP configuration.
 978               */
 979             acpi_boot_table_init();
 980
 981             early_acpi_boot_init();
 982
 983#ifdef CONFIG_ACPI_NUMA
 984             /*
 985               * Parse SRAT to discover nodes.
 986               */
 987             acpi = acpi_numa_init();
 988#endif
 989
 990#ifdef CONFIG_K8_NUMA
```

```
 991             if (!acpi)
 992                     k8 = !k8_numa_init(0, max_pfn);
 993#endif
 994
 995             initmem_init(0, max_pfn, acpi, k8);
 996#ifndef CONFIG_NO_BOOTMEM
 997             early_res_to_bootmem(0, max_low_pfn<<PAGE_SHIFT);
 998#endif
 999
1000             dma32_reserve_bootmem();
……
1005
1006             x86_init.paging.pagetable_setup_start(swapper_pg_dir);
1007             paging_init();
1008             x86_init.paging.pagetable_setup_done(swapper_pg_dir);
1009
1010             tboot_probe();
1011
……
1016             generic_apic_probe();
1017
1018             early_quirks();
1019
1020             /*
1021              * Read APIC and some other early information from ACPI tables.
1022              */
1023             acpi_boot_init();
1024
1025             sfi_init();
1026
1027             /*
1028              * get boot-time SMP configuration:
1029              */
1030             if (smp_found_config)
1031                     get_smp_config();
1032
1033             prefill_possible_map();
1034
……
1038
1039             init_apic_mappings();
1040             ioapic_init_mappings();
1041
1042             /* need to wait for io_apic is mapped */
```

```
1043            probe_nr_irqs_gsi();
1044
1045            kvm_guest_init();
1046
1047            e820_reserve_resources();
1048            e820_mark_nosave_regions(max_low_pfn);
1049
1050            x86_init.resources.reserve_resources();
1051
1052            e820_setup_gap();
1053
1054#ifdef CONFIG_VT
1055#if defined(CONFIG_VGA_CONSOLE)
1056        if (!efi_enabled || (efi_mem_type(0xa0000) != EFI_CONVENTIONAL_MEMORY))
1057                    conswitchp = &vga_con;
1058#elif defined(CONFIG_DUMMY_CONSOLE)
1059            conswitchp = &dummy_con;
1060#endif
1061#endif
1062            x86_init.oem.banner();
1063
1064            mcheck_init();
1065}
```

浩浩荡荡 300 多行代码，我们去掉了其中 64 位 x86 体系的相关代码，以及一些未配置的编译选项，如 CONFIG_EFI、CONFIG_CMDLINE_BOOL、CONFIG_ACPI_SLEEP 等的相关代码，剩余对我们有用的代码如上所示，也有将近 200 行。我们这里只挑几个重要的来讲解一下。

5.2.1 拷贝可用内存区信息

首先 setup_arch 第一步要做的就是保存 arch/x86/kernel/head_32.S 初始化的 new_cpu_data 数据到 boot_cpu_data 中。new_cpu_data 主要是保存 CPU 的相关信息，在哪儿初始化的？还记得我们在 arch/x86/kernel/head_32.S 中忽略过的 checkCPUtype 吗？就在那里，感兴趣的同学可以去探究一下。

784 行，setup_memory_map()函数，进入 start_kernel 内核初始化函数中第一个内存管理函数就是由它来实现。

```
1204 void __init setup_memory_map(void)
1205 {
1206        char *who;
```

```
1207
1208        who = x86_init.resources.memory_setup();
1209        memcpy(&e820_saved, &e820, sizeof(struct e820map));
1210        printk(KERN_INFO "BIOS-provided physical RAM map:\n");
1211        e820_print_map(who);
1212 }
```

setup_memory_map()最终调用 x86_init.resources.memory_setup()实现对 e820 内存图的优化，并根据 boot_params 中 e820_map 字段的值来设置全局变量 e820 的值。这个全局变量是一个 e820map 结构：

```
struct e820map {
    __u32 nr_map;
    struct e820entry map[E820_X_MAX];
};
```

还记得吗？我们在执行实模式下代码 main 函数的时候调用了一个 detect_memory_e820()函数，当时该函数通过 BIOS 服务程序 int 0x15 获得系统启动后的所有可用空间，共有 boot_params.e820_entries 个可用空间，每块空间作为 boot_params.e820_map[]数组的元素，存放着他们的起始地址、大小和元素。

这里说个题外话，探测一个 PC 机内存的最好方法是就是通过调用 INT 0x15，eax = 0xe820 来实现。这个功能在 2002 年以后被所有 PC 机所使用，这是唯一能够探测超过 4G 大小内存的方案，当然，这个方法也可以被认为是内存的最终检测方法。实际上，这个函数返回一个非排序列表,这个列表包含了那些没有使用的内存信息的项,并且可能返回存在覆盖的区域。在 linux 中每个列表项被存放在 ES:EDI 指定的内存区域中。每个项均有一定的格式：即 2 个 8 字节字段，一个 2 字节字段。我们前面看见了，对于内存探测的实现由函数 detect_memory_e820 来实现的，在这个函数中，使用了一个 do...while()循环来实现，并将所探测的内容写入 boot_params.e820_map 数组中。

那么，x86_init.resources.memory_setup()是个什么函数呢？在 arch/x86/kernel/x86_init.c 的 32 行，我们看到又有一个__initdata 的定义：

```
32struct x86_init_ops x86_init __initdata = {
33
34          .resources = {
35                  .probe_roms             = x86_init_noop,
36                  .reserve_resources      = reserve_standard_io_resources,
37                  .memory_setup           = default_machine_specific_memory_setup,
38          },
39
40          .mpparse = {
41                  .mpc_record             = x86_init_uint_noop,
42                  .setup_ioapic_ids       = x86_init_noop,
43                  .mpc_apic_id            = default_mpc_apic_id,
44                  .smp_read_mpc_oem       = default_smp_read_mpc_oem,
```

```
45                      .mpc_oem_bus_info           = default_mpc_oem_bus_info,
46                      .find_smp_config            = default_find_smp_config,
47                      .get_smp_config             = default_get_smp_config,
48              },
49
50              .irqs = {
51                      .pre_vector_init            = init_ISA_irqs,
52                      .intr_init                  = native_init_IRQ,
53                      .trap_init                  = x86_init_noop,
54              },
55
56              .oem = {
57                      .arch_setup                 = x86_init_noop,
58                      .banner                     = default_banner,
59              },
60
61              .paging = {
62                      .pagetable_setup_start    = native_pagetable_setup_start,
63                      .pagetable_setup_done     = native_pagetable_setup_done,
64              },
65
66              .timers = {
67                      .setup_percpu_clockev     = setup_boot_APIC_clock,
68                      .tsc_pre_init             = x86_init_noop,
69                      .timer_init               = hpet_time_init,
70              },
71
72              .iommu = {
73                      .iommu_init               = iommu_init_noop,
74              },
75
76              .pci = {
77                      .init                     = x86_default_pci_init,
78                      .init_irq                 = x86_default_pci_init_irq,
79                      .fixup_irqs               = x86_default_pci_fixup_irqs,
80              },
81};
```

这里，跟其他__initdata 数据一样，在编译的时候就存放在 init 数据区了。那么 x86_init.resources.memory_setup 也就是 37 行的那个 default_machine_specific_memory_setup 函数了。所以我们看到这个函数：

```
1166char *__init default_machine_specific_memory_setup(void)
1167{
```

```
1168        char *who = "BIOS-e820";
1169        u32 new_nr;
1170        /*
1171         * Try to copy the BIOS-supplied E820-map.
1172         *
1173         * Otherwise fake a memory map; one section from 0k->640k,
1174         * the next section from 1mb->appropriate_mem_k
1175         */
1176        new_nr = boot_params.e820_entries;
1177        sanitize_e820_map(boot_params.e820_map,
1178                          ARRAY_SIZE(boot_params.e820_map),
1179                          &new_nr);
1180        boot_params.e820_entries = new_nr;
1181        if (append_e820_map(boot_params.e820_map, boot_params.e820_entries)
1182          < 0) {
1183                u64 mem_size;
1184
1185                /* compare results from other methods and take the greater */
1186                if (boot_params.alt_mem_k
1187                      < boot_params.screen_info.ext_mem_k) {
1188                        mem_size = boot_params.screen_info.ext_mem_k;
1189                        who = "BIOS-88";
1190                } else {
1191                        mem_size = boot_params.alt_mem_k;
1192                        who = "BIOS-e801";
1193                }
1194
1195                e820.nr_map = 0;
1196                e820_add_region(0, LOWMEMSIZE(), E820_RAM);
1197                e820_add_region(HIGH_MEMORY, mem_size << 10, E820_RAM);
1198        }
1199
1200        /* In case someone cares... */
1201        return who;
1202}
```

首先 1177 行为所有可用内存区空间进行“消毒”。至于 sanitize“消毒”算法，有兴趣的同学，特别是对信息安全感兴趣的同学可以去研究一下。最重点的是 1181 行调用 append_e820_map 函数，将 boot_params.e820_map[]数组中所有的内存区物理信息拷贝到全局数据 e820 中，最终返回"BIOS-e820"字符串，赋给 setup_memory_map()函数中的内部变量 who。

之后 setup_memory_map 函数将 e820 的数据保存到 e820_saved 中，相当于备份。最后调用 e820_print_map 打印出内存分布图，注意这里，e820.nr_map 就是可以的内存区的数量：

```
151void __init e820_print_map(char *who)
152{
153             int i;
154
155             for (i = 0; i < e820.nr_map; i++) {
156                          printk(KERN_INFO " %s: %016Lx - %016Lx ", who,
157                                       (unsigned long long) e820.map[i].addr,
158                                       (unsigned long long)
159                                       (e820.map[i].addr + e820.map[i].size));
160                          e820_print_type(e820.map[i].type);
161                          printk(KERN_CONT "\n");
162             }
163}
```

793~796 行，调用系统编译生成的数据来初始化 init_mm 的 start_code、end_code、end_data 以及.brk 等字段；同时也初始化 code_resource 的 start、end 字段和 data_resource 的 start、end 字段以及 bss_resource 的 start、end 字段。这个 init_mm 还记得吗？初始化 0 号进程的时候，它的 task_struct 的 active_mm 就被赋成了这个。它是一个 mm_struct 结构，在编译 vmlinux 的时候就只给他初始化了一下字段：

```
struct mm_struct init_mm = {
        .mm_rb          = RB_ROOT,
        .pgd        = swapper_pg_dir,
        .mm_users       = ATOMIC_INIT(2),
        .mm_count       = ATOMIC_INIT(1),
        .mmap_sem       = __RWSEM_INITIALIZER(init_mm.mmap_sem),
        .page_table_lock =     __SPIN_LOCK_UNLOCKED(init_mm.page_table_lock),
        .mmlist         = LIST_HEAD_INIT(init_mm.mmlist),
        .cpu_vm_mask        = CPU_MASK_ALL,
};
```

这里初始化它的 start_code、end_code、end_data 以及.brk 等字段分别为_text、_etext、_edata 和_brk_end，是一步极其重要的过程，表明了当前 0 号进程进程的虚拟内存技术就可以用了。至于后面那几个 resource，通过 870~872 行代码将其作为一个 IO resource 保存起来，有关 IO 端口的相关知识，请查阅博客“ Linux I/O 体系结构 ”
http://blog.csdn.net/yunsongice/archive/2010/06/07/5652671.aspx

5.2.2 获得总页面数

回到 setup_arch()中，接下来，继续走，891 行调用 e820_end_of_ram_pfn()函数根据 e820 的数据来获得 32 位可用物理内存地址的最大值并右移 PAGE_SHIFT，也就是 12 位，最后由函数 e820_end_pfn 返回这个 20 位的值，保存在内部变量 max_pfn 中，作为总的页面数量：

```
855static unsigned long __init e820_end_pfn(unsigned long limit_pfn, unsigned type)
 856{
 857             int i;
 858             unsigned long last_pfn = 0;
 859             unsigned long max_arch_pfn = MAX_ARCH_PFN;
 860
 861             for (i = 0; i < e820.nr_map; i++) {
 862                          struct e820entry *ei = &e820.map[i];
 863                          unsigned long start_pfn;
 864                          unsigned long end_pfn;
 865
 866                          if (ei->type != type)
 867                                       continue;
 868
 869                          start_pfn = ei->addr >> PAGE_SHIFT;
 870                          end_pfn = (ei->addr + ei->size) >> PAGE_SHIFT;
 871
 872                          if (start_pfn >= limit_pfn)
 873                                       continue;
 874                          if (end_pfn > limit_pfn) {
 875                                       last_pfn = limit_pfn;
 876                                       break;
 877                          }
 878                          if (end_pfn > last_pfn)
 879                                       last_pfn = end_pfn;
 880             }
 881
 882             if (last_pfn > max_arch_pfn)
 883                          last_pfn = max_arch_pfn;
 884
 885             printk(KERN_INFO "last_pfn = %#lx max_arch_pfn = %#lx\n",
 886                                        last_pfn, max_arch_pfn);
 887             return last_pfn;
 888}
 889unsigned long __init e820_end_of_ram_pfn(void)
 890{
 891             return e820_end_pfn(MAX_ARCH_PFN, E820_RAM);
 892}
```

紧接着，setup_arch 的第 902 行调用 find_low_pfn_range()，它根据 max_pfn 是否大于 MAXMEM_PFN 来判断是否启用了高端内存区。这个 MAXMEM_PFN 值等于多少，大家自己去查，不是很复杂，只是这个值跟 max_pfn 一样，也是除去低 12 位的 20 位的一个数值。如果启用了，就调用 highmem_pfn_init() 函数将全局变量 max_low_pfn 设置为 MAXMEM_PFN；并设置全局变量 highmem_pages 为 max_pfn - MAXMEM_PFN，作为高端

页面的数量：

```
660void __init highmem_pfn_init(void)
661{
662             max_low_pfn = MAXMEM_PFN;
663
664             if (highmem_pages == -1)
665                         highmem_pages = max_pfn - MAXMEM_PFN;
666
667             if (highmem_pages + MAXMEM_PFN < max_pfn)
668                         max_pfn = MAXMEM_PFN + highmem_pages;
669
670             if (highmem_pages + MAXMEM_PFN > max_pfn) {
671                         printk(KERN_WARNING MSG_HIGHMEM_TOO_SMALL,
672                                     pages_to_mb(max_pfn - MAXMEM_PFN),
673                                     pages_to_mb(highmem_pages));
674                         highmem_pages = 0;
675             }
……
692}
```

5.2.3 着手建立永久内核页表

得到了总的页面数 max_pfn 和高端页面数 highmem_pages 之后，来到 setup_arch 的 947 行，调用 init_memory_mapping()函数来建立系统初始化阶段的临时分页体系，传入的参数意义代表从 0~max_low_pfn 对应的 32 位物理地址（低 12 位全为 0，也就是页面对齐），在函数 init_memory_mapping 函数中先后调用下面的几个函数来设置内存相关数据(因为 bootmem 此时没有初始化)：

find_early_table_space()
kernel_physical_mapping_init()
early_ioremap_page_table_range_init()
load_cr3()
reserve_early()

其中，首先 find_early_table_space 所实现的功能是相当重要的：

```
32static void __init find_early_table_space(unsigned long end, int use_pse,
33                                                      int use_gbpages)
34{
35             unsigned long puds, pmds, ptes, tables, start;
36
37             puds = (end + PUD_SIZE - 1) >> PUD_SHIFT;
```

```
38              tables = roundup(puds * sizeof(pud_t), PAGE_SIZE);
39
40              if (use_gbpages) {
41                      unsigned long extra;
42
43                      extra = end - ((end>>PUD_SHIFT) << PUD_SHIFT);
44                      pmds = (extra + PMD_SIZE - 1) >> PMD_SHIFT;
45              } else
46                      pmds = (end + PMD_SIZE - 1) >> PMD_SHIFT;
47
48              tables += roundup(pmds * sizeof(pmd_t), PAGE_SIZE);
49
50              if (use_pse) {
51                      unsigned long extra;
52
53                      extra = end - ((end>>PMD_SHIFT) << PMD_SHIFT);
54#ifdef CONFIG_X86_32
55                      extra += PMD_SIZE;
56#endif
57                      ptes = (extra + PAGE_SIZE - 1) >> PAGE_SHIFT;
58              } else
59                      ptes = (end + PAGE_SIZE - 1) >> PAGE_SHIFT;
60
61              tables += roundup(ptes * sizeof(pte_t), PAGE_SIZE);
62
63#ifdef CONFIG_X86_32
64              /* for fixmap */
65              tables += roundup(__end_of_fixed_addresses * sizeof(pte_t), PAGE_SIZE);
66#endif
67
68              /*
69                * RED-PEN putting page tables only on node 0 could
70                * cause a hotspot and fill up ZONE_DMA. The page tables
71                * need roughly 0.5KB per GB.
72                */
73#ifdef CONFIG_X86_32
74              start = 0x7000;
75#else
76              start = 0x8000;
77#endif
78              e820_table_start = find_e820_area(start, max_pfn_mapped<<PAGE_SHIFT,
79                                                tables, PAGE_SIZE);
80              if (e820_table_start == -1UL)
81                      panic("Cannot find space for the kernel page tables");
```

```
 82
 83             e820_table_start >>= PAGE_SHIFT;
 84             e820_table_end = e820_table_start;
 85             e820_table_top = e820_table_start + (tables >> PAGE_SHIFT);
 86
 87             printk(KERN_DEBUG "kernel direct mapping tables up to %lx @ %lx-%lx\n",
 88                                     end, e820_table_start << PAGE_SHIFT, e820_table_top <<
PAGE_SHIFT);
 89}
```

我们看到它确定了 PUD 和 PMD 以及 PTE、固定内存映射等所有的选项所使用的内存空间的大小；之后调用 find_e820_area 函数从 e820.map[]数组中寻找到一块能够容纳所有页表项的内存段：

```
743u64 __init find_e820_area(u64 start, u64 end, u64 size, u64 align)
744{
745             int i;
746
747             for (i = 0; i < e820.nr_map; i++) {
748                             struct e820entry *ei = &e820.map[i];
749                             u64 addr;
750                             u64 ei_start, ei_last;
751
752                             if (ei->type != E820_RAM)
753                                             continue;
754
755                             ei_last = ei->addr + ei->size;
756                             ei_start = ei->addr;
757                             addr = find_early_area(ei_start, ei_last, start, end,
758                                                                     size, align);
759
760                             if (addr != -1ULL)
761                                             return addr;
762             }
763             return -1ULL;
764}
```

find_e820_area 中检测 e820.map 的每一个元素，这个元素代表的内存区必须是 E820_RAM，然后调用 find_early_area 获得 tables 的首地址：

```
539u64 __init find_early_area(u64 ei_start, u64 ei_last, u64 start, u64 end,
540                                     u64 size, u64 align)
541{
542             u64 addr, last;
```

```
543
544             addr = round_up(ei_start, align);
545             if (addr < start)
546                     addr = round_up(start, align);
547             if (addr >= ei_last)
548                     goto out;
549             while (bad_addr(&addr, size, align) && addr+size <= ei_last)
550                     ;
551             last = addr + size;
552             if (last > ei_last)
553                     goto out;
554             if (last > end)
555                     goto out;
556
557             return addr;
558
559out:
560             return -1ULL;
561}
```

这个 find_early_area 函数我们要好好说道说道。早系统初始化时，什么内存管理器、内存模型这些东西都没有建立，那么获得内存的最低级函数就是这个 find_early_area 函数。它接收 6 个参数：ei_start 和 ei_last 表示分配范围，我们看到这个范围不能超出某个 e820.map[i]元素代表的范围；start 和 end 代表期望分配的范围；size 为期望分配大小；align 表示对齐方式。我们看到，如果 addr >= ei_last，或者 last > ei_last，或者 last > end 都会分配失败。这个条件看上去很苛刻，不过 e820.map[]数组有那么多元素，总有一个元素会满足的，不然所有的 Linux 都无法运行了。

round_up 以及还有一个双胞胎弟弟 round_down 的定义如下：

```
#define __round_mask(x, y) ((__typeof__(x))((y)-1))
#define round_up(x, y) ((((x)-1) | __round_mask(x, y))+1)
#define round_down(x, y) ((x) & ~__round_mask(x, y))
```

获得了获得页表 tables 的首地址之后，find_early_table_space 就用它设置 e820_table_start、e820_table_end 和 e820_table_top 的值。这三个全局变量相当的重要，分别表示这个将要容纳所有内核页表的空间的起始、结束和顶部页对齐地址头 20 位（此时 e820_table_start 和 e820_table_end 是相等的），在后面的页表初始化阶段将会使用到这三个变量。

看了这么多代码，很多同学可能有些迷糊了。为啥我们在 arch/x86/kernel/head_32.S 中已经初始化了分页环境，建立了一个临时页表了，为啥这里还要建一个呢？其实，这就是 ULK-3 书中提到的内核临时页表和最终内核页表的概念。我在博客“高端内存映射”http://blog.csdn.net/yunsongice/archive/2010/01/26/5258589.aspx 中也提到过这一点。

大家好好回忆一下，当时建立的临时页表个数只取决于_end 的位置，也就是把解压缩后的

内核代码映射出来了。而这里，是要把所有可用的 RAM（注意这里是包括了已经映射了的内核代码、页目录）以页为单位分成多个页，每个页一个比特，提供一个初始阶段内存的分配和释放管理平台。

目前我的电脑，在 arch/x86/kernel/head_32.S 里只是映射了大概 4M 的 vmlinux 代码。内核尺寸在 4M 左右(不压缩)，一般需要连续映射 3 个页面表。现在要把所有 RAM 映射到内核空间。那么内核要根据 e820 物理内存的布局，也就是 RAM 的结点布局对多个结点及结点的管理区作初化，最终把除去内核之外所有剩余的页交给页框分配器，同时也完成了页框分配器的初始化。

5.2.4 第二次启动分页管理

回到 init_memory_mapping()函数，之后 nr_range 次调用 kernel_physical_mapping_init()，nr_range 为 map_range 数组的总的元素数，前面总共执行了两次 save_mr，所以 nr_range 为 2，分别是 0~1<<9 和 2MB~end>>12 的 map_range 结构，代表 4k 和 2MB 两种形式的内存页表。不过为了方便起见，我们暂时忽略 2MB 页面大小的体系，只分析传统 4k 页面体系。

```
238unsigned long __init
239kernel_physical_mapping_init(unsigned long start,
240                                        unsigned long end,
241                                        unsigned long page_size_mask)
242{
243         int use_pse = page_size_mask == (1<<PG_LEVEL_2M);
244         unsigned long last_map_addr = end;
245         unsigned long start_pfn, end_pfn;
246         pgd_t *pgd_base = swapper_pg_dir;
247         int pgd_idx, pmd_idx, pte_ofs;
248         unsigned long pfn;
249         pgd_t *pgd;
250         pmd_t *pmd;
251         pte_t *pte;
252         unsigned pages_2m, pages_4k;
253         int mapping_iter;
254
255         start_pfn = start >> PAGE_SHIFT;
256         end_pfn = end >> PAGE_SHIFT;
257
258         /*
259          * First iteration will setup identity mapping using large/small pages
260          * based on use_pse, with other attributes same as set by
261          * the early code in head_32.S
262          *
263          * Second iteration will setup the appropriate attributes (NX, GLOBAL..)
```

```
264             * as desired for the kernel identity mapping.
265             *
266             * This two pass mechanism conforms to the TLB app note which says:
267             *
268             *      "Software should not write to a paging-structure entry in a way
269             *       that would change, for any linear address, both the page size
270             *       and either the page frame or attributes."
271             */
272            mapping_iter = 1;
273
274            if (!cpu_has_pse)
275                    use_pse = 0;
276
277repeat:
278            pages_2m = pages_4k = 0;
279            pfn = start_pfn;
280            pgd_idx = pgd_index((pfn<<PAGE_SHIFT) + PAGE_OFFSET);
281            pgd = pgd_base + pgd_idx;
282            for (; pgd_idx < PTRS_PER_PGD; pgd++, pgd_idx++) {
283                    pmd = one_md_table_init(pgd);
284
285                    if (pfn >= end_pfn)
286                            continue;
287#ifdef CONFIG_X86_PAE
288                    pmd_idx = pmd_index((pfn<<PAGE_SHIFT) + PAGE_OFFSET);
289                    pmd += pmd_idx;
290#else
291                    pmd_idx = 0;
292#endif
293                    for (; pmd_idx < PTRS_PER_PMD && pfn < end_pfn;
294                            pmd++, pmd_idx++) {
295                        unsigned int addr = pfn * PAGE_SIZE + PAGE_OFFSET;
296
297                        /*
298                          * Map with big pages if possible, otherwise
299                          * create normal page tables:
300                          */
301                        if (use_pse) {
302                                unsigned int addr2;
303                                pgprot_t prot = PAGE_KERNEL_LARGE;
304                                /*
305                                  * first pass will use the same initial
306                                  * identity mapping attribute + _PAGE_PSE.
307                                  */
```

```
308                                     pgprot_t init_prot =
309                                             __pgprot(PTE_IDENT_ATTR |
310                                                     _PAGE_PSE);
311
312                                     addr2 = (pfn + PTRS_PER_PTE-1) * PAGE_SIZE +
313                                             PAGE_OFFSET + PAGE_SIZE-1;
314
315                                     if (is_kernel_text(addr) ||
316                                         is_kernel_text(addr2))
317                                             prot = PAGE_KERNEL_LARGE_EXEC;
318
319                                     pages_2m++;
320                                     if (mapping_iter == 1)
321                                             set_pmd(pmd, pfn_pmd(pfn, init_prot));
322                                     else
323                                             set_pmd(pmd, pfn_pmd(pfn, prot));
324
325                                     pfn += PTRS_PER_PTE;
326                                     continue;
327                             }
328                             pte = one_page_table_init(pmd);
329
330                             pte_ofs = pte_index((pfn<<PAGE_SHIFT) + PAGE_OFFSET);
331                             pte += pte_ofs;
332                             for (; pte_ofs < PTRS_PER_PTE && pfn < end_pfn;
333                                     pte++, pfn++, pte_ofs++, addr += PAGE_SIZE) {
334                                     pgprot_t prot = PAGE_KERNEL;
335                                     /*
336                                      * first pass will use the same initial
337                                      * identity mapping attribute.
338                                      */
339                                     pgprot_t init_prot = __pgprot(PTE_IDENT_ATTR);
340
341                                     if (is_kernel_text(addr))
342                                             prot = PAGE_KERNEL_EXEC;
343
344                                     pages_4k++;
345                                     if (mapping_iter == 1) {
346                                             set_pte(pte, pfn_pte(pfn, init_prot));
347                                             last_map_addr = (pfn << PAGE_SHIFT) +
PAGE_SIZE;
348                                     } else
349                                             set_pte(pte, pfn_pte(pfn, prot));
350                             }
```

```
351                     }
352             }
353             if (mapping_iter == 1) {
354                     /*
355                      * update direct mapping page count only in the first
356                      * iteration.
357                      */
358                     update_page_count(PG_LEVEL_2M, pages_2m);
359                     update_page_count(PG_LEVEL_4K, pages_4k);
360
361                     /*
362                      * local global flush tlb, which will flush the previous
363                      * mappings present in both small and large page TLB's.
364                      */
365                     __flush_tlb_all();
366
367                     /*
368                      * Second iteration will set the actual desired PTE attributes.
369                      */
370                     mapping_iter = 2;
371                     goto repeat;
372             }
373             return last_map_addr;
374}
```

通过作者的注释， 可以了解到这个函数的作用是把整个物理内存地址都映射到从内核空间的开始地址，即从 0xc0000000 的整个内核空间中，直到物理内存映射完毕为止。这个函数比较长， 而且用到很多关于内存管理方面的宏定义，理解了这个函数， 就能大概理解内核是如何建立页表的，将这个抽象的模型完全的理解。

函数开始定义了 4 个变量 pgd_t *pgd， pmd_t *pmd， pte_t *pte， pfn；pgd 指向一个目录项开始的地址，pmd 指向一个中间目录开始的地址，pte 指向一个页表开始的地址 pfn 是页框号被初始为 0。注意这个页全局目录地址 swapper_pg_dir，在文件 arch/x86/kernel/head_32.S 第一次定义后，就再也没变过，前面 find_early_table_space 函数只是为页表分配新的空间，然后在这里为页表项赋值。看到 280 行，有个 pgd_index 宏，确定从页全局目录的哪个位置开始设置内核临时页表：

#define pgd_index(address) (((address) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))

我们得到的 pgd_idx 是 768，也就是从虚拟地址 0xC0000000 处开始映射，定位主内核页全局目录的起始项 pgd=768，也是内核要从目录表中第 768 个表项开始进行设置。从 768 到 1024 这个 256 个表项被 linux 内核设置成内核目录项，低 768 个目录项被用户空间使用。pgd = pgd_base + pgd_idx； pgd 便指向了第 768 个表项。

然后函数开始一个循环即开始填充从 768 到 1024 这 256 个目录项的内容。one_md_table_init() 函数根据 pgd 找到指向的 pmd 表。279 行 pfn = start_pfn，也就是为 0，物理地址从 0x0000 0000

开始，起始页框号(page frame number)为 pfn，因为我们只分析 4k 的那个 map_range 结构，这个结构的 start 是 0，end 是 max_low_pfn<<PAGE_SHIFT，也就是所有能使用页面的最后一个页面的地址。那么传递给 kernel_physical_mapping_init 的就是这样两个参数再加一个 mask。对这样一个范围，0~max_low_pfn，函数 282~352 行代码就设置页表和页目录的值。

285 行的 if (pfn >= end_pfn)很关键，前面讲了 max_low_pfn 代表着整个物理内存一共有多少页框。当 pfn 大于 max_low_pfn 的时候，表明内核已经把整个物理内存都映射到了系统空间中，所以剩下有没被填充的表项就直接忽略了。因为内核已经可以映射整个物理空间了，没必要继续填充剩下的表项。

紧接 293 行第 2 个 for 循环，在 linux 的 3 级映射模型中，是要设置 pmd 表的， 但在 2 级映射中忽略， 只循环一次，直接进行页表 pte 的设置。

315 行，is_kernel_text 函数根据前面提到的 addr 来判断 addr 线性地址是否属于内核代码段，它同样在 mm/init.c 中定义:

```
static inline int is_kernel_text(unsigned long addr)
{
    if (addr >= PAGE_OFFSET && addr <= (unsigned long)__init_end)
        return 1;
    return 0;
}
```

__init_end 是个内核符号，咱们很熟悉了，在内核链接的时候生成的，表示内核代码段的终止地址。如果 address 属于内核代码段， 那么在设置页表项的时候就要加个 PAGE_KERNEL_EXEC 属性，如果不是，则加个 PAGE_KERNEL 属性：

```
#define _PAGE_KERNEL_EXEC \
        (_PAGE_PRESENT | _PAGE_RW | _PAGE_DIRTY | _PAGE_ACCESSED)

#define _PAGE_KERNEL \
        (_PAGE_PRESENT | _PAGE_RW | _PAGE_DIRTY | _PAGE_ACCESSED | _PAGE_NX)
```

283 行，直接返回 pgd（pud、pmd 和 pgd 指向同一个页目录项，怀疑这句话的同学去看看 one_md_table_init 函数的内容就知道了）;295 行计算第 pfn 个页框对应的内核空间的线性地址；321 行填写一个页目录项 pmd(pgd)，并填写该目录项所对应的页表的所有项；346 行为页目录项 pmd 分配页表 pte，将该页表 pte 的物理地址写入 pmd 中，并初始化页表 pte 的每个页表项。最后通过 set_pte(pte, pfn_pte(pfn, PAGE_KERNEL));来设置页表项，先通过 pfn_pte 宏根据页框号和页表项的属性值合并成一个页表项值，然户在用 set_pte 宏把页表项值写到页表项里。

其实调用 early_ioremap_page_table_range_init()也实现了类似的功能。那么，调用完毕 kernel_physical_mapping_init 之后，4k 和 2MB 两种分页体系的页全局目录和页表就通过 set_pmd 和 set_pte 等咱们熟悉的动作建立完成了，涵盖了物理地址从 0 到 max_low_pfn<<PAGE_SHIFT，接下来使用 load_cr3 设置存器 cr3 的值为页全局目录的首地

址 swapper_pg_dir。

将控制 swapper_pg_dir 送入控制寄存器 cr3. 每当重新设置 cr3 时， CPU 就会将页面映射目录所在的页面装入 CPU 内部高速缓存中的 TLB 部分。现在内存中(实际上是高速缓存中）的映射目录变了，就要再让 CPU 装入一次。由于页面映射机制本来就是开启着的，所以从这条指令以后就扩大了系统空间中有映射区域的大小，使整个映射覆盖到整个物理内存(高端内存)除外。实际上此时 swapper_pg_dir 中已经改变的目录项很可能还在高速缓存中，所以还要通过__flush_tlb_all()将高速缓存中的内容冲刷到内存中，这样才能保证内存中映射目录内容的一致性。

最后，init_memory_mapping 返回了所映射页的数量值，并将值保存在 max_low_pfn_mapped 中。而 32 位 x86 体系中，max_pfn_mapped 的值也为 max_low_pfn_mapped。

5.2.5 建立内存管理架构

回到 setup_arch 函数的中，第 995 行调用 initmem_init 来启用初始化期间的内存管理器 early。这个函数在两个文件中有定义，arch/x86/mm/init_32.c 和 arch/x86/mm/numa_32.c，取决于是否启动了编译选项 CONFIG_NEED_MULTIPLE_NODES。这个编译选项是什么意思？这得从 NUMA 说起。NUMA 翻译成中文就叫“ 非对称内存访问体系 ”，其目的是为多 CPU，或大型计算机集群提供一个分布式内存访问环境，而每一个分布式节点就叫做 NODE。我们单机系统通常是 UMA，即“ 对称内存访问体系 ”，其实就是只有一个 NODE 的环境。

而每个 NODE 下物理内存分成几个 Zone（区域），Zone 再对物理页面进行管理。所以，不管是我们的 PC，还是大型集群服务器，只要安装了 Linux 操作系统，就是一个 NODE->Zone->Page 这样一个三层物理内存管理体系。

我的这台烂 PC，居然定义了这个选项的，所以实际上应该是调用 arch/x86/mm/numa_32.c 中的那个 initmem_init 函数。可能考虑到它是双核的原因吧，不过我们简单起见，只考虑仅有一个 NODE 的情况，所以仅对 arch/x86/mm/init_32.c 中的 initmem_init 函数进行分析，对 NUMA 感兴趣的可以去研究另一个 initmem_init 函数。

