

Linux 内核分析方法谈

作者：华中科技大学 喻锋荣 (hustyuucc@163.net)

Linux 的最大的好处之一就是它的源码公开。同时，公开的核心源码也吸引着无数的电脑爱好者和程序员；他们把解读和分析 Linux 的核心源码作为自己的最大兴趣，把修改 Linux 源码和改造 Linux 系统作为自己对计算机技术追求的最大目标。

Linux 内核源码是很具吸引力的，特别是当你弄懂了一个分析了好久都没搞懂的问题；或者是被你修改过了的内核，顺利通过编译，一切运行正常的时候。那种成就感真是油然而生！而且，对内核的分析，除了出自对技术的狂热追求之外，这种令人生畏的劳动所带来的回报也是非常令人着迷的，这也正是它拥有众多追随者的主要原因：

✍ 首先，你可以从中学到很多的计算机的底层知识，如后面将讲到的系统的引导和硬件提供的中断机制等；其它，象虚拟存储的实现机制，多任务机制，系统保护机制等等，这些都是非都源码不能体会的。

✍ 同时，你还将从操作系统的整体结构中，体会整体设计在软件设计中的份量和作用，以及一些宏观设计的方法和技巧：Linux 的内核为上层应用提供一个与具体硬件不相关的平台；同时在内核内部，它又把代码分为与体系结构和硬件相关的部分，和可移植的部分；再例如，Linux 虽然不是微内核的，但他把大部分的设备驱动处理成相对独立的内核模块，这样减小了内核运行的开销，增强了内核代码的模块独立性。。。

✍ 而且你还能从对内核源码的分析中，体会到它在解决某个具体细节问题时，方法的巧妙：如后面将分析到了的 Linux 通过 Botom_half 机制来加快系统对中断的处理。

✍ 最重要的是：在源码的分析过程中，你将会被一点一点地、潜移默化地专业化。一个专业的程序员，总是把代码的清晰性，兼容性，可移植性放在很重要的位置。他们总是通过定义大量的宏，来增强代码的清晰度和可读性，而又不增加编译后的代码长度和代码的运行效率；他们总是在编码的同时，就考虑到了以后的代码维护和升级。。。甚至，只要分析百分之一的代码后，你就会深刻地体会到，什么样的代码才是一个专业的程序员写的，什么样的代码是一个业余爱好者写的。而这一点是任何没有真正分析过标准代码的人都无法体会到的。

然而，由于内核代码的冗长，和内核体系结构的庞杂，所以分析内核也是一个很艰难，很需要毅力的事；在缺乏指导和交流的情况下，尤其如此。只有方法正确，才能事半功倍。正是基于这种考虑，作者希望通过此文能给大家一些借鉴和启迪。

由于本人所进行的分析都是基于 2.2.5 版本的内核；所以，如果没有特别说明，以下分析都是基于 i386 单处理器的 2.2.5 版本的 Linux 内核。所有源文件均是相对于目录 /usr/src/linux 的。

方法之一、从何入手

要分析 Linux 内核源码，首先必须找到各个模块的位置，也即要弄懂源码的文件组织形式。虽然对于有经验的高手而言，这个不是很难；但对于很多初级的 Linux 爱好者，和那些对源码分析很有兴趣但接触不多的人来说，这还是很有必要的。

1. Linux 核心源程序通常都安装在 `/usr/src/linux` 下，而且它有一个非常简单的编号约定：任何偶数的核心（的第二个数为偶数，例如 2.0.30）都是一个稳定地发行的核心，而任何奇数的核心（例如 2.1.42）都是一个开发中的核心。

2. 核心源程序的文件按树形结构进行组织，在源程序树的最上层，即目录 `/usr/src/linux` 下有这样一些目录和文件：

COPYING: GPL 版权申明。对具有 GPL 版权的源代码改动而形成的程序，或使用 GPL 工具产生的程序，具有使用 GPL 发表的义务，如公开源代码；
CREDITS: 光荣榜。对 Linux 做出过很大贡献的一些人的信息；
MAINTAINERS: 维护人员列表，对当前版本的内核各部分都有谁负责；
Makefile: 第一个 Makefile 文件。用来组织内核的各模块，记录了个模块间的相互这间的联系和依托关系，编译时使用；仔细阅读各子目录下的 Makefile 文件对弄清各个文件这间的联系和依托关系很有帮助；
ReadMe: 核心及其编译配置方法简单介绍；
Rules.make: 各种 Makefilemake 所使用的一些共同规则；
REPORTING-BUGS: 有关报告 Bug 的一些内容；

Arch/ : arch 子目录包括了所有和体系结构相关的核心代码。它的每一个子目录都代表一种支持的体系结构，例如 i386 就是关于 intel cpu 及与之相兼容体系结构的子目录。PC 机一般都基于此目录；

Include/: include 子目录包括编译核心所需要的大部分头文件。与平台无关的头文件在 `include/linux` 子目录下，与 intel cpu 相关的头文件在 `include/asm-i386` 子目录下，而 `include/scsi` 目录则是有关 scsi 设备的头文件目录；

Init/ : 这个目录包含核心的初始化代码（注：不是系统的引导代码），包含两个文件 `main.c` 和 `Version.c`，这是研究核心如何工作的好的起点之一。

Mm/ : 这个目录包括所有独立于 cpu 体系结构的内存管理代码，如页式存储管理内存的分配和释放等；而和体系结构相关的内存管理代码则位于 `arch/*/mm/`，例如 `arch/i386/mm/Fault.c`

Kernel/ : 主要的核心代码，此目录下的文件实现了大多数 linux 系统的内核函数，其中最重要的文件当属 `sched.c`；同样，和体系结构相关的代码在 `arch/*/kernel` 中；

Drivers/ : 放置系统所有的设备驱动程序；每种驱动程序又各占用一个子目录：如，`/block` 下为块设备驱动程序，比如 `ide (ide.c)`。如果你希望查看所有可能包含文

件系统的设备是如何初始化的,你可以看 `drivers/block/genhd.c` 中的 `device_setup()`。它不仅初始化硬盘,也初始化网络,因为安装 `nfs` 文件系统的时候需要网络;

`Documentation/`: 文档目录,没有内核代码,只是一套有用的文档,可惜都是 `English` 的,看看应该有用的哦;

`Fs/`: 所有的文件系统代码和各种类型的文件操作代码,它的每一个子目录支持一个文件系统,例如 `fat` 和 `ext2`;

`Ipc/`: 这个目录包含核心的进程间通讯的代码;

`Lib/`: 放置核心的库代码;

`Net/`: 核心与网络相关的代码;

`Modules/`: 模块文件目录,是个空目录,用于存放编译时产生的模块目标文件;

`Scripts/`: 描述文件,脚本,用于对核心的配置;

一般,在每个子目录下,都有一个 `Makefile` 和一个 `Readme` 文件,仔细阅读这两个文件,对内核源码的理解很有用。

对 Linux 内核源码的分析,有几个很好的入口点:一个就是系统的引导和初始化,即从机器加电到系统核心的运行;另外一个就是系统调用,系统调用是用户程序或操作调用核心所提供的功能的接口。对于那些对硬件比较熟悉的爱好者,从系统的引导入手进行分析,可能来的容易一些;而从系统调用下口,则可能更合适于那些在 `dos` 或 `Uinx`、`Linux` 下有过 `C` 编程经验的高手。这两点,在后面还将介绍到。

方法之二、以程序流程为线索，一线串珠

从表面上看，Linux 的源码就象一团乱麻，其实它是一个组织得有条有理的蛛网。要把整个结构分析清楚，除了找出线头，还得理顺各个部分之间的关系，有条不紊的一点一点的分析。

所谓以程序流程为线索、一线串珠，就是指根据程序的执行流程，把程序执行过程所涉及到的代码分析清楚。这种方法最典型的应用有两个：一是系统的初始化过程；二是应用程序的执行流程：从程序的装载，到运行，一直到程序的退出。