```
/* arch/x86/mm/init_32.c */
708void __init initmem_init(unsigned long start_pfn, unsigned long end_pfn,
709                                                  int acpi, int k8)
710{
711#ifdef CONFIG_HIGHMEM
712             highstart_pfn = highend_pfn = max_pfn;
713             if (max_pfn > max_low_pfn)
714                          highstart_pfn = max_low_pfn;
715             e820_register_active_regions(0, 0, highend_pfn);
716             sparse_memory_present_with_active_regions(0);
717             printk(KERN_NOTICE "%ldMB HIGHMEM available.\n",
718                          pages_to_mb(highend_pfn - highstart_pfn));
```

```
719             num_physpages = highend_pfn;
720             high_memory = (void *) __va(highstart_pfn * PAGE_SIZE - 1) + 1;
721#else
722             e820_register_active_regions(0, 0, max_low_pfn);
723             sparse_memory_present_with_active_regions(0);
724             num_physpages = max_low_pfn;
725             high_memory = (void *) __va(max_low_pfn * PAGE_SIZE - 1) + 1;
726#endif
727#ifdef CONFIG_FLATMEM
728             max_mapnr = num_physpages;
729#endif
730             __vmalloc_start_set = true;
731
732             printk(KERN_NOTICE "%ldMB LOWMEM available.\n",
733                             pages_to_mb(max_low_pfn));
734
735             setup_bootmem_allocator();
736}
```

这个函数首先调用 e820_register_active_regions 设置 early_node_map[]，传入给它的参数是 0、0 和 max_low_pfn（或者是 highend_pfn，取决于是否定义了编译选项 CONFIG_FLATMEM）：

```
932void __init e820_register_active_regions(int nid, unsigned long start_pfn,
933                                                 unsigned long last_pfn)
934{
935             unsigned long ei_startpfn;
936             unsigned long ei_endpfn;
937             int i;
938
939             for (i = 0; i < e820.nr_map; i++)
940                     if (e820_find_active_region(&e820.map[i],
941                                                 start_pfn, last_pfn,
942                                                 &ei_startpfn, &ei_endpfn))
943                             add_active_range(nid, ei_startpfn, ei_endpfn);
944}
```

e820_register_active_regions 函数的 940 行，调用 e820_find_active_region 函数，其中 ei_startpfn 和 ei_endpfn 是结果参数：

```
902int __init e820_find_active_region(const struct e820entry *ei,
903                                 unsigned long start_pfn,
904                                 unsigned long last_pfn,
905                                 unsigned long *ei_startpfn,
906                                 unsigned long *ei_endpfn)
```

```
907{
908             u64 align = PAGE_SIZE;
909
910             *ei_startpfn = round_up(ei->addr, align) >> PAGE_SHIFT;
911             *ei_endpfn = round_down(ei->addr + ei->size, align) >> PAGE_SHIFT;
912
913             /* Skip map entries smaller than a page */
914             if (*ei_startpfn >= *ei_endpfn)
915                     return 0;
916
917             /* Skip if map is outside the node */
918             if (ei->type != E820_RAM || *ei_endpfn <= start_pfn ||
919                                                     *ei_startpfn >= last_pfn)
920                     return 0;
921
922             /* Check for overlaps */
923             if (*ei_startpfn < start_pfn)
924                     *ei_startpfn = start_pfn;
925             if (*ei_endpfn > last_pfn)
926                     *ei_endpfn = last_pfn;
927
928             return 1;
929}
```

e820_find_active_region 函数的 910 行、911 行计算出 e820.map[i]元素的起始和结束页框的页号，并赋给结果参数 ei_startpfn 和 ei_endpfn。那么回到 e820_register_active_regions，943 行调用 add_active_range 函数，并传入参数 nid、ei_startpfn 和 ei_endpfn。这个 nid 是什么东西？在 initmem_init 中传入的是 0，表示 0 号 NODE。除此之外，传入 add_active_range 的参数还有刚才获得的 ei_startpfn 和 ei_endpfn 页框号，其中，ei_startpfn 就是 0：

```
3984void __init add_active_range(unsigned int nid, unsigned long start_pfn,
3985                                                    unsigned long end_pfn)
3986{
3987             int i;
3988
3989             mminit_dprintk(MMINIT_TRACE, "memory_register",
3990                                     "Entering add_active_range(%d, %#lx, %#lx) "
3991                                     "%d entries of %d used\n",
3992                                     nid, start_pfn, end_pfn,
3993                                     nr_nodemap_entries, MAX_ACTIVE_REGIONS);
3994
3995             mminit_validate_memmodel_limits(&start_pfn, &end_pfn);
3996
3997             /* Merge with existing active regions if possible */
```

```
3998        for (i = 0; i < nr_nodemap_entries; i++) {
3999                if (early_node_map[i].nid != nid)
4000                        continue;
4001
4002                /* Skip if an existing region covers this new one */
4003                if (start_pfn >= early_node_map[i].start_pfn &&
4004                                end_pfn <= early_node_map[i].end_pfn)
4005                        return;
4006
4007                /* Merge forward if suitable */
4008                if (start_pfn <= early_node_map[i].end_pfn &&
4009                                end_pfn > early_node_map[i].end_pfn) {
4010                        early_node_map[i].end_pfn = end_pfn;
4011                        return;
4012                }
4013
4014                /* Merge backward if suitable */
4015                if (start_pfn < early_node_map[i].start_pfn &&
4016                                end_pfn >= early_node_map[i].start_pfn) {
4017                        early_node_map[i].start_pfn = start_pfn;
4018                        return;
4019                }
4020        }
4021
4022        /* Check that early_node_map is large enough */
4023        if (i >= MAX_ACTIVE_REGIONS) {
4024                printk(KERN_CRIT "More than %d memory regions, truncating\n",
4025                                                        MAX_ACTIVE_REGIONS);
4026                return;
4027        }
4028
4029        early_node_map[i].nid = nid;
4030        early_node_map[i].start_pfn = start_pfn;
4031        early_node_map[i].end_pfn = end_pfn;
4032        nr_nodemap_entries = i + 1;
4033}
```

3989 – 3995 行代码主要是打印一些相关信息，我们主要还是关注全局变量 early_node_map 数组是怎样被初始化的。那么 early_node_map 到底是个什么东西呢？这个东西定义在 mm/page_alloc.c：

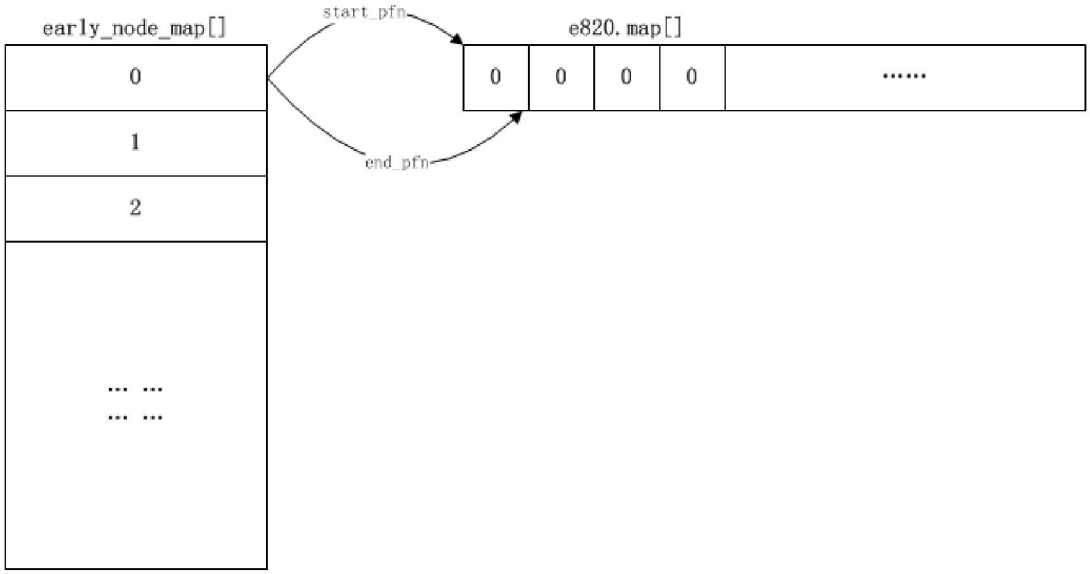
```
static struct node_active_region __meminitdata early_node_map[MAX_ACTIVE_REGIONS];
struct node_active_region {
    unsigned long start_pfn;
    unsigned long end_pfn;
```

```
    int nid;
};
```

这里关键是数组的大小，也就是 MAX_ACTIVE_REGIONS 的值，代表 NUMA 活动节点的最大数量，本来应该等于 CONFIG_MAX_ACTIVE_REGIONS。但我们在.config 中没有配置它，所以它的值为 256。

nr_nodemap_entries，也是一个全局的值，它的值在编译的时候被设置为 1，那么函数 3998 – 4020 测试一下 start_pfn 和 end_pfn 是否与 early_node_map[]的某个元素重合，如果有重合，则返回。而我们这里，early_node_map[0]也是空的，全是 0，所以直接来到 4029 行，early_node_map[0].nid 给赋成 0；early_node_map[i].start_pfn 给赋值成 start_pfn，我们传进来的是 e820.map[i]起始地址对应的页号；early_node_map[i].end_pfn 给赋值成 end_pfn，我们传进来的是 e820.map[i]最后一个页框号；最后 nr_nodemap_entries 加 1，返回。返回后，全局变量 nr_nodemap_entries 就是全局数组 early_node_map 的 size。

回到 e820_register_active_regions，最后 939 行 for 循环执行完成后，全局 early_node_map 数组就存放了每个可用 e820 RAM 的起始地址和结束地址对应的页框号，他们的 nid 都是 0，共有 e820.nr_map 个元素；全局变量 nr_nodemap_entries 的值最后也变成了 e820 RAM 可用内存区的总数 e820.nr_map：



initmem_init 函数在设置好全局变量 early_node_map[]数组之后，随后 716 或 723 行调用的是 sparse_memory_present_with_active_regions(0)：

```
3471void __init sparse_memory_present_with_active_regions(int nid)
3472{
3473            int i;
3474
3475            for_each_active_range_index_in_nid(i, nid)
3476                        memory_present(early_node_map[i].nid,
3477                                                early_node_map[i].start_pfn,
```

```
3478                                    early_node_map[i].end_pfn);
3479}
```

3475 行，有一个 for_each_active_range_index_in_nid 宏，这个宏后面会经常用到，这里只讲一次，来自 mm/page_alloc.c：

```
#define for_each_active_range_index_in_nid(i, nid) \
        for (i = first_active_region_index_in_nid(nid); i != -1; \
                                i = next_active_region_index_in_nid(i, nid))
```

其中，first_active_region_index_in_nid 函数定义为：

```
static int __meminit first_active_region_index_in_nid(int nid)
{
    int i;
    for (i = 0; i < nr_nodemap_entries; i++)
        if (nid == MAX_NUMNODES || early_node_map[i].nid == nid)
            return i;
    return -1;
}
```

也就是说，i 的初始值为该函数的返回值，取决于 early_node_map[0].nid 的值。由于我们也没有配置 CONFIG_NODES_SHIFT，MAX_NUMNODES 为 1；全局变量 early_node_map 的所有 nid 都是 0，所以 first_active_region_index_in_nid 返回 0。

而 next_active_region_index_in_nid 函数定义如下：

```
static int __meminit next_active_region_index_in_nid(int index, int nid)
{
    for (index = index + 1; index < nr_nodemap_entries; index++)
        if (nid == MAX_NUMNODES || early_node_map[index].nid == nid)
            return index;

    return -1;
}
```

除非 i > nr_nodemap_entries，否则它将返回 index，即从 1 到 nr_nodemap_entries，也就是遍历了所有 early_node_map 数组的元素。for_each_active_range_index_in_nid 就是这么一个循环。这说明了一个什么问题？那就是验证了前面的那句话，笔者的这个单机 PC 的 NUMA 只有一个 NODE。我们没有配置 CONFIG_HAVE_MEMORY_PRESENT，所以 memory_present 是一个空函数。

回到 initmem_init 中，第三步是调用 setup_bootmem_allocator()函数来设置引导启动阶段所涉及到的页映射位：

```
775void __init setup_bootmem_allocator(void)
 776{
 777#ifndef CONFIG_NO_BOOTMEM
 778        int nodeid;
```

```
 779             unsigned long bootmap_size, bootmap;
 780             /*
 781              * Initialize the boot-time allocator (with low memory only):
 782              */
 783             bootmap_size = bootmem_bootmap_pages(max_low_pfn)<<PAGE_SHIFT;
 784             bootmap = find_e820_area(0, max_pfn_mapped<<PAGE_SHIFT, bootmap_size,
 785                                                      PAGE_SIZE);
 786             if (bootmap == -1L)
 787                     panic("Cannot find bootmem map of size %ld\n", bootmap_size);
 788             reserve_early(bootmap, bootmap + bootmap_size, "BOOTMAP");
 789#endif
 790
 791             printk(KERN_INFO "    mapped low ram: 0 - %08lx\n",
 792                      max_pfn_mapped<<PAGE_SHIFT);
 793             printk(KERN_INFO "    low ram: 0 - %08lx\n", max_low_pfn<<PAGE_SHIFT);
 794
 795#ifndef CONFIG_NO_BOOTMEM
 796             for_each_online_node(nodeid) {
 797                      unsigned long start_pfn, end_pfn;
 798
 799#ifdef CONFIG_NEED_MULTIPLE_NODES
……
 806#else
 807                      start_pfn = 0;
 808                      end_pfn = max_low_pfn;
 809#endif
 810                      bootmap = setup_node_bootmem(nodeid, start_pfn, end_pfn,
 811                                                                  bootmap);
 812             }
 813#endif
 814
 815             after_bootmem = 1;
 816}
```

我们看到，由于我们.config 文件中配置了 CONFIG_NO_BOOTMEM，所以这个函数也是什么都不干，仅仅打印几个信息并把全局变量 after_init_bootmem 设置为 1。这个函数结束后，我们的 initmem_init 也就结束了，至此，内核永久页表建立好了，并且初始化时期内存管理相关的数据结构框架都搭建起来了，下面就该向这个框架内“ 添砖加瓦 ”了。

5.2.6 添砖加瓦

回到 setup_arch，来到 1007，调用 paging_init()进行页面初始化。

```
825void __init paging_init(void)
826{
827             pagetable_init();
828
829             __flush_tlb_all();
830
831             kmap_init();
832
833             /*
834               * NOTE: at this point the bootmem allocator is fully available.
835               */
836             sparse_init();
837             zone_sizes_init();
838}
```

我们看到，paging_init()分别调用下面的几个函数来实现页面的初始化工作：
pagetable_init();
kmap_init();
sparse_init();
zone_sizes_init();

首先 pagetable_init()初始化页全局目录的目录项：

```
544 static void __init pagetable_init(void)
545 {
546         pgd_t *pgd_base = swapper_pg_dir;
547
548         permanent_kmaps_init(pgd_base);
549 }
```

如果 CONFIG_HIGHMEM 被配置（当前大多数系统肯定都是大内存，都会有配置），则调用 permanent_kmaps_init 函数，把页全局目录 swapper_pg_dir 地址传给他。该函数调用 page_table_range_init 设置目录项。随后 paging_init 的两个函数 kmap_init() 和 sparse_init() 分别对第一个页表项进行缓存，以及映射非线性的段（也就是不连续的段）。至于高端内存的映射，请查看博客“ 高端内存映射 ”
http://blog.csdn.net/yunsongice/archive/2010/01/26/5258589.aspx
相关内容。

paging_init 的最后调用 zone_sizes_init()：

```
739static void __init zone_sizes_init(void)
740{
741             unsigned long max_zone_pfns[MAX_NR_ZONES];
742             memset(max_zone_pfns, 0, sizeof(max_zone_pfns));
```

```
743             max_zone_pfns[ZONE_DMA] =
744                     virt_to_phys((char *)MAX_DMA_ADDRESS) >> PAGE_SHIFT;
745             max_zone_pfns[ZONE_NORMAL] = max_low_pfn;
746#ifdef CONFIG_HIGHMEM
747             max_zone_pfns[ZONE_HIGHMEM] = highend_pfn;
748#endif
749
750             free_area_init_nodes(max_zone_pfns);
751}
```

zone_sizes_init 的 741~749 行设置一个内部变量 max_zone_pfns 数组其只有三个元素：ZONE_DMA、ZONE_NORMAL 和 ZONE_HIGHMEM，表示我们 NUMA 体系中一个 NODE 的三个 Zone。那么元素的具体值就是每个 Zone 的理论上能使用的页框数最大值。

最后 750 行调用 free_area_init_nodes()来实现节点空闲区域的初始化，这函数很复杂，其本质上就等于：

```
void __init free_area_init_nodes(unsigned long *max_zone_pfn)
{
    for_each_online_node(nid) {
        pg_data_t *pgdat = NODE_DATA(nid);
        free_area_init_node(nid, NULL,
                find_min_pfn_for_node(nid), NULL);

        /* Any memory on that node */
        if (pgdat->node_present_pages)
            node_set_state(nid, N_HIGH_MEMORY);
        check_for_regular_memory(pgdat);
    }
}
```

for_each_online_node 是什么呢？它定义为：

```
#define for_each_online_node(node) for_each_node_state(node, N_ONLINE)
#define for_each_node_state(__node, __state) \
    for_each_node_mask((__node), node_states[__state])
#define for_each_node_mask(node, mask)                  \
    for ((node) = first_node(mask);                 \
        (node) < MAX_NUMNODES;                  \
        (node) = next_node((node), (mask)))
```

而 first_node 和 next_node 又分别是：

```
#define first_node(src) __first_node(&(src))
static inline int __first_node(const nodemask_t *srcp)
{
    return min_t(int, MAX_NUMNODES, find_first_bit(srcp->bits, MAX_NUMNODES));
```

```
}
#define next_node(n, src) __next_node((n), &(src))
static inline int __next_node(int n, const nodemask_t *srcp)
{
	return min_t(int,MAX_NUMNODES,find_next_bit(srcp->bits, MAX_NUMNODES, n+1));
}
```

好了，由于我们假设只有一个节点，所以这也是一个只执行一次的循环。这里有个奇怪的 NODE_DATA(nid)，这个 nid 我们知道，是 0。那么 NODE_DATA 宏是什么？来看它的定义：

```
extern struct pglist_data *node_data[];
#define NODE_DATA(nid)    (node_data[nid])
```

来了，最著名的 pglist_data 结构出现了：

```
600typedef struct pglist_data {
601             struct zone node_zones[MAX_NR_ZONES];
602             struct zonelist node_zonelists[MAX_ZONELISTS];
603             int nr_zones;
604#ifdef CONFIG_FLAT_NODE_MEM_MAP /* means !SPARSEMEM */
605             struct page *node_mem_map;
606#ifdef CONFIG_CGROUP_MEM_RES_CTLR
607             struct page_cgroup *node_page_cgroup;
608#endif
609#endif
610#ifndef CONFIG_NO_BOOTMEM
611             struct bootmem_data *bdata;
612#endif
613#ifdef CONFIG_MEMORY_HOTPLUG
614             /*
615               * Must be held any time you expect node_start_pfn, node_present_pages
616               * or node_spanned_pages stay constant.    Holding this will also
617               * guarantee that any pfn_valid() stays that way.
618               *
619               * Nests above zone->lock and zone->size_seqlock.
620               */
621             spinlock_t node_size_lock;
622#endif
623             unsigned long node_start_pfn;
624             unsigned long node_present_pages; /* total number of physical pages */
625             unsigned long node_spanned_pages; /* total size of physical page
626                                                  range, including holes */
627             int node_id;
628             wait_queue_head_t kswapd_wait;
629             struct task_struct *kswapd;
630             int kswapd_max_order;
```

```
631} pg_data_t;
```

这个结构是每个 NUMA 节点一个，那么我 PC 上唯一的 0 号 NODE 就是 NODE_DATA(0)，也就是全局数组 node_data[]的 0 号元素。

再来看接下来的 free_area_init_node 函数，来自 mm/page_alloc.c：

```
3932void __paginginit free_area_init_node(int nid, unsigned long *zones_size,
3933                 unsigned long node_start_pfn, unsigned long *zholes_size)
3934{
3935         pg_data_t *pgdat = NODE_DATA(nid);
3936
3937         pgdat->node_id = nid;
3938         pgdat->node_start_pfn = node_start_pfn;
3939         calculate_node_totalpages(pgdat, zones_size, zholes_size);
3940
3941         alloc_node_mem_map(pgdat);
3942#ifdef CONFIG_FLAT_NODE_MEM_MAP
3943                 printk(KERN_DEBUG "free_area_init_node: node %d, pgdat %08lx,
node_mem_map %08lx\n",
3944                 nid, (unsigned long)pgdat,
3945                 (unsigned long)pgdat->node_mem_map);
3946#endif
3947
3948         free_area_init_core(pgdat, zones_size, zholes_size);
3949}
```

传递给他的参数作为整个 pg_data_t 的第一个页号，由函数 find_min_pfn_for_node(0)计算出：

```
static unsigned long __init find_min_pfn_for_node(int nid)
{
    int i;
    unsigned long min_pfn = ULONG_MAX;

    /* Assuming a sorted map, the first range found has the starting pfn */
    for_each_active_range_index_in_nid(i, nid)
        min_pfn = min(min_pfn, early_node_map[i].start_pfn);

    if (min_pfn == ULONG_MAX) {
        printk(KERN_WARNING
            "Could not find start_pfn for node %d\n", nid);
        return 0;
    }

    return min_pfn;
```

}
其实很简单，就是位于 early_node_map[]数组中第一个可用的页号（ULONG_MAX 是 32 个 1），即第一个空闲页面的页号。注意这里，最后返回的其实是 early_node_map[]数组中，start_pfn 最小的那个元素的值。

3935 行的 pgdat 我们已经知道了，随后初始化它的两个字段：NUMA 节点号和第一个 Zone 的第一个页框号。接着 3939 行计算 pgdat 的所有 Zone 的总共页面数和可用页面数，针对我们的 x86 体系 MAX_NR_ZONES 等于 3：

```
3699static void __meminit calculate_node_totalpages(struct pglist_data *pgdat,
3700                        unsigned long *zones_size, unsigned long *zholes_size)
3701{
3702        unsigned long realtotalpages, totalpages = 0;
3703        enum zone_type i;
3704
3705        for (i = 0; i < MAX_NR_ZONES; i++)
3706                totalpages += zone_spanned_pages_in_node(pgdat->node_id, i,
3707                                                                zones_size);
3708        pgdat->node_spanned_pages = totalpages;
3709
3710        realtotalpages = totalpages;
3711        for (i = 0; i < MAX_NR_ZONES; i++)
3712                realtotalpages -=
3713                        zone_absent_pages_in_node(pgdat->node_id, i,
3714                                                                zholes_size);
3715        pgdat->node_present_pages = realtotalpages;
3716        printk(KERN_DEBUG "On node %d totalpages: %lu\n", pgdat->node_id,
3717                                                        realtotalpages);
3718}
```

然后 free_area_init_node 的 3941 行，调用 alloc_node_mem_map()为我们的 NODE(0)的所有页描述符 page 分配空闲区域：

```
3891static void __init_refok alloc_node_mem_map(struct pglist_data *pgdat)
3892{
3893        /* Skip empty nodes */
3894        if (!pgdat->node_spanned_pages)
3895                return;
3896
3897#ifdef CONFIG_FLAT_NODE_MEM_MAP
3898        /* ia64 gets its own node_mem_map, before this, without bootmem */
3899        if (!pgdat->node_mem_map) {
3900                unsigned long size, start, end;
3901                struct page *map;
```

```
3902
3903                    /*
3904                     * The zone's endpoints aren't required to be MAX_ORDER
3905                     * aligned but the node_mem_map endpoints must be in order
3906                     * for the buddy allocator to function correctly.
3907                     */
3908                    start = pgdat->node_start_pfn & ~(MAX_ORDER_NR_PAGES - 1);
3909                    end = pgdat->node_start_pfn + pgdat->node_spanned_pages;
3910                    end = ALIGN(end, MAX_ORDER_NR_PAGES);
3911                    size =  (end - start) * sizeof(struct page);
3912                    map = alloc_remap(pgdat->node_id, size);
3913                    if (!map)
3914                            map = alloc_bootmem_node(pgdat, size);
3915                    pgdat->node_mem_map = map + (pgdat->node_start_pfn - start);
3916            }
3917#ifndef CONFIG_NEED_MULTIPLE_NODES
3918            /*
3919             * With no DISCONTIG, the global mem_map is just set as node 0's
3920             */
3921            if (pgdat == NODE_DATA(0)) {
3922                    mem_map = NODE_DATA(0)->node_mem_map;
3923#ifdef CONFIG_ARCH_POPULATES_NODE_MAP
3924                    if (page_to_pfn(mem_map) != pgdat->node_start_pfn)
3925                            mem_map  -=  (pgdat->node_start_pfn  -
ARCH_PFN_OFFSET);
3926#endif /* CONFIG_ARCH_POPULATES_NODE_MAP */
3927            }
3928#endif
3929#endif /* CONFIG_FLAT_NODE_MEM_MAP */
3930}
```

看到 3897 行，我在.config 中找了找，没有看到 CONFIG_FLAT_NODE_MEM_MAP 配置选项，估计是最新的内核已经不把 FLATMEM 作为默认内存模型了。既然提到了内存模型（Memory Model），就多给大家补充一下这方面的知识。什么是内存模型，针对 x86 体系就是对页面的管理算法及其相关的数据结构。Linux 目前支持三种内存管理模型：FLATMEM（平坦模型）、DISCONTIGMEM（折扣模型）和 SPARSEMEM（稀疏模型）。不知道从哪个 Linux 版本以后，就把 SPARSEMEM 作为默认的内存模型了。

FLATMEM 模型将所有的页描述符，全都放到全局数组 mem_map 中：
struct page *mem_map;
mem_map 的每一个元素就是一个 page 结构，那么当我要获得一个页面的信息，就把对应页面的页号 pfn 加上首地址 mem_map 就能得到对应的页描述符结构 page。

DISCONTIGMEM 和 SPARSEMEM 模型则要复杂得多，牵涉到一个 vmemmap 全局变量和

mem_section 数据结构。笔者对这两个模型没有深入的研究，所以不敢妄言。下面就假设我们配置了 CONFIG_FLAT_NODE_MEM_MAP，针对 FLATMEM 模型来对余下的代码进行讲解。

3899 行 pgdat->node_mem_map 还没有初始化，肯定是空的，所以进入条件分支。设置内部变量 start、end、size 分别为本节点 pglist_data 的第一个可用页面页号 node_start_pfn，最后一个可用页面页号，以及总的 page 大小。3912 行首先尝试一下 alloc_remap 分配，由于我们没有配置 CONFIG_HAVE_ARCH_ALLOC_REMAP，所以肯定会失败，那么肯定会调用 alloc_bootmem_node 宏进行分配：

```
#define alloc_bootmem_node(pgdat, x) \
        __alloc_bootmem_node(pgdat, x, SMP_CACHE_BYTES, __pa(MAX_DMA_ADDRESS))
```

注意 SMP_CACHE_BYTES 是 L1 Cache 相关的宏，保证存取的高效率。而后面的那个 MAX_DMA_ADDRESS：

```
#define MAX_DMA_ADDRESS         (PAGE_OFFSET + 0x1000000)
```

也就是说等于 0xc0000000 + 0x1000000=0xc1000000。

这个宏最终会走到__alloc_bootmem_node 函数，传递给他的参数是：本节点 pglist_data、需求大小 size、缓存对齐参数 SMP_CACHE_BYTES 和 DMA 最大地址 0xc1000000：

```
 833void * __init __alloc_bootmem_node(pg_data_t *pgdat, unsigned long size,
 834                                    unsigned long align, unsigned long goal)
 835{
 836             if (WARN_ON_ONCE(slab_is_available()))
 837                         return kzalloc_node(size, GFP_NOWAIT, pgdat->node_id);
 838
 839#ifdef CONFIG_NO_BOOTMEM
 840             return __alloc_memory_core_early(pgdat->node_id, size, align,
 841                                              goal, -1ULL);
 842#else
 843             return ___alloc_bootmem_node(pgdat->bdata, size, align, goal, 0);
 844#endif
 845}
```

836 行，WARN_ON_ONCE(slab_is_available())，内有只要有万分之一可能性都要去尝试一下，比如这里有可能一个计算集群中的其他节点先于本地节点初始化了一个 slab 分配器，就去尝试使用它来分配。

我们配置了 CONFIG_NO_BOOTMEM，所以调用__alloc_memory_core_early 函数：

```
3412void * __init __alloc_memory_core_early(int nid, u64 size, u64 align,
3413                                        u64 goal, u64 limit)
3414{
3415             int i;
```

```
3416        void *ptr;
3417
3418        /* need to go over early_node_map to find out good range for node */
3419        for_each_active_range_index_in_nid(i, nid) {
3420                u64 addr;
3421                u64 ei_start, ei_last;
3422
3423                ei_last = early_node_map[i].end_pfn;
3424                ei_last <<= PAGE_SHIFT;
3425                ei_start = early_node_map[i].start_pfn;
3426                ei_start <<= PAGE_SHIFT;
3427                addr = find_early_area(ei_start, ei_last,
3428                                                goal, limit, size, align);
3429
3430                if (addr == -1ULL)
3431                        continue;
3432
3433#if 0
3434                        printk(KERN_DEBUG "alloc (nid=%d %llx - %llx) (%llx
- %llx) %llx %llx => %llx\n",
3435                                        nid,
3436                                        ei_start, ei_last, goal, limit, size,
3437                                        align, addr);
3438#endif
3439
3440                ptr = phys_to_virt(addr);
3441                memset(ptr, 0, size);
3442                reserve_early_without_check(addr, addr + size, "BOOTMEM");
3443                return ptr;
3444        }
3445
3446        return NULL;
3447}
```

for_each_active_range_index_in_nid(i, nid)我们见过了，遍历所有 early_node_map 数组的元素。3427 行，调用最低级的函数 find_early_area 在 early_node_map[]数组的某个元素中找到一个可用的空间，来存放 mem_map 数组，对这个函数还有一些疑惑的同学可以在“着手建立永久内核页表”一节中找到答案。

最后，通过 pgdat->node_mem_map = map + (pgdat->node_start_pfn - start)，将 pglist_data 的 node_mem_map 字段指向这个 size 大小的 page 区域。

存放所有页描述符 page 的空间有了，回到 free_area_init_node 函数中。3948 行，调用 free_area_init_core，传递给他的参数是本节点的 pglist_data、为 NULL 的 zones_size 和

zholes_size：

```
3793/*
3794 * Set up the zone data structures:
3795 *     - mark all pages reserved
3796 *     - mark all memory queues empty
3797 *     - clear the memory bitmaps
3798 */
3799static void __paginginit free_area_init_core(struct pglist_data *pgdat,
3800                 unsigned long *zones_size, unsigned long *zholes_size)
3801{
3802         enum zone_type j;
3803         int nid = pgdat->node_id;
3804         unsigned long zone_start_pfn = pgdat->node_start_pfn;
3805         int ret;
3806
3807         pgdat_resize_init(pgdat);
3808         pgdat->nr_zones = 0;
3809         init_waitqueue_head(&pgdat->kswapd_wait);
3810         pgdat->kswapd_max_order = 0;
3811         pgdat_page_cgroup_init(pgdat);
3812
3813         for (j = 0; j < MAX_NR_ZONES; j++) {
3814                 struct zone *zone = pgdat->node_zones + j;
3815                 unsigned long size, realsize, memmap_pages;
3816                 enum lru_list l;
3817
3818                 size = zone_spanned_pages_in_node(nid, j, zones_size);
3819                 realsize = size - zone_absent_pages_in_node(nid, j,
3820                                                                 zholes_size);
3821
3822                 /*
3823                  * Adjust realsize so that it accounts for how much memory
3824                  * is used by this zone for memmap. This affects the watermark
3825                  * and per-cpu initialisations
3826                  */
3827                 memmap_pages =
3828                         PAGE_ALIGN(size * sizeof(struct page)) >> PAGE_SHIFT;
3829                 if (realsize >= memmap_pages) {
3830                         realsize -= memmap_pages;
3831                         if (memmap_pages)
3832                                 printk(KERN_DEBUG
3833                                        "  %s zone: %lu pages used for
memmap\n",
```

```
3834                                            zone_names[j], memmap_pages);
3835                    } else
3836                            printk(KERN_WARNING
3837                                    "    %s zone: %lu pages exceeds realsize %lu\n",
3838                                    zone_names[j], memmap_pages, realsize);
3839
3840                    /* Account for reserved pages */
3841                    if (j == 0 && realsize > dma_reserve) {
3842                            realsize -= dma_reserve;
3843                            printk(KERN_DEBUG "    %s zone: %lu pages reserved\n",
3844                                            zone_names[0], dma_reserve);
3845                    }
3846
3847                    if (!is_highmem_idx(j))
3848                            nr_kernel_pages += realsize;
3849                    nr_all_pages += realsize;
3850
3851                    zone->spanned_pages = size;
3852                    zone->present_pages = realsize;
3853#ifdef CONFIG_NUMA
3854                    zone->node = nid;
3855                    zone->min_unmapped_pages = (realsize*sysctl_min_unmapped_ratio)
3856                                                    / 100;
3857                    zone->min_slab_pages = (realsize * sysctl_min_slab_ratio) / 100;
3858#endif
3859                    zone->name = zone_names[j];
3860                    spin_lock_init(&zone->lock);
3861                    spin_lock_init(&zone->lru_lock);
3862                    zone_seqlock_init(zone);
3863                    zone->zone_pgdat = pgdat;
3864
3865                    zone->prev_priority = DEF_PRIORITY;
3866
3867                    zone_pcp_init(zone);
3868                    for_each_lru(l) {
3869                            INIT_LIST_HEAD(&zone->lru[l].list);
3870                            zone->reclaim_stat.nr_saved_scan[l] = 0;
3871                    }
3872                    zone->reclaim_stat.recent_rotated[0] = 0;
3873                    zone->reclaim_stat.recent_rotated[1] = 0;
3874                    zone->reclaim_stat.recent_scanned[0] = 0;
3875                    zone->reclaim_stat.recent_scanned[1] = 0;
3876                    zap_zone_vm_stats(zone);
3877                    zone->flags = 0;
```

```
3878                    if (!size)
3879                            continue;
3880
3881                    set_pageblock_order(pageblock_default_order());
3882                    setup_usemap(pgdat, zone, size);
3883                    ret = init_currently_empty_zone(zone, zone_start_pfn,
3884                                                    size, MEMMAP_EARLY);
3885                    BUG_ON(ret);
3886                    memmap_init(size, nid, j, zone_start_pfn);
3887                    zone_start_pfn += size;
3888            }
3889}
```

3807 行，调用 pgdat_resize_init 函数初始化 pglist_data 的自旋锁：

```
void pgdat_resize_init(struct pglist_data *pgdat)
{
        spin_lock_init(&pgdat->node_size_lock);
}
```

3809 行，初始化 pglist_data 的 kswapd_wait 字段，作为等待队列的头。有关等待队列的知识如果还欠缺的话，请访问博客“非运行状态进程的组织”http://blog.csdn.net/yunsongice/archive/2010/04/25/5526070.aspx。

3813，进入一个循环，循环次数 MAX_NR_ZONES，我们知道是 3。每个 NODE 的 Zone 信息都存放在 pglist_data 的 node_zones 数组中，所以 3814 首先获得每个元素的地址，赋给内部变量 zone。随后 zone_spanned_pages_in_node 为根据传递进来的 zholes_size 去掉一些 page。由于我们传递进来的是 NULL，所以这个函数啥也不做，得到的 size 实际上就是页框 page 的总量，即 early_node_map[i].end_pfn 减去 early_node_map[i].start_pfn 的结果：

```
void __meminit get_pfn_range_for_nid(unsigned int nid,
                        unsigned long *start_pfn, unsigned long *end_pfn)
{
        int i;
        *start_pfn = -1UL;
        *end_pfn = 0;

        for_each_active_range_index_in_nid(i, nid) {
                *start_pfn = min(*start_pfn, early_node_map[i].start_pfn);
                *end_pfn = max(*end_pfn, early_node_map[i].end_pfn);
        }

        if (*start_pfn == -1UL)
                *start_pfn = 0;
}
```

继续走，3819 行 zone_absent_pages_in_node 也是个空函数，所以 realsize 与 size 的值相等。3827 行，计算出所有页框的大小：PAGE_ALIGN(size * sizeof(struct page)) >> PAGE_SHIFT。结果左移 PAGE_SHIFT 位的意思我们很熟悉了，就是计算出要装下 size 个页描述符需要几个实际的页框，最后把这个数赋值给内部变量 memmap_pages。这个逻辑关系千万要注意！

继续，2829 - 3838 其实是个出错信息的打印，因为如果计算出来的 memmap_pages 比现在已经有的 realsize 还大，那我 PC 内存可就太小了，可以退休了。3841 - 3849，让 realsize 去掉一些保留的页面并增加一些内核使用的页面数量，最后将计算出的总的页框数量 size 和实际使用的页框数量 realsize 赋值给 zone 的 spanned_pages 字段和 present_pages。

后面 3853 - 3885 初始化每个 pgdat->node_zones 数组的其他字段，最后 3886 行调用 memmap_init(size, nid, j, zone_start_pfn)初始化所有的页描述符：

```
#ifndef __HAVE_ARCH_MEMMAP_INIT
#define memmap_init(size, nid, zone, start_pfn) \
    memmap_init_zone((size), (nid), (zone), (start_pfn), MEMMAP_EARLY)
#endif
```

.config 文件中找不到__HAVE_ARCH_MEMMAP_INIT，所以内核调用 memmap_init_zone()初始化所有的页描述符，下面是具体的初始化，传入的参数是总页框数 size，节点号 nid（为 0），zone 号（0~2），以及第一个可用页面的页号 zone_start_pfn：

```
2995void __meminit memmap_init_zone(unsigned long size, int nid, unsigned long zone,
2996                 unsigned long start_pfn, enum memmap_context context)
2997{
2998         struct page *page;
2999         unsigned long end_pfn = start_pfn + size;
3000         unsigned long pfn;
3001         struct zone *z;
3002
3003         if (highest_memmap_pfn < end_pfn - 1)
3004                 highest_memmap_pfn = end_pfn - 1;
3005
3006         z = &NODE_DATA(nid)->node_zones[zone];
3007         for (pfn = start_pfn; pfn < end_pfn; pfn++) {
3008                 /*
3009                  * There can be holes in boot-time mem_map[]s
3010                  * handed to this function.    They do not
3011                  * exist on hotplugged memory.
3012                  */
3013                 if (context == MEMMAP_EARLY) {
3014                         if (!early_pfn_valid(pfn))
3015                                 continue;
3016                         if (!early_pfn_in_nid(pfn, nid))
3017                                 continue;
```

```
3018                    }
3019                    page = pfn_to_page(pfn);
3020                    set_page_links(page, zone, nid, pfn);
3021                    mminit_verify_page_links(page, zone, nid, pfn);
3022                    init_page_count(page);
3023                    reset_page_mapcount(page);
3024                    SetPageReserved(page);
3025                    /*
3026                     * Mark the block movable so that blocks are reserved for
3027                     * movable at startup. This will force kernel allocations
3028                     * to reserve their blocks rather than leaking throughout
3029                     * the address space during boot when many long-lived
3030                     * kernel allocations are made. Later some blocks near
3031                     * the start are marked MIGRATE_RESERVE by
3032                     * setup_zone_migrate_reserve()
3033                     *
3034                     * bitmap is created for zone's valid pfn range. but memmap
3035                     * can be created for invalid pages (for alignment)
3036                     * check here not to call set_pageblock_migratetype() against
3037                     * pfn out of zone.
3038                     */
3039                    if ((z->zone_start_pfn <= pfn)
3040                        && (pfn < z->zone_start_pfn + z->spanned_pages)
3041                        && !(pfn & (pageblock_nr_pages - 1)))
3042                            set_pageblock_migratetype(page, MIGRATE_MOVABLE);
3043
3044                    INIT_LIST_HEAD(&page->lru);
3045#ifdef WANT_PAGE_VIRTUAL
3046                    /* The shift won't overflow because ZONE_NORMAL is below 4G. */
3047                    if (!is_highmem_idx(zone))
3048                            set_page_address(page, __va(pfn << PAGE_SHIFT));
3049#endif
3050            }
3051}
```

前面 alloc_node_mem_map 为我们的 NODE(0)的所有页描述符 page 分配空闲区域，并由全局变量 mem_map 指向，所以 3007 行 for (pfn = start_pfn; pfn < end_pfn; pfn++) 对每个空闲页框，都指向一系列初始化的操作，其中 3019 行通过宏 pfn_to_page(pfn) 得到页框号对应的页描述符 page 的地址，并把它赋给内部变量 page：

#define __pfn_to_page(pfn) (mem_map + ((pfn) - ARCH_PFN_OFFSET))

后面的代码就是对这个页描述符进行一系列的初始化。

到此为止，setup_arch()中比较重要的内容基本介绍完毕，主要涉及内存管理系统的一些底层初始化代码。经过这一番初始化后，内存初始化已经完成了特定系统的基本初始化，接下来，

start_kernel 开始构建内存管理的经典算法应用：伙伴系统和 slab 内存管理。对这两个概念不太熟悉的同学请查阅博客“伙伴系统算法”
http://blog.csdn.net/yunsongice/archive/2010/01/22/5225155.aspx
和“slab 分配器”http://blog.csdn.net/yunsongice/archive/2010/01/30/5272715.aspx。

5.3 设置每 CPU 环境

回到 start_kernel，563 行调用 mm_init_owner 函数，将 init_mm 的 owner 字段指回 init_task。这个函数可以说进入 start_kernel 以来最简单的函数了。继续走，setup_command_line 也很简单：

```
static void __init setup_command_line(char *command_line)
{
	saved_command_line = alloc_bootmem(strlen (boot_command_line)+1);
	static_command_line = alloc_bootmem(strlen (command_line)+1);
	strcpy (saved_command_line, boot_command_line);
	strcpy (static_command_line, command_line);
}
```

把刚才在 setup_arch()中拷贝进来的 command_line，拷贝到全局变量 saved_command_line 和 static_command_line 所指向的内存单元中。这个内存单元通过 alloc_bootmem 函数在刚刚建立好的内存管理环境中进行分配。

继续走，565 行，setup_nr_cpu_ids()函数，在多 CPU 情况下，调用同一文件中的：

```
static void __init setup_nr_cpu_ids(void)
{
	nr_cpu_ids = find_last_bit(cpumask_bits(cpu_possible_mask),NR_CPUS) + 1;
}
```

nr_cpu_ids 是一个特殊的值，在单 CPU 情况下是 1；而 SMP 情况下，又是一个全局变量，被 find_last_bit 函数设置，针对 x86 体系其本质上会调用 bsr 汇编指令。这里我大概介绍一下这个指令的概念，可能有不对的地方，请高手指教。386 以上的 CPU 有一对指令 BSF/BSR——正/反向位扫描。这个指令的使用方法是：BSF dest,src，影响标志位 ZF。这个指令的意思是，扫描源操作数中的第一个被设置的位，如果发现某一位被设置了，则设置 ZF 位并将第一个被设置位的索引装载到目的操作数中；如果没有发现被设置的位，则清除 ZF。BSF 正向扫描各个位(从第 0 位到第 N 位)，BSR 相反(从第 N 位到第 0 位)。

继续走，566 行，setup_per_cpu_areas，来自 arch/x86/kernel/setup_percpu.c。这个函数只是设置一下 SMP 的每 CPU 存储区，也就是说为系统中的每个 cpu 的 per_cpu 变量申请空间。函数比较复杂，我这里只把整个函数列出来，对 SMP 感兴趣的同学可以尝试深入分析一下：

```
void __init setup_per_cpu_areas(void)
```

```
{
	unsigned int cpu;
	unsigned long delta;
	int rc;

	pr_info("NR_CPUS:%d nr_cpumask_bits:%d nr_cpu_ids:%d nr_node_ids:%d\n",
		NR_CPUS, nr_cpumask_bits, nr_cpu_ids, nr_node_ids);

	/*
	 * Allocate percpu area.  Embedding allocator is our favorite;
	 * however, on NUMA configurations, it can result in very
	 * sparse unit mapping and vmalloc area isn't spacious enough
	 * on 32bit.  Use page in that case.
	 */
#ifdef CONFIG_X86_32
	if (pcpu_chosen_fc == PCPU_FC_AUTO && pcpu_need_numa())
		pcpu_chosen_fc = PCPU_FC_PAGE;
#endif
	rc = -EINVAL;
	if (pcpu_chosen_fc != PCPU_FC_PAGE) {
		const size_t atom_size = cpu_has_pse ? PMD_SIZE : PAGE_SIZE;
		const size_t dyn_size = PERCPU_MODULE_RESERVE +
			PERCPU_DYNAMIC_RESERVE - PERCPU_FIRST_CHUNK_RESERVE;

		rc = pcpu_embed_first_chunk(PERCPU_FIRST_CHUNK_RESERVE,
					    dyn_size, atom_size,
					    pcpu_cpu_distance,
					    pcpu_fc_alloc, pcpu_fc_free);
		if (rc < 0)
			pr_warning("%s allocator failed (%d), falling back to page size\n",
				   pcpu_fc_names[pcpu_chosen_fc], rc);
	}
	if (rc < 0)
		rc = pcpu_page_first_chunk(PERCPU_FIRST_CHUNK_RESERVE,
					   pcpu_fc_alloc, pcpu_fc_free,
					   pcpup_populate_pte);
	if (rc < 0)
		panic("cannot initialize percpu area (err=%d)", rc);

	/* alrighty, percpu areas up and running */
	delta = (unsigned long)pcpu_base_addr - (unsigned long)__per_cpu_start;
	for_each_possible_cpu(cpu) {
		per_cpu_offset(cpu) = delta + pcpu_unit_offsets[cpu];
		per_cpu(this_cpu_off, cpu) = per_cpu_offset(cpu);
```

```
            per_cpu(cpu_number, cpu) = cpu;
            setup_percpu_segment(cpu);
            setup_stack_canary_segment(cpu);
            /*
             * Copy data used in early init routines from the
             * initial arrays to the per cpu data areas.  These
             * arrays then become expendable and the *_early_ptr's
             * are zeroed indicating that the static arrays are
             * gone.
             */
#ifdef CONFIG_X86_LOCAL_APIC
            per_cpu(x86_cpu_to_apicid, cpu) =
                early_per_cpu_map(x86_cpu_to_apicid, cpu);
            per_cpu(x86_bios_cpu_apicid, cpu) =
                early_per_cpu_map(x86_bios_cpu_apicid, cpu);
#endif
#ifdef CONFIG_X86_64
            per_cpu(irq_stack_ptr, cpu) =
                per_cpu(irq_stack_union.irq_stack, cpu) +
                IRQ_STACK_SIZE - 64;
#ifdef CONFIG_NUMA
            per_cpu(x86_cpu_to_node_map, cpu) =
                early_per_cpu_map(x86_cpu_to_node_map, cpu);
#endif
#endif
            /*
             * Up to this point, the boot CPU has been using .data.init
             * area.  Reload any changed state for the boot CPU.
             */
            if (cpu == boot_cpu_id)
                switch_to_new_gdt(cpu);
        }

        /* indicate the early static arrays will soon be gone */
#ifdef CONFIG_X86_LOCAL_APIC
        early_per_cpu_ptr(x86_cpu_to_apicid) = NULL;
        early_per_cpu_ptr(x86_bios_cpu_apicid) = NULL;
#endif
#if defined(CONFIG_X86_64) && defined(CONFIG_NUMA)
        early_per_cpu_ptr(x86_cpu_to_node_map) = NULL;
#endif

#if defined(CONFIG_X86_64) && defined(CONFIG_NUMA)
        /*
```

```
         * make sure boot cpu node_number is right, when boot cpu is on the
         * node that doesn't have mem installed
         */
        per_cpu(node_number, boot_cpu_id) = cpu_to_node(boot_cpu_id);
#endif

        /* Setup node to cpumask map */
        setup_node_to_cpumask_map();

        /* Setup cpu initialized, callin, callout masks */
        setup_cpu_local_masks();
}
```

在该函数中，为每个 CPU 分配一段专有数据区，并将.data.percpu 中的数据拷贝到其中， 每个 CPU 各有一份。由于数据从__per_cpu_start 处转移到各 CPU 自己的专有数据区中了， 因此存取其中的变量就不能再用原先的值了，比如存取 per_cpu__runqueues 就不能再用 per_cpu__runqueues 了，需要做一个偏移量的调整，即需要加上各 CPU 自己的专有数据区首地址相对于__per_cpu_start 的偏移量。在这里也就是__per_cpu_offset[i]，其中 CPU i 的专有数据区相对于__per_cpu_start 的偏移量为__per_cpu_offset[i]。

这样，就可以方便地计算专有数据区中各变量的新地址，比如对于 per_cpu_runqueues， 其新地址即变成 per_cpu_runqueues+__per_cpu_offset[i]。

经过这样的处理，.data.percpu 这个 section 在系统初始化后就可以释放了。为什么要释放它？OK，自己去看 arch/x86/kernel/vmlinux.lds 文件，整个.data.percpu 这个 section 都在__init_begin 和__init_end 之间，也就是说，该 section 所占内存会在系统启动后释放(free)掉。

继续走，start_kernel 的 567 行 smp_prepare_boot_cpu 函数，来自 arch/x86/include/asm/smp.h：

```
static inline void smp_prepare_boot_cpu(void)
{
        smp_ops.smp_prepare_boot_cpu();
}
```

全局变量 smp_ops 也是一个 smp_ops 结构，在代码 arch/x86/kernel/smp.c 中被初始化成：

```
struct smp_ops smp_ops = {
        .smp_prepare_boot_cpu = native_smp_prepare_boot_cpu,
        .smp_prepare_cpus = native_smp_prepare_cpus,
        .smp_cpus_done          = native_smp_cpus_done,

        .smp_send_stop          = native_smp_send_stop,
        .smp_send_reschedule    = native_smp_send_reschedule,

        .cpu_up                 = native_cpu_up,
```

```
    .cpu_die        = native_cpu_die,
    .cpu_disable        = native_cpu_disable,
    .play_dead          = native_play_dead,

    .send_call_func_ipi = native_send_call_func_ipi,
    .send_call_func_single_ipi = native_send_call_func_single_ipi,
};
```

所以，567 行 smp_prepare_boot_cpu 函数最终调用 native_smp_prepare_boot_cpu 函数。该函数最终会调用 switch_to_new_gdt 函数，传给他的参数是当前 CPU 的编号：

```
void switch_to_new_gdt(int cpu)
{
    struct desc_ptr gdt_descr;

    gdt_descr.address = (long)get_cpu_gdt_table(cpu);
    gdt_descr.size = GDT_SIZE - 1;
    load_gdt(&gdt_descr);
    /* Reload the per-cpu base */

    load_percpu_segment(cpu);
}
```

load_gdt 很熟悉了，就是调用 lgdt 汇编指令加载 GDT 表。那么，自系统启动至此，已经有三次加载 GDT 了，为啥这里又来加载一次呢？这里面的关键是 get_cpu_gdt_table 函数，来自 arch/x86/include/asm/desc.h：

```
static inline struct desc_struct *get_cpu_gdt_table(unsigned int cpu)
{
    return per_cpu(gdt_page, cpu).gdt;
}
```

好了，SMP 的重中之重来了，per_cpu 宏，这里重点介绍它：

```
#define per_cpu(var, cpu) \
    (*SHIFT_PERCPU_PTR(&(var), per_cpu_offset(cpu)))
```

我们看到传入这个宏的两个参数是 gdt_page 和 cpu。gdt_page 还记得吧，我们在“初始化 GDT” http://blog.csdn.net/yunsongice/archive/2010/12/31/6110703.aspx 中讲过，包含 32 个 8 字节的段描述符；cpu，刚刚传进来的当前 CPU 的 id。翻译过来就是：

```
*SHIFT_PERCPU_PTR(&(gdt_page), per_cpu_offset(cpu))

#define SHIFT_PERCPU_PTR(__p, __offset)     ({                          \
    __verify_pcpu_ptr((__p));                               \
    RELOC_HIDE((typeof(*(__p)) __kernel __force *)(__p), (__offset)); \
})
```

__verify_pcpu_ptr 是 GCC 的一个优化过程，我们忽略，所以继续翻译就是：
RELOC_HIDE((typeof(*(&(gdt_page))) __kernel __force *)(&(gdt_page)), (per_cpu_offset(cpu))

```
#define RELOC_HIDE(ptr, off)                    \
  ({ unsigned long __ptr;                       \
     __asm__ ("" : "=r"(__ptr) : "0"(ptr));     \
     (typeof(ptr)) (__ptr + (off)); })
```

对于 per_cpu(gdt_page, cpu)，将等效地扩展为：
__per_cpu_offset[smp_processor_id()] + per_cpu__gdt_page
并且是一个 lvalue，也就是说可以进行赋值操作。这正好是上述 per_cpu__runqueues 变量在对应 CPU 的专有数据区中的新地址。

由于不同的每 cpu 变量有不同的偏移量，并且不同的 CPU 其专有数据区首地址不同，因此，通过 per_cpu (var,cpu)便访问到了不同的变量。对这个概念还不是很清楚的同学请查阅一下博客“每 CPU 变量”http://blog.csdn.net/yunsongice/archive/2010/05/18/5605239.aspx。

5.4 初始化内存管理区列表

回到 start_kernel 函数，569 行的 build_all_zonelists()函数，来自 mm/page_alloc.c：

```
2815void build_all_zonelists(void)
2816{
2817            set_zonelist_order();
2818
2819            if (system_state == SYSTEM_BOOTING) {
2820                        __build_all_zonelists(NULL);
2821                        mminit_verify_zonelist();
2822                        cpuset_init_current_mems_allowed();
2823            } else {
2824                        /* we have to stop all cpus to guarantee there is no user
2825                             of zonelist */
2826                        stop_machine(__build_all_zonelists, NULL, NULL);
2827                        /* cpuset refresh routine should be here */
2828            }
2829            vm_total_pages = nr_free_pagecache_pages();
2830            /* ……一大堆注释*/
2837            if (vm_total_pages < (pageblock_nr_pages * MIGRATE_TYPES))
2838                        page_group_by_mobility_disabled = 1;
2839            else
2840                        page_group_by_mobility_disabled = 0;
2841
```

```
2842            printk("Built %i zonelists in %s order, mobility grouping %s.    "
2843                    "Total pages: %ld\n",
2844                            nr_online_nodes,
2845                            zonelist_order_name[current_zonelist_order],
2846                            page_group_by_mobility_disabled ? "off" : "on",
2847                            vm_total_pages);
2848#ifdef CONFIG_NUMA
2849            printk("Policy zone: %s\n", zone_names[policy_zone]);
2850#endif
2851}
```

其本质上调用__build_all_zonelists(NULL)：

```
2780/* return values int ....just for stop_machine() */
2781static int __build_all_zonelists(void *dummy)
2782{
2783            int nid;
2784            int cpu;
2785
2786#ifdef CONFIG_NUMA
2787            memset(node_load, 0, sizeof(node_load));
2788#endif
2789            for_each_online_node(nid) {
2790                    pg_data_t *pgdat = NODE_DATA(nid);
2791
2792                    build_zonelists(pgdat);
2793                    build_zonelist_cache(pgdat);
2794            }
2795
2796            /* ……一大堆注释*/
2809            for_each_possible_cpu(cpu)
2810                    setup_pageset(&per_cpu(boot_pageset, cpu), 0);
2811
2812            return 0;
2813}
```

2789 行，for_each_online_node 我们很熟悉了，只执行一次的循环。2790 行是最著名的 pg_data_t，就是 NODE_DATA(0)的那个结构。随后执行 build_zonelists 函数：

```
2637static void build_zonelists(pg_data_t *pgdat)
2638{
2639            int j, node, load;
2640            enum zone_type i;
2641            nodemask_t used_mask;
```

```
2642        int local_node, prev_node;
2643        struct zonelist *zonelist;
2644        int order = current_zonelist_order;
2645
2646        /* initialize zonelists */
2647        for (i = 0; i < MAX_ZONELISTS; i++) {
2648                zonelist = pgdat->node_zonelists + i;
2649                zonelist->_zonerefs[0].zone = NULL;
2650                zonelist->_zonerefs[0].zone_idx = 0;
2651        }
2652
2653        /* NUMA-aware ordering of nodes */
2654        local_node = pgdat->node_id;
2655        load = nr_online_nodes;
2656        prev_node = local_node;
2657        nodes_clear(used_mask);
2658
2659        memset(node_order, 0, sizeof(node_order));
2660        j = 0;
2661
2662        while ((node = find_next_best_node(local_node, &used_mask)) >= 0) {
2663                int distance = node_distance(local_node, node);
2664
2665                /*
2666                  * If another node is sufficiently far away then it is better
2667                  * to reclaim pages in a zone before going off node.
2668                  */
2669                if (distance > RECLAIM_DISTANCE)
2670                        zone_reclaim_mode = 1;
2671
2672                /*
2673                  * We don't want to pressure a particular node.
2674                  * So adding penalty to the first node in same
2675                  * distance group to make it round-robin.
2676                  */
2677                if (distance != node_distance(local_node, prev_node))
2678                        node_load[node] = load;
2679
2680                prev_node = node;
2681                load--;
2682                if (order == ZONELIST_ORDER_NODE)
2683                        build_zonelists_in_node_order(pgdat, node);
2684                else
2685                        node_order[j++] = node; /* remember order */
```

```
2686            }
2687
2688            if (order == ZONELIST_ORDER_ZONE) {
2689                    /* calculate node order -- i.e., DMA last! */
2690                    build_zonelists_in_zone_order(pgdat, j);
2691            }
2692
2693            build_thisnode_zonelists(pgdat);
2694}
```

build_zonelists 函数 2647-2651 初始化 NODE_DATA(0)的 node_zonelist 字段。我们继续走：

```
2697static void build_zonelist_cache(pg_data_t *pgdat)
2698{
2699            struct zonelist *zonelist;
2700            struct zonelist_cache *zlc;
2701            struct zoneref *z;
2702
2703            zonelist = &pgdat->node_zonelists[0];
2704            zonelist->zlcache_ptr = zlc = &zonelist->zlcache;
2705            bitmap_zero(zlc->fullzones, MAX_ZONES_PER_ZONELIST);
2706            for (z = zonelist->_zonerefs; z->zone; z++)
2707                    zlc->z_to_n[z - zonelist->_zonerefs] = zonelist_node_idx(z);
2708}
```

build_zonelist_cache 函数初始化内存管理区的缓存，我这里就不深入下去了。回到 build_all_zonelists()函数中，略去调试的代码，以及设置几个关于 zone 的策略的全局变量的代码，该函数就结束了。

5.5 利用 early_res 分配内存

回到 start_kernel 中，570 行，page_alloc_init()函数：

```
void __init page_alloc_init(void)
{
    hotcpu_notifier(page_alloc_cpu_notify, 0);
}
```

这个函数会调用 hotcpu_notifier 函数。当然，在编译选项 CONFIG_HOTPLUG_CPU 起作用时，这个函数才有效。这个编译选项就是性感的热插拔技术，而且是 CPU 的热插拔。如果定义了，就去新建一个 notifier_block 结构，并调用 register_cpu_notifier 函数，将新建的这个结构挂到全局 cpu_chain 链中，具体的代码我就不去分析了。

继续走，572 行是一个 printk 函数，用于打印刚刚 setup_arch 拷贝到 boot_command_line 的

信息。随后 573 行，调用 parse_early_param 函数对 boot_command_line 进行解析。它跟 574 行一样，都是调用 parse_args 进行解析。这个函数来自 kernel/params.c，函数本身比较简单，就是把解析后的结果放到 start_kernel 的内部变量__start___param 数组中。

继续走，581 行，pidhash_init 函数，又遇到了一个重点函数，来自 kernel/pid.c：

```
501void __init pidhash_init(void)
502{
503         int i, pidhash_size;
504
505         pid_hash = alloc_large_system_hash("PID", sizeof(*pid_hash), 0, 18,
506                                            HASH_EARLY | HASH_SMALL,
507                                            &pidhash_shift, NULL, 4096);
508         pidhash_size = 1 << pidhash_shift;
509
510         for (i = 0; i < pidhash_size; i++)
511                 INIT_HLIST_HEAD(&pid_hash[i]);
512}
```

pid_hash 是个全局变量：
static struct hlist_head *pid_hash;

为什么说它是重点函数呢，因为这里涉及到了初始化期间，在 slab 分配器还不存在的时候，只有前面刚刚建立好的初始化阶段的内存管理体系，如何分配一个内存空间来存放 pid 散列表。而 pid 散列表这么一个东西，它跟物理内存大小有关，也就是 1GB 的内存空间就可能有 16~4096 个散列项。不管怎样，调用 alloc_large_system_hash 函数：

```
4863void *__init alloc_large_system_hash(const char *tablename,
4864                                      unsigned long bucketsize,
4865                                      unsigned long numentries,
4866                                      int scale,
4867                                      int flags,
4868                                      unsigned int *_hash_shift,
4869                                      unsigned int *_hash_mask,
4870                                      unsigned long limit)
4871{
4872         unsigned long long max = limit;
4873         unsigned long log2qty, size;
4874         void *table = NULL;
4875
4876         /* allow the kernel cmdline to have a say */
4877         if (!numentries) {
4878                 /* round applicable memory size up to nearest megabyte */
4879                 numentries = nr_kernel_pages;
```

```
4880                    numentries += (1UL << (20 - PAGE_SHIFT)) - 1;
4881                    numentries >>= 20 - PAGE_SHIFT;
4882                    numentries <<= 20 - PAGE_SHIFT;
4883
4884                    /* limit to 1 bucket per 2^scale bytes of low memory */
4885                    if (scale > PAGE_SHIFT)
4886                            numentries >>= (scale - PAGE_SHIFT);
4887                    else
4888                            numentries <<= (PAGE_SHIFT - scale);
4889
4890                    /* Make sure we've got at least a 0-order allocation.. */
4891                    if (unlikely(flags & HASH_SMALL)) {
4892                            /* Makes no sense without HASH_EARLY */
4893                            WARN_ON(!(flags & HASH_EARLY));
4894                            if (!(numentries >> *_hash_shift)) {
4895                                    numentries = 1UL << *_hash_shift;
4896                                    BUG_ON(!numentries);
4897                            }
4898                    } else if (unlikely((numentries * bucketsize) < PAGE_SIZE))
4899                            numentries = PAGE_SIZE / bucketsize;
4900            }
4901            numentries = roundup_pow_of_two(numentries);
4902
4903            /* limit allocation size to 1/16 total memory by default */
4904            if (max == 0) {
4905                    max = ((unsigned long long)nr_all_pages << PAGE_SHIFT) >> 4;
4906                    do_div(max, bucketsize);
4907            }
4908
4909            if (numentries > max)
4910                    numentries = max;
4911
4912            log2qty = ilog2(numentries);
4913
4914            do {
4915                    size = bucketsize << log2qty;
4916                    if (flags & HASH_EARLY)
4917                            table = alloc_bootmem_nopanic(size);
4918                    else if (hashdist)
4919                            table = __vmalloc(size, GFP_ATOMIC, PAGE_KERNEL);
4920                    else {
4921                            /*
4922                              * If bucketsize is not a power-of-two, we may free
4923                              * some pages at the end of hash table which
```

```
4924                                     * alloc_pages_exact() automatically does
4925                                     */
4926                                    if (get_order(size) < MAX_ORDER) {
4927                                            table = alloc_pages_exact(size, GFP_ATOMIC);
4928                                            kmemleak_alloc(table, size, 1, GFP_ATOMIC);
4929                                    }
4930                            }
4931            } while (!table && size > PAGE_SIZE && --log2qty);
4932
4933            if (!table)
4934                    panic("Failed to allocate %s hash table\n", tablename);
4935
4936            printk(KERN_INFO "%s hash table entries: %d (order: %d, %lu bytes)\n",
4937                    tablename,
4938                    (1U << log2qty),
4939                    ilog2(size) - PAGE_SHIFT,
4940                    size);
4941
4942            if (_hash_shift)
4943                    *_hash_shift = log2qty;
4944            if (_hash_mask)
4945                    *_hash_mask = (1 << log2qty) - 1;
4946
4947            return table;
4948}
```

这个函数也是很简单的，我们就不一行一行地分析了。注意，传进来的参数 numentries 为 0，所以要进入 4877 行的条件语句，把它赋值成 nr_kernel_pages。而这个值被定义到.meminit.data 段中的，所以，numentries 使用的初始化期间的数据段。4914 行之前的代码都是在计算这个 numentries，也就是我们散列项数的值。前面说了，跟内存大小相关，当计算出来后，由于我们传入的 flag 是 HASH_EARLY，所以调用 alloc_bootmem_nopanic 宏。还要注意，这时候 slab 分配器还没初始化，也就是根本不存在，所以，不能通过 vmalloc 或 kmalloc 来分配内存空间。

好了，numentries 得到了，那么这个散列表需要多大呢？4912 行做一个 log 计算，得到这个 table 的 size，传入 alloc_bootmem_nopanic 宏中。

```
#define alloc_bootmem_nopanic(x) \
        __alloc_bootmem_nopanic(x, SMP_CACHE_BYTES, __pa(MAX_DMA_ADDRESS))

void * __init __alloc_bootmem_nopanic(unsigned long size, unsigned long align,
                                      unsigned long goal)
{
        unsigned long limit = 0;
```

```
#ifdef CONFIG_NO_BOOTMEM
	limit = -1UL;
#endif

	return ___alloc_bootmem_nopanic(size, align, goal, limit);
}


static void * __init ___alloc_bootmem_nopanic(unsigned long size,
					unsigned long align,
					unsigned long goal,
					unsigned long limit)
{
#ifdef CONFIG_NO_BOOTMEM
	void *ptr;

	if (WARN_ON_ONCE(slab_is_available()))
		return kzalloc(size, GFP_NOWAIT);

restart:

	ptr = __alloc_memory_core_early(MAX_NUMNODES, size, align, goal, limit);

	if (ptr)
		return ptr;

	if (goal != 0) {
		goal = 0;
		goto restart;
	}

	return NULL;
#else
	bootmem_data_t *bdata;
	void *region;

restart:
	region = alloc_arch_preferred_bootmem(NULL, size, align, goal, limit);
	if (region)
		return region;

	list_for_each_entry(bdata, &bdata_list, list) {
		if (goal && bdata->node_low_pfn <= PFN_DOWN(goal))
```

```
                continue;
        if (limit && bdata->node_min_pfn >= PFN_DOWN(limit))
                break;

        region = alloc_bootmem_core(bdata, size, align, goal, limit);
        if (region)
                return region;
    }

    if (goal) {
        goal = 0;
        goto restart;
    }

    return NULL;
#endif
}
```

我们看到，上面的函数层层包装，由于配置了 CONFIG_NO_BOOTMEM，所以最后调用 __alloc_memory_core_early 函数（注意，在调用它之前仍然要先检查一下 slab 环境是否已经建立起来了，如果是，就用 slab 来分配）。

还记得我们在 setup_arch()时，调用 initmem_init 来初始化的 early_node_map 数组吗？这里就用到了，__alloc_memory_core_early 函数也是咱们的老熟人了，大家可以回头看看“添砖加瓦”相关的内容。那么这里就有一个非常重要的问题了。我们前面为页框分配空间的时候，当时调用 __alloc_memory_core_early 函数时，传给他的参数 goal 的值是宏 MAX_DMA_ADDRESS 的物理地址，也就是 0x1000000，而__pa(MAX_DMA_ADDRESS)也是这次的 goal 参数。将来我们也会看到，初始化阶段所有的 goal 都是它。那么我们在调用最底层 find_early_area 函数时，岂不是全部都只从一个固定物理地址开始分配？这不乱套了？

要回答这个问题，我们还是得回顾一下 find_early_area 函数，来自 kernel/early_res.c 文件：