为了简便起见，遵从循序渐进的原理，现就系统的初始化过程来具体的介绍这种方法。系统的初始化流程包括：系统引导，实模式下的初始化，保护模式下的初始化共三个部分。下面将一一介绍。

I、系统的引导

Linux 系统的常见引导方式有两种：Lilo 引导和 Loadin 引导；同时 linux 内核也自带了一个 bootsect-loader。由于它只能实现 linux 的引导，不像前两个那样具有很大的灵活性（lilo 可实现多重引导、loadin 可在 dos 下引导 linux），所以在普通应用场合实际上很少使用 bootsect-loader。当然，bootsect-loader 也具有它自己的优点：短小没有多余的代码、附带在内核源码中、是内核源码的有机组成部分，等等。

bootsect-loader 在内和源码中对应的程序是 /Arch/i386/boot/bootsect.S。下面将主要是针对此文件进行的分析。

一、 几个相关文件：

- <1.> /Arch/i386/boot/bootsect.S
- <2.> /include/linux/config.h
- <3.> /include/asm/boot.h
- <4.> /include/linux/autoconf.h

二、 引导过程分析：

对于 Intel x86 PC，开启电源后，机器就会开始执行 ROM BIOS 的一系列系统测试动作，包括检查 RAM，keyboard，显示器，软硬磁盘等等。执行完 bios 的系统测试之后，紧接着控制权会转移给 ROM 中的启动程序(ROM bootstrap routine)；这个程序会将磁盘上的第 0 轨第 0 扇区（叫 boot sector 或 MBR <Master Boot Record>，系统的引导程序就放在此处）读入内存中，并放到自 0x07C0:0x0000 开始的 512 个字节处；然后处理机将跳到此处开始执行这一引导程序；也即装入 MBR 中的引导程序后，CS:IP = 0x07C0:0x0000。加电后处理机运行在与 8086 相兼容的实模式下。

如果要用 bootsect-loader 进行系统引导，则必须把 bootsect.S 编译连接后对应的二进制代码置于 MBR；当 ROM BIOS 把 bootsect.S 编译连接后对应的二进制代码装入内存后，机器的控制权就完全转交给 bootsect；也就是说，bootsect 将是第一个被读入内存中并执行的程序。

Bootsect 接管机器控制权后，将依次进行以下一些动作：

1. 首先，bootsect 将它“自己”(自位置 0x07C0:0x0000 开始的 512 个字节)从被 ROM BIOS 载入的地址 0x07C0:0x0000 处搬到 0x9000:0000 处；这一任务由 bootsect.S 的前十条指令完成；第十一条指令“`jmp go, INITSEG`”则把机器跳转到“新”的 bootsect 的“`jmp go, INITSEG`”后的那条指令“`go: mov di, #0x4000-12`”；之后，继续执行 bootsect 的剩下的代码；在 bootsect.S 中定义了几个常量：

<code>BOOTSEG = 0x07C0</code>	bios 载入 MBR 的约定位置的段址；
<code>INITSEG = 0x9000</code>	bootsect.S 的前十条指令将自己搬到此处(段址)
<code>SETUPSEG = 0x9020</code>	装入 Setup.S 的段址
<code>SYSSEG = 0x1000</code>	系统区段址

对于这些常量可参见 `/include/asm/boot.h` 中的定义；这些常量在下面的分析中将会经常用到；

2. 以 0x9000:0x4000-12 为栈底，建立自己的栈区；其中 0x9000:0x4000-12 到 0x9000:0x4000 的一十二个字节预留作磁盘参数表区；

3. 在 0x9000:0x4000-12 到 0x9000:0x4000 的一十二个预留字节中建立新的磁盘参数表，之所以叫“新”的磁盘参数表，是相对于 bios 建立的磁盘参数表而言的。由于设计者考虑到有些老的 bios 不能准确地识别磁盘“每个磁道的扇区数”，从而导致 bios 建立的磁盘参数表妨碍磁盘的最高性能发挥，所以，设计者就在 bios 建立的磁盘参数表的基础上通过枚举法测试，试图建立准确的“新”的磁盘参数表(这是在后继步骤中完成的)；并把参数表的位置由原来的 0x0000:0x0078 搬到 0x9000:0x4000-12；且修改老的磁盘参数表区使之指向新的磁盘参数表；

4. 接下来就到了 `load_setup` 子过程；它调用 0x13 中断的第 2 号服务；把第 0 道第 2 扇区开始的连续的 `setup_sects` (为常量 4) 个扇区读到紧邻 bootsect 的内存区；即 0x9000:0x0200 开始的 2048 个字节；而这四个扇区的内容即是 `/arch/i386/boot/setup.S` 编译连接后对应的二进制代码；也就是说，如果要用 bootsect-loader 进行系统引导，不仅必须把 bootsect.S 编译连接后对应的二进制代码置于 MBR，而且还得把 `setup.S` 编译连接后对应的二进制代码置于紧跟 MBR 后的连续四个扇区中；当然，由于 `setup.S` 对应的可执行码是由 bootsect 装载的，所以，在我们的这个项目中可以通过修改 bootsect 来根据需要随意地放置 `setup.S` 对应的可执行码；

5. `load_setup` 子过程的唯一出口是 `probe_loop` 子过程；该过程通过枚举法测试磁盘“每个磁道的扇区数”；

6. 接下来几个子过程比较清晰易懂：打印我们熟悉的“Loading”；读入系统到 0x1000:0x0000；关掉软驱马达；根据的 5 步测出的“每个磁道的扇区数”确定磁盘类型；最后跳转到 0x9000:0x0200，即 `setup.S` 对应的可执行码的入口，将机器控制权转交 `setup.S`；整个 bootsect 代码运行完毕；

三、 引导过程执行完后的内存印象图：



出于简便考虑，在此分析中，我忽略了对大内核的处理的分析，因为对大内核的处理，只是此引导过程中的一个很小的部分，并不影响对整体的把握。完成了系统的引导后，系统将进入到初始化处理阶段。系统的初始化分为实模式和保护模式两部分。