```
539u64 __init find_early_area(u64 ei_start, u64 ei_last, u64 start, u64 end,
540                           u64 size, u64 align)
541{
542        u64 addr, last;
543
544        addr = round_up(ei_start, align);
545        if (addr < start)
546                addr = round_up(start, align);
547        if (addr >= ei_last)
548                goto out;
549        while (bad_addr(&addr, size, align) && addr+size <= ei_last)
```

```
550                     ;
551             last = addr + size;
552             if (last > ei_last)
553                     goto out;
554             if (last > end)
555                     goto out;
556
557             return addr;
558
559out:
560             return -1ULL;
561}
```

这个函数我们前面仅仅是做了简单说明，其实看起简单的东西未必就能一下子吃透，需要花费大量的功夫。前面在讲 find_early_area 获得永久内核页表的首地址的时候，我们就漏掉了一个处理重叠问题的一个重要的步骤。既然所有的 goal 都是同一个值，那么肯定会有重叠，find_early_area 函数的 549 行那个 while 循环就是处理这个重叠。主要是调用 bad_addr 函数，位于同一个文件中：

```
483static inline int __init bad_addr(u64 *addrp, u64 size, u64 align)
484{
485             int i;
486             u64 addr = *addrp;
487             int changed = 0;
488             struct early_res *r;
489again:
490             i = find_overlapped_early(addr, addr + size);
491             r = &early_res[i];
492             if (i < max_early_res && r->end) {
493                     *addrp = addr = round_up(r->end, align);
494                     changed = 1;
495                     goto again;
496             }
497             return changed;
498}
```

注意，我们传递给 bad_addr 的参数是 addr 对应的地址，意思就是让该函数去探测 addr 对应的 size 个内存单元是否已经被使用，即发生重叠冲突，如果是，就修改 addr 本身，最终目的是确保 addr 对应的 size 个内存单元没有被任何其他数据结构所占据。

那么，bad_addr 是如何处理这个冲突的呢？这里要介绍一个 early_res 体系，看到 kernel/early_res.c 文件的前几行代码：

```
18#define MAX_EARLY_RES_X 32
```

```
19
20struct early_res {
21        u64 start, end;
22        char name[15];
23        char overlap_ok;
24};
25static struct early_res early_res_x[MAX_EARLY_RES_X] __initdata;
26
27static int max_early_res __initdata = MAX_EARLY_RES_X;
28static struct early_res *early_res __initdata = &early_res_x[0];
29static int early_res_count __initdata;
```

在我们编译 vmlinux 时，内核初始化代码的数据区有一个 early_res_x 数组，存放着 32 个 early_res 类型的变量，而且还有一个全局指针 early_res 指向这个数组的第一个元素，还有一个全局变量 max_early_res 为 32。

那么，bad_addr 函数的 490 行就调用 find_overlapped_early 函数对这个 addr 进行调整：

```
31static int __init find_overlapped_early(u64 start, u64 end)
32{
33        int i;
34        struct early_res *r;
35
36        for (i = 0; i < max_early_res && early_res[i].end; i++) {
37                r = &early_res[i];
38                if (end > r->start && start < r->end)
39                        break;
40        }
41
42        return i;
43}
```

我们看到，如果地址区间[addr, addr + size]落到 early_res[]数组的某个元素的[start, end]范围，就表示发生了冲突，则会返回小于 max_early_res 的 i，那么 bad_addr 的 493 行就会对这个 addr 进行处理：*addrp = addr = round_up(r->end, align)；强制它等于那个 early_res[i]元素的 end 值。然后返回等于 1 的 changed，让 find_early_area 中的那个 while 循环再做同样的检查，直到 find_overlapped_early 返回等于 max_early_res 的 i，从而跳出 while 循环，得到不与任何内核数据结构冲突的 addr。

当__alloc_memory_core_early 函数执行完 find_early_area，获得一个正确的 addr 时，会把他变成虚拟地址，并 memset 它，随后调用函数：
reserve_early_without_check(addr, addr + size, "BOOTMEM");

```
void __init reserve_early_without_check(u64 start, u64 end, char *name)
```

```
{
	struct early_res *r;

	if (start >= end)
		return;

	__check_and_double_early_res(start, end);

	r = &early_res[early_res_count];

	r->start = start;
	r->end = end;
	r->overlap_ok = 0;
	if (name)
		strncpy(r->name, name, sizeof(r->name) - 1);
	early_res_count++;
}
```

这个函数没有问题，同样来自 kernel/early_res.c 文件。就是把刚刚分配的 addr 和 addr + size 作为 early_res[]保存起来，避免在伙伴系统和 slab 体系初始化之前，其他的通过 _alloc_memory_core_early 函数分配空间的函数与其发生冲突。

至此，针对初始化期间的内存管理就全介绍完了，不过我还有一点要说。初始化期间为各个数据结构分配物理页面都是通过_alloc_memory_core_early 函数。内核发展到现在，已经抛弃了以前的 BOOTMEM 分配体系，而只是通过_alloc_memory_core_early 函数调用 find_early_area 函数进行简单的分配，大幅提高了系统初始化期间的性能。如果大家还对过去的那套初始化分配体系感兴趣，可以尝试一下取消编译配置 CONFIG_NO_BOOTMEM。

另外，这个 early_res[]数组最大不能超过该 32 个，所以即使效率很高，但初始化阶段借助 early_res 体系分配空间的次数也不能超过 32 次。前面已经用了一次了，也就是给 pgdat->node_mem_map 分配页面的时候，这里是第二次。后面还有，有兴趣的同学可以数一数，是不是没有超过 32 次。

可能细心的同学会问，我们在“着手建立永久内核页表”不是也用了 find_early_area 来为页表分配空间么。但是你回过头去看看，那个是个传递给 find_early_area 的 start 参数是 0x7000（find_early_table_space 函数的 74 行），跟我们的__pa(MAX_DMA_ADDRESS)相差甚远，所以打死都不会冲突的。

还有一点就是，通过_alloc_memory_core_early 分配的页面是会释放的。所以大家要有一个概念了，只要当伙伴系统建立起来后，我们通过 alloc_page 函数分配的页面，其对应的释放函数是 free_page。而_alloc_memory_core_early 函数也有对应的 free 函数的，那就是 free_all_memory_core_early 函数，用于释放我们初始化期间占有的临时内存，后面一定会碰到。

5.6 触碰虚拟文件系统

回到 start_kernel 中，下面我们该第一次接触文件系统了，582 行执行 vfs_caches_init_early：

```
void __init vfs_caches_init_early(void)
{
	dcache_init_early();
	inode_init_early();
}
```

vfs_caches_init_early 调用两个函数 dcache_init_early 和 inode_init_early。内核第一次触碰文件系统的主要目的就是初始化 VFS 的两个重要数据结构 dcache 和 inode 的缓存。
dcache_init_early 来自 fs/dcache.c：

```
2287static void __init dcache_init_early(void)
2288{
2289		int loop;
2290
2291		/* If hashes are distributed across NUMA nodes, defer
2292		 * hash allocation until vmalloc space is available.
2293		 */
2294		if (hashdist)
2295			return;
2296
2297		dentry_hashtable =
2298			alloc_large_system_hash("Dentry cache",
2299						sizeof(struct hlist_head),
2300						dhash_entries,
2301						13,
2302						HASH_EARLY,
2303						&d_hash_shift,
2304						&d_hash_mask,
2305						0);
2306
2307		for (loop = 0; loop < (1 << d_hash_shift); loop++)
2308			INIT_HLIST_HEAD(&dentry_hashtable[loop]);
2309}
```

inode_init_early 来自 fs/inode.c：

```
1539void __init inode_init_early(void)
1540{
1541		int loop;
```

```
1542
1543            /* If hashes are distributed across NUMA nodes, defer
1544             * hash allocation until vmalloc space is available.
1545             */
1546            if (hashdist)
1547                    return;
1548
1549            inode_hashtable =
1550                    alloc_large_system_hash("Inode-cache",
1551                                            sizeof(struct hlist_head),
1552                                            ihash_entries,
1553                                            14,
1554                                            HASH_EARLY,
1555                                            &i_hash_shift,
1556                                            &i_hash_mask,
1557                                            0);
1558
1559            for (loop = 0; loop < (1 << i_hash_shift); loop++)
1560                    INIT_HLIST_HEAD(&inode_hashtable[loop]);
1561}
```

两个函数都是调用 alloc_large_system_hash 函数分别为 dentry_hashtable 和 inode_hashtable 分配空间。至于这两个散列的用途，请参考博客“把 Linux 中的 VFS 对象串联起来”http://blog.csdn.net/yunsongice/archive/2010/06/21/5683859.aspx

5.7 初始化异常服务

继续走，start_kernel 的 583 行，sort_main_extable，把编译期间，kbuild 设置的异常表，也就是__start___ex_table 和__stop___ex_table 之中的所有元素进行排序。

584 行，调用 trap_init 函数，重要的函数，初始化中断向量表。该函数来自 arch/x86/kernel/traps.c

```
 882void __init trap_init(void)
 883{
 884            int i;
 885
 886#ifdef CONFIG_EISA
 887            void __iomem *p = early_ioremap(0x0FFFD9, 4);
 888
 889            if (readl(p) == 'E' + ('I'<<8) + ('S'<<16) + ('A'<<24))
 890                    EISA_bus = 1;
```

```
891             early_iounmap(p, 4);
892#endif
893
894             set_intr_gate(0, &divide_error);
895             set_intr_gate_ist(1, &debug, DEBUG_STACK);
896             set_intr_gate_ist(2, &nmi, NMI_STACK);
897             /* int3 can be called from all */
898             set_system_intr_gate_ist(3, &int3, DEBUG_STACK);
899             /* int4 can be called from all */
900             set_system_intr_gate(4, &overflow);
901             set_intr_gate(5, &bounds);
902             set_intr_gate(6, &invalid_op);
903             set_intr_gate(7, &device_not_available);
904#ifdef CONFIG_X86_32
905             set_task_gate(8, GDT_ENTRY_DOUBLEFAULT_TSS);
906#else
907             set_intr_gate_ist(8, &double_fault, DOUBLEFAULT_STACK);
908#endif
909             set_intr_gate(9, &coprocessor_segment_overrun);
910             set_intr_gate(10, &invalid_TSS);
911             set_intr_gate(11, &segment_not_present);
912             set_intr_gate_ist(12, &stack_segment, STACKFAULT_STACK);
913             set_intr_gate(13, &general_protection);
914             set_intr_gate(14, &page_fault);
915             set_intr_gate(15, &spurious_interrupt_bug);
916             set_intr_gate(16, &coprocessor_error);
917             set_intr_gate(17, &alignment_check);
918#ifdef CONFIG_X86_MCE
919             set_intr_gate_ist(18, &machine_check, MCE_STACK);
920#endif
921             set_intr_gate(19, &simd_coprocessor_error);
922
923             /* Reserve all the builtin and the syscall vector: */
924             for (i = 0; i < FIRST_EXTERNAL_VECTOR; i++)
925                     set_bit(i, used_vectors);
926
927#ifdef CONFIG_IA32_EMULATION
928             set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
929             set_bit(IA32_SYSCALL_VECTOR, used_vectors);
930#endif
931
932#ifdef CONFIG_X86_32
933             if (cpu_has_fxsr) {
934                     printk(KERN_INFO "Enabling fast FPU save and restore... ");
```

```
935                     set_in_cr4(X86_CR4_OSFXSR);
936                     printk("done.\n");
937             }
938             if (cpu_has_xmm) {
939                     printk(KERN_INFO
940                             "Enabling unmasked SIMD FPU exception support... ");
941                     set_in_cr4(X86_CR4_OSXMMEXCPT);
942                     printk("done.\n");
943             }
944
945             set_system_trap_gate(SYSCALL_VECTOR, &system_call);
946             set_bit(SYSCALL_VECTOR, used_vectors);
947#endif
948
949             /*
950              * Should be a barrier for any external CPU state:
951              */
952             cpu_init();
953
954             x86_init.irqs.trap_init();
955}
```

我们没有配置 CONFIG_EISA 编译选项，所以 886~892 行代码忽略。894~921 行，对中断向量表 0 - 19 号异常设置对应的服务程序。我们看到主要有四种类型的设置函数：

set_intr_gate

set_system_intr_gate

set_system_trap_gate

set_task_gate

其实这里涉及到一个 x86 体系的“门”概念，这里简单介绍一下：x86 保护模式下物理地址段具有 4 种特级——0 到 4 级——，而 Linux 只使用了表示内核态的 0 级和用户态的 3 级。当异常和中断发生时，必然会引起地址的转换，比如从 3 级段中，转移到 0 级段中去执行相应的代码。如何转移？Intel 提供了一个审查机制，来保护我们的状态切换，这就是“门”的概念。

X86 体系一个提供了四种门，即任务门（Task Gate）、中断门（Interrupt Gate）、陷阱门（Trap Gate）和调用门（Call Gate）。Linux 只使用了前三种门的描述符，我们看到上面四种函数分别就对应这些门的设置。这里插一句，内核 2.4.0 以后的版本进行进程切换的时候就不使用任务门了，因为描述一个进程所需要的信息远远多于一个任务门所能表达的内容，因为后者仅仅是一个硬件上的概念。所以，我们看到 905 行，内核只有双重故障异常，也就是 8 号中断向量使用到了任务门，且仅仅是 32 位 x86 系统中。

有些函数还加入了_ist 后缀，不过不管是 set_intr_gate 还是 set_intr_gate_ist 函数，最终都调用_set_gate 函数，并把中断号、门类型、处理函数地址、DPL 的值、ist 和目录段寄存器作

为参数传给它：

```
static inline void _set_gate(int gate, unsigned type, void *addr,
                             unsigned dpl, unsigned ist, unsigned seg)
{
        gate_desc s;
        pack_gate(&s, type, (unsigned long)addr, dpl, ist, seg);
        /*
         * does not need to be atomic because it is only done once at
         * setup time
         */
        write_idt_entry(idt_table, gate, &s);
}
```

gate_desc 结构我们已经见过了，这个结构是一个中断描述符，我们在“初始化中断描述符表”中设置了 256 个这样的门描述符，idt_table 就是门描述符所组成的一个表的首地址，也叫中断描述符表（进入保护模式后，中断向量表 IDT 的每个表项不再是 4 个字节，而是 8 个字节的门描述符 gate_desc）。我们看到先是调用 pack_gate 根据所传进来的参数初始化门描述符 gate_desc：

```
static inline void pack_gate(gate_desc *gate, unsigned char type,
                             unsigned long base, unsigned dpl, unsigned flags,
                             unsigned short seg)
{
        gate->a = (seg << 16) | (base & 0xffff);
        gate->b = (base & 0xffff0000) |
                  (((0x80 | type | (dpl << 5)) & 0xff) << 8);
}
```

我们看到，32 位的 x86 系统，传递进来的 ist 参数是没有用的，所以加不加_ist 后缀意义不大，随后调用 write_idt_entry 设置中断描述符表的某个元素：

```
#define write_idt_entry(dt, entry, g)           \
        native_write_idt_entry(dt, entry, g)
static inline void native_write_idt_entry(gate_desc *idt, int entry,
                                          const gate_desc *gate)
{
        memcpy(&idt[entry], gate, sizeof(*gate));
}
```

好了，trap_init 函数设置了中断描述符表的前 19 个表项后，在 945 行还做了一个及其重要的工作，就是设置一个系统门：

```
#define IA32_SYSCALL_VECTOR             0x80
#ifdef CONFIG_X86_32
# define SYSCALL_VECTOR                 0x80
#endif
```

这就是 Linux 的系统调用对应的中断门，我们看到它的中断向量号是 0x80 号。随后 952 行调用 cpu_init()函数，来自 arch/x86/kernel/cpu/common.c：

```
1200void __cpuinit cpu_init(void)
1201{
1202            int cpu = smp_processor_id();
1203            struct task_struct *curr = current;
1204            struct tss_struct *t = &per_cpu(init_tss, cpu);
1205            struct thread_struct *thread = &curr->thread;
1206
1207            if (cpumask_test_and_set_cpu(cpu, cpu_initialized_mask)) {
1208                        printk(KERN_WARNING "CPU#%d already initialized!\n", cpu);
1209                        for (;;)
1210                                    local_irq_enable();
1211            }
1212
1213            printk(KERN_INFO "Initializing CPU#%d\n", cpu);
1214
1215            if (cpu_has_vme || cpu_has_tsc || cpu_has_de)
1216
clear_in_cr4(X86_CR4_VME|X86_CR4_PVI|X86_CR4_TSD|X86_CR4_DE);
1217
1218            load_idt(&idt_descr);
1219            switch_to_new_gdt(cpu);
1220
1221            /*
1222             * Set up and load the per-CPU TSS and LDT
1223             */
1224            atomic_inc(&init_mm.mm_count);
1225            curr->active_mm = &init_mm;
1226            BUG_ON(curr->mm);
1227            enter_lazy_tlb(&init_mm, curr);
1228
1229            load_sp0(t, thread);
1230            set_tss_desc(cpu, t);
1231            load_TR_desc();
1232            load_LDT(&init_mm.context);
1233
1234            t->x86_tss.io_bitmap_base = offsetof(struct tss_struct, io_bitmap);
1235
1236#ifdef CONFIG_DOUBLEFAULT
1237            /* Set up doublefault TSS pointer in the GDT */
1238            __set_tss_desc(cpu, GDT_ENTRY_DOUBLEFAULT_TSS, &doublefault_tss);
```

```
1239#endif
1240
1241            clear_all_debug_regs();
1242
1243            /*
1244              * Force FPU initialization:
1245              */
1246            if (cpu_has_xsave)
1247                        current_thread_info()->status = TS_XSAVE;
1248            else
1249                        current_thread_info()->status = 0;
1250            clear_used_math();
1251            mxcsr_feature_mask_init();
1252
1253            /*
1254              * Boot processor to setup the FP and extended state context info.
1255              */
1256            if (smp_processor_id() == boot_cpu_id)
1257                        init_thread_xstate();
1258
1259            xsave_init();
1260}
```

首先获得当前 cpu 的编号，然后获得 0 号进程的 task_struct 和 thread_struct 以及本 cpu 上的 tss 结构。随后 1218 行通过 load_idt 函数加载刚刚设置好的 idt_descr 数组首地址到 IDTR 寄存器；219 行加载本 CPU 的全局描述符表地址：

```
void switch_to_new_gdt(int cpu)
{
      struct desc_ptr gdt_descr;

      gdt_descr.address = (long)get_cpu_gdt_table(cpu);
      gdt_descr.size = GDT_SIZE - 1;
      load_gdt(&gdt_descr);
      /* Reload the per-cpu base */

      load_percpu_segment(cpu);
}
```

随后 1225 行加载 0 号进程的 active_mm 指针指向全局变量 init_mm；后面还做了一些其他的初始化工作。这个函数的所做的工作很多前面已经完成了，这里只是再次确保这些规定动作没有任何遗漏，充分地显示了 Linux 内核的完整性和健壮性。

trap_init 的最后一行调用 x86_init.irqs.trap_init 函数，也就是 x86_init_noop，是个空函数，所以收工了。我们看到 trap_init 函数主要是对包括大部分异常的服务程序设置。至于中断门、陷阱门及系统门的相关基础知识，请访问我的博客“ 中断描述符表 ”

http://blog.csdn.net/yunsongice/archive/2010/02/11/5306387.aspx。

5.8 初始化内存管理

回到 start_kernel，下一个函数执行 mm_init()。这个函数很重要了，来自同一个文件。

```
static void __init mm_init(void)
{
        /*
         * page_cgroup requires countinous pages as memmap
         * and it's bigger than MAX_ORDER unless SPARSEMEM.
         */
        page_cgroup_init_flatmem();
        mem_init();
        kmem_cache_init();
        pgtable_cache_init();
        vmalloc_init();
}
```

这五个函数，其中由于我们没有配置 CONFIG_CGROUP_MEM_RES_CTLR，所以第一个函数 page_cgroup_init_flatmem 是个空函数。其余几个函数各个都是重点。

该函数执行完后不能再用像 alloc_bootmem()、alloc_bootmem_low()、alloc_bootmem_pages() 等申请低端内存的函数来申请内存，也就不能申请大块的连续物理内存了。

5.8.1 启用伙伴算法

首先是 mem_init，来自 arch/x86/mm/init_32.c：

```
867void __init mem_init(void)
 868{
 869             int codesize, reservedpages, datasize, initsize;
 870             int tmp;
 871
 872             pci_iommu_alloc();
 873
 874#ifdef CONFIG_FLATMEM
 875             BUG_ON(!mem_map);
 876#endif
 877             /* this will put all low memory onto the freelists */
 878             totalram_pages += free_all_bootmem();
 879
```

```
880             reservedpages = 0;
881             for (tmp = 0; tmp < max_low_pfn; tmp++)
882                     /*
883                      * Only count reserved RAM pages:
884                      */
885                     if (page_is_ram(tmp) && PageReserved(pfn_to_page(tmp)))
886                             reservedpages++;
887
888             set_highmem_pages_init();
889
890             codesize =    (unsigned long) &_etext - (unsigned long) &_text;
891             datasize =    (unsigned long) &_edata - (unsigned long) &_etext;
892             initsize =    (unsigned long) &__init_end - (unsigned long) &__init_begin;
893
894             printk(KERN_INFO "Memory: %luk/%luk available (%dk kernel code, "
895                             "%dk reserved, %dk data, %dk init, %ldk highmem)\n",
896                     nr_free_pages() << (PAGE_SHIFT-10),
897                     num_physpages << (PAGE_SHIFT-10),
898                     codesize >> 10,
899                     reservedpages << (PAGE_SHIFT-10),
900                     datasize >> 10,
901                     initsize >> 10,
902                     totalhigh_pages << (PAGE_SHIFT-10));
903
904             printk(KERN_INFO "virtual kernel memory layout:\n"
905                     "       fixmap    : 0x%08lx - 0x%08lx      (%4ld kB)\n"
906#ifdef CONFIG_HIGHMEM
907                     "       pkmap     : 0x%08lx - 0x%08lx      (%4ld kB)\n"
908#endif
909                     "       vmalloc : 0x%08lx - 0x%08lx      (%4ld MB)\n"
910                     "       lowmem    : 0x%08lx - 0x%08lx      (%4ld MB)\n"
911                     "         .init : 0x%08lx - 0x%08lx      (%4ld kB)\n"
912                     "         .data : 0x%08lx - 0x%08lx      (%4ld kB)\n"
913                     "         .text : 0x%08lx - 0x%08lx      (%4ld kB)\n",
914                     FIXADDR_START, FIXADDR_TOP,
915                     (FIXADDR_TOP - FIXADDR_START) >> 10,
916
917#ifdef CONFIG_HIGHMEM
918                     PKMAP_BASE, PKMAP_BASE+LAST_PKMAP*PAGE_SIZE,
919                     (LAST_PKMAP*PAGE_SIZE) >> 10,
920#endif
921
922                     VMALLOC_START, VMALLOC_END,
923                     (VMALLOC_END - VMALLOC_START) >> 20,
```

```
 924
 925                     (unsigned long)__va(0), (unsigned long)high_memory,
 926                     ((unsigned long)high_memory - (unsigned long)__va(0)) >> 20,
 927
 928                     (unsigned long)&__init_begin, (unsigned long)&__init_end,
 929                     ((unsigned long)&__init_end -
 930                      (unsigned long)&__init_begin) >> 10,
 931
 932                     (unsigned long)&_etext, (unsigned long)&_edata,
 933                     ((unsigned long)&_edata - (unsigned long)&_etext) >> 10,
 934
 935                     (unsigned long)&_text, (unsigned long)&_etext,
 936                     ((unsigned long)&_etext - (unsigned long)&_text) >> 10);
 937
 938             /*
 939              * Check boundaries twice: Some fundamental inconsistencies can
 940              * be detected at build time already.
 941              */
 942#define __FIXADDR_TOP (-PAGE_SIZE)
 943#ifdef CONFIG_HIGHMEM
 944                     BUILD_BUG_ON(PKMAP_BASE + LAST_PKMAP*PAGE_SIZE >
FIXADDR_START);
 945                 BUILD_BUG_ON(VMALLOC_END                           >
PKMAP_BASE);
 946#endif
 947#define high_memory (-128UL << 20)
 948                 BUILD_BUG_ON(VMALLOC_START                         >=
VMALLOC_END);
 949#undef high_memory
 950#undef __FIXADDR_TOP
 951
 952#ifdef CONFIG_HIGHMEM
 953                   BUG_ON(PKMAP_BASE + LAST_PKMAP*PAGE_SIZE         >
FIXADDR_START);
 954                 BUG_ON(VMALLOC_END                                 >
PKMAP_BASE);
 955#endif
 956                 BUG_ON(VMALLOC_START                               >=
VMALLOC_END);
 957                    BUG_ON((unsigned long)high_memory               >
VMALLOC_START);
 958
 959             if (boot_cpu_data.wp_works_ok < 0)
 960                     test_wp_bit();
```

```
961
962             save_pg_dir();
963             zap_low_mappings(true);
964}
```

872 行，Intel IOMMU 架构在 Linux 上的初始化函数 pci_iommu_alloc。这个函数不是我们关注的重点，我们就不深入下去了，这里仅仅粗略地介绍一下。该函数首先通过读取 DMA Remapping table，来判断判断是否支持 DMAR 设备。随后调用 pci_swiotlb_init 函数对其进行初始化，解析 DMAR table，并逐一打印每个 dmar 项。最后设置全局变量 dma_ops，把初始化后的 swiotlb_dma_ops 传递给它，后者定义了 IOMMU 架构中所有的 swiotlb 方法。对 IOMMU 感兴趣的同学可以去查阅相关资料，这里就不详细介绍了。

878 行，totalram_pages 这个全局变量我们第一次遇见。它编译的时候初始化为 0，现在它就等于 free_all_bootmem 函数的返回值，该函数在 mm/bootmem.c 中定义：

```
unsigned long __init free_all_bootmem(void)
{
#ifdef CONFIG_NO_BOOTMEM
        /*
         * We need to use MAX_NUMNODES instead of NODE_DATA(0)->node_id
         *    because in some case like Node0 doesnt have RAM installed
         *    low ram will be on Node1
         * Use MAX_NUMNODES will make sure all ranges in early_node_map[]
         *    will be used instead of only Node0 related
         */
        return free_all_memory_core_early(MAX_NUMNODES);
#else
        unsigned long total_pages = 0;
        bootmem_data_t *bdata;

        list_for_each_entry(bdata, &bdata_list, list)
                total_pages += free_all_bootmem_core(bdata);

        return total_pages;
#endif
}
```

我们看到，由于 CONFIG_NO_BOOTMEM 起作用，并且 MAX_NUMNODES 为 1，所以函数直接调用 free_all_memory_core_early(1)，怎么样，前面说得没错吧，终于碰到了这个函数：

```
200unsigned long __init free_all_memory_core_early(int nodeid)
201{
202             int i;
203             u64 start, end;
```

```
204             unsigned long count = 0;
205             struct range *range = NULL;
206             int nr_range;
207
208             nr_range = get_free_all_memory_range(&range, nodeid);
209
210             for (i = 0; i < nr_range; i++) {
211                             start = range[i].start;
212                             end = range[i].end;
213                             count += end - start;
214                             __free_pages_memory(start, end);
215             }
216
217             return count;
218}
```

205 行的那个 range 结构很简单：

```
struct range {
        u64     start;
        u64     end;
};
```

所以首先 208 行调用 get_free_all_memory_range 函数：

```
393int __init get_free_all_memory_range(struct range **rangep, int nodeid)
 394{
 395             int i, count;
 396             u64 start = 0, end;
 397             u64 size;
 398             u64 mem;
 399             struct range *range;
 400             int nr_range;
 401
 402             count    = 0;
 403             for (i = 0; i < max_early_res && early_res[i].end; i++)
 404                             count++;
 405
 406             count *= 2;
 407
 408             size = sizeof(struct range) * count;
 409             end = get_max_mapped();
 410#ifdef MAX_DMA32_PFN
 411             if (end > (MAX_DMA32_PFN << PAGE_SHIFT))
 412                             start = MAX_DMA32_PFN << PAGE_SHIFT;
```

```
413#endif
414             mem = find_fw_memmap_area(start, end, size, sizeof(struct range));
415             if (mem == -1ULL)
416                     panic("can not find more space for range free");
417
418             range = __va(mem);
419             /* use early_node_map[] and early_res to get range array at first */
420             memset(range, 0, size);
421             nr_range = 0;
422
423             /* need to go over early_node_map to find out good range for node */
424             nr_range = add_from_early_node_map(range, count, nr_range, nodeid);
425#ifdef CONFIG_X86_32
426             subtract_range(range, count, max_low_pfn, -1ULL);
427#endif
428             subtract_early_res(range, count);
429             nr_range = clean_sort_range(range, count);
430
431             /* need to clear it ? */
432             if (nodeid == MAX_NUMNODES) {
433                     memset(&early_res[0], 0,
434                                     sizeof(struct early_res) * max_early_res);
435                     early_res = NULL;
436                     max_early_res = 0;
437             }
438
439             *rangep = range;
440             return nr_range;
441}
```

403 行，全局变量 max_early_res 和 early_res[]数组，老熟人了，一个循环得到目前已经分配了 early_res 元素的个数，把它的值乘以 2 赋给 size。409 行，调用 get_max_mapped 函数：

```
u64 __init get_max_mapped(void)
{
        u64 end = max_pfn_mapped;
        end <<= PAGE_SHIFT;
        return end;
}
```

该函数返回我们的老熟人，最后一个页框 max_pfn_mapped 对应的物理地址，赋值给内部变量 end（start 在 396 行被赋值为 0）。然后 414 行，调用 find_fw_memmap_area 函数，传给他的参数是 start、end、size 和 range 结构的大小：

```
u64 __init find_fw_memmap_area(u64 start, u64 end, u64 size, u64 align)
{
```

```
        return find_e820_area(start, end, size, align);
}
```

find_e820_area 不用多说了吧，从 e820.map[]数组中寻找到一块能够容纳 size 个字节的内存段，该内存段的首物理地址赋值给 get_free_all_memory_range 的内部变量 mem。418~421 行初始化这块区域。随后 424 行调用 add_from_early_node_map 函数：

```
int __init add_from_early_node_map(struct range *range, int az,
                                   int nr_range, int nid)
{
        int i;
        u64 start, end;

        /* need to go over early_node_map to find out good range for node */
        for_each_active_range_index_in_nid(i, nid) {
                start = early_node_map[i].start_pfn;
                end = early_node_map[i].end_pfn;
                nr_range = add_range(range, az, nr_range, start, end);
        }
        return nr_range;
}

int add_range(struct range *range, int az, int nr_range, u64 start, u64 end)
{
        if (start >= end)
                return nr_range;

        /* Out of slots: */
        if (nr_range >= az)
                return nr_range;

        range[nr_range].start = start;
        range[nr_range].end = end;

        nr_range++;

        return nr_range;
}
```

执行完毕 add_from_early_node_map 函数之后，range 执行的这块区域中，就形成了一个 range[nr_range]数组，每个数组元素对应 early_node_map[]的数组元素，表示 nr_range 块空闲内存空间的起始页框号和结束页框号。426 行 subtract_range 函数检验一下这个 range 是否有问题，并进行调整。428 行，调用 subtract_early_res 对产生冲突的地址进行调整：

```
static void __init subtract_early_res(struct range *range, int az)
{
	int i, count;
	u64 final_start, final_end;
	int idx = 0;

	count  = 0;
	for (i = 0; i < max_early_res && early_res[i].end; i++)
		count++;

	/* need to skip first one ?*/
	if (early_res != early_res_x)
		idx = 1;

#define DEBUG_PRINT_EARLY_RES 1

#if DEBUG_PRINT_EARLY_RES
	printk(KERN_INFO "Subtract (%d early reservations)\n", count);
#endif
	for (i = idx; i < count; i++) {
		struct early_res *r = &early_res[i];
#if DEBUG_PRINT_EARLY_RES
		printk(KERN_INFO "    #%d [%010llx - %010llx] %15s\n", i,
			r->start, r->end, r->name);
#endif
		final_start = PFN_DOWN(r->start);
		final_end = PFN_UP(r->end);
		if (final_start >= final_end)
			continue;
		subtract_range(range, az, final_start, final_end);
	}

}
```

对 early_res 体系熟悉的同学对上述代码一定不会困惑，我们看到 subtract_early_res 对地址进行调整，去掉那些已经被占用了的地址空间。回到 get_free_all_memory_range，最后两行，把 range 赋给结果参数 rangep，并且返回最终的 range 数组的元素个数 nr_range。

回到 free_all_memory_core_early 函数中，内部变量 range 有了，其元素个数 nr_range 也有了，那么 210~215 执行一个循环，将 range 数组的每一个元素调用__free_pages_memory 进行释放：

```
174static void __init __free_pages_memory(unsigned long start, unsigned long end)
 175{
```

```
176         int i;
177         unsigned long start_aligned, end_aligned;
178         int order = ilog2(BITS_PER_LONG);
179
180         start_aligned = (start + (BITS_PER_LONG - 1)) & ~(BITS_PER_LONG - 1);
181         end_aligned = end & ~(BITS_PER_LONG - 1);
182
183         if (end_aligned <= start_aligned) {
184                 for (i = start; i < end; i++)
185                         __free_pages_bootmem(pfn_to_page(i), 0);
186
187                 return;
188         }
189
190         for (i = start; i < start_aligned; i++)
191                 __free_pages_bootmem(pfn_to_page(i), 0);
192
193         for (i = start_aligned; i < end_aligned; i += BITS_PER_LONG)
194                 __free_pages_bootmem(pfn_to_page(i), order);
195
196         for (i = end_aligned; i < end; i++)
197                 __free_pages_bootmem(pfn_to_page(i), 0);
198}
```

函数主要执行 183~188 行代码，通过__free_pages_bootmem 函数释放对应号码的页框，从号码从 start 到 end 号。

下面来看看__free_pages_bootmem：

```
637void __meminit __free_pages_bootmem(struct page *page, unsigned int order)
638{
639         if (order == 0) {
640                 __ClearPageReserved(page);
641                 set_page_count(page, 0);
642                 set_page_refcounted(page);
643                 __free_page(page);
644         } else {
645                 int loop;
646
647                 prefetchw(page);
648                 for (loop = 0; loop < BITS_PER_LONG; loop++) {
649                         struct page *p = &page[loop];
650
651                         if (loop + 1 < BITS_PER_LONG)
```

```
652                                     prefetchw(p + 1);
653                             __ClearPageReserved(p);
654                             set_page_count(p, 0);
655                     }
656
657                     set_page_refcounted(page);
658                     __free_pages(page, order);
659             }
660}
```

我们传递进来的参数 order 为 0，所以来到 643 行，针对这个页面 page，著名的伙伴算法到来了，我们来看它的定义：

#define __free_page(page) __free_pages((page), 0)

释放页框的所有内核宏和函数都依赖于__free_pages()函数。它接收的参数为将要释放的第一个页框的页描述符的地址(page)和将要释放的一组连续页框的数量的对数（order）。该函数执行如下步骤：

1. 检查第一个页框是否真正属于动态内存（它的 PG_reserved 标志被清 0）；如果不是，则终止。
2. 减少 page->_count 使用计数器的值；如果它仍然大于或等于 0，则终止。
3. 如果 order 等于 0，那么该函数调用 free_hot_page()来释放页框给适当内存管理区的每 CPU 热高速缓存。
4. 如果 order 大于 0，那么它将页框加入到本地链表中，并调用 free_pages_bulk()函数把它们释放到适当内存管理区的伙伴系统中。

我们这里 order 为 0，所以调用 free_hot_page()，最终会调用__free_one_page。由于前面的 pglist 和 zone 的体系已经建立好，该函数对当前页面 page 对应的那个 zone 的 free_area 数组进行处理。由于这个地方是第一次触及该数组，那么这一次 free_hot_page 调用的 __free_one_page 将会找到全部伙伴，等于是初始化了整个伙伴算法系统。好了，怀疑我这句话的同志可以去看看博客“伙伴系统算法”

http://blog.csdn.net/yunsongice/archive/2010/01/22/5225155.aspx

回到 mem_init 函数中，伙伴系统建立起来以后，free_all_bootmem 返回空闲页面的总数给全局参数 totalram_pages。随后 880~886 行代码计算被保留的页面数，保存在内部变量 reservedpages 中。888 行，set_highmem_pages_init 函数，通过调用 add_highpages_work_fn 函数初始化 876MB 以上的高端页面，并把他们加入伙伴系统，最后计算出包含了这些高端页面的新的可用页面的数量 totalram_pages。

继续走，890 行，让内部变量 codesize、datasize、initsize 分别等于内核代码段、数据段和初始化相关函数指针空间段的大小。随后 894~936 行打印相关信息。942~957 是一群调试信息，略去。962 行 save_pg_dir()函数，来自同一文件：

```
char swsusp_pg_dir[PAGE_SIZE]
    __attribute__ ((aligned(PAGE_SIZE)));
static inline void save_pg_dir(void)
```

```
{
	memcpy(swsusp_pg_dir, swapper_pg_dir, PAGE_SIZE);
}
```

很简单，就是把页全局目录拷贝到全局变量 swsusp_pg_dir 数组中，做个备份。963 行，执行 zap_low_mappings(true)函数，这个函数也来自于同一个文件：

```
void zap_low_mappings(bool early)
{
	int i;

	/*
	 * Zap initial low-memory mappings.
	 *
	 * Note that "pgd_clear()" doesn't do it for
	 * us, because pgd_clear() is a no-op on i386.
	 */
	for (i = 0; i < KERNEL_PGD_BOUNDARY; i++) {
#ifdef CONFIG_X86_PAE
		set_pgd(swapper_pg_dir+i, __pgd(1 + __pa(empty_zero_page)));
#else
		set_pgd(swapper_pg_dir+i, __pgd(0));
#endif
	}

	if (early)
		__flush_tlb();
	else
		flush_tlb_all();
}
```

这个函数很简单，就是把前面我们在 arch/x86/kernel/head_32.S 中设置的页全局目录的前若干项清零。这若干项到底是多少项呢？我们看看 KERNEL_PGD_BOUNDARY 是什么东西：

```
#define KERNEL_PGD_BOUNDARY pgd_index(PAGE_OFFSET)
#define pgd_index(address) (((address) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))
#define PGDIR_SHIFT  22
#define PTRS_PER_PGD      1024
```

不错，0xc0000000>>22 & 1023= 768，这些也全局目录项代表虚拟地址前 3G 的页面，也就是所谓的用户区，我们在这里把它全清零了。

5.8.2 初始化 slab 分配器

回到 mm_init()函数，继续走，下一个函数 kmem_cache_init()，也是重点函数，用于初始化内核 slab 分配体系。这个函数来自文件 mm/slab.c

```
1375void __init kmem_cache_init(void)
1376{
1377        size_t left_over;
1378        struct cache_sizes *sizes;
1379        struct cache_names *names;
1380        int i;
1381        int order;
1382        int node;
1383
1384        if (num_possible_nodes() == 1)
1385                use_alien_caches = 0;
1386
1387        for (i = 0; i < NUM_INIT_LISTS; i++) {
1388                kmem_list3_init(&initkmem_list3[i]);
1389                if (i < MAX_NUMNODES)
1390                        cache_cache.nodelists[i] = NULL;
1391        }
1392        set_up_list3s(&cache_cache, CACHE_CACHE);
1393
……
1398        if (totalram_pages > (32 << 20) >> PAGE_SHIFT)
1399                slab_break_gfp_order = BREAK_GFP_ORDER_HI;
1400
……
1420
1421        node = numa_node_id();
1422
1423        /* 1) create the cache_cache */
1424        INIT_LIST_HEAD(&cache_chain);
1425        list_add(&cache_cache.next, &cache_chain);
1426        cache_cache.colour_off = cache_line_size();
1427        cache_cache.array[smp_processor_id()] = &initarray_cache.cache;
1428        cache_cache.nodelists[node] = &initkmem_list3[CACHE_CACHE + node];
1429
……
1434        cache_cache.buffer_size = offsetof(struct kmem_cache, nodelists) +
1435                                  nr_node_ids * sizeof(struct kmem_list3 *);
1436#if DEBUG
```

```
1437            cache_cache.obj_size = cache_cache.buffer_size;
1438#endif
1439            cache_cache.buffer_size = ALIGN(cache_cache.buffer_size,
1440                                                    cache_line_size());
1441            cache_cache.reciprocal_buffer_size =
1442                    reciprocal_value(cache_cache.buffer_size);
1443
1444            for (order = 0; order < MAX_ORDER; order++) {
1445                    cache_estimate(order, cache_cache.buffer_size,
1446                            cache_line_size(), 0, &left_over, &cache_cache.num);
1447                    if (cache_cache.num)
1448                            break;
1449            }
1450            BUG_ON(!cache_cache.num);
1451            cache_cache.gfporder = order;
1452            cache_cache.colour = left_over / cache_cache.colour_off;
1453            cache_cache.slab_size = ALIGN(cache_cache.num * sizeof(kmem_bufctl_t) +
1454                                            sizeof(struct slab), cache_line_size());
1455
1456            /* 2+3) create the kmalloc caches */
1457            sizes = malloc_sizes;
1458            names = cache_names;
1459
……
1466            sizes[INDEX_AC].cs_cachep = kmem_cache_create(names[INDEX_AC].name,
1467                                                    sizes[INDEX_AC].cs_size,
1468                                                    ARCH_KMALLOC_MINALIGN,
1469
ARCH_KMALLOC_FLAGS|SLAB_PANIC,
1470                                                    NULL);
1471
1472            if (INDEX_AC != INDEX_L3) {
1473                    sizes[INDEX_L3].cs_cachep =
1474                            kmem_cache_create(names[INDEX_L3].name,
1475                                    sizes[INDEX_L3].cs_size,
1476                                    ARCH_KMALLOC_MINALIGN,
1477                                    ARCH_KMALLOC_FLAGS|SLAB_PANIC,
1478                                    NULL);
1479            }
1480
1481            slab_early_init = 0;
1482
1483            while (sizes->cs_size != ULONG_MAX) {
……
```

```
1491                    if (!sizes->cs_cachep) {
1492                            sizes->cs_cachep = kmem_cache_create(names->name,
1493                                    sizes->cs_size,
1494                                    ARCH_KMALLOC_MINALIGN,
1495
ARCH_KMALLOC_FLAGS|SLAB_PANIC,
1496                                    NULL);
1497                    }
1498#ifdef CONFIG_ZONE_DMA
1499                    sizes->cs_dmacachep = kmem_cache_create(
1500                                    names->name_dma,
1501                                    sizes->cs_size,
1502                                    ARCH_KMALLOC_MINALIGN,
1503
ARCH_KMALLOC_FLAGS|SLAB_CACHE_DMA|
1504                                            SLAB_PANIC,
1505                                    NULL);
1506#endif
1507                    sizes++;
1508                    names++;
1509            }
1510            /* 4) Replace the bootstrap head arrays */
1511            {
1512                    struct array_cache *ptr;
1513
1514                    ptr = kmalloc(sizeof(struct arraycache_init), GFP_NOWAIT);
1515
1516                    BUG_ON(cpu_cache_get(&cache_cache) != &initarray_cache.cache);
1517                    memcpy(ptr, cpu_cache_get(&cache_cache),
1518                            sizeof(struct arraycache_init));
1519                    /*
1520                     * Do not assume that spinlocks can be initialized via memcpy:
1521                     */
1522                    spin_lock_init(&ptr->lock);
1523
1524                    cache_cache.array[smp_processor_id()] = ptr;
1525
1526                    ptr = kmalloc(sizeof(struct arraycache_init), GFP_NOWAIT);
1527
1528                    BUG_ON(cpu_cache_get(malloc_sizes[INDEX_AC].cs_cachep)
1529                            != &initarray_generic.cache);
1530                    memcpy(ptr, cpu_cache_get(malloc_sizes[INDEX_AC].cs_cachep),
1531                            sizeof(struct arraycache_init));
1532                    /*
```

```
1533                     * Do not assume that spinlocks can be initialized via memcpy:
1534                     */
1535                    spin_lock_init(&ptr->lock);
1536
1537                    malloc_sizes[INDEX_AC].cs_cachep->array[smp_processor_id()] =
1538                            ptr;
1539            }
1540            /* 5) Replace the bootstrap kmem_list3's */
1541            {
1542                    int nid;
1543
1544                    for_each_online_node(nid) {
1545                            init_list(&cache_cache, &initkmem_list3[CACHE_CACHE +
nid], nid);
1546
1547                            init_list(malloc_sizes[INDEX_AC].cs_cachep,
1548                                    &initkmem_list3[SIZE_AC + nid], nid);
1549
1550                            if (INDEX_AC != INDEX_L3) {
1551                                    init_list(malloc_sizes[INDEX_L3].cs_cachep,
1552                                            &initkmem_list3[SIZE_L3 + nid], nid);
1553                            }
1554                    }
1555            }
1556
1557            g_cpucache_up = EARLY;
1558}
```

去掉了若干行代码，别担心，全是注释。1387 行，宏 NUM_INIT_LISTS 的值为

```
#define NUM_INIT_LISTS (3 * MAX_NUMNODES)
```

也就是 3。执行 3 次循环，调用 kmem_list3_init 函数初始化全局变量 initkmem_list3[]数组。该数组的定义也在同一个文件：

```
struct kmem_list3 {
        struct list_head slabs_partial; /* partial list first, better asm code */
        struct list_head slabs_full;
        struct list_head slabs_free;
        unsigned long free_objects;
        unsigned int free_limit;
        unsigned int colour_next;       /* Per-node cache coloring */
        spinlock_t list_lock;
        struct array_cache *shared;     /* shared per node */
        struct array_cache **alien;     /* on other nodes */
        unsigned long next_reap;        /* updated without locking */
        int free_touched;               /* updated without locking */
```

```
};
struct kmem_list3 __initdata initkmem_list3[NUM_INIT_LISTS];
```

而初始化它每个元素的函数也很简单，位于同一个文件中：

```
static void kmem_list3_init(struct kmem_list3 *parent)
{
        INIT_LIST_HEAD(&parent->slabs_full);
        INIT_LIST_HEAD(&parent->slabs_partial);
        INIT_LIST_HEAD(&parent->slabs_free);
        parent->shared = NULL;
        parent->alien = NULL;
        parent->colour_next = 0;
        spin_lock_init(&parent->list_lock);
        parent->free_objects = 0;
        parent->free_touched = 0;
}
```

随后 1392 行，调用 set_up_list3s 函数为全局变量 cache_cache 初始化它的 lists 字段。全局变量 cache_cache 太重要了，它是 slab 体系的核心数据结构，其定义如下：

```
static struct kmem_cache cache_cache = {
        .batchcount = 1,
        .limit = BOOT_CPUCACHE_ENTRIES,
        .shared = 1,
        .buffer_size = sizeof(struct kmem_cache),
        .name = "kmem_cache",
};

struct kmem_cache {
/* 1) per-cpu data, touched during every alloc/free */
        struct array_cache *array[NR_CPUS];
/* 2) Cache tunables. Protected by cache_chain_mutex */
        unsigned int batchcount;
        unsigned int limit;
        unsigned int shared;

        unsigned int buffer_size;
        u32 reciprocal_buffer_size;
/* 3) touched by every alloc & free from the backend */

        unsigned int flags;         /* constant flags */
        unsigned int num;           /* # of objs per slab */
```

```
/* 4) cache_grow/shrink */
        /* order of pgs per slab (2^n) */
        unsigned int gfporder;

        /* force GFP flags, e.g. GFP_DMA */
        gfp_t gfpflags;

        size_t colour;                  /* cache colouring range */
        unsigned int colour_off;  /* colour offset */
        struct kmem_cache *slabp_cache;
        unsigned int slab_size;
        unsigned int dflags;        /* dynamic flags */

        /* constructor func */
        void (*ctor)(void *obj);

/* 5) cache creation/removal */
        const char *name;
        struct list_head next;

/* 6) statistics */
#ifdef CONFIG_DEBUG_SLAB
……slab 调试相关数据结构，省略。
#endif /* CONFIG_DEBUG_SLAB */
……一些注释……
        struct kmem_list3 *nodelists[MAX_NUMNODES];
        /*
          * Do not add fields after nodelists[]
          */
};
```

kmem_cache 数据结构的注释写得很详细，大家可以去仔细看看。set_up_list3s 函数在本文件中：

```
#define    CACHE_CACHE 0
static void __init set_up_list3s(struct kmem_cache *cachep, int index)
{
        int node;

        for_each_online_node(node) {
                cachep->nodelists[node] = &initkmem_list3[index + node];
                cachep->nodelists[node]->next_reap = jiffies +
                        REAPTIMEOUT_LIST3 +
                        ((unsigned long)cachep) % REAPTIMEOUT_LIST3;
```

```
        }
}
```

很简单，就是把全局 cache_cache 变量的 nodelists[0]设置成刚才初始化好的 initkmem_list3[0]的地址。如果是 NUMA 体系，则每个 NODE 有一个 initkmem_list3 数组，由 cache_cache 的 nodelists[]数组的每个元素指向。

继续走，1424 行，初始化一个内核全局链表 cache_chain，这个东西就是个很简单的 list_head 结构，定义在同一个文件中：
static struct list_head cache_chain;
随后调用 list_add 将它与 cache_cache 链接起来，接下来 1426~1454 行初始化这个 cache_cache 的其他字段

来到 1457 行，又一个重要的全局变量 malloc_sizes。这个变量关系着通用 slab 分配器的初始化，有关专用/通用 slab 分配器的概念是 Linux kernel 内存管理的核心内容，对这个概念不熟悉的同学请重新学习一下 Linux 内核管理。来看这个变量的定义，在同一文件的 570 行：

```
570 struct cache_sizes malloc_sizes[] = {
571 #define CACHE(x) { .cs_size = (x) },
572 #include <linux/kmalloc_sizes.h>
573         CACHE(ULONG_MAX)
574 #undef CACHE
575};
```

cache_sizes 是个如下结构：

```
struct cache_sizes {
        size_t                  cs_size;
        struct kmem_cache *cs_cachep;
#ifdef CONFIG_ZONE_DMA
        struct kmem_cache *cs_dmacachep;
#endif
};
```

那么 malloc_sizes[]数组的全部元素来自 linux/kmalloc_sizes.h 文件，下面就来看看这个文件的全部内容：

```
#if (PAGE_SIZE == 4096)
        CACHE(32)
#endif
        CACHE(64)
#if L1_CACHE_BYTES < 64
        CACHE(96)
#endif
        CACHE(128)
```

```
#if L1_CACHE_BYTES < 128
        CACHE(192)
#endif
        CACHE(256)
        CACHE(512)
        CACHE(1024)
        CACHE(2048)
        CACHE(4096)
        CACHE(8192)
        CACHE(16384)
        CACHE(32768)
        CACHE(65536)
        CACHE(131072)
#if KMALLOC_MAX_SIZE >= 262144
        CACHE(262144)
#endif
#if KMALLOC_MAX_SIZE >= 524288
        CACHE(524288)
#endif
#if KMALLOC_MAX_SIZE >= 1048576
        CACHE(1048576)
#endif
#if KMALLOC_MAX_SIZE >= 2097152
        CACHE(2097152)
#endif
#if KMALLOC_MAX_SIZE >= 4194304
        CACHE(4194304)
#endif
#if KMALLOC_MAX_SIZE >= 8388608
        CACHE(8388608)
#endif
#if KMALLOC_MAX_SIZE >= 16777216
        CACHE(16777216)
#endif
#if KMALLOC_MAX_SIZE >= 33554432
        CACHE(33554432)
#endif
```

全局变量 malloc_sizes 数组的初始化就在编译 vmlinux 的时候定义成上述形式，其首地址在函数中被赋给了内部变量 sizes。1458 行，cache_names 是一个跟 malloc_sizes 差不多的全局变量数组：

```
static struct cache_names __initdata cache_names[] = {
#define CACHE(x) { .name = "size-" #x, .name_dma = "size-" #x "(DMA)" },
#include <linux/kmalloc_sizes.h>
```

```
        {NULL,}
#undef CACHE
};
```

其首地址同样被赋给了内部变量 names。那么，1466-1509 行，调用 kmem_cache_create 函数为每一个通用 slab 分配器初始化 cache。。这个函数首先根据参数确定处理新高速缓存的最佳方法（例如，是在 slab 的内部还是外部包含 slab 描述符）。然后它从 cache_cache 普通高速缓存中为新的高速缓存分配一个高速缓存描述符 kmem_cache_t：

```
(kmem_cache_t *) kmem_cache_alloc(&cache_cache, SLAB_KERNEL);
```

并把这个描述符插入到高速缓存描述符的 cache_chain 链表中（当获得了用于保护链表避免被同时访问的 cache_chain_sem 信号量后，插入操作完成）。具体的细节我就不多说了，有兴趣的同学可以参照博文“ slab 分配器 ”

http://blog.csdn.net/yunsongice/archive/2010/01/30/5272715.aspx

以及源代码进行分析。

上述代码执行完毕后，slab 通用分配器 kmalloc 函数就可以使用了。所以我们看到 1514 行调用 kmalloc 分配了一个 arraycache_init 结构，随后 1516~1538 行代码初始化涉及每 CPU 的 cache_cache.array，把以前初始化时期那些没用的数据给覆盖掉。

1541~1555 调用 init_list 函数把 cache_cache 和 malloc_sizes[INDEX_AC].cs_cachep 的 kmem_list3 结构清空，因为没用了：

```
static void init_list(struct kmem_cache *cachep, struct kmem_list3 *list,
                      int nodeid)
{
        struct kmem_list3 *ptr;

        ptr = kmalloc_node(sizeof(struct kmem_list3), GFP_NOWAIT, nodeid);
        BUG_ON(!ptr);

        memcpy(ptr, list, sizeof(struct kmem_list3));
        /*
         * Do not assume that spinlocks can be initialized via memcpy:
         */
        spin_lock_init(&ptr->list_lock);

        MAKE_ALL_LISTS(cachep, ptr, nodeid);
        cachep->nodelists[nodeid] = ptr;
}
```

kmem_cache_init 函数的最后一行，把全局变量 g_cpucache_up 设置成 EARLY，slab 分配器就初始化完了。

5.8.3 初始化非连续内存区

回到 mm_init()函数，继续走，下一个函数 pgtable_cache_init ()，不知道咋的，是个空函数，也许是保留着以后开发吧。最后一个函数是 vmalloc_init()，来自 mm/vmalloc.c：

```
1088void __init vmalloc_init(void)
1089{
1090             struct vmap_area *va;
1091             struct vm_struct *tmp;
1092             int i;
1093
1094             for_each_possible_cpu(i) {
1095                          struct vmap_block_queue *vbq;
1096
1097                          vbq = &per_cpu(vmap_block_queue, i);
1098                          spin_lock_init(&vbq->lock);
1099                          INIT_LIST_HEAD(&vbq->free);
1100             }
1101
1102             /* Import existing vmlist entries. */
1103             for (tmp = vmlist; tmp; tmp = tmp->next) {
1104                          va = kzalloc(sizeof(struct vmap_area), GFP_NOWAIT);
1105                          va->flags = tmp->flags | VM_VM_AREA;
1106                          va->va_start = (unsigned long)tmp->addr;
1107                          va->va_end = va->va_start + tmp->size;
1108                          __insert_vmap_area(va);
1109             }
1110
1111             vmap_area_pcpu_hole = VMALLOC_END;
1112
1113             vmap_initialized = true;
1114}
```

1090 行，vmap_area 结构，线性区描述符，就是以前的 vm_area_struct 结构，它的字段如下所示：

```
struct vmap_area {
      unsigned long va_start;
      unsigned long va_end;
      unsigned long flags;
      struct rb_node rb_node;         /* address sorted rbtree */
      struct list_head list;        /* address sorted list */
      struct list_head purge_list;     /* "lazy purge" list */
```

```
        void *private;
        struct rcu_head rcu_head;
};
```

内核内存管理初始化至此，已经接近尾声了，回忆一下：

（1）内存区的开始部分包含的是对前 896MB RAM 进行映射的线性地址。直接映射的物理内存末尾所对应的线性地址保存在 high_memory 全局变量中。当物理内存小于 896MB，则线性地址 0xc0000000 以后的 896MB 与其一一对应；当物理内存大于 896MB 而小于 4GB 时，只直接映射前 896MB 的地址到 0xc0000000 以后的线性空间，然后把线性空间的其他部分与 896MB 和 4GB 物理空间映射起来，称为动态重映射，这是本博的重点；当物理内存大于 4GB，则需要考虑 PAE 的情况，其他的东东没什么区别，我们不做过多的回忆了。

（2）内核的页表由内核页全局目录变量 swapper_pg_dir 维护；pagetable_init()建立内核页表项。

（3）内存区的结尾部分包含的是固定映射的线性地址，主要用于存放一些常量线性地址，具体查看“高端内存映射”博文。

（4）从 PKMAP_BASE 开始，我们查找用于高端内存页框的永久内核映射的线性地址，具体查看“高端内存映射”博文。

（5）其余的线性地址可以用于非连续内存区。在物理内存映射的末尾与第一个内存区之间插入一个大小为 8MB（宏 VMALLOC_OFFSET）的安全区，目的是为了“捕获”对内存的越界访问。出于同样的理由，插入其他 4KB 大小的安全区来隔离非连续的内存区。

那么，vmalloc_init 函数就是为第（5）项——非连续内存区管理进行初始化。非连续内存区保留的线性地址空间的起始地址由 VMALLOC_START 宏定义，而末尾地址由 VMALLOC_END 宏定义：

```
#define VMALLOC_START ((unsigned long)high_memory + VMALLOC_OFFSET)
# define VMALLOC_END    (PKMAP_BASE - 2 * PAGE_SIZE)
```

关于非连续内存区的相关知识，请查阅博客“非连续内存区”
http://blog.csdn.net/yunsongice/archive/2010/04/27/5536197.aspx
相关内容。

5.9 初始化调度程序

回到 start_kernel 函数中，mm_init()执行后，所有的绝大多数内存管理的初始化都完毕，后面的代码可以开开心心的使用 Linux 复杂、庞大而又高效的内存管理器了。来看下一个函数，超级重点的进程调度初始化函数 sched_init()。不过自从 Linux 2.6.23（2007 年 5 月），内核引入了一种所谓的完全公平调度程序（Completely Fair Scheduler，CFS），试图按照对 CPU 时间的“最大需求（gravest need）”运行任务；这有助于确保每个进程可以获得对 CPU 的

公平共享。

虽然 CFS 理论上非常高效，但它的实际性能并不如预期，因为如果某个任务休眠时间 “ 非常短 ”，那么 CFS 不会将该任务视为休眠任务 —— 短暂休眠的进程可能会获得一些额外时间，但是决不会超过它的未休眠时间。因此，很多应用软件并没有享受到实际的好处。比如我们的 Android 就没有使用 CFS，仍然使用传统的 O(1)调度程序。

因此，我们避免给大家带来争议的 CFS 算法，还是仍然分析 O(1)调度的初始化程序，那么我们这里就分析一个版本稍微低一点的内核——Linux 2.6.18，来看看它的 kernel/sched.c：

```
6728void __init sched_init(void)
6729{
6730            int i, j, k;
6731
6732            for_each_possible_cpu(i) {
6733                        struct prio_array *array;
6734                        struct rq *rq;
6735
6736                        rq = cpu_rq(i);
6737                        spin_lock_init(&rq->lock);
6738                        lockdep_set_class(&rq->lock, &rq->rq_lock_key);
6739                        rq->nr_running = 0;
6740                        rq->active = rq->arrays;
6741                        rq->expired = rq->arrays + 1;
6742                        rq->best_expired_prio = MAX_PRIO;
6743
6744#ifdef CONFIG_SMP
6745                        rq->sd = NULL;
6746                        for (j = 1; j < 3; j++)
6747                                    rq->cpu_load[j] = 0;
6748                        rq->active_balance = 0;
6749                        rq->push_cpu = 0;
6750                        rq->migration_thread = NULL;
6751                        INIT_LIST_HEAD(&rq->migration_queue);
6752#endif
6753                        atomic_set(&rq->nr_iowait, 0);
6754
6755                        for (j = 0; j < 2; j++) {
6756                                    array = rq->arrays + j;
6757                                    for (k = 0; k < MAX_PRIO; k++) {
6758                                                INIT_LIST_HEAD(array->queue + k);
6759                                                __clear_bit(k, array->bitmap);
6760                                    }
6761                                    // delimiter for bitsearch
```

```
6762                                    __set_bit(MAX_PRIO, array->bitmap);
6763                            }
6764            }
6765
6766            set_load_weight(&init_task);
6767
6768#ifdef CONFIG_RT_MUTEXES
6769            plist_head_init(&init_task.pi_waiters, &init_task.pi_lock);
6770#endif
6771
6772            /*
6773             * The boot idle thread does lazy MMU switching as well:
6774             */
6775            atomic_inc(&init_mm.mm_count);
6776            enter_lazy_tlb(&init_mm, current);
6777
6778            /*
6779             * Make us the idle thread. Technically, schedule() should not be
6780             * called from this thread, however somewhere below it might be,
6781             * but because we are the idle thread, we just pick up running again
6782             * when this runqueue becomes "idle".
6783             */
6784            init_idle(current, smp_processor_id());
6785}
```

6733 行，prio_array 结构，用于计算进程优先级的数据结构，来自同一个文件：

```
struct prio_array {
        unsigned int nr_active;
        DECLARE_BITMAP(bitmap, MAX_PRIO+1); /* include 1 bit for delimiter */
        struct list_head queue[MAX_PRIO];
};
#define MAX_USER_RT_PRIO    100
#define MAX_RT_PRIO             MAX_USER_RT_PRIO
#define MAX_PRIO            (MAX_RT_PRIO + 40)
```

prio_array 数据结构都表示一个可运行进程的集合，并包括 140 个双向链表头（每个链表对应一个可能的进程优先级）、一个优先级位图和一个集合中所包含的进程数量的计数器。

6771 行，rq 结构，著名的运行队列（run queue），操作系统课程的核心概念：

```
struct rq {
        spinlock_t lock;
        unsigned long nr_running;
        unsigned long raw_weighted_load;
#ifdef CONFIG_SMP
```

```
        unsigned long cpu_load[3];
#endif
        unsigned long long nr_switches;

        unsigned long nr_uninterruptible;

        unsigned long expired_timestamp;
        unsigned long long timestamp_last_tick;
        struct task_struct *curr, *idle;
        struct mm_struct *prev_mm;
        struct prio_array *active, *expired, arrays[2];
        int best_expired_prio;
        atomic_t nr_iowait;

#ifdef CONFIG_SMP
        struct sched_domain *sd;

        /* For active balancing */
        int active_balance;
        int push_cpu;
        int cpu;            /* cpu of this runqueue */

        struct task_struct *migration_thread;
        struct list_head migration_queue;
#endif

#ifdef CONFIG_SCHEDSTATS
……
#endif
        struct lock_class_key rq_lock_key;
};
```

rq 数据结构中最重要的字段是与可运行进程的链表相关的字段。系统中的每个可运行进程属于且只属于一个运行队列。只要可运行进程保持在同一个运行队列中，它就只可能在拥有该运行队列的 CPU 上执行。运行队列 arrays[2]字段是一个包含两个 prio_array 结构的数组。每个数据结构都表示一个可运行进程的集合，并包括 140 个双向链表头（每个链表对应一个可能的进程优先级）、一个优先级位图和本运行队列所包含的进程数量计数器。

那么，这个 rq 是如何初始化的呢？在 kernel/sched.c 的 269 行有这么一行代码：

```
static DEFINE_PER_CPU(struct rq, runqueues);
```

DEFINE_PER_CPU 还记得吧：

```
11 #define DEFINE_PER_CPU(type, name) \
12 __attribute__((__section__(".data.percpu"))) __typeof__(type) per_cpu__##name
```

也就是说，初始化的时候，把类型为 rq 的每 CPU 变量 per_cpu__runqueues 存放进.data.percpu 段中。于是 6773 行代码得到这个变量，并且把他的地址赋给内部变量 rq。然后 6774 行开始初始化这个 rq。

最后，sched_init 函数调用 init_idle，来自同一个文件，传递给他的参数是当前的进程，也就是 0 号进程的 task_struct 结构和当前 CPU 的 id：

```
void __devinit init_idle(struct task_struct *idle, int cpu)
{
	struct rq *rq = cpu_rq(cpu);
	unsigned long flags;

	idle->timestamp = sched_clock();
	idle->sleep_avg = 0;
	idle->array = NULL;
	idle->prio = idle->normal_prio = MAX_PRIO;
	idle->state = TASK_RUNNING;
	idle->cpus_allowed = cpumask_of_cpu(cpu);
	set_task_cpu(idle, cpu);

	spin_lock_irqsave(&rq->lock, flags);
	rq->curr = rq->idle = idle;
#if defined(CONFIG_SMP) && defined(__ARCH_WANT_UNLOCKED_CTXSW)
	idle->oncpu = 1;
#endif
	spin_unlock_irqrestore(&rq->lock, flags);

	/* Set the preempt count _outside_ the spinlocks! */
#if defined(CONFIG_PREEMPT) && !defined(CONFIG_PREEMPT_BKL)
	task_thread_info(idle)->preempt_count = (idle->lock_depth >= 0);
#else
	task_thread_info(idle)->preempt_count = 0;
#endif
}
```

init_idle 函数把 0 号进程的 task_struct 结构调度相关的字段给赋上值，具体的细节比较简单，我们就不一一说明了。init_idle 函数结束后，整个内核的最核心部分——调度程序——就算完成了，如果您对 O(1)调度器有更多的兴趣，请访问博客“ 进程调度的数据结构和优先级 ” http://blog.csdn.net/yunsongice/archive/2010/04/25/5526256.aspx。

回到 start_kernel，596 行 preempt_disable()，禁止内核抢占。不过我们的.config 已经没有设置 CONFIG_PREEMPT 了，所以这个函数是个空函数。597~601，再一次检查一下是否已经屏蔽了所有中断：local_irq_disable 函数，我们早已见过了。

继续走，602 行调用 rcu_init()初始化 rcu 机制。对这个读-拷贝-更新（RCU）机制感兴趣的同学可以参考博客“ RCU 机制 ”
http://blog.csdn.net/yunsongice/archive/2010/05/19/5607680.aspx

start_kernel 的 603 行，调用 radix_tree_init()初始化页高速缓存需要使用的基树：

```
void __init radix_tree_init(void)
{
	radix_tree_node_cachep = kmem_cache_create("radix_tree_node",
			sizeof(struct radix_tree_node), 0,
			SLAB_PANIC | SLAB_RECLAIM_ACCOUNT,
			radix_tree_node_ctor);
	radix_tree_init_maxindex();
	hotcpu_notifier(radix_tree_callback, 0);
}
```

函数很简单，调用 slab 分配器分配一个专有的 cache 结构 kmem_cache，其首地址由全局变量 radix_tree_node_cachep 指向，用来缓存 radix_tree_node 结构，其名称也叫 radix_tree_node。

有关基树的详细内容，请查阅博客“ 磁盘高速缓存 ”
http://blog.csdn.net/yunsongice/archive/2010/08/23/5833154.aspx

5.10 初始化中断处理系统

start_kernel 接下来要做的事是初始化中断处理系统。整个内核的中断系统的核心就是我们在“ 初始化中断描述符表 ”里面设置的那个中断描述符表。而这个表的前 19 个表项我们已经在“ 初始化异常服务 ”中设置为了一些中断和异常的服务。内核接下来会如何设置其余的表项呢？

前面提到过高级可编程中断控制器 APIC，这里简单地提一下它的体系结构，简单地说就是由两部分组成：本地高级中断控制器（Local APIC，LAPIC），位于每个 CPU 中，主要负责传递中断信号到指定的处理器中；I/O 高级中断控制器（I/O APIC，）负责搜集来自 I/O 设备的中断信号并分发给 LAPIC，形成一个多级 APIC 中断分发网络。

接下来的大部分初始化工作都将围绕着 LAPIC 和 IOAPIC 这两个东西进行。

5.10.1 设置 APIC 中断服务

回到 start_kernel，由于我们没有配置 CONFIG_SPARSE_IRQ，所以 605 行的 early_irq_init()函数是个空函数。606 行，调用 init_IRQ 函数，其本质上是调用 x86_init.irqs.intr_init 函数，

也就是前面“拷贝可用内存区信息”的那个 x86_init 中看到的那个 native_init_IRQ 函数，用来初始化 LAPIC：