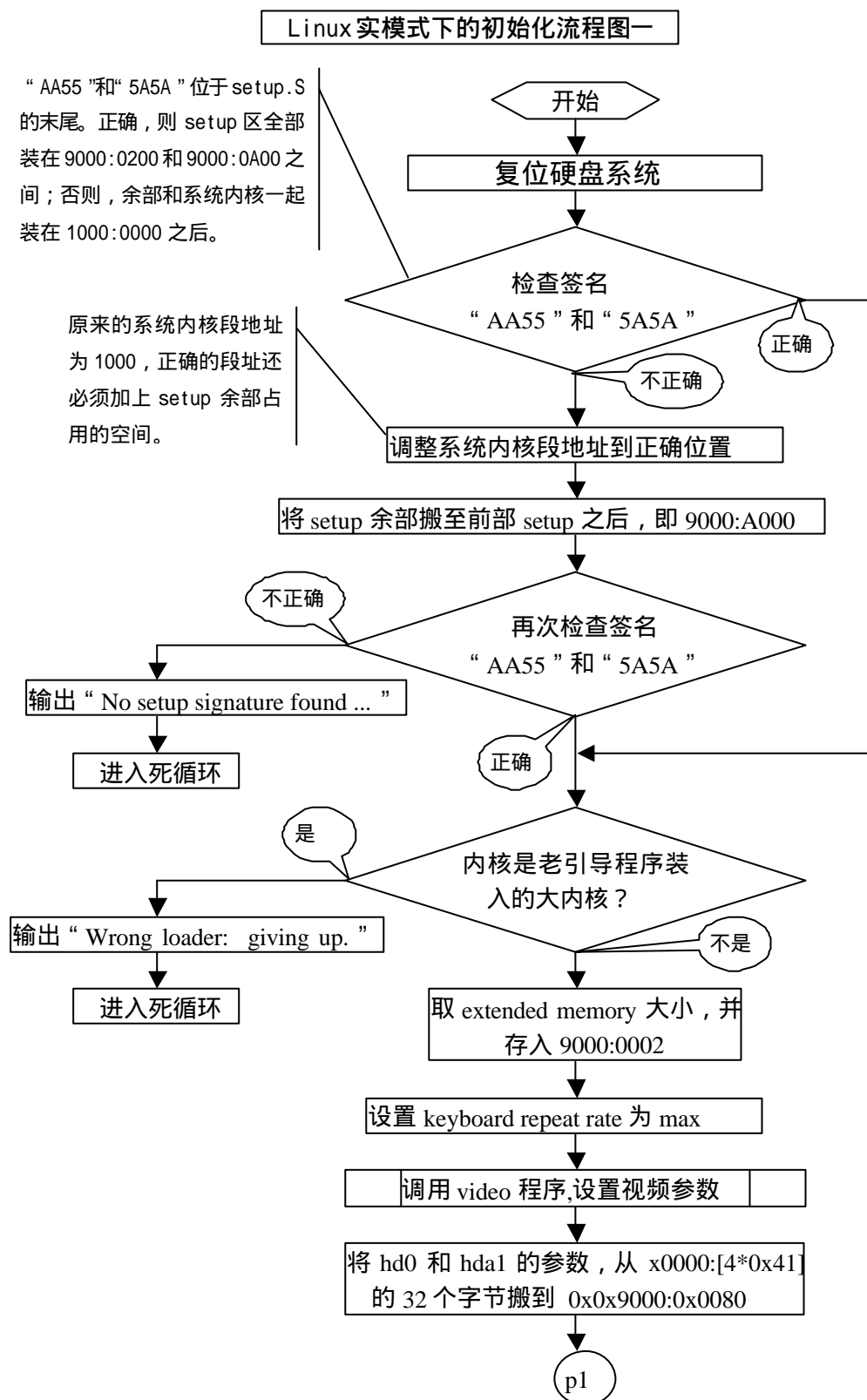
II、实模式下的初始化

实模式下的初始化,主要是指从内核引导成功后，到进入保护模式之前系统所做的一些处理。在内核源码中对应的程序是 /Arch/i386/boot/setup.S ；以下部分主要是针对此文件进行的分析。这部分的分析主要是要弄懂它的处理流程和 INITSEG(9000:0000)段参数表的建立，此参数表包含了很多硬件参数，这些都是以后进行保护模式下初始化，以及核心建立的基础。

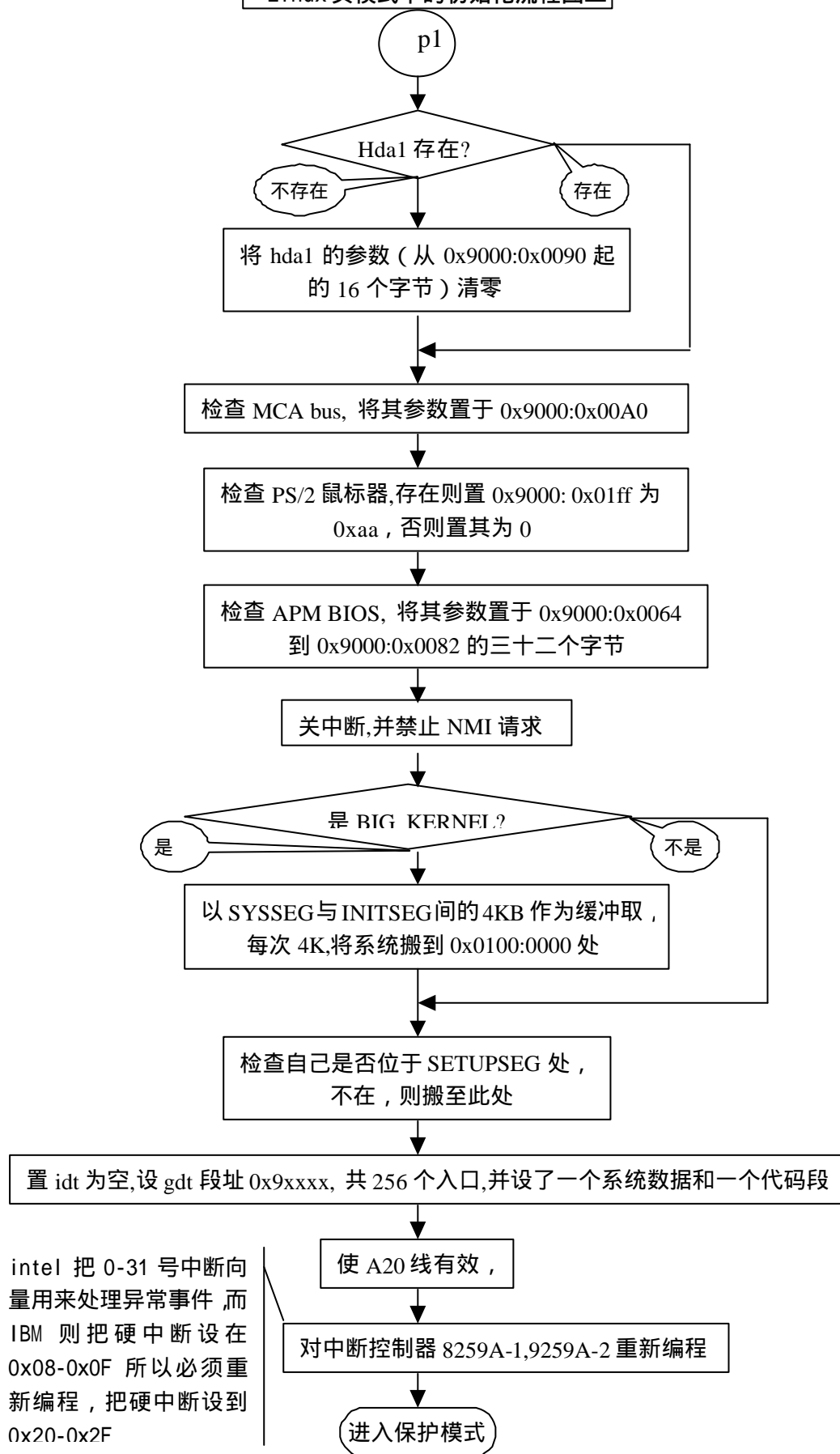
一、 个其它相关文件：

- <1.> /Arch/i386/boot/bootsect.S
- <2.> /include/linux/config.h
- <3.> /include/asm/boot.h
- <4.> /include/asm/segment.h
- <5.> /include/linux/version.h
- <6.> /include/linux/compile.h

二、实模式下的初始化过程分析：



Linux 实模式下的初始化流程图二



INITSEG(9000:0000)段参数表 : (参见 Include/linux/tty.h)