```
235void __init native_init_IRQ(void)
236{
237             int i;
238
239             /* Execute any quirks before the call gates are initialised: */
240             x86_init.irqs.pre_vector_init();
241
242             apic_intr_init();
243
244             /*
245               * Cover the whole vector space, no vector can escape
246               * us. (some of these will be overridden and become
247               * 'special' SMP interrupts)
248               */
249             for (i = FIRST_EXTERNAL_VECTOR; i < NR_VECTORS; i++) {
250                             /* IA32_SYSCALL_VECTOR could be used in trap_init already. */
251                             if (!test_bit(i, used_vectors))
252                                             set_intr_gate(i, interrupt[i-FIRST_EXTERNAL_VECTOR]);
253             }
254
255             if (!acpi_ioapic)
256                             setup_irq(2, &irq2);
257
258#ifdef CONFIG_X86_32
259             /*
260               * External FPU? Set up irq13 if so, for
261               * original braindamaged IBM FERR coupling.
262               */
263             if (boot_cpu_data.hard_math && !cpu_has_fpu)
264                             setup_irq(FPU_IRQ, &fpu_irq);
265
266             irq_ctx_init(smp_processor_id());
267#endif
268}
```

该函数初始化中断向量表中前面一些我们没有设置的表项，首先是 240，调用 x86_init.irqs.pre_vector_init 函数，也就是 init_ISA_irqs 函数：

```
101void __init init_ISA_irqs(void)
102{
103             int i;
```

```
104
105#if defined(CONFIG_X86_64) || defined(CONFIG_X86_LOCAL_APIC)
106             init_bsp_APIC();
107#endif
108             legacy_pic->init(0);
109
110             /*
111              * 16 old-style INTA-cycle interrupts:
112              */
113             for (i = 0; i < legacy_pic->nr_legacy_irqs; i++) {
114                          struct irq_desc *desc = irq_to_desc(i);
115
116                          desc->status = IRQ_DISABLED;
117                          desc->action = NULL;
118                          desc->depth = 1;
119
120                          set_irq_chip_and_handler_name(i, &i8259A_chip,
121                                                                  handle_level_irq, "XT");
122             }
123}
```

106 行，由于我们配置了 CONFIG_X86_LOCAL_APIC，所以首先调用 init_bsp_APIC 函数来配置本地 LAPIC 芯片。该函数调用 apic_read 或 apic_write 调用全局变量 apic 的 read 和 write 方法。先说说这个全局变量 apic 的数据结构，其代表一块 LAPIC 控制器芯片，于 arch/x86/include/asm/apic.h 文件中定义：

```
struct apic {
        char *name;

        int (*probe)(void);
        int (*acpi_madt_oem_check)(char *oem_id, char *oem_table_id);
        int (*apic_id_registered)(void);

……


        /* apic ops */
        u32 (*read)(u32 reg);
        void (*write)(u32 reg, u32 v);
        u64 (*icr_read)(void);
        void (*icr_write)(u32 low, u32 high);
        void (*wait_icr_idle)(void);
        u32 (*safe_wait_icr_idle)(void);
};
```

而全局变量 apic 在文件 arch/x86/kernel/apic/probe_32.c 文件中被定义成：
struct apic *apic = &apic_default;
于是乎，编译 vmlinux 的时候，apic_default 就成为了该变量实际的值：

```
struct apic apic_default = {

        .name                   = "default",
        .probe                  = probe_default,
        .acpi_madt_oem_check        = NULL,
        .apic_id_registered     = default_apic_id_registered,

……

        .read                   = native_apic_mem_read,
        .write                  = native_apic_mem_write,
        .icr_read               = native_apic_icr_read,
        .icr_write              = native_apic_icr_write,
        .wait_icr_idle              = native_apic_wait_icr_idle,
        .safe_wait_icr_idle     = native_safe_apic_wait_icr_idle,
};
```

我们看到，init_bsp_APIC 函数实际上调用 native_apic_mem_read 和 native_apic_mem_write 等方法。回到 init_ISA_irqs 中，108 行又是一个全局变量，legacy_pic：
struct legacy_pic *legacy_pic = &default_legacy_pic;
其值被初始化成了 default_legacy_pic：

```
struct legacy_pic default_legacy_pic = {
        .nr_legacy_irqs = NR_IRQS_LEGACY,
        .chip    = &i8259A_chip,
        .mask_all    = mask_8259A,
        .restore_mask = unmask_8259A,
        .init = init_8259A,
        .irq_pending = i8259A_irq_pending,
        .make_irq = make_8259A_irq,
};

struct legacy_pic *legacy_pic = &default_legacy_pic;
```

我们熟悉的 8259A 中断控制器芯片就是这个 default_legacy_pic，所以 init_ISA_irqs 的 108 行是调用该芯片的初始化函数 init_8259A，初始化了 8259A 的一些硬件状态，保证 8259A 能够正常工作：

```

static void init_8259A(int auto_eoi)
```

```
{
	unsigned long flags;

	i8259A_auto_eoi = auto_eoi;

	raw_spin_lock_irqsave(&i8259A_lock, flags);

	outb(0xff, PIC_MASTER_IMR);	/* mask all of 8259A-1 */
	outb(0xff, PIC_SLAVE_IMR);	/* mask all of 8259A-2 */

	/*
	 * outb_pic - this has to work on a wide range of PC hardware.
	 */
	outb_pic(0x11, PIC_MASTER_CMD); /* ICW1: select 8259A-1 init */

	/* ICW2: 8259A-1 IR0-7 mapped to 0x30-0x37 on x86-64,
	    to 0x20-0x27 on i386 */
	outb_pic(IRQ0_VECTOR, PIC_MASTER_IMR);

	/* 8259A-1 (the master) has a slave on IR2 */
	outb_pic(1U << PIC_CASCADE_IR, PIC_MASTER_IMR);

	if (auto_eoi)	/* master does Auto EOI */
		outb_pic(MASTER_ICW4_DEFAULT | PIC_ICW4_AEOI, PIC_MASTER_IMR);
	else		/* master expects normal EOI */
		outb_pic(MASTER_ICW4_DEFAULT, PIC_MASTER_IMR);

	outb_pic(0x11, PIC_SLAVE_CMD);	/* ICW1: select 8259A-2 init */

	/* ICW2: 8259A-2 IR0-7 mapped to IRQ8_VECTOR */
	outb_pic(IRQ8_VECTOR, PIC_SLAVE_IMR);
	/* 8259A-2 is a slave on master's IR2 */
	outb_pic(PIC_CASCADE_IR, PIC_SLAVE_IMR);
	/* (slave's support for AEOI in flat mode is to be investigated) */
	outb_pic(SLAVE_ICW4_DEFAULT, PIC_SLAVE_IMR);

	if (auto_eoi)
		/*
		 * In AEOI mode we just have to mask the interrupt
		 * when acking.
		 */
		i8259A_chip.mask_ack = disable_8259A_irq;
	else
		i8259A_chip.mask_ack = mask_and_ack_8259A;
```

```
    udelay(100);        /* wait for 8259A to initialize */

    outb(cached_master_mask, PIC_MASTER_IMR); /* restore master IRQ mask */
    outb(cached_slave_mask, PIC_SLAVE_IMR);     /* restore slave IRQ mask */

    raw_spin_unlock_irqrestore(&i8259A_lock, flags);
}
```

函数具体内容我就不详细分析了，无非是些 outb 或者 outb_pic 通过这个芯片所占用的总线端口对它进行初始化。回到 init_ISA_irqs，113 行，刚才看见 nr_legacy_irqs 为 NR_IRQS_LEGACY，也就是 16 次循环，将全局 irq_desc[irq]数组的 16 个 irq_desc 元素进行初始化；然后 16 次通过 set_irq_chip_and_handler_name 函数将每个 irq_desc 的 chip 字段通过 set_irq_chip 设置成全局变量 i8259A_chip，以及将每个 irq_desc 的 handle_irq 和 name 字段通过__set_irq_handler 分别设置为 handle_level_irq 和” XT”。

回到 native_init_IRQ 中，随后 242 行调用 apic_intr_init()函数：

```
202static void __init apic_intr_init(void)
 203{
 204        smp_intr_init();
 205
 206#ifdef CONFIG_X86_THERMAL_VECTOR
 207        alloc_intr_gate(THERMAL_APIC_VECTOR, thermal_interrupt);
 208#endif
 209#ifdef CONFIG_X86_MCE_THRESHOLD
 210        alloc_intr_gate(THRESHOLD_APIC_VECTOR, threshold_interrupt);
 211#endif
 212#if defined(CONFIG_X86_MCE) && defined(CONFIG_X86_LOCAL_APIC)
 213        alloc_intr_gate(MCE_SELF_VECTOR, mce_self_interrupt);
 214#endif
 215
 216#if defined(CONFIG_X86_64) || defined(CONFIG_X86_LOCAL_APIC)
 217        /* self generated IPI for local APIC timer */
 218        alloc_intr_gate(LOCAL_TIMER_VECTOR, apic_timer_interrupt);
 219
 220        /* IPI for X86 platform specific use */
 221        alloc_intr_gate(X86_PLATFORM_IPI_VECTOR, x86_platform_ipi);
 222
 223        /* IPI vectors for APIC spurious and error interrupts */
 224        alloc_intr_gate(SPURIOUS_APIC_VECTOR, spurious_interrupt);
 225        alloc_intr_gate(ERROR_APIC_VECTOR, error_interrupt);
 226
 227        /* Performance monitoring interrupts: */
```

```
228# ifdef CONFIG_PERF_EVENTS
229             alloc_intr_gate(LOCAL_PENDING_VECTOR, perf_pending_interrupt);
230# endif
231
232#endif
233}
```

smp_intr_init()函数，32 位 x86 体系中是一个空函数。后面 206～229 设置中断描述符表中 APIC 相关的中断服务层序，也就是把位于 32~255 之间，除去系统调用外其他中断向量的中断处理程序设置为 interrupt[i]。interrupt 共有 NR_VECTORS-FIRST_EXTERNAL_VECTOR 个表项：

#define NR_VECTORS 256

#define FIRST_EXTERNAL_VECTOR 0x20 //32

并且 interrupt 数组在 arch/x86/kernel/entry_32.S 中定义并初始化：

```
823.section .init.rodata,"a"
824ENTRY(interrupt)
825.text
826             .p2align 5
827             .p2align CONFIG_X86_L1_CACHE_SHIFT
828ENTRY(irq_entries_start)
829             RING0_INT_FRAME
830vector=FIRST_EXTERNAL_VECTOR
831.rept (NR_VECTORS-FIRST_EXTERNAL_VECTOR+6)/7
832             .balign 32
833    .rept 7
834        .if vector < NR_VECTORS
835            .if vector <> FIRST_EXTERNAL_VECTOR
836                CFI_ADJUST_CFA_OFFSET -4
837            .endif
8381:            pushl $(~vector+0x80)      /* Note: always in signed byte range */
839                CFI_ADJUST_CFA_OFFSET 4
840            .if ((vector-FIRST_EXTERNAL_VECTOR)%7) <> 6
841                jmp 2f
842            .endif
843            .previous
844                .long 1b
845            .text
846vector=vector+1
847        .endif
848    .endr
8492:            jmp common_interrupt
850.endr
851END(irq_entries_start)
```

```
852
853.previous
854END(interrupt)
```

看到上面的代码，从 824 到 854 行都属于数据段中，看到 828~851 之间的代码，虽然表面上看上去它们是分隔开的，但是实际上编译之后会在数据段中一块连续的区域中。我们看到从 831 行~850 行，执行循环 NR_VECTORS-FIRST_EXTERNAL_VECTOR+6 次，也就是说是 256-32+6=230 次，那么代码展开后，这段数据的内容就类似于：

```
ENTRY(interrupt)
        .long 1b
        .long 1b
        .long 1b
……
```

即构成 225 个项的 interrupt[]数组，每个元素占 64 位，也就是 8 个字节。我们看到每个元素都是 1b，即上面代码 1 标号后的代码片段：

```
pushl $(~vector+0x80)
CFI_ADJUST_CFA_OFFSET 4
.if ((vector-FIRST_EXTERNAL_VECTOR)%7) <> 6
         jmp 2f
.endif
```

因此，都会执行代码 2 标号的那个 common_interrupt，其参数为 push 进去的~vector+0x80。举个例子，比如中断向量 32，即 IRQ0，对应的中断处理程序 interrupt[0]是

```
pushl $(~0 + 0x80)
CFI_ADJUST_CFA_OFFSET 4
jmp common_interrupt
```

至于 common_interrupt 随后如何执行，具体的细节请查阅博客“中断处理（续）”http://blog.csdn.net/yunsongice/archive/2010/03/07/5353896.aspx。

那么在 native_init_IRQ 设置好了中断描述符表的 32~256 项之后，也就是 interrupt[]的 0 到 224 元素作为其处理函数之后，我们就可以设置它了。所以看到随后 255、256 行，在没有配置 CONFIG_ACPI，也就是说，没有启动 ACPI 控制器的情况下，调用老式的 setup_irq 函数把 2 号 IRQ 设置成一个空函数。263、264 行，调用 setup_irq 函数把 13 号 IRQ 设置成 FPU 容错处理函数：

```
int setup_irq(unsigned int irq, struct irqaction *act)
{
      struct irq_desc *desc = irq_to_desc(irq);
      return __setup_irq(irq, desc, act);
}
```

__setup_irq 本质上就是通过 irq 号得到对应的 irq_desc 描述符。然后调用__setup_irq()函数初始化它和 irqaction，并把这个描述符插入到合适的 IRQ 链表。具体的代码内容比较多，我就不一一分析了，对上述过程还想深入研究得可以去浏览一下博客“中断处理（续）”

http://blog.csdn.net/yunsongice/archive/2010/03/07/5353896.aspx。

最后，由于我们没有配置 CONFIG_4KSTACKS 编译选项，所以 native_init_IRQ 的最后一个函数 irq_ctx_init 是空函数，那么 native_init_IRQ 结束后，整个 init_IRQ 函数就结束了，APIC 和各本地 PIC 的相关的中断服务就算初始化完毕了。此时，内核全局变量 irq_desc[]数组的 16 个元素的 chip 字段都是 i8259A_chip；handle_irq 字段是 handle_level_irq；name 字段为“XT”；同时，interrupt[]数组的每一个元素作为中断描述符表表项 32~255 的服务程序，且仅有 interrupt[2]和 interrupt[13]分别被初始化成了一个空的处理函数 no_action 和一个 FPU 容错函数 math_error_irq，其余服务程序都还没有被初始化。

5.10.2 初始化本地软时钟

native_init_IRQ 结束后，init_IRQ 也就结束了，回到 start_kernel 中，607 行，prio_tree_init 函数很简单：

```
void __init prio_tree_init(void)
{
    unsigned int i;

    for (i = 0; i < ARRAY_SIZE(index_bits_to_maxindex) - 1; i++)
        index_bits_to_maxindex[i] = (1UL << (i + 1)) - 1;
    index_bits_to_maxindex[ARRAY_SIZE(index_bits_to_maxindex) - 1] = ~0UL;
}
```

初始化全局变量 index_bits_to_maxindex[]数组，不在话下。继续走，608 行，init_timers，来自 kernel/timer.c：

```
1736void __init init_timers(void)
1737{
1738        int err = timer_cpu_notify(&timers_nb, (unsigned long)CPU_UP_PREPARE,
1739                                (void *)(long)smp_processor_id());
1740
1741        init_timer_stats();
1742
1743        BUG_ON(err != NOTIFY_OK);
1744        register_cpu_notifier(&timers_nb);
1745        open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
1746}
```

init_timers 函数首先 1738 初始化本地 CPU 上的软时钟相关的数据结构，该结构是一个 notifier_block 类型全局变量，定义在 kernel/time.c：

```
static struct notifier_block __cpuinitdata timers_nb = {
    .notifier_call   = timer_cpu_notify,
```

};

timers_nb 的回调函数 timer_cpu_notify 用于初始化指定 CPU 上的软时钟相关的数据结构：

```
static int __cpuinit timer_cpu_notify(struct notifier_block *self,
                        unsigned long action, void *hcpu)
{
        long cpu = (long)hcpu;
        switch(action) {
        case CPU_UP_PREPARE:
        case CPU_UP_PREPARE_FROZEN:
                if (init_timers_cpu(cpu) < 0)
                        return NOTIFY_BAD;
                break;
#ifdef CONFIG_HOTPLUG_CPU
        case CPU_DEAD:
        case CPU_DEAD_FROZEN:
                migrate_timers(cpu);
                break;
#endif
        default:
                break;
        }
        return NOTIFY_OK;
}
```

我们指定的是 CPU_UP_PREPARE，所以肯定执行 init_timers_cpu(cpu)：

```
1513static int __cpuinit init_timers_cpu(int cpu)
1514{
1515             int j;
1516             struct tvec_base *base;
1517             static char __cpuinitdata tvec_base_done[NR_CPUS];
1518
1519             if (!tvec_base_done[cpu]) {
1520                          static char boot_done;
1521
1522                          if (boot_done) {
1523                                       /*
1524                                        * The APs use this path later in boot
1525                                        */
1526                                       base = kmalloc_node(sizeof(*base),
1527                                                                    GFP_KERNEL | __GFP_ZERO,
```

```
1528                                                    cpu_to_node(cpu));
1529                    if (!base)
1530                            return -ENOMEM;
1531
1532                    /* Make sure that tvec_base is 2 byte aligned */
1533                    if (tbase_get_deferrable(base)) {
1534                            WARN_ON(1);
1535                            kfree(base);
1536                            return -ENOMEM;
1537                    }
1538                    per_cpu(tvec_bases, cpu) = base;
1539            } else {
1540                    /*
1541                      * This is for the boot CPU - we use compile-time
1542                      * static initialisation because per-cpu memory isn't
1543                      * ready yet and because the memory allocators are not
1544                      * initialised either.
1545                      */
1546                    boot_done = 1;
1547                    base = &boot_tvec_bases;
1548            }
1549            tvec_base_done[cpu] = 1;
1550    } else {
1551            base = per_cpu(tvec_bases, cpu);
1552    }
1553
1554    spin_lock_init(&base->lock);
1555
1556    for (j = 0; j < TVN_SIZE; j++) {
1557            INIT_LIST_HEAD(base->tv5.vec + j);
1558            INIT_LIST_HEAD(base->tv4.vec + j);
1559            INIT_LIST_HEAD(base->tv3.vec + j);
1560            INIT_LIST_HEAD(base->tv2.vec + j);
1561    }
1562    for (j = 0; j < TVR_SIZE; j++)
1563            INIT_LIST_HEAD(base->tv1.vec + j);
1564
1565    base->timer_jiffies = jiffies;
1566    base->next_timer = base->timer_jiffies;
1567    return 0;
1568}
```

1517 定义个全局变量 tvec_base_done[]数组，每个元素对应一个 CPU 的软时钟初始化状态。随后 1519~1552 行创建本地软时钟数据结构 tvec_base。该结构定义如下：

```
struct tvec_base {
	spinlock_t lock;
	struct timer_list *running_timer;
	unsigned long timer_jiffies;
	unsigned long next_timer;
	struct tvec_root tv1;
	struct tvec tv2;
	struct tvec tv3;
	struct tvec tv4;
	struct tvec tv5;
} ____cacheline_aligned;
```

然后 1556~1566 行初始化这个 tvec_base 结构。我们看到，最重要的是 1565 行，tvec_base 结构的 timer_jiffies 字段被设置成了超级重要的 jiffies 宏，即自系统启动以来产生的节拍的总数，通过如下函数获得（本质上是一个汇编指令 syscall）：

```
# define jiffies raid6_jiffies()
static inline uint32_t raid6_jiffies(void)
{
	struct timeval tv;
	gettimeofday(&tv, NULL);
	return tv.tv_sec*1000 + tv.tv_usec/1000;
}
static __always_inline int gettimeofday(struct timeval *tv, struct timezone *tz)
{
	int ret;
	asm volatile("syscall"
		: "=a" (ret)
		: "0" (__NR_gettimeofday),"D" (tv),"S" (tz)
		: __syscall_clobber );
	return ret;
}
```

回到 init_timers，当初始化了本地 CPU 上的软时钟数据结构之后，1741 行，调用 init_timer_stats，初始化每 CPU 变量 tstats_lookup_lock 作为自旋锁。然后 1744 行调用我们已经见过的 register_cpu_notifier 函数，将新建的这个 timers_nb 结构挂到全局 cpu_chain 链中，作为通知链注册。

最后，1655 行，调用 open_softirq 初始化时钟的软中断处理函数：

```
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
	softirq_vec[nr].action = action;
}
```

软中断的概念如果还不熟悉的，请参考博客“ 下半部分 ”

http://blog.csdn.net/yunsongice/archive/2010/03/07/5354011.aspx

5.10.3 软中断初始化

open_softirq 结束后，init_timers 就结束了，整个内核就可以享受时钟服务了，接下来 start_kernel 的 609 行调用 hrtimers_init。由于我们没有配置 CONFIG_HIGH_RES_TIMERS，所以这个函数仅仅是把全局 notifier_block 变量 hrtimer_cpu_notify 加入通知链，供将来的内核各模块使用。

然后，start_kernel 的 610 行调用 softirq_init 来初始化整个软中断系统：

```
674void __init softirq_init(void)
675{
676             int cpu;
677
678             for_each_possible_cpu(cpu) {
679                         int i;
680
681                         per_cpu(tasklet_vec, cpu).tail =
682                                     &per_cpu(tasklet_vec, cpu).head;
683                         per_cpu(tasklet_hi_vec, cpu).tail =
684                                     &per_cpu(tasklet_hi_vec, cpu).head;
685                         for (i = 0; i < NR_SOFTIRQS; i++)
686                                     INIT_LIST_HEAD(&per_cpu(softirq_work_list[i], cpu));
687             }
688
689             register_hotcpu_notifier(&remote_softirq_cpu_notifier);
690
691             open_softirq(TASKLET_SOFTIRQ, tasklet_action);
692             open_softirq(HI_SOFTIRQ, tasklet_hi_action);
693}
```

我们看到 681~686 行，初始化每个 CPU 的 tasklet_vec、tasklet_hi_vec 和 softirq_work_list[]结构。689 行，如果定义了 CONFIG_HOTPLUG_CPU 配置选项，就将全局 notifier_block 类型变量 remote_softirq_cpu_notifier 注册到通知链中。

随后 691、692 行将 TASKLET_SOFTIRQ 和 HI_SOFTIRQ 对应的软中断打开，这样 I/O 驱动程序中实现可延迟函数的首选方法 tasklet 就可以使用了，对这方面感兴趣的同学仍然请访问博客“ 下半部分 ”
http://blog.csdn.net/yunsongice/archive/2010/03/07/5354011.aspx

start_kernel 的 611 行 timekeeping_init 函数：

```
534void __init timekeeping_init(void)
535{
536             struct clocksource *clock;
537             unsigned long flags;
538             struct timespec now, boot;
539
540             read_persistent_clock(&now);
541             read_boot_clock(&boot);
542
543             write_seqlock_irqsave(&xtime_lock, flags);
544
545             ntp_init();
546
547             clock = clocksource_default_clock();
548             if (clock->enable)
549                          clock->enable(clock);
550             timekeeper_setup_internals(clock);
551
552             xtime.tv_sec = now.tv_sec;
553             xtime.tv_nsec = now.tv_nsec;
554             raw_time.tv_sec = 0;
555             raw_time.tv_nsec = 0;
556             if (boot.tv_sec == 0 && boot.tv_nsec == 0) {
557                          boot.tv_sec = xtime.tv_sec;
558                          boot.tv_nsec = xtime.tv_nsec;
559             }
560             set_normalized_timespec(&wall_to_monotonic,
561                                                  -boot.tv_sec, -boot.tv_nsec);
562             update_xtime_cache(0);
563             total_sleep_time.tv_sec = 0;
564             total_sleep_time.tv_nsec = 0;
565             write_sequnlock_irqrestore(&xtime_lock, flags);
566}
```

这个函数主要是通过读取 CMOS 上的时钟来初始化全局时间变量 xtime、raw_time 以及 total_sleep_time。具体的就不去分析了，主要讲讲读取 CMOS 数据的方法，来看函数 read_persistent_clock：

```
void read_persistent_clock(struct timespec *ts)
{
        unsigned long retval, flags;

        spin_lock_irqsave(&rtc_lock, flags);
        retval = x86_platform.get_wallclock();
        spin_unlock_irqrestore(&rtc_lock, flags);
```

```
	ts->tv_sec = retval;
	ts->tv_nsec = 0;
}
```

而全局变量 x86_platform 在编译时候被初始化如下：

```
struct x86_platform_ops x86_platform = {
	.calibrate_tsc			= native_calibrate_tsc,
	.get_wallclock			= mach_get_cmos_time,
	.set_wallclock			= mach_set_rtc_mmss,
	.iommu_shutdown			= iommu_shutdown_noop,
	.is_untracked_pat_range		= is_ISA_range,
	.nmi_init			= default_nmi_init
};
```

所以通过的是 mach_get_cmos_time 函数来读取 CMOS 中的时间数据。这个函数本质上是通过 CMOS_READ 宏来进行，比如 CMOS_READ(RTC_SECONDS)

Linux 只用 RTC 来获取时间和日期，内核通过 0x70 和 Ox71 I/O 端口访问 RTC。系统管理员通过执行 Unix 系统时钟程序（直接作用于这两个 I/O 端口）可以设置时钟。MC146818 RTC 芯片（或其他兼容芯片，如 DS12887）可以在 IRQ8 上产生周期性的中断，中断的频率在 2HZ～8192HZ 之间。与 MC146818 RTC 对应的设备驱动程序实现在 include/linux/mc146818rtc.h 和 drivers/char/mc146818rtc.c 文件中，而对应的设备文件是/dev/mc146818rtc （major=10，minor=135，只读字符设备）。因此用户进程可以通过对她进行编程以使得当 RTC 到达某个特定的时间值时激活 IRQ8 线，从而将 RTC 当作一个闹钟来用。

所以，宏 RTC_SECONDS 来自文件 include/linux/mc146818rtc.h

```
#define RTC_SECONDS		0
#define CMOS_READ(addr) rtc_cmos_read(addr)
#define RTC_PORT(x)   (0x70 + (x))
unsigned char rtc_cmos_read(unsigned char addr)
{
	unsigned char val;

	lock_cmos_prefix(addr);
	outb(addr, RTC_PORT(0));
	val = inb(RTC_PORT(1));
	lock_cmos_suffix(addr);

	return val;
}
```

5.10.4 初始化定时器中断

回到 start_kernel，612 行 time_init 函数：

```
void __init time_init(void)
{
    late_time_init = x86_late_time_init;
}
```

函数 x86_late_time_init 实际上是初始化 tsc 时钟源。在 time_init 中只是把该函数的地址赋给全局变量 late_time_init，以后某个时刻肯定会调用它的，这里先提前详细分析一下他：

```
static __init void x86_late_time_init(void)
{
    x86_init.timers.timer_init();
    tsc_init();
}
```

x86_init.timers.timer_init 实际是函数 hpet_time_init，来自文件 kernel/timer.c：

```
101 void __init hpet_time_init(void)
102 {
103         if (!hpet_enable())
104                 setup_pit_timer();
105         setup_default_timer_irq();
106}
```

由于我们没有配置 CONFIG_HPET_TIMER，所以 103 行 hpet_enable 返回 0，调用 104 行的函数 setup_pit_timer，位于 arch/x86/kernel/i8253.c：

```
104void __init setup_pit_timer(void)
105{
106             /*
107              * Start pit with the boot cpu mask and make it global after the
108              * IO_APIC has been initialized.
109              */
110             pit_ce.cpumask = cpumask_of(smp_processor_id());
111             pit_ce.mult = div_sc(CLOCK_TICK_RATE, NSEC_PER_SEC, pit_ce.shift);
112             pit_ce.max_delta_ns = clockevent_delta2ns(0x7FFF, &pit_ce);
113             pit_ce.min_delta_ns = clockevent_delta2ns(0xF, &pit_ce);
114
115             clockevents_register_device(&pit_ce);
116             global_clock_event = &pit_ce;
117}
```

整个函数的 115 行之前都是在通过汇编指令初始化全局变量 pit_ce，这个 clock_event_device 类型的变量在编译的时候已经被初始化成了：

```
static struct clock_event_device pit_ce = {
	.name		= "pit",
	.features	= CLOCK_EVT_FEAT_PERIODIC | CLOCK_EVT_FEAT_ONESHOT,
	.set_mode	= init_pit_timer,
	.set_next_event = pit_next_event,
	.shift		= 32,
	.irq		= 0,
};
```

我们看到它的 irq 号为 0，也就是对应 32 号中断描述符表项，即 interrupt[0]处理函数。115 行把它注册到通知链和 clockevent_devices 链表中：

```
void clockevents_register_device(struct clock_event_device *dev)
{
	unsigned long flags;
……//一些调试代码
	raw_spin_lock_irqsave(&clockevents_lock, flags);
	list_add(&dev->list, &clockevent_devices);
	clockevents_do_notify(CLOCK_EVT_NOTIFY_ADD, dev);
	clockevents_notify_released();
	raw_spin_unlock_irqrestore(&clockevents_lock, flags);
}
```

最后将全局 global_clock_event 设置成这个定时器，然后回到 hpet_time_init 中，105 行，调用 setup_default_timer_irq()：

```
void __init setup_default_timer_irq(void)
{
	setup_irq(0, &irq0);
}
static struct irqaction irq0	= {
	.handler = timer_interrupt,
	.flags = IRQF_DISABLED | IRQF_NOBALANCING | IRQF_IRQPOLL | IRQF_TIMER,
	.name = "timer"
};
```

很简单，调用 setup_irq 把 interrupt[0]对应的处理函数设置成 timer_interrupt。至于随后如何处理的，请查看博客“ 定时器中断 ”
http://blog.csdn.net/yunsongice/archive/2010/03/07/5354137.aspx

5.11 走进 start_kernel 尾声

中断体系建立起来后，虽然后面还有很多行代码，但是都是些比较好理解的初始化函数了，也就是说 start_kernel 进入尾声了。

5.11.1 初始化 slab 的后续工作

继续分析 start_kenel 的下一个函数，613 行，profile_init 函数，用于对系统剖析做相关初始化，系统剖析用于系统调用：

```
int __ref profile_init(void)
{
	int buffer_bytes;
	if (!prof_on)
		return 0;

	/* only text is profiled */
	prof_len = (_etext - _stext) >> prof_shift;
	buffer_bytes = prof_len*sizeof(atomic_t);

	if (!alloc_cpumask_var(&prof_cpu_mask, GFP_KERNEL))
		return -ENOMEM;

	cpumask_copy(prof_cpu_mask, cpu_possible_mask);

	prof_buffer = kzalloc(buffer_bytes, GFP_KERNEL|__GFP_NOWARN);
	if (prof_buffer)
		return 0;

	prof_buffer = alloc_pages_exact(buffer_bytes,
					GFP_KERNEL|__GFP_ZERO|__GFP_NOWARN);
	if (prof_buffer)
		return 0;

	prof_buffer = vmalloc(buffer_bytes);
	if (prof_buffer) {
		memset(prof_buffer, 0, buffer_bytes);
		return 0;
	}

	free_cpumask_var(prof_cpu_mask);
	return -ENOMEM;
}
```

函数无非就是初始化一些简单的全局变量。继续在 start_kenel 中前进，614~616 行，根据 irqs_disabled 是否返回成功而打印一些信息。617 行，由于 CONFIG_PROVE_LOCKING 没有被配置，所以 early_boot_irqs_on 是一个空函数。618 行，local_irq_enable 函数将打开可屏蔽中断，与 549 行的 local_irq_disable 遥相呼应。621 行设置全局 GFP 常量 gfp_allowed_mask，注释上写得很清楚，中断处理模块已经成型，所以可以进行 GFP 分配了。继续走，623 行，调用 kmem_cache_init_late 函数。我们在“ 初始化内存管理 ”讲解的那个 mm_init()中曾经调用过一个 kmem_cache_init，用于初始化内核 slab 分配体系，还记得吧。那么这个加了_late 后缀的函数是啥意思呢，它也来自 mm/slab.c：

```
void __init kmem_cache_init_late(void)
{
	struct kmem_cache *cachep;

	/* 6) resize the head arrays to their final sizes */
	mutex_lock(&cache_chain_mutex);
	list_for_each_entry(cachep, &cache_chain, next)
		if (enable_cpucache(cachep, GFP_NOWAIT))
			BUG();
	mutex_unlock(&cache_chain_mutex);

	/* Done! */
	g_cpucache_up = FULL;

	/* Annotate slab for lockdep -- annotate the malloc caches */
	init_lock_keys();

	/*
	 * Register a cpu startup notifier callback that initializes
	 * cpu_cache_get for all new cpus
	 */
	register_cpu_notifier(&cpucache_notifier);