参数名	偏移量(段址均为 0x9000)	长度 Byte	参考文件
PARAM_CURSOR_POS	0x0000	2	Arch/i386/boot/video.S
extended mem Size	0x0002	2	Arch/i386/boot/setup.S
PARAM_VIDEO_PAGE	0x0004	2	Arch/i386/boot/video.S
PARAM_VIDEO_MODE	0x0006	1	Arch/i386/boot/video.S
PARAM_VIDEO_COLS	0x0007	1	Arch/i386/boot/video.S
没用	0x0008	2	Include/linux/tty.h
PARAM_VIDEO_EGA_BX	0x000a	2	Arch/i386/boot/video.S
没用	0x000c	2	Include/linux/tty.h
PARAM_VIDEO_LINES	0x000e	1	Arch/i386/boot/video.S
PARAM_HAVE_VGA	0x000f	1	Arch/i386/boot/video.S
PARAM_FONT_POINTS	0x0010	2	Arch/i386/boot/video.S
PARAM_LFB_WIDTH	0x0012	2	Arch/i386/boot/video.S
PARAM_LFB_HEIGHT	0x0014	2	Arch/i386/boot/video.S
PARAM_LFB_DEPTH	0x0016	2	Arch/i386/boot/video.S
PARAM_LFB_BASE	0x0018	4	Arch/i386/boot/video.S
PARAM_LFB_SIZE	0x001c	4	Arch/i386/boot/video.S
暂未用	0x0020	4	Include/linux/tty.h
PARAM_LFB_LINELENGTH	0x0024	2	Arch/i386/boot/video.S
PARAM_LFB_COLORS	0x0026	6	Arch/i386/boot/video.S
暂未用	0x002c	2	Arch/i386/boot/video.S
PARAM_VESAPM_SEG	0x002e	2	Arch/i386/boot/video.S
PARAM_VESAPM_OFF	0x0030	2	Arch/i386/boot/video.S
PARAM_LFB_PAGES	0x0032	2	Arch/i386/boot/video.S
保留	0x0034--0x003f		Include/linux/tty.h
APM BIOS Version	0x0040	2	Arch/i386/boot/setup.S
BIOS code segment	0x0042	2	Arch/i386/boot/setup.S
BIOS entry offset	0x0044	4	Arch/i386/boot/setup.S
BIOS 16 bit code seg	0x0048	2	Arch/i386/boot/setup.S
BIOS data segment	0x004a	2	Arch/i386/boot/setup.S
支持 32 位标志	0x004c	2	Arch/i386/boot/setup.S
BIOS code seg length	0x004e	4	Arch/i386/boot/setup.S
BIOS data seg length	0x0052	2	Arch/i386/boot/setup.S
hd0 参数	0x0080	16	Arch/i386/boot/setup.S
hd0 参数	0x0090	16	Arch/i386/boot/setup.S
PS/2 device 标志	0x01ff	1	Arch/i386/boot/setup.S

* 注 : Include/linux/tty.h : CL_MAGIC and CL_OFFSET here
 Include/linux/tty.h :
 unsigned char rsvd_size; /* 0x2c */

```
unsigned char  rsvd_pos;    /* 0x2d */  
0 表示没有 APM BIOS  
0x0002 置位表示支持 32 位模式  
0 表示没有, 0x0aa 表示有鼠标器
```

III、保护模式下的初始化

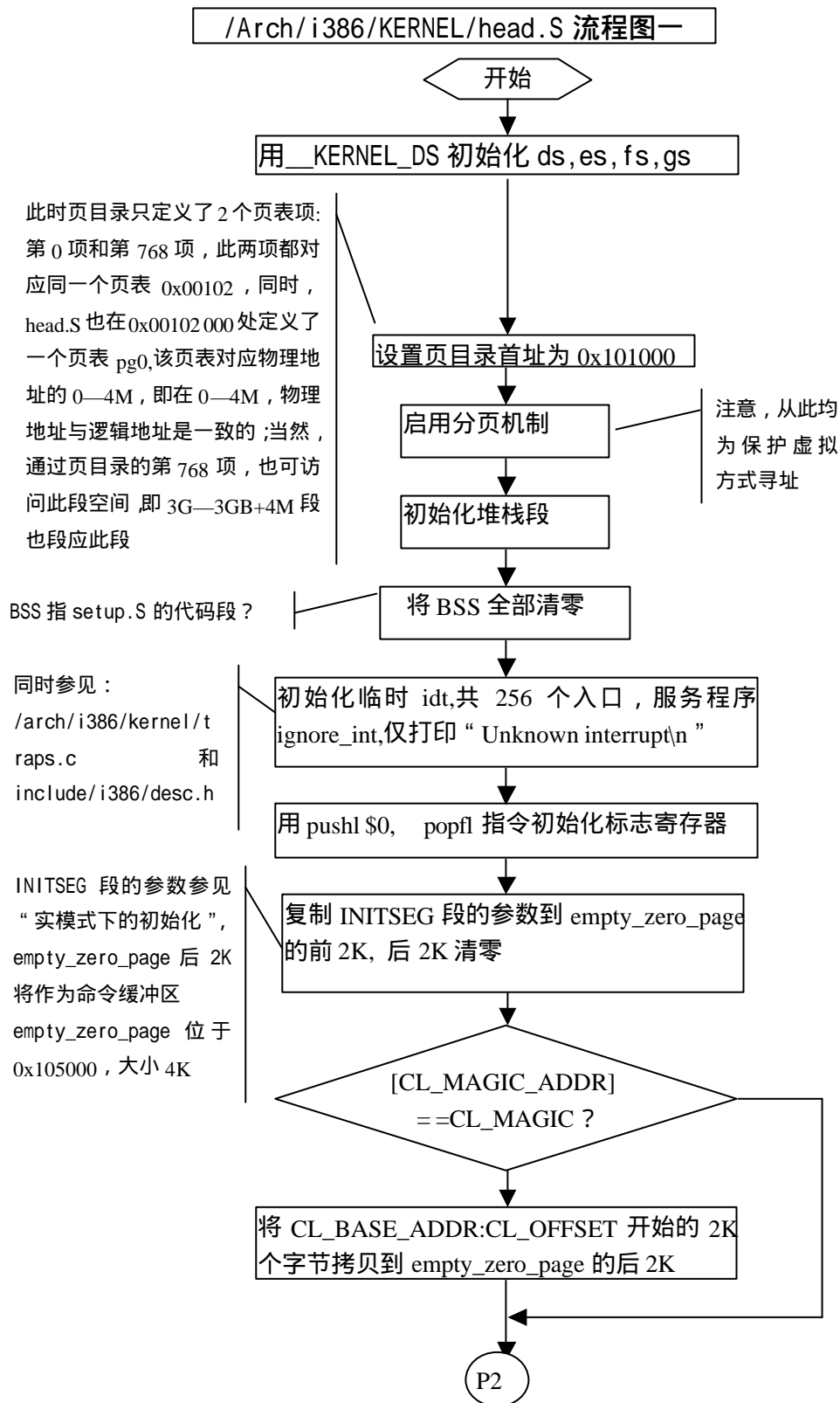
保护模式下的初始化, 是指处理机进入保护模式后到运行系统第一个内核程序过程中系统所做的一些处理。保护模式下的初始化在内核源码中对应的程序是 /Arch/i386/boot/compressed/head.S 和 /Arch/i386/KERNEL/head.S ; 以下部分主要是针对这两个文件进行的分析。

一、 几个相关文件:

```
<1.>    /Arch/i386/boot/compressed/head.S  
<2.>    /Arch/i386/KERNEL/head.S  
<3.>    //Arch/i386/boot/compressed/MISC.c  
<4.>    /Arch/i386/boot/setup.S  
<5.>    /include/asm/segment.h  
<6.>    /arch/i386/kernel/traps.c  
<7.>    /include/i386/desc.h  
<8.>    /include/asm-i386/processor.h
```

二、 保护模式下的初始化过程分析:

一、/Arch/i386/KERNEL/head.S 流程：



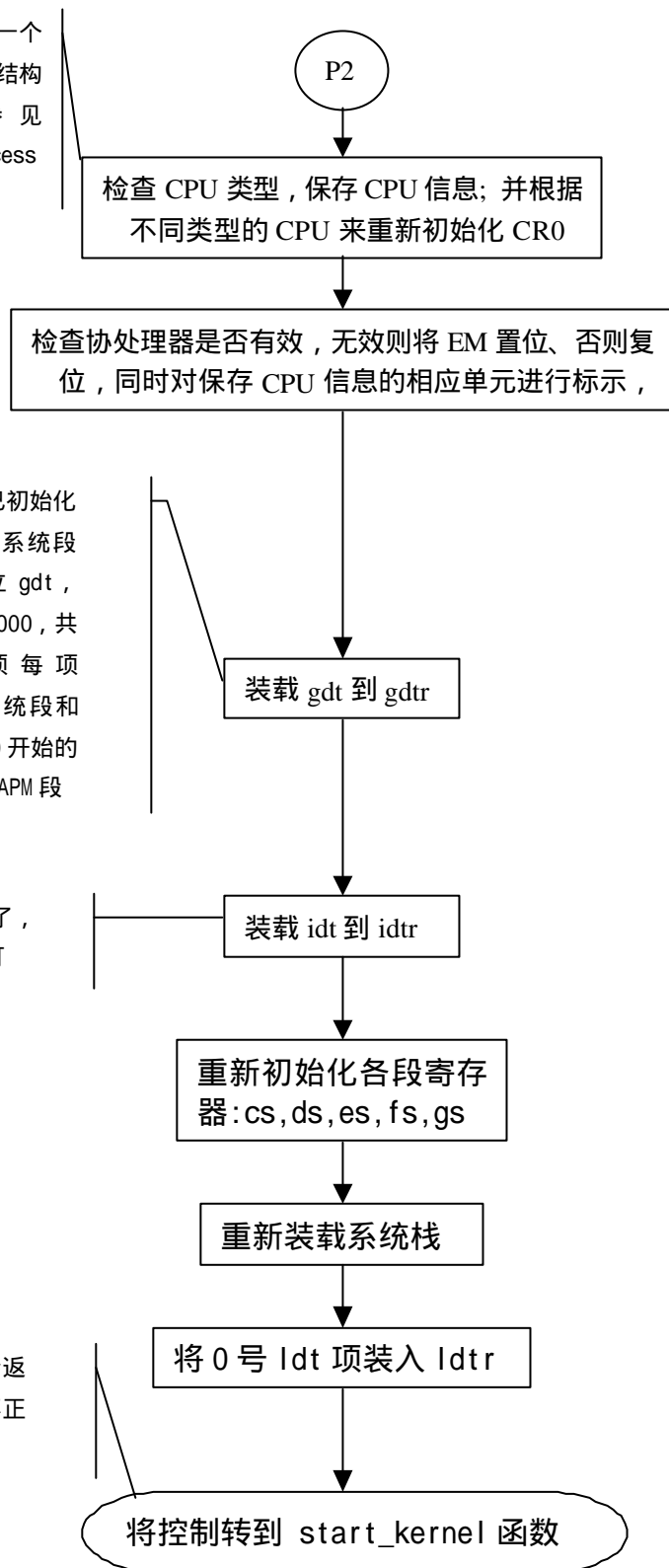
/Arch/i386/KERNEL/head.S 流程图二

有关 CPU 的信息被放在一个命名 boot_cpu_data 的结构 cpuinfo_x86 中，参见 include/asm-i386/processor.h

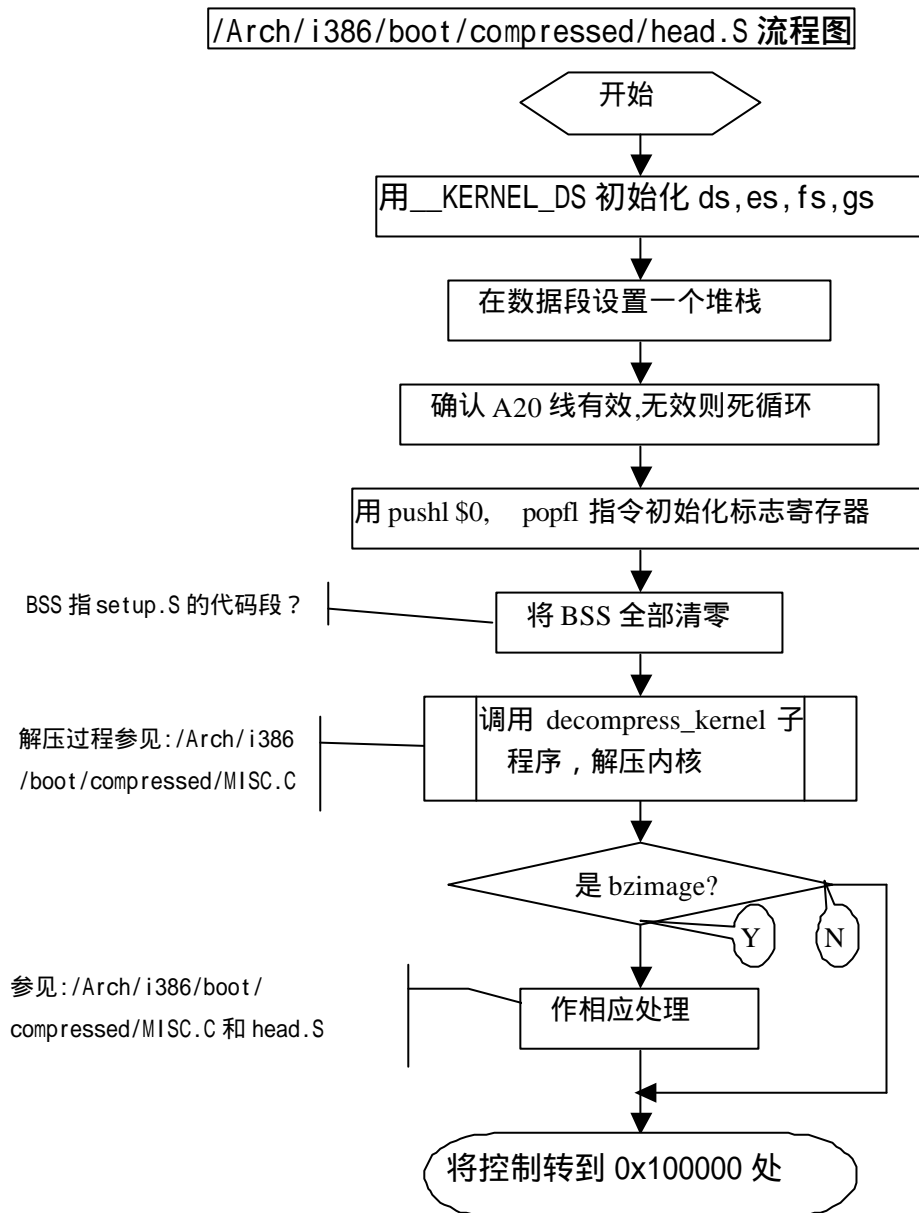
在进入保护模式前，已初始化了一个只含两个有效系统段的 Gdt；此处重新建立 gdt，新的 gdt 位于 0x106000，共 12+2*NR_TASKS 个项 每项 8byte，设置了两个系统段和两个用户段，都是从 0 开始的 4G 大小，还设了 4 个 APM 段

此时 idt 已经初始化了，只需装载到 idtr 即可

正常情况下，不会返回，若返回，则不正常，所以死循环



二、/Arch/i386/boot/compressed/head.S 流程：



1. 从流程图中可以看到，保护模式下的初始化主要干了这样几件事：

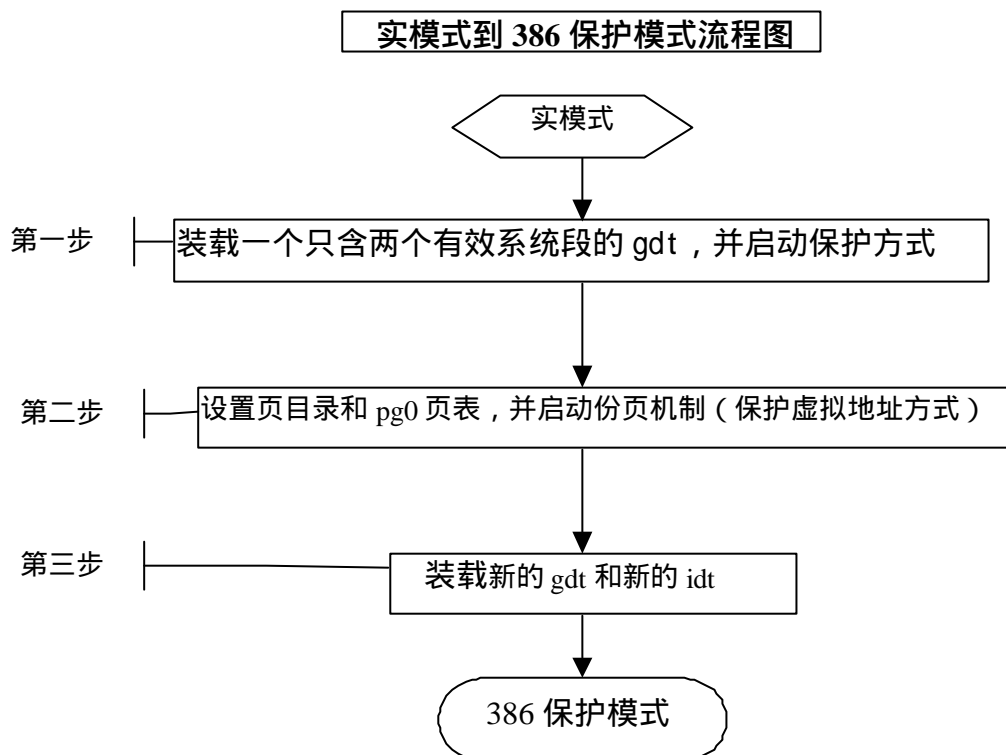
- A. 解压内核到 0x100000 处、
- B. 建立页目录和 pg0 页表并启动分页功能(即虚存管理功能)、
- C. 保存实模式下测到的硬件信息到 empty_zero_page、初始化命令缓存区、
- D. 检测 cpu 类型、检查协处理器、
- E. 重新建立 gdt 全局描述符表、和中断描述附表 idt；