	/*
	 * The reap timers are started later, with a module init call: That part
	 * of the kernel is not yet operational.
	 */
}
```

很简单，就是做几个 slab 分配器初始化的后续工作，其实就只做一件事，遍历 cache_chain 链表，把里面的所有作为 slab 缓存头的 kmem_cache 结构通过 enable_cpucache 函数重新计算他们的 batchcount、limit 和 shared 字段，并为 NUMA 体系中那些还没有初始化 nodelists[node]的节点进行初始化工作。最后调用 register_cpu_notifier 函数，将全局变量 cpucache_notifier 挂到全局 cpu_chain 链中，具体的代码我就不去分析了。

5.11.2 启动 console

回到 start_kenel 函数中，下一个函数 630 行的 console_init()，用于初始化系统控制台结构。该函数执行后可调用 printk()函数将 log_buf 中符合打印级别要求的系统信息打印到控制台上。还记得我们在讲 printk 函数的时候说过，这个函数是把字符串写到 log_buf 中，而启动了系统控制台后，这个 log_buf 中的信息就会显示在屏幕上方了。

```
void __init console_init(void)
{
	initcall_t *call;

	/* Setup the default TTY line discipline. */
	tty_ldisc_begin();

	/*
	 * set up the console device so that later boot sequences can
	 * inform about problems etc..
	 */
	call = __con_initcall_start;
	while (call < __con_initcall_end) {
		(*call)();
		call++;
	}
}
```

函数虽然简单，但涉及到的东西很多。首先看到 tty_ldisc_begin 函数：

```
void tty_ldisc_begin(void)
{
	/* Setup the default TTY line discipline. */
	(void) tty_register_ldisc(N_TTY, &tty_ldisc_N_TTY);
}
```

tty_register_ldisc(N_TTY, &tty_ldisc_N_TTY）这段代码主要的用途就是注册 tty 线路规程的，大家研究 tty 的驱动就会发现，在用户和硬件之间的 tty 驱动是分了三层的，其中最底层是 tty 驱动程序了，主要负责从硬件接受数据，和格式化上层发下来的数据后给硬件。

在驱动程序之上就是线路规程，他负责把从 tty 核心层或者 tty 驱动层接受的数据进行特殊的按着某个协议的格式化，就像是 ppp 或者蓝牙协议，然后在分发出去的。

在 tty 线路规程之上就是 tty 核心层。大家可参考 ldd3 学习一下。那么，如何初始化终端呢？这又要从编译说起了，看看我的 vmlinux.lds.S 中连接脚本汇编中有这段代码：

```
__con_initcall_start = .;
*(.con_initcall.init)
__con_initcall_end = .;
```

原来的 call = __con_initcall_start 就是把__con_initcall_start 的虚拟地址给 call，去执行在__con_initcall_start = .;和__con_initcall_end = .;之间的 con_initcall.init。这就应该是 linux 惯用做法，把某个实际初始化函数的指针数据是放到了 con_initcall.init 段中，那么是怎么放的呢？

在 linux/init.h 里面有这么一句宏定义：

```
#define console_initcall(fn) \
        static initcall_t __initcall_##fn \
        __used __section(.con_initcall.init) = fn
```

在我的 Linux2.6.34.1 的所有驱动程序里面，大约有 48 个文件调用了这个 console_initcall 宏，那么到底用的是哪个驱动程序呢？这得看看配置文件了，找到了 CONFIG_SERIAL_8250，那么肯定是调用的 drivers/serial/8250.c 中的文件代码：

```
console_initcall(serial8250_console_init);
```

这回就看出来了，通过#define console_initcall(fn) \宏，调用的是 serial8250_console_init 函数。如果对 tty 的驱动程序感兴趣的同学可以好好分析一下这个代码，涉及到与显示器相连接的串口终端 8250 的驱动，很典型的一个串口终端驱动程序。

5.11.3 一些简单的函数

start_kenel 的下一个函数 632 行的 panic 函数。这个函数是绝对不会被执行的，因为它是当系统发现无法继续运行下去的故障时才调用它，会导致系统中止，然后由系统显示错误号。这里是当全局变量 panic_later 被设置是，才会调用。内核的 panic 函数位于 kernel/panic.c 文件中。

接下来，由于我们没有 CONFIG_LOCKDEP，所以 634 汗的 lockdep_info 宏是个空宏。因为没有配置 CONFIG_DEBUG_LOCKING_API_SELFTESTS，641 行 locking_selftest()也是个空函数。643～652 的代码根据全局变量 initrd_start 的值打印一些信息。

继续走，由于我们没有配置 CONFIG_CGROUP_MEM_RES_CTLR，653 行的 page_cgroup_init 是一个空函数。654 行，没有配置 CONFIG_DEBUG_PAGEALLOC，debug_pagealloc_enabled 也是个空函数。655 行的 kmemtrace_init 是个待开发函数，

656 行 kmemleak_init 函数是有效取决于 CONFIG_DEBUG_KMEMLEAK 编译选项，我在.config 文件中没找到，所以忽略。

657 行 debug_objects_mem_init 函数是否有用取决于 CONFIG_DEBUG_OBJECTS 编译选项，我们看到，仍然是“ is not set ”，所以不去管它。

658 行，idr_init_cache 函数，很简单，来自 lib/idr.c：

```
void __init idr_init_cache(void)
{
	idr_layer_cache = kmem_cache_create("idr_layer_cache",
				sizeof(struct idr_layer), 0, SLAB_PANIC, NULL);
}
```

创建一个 idr_layer 结构的 slab 缓存，其头部由全局变量 idr_layer_cache 指向。

659 行，setup_per_cpu_pageset：

```
void __init setup_per_cpu_pageset(void)
{
	struct zone *zone;
	int cpu;

	for_each_populated_zone(zone) {
		zone->pageset = alloc_percpu(struct per_cpu_pageset);
		for_each_possible_cpu(cpu) {
			struct per_cpu_pageset *pcp = per_cpu_ptr(zone->pageset, cpu);
			setup_pageset(pcp, zone_batchsize(zone));
			if (percpu_pagelist_fraction)
				setup_pagelist_highmark(pcp,
					(zone->present_pages /
						percpu_pagelist_fraction));
		}
	}
}
```

函数很简单，为每个 NUMA 节点的 zone 结构的 per_cpu_pageset 字段分配空间，前提是如果那个 zone 的 present_pages 不为 0。

继续走，660 行，numa_policy_init 函数，初始化 NUMA 策略：

```
/* assumes fs == KERNEL_DS */
void __init numa_policy_init(void)
{
	nodemask_t interleave_nodes;
	unsigned long largest = 0;
	int nid, prefer = 0;

	policy_cache = kmem_cache_create("numa_policy",
```

```
                                sizeof(struct mempolicy),
                                0, SLAB_PANIC, NULL);

        sn_cache = kmem_cache_create("shared_policy_node",
                                sizeof(struct sp_node),
                                0, SLAB_PANIC, NULL);

        /*
         * Set interleaving policy for system init. Interleaving is only
         * enabled across suitably sized nodes (default is >= 16MB), or
         * fall back to the largest node if they're all smaller.
         */
        nodes_clear(interleave_nodes);
        for_each_node_state(nid, N_HIGH_MEMORY) {
                unsigned long total_pages = node_present_pages(nid);

                /* Preserve the largest node */
                if (largest < total_pages) {
                        largest = total_pages;
                        prefer = nid;
                }

                /* Interleave this node? */
                if ((total_pages << PAGE_SHIFT) >= (16 << 20))
                        node_set(nid, interleave_nodes);
        }

        /* All too small, use the largest */
        if (unlikely(nodes_empty(interleave_nodes)))
                node_set(prefer, interleave_nodes);

        if (do_set_mempolicy(MPOL_INTERLEAVE, 0, &interleave_nodes))
                printk("numa_policy_init: interleaving failed\n");
}
```

函数很简单，通过 node_set 宏为每个节点的 nodemask_t->bits 进行设置，以满足相应的策略。回到 start_kenel 中，661、662 行，来了，执行 late_time_init 函数。这个函数就是我们刚才在“初始化定时器中断”中分析过的 x86_late_time_init。内核为啥要设计成这幅颜色，还得请高人们多多指点。

663 行，sched_clock_init，来自 kernel/sched_clock.c：

```
void sched_clock_init(void)
{
```

```
    u64 ktime_now = ktime_to_ns(ktime_get());
    int cpu;

    for_each_possible_cpu(cpu) {
        struct sched_clock_data *scd = cpu_sdc(cpu);

        scd->tick_raw = 0;
        scd->tick_gtod = ktime_now;
        scd->clock = ktime_now;
    }

    sched_clock_running = 1;
}
```

这段代码牵涉到一个每 CPU 变量 sched_clock_data，该函数很简单，就是把每个 CPU 的每 CPU 变量 sched_clock_data 初始化成 ktime_now。

5.11.4 校准 CPU 时钟速度

回到 start_kernel 中，在 start_kernel 的尾声有两个重要的函数，664 行的 calibrate_delay 就是其中一个，来自 init/calibrate.c：

```
122void __cpuinit calibrate_delay(void)
123{
124         unsigned long ticks, loopbit;
125         int lps_precision = LPS_PREC;
126         static bool printed;
127
128         if (preset_lpj) {
129                 loops_per_jiffy = preset_lpj;
130                 if (!printed)
131                         pr_info("Calibrating delay loop (skipped) "
132                                 "preset value.. ");
133         } else if ((!printed) && lpj_fine) {
134                 loops_per_jiffy = lpj_fine;
135                 pr_info("Calibrating delay loop (skipped), "
136                         "value calculated using timer frequency.. ");
137         } else if ((loops_per_jiffy = calibrate_delay_direct()) != 0) {
138                 if (!printed)
139                         pr_info("Calibrating delay using timer "
140                                 "specific routine.. ");
141         } else {
```

```
142                     loops_per_jiffy = (1<<12);
143
144                     if (!printed)
145                             pr_info("Calibrating delay loop... ");
146                     while ((loops_per_jiffy <<= 1) != 0) {
147                             /* wait for "start of" clock tick */
148                             ticks = jiffies;
149                             while (ticks == jiffies)
150                                     /* nothing */;
151                             /* Go .. */
152                             ticks = jiffies;
153                             __delay(loops_per_jiffy);
154                             ticks = jiffies - ticks;
155                             if (ticks)
156                                     break;
157                     }
158
159                     /*
160                       * Do a binary approximation to get loops_per_jiffy set to
161                       * equal one clock (up to lps_precision bits)
162                       */
163                     loops_per_jiffy >>= 1;
164                     loopbit = loops_per_jiffy;
165                     while (lps_precision-- && (loopbit >>= 1)) {
166                             loops_per_jiffy |= loopbit;
167                             ticks = jiffies;
168                             while (ticks == jiffies)
169                                     /* nothing */;
170                             ticks = jiffies;
171                             __delay(loops_per_jiffy);
172                             if (jiffies != ticks)   /* longer than 1 tick */
173                                     loops_per_jiffy &= ~loopbit;
174                     }
175             }
176             if (!printed)
177                     pr_cont("%lu.%02lu BogoMIPS (lpj=%lu)\n",
178                             loops_per_jiffy/(500000/HZ),
179                             (loops_per_jiffy/(5000/HZ)) % 100, loops_per_jiffy);
180
181             printed = true;
182}
```

要看懂上面的代码，需要了解一些背景知识。首先，BogoMIPS 值是什么意思？Bogo 就是 Bogus(伪)的意思，MIPS 是 millions of instructions per second(百万条指令每秒)的缩写。这样

我们就知道了其实这个函数是 linux 内核中一个 cpu 性能测试函数。由于内核对这个数值的精度要求不高，所以内核使用了一个十分简单而有效的算法用于得到这个值。

这个值虽然不准确，但也足以令我们心动。如果你想了解自己机器的 BogoMIPS，你可以察看/proc/cpuinfo 文件中的 bogomips 一行。在你知道了自己 cpu 的 BogoMIPS 之后，如果你觉得不过瘾，那么让我们一起来看看 calibrate_delay 函数是怎么完成工作的。

首先，125 行定义计算 BogoMIPS 的精度，这个值越大，则计算出的 BogoMIPS 越精确。我们看到 LPS_PREC 宏被定义成了 8：

```
#define LPS_PREC 8
```

看到 129 行，loops_per_jiffy 为每个时钟滴答执行一个极短的循环的次数。这个全局变量的值在我们配置了 ARCH_HAS_READ_CURRENT_TIMER 编译选项的情况下，由函数 calibrate_delay_direct 得出。很遗憾，我们并没有配置它，所以 142 行，loops_per_jiffy 就等于 1<<12，即 1 后面 12 个“ 0 ”。

第 146 至 157 行，是第一次计算 loops_per_jiffy 的值，这次计算只是一个粗略的计算，为下面的计算打好基础。

可以想象我们要计算 loops_per_jiffy 的值，可以在一个滴答的开始时，立即重复执行一个极短的循环，当一个滴答结束时，这个循环执行了多少次就是我们要求的初步的值。系统用 jiffies 全局变量记录了从系统开始工作到现在为止，所经过的滴答数。它会被内核自动更新——每次定时器中断发生时（每个节拍）它便加 1。这行语句用于记录当前滴答数到 tick 变量中。

注意 149 行这是一个没有循环体得空循环，150 行仅有一个“ ；”号。这条循环语句是通过判断 tick 的值与 jiffies 的值是否不同，来判断 jiffies 是否变化，即是否一个新的滴答开始了。

152~156 行用于判断执行的延时是否超过一个滴答。一般 loops_per_jiffy 的初始值并不大，所以循环会逐步加大 loops_per_jiffy 的值，直到延时超过一个滴答。这里面的延时方法主要是 153 行那个__delay(loops_per_jiffy)计算：

```
void __delay(unsigned long loops)
{
	delay_fn(loops);
}
static void (*delay_fn)(unsigned long) = delay_loop;
static void delay_loop(unsigned long loops)
{
	asm volatile(
		"	test %0,%0	\n"
		"	jz 3f		\n"
		"	jmp 1f		\n"
		".align 16		\n"
		"1:	jmp 2f		\n"
```

```
            ".align 16        \n"
            "2:   dec %0          \n"
            "     jnz 2b          \n"
            "3:   dec %0          \n"
            : /* we don't need output */
            :"a" (loops)
        );
}
```

我们可以看出，前一次 loops_per_jiffy 的值还因太小不合适时，经过一次增大，它提高了两倍，满足了循环条件，跳出循环，而这个值实在是误差太大，所以我们还要经过第二次计算。这里还要注意的是通过上面的分析，我们可以知道更加精确 loops_per_jiffy 的值应该在现在的值与它的一半之间。

所以 163 行，对 loops_per_jiffy 进行折半，这里开始就是第二次计算了。它用折半查找法在我们上面所说的范围内计算出了更精确的 loops_per_jiffy 的值。

164 行定义查找范围的最小值，我把它称为起点。165 行进入循环，将查找范围减半。我们看到 loopbit >>= 1 是在重新定义起点，起点在“ 原起点加 loopbit 减半范围 ”处，即新起点在原先起点与终点的中间。这时我们可以看出 loops_per_jiffy 在“ 新起点 ”与“ 新起点加减半范围（新终点）”之间。

第 167 至 173 行前面一致，都是等待新的滴答，执行延时。如果延时过短，说明 loops_per_jiffy 的值小了，将会跳过这部分，再次进入循环。它将是通过不断的折半方式来增大。如果延时过长，说明 loops_per_jiffy 的值大了，将起点重新返回原起点，当再次进入循环，由于范围减半，故可以达到减小的效果。

从这里我们可以看出它好像是个死循环，所以加入了 lps_precision 变量，来控制循环，即 LPS_PREC 越大，循环次数越多，越精确。可能这些不太好懂，总的说来，它首先将 loops_per_jiffy 的值定为原估算值的 1/2，作为起点值(我这样称呼它)，以估算值为终点值，然后找出起点值到终点值的中间值。用上面相同的方法执行一段时间的延时循环，如果延时超过了一个 tick，说明 loops_per_jiffy 值偏大，则仍以原起点值为起点值，以原中间值为终点值，以起点值和终点值的中间为中间值继续进行查找。如果没有超过一个 tick，说明 loops_per_jiffy 偏小，则以原中间值为起点值，以原终点值为终点值继续查找。

最后 176~179 行计算出 BogoMIPS，并打印，至此，我们的 calibrate_delay()函数就跑完了。

5.11.5 创建一些 slab 缓存

回到 start_kernel 中，下一个函数是 pidmap_init，来自 kernel/pid.c：

```
void __init pidmap_init(void)
{
```

```
    init_pid_ns.pidmap[0].page = kzalloc(PAGE_SIZE, GFP_KERNEL);
    /* Reserve PID 0. We never call free_pidmap(0) */
    set_bit(0, init_pid_ns.pidmap[0].page);
    atomic_dec(&init_pid_ns.pidmap[0].nr_free);

    init_pid_ns.pid_cachep = KMEM_CACHE(pid,
            SLAB_HWCACHE_ALIGN | SLAB_PANIC);
}
```

该函数用于初始化 pid_namespace 结构的全局变量 init_pid_ns，该全局变量在内核编译的时候被初始化为：

```
struct pid_namespace init_pid_ns = {
    .kref = {
        .refcount       = ATOMIC_INIT(2),
    },
    .pidmap = {
        [ 0 ... PIDMAP_ENTRIES-1] = { ATOMIC_INIT(BITS_PER_PAGE), NULL }
    },
    .last_pid = 0,
    .level = 0,
    .child_reaper = &init_task,
};
```

这里为它数组字段 pidmap 的 0 号元素分配一个页面，作为 0 号进程的 pid。这里简单地讲讲为什么要在全局建立一个 pid 的位图。在操作系统中，每一个进程或线程都有一个唯一的编号——pid，这个号码是一个 int 整形字段，在缺省情况下，最大的 pid 号是 32767。
那么，系统在分配 pid 号的时候，就根据这个 pidmap 中的值来判断，取当前系统最大 pid 的值从 pidmap 中寻找下一个空的 pid。如果找不到了，就从 pidmap 表中的 0 号项开始寻找那些被释放了的进程或线程留下来的 pid。只要不是系统同时有 32767 个进程在运行，就总能找到。

当然，在一些超级计算机系统上，3 万多个进程限制显得有些过分，所以系统管理员可以通过往/proc/sys/kernel/pid_max 这个文件中写入一个更小的值来减小 PID 的上限值，使 PID 的上限小于 32767。在 64 位体系结构中，系统管理员可以把 PID 的上限扩大到 4194304。

接着走，666 行，调用 anon_vma_init，来自 mm/rmap.c：

```
void __init anon_vma_init(void)
{
    anon_vma_cachep = kmem_cache_create("anon_vma", sizeof(struct anon_vma),
            0, SLAB_DESTROY_BY_RCU|SLAB_PANIC, anon_vma_ctor);
    anon_vma_chain_cachep = KMEM_CACHE(anon_vma_chain, SLAB_PANIC);
}
```

很简单，分配一个 anon_vma_cachep 作为 anon_vma 的 slab 缓存。这个技术是 PFRA（页框回收算法）技术中的组成部分。这个技术为定位而生——快速的定位指向同一页框的所有页表项。

此技术中，涉及到几个关键的数据结构：anon_vma、vm_area_struct、mm_struct 和 page。就是通过这几个结构中的部分字段，使得线性区彼此之间形成链表，线性区与页表项之间存在联系。首先以 anon_vma 为链表头，形成线性区之间的双向循环链表。此循环链表中的元素就是 vma_area_struct，内核将匿名线性区插入此中。而此结构中有两个字段是用于此链表的：anon_vma、anon_vma_node，前一个指向 anon_vma,后一个指向链表的前一个和后一个元素。

而同时内核将 anon_vma 字段存放在 page 中的 mapping 字段中。而具体运作过程就是当进程引用的页框插入另一个进程的页表项时，内核将第二个进程的匿名线性区插入到 anon_vma 的双向循环链表中。而在 vma_area_sturct 中，还有一个字段：vm_mm 指向对应的内存描述符，而内存描述符中有一个字段：pgd，存放的是页全局目录。通过这样的结构，页表项就可以从匿名页的起始线性地址得到，而该线性地址可以由线性区描述符以及页描述符的 index 字段得到。

继续走，667~670 行，efi_enter_virtual_mode 函数，由于我们没有配置 CONFIG_EFI，所以全局变量为 0，不会去执行它。671 行，thread_info_cache_init 函数，待开发。672 行，执行一个 cred_init 函数，创建一个 cred 结构的 slab 缓存：

```
void __init cred_init(void)
{
	/* allocate a slab in which we can store credentials */
	cred_jar = kmem_cache_create("cred_jar", sizeof(struct cred),
				     0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
}
```

673 行，fork_init 函数，用来根据物理内存大小计算允许创建进程/线程的数量，来自 fork.c：

```
void __init fork_init(unsigned long mempages)
{
#ifndef __HAVE_ARCH_TASK_STRUCT_ALLOCATOR
#ifndef ARCH_MIN_TASKALIGN
#define ARCH_MIN_TASKALIGN	L1_CACHE_BYTES
#endif
	/* create a slab on which task_structs can be allocated */
	task_struct_cachep =
		kmem_cache_create("task_struct", sizeof(struct task_struct),
			ARCH_MIN_TASKALIGN, SLAB_PANIC | SLAB_NOTRACK, NULL);
#endif

	/* do the arch specific task caches init */
```

```
    arch_task_cache_init();

    /*
     * The default maximum number of threads is set to a safe
     * value: the thread structures can take up at most half
     * of memory.
     */
    max_threads = mempages / (8 * THREAD_SIZE / PAGE_SIZE);

    /*
     * we need to allow at least 20 threads to boot a system
     */
    if(max_threads < 20)
        max_threads = 20;

    init_task.signal->rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
    init_task.signal->rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
    init_task.signal->rlim[RLIMIT_SIGPENDING] =
        init_task.signal->rlim[RLIMIT_NPROC];
}
```

注意这里，内核中最重要的 task_struct 结构的 slab 缓存，就是在这个函数中建立的。然后调用 arch_task_cache_init 函数，建立 thread_xstate 的 slab 缓存：

```
void arch_task_cache_init(void)
{
        task_xstate_cachep =
            kmem_cache_create("task_xstate", xstate_size,
                __alignof__(union thread_xstate),
                SLAB_PANIC | SLAB_NOTRACK, NULL);
}
```

最后根据全局变量 totalram_pages，即可用的页面数来计算出允许创建进程/线程的数量，赋给 init_task 的相关字段。

回到 start_kernel 中，下一个函数是 674 行的 proc_caches_init，来自 fork.c：

```
void __init proc_caches_init(void)
{
    sighand_cachep = kmem_cache_create("sighand_cache",
            sizeof(struct sighand_struct), 0,
            SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_DESTROY_BY_RCU|
            SLAB_NOTRACK, sighand_ctor);
    signal_cachep = kmem_cache_create("signal_cache",
            sizeof(struct signal_struct), 0,
            SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_NOTRACK, NULL);
```

```
        files_cachep = kmem_cache_create("files_cache",
                        sizeof(struct files_struct), 0,
                        SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_NOTRACK, NULL);
        fs_cachep = kmem_cache_create("fs_cache",
                        sizeof(struct fs_struct), 0,
                        SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_NOTRACK, NULL);
        mm_cachep = kmem_cache_create("mm_struct",
                        sizeof(struct mm_struct), ARCH_MIN_MMSTRUCT_ALIGN,
                        SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_NOTRACK, NULL);
        vm_area_cachep = KMEM_CACHE(vm_area_struct, SLAB_PANIC);
        mmap_init();
}
```

几个主要的数据结构 sighand_struct、signal_struct、files_struct、fs_struct、mm_struct 和 vm_area_struct 的 slab 缓存的建立。随后调用 mmap_init 初始化每 CPU 计数器，也就是全局 percpu_counter 结构的变量 vm_committed_as。

回到 start_kernel 中，675 行，执行 buffer_init，初始化页高速缓存：

```
void __init buffer_init(void)
{
        int nrpages;

        bh_cachep = kmem_cache_create("buffer_head",
                        sizeof(struct buffer_head), 0,
                                (SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|
                                SLAB_MEM_SPREAD),
                                NULL);

        /*
         * Limit the bh occupancy to 10% of ZONE_NORMAL
         */
        nrpages = (nr_free_buffer_pages() * 10) / 100;
        max_buffer_heads = nrpages * (PAGE_SIZE / sizeof(struct buffer_head));
        hotcpu_notifier(buffer_cpu_notify, 0);
}
```

著名的页高速缓存头 buffer_head 就是在这里被初始化的，系统建立了它的 slab 缓存，然后设置全局变量 max_buffer_heads 作为系统能最大限度使用的高速缓存页面的数量。最后把 buffer_cpu_notify 加入到通知链中。有关页高速缓存的相关内容，请访问博客“磁盘高速缓存”http://blog.csdn.net/yunsongice/archive/2010/08/23/5833154.aspx。

start_kernel 的 676 行，key_init 函数：

```
void __init key_init(void)
{
	/* allocate a slab in which we can store keys */
	key_jar = kmem_cache_create("key_jar", sizeof(struct key),
			0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);

	/* add the special key types */
	list_add_tail(&key_type_keyring.link, &key_types_list);
	list_add_tail(&key_type_dead.link, &key_types_list);
	list_add_tail(&key_type_user.link, &key_types_list);

	/* record the root user tracking */
	rb_link_node(&root_key_user.node,
		     NULL,
		     &key_user_tree.rb_node);

	rb_insert_color(&root_key_user.node,
			&key_user_tree);

}
```

很简单，建立一个 key 结构的 slab 缓存，并初始化一些全局变量，然后建立一个 key_user_tree 为根节点的红黑树。

start_kernel 的 677 行，security_init 函数。这个函数跟上一个函数一样，都是系统安全相关的内容，感兴趣的同学可以好好研究一下。

5.12 安装根文件系统

start_kernel 下步是另一个重要的函数，678 行的 vfs_caches_init，用于初始化 VFS 那些数据结构的 slab 缓存，来自 fs/dcache.c：

```
2355void __init vfs_caches_init(unsigned long mempages)
2356{
2357		unsigned long reserve;
2358
2359		/* Base hash sizes on available memory, with a reserve equal to
2360		     150% of current kernel size */
2361
2362		reserve = min((mempages - nr_free_pages()) * 3/2, mempages - 1);
2363		mempages -= reserve;
2364
```

```
2365            names_cachep = kmem_cache_create("names_cache", PATH_MAX, 0,
2366                                    SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
2367
2368            dcache_init();
2369            inode_init();
2370            files_init(mempages);
2371            mnt_init();
2372            bdev_cache_init();
2373            chrdev_init();
2374}
```

5.12.1 创建 VFS 相关 slab 缓存

函数首先根据传递进来的 totalram_pages 参数计算还可以作为缓存的页面数量，然后建立一个存放文件名称的 slab 缓存，PATH_MAX 的大小为 4096。随后调用 dcache_init，来自同一文件：

```
2311static void __init dcache_init(void)
2312{
2313            int loop;
2314
2315            /*
2316              * A constructor could be added for stable state like the lists,
2317              * but it is probably not worth it because of the cache nature
2318              * of the dcache.
2319              */
2320            dentry_cache = KMEM_CACHE(dentry,
2321
SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|SLAB_MEM_SPREAD);
2322
2323            register_shrinker(&dcache_shrinker);
2324
2325            /* Hash may have been set up in dcache_init_early */
2326            if (!hashdist)
2327                        return;
2328
2329            dentry_hashtable =
2330                        alloc_large_system_hash("Dentry cache",
2331                                                sizeof(struct hlist_head),
2332                                                dhash_entries,
2333                                                13,
2334                                                0,
```

```
2335                                                    &d_hash_shift,
2336                                                    &d_hash_mask,
2337                                                    0);
2338
2339            for (loop = 0; loop < (1 << d_hash_shift); loop++)
2340                    INIT_HLIST_HEAD(&dentry_hashtable[loop]);
2341}
```

建立 dentry 和 dentry_hashtable，但是由于前面先建立了一个 dentry_hashtable，所以这里就只建立 dentry 的 slab 缓存。继续走，vfs_caches_init 的 2369 行，来自 fs/inode.c 的 inode_init 函数：

```
1563void __init inode_init(void)
1564{
1565            int loop;
1566
1567            /* inode slab cache */
1568            inode_cachep = kmem_cache_create("inode_cache",
1569                                                    sizeof(struct inode),
1570                                                    0,
1571
(SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|
1572                                                    SLAB_MEM_SPREAD),
1573                                                    init_once);
1574            register_shrinker(&icache_shrinker);
1575
1576            /* Hash may have been set up in inode_init_early */
1577            if (!hashdist)
1578                    return;
1579
1580            inode_hashtable =
1581                    alloc_large_system_hash("Inode-cache",
1582                                                    sizeof(struct hlist_head),
1583                                                    ihash_entries,
1584                                                    14,
1585                                                    0,
1586                                                    &i_hash_shift,
1587                                                    &i_hash_mask,
1588                                                    0);
1589
1590            for (loop = 0; loop < (1 << i_hash_shift); loop++)
1591                    INIT_HLIST_HEAD(&inode_hashtable[loop]);
1592}
```

inode_init 跟前面差不多，建立 inode 和 inode_hashtable，但是由于前面先建立了一个 inode_hashtable，所以这里就只建立 inode 的 slab 缓存。vfs_caches_init 的 2370 行，files_init，来自 fs/file_table.c：

```
415void __init files_init(unsigned long mempages)
416{
417         int n;
418
419         filp_cachep = kmem_cache_create("filp", sizeof(struct file), 0,
420                                 SLAB_HWCACHE_ALIGN | SLAB_PANIC, NULL);
421
422         /*
423          * One file with associated inode and dcache is very roughly 1K.
424          * Per default don't use more than 10% of our memory for files.
425          */
426
427         n = (mempages * (PAGE_SIZE / 1024)) / 10;
428         files_stat.max_files = n;
429         if (files_stat.max_files < NR_FILE)
430                 files_stat.max_files = NR_FILE;
431         files_defer_init();
432         percpu_counter_init(&nr_files, 0);
433}
```

files_init 先建立 file 的 slab 缓存，然后根据可用页面数量初始化全局 files_stat_struct 类型变量 files_stat。继续走，vfs_caches_init 的 2371 行，mnt_init 函数，来自 fs/namespace.c：

```
2321void __init mnt_init(void)
2322{
2323         unsigned u;
2324         int err;
2325
2326         init_rwsem(&namespace_sem);
2327
2328         mnt_cache = kmem_cache_create("mnt_cache", sizeof(struct vfsmount),
2329                                 0, SLAB_HWCACHE_ALIGN | SLAB_PANIC, NULL);
2330
2331         mount_hashtable = (struct list_head *)__get_free_page(GFP_ATOMIC);
2332
2333         if (!mount_hashtable)
2334                 panic("Failed to allocate mount hash table\n");
2335
2336         printk("Mount-cache hash table entries: %lu\n", HASH_SIZE);
2337
```

```
2338            for (u = 0; u < HASH_SIZE; u++)
2339                    INIT_LIST_HEAD(&mount_hashtable[u]);
2340
2341            err = sysfs_init();
2342            if (err)
2343                    printk(KERN_WARNING "%s: sysfs_init error: %d\n",
2344                            __func__, err);
2345            fs_kobj = kobject_create_and_add("fs", NULL);
2346            if (!fs_kobj)
2347                    printk(KERN_WARNING "%s: kobj create error\n", __func__);
2348            init_rootfs();
2349            init_mount_tree();
2350}
```

2328 行，建立 vfsmount 的 slab 缓存，初始化全局 mount_hashtable 散列，初始化 sysfs 合 rootfs 文件系统。rootfs 文件系统在初始化阶段很重要，是第一个文件系统，根文件系统，所以接下来我们就来详细看看 2348 行和 2349 行两个函数都干了些什么。

5.12.2 安装 rootfs

在安装 rootfs 之前，首先得通过 2348 行的 init_rootfs()文件对其进行注册，来自 fs/ramfs/inode.c：

```
308 int __init init_rootfs(void)
309 {
310         int err;
311
312  err = bdi_init(&ramfs_backing_dev_info);
313  if (err)
314         return err;
315
316  err = register_filesystem(&rootfs_fs_type);
317  if (err)
318         bdi_destroy(&ramfs_backing_dev_info);
319
320         return err;
321}
```

312 行，一个全局 backing_dev_info 结构的 ramfs_backing_dev_info 变量，定义在同一个文件中：

```
static struct backing_dev_info ramfs_backing_dev_info = {
        .name           = "ramfs",
        .ra_pages = 0,  /* No readahead */
```

```
    .capabilities    = BDI_CAP_NO_ACCT_AND_WRITEBACK |
                BDI_CAP_MAP_DIRECT | BDI_CAP_MAP_COPY |
                BDI_CAP_READ_MAP | BDI_CAP_WRITE_MAP | BDI_CAP_EXEC_MAP,
};
```

这里又得讲一个背景知识。ramfs 是什么？ramfs 是一个非常简单的文件系统，它输出 Linux 的磁盘缓存机制(页缓存和目录缓存)作为一个大小动态的基于内存的文件系统。

通常，一个正常的文件系统，如 ext2 的所有的文件由 Linux 被缓存在页高速缓存中。为了提高系统性能，这些页面的前若干页和后若干页的数据总是保存的，这样可以通过预读外存(一般被挂载的是块设备文件系统)而提升性能；但是这些页面又是标记为可用(空闲)的，所以内存管理系统可以将这些页面作为别用。类似的，在数据写回后备存储时，数据一写回文件就立即被标记为可用，但周围的缓存被保留着直至 MM 重新分配内存。

ramfs 并没有这样的预读和回写机制。文件写入 ramfs 象往常一样，来分配目录和页的缓存，但这里并没有地方可写回它们。这意味着页的数据不再标记为可用，因此当希望回收内存时，内存不能通过 MM 来释放。

实现 ramfs 所需的代码总量是极少的，因为所有的工作由现有的 Linux 缓存结构来完成。实际上，你现正在挂载磁盘缓存作为一个文件系统。据此，ramfs 并不是一个可通过菜单配置项来卸载的可选组件，它可节省的空间是微不足道的。

而 rootfs 是一个特定的 ramfs 的实例，它始终存在于 2.6 的系统。你不能卸载 rootfs，这个理由近似于你不能杀死 init 进程。它小巧且简单的为内核确保某些列表不能为空，而不是拥有特定的代码来检查和处理一个空列表。

大多数的系统挂载另一个文件系统到 rootfs 并忽略它。一个空白 ramfs 实例的空间总量占用是极小的。因此，init_rootfs 的 312 行调用 bdi_init 来初始化全局变量 ramfs_backing_dev_info：