2. 从页目录和 pg0 页表可以看出，0—4M 物理内存被用作系统区，它被映射到系统段线性空间的 0—4M 和 3G—3G+4M；即系统可以通过访问这两个段来访问实际的 0—4M 物理内存，也就是系统所在的区域；

3. 本来在实模式下初始化时已经建立了全局描述符表 gdt, 而此处重新建立全局描述符表 gdt 则主要是出于两个原因: 一个就是若内核是大内核 bzimag, 则以前建立的 gdt, 可能已经在解压时被覆盖掉了所以, 在这个源码文件中均只采用相对转移指令 jxx nf 或 jxx nb; 二是以前建立的 gdt 是建立在实地址方式下的, 而现在则是在启用保护虚拟地址方式之后建立的, 也即现在的 gdt 是建立在逻辑地址 (即线性地址) 上的;

4. 每次建立新的 gdt 后和启用保护虚拟地址方式后都必须重新装载系统栈和重新初始化各段寄存器: cs, ds, es, fs, gs;

5. 从实模式下的初始化和保护模式下的初始化过程可以看出, linux 系统由实模式进入到保护模式的过程大致如下:



6. 由于分页机制只能在保护模式下启动, 不能在实模式下启动, 所以第一步是必要的; 又因为在 386 保护模式下 gdt 和 idt 是建立在逻辑地址 (线性地址) 上的, 所以第三步也是必要的;

7. 经过实模式和保护模式下的初始后，主要系统数据分布如下：

初始后主要系统数据分布表

位置	系统数据	大小
0x101000	页目录 swapper_pg_dir	4K
0x102000	页表 pg0	4K
0x103000	empty_bad_page	4K
0x104000	empty_bad_page_table	4K
0x105000	empty_zero_page	4K
0x105000	系统硬件参数	2K
0x105800	命令缓冲区	2K
0x106000	全局描述附表 gdt_table	4192B

从上面对 Linux 系统的初始化过程的分析可以看出，以程序执行流程为线索、一线串珠，就是按照程序的执行先后顺序，看懂程序执行的各个阶段所进行的处理，及其各阶段之间的相互联系。而流程图应该是这种分析方法最合适的表达工具。

事实上，以程序执行流程为线索，是分析任何源代码都首选的方法。由于操作系统的特殊性，光用这种方法是远远不够的。当然用这种方法来分析系统的初始化过程或用户进程的执行流程应该说是很有效的。

方法之三、以数据结构为基点，触类旁通

结构化程序设计思想认为：程序 = 数据结构 + 算法。数据结构体现了整个系统的构架，所以数据结构通常都是代码分析的很好的着手点，对 Linux 内核分析尤其如此。比如，把进程控制块结构分析清楚了，就对进程有了基本的把握；再比如，把页目录结构和页表结构弄懂了，两级虚存映射和内存管理也就掌握得差不多了。为了体现循序渐进的思想，在这我就以 Linux 对中断机制的处理来介绍这种方法。

首先，必须指出的是：在此处，中断指广义的中断概义，它指所有通过 idt 进行的控制转移的机制和处理；它覆盖以下几个常用的概义：中断、异常、可屏蔽中断、不可屏蔽中断、硬中断、软中断

I、硬件提供的中断机制和约定

一. 中断向量寻址：

硬件提供可供 256 个服务程序中断进入的入口，即中断向量；

中断向量在保护模式下的实现机制是中断描述符表 idt，idt 的位置由 idtr 确定，idtr 是个 48 位的寄存器，高 32 位是 idt 的基址，低 16 位为 idt 的界限(通常为 $2k=256*8$)；

idt 中包含 256 个中断描述符，对应 256 个中断向量；每个中断描述符 8 位，其结构如图一：

图一、中断描述符格式

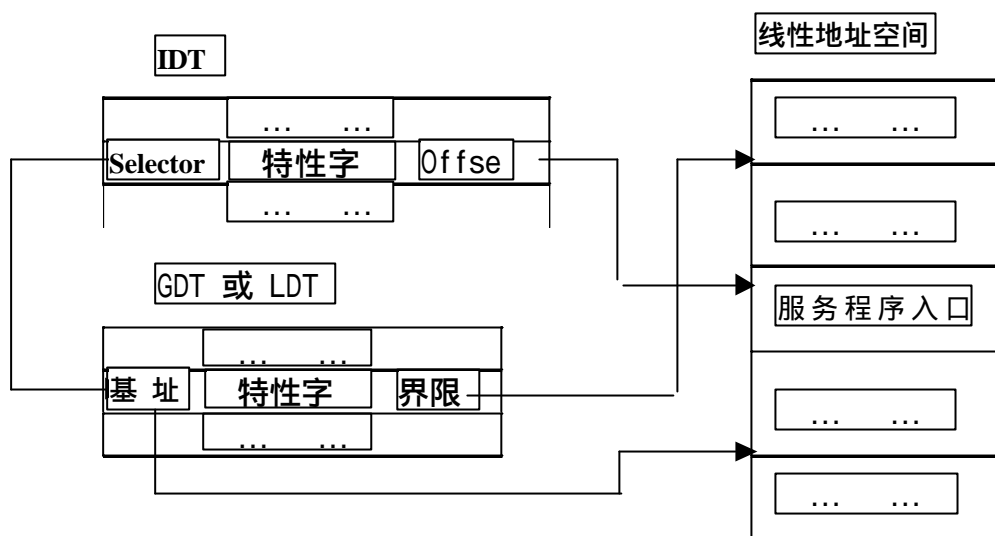
0-1	Offset(15...0)			
2-3	Selector			
	15	14	13,12 8,7	0
4-5	P	DPL	0x0E	0x00
6-7	Offset(31...16)			

注：P=1，该描述符有效，P=0，无效，使用则触发异常
DPL, 00-11, 对应四个特权级（00 最高）

中断进入过程如图二所示。

当中断是由低特权级转到高特权级(即当前特权级 $CPL > DPL$)时，将进行堆栈的转移；内层堆栈的选择由当前 tss 的相应字段确定，而且内层堆栈将依次被压入如下数据：外层 SS, 外层 ESP, EFLAGS, 外层 CS, 外层 EIP； 中断返回过程为一逆过程；

图二、中断进入示意图



二. 异常处理机制：

intel 公司保留 0-31 号中断向量用来处理异常事件：当产生一个异常时，处理机就会自动把控制转移到相应的处理程序的入口，异常的处理程序由操作系统提供，中断向量和异常事件对应如表一：

表一、中断向量和异常事件对应表

中断向量号	异常事件	Linux 的处理程序
0	除法错误	Divide_error
1	调试异常	Debug
2	NMI 中断	Nmi
3	单字节, int 3	Int3
4	溢出	Overflow
5	边界监测中断	Bounds
6	无效操作码	Invalid_op
7	设备不可用	Device_not_available
8	双重故障	Double_fault
9	协处理器段溢出	Coprocessor_segment_overrun
10	无效 TSS	Invalid_tss
11	缺段中断	Segment_not_present
12	堆栈异常	Stack_segment
13	一般保护异常	General_protection
14	页异常	Page_fault
15		Spurious_interrupt_bug
16	协处理器出错	Coprocessor_error
17	对齐检查中断	Alignment_check

三. 可编程中断控制器 8259A :

为更好的处理外部设备，x86 微机提供了两片可编程中断控制器，用来辅助 cpu 接受外部的中断信号；对于中断，cpu 只提供两个外接引线：NMI 和 INTR；

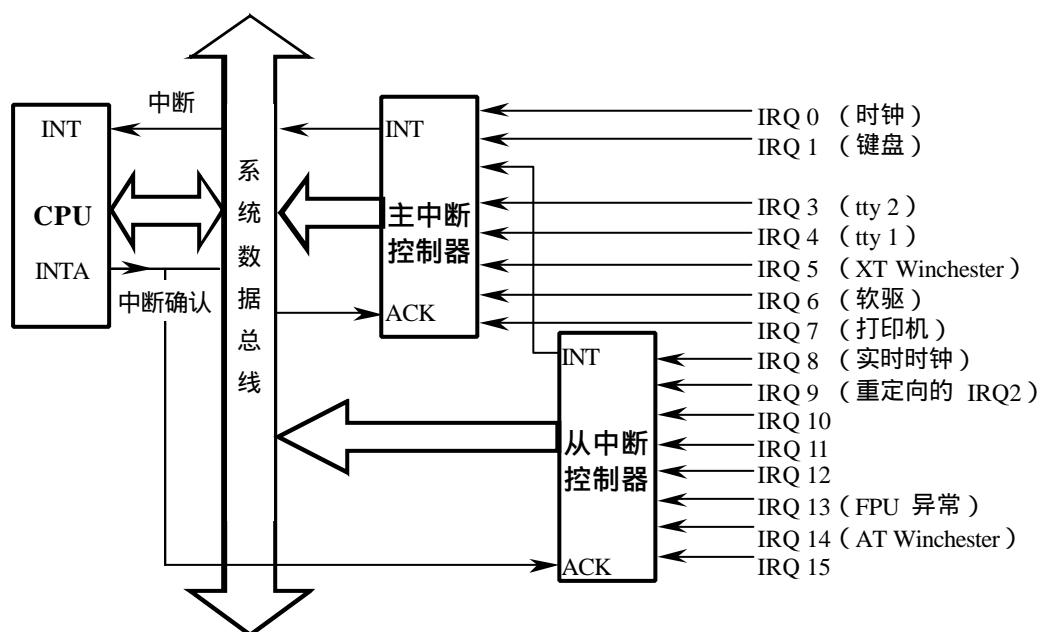
NMI 只能通过端口操作来屏蔽，它通常用于：电源掉电和物理存储器奇偶验错；

INTR 可通过直接设置中断屏蔽位来屏蔽，它可用来接受外部中断信号，但只有一个引线，不够用，所以它通过外接两片级联了的 8259A，以接受更多的外部中断信号。8259A 主要完成这样一些任务：

- a. 中断优先级排队管理，
- b. 接受外部中断请求
- c. 向 cpu 提供中断类型号

外部设备产生的中断信号在 IRQ（中断请求）管脚上首先由中断控制器处理。中断控制器可以响应多个中断输入，它的输出连接到 CPU 的 INT 管脚，信号可通过 INT 管脚，通知处理器产生了中断。如果 CPU 这时可以处理中断，CPU 会通过 INTA（中断确认）管脚上的信号通知中断控制器已接受中断，这时，中断控制器可将一个 8 位数据放置在数据总线上，这一 8 位数据也称为中断向量号，CPU 依据中断向量号和中断描述符表（IDT）中的信息自动调用相应的中断服务程序。图三中，两个中断控制器级联了起来，从属中断控制器的输出连接到了主中断控制器的第 3 个中断信号输入，这样，该系统可处理的外部中断数量最多可达 15 个，图的右边是 i386 PC 中各中断输入管脚的一般分配。可通过对 8259A 的初始化，使这 15 个外接引脚对应 256 个中断向量的任何 15 个连续的向量；由于 intel 公司保留 0-31 号中断向量用来处理异常事件(而默认情况下，IBM bios 把硬中断设在 0x08-0x0f)，所以，硬中断必须设在 31 以后，linux 则在实模式下初始化时把其设在 0x20-0x2F，对此下面还将具体说明。

图三、i386 PC 可编程中断控制器 8259A 级链示意图



II、Linux 的中断处理

硬件中断机制提供了 256 个入口，即 idt 中包含的 256 个中断描述符（对应 256 个中断向量）。

而 0-31 号中断向量被 intel 公司保留用来处理异常事件，不能另作它用。对这 0-31 号中断向量，操作系统只需提供异常的处理程序，当产生一个异常时，处理机就会自动把控制转移到相应的处理程序的入口，运行相应的处理程序；而事实上，对于这 32 个处理异常的中断向量，此版本（2.2.5）的 Linux 只提供了 0-17 号中断向量的处理程序，其对应处理程序参见 表一、中断向量和异常事件对应表；也就是说，17-31 号中断向量是空着未用的。

既然 0-31 号中断向量已被保留，那么，就是剩下 32-255 共 224 个中断向量可用。这 224 个中断向量又是怎么分配的呢？在此版本（2.2.5）的 Linux 中，除了 0x80 (SYSCALL_VECTOR) 用作系统调用总入口之外，其他都用在外部硬件中断源上，其中包括可编程中断控制器 8259A 的 15 个 irq；事实上，当没有定义 CONFIG_X86_IO_APIC 时，其他 223(除 0x80 外)个中断向量，只利用了从 32 号开始的 15 个，其它 208 个空着未用。

这些中断服务程序入口的设置将在下面有详细说明。

一．相关数据结构

A. 中断描述符表 idt：也就是中断向量表，相当如一个数组，保存着各中断服务例程的入口。（详细描述参见图一、中断描述符格式）

B. 与硬中断相关数据结构：

与硬中断相关数据结构主要有三个：

一：定义在/arch/i386/kernel/irq.h 中的

```
struct hw_interrupt_type {
    const char * typename;
    void (*startup)(unsigned int irq);
    void (*shutdown)(unsigned int irq);
    void (*handle)(unsigned int irq, struct pt_regs * regs);
    void (*enable)(unsigned int irq);
    void (*disable)(unsigned int irq);
};
```

二：定义在/arch/i386/kernel/irq.h 中的

```
typedef struct {
    unsigned int status;          /* IRQ status - IRQ_INPROGRESS,
IRQ_DISABLED */
    struct hw_interrupt_type *handler; /* handle/enable/disable
functions */
    struct irqaction *action;      /* IRQ action list */
};
```

```

        unsigned int depth;                /* Disable depth for nested irq disables
*/
    } irq_desc_t;