```
int bdi_init(struct backing_dev_info *bdi)
{
    int i, err;

    bdi->dev = NULL;

    bdi->min_ratio = 0;
    bdi->max_ratio = 100;
    bdi->max_prop_frac = PROP_FRAC_BASE;
    spin_lock_init(&bdi->wb_lock);
    INIT_RCU_HEAD(&bdi->rcu_head);
    INIT_LIST_HEAD(&bdi->bdi_list);
    INIT_LIST_HEAD(&bdi->wb_list);
    INIT_LIST_HEAD(&bdi->work_list);
```

```
        bdi_wb_init(&bdi->wb, bdi);
        bdi->wb_mask = 1;
        bdi->wb_cnt = 1;

        for (i = 0; i < NR_BDI_STAT_ITEMS; i++) {
                err = percpu_counter_init(&bdi->bdi_stat[i], 0);
                if (err)
                        goto err;
        }

        bdi->dirty_exceeded = 0;
        err = prop_local_init_percpu(&bdi->completions);

        if (err) {
err:
                while (i--)
                        percpu_counter_destroy(&bdi->bdi_stat[i]);
        }

        return err;
}
```

init_rootfs 的 316 行调用 register_filesystem(&rootfs_fs_type)将 rootfs 的 file_system_type 结构的全局变量 rootfs_fs_type 注册到内核。该变量在编译时被初始化为：

```
static struct file_system_type rootfs_fs_type = {
        .name           = "rootfs",
        .get_sb         = rootfs_get_sb,
        .kill_sb    = kill_litter_super,
};
```

注册文件系统的实质就是把 rootfs_fs_type 加到全局 file_systems 指针所指向的整个 file_system_type 链中。要详细关注 register_filesystem 程序，请学习博客“文件系统注册”http://blog.csdn.net/yunsongice/archive/2010/06/21/5684879.aspx

回到 mnt_init 的 2349 行的 init_mount_tree()函数来自 fs/namespace.c 文件：

```
2298static void __init init_mount_tree(void)
2299{
2300             struct vfsmount *mnt;
2301             struct mnt_namespace *ns;
2302             struct path root;
2303
2304             mnt = do_kern_mount("rootfs", 0, "rootfs", NULL);
2305             if (IS_ERR(mnt))
```

```
2306                    panic("Can't create rootfs");
2307            ns = create_mnt_ns(mnt);
2308            if (IS_ERR(ns))
2309                    panic("Can't allocate initial namespace");
2310
2311            init_task.nsproxy->mnt_ns = ns;
2312            get_mnt_ns(ns);
2313
2314            root.mnt = ns->root;
2315            root.dentry = ns->root->mnt_root;
2316
2317            set_fs_pwd(current->fs, &root);
2318            set_fs_root(current->fs, &root);
2319}
```

这个函数在注册好 rootfs 之后用于完成安装它的任务，2304 行，首先调用 do_kern_mount 安装 rootfs。具体的内容请查阅博客“文件系统安装”
http://blog.csdn.net/yunsongice/archive/2010/06/21/5685067.aspx。
这里只是简单地讲一讲关键过程，do_kern_mount 会调用 vfs_kern_mount 函数，把这个通过 get_fs_type(rootfs)从 file_system_type 链中取到的刚刚注册的 rootfs_fs_type 全局变量作为参数传给它。随后 vfs_kern_mount 函数调用 rootfs_fs_type 的 get_sb 函数，也就是 rootfs_get_sb：

```
static int rootfs_get_sb(struct file_system_type *fs_type,
    int flags, const char *dev_name, void *data, struct vfsmount *mnt)
{
    return get_sb_nodev(fs_type, flags|MS_NOUSER, data, ramfs_fill_super,
                mnt);
}
```

get_sb_nodev 初始化 rootfs 的 super_block 结构，随后会调用传递进来的 ramfs_fill_super 函数参数。这个函数继续初始化 rootfs 的 super_block 结构，并执行一个非常重要的 d_alloc_root 函数将 rootfs 的根目录初始化为“/”，也就是我们常说的根目录：

```
struct dentry * d_alloc_root(struct inode * root_inode)
{
    struct dentry *res = NULL;
    if (root_inode) {
        static const struct qstr name = { .name = "/", .len = 1 };
        res = d_alloc(NULL, &name);
        if (res) {
            res->d_sb = root_inode->i_sb;
            res->d_parent = res;
            d_instantiate(res, root_inode);
        }
    }
    return res;
}
```

回到 init_mount_tree 中，2307 行，为进程 0 的命名空间分配一个 namespace 对象，并将它插入到由 do_kern_mount()函数返回的已安装文件系统描述符中：

```
namespace = kmalloc(sizeof(*namespace), GFP_KERNEL);
list_add(&mnt->mnt_list, &namespace->list);
namespace->root = mnt;
mnt->mnt_namespace = init_task.namespace = namespace;
```

将系统中其他每个进程的 namespace 字段设置为 namespace 对象的地址；同时初始化引用计数器 namespace->count（缺省情况下，所有的进程共享同一个初始 namespace）。

最后两行，切换当前进程，也就是 0 号进程的根目录和当前目录为“ / ”。走到这里，内核第一个文件系统 rootfs 就算安装完毕了。那么为什么内核不怕麻烦，要在安装实际根文件系统之前安装 rootfs 文件系统呢？这是因为，rootfs 文件系统允许内核容易地改变实际根文件系统。实际上，在大多数情况下，系统初始化是内核会逐个地安装和卸载几个根文件系统。例如，一个发布版的初始启动光盘可能把具有一组最小驱动程序的内核装人 RAM 中，内核把存放在 ramdisk 中的一个最小的文件系统作为根安装。接下来，在这个初始根文件系统中的程序探测系统的硬件（例如，它们判断硬盘是否是 EIDE、SCSI 等等），装入所有必需的内核模块，并从物理块设备重新安装根文件系统。

来看 vfs_caches_init 的最后两个函数：

```
void __init bdev_cache_init(void)
{
    int err;
    struct vfsmount *bd_mnt;

    bdev_cachep = kmem_cache_create("bdev_cache", sizeof(struct bdev_inode),
            0, (SLAB_HWCACHE_ALIGN|SLAB_RECLAIM_ACCOUNT|
                SLAB_MEM_SPREAD|SLAB_PANIC),
            init_once);
    err = register_filesystem(&bd_type);
    if (err)
        panic("Cannot register bdev pseudo-fs");
    bd_mnt = kern_mount(&bd_type);
    if (IS_ERR(bd_mnt))
        panic("Cannot create bdev pseudo-fs");
    kmemleak_not_leak(bd_mnt);
    blockdev_superblock = bd_mnt->mnt_sb;    /* For writeback */
}
```

bdev_cache_init 函数初始化块设备驱动的 bdev_inode 结构的 slab 缓存，并且注册块设备成一个文件系统，最后把 blockdev_superblock 全局变量指向这个文件系统的超级块。

```
void __init chrdev_init(void)
{
	cdev_map = kobj_map_init(base_probe, &chrdevs_lock);
	bdi_init(&directly_mappable_cdev_bdi);
}
```

chrdev_init 函数比较简单了，初始化字符设备驱动，也就是向 sysfs 文件系统中注册，我就不详细分析了。

5.12.3 安装 proc 文件系统

回到 start_kernel 的 679 行，signals_init 函数：

```
void __init signals_init(void)
{
	sigqueue_cachep = KMEM_CACHE(sigqueue, SLAB_PANIC);
}
```

很简单，建立数据结构 sigqueue 的 slab 缓存。681 行，page_writeback_init 函数，将全局变量 ratelimit_pages 设置成 vm_total_pages / (num_online_cpus() * 32)，num_online_cpus()宏返回当前 CPU 可用数。而全局变量 vm_total_pages 是在前面 build_all_zonelists 函数中被初始化的。

683 行，调用 proc_root_init 函数，初始化 proc 文件系统：

```
void __init proc_root_init(void)
{
	int err;

	proc_init_inodecache();
	err = register_filesystem(&proc_fs_type);
	if (err)
		return;
	proc_mnt = kern_mount_data(&proc_fs_type, &init_pid_ns);
	err = PTR_ERR(proc_mnt);
	if (IS_ERR(proc_mnt)) {
		unregister_filesystem(&proc_fs_type);
		return;
	}

	proc_symlink("mounts", NULL, "self/mounts");
```

```
        proc_net_init();

#ifdef CONFIG_SYSVIPC
        proc_mkdir("sysvipc", NULL);
#endif
        proc_mkdir("fs", NULL);
        proc_mkdir("driver", NULL);
        proc_mkdir("fs/nfsd", NULL); /* somewhere for the nfsd filesystem to be mounted */
#if                defined(CONFIG_SUN_OPENPROMFS)                ||
defined(CONFIG_SUN_OPENPROMFS_MODULE)
        /* just give it a mountpoint */
        proc_mkdir("openprom", NULL);
#endif
        proc_tty_init();
#ifdef CONFIG_PROC_DEVICETREE
        proc_device_tree_init();
#endif
        proc_mkdir("bus", NULL);
        proc_sys_init();
}
```

这个函数也很好理解，首先调用 register_filesystem 注册 proc 文件系统：

```
static struct file_system_type proc_fs_type = {
        .name           = "proc",
        .get_sb         = proc_get_sb,
        .kill_sb    = proc_kill_sb,
};
```

跟 rootfs 一样，proc 文件系统的 proc_get_sb 会 proc_fill_super 函数，把/proc 作为文件系统的跟目录赋给其 super_block 的 s_root 字段。然后 proc_root_init 会在/proc 目录下建立一个符号链接文件和若干个子目录，具体的代码就不去管它了。

回到 start_kernel 中，CONFIG_CGROUPS 没有被我们配置，所以 685 行的 cgroup_init 函数是一个空函数。同样，也没有配置 CONFIG_CPUSETS，686 行的 cpuset_init 也是空函数。start_kernel 的随后三个函数，taskstats_init_early、delayacct_init 和 check_bugs 都是跟系统性能和缺陷测试相关的函数，我们这里略过。

看到 692 行，这里有个 acpi_early_init 函数，用于高级配置和电源管理接口（Advanced Configuration and Power Management Interface，ACPI）的初始化。如果对 ACPI（注意不是 APIC 喔，不要搞混淆了）想做深入了解的同学可以先回到前面的 setup_arch 中，分析一下它的 979 行的 acpi_boot_table_init()函数和 1023 行的 acpi_boot_init 函数；然后再回过头来看这里的 acpi_early_init()都干了些什么：在 dynamic memory 中申请 ACPI tables 的空间，然后从别处把它们 copy 过来；初试化所有 ACPI 相关的全局变量；从 DSDT/SSDTs/PSDT 表获取数据，构建 ACPI namespace。

回到start_kernel中，由于没有配置CONFIG_SFI和CONFIG_FTRACE_MCOUNT_RECORD，所以随后两行的 sfi_init 和 ftrace_init 是两个空函数。

6 后 start_kernel 时代

至此，start_kernel()函数完成了 Linux 内核的初始化工作。几乎每天内核部件都是由这个函数进行初始化的，下面让我们再来回顾一下其中最重要的部分：

- 调用 setup_arch()函数，根据处理器硬件平台设置系统；解析 linux 命令行参数；设置 0 号进程的内存描述结构 init_mm；系统内存管理初始化；统计并注册系统各种资源；以及其它项目的初始化等。
- 调用 sched_init()函数来初始化调度程序。
- 调用 build_all_zonelists()函数来初始化内存管理区。
- 调用 mem_init()函数来初始化伙伴系统分配程序。
- 调用 kmem_cache_init()函数来初始化 slab 分配器。
- 调用 vmalloc_init()函数来初始化非连续内存区。
- 调用 trap_init()函数和 init_IRQ()函数以完成 IDT 初始化。
- 调用 softirq_init()函数初始化软中断的 TASKLET_SOFTIRQ 和 HI_SOFTIRQ。
- 调用 time_init()函数来初始化系统日期和时间。
- 调用 calibrate_delay()函数以确定 CPU 时钟的速度。

顺便插入说一句，这个函数的代码被放在.text.init 中，我们在链接的时候以__init_begin 开始，__init_end 结尾，就向模块初始化函数一样，它仅仅被执行一次，内核在最后阶段 init_post 函数会调用 free_init_pages 收回这个分节所占用的内存，到时候咱们再来看。

6.1 创建 1 号进程

好了，start_kernel 在初始化了内核之后，最后一行代码是

```
/* Do the rest non-__init'ed, we're now alive */
rest_init();
```

可以看到 start_kernel 最后是调用 rest_init 函数进行后续的初始化，来自 init/main.c 文件：

```
424static noinline void __init_refok rest_init(void)
425        __releases(kernel_lock)
426{
427        int pid;
428
429        rcu_scheduler_starting();
430        kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
431        numa_default_policy();
432        pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
433        rcu_read_lock();
```

```
434             kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
435             rcu_read_unlock();
436             unlock_kernel();
437
438             /*
439               * The boot idle thread must execute schedule()
440               * at least once to get things moving:
441               */
442             init_idle_bootup_task(current);
443             preempt_enable_no_resched();
444             schedule();
445             preempt_disable();
446
447             /* Call into cpu_idle with preempt disabled */
448             cpu_idle();
449}
```

430 行，它首先就是执行 kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND)得到一个 pid 为 1 的进程，这是一个内核线程，执行 kernel_init 函数。有关内核线程的知识，请查阅博客“内核线程”http://blog.csdn.net/yunsongice/archive/2010/04/23/5522012.aspx。

432 行，执行 kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES)来启动内核线程 kthreadd，它的工作是用来运行 kthread_create_list 全局链表中的 kthread

注意，当前计算机早已淘汰了内核抢占机制，所以一般 CONFIG_PREEMPT 都没有被设置，443 和 445 行代码是两行空代码。445 行代码在 rq 中找到一个已准备好的进程让它先运行，如果没有，就 448 行执行 cpu_idle()：

```
void cpu_idle(void)
{
        int cpu = smp_processor_id();
        boot_init_stack_canary();

        current_thread_info()->status |= TS_POLLING;

        /* endless idle loop with no priority at all */
        while (1) {
                tick_nohz_stop_sched_tick(1);
                while (!need_resched()) {

                        check_pgt_cache();
                        rmb();

                        if (cpu_is_offline(cpu))
```

```
                play_dead();

            local_irq_disable();
            /* Don't trace irqs off for idle */
            stop_critical_timings();
            pm_idle();
            start_critical_timings();
        }
        tick_nohz_restart_sched_tick();
        preempt_enable_no_resched();
        schedule();
        preempt_disable();
    }
}
```

它基本就是节省 cpu 的体力，进入 idle 循环消耗空闲的时间片，谁要 CPU 就让给谁，所以就不说了。我们沿着流程走，当 1 号进程 do_fork 好了之后，schedule 就会在运行队列 rq 中选中它，执行它的代码，也就是 kernel_init 函数，也来自同一个文件：

```
854static int __init kernel_init(void * unused)
855{
856         lock_kernel();
857
858         /*
859          * init can allocate pages on any node
860          */
861         set_mems_allowed(node_states[N_HIGH_MEMORY]);
862         /*
863          * init can run on any cpu.
864          */
865         set_cpus_allowed_ptr(current, cpu_all_mask);
866         /*
867          * Tell the world that we're going to be the grim
868          * reaper of innocent orphaned children.
869          *
870          * We don't want people to have to make incorrect
871          * assumptions about where in the task array this
872          * can be found.
873          */
874         init_pid_ns.child_reaper = current;
875
876         cad_pid = task_pid(current);
877
878         smp_prepare_cpus(setup_max_cpus);
```

```
879
880             do_pre_smp_initcalls();
881             start_boot_trace();
882
883             smp_init();
884             sched_init_smp();
885
886             do_basic_setup();
887
888             /* Open the /dev/console on the rootfs, this should never fail */
889             if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
890                     printk(KERN_WARNING "Warning: unable to open an initial console.\n");
891
892             (void) sys_dup(0);
893             (void) sys_dup(0);
894             /*
895              * check if there is an early userspace init.  If yes, let it do all
896              * the work
897              */
898
899             if (!ramdisk_execute_command)
900                     ramdisk_execute_command = "/init";
901
902             if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0) {
903                     ramdisk_execute_command = NULL;
904                     prepare_namespace();
905             }
906
907             /*
908              * Ok, we have completed the initial bootup, and
909              * we're essentially up and running. Get rid of the
910              * initmem segments and start the user-mode stuff..
911              */
912
913             init_post();
914             return 0;
915}
```

856~884 是一系列初始化，CONFIG_CPUSETS 没有被配置，所以 861 行 set_mems_allowed 是个空函数；865 行 set_cpus_allowed_ptr 函数用来修改进程的 CPU 亲和力
段段段段段段段段段段节节节节节节节节节节节节节
kernel_init 函数的一开始就调用了 lock_kernel()函数，当编译时选上了 CONFIG_LOCK_KERNEL，就加上大内核锁，否
则啥也不做，紧接着就调用了函数 set_cpus_allowed_ptr，由于这些函数对 init 进程的调起还

是有影响的，我们还是一个一个来瞧瞧吧，
不要忘了啥东东最好，

```
static inline int set_cpus_allowed_ptr(struct task_struct *p,
                                       const cpumask_t *new_mask)
{
        if (!cpu_isset(0, *new_mask))
                return -EINVAL;
        return 0;
}
```

这函数其实就调用了 cpu_isset 宏，定义在文件"include/linux/cpumask.h 中，如下：

```
#define cpu_isset(cpu, cpumask) test_bit((cpu), (cpumask).bits)
```

再来看看 set_cpus_allowed_ptr 的第二个参数类型吧，也定义在文件 include/linux/cpumask.h 中，具体如下：

```
typedef struct { DECLARE_BITMAP(bits, NR_CPUS); } cpumask_t;
```

接着尾随着 DECLAR_BITMAP 宏到文件 include/linux/types.h 中，定义如下：

```
#define DECLARE_BITMAP(name,bits) \
        unsigned long name[BITS_TO_LONGS(bits)]
```

而宏 BITS_TO_LONGS 定义在文件 include/linux/bitops.h 中，实现如下：

```
#define BITS_TO_LONGS(nr)       DIV_ROUND_UP(nr, BITS_PER_BYTE * sizeof(long))
```

DIV_ROUND_UP 宏定义在文件 include/linux/kernel.h 中，BITS_PER_BYTE 宏定义在文件 include/linux/bitops.h 中，实现如下：

```
#define DIV_ROUND_UP(n,d) (((n) + (d) - 1) / (d))
#define BITS_PER_BYTE           8
```

即当 NR_CPUS 为 1～32 时，cpumask_t 类型为

```
struct {
unsigned long bits[1];
        }
```

然后来看看在 set_cpus_allowed_ptr(current, CPU_MASK_ALL_PTR); 中的 CPU_MASK_ALL_PTR 宏，定义在 include/linux/cpumask.h 中：

```
#define CPU_MASK_ALL_PTR        (&CPU_MASK_ALL)
```

而 CPU_MASK_ALL 宏也定义在文件 include/linux/cpumask.h 中：

```
#define CPU_MASK_ALL                                                    \
(cpumask_t) { {                                                         \
        [BITS_TO_LONGS(NR_CPUS)-1]      =       CPU_MASK_LAST_WORD      \
} }
```

NR_CPUS 宏定义在文件 include/linux/threads.h 中，实现如下：

```
#ifdef CONFIG_SMP
#define NR_CPUS         CONFIG_NR_CPUS
#else
#define NR_CPUS         1
#endif
```

CPU_MASK_LAST_WORD 宏定义在文件 include/linux/cpumask.h 中，实现如下：

```
#define CPU_MASK_LAST_WORD BITMAP_LAST_WORD_MASK(NR_CPUS)
```

BITMAP_LAST_WORD_MASK(NR_CPUS)宏定义在文件 include/linux/bitmap.h 中，实现如下：

```
#define BITMAP_LAST_WORD_MASK(nbits)                                    \
(                                                                       \
        ((nbits) % BITS_PER_LONG) ?                                     \
                (1ULcpu_isset(0,CPU_MASK_ALL_PTR)                       ー
ー>test_bit(0,CPU_MASK_ALL_PTR.bits)
```

即当 NR_CPUS 为 n 时，就把 usigned long bits[0]的第 n 位置 1，应该就如注释所说的，init 能运行在任何 CPU 上吧。

现

在 kernel_init 中的 set_cpus_allowed_ptr(current, CPU_MASK_ALL_PTR);

分析完了，我们接着往下看，首先 init_pid_ns.child_reaper = current;

init_pid_ns 定义在 kernel/pid.c 文件中

```
struct pid_namespace init_pid_ns = {
        .kref = {
                .refcount       = ATOMIC_INIT(2),
        },
        .pidmap = {
                [ 0 ... PIDMAP_ENTRIES-1] = { ATOMIC_INIT(BITS_PER_PAGE),
NULL }
        },
        .last_pid = 0,
        .level = 0,
        .child_reaper = &init_task,
};
```

它是一个 pid_namespace 结构的变量，先来看看 pid_namespace 的结构，它定义在文件 include/linux/pid_namespace.h 中，具体定义如下：

```
struct pid_namespace {
        struct kref kref;
        struct pidmap pidmap[PIDMAP_ENTRIES];
        int last_pid;
        struct task_struct *child_reaper;
        struct kmem_cache *pid_cachep;
        unsigned int level;
        struct pid_namespace *parent;
#ifdef CONFIG_PROC_FS
        struct vfsmount *proc_mnt;
#endif
};
```

即把当前进程设为接受其它孤儿进程的进程，然后取得该进程的进程 ID，如：

```
cad_pid = task_pid(current);
```

然后调用 smp_prepare_cpus(setup_max_cpus);如果编译时没有指定 CONFIG_SMP，它什么也不做，接着往下看，调用 do_pre_smp_initcalls()函数，它定义在 init/main.c 文件中，实现如下：

```
static void __init do_pre_smp_initcalls(void)
{
        extern int spawn_ksoftirqd(void);
        migration_init();
        spawn_ksoftirqd();
        if (!nosoftlockup)
                spawn_softlockup_task();
}
```

其中 migration_init()定义在文件 include/linux/sched.h 中，具体实现如下:

```
#ifdef CONFIG_SMP
void migration_init(void);
#else
static inline void migration_init(void)
{
}
#endif
```

好像什么也没有做，然后是调用 spawn_ksoftirqd()函数，定义在文件 kernel/softirq.c 中，代码如下：

```
__init int spawn_ksoftirqd(void)
{
        void *cpu = (void *)(long)smp_processor_id();
        int err = cpu_callback(&cpu_nfb, CPU_UP_PREPARE, cpu);
        BUG_ON(err == NOTIFY_BAD);
        cpu_callback(&cpu_nfb, CPU_ONLINE, cpu);
        register_cpu_notifier(&cpu_nfb);
        return 0;
}
```

在该函数中，首先调用 smp_processor_id 函数获得当前 CPU 的 ID 并把它赋值给变量 cpu，然后把 cpu 连同&cpu_nfb，CPU_UP_PREPARE 传递给函数 cpu_callback，我们先看 cpu_callback 的前几行：

```
static int __cpuinit cpu_callback(struct notifier_block *nfb,
                                  unsigned long action,
                                  void *hcpu)
{
        int hotcpu = (unsigned long)hcpu;
        struct task_struct *p;
        switch (action) {
        case CPU_UP_PREPARE:
        case CPU_UP_PREPARE_FROZEN:
                p = kthread_create(ksoftirqd, hcpu, "ksoftirqd/%d", hotcpu);
                if (IS_ERR(p)) {
                        printk("ksoftirqd for %i failed\n", hotcpu);
                        return NOTIFY_BAD;
                }
```

```
                kthread_bind(p, hotcpu);
                per_cpu(ksoftirqd, hotcpu) = p;
                break;
```

从
上述代码可以看出当 action 为 CPU_PREPARE 时，将创建一个内核线程并把它赋值给 p，该进程所要运行的函数为 ksoftirqd，传递给该函
数的参数为 hcpu，而紧跟其后的 ” ksoftirqd/%d ” ,hotcpu 为该进程的名字参数，这就是我们在终端用命令 ps -ef | grep
ksoftirqd 所看到的线程；如果进程创建失败，打印出错信息，否则把创建的线程 p 绑定到当前 CPU 的 ID 上，这就是
kthread_bind(p,hotcpu)所做的，接下来的几行为：

```
case CPU_ONLINE:
        case CPU_ONLINE_FROZEN:
                wake_up_process(per_cpu(ksoftirqd, hotcpu));
                break;
```

即
在 spawn_ksoftirqd 函数中 cpu_callback(&cpu_nfb, CPU_ONLINE,
cpu);的 action 为 CPU_ONLINE 时，将调用 wake_up_process 函数来唤醒当前 CPU 上的 ksoftirqd 进程。最后调用
register_cpu_notifier(&cpu_nfb)；其实也没做什么，只是简单的返回 0。返回到
do_pre_smp_initcalls 函数中，接着往下看：

```
if (!nosoftlockup)
                spawn_softlockup_task();
```

spawn_softlockup_task()函数定义在文件 include/linux/sched.h 中，是个空函数。
到
现在为止，do_pre_smp_initcalls 分析完了，它主要就是创建进程 ksoftirqd，把它绑定到当前 CPU 上，然后再把该进程拷贝给每
个 CPU，并唤醒所有 CPU 上的进程 ksoftirqd，就是当我们执行 ps -ef | grep ksoftirqd 的时候所看到的：

```
root         4     2  0 08:30 ?        00:00:03 [ksoftirqd/0]
root         7     2  0 08:30 ?        00:00:02 [ksoftirqd/1]
```

革命尚未成功，同志仍需努力！接着享受吧，呵呵！
现在到了 kernel_init 函数中的 smp_init();了
如果在编译时没有选择 CONFIG_SMP，若定义 CONFIG_X86_LOCAL_APIC 则去调用 APIC_init_uniprocessor()函数，否则什么也不做，具体代码定义在文件 init/main.c 中：

```
#ifndef CONFIG_SMP
#ifdef CONFIG_X86_LOCAL_APIC
static void __init smp_init(void)
{
        APIC_init_uniprocessor();
}
#else
#define smp_init()      do { } while (0)
#endif
```

如果在编译时选择了 CONFIG_SMP 呢，那么它的实现就如下喽：

```
/* Called by boot processor to activate the rest. */
static void __init smp_init(void)
{
        unsigned int cpu;
        /* FIXME: This should be done in userspace --RR */
        for_each_present_cpu(cpu) {
                if (num_online_cpus() >= setup_max_cpus)
                        break;
                if (!cpu_online(cpu))
                        cpu_up(cpu);
        }
        /* Any cleanup work */
        printk(KERN_INFO "Brought up %ld CPUs\n", (long)num_online_cpus());
        smp_cpus_done(setup_max_cpus);
}
```

来看看这个函数的，for_each_present_cpu(cpu)宏在文件 include/linux/cpumask.h 中实现：

```
#define for_each_present_cpu(cpu) for_each_cpu_mask((cpu), cpu_present_map)
```

而 for_each_cpu_mask(cpu,mask)宏也在文件 include/linux/cpumask.h 中实现：

```
#if NR_CPUS > 1
#define for_each_cpu_mask(cpu, mask)                    \
        for ((cpu) = first_cpu(mask);                   \
                (cpu) < NR_CPUS;                        \
                (cpu) = next_cpu((cpu), (mask)))
#else /* NR_CPUS == 1 */
#define for_each_cpu_mask(cpu, mask)                    \
        for ((cpu) = 0; (cpu) < 1; (cpu)++, (void)mask)
#endif /* NR_CPUS */
```

即对于每个 cpu 都要执行大括号里的语句，如果当前 cpu 没激活就把它激活的，该函数然后打印一些 cpu 信息，如当前激活的 cpu 数目。

kernel_init 中紧跟 smp_init()函数后的是 sched_init_smp()函数和 do_basic_setup()函数，而其后便是最后一个函数 init_post()，在该函数中将调起 init 进程。由于内容较多，下次分析.....

之后：

1.调用 do_basic_setup 函数

2.然后就 init 进程的执行了

3.最后内核就处于不断等待服务的过程了。

6.2 子系统的初始化

所以接下来说 do_basic_setup 函数，任然是来自 init/main.c：

```
778/*
```

```
779 * Ok, the machine is now initialized. None of the devices
780 * have been touched yet, but the CPU subsystem is up and
781 * running, and memory and process management works.
782 *
783 * Now we can finally start doing some real work..
784 */
785static void __init do_basic_setup(void)
786{
787             init_workqueues();
788             cpuset_init_smp();
789             usermodehelper_init();
790             init_tmpfs();
791             driver_init();
792             init_irq_proc();
793             do_ctors();
794             do_initcalls();
795}
```

第一个函数 init_workqueues，初始化工作队列 events，每个 CPU 一个，对工作队列机制感兴趣的同学请访问博客“ 下半部分 ”
http://blog.csdn.net/yunsongice/archive/2010/03/07/5354011.aspx。

第二个函数 cpuset_init_smp 是个空函数，因为没有配置 CONFIG_CPUSETS。

第三个函数 usermodehelper_init，初始化工作队列 khelper，每个 CPU 一个。

第四个函数 init_tmpfs，注册并安装 tmpfs 文件系统，它的 file_system_type 结构如下：

```
static struct file_system_type tmpfs_fs_type = {
      .owner          = THIS_MODULE,
      .name           = "tmpfs",
      .get_sb         = shmem_get_sb,
      .kill_sb    = kill_litter_super,
};
```

第五个函数 driver_init，建立设备驱动模型 sysfs 的 kset、kobject 和 subsystem 结构，并向其中注册 cpu、内存和总线的驱动。

第六个函数 init_irq_proc，向/proc 文件系统中增加子目录 irq 来显示中断描述符表中的所有元素。

第七个函数 do_ctors，不是太明白，忽略它。

最后来说说第八个函数 do_initcalls();

首先说一下，内核中有一个专门的节，用来存放初始化末尾要被调用的函数。举个例子 init/initramfs.c 中的最后一句是：

```
rootfs_initcall(populate_rootfs);
```

而：

```
#define rootfs_initcall(fn)                    __define_initcall("rootfs",fn,rootfs)
#define __define_initcall(level,fn,id) \
        static initcall_t __initcall_##fn##id __attribute_used__ \
        __attribute__((__section__(".initcall" level ".init"))) = fn
```

可以看出来这个 populate_rootfs 被放在.initcallrootfs.init 节中了。然后呢，这个节在链接的时候会被 output 到.initcall.init 节中 （为了避免交叉过多）。

好了，在明白了 initcall 相关信息后，继续看看 do_initcalls()；
它的本质就是

```
        for (call = __initcall_start; call < __initcall_end; call++)
                result = (*call)();
```

这样所有的 initcall 就会被调用了，这是为了避免把所有代码集合到一块的一个方法，虽然这带来了一个问题，谁在前谁在后？内核专门为这个大节分了一些类，哪些在前哪些在后连接的时候会安排的。

正好，现在知道 populate_rootfs 会被执行，它处理 initfamfs 和 initrd，首先是执行 unpack_to_rootfs，然后检查是否有 initrd。代码就不列出来了。

再回到调用 do_basic_setup()的 kernel_init 中：接下来

```
    if (!ramdisk_execute_command)
            ramdisk_execute_command = "/init";
    if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0) {
            ramdisk_execute_command = NULL;
            prepare_namespace();
    }
```

这段代码检查是否有必要 mount 根文件系统，如果 vmlinuz 中带有 initfamfs，而且其中已经有 init，那么就不这么做了(我现在工作用的目标系统就是这样的，里面有个 init)，否则的话内核还要 mount init 所在的（也是所有用户态进程的最除根文件系统）根文件系统，挂在根文件系统和执行 init 是 linux 启动过程最后要做的事情。

好了，就不说 mount_root 的细节了，prepare_namespace 还是很值得一看的，可以去翻源代码。如果它失败了，就是 panic VFS no root found 这样的错误了。现在假设已经有了根文件系统了，这样就到了 kernel_init 中的最后一条函数调用了——init_post。

6.3 启动 shell 环境

init_post 函数来自同一个文件的 814 行：

```
811/* This is a non __init function. Force it to be noinline otherwise gcc
812 * makes it inline to init() and it becomes part of init.text section
813 */
814static noinline int init_post(void)
815         __releases(kernel_lock)
816{
817         /* need to finish all async __init code before freeing the memory */
818         async_synchronize_full();
819         free_initmem();
820         unlock_kernel();
821         mark_rodata_ro();
822         system_state = SYSTEM_RUNNING;
823         numa_default_policy();
824
825
826         current->signal->flags |= SIGNAL_UNKILLABLE;
827
828         if (ramdisk_execute_command) {
829                   run_init_process(ramdisk_execute_command);
830                   printk(KERN_WARNING "Failed to execute %s\n",
831                                       ramdisk_execute_command);
832         }
833
834         /*
835           * We try each of these until one succeeds.
836           *
837           * The Bourne shell can be used instead of init if we are
838           * trying to recover a really broken machine.
839           */
840         if (execute_command) {
841                   run_init_process(execute_command);
842                   printk(KERN_WARNING "Failed to execute %s.    Attempting "
843                                                 "defaults...\n", execute_command);
844         }
845         run_init_process("/sbin/init");
846         run_init_process("/etc/init");
847         run_init_process("/bin/init");
848         run_init_process("/bin/sh");
849
```

```
850             panic("No init found.  Try passing init= option to kernel. "
851                     "See Linux Documentation/init.txt for guidance.");
852}
```

819 行，这就是上面说到的那个释放 init 代码的函数了，它显然不能是__init 打头：

```
void free_initmem(void)
{
    free_init_pages("unused kernel memory",
            (unsigned long)(&__init_begin),
            (unsigned long)(&__init_end));
}
```

这个函数基本上就是执行 init 了，失败就 panic 了。顺便说一句，/dev/console 最后被 012 描述符引用，也就是所有没有 reopen 的进程的标准输入输出和出错。

到此，内核启动过程就完成了，init 根据根文件系统的配置在初始化用户态的进程，启动系统。