```

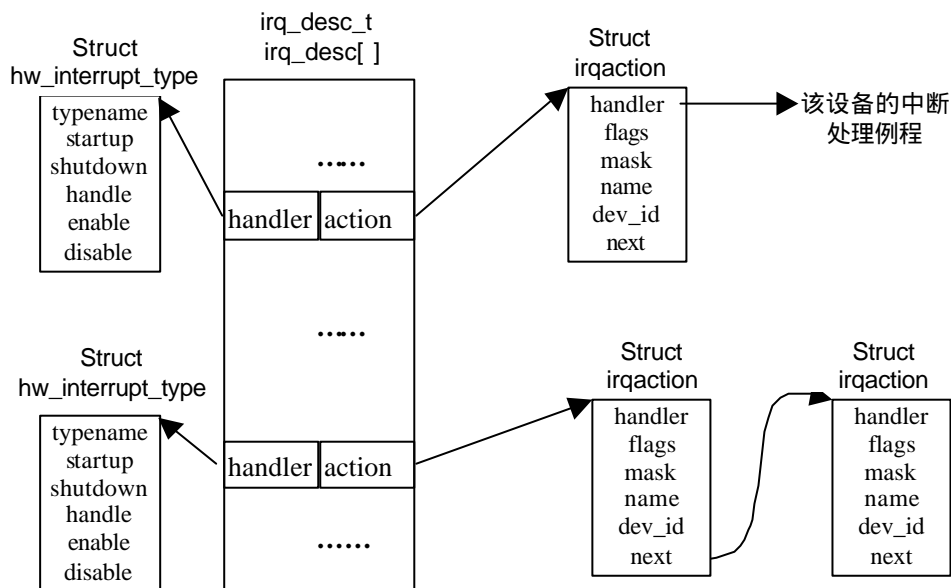
三：定义在 include/linux/ interrupt.h 中的

```

struct irqaction {
    void (*handler)(int, void *, struct pt_regs *);
    unsigned long flags;
    unsigned long mask;
    const char *name;
    void *dev_id;
    struct irqaction *next;
};

```

三者关系如下：



图四、与硬中断相关的几个数据结构各关系

各结构成员详述如下：

a) `struct irqaction` 结构,它包含了内核接收到特定 IRQ 之后应该采取的操作,其成员如下：

- ✍ `handler`：是一指向某个函数的指针。该函数就是所在结构对相应中断的处理函数。
- ✍ `flags`：取值只有 `SA_INTERRUPT` (中断可嵌套), `SA_SAMPLE_RANDOM` (这个中断是源于物理随机性的), 和 `SA_SHIRQ` (这个 IRQ 和其它 `struct irqaction` 共享)。
- ✍ `mask`：在 x86 或者体系结构无关的代码中不会使用 (除非将其设置为 0); 只有在 SPARC64 的移植版本中要跟踪有关软盘的信息时才会使用它。
- ✍ `name`：产生中断的硬件设备的名字。因为不止一个硬件可以共享一个 IRQ。
- ✍ `dev_id`：标识硬件类型的一个唯一的 ID。Linux 支持的所有硬件设备的每一种类

型，都有一个由制造厂商定义的在此成员中记录的设备 ID。

✎✎ next：如果 IRQ 是共享的，那么这就是指向队列中下一个 struct irqaction 结构的指针。通常情况下，IRQ 不是共享的，因此这个成员就为空。

b) struct hw_interrupt_type 结构，它是一个抽象的中断控制器。这包含一系列的指向函数的指针，这些函数处理控制器特有的操作：

✎✎ typename：控制器的名字。

✎✎ startup：允许从给定的控制器的 IRQ 所产生的事件。

✎✎ shutdown：禁止从给定的控制器的 IRQ 所产生的事件。

✎✎ handle：根据提供给该函数的 IRQ，处理唯一的中断。

✎✎ enable 和 disable：这两个函数基本上和 startup 和 shutdown 相同；

c) 另外一个数据结构是 irq_desc_t，它具有如下成员：

✎✎ status：一个整数。代表 IRQ 的状态：IRQ 是否被禁止了，有关 IRQ 的设备当前是否正被自动检测，等等。

✎✎ handler：指向 hw_interrupt_type 的指针。

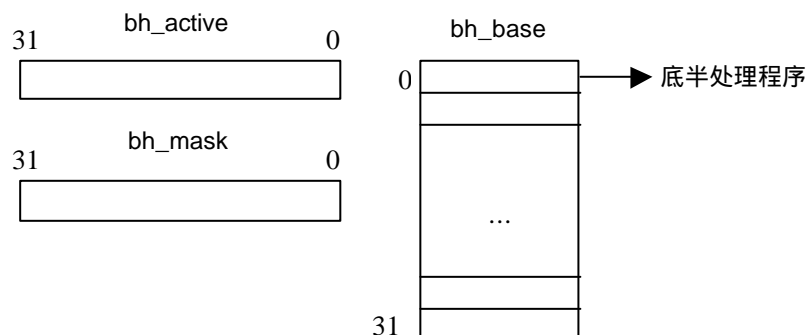
✎✎ action：指向 irqaction 结构组成的队列的头。正常情况下每个 IRQ 只有一个操作，因此链接列表的正常长度是 1（或者 0）。但是，如果 IRQ 被两个或者多个设备所共享，那么这个队列中就有多个操作。

✎✎ depth：irq_desc_t 的当前用户的个数。主要是用来保证在中断处理过程中 IRQ 不会被禁止。

irq_desc 是 irq_desc_t 类型的数组。对于每一个 IRQ 都有一个数组入口，即数组把每一个 IRQ 映射到和它相关的处理程序和 irq_desc_t 中的其它信息。

C. 与 Bottom_half 相关的数据结构：

图五、底半处理数据结构示意图



✎✎ bh_mask_count：计数器。对每个 enable/disable 请求嵌套对进行计数。这些请求通过调用 enable_bh 和 disable_bh 实现。每个禁止请求都增加计数器；每个使能请求都减小计数器。当计数器达到 0 时，所有未完成的禁止语句都已经被使能语句所匹配了，因此下半部分最终被重新使能。（定义在 kernel/softirq.c 中）

- ✎✎ `bh_mask` 和 `bh_active`——它们共同决定下半部分是否运行。它们两个都有 32 位，而每一个下半部分都占用一位。当一个上半部分（或者一些其它代码）决定其下半部分需要运行时，就通过设置 `bh_active` 中的一位来标记下半部分。不管是否做这样的标记，下半部分都可以通过清空 `bh_mask` 中的相关位来使之失效。因此，对 `bh_mask` 和 `bh_active` 进行位 AND 运算就能够表明应该运行哪一个下半部分。特别是如果位与运算的结果是 0，就没有下半部分需要运行。
- ✎✎ `bh_base`——是一组简单的指向下半部分处理函数的指针。

`bh_base` 代表的指针数组中可包含 32 个不同的底半处理程序。`bh_mask` 和 `bh_active` 的数据位分别代表对应的底半处理过程是否安装和激活。如果 `bh_mask` 的第 N 位为 1，则说明 `bh_base` 数组的第 N 个元素包含某个底半处理过程的地址；如果 `bh_active` 的第 N 位为 1，则说明必须由调度程序在适当的时候调用第 N 个底半处理过程。

二. 向量的设置和相关数据的初始化：

✎✎ 在实模式下的初始化过程中，通过对中断控制器 8259A-1, 9259A-2 重新编程，把硬中断设到 0x20-0x2F。即把 IRQ0—IRQ15 分别与 0x20-0x2F 号中断向量对应起来；当对应的 IRQ 发生了时，处理机就会通过相应的中断向量，把控制转到对应的中断服务例程。（源码在 `Arch/i386/boot/setup.S` 文件中；相关内容可参见 实模式下的初始化 部分）

✎✎ 在保护模式下的初始化过程中，设置并初始化 `idt`，共 256 个入口，服务程序均为 `ignore_int`，该服务程序仅打印 “Unknown interrupt\n”。（源码参见 `Arch/i386/KERNEL/head.S` 文件；相关内容可参见 保护模式下的初始化 部分）

✎✎ 在系统初始化完成后运行的第一个内核程序 `asmlinkage void __init start_kernel(void)`（源码在文件 `init/main.c` 中）中，通过调用 `void __init trap_init(void)` 函数，把各自陷和中断服务程序的入口地址设置到 `idt` 表中，即将表一中对应的处理程序入口设置到相应的中断向量表项中；在此版本（2.2.5）的 Linux 只设置 0-17 号中断向量。（`trap_init(void)` 函数定义在 `arch/i386/kernel/traps.c` 中；相关内容可参见 详解系统调用 部分）

✎✎ 在同一个函数 `void __init trap_init(void)` 中，通过调用函数 `set_system_gate(SYSCALL_VECTOR, &system_call)`；把系统调用总控程序的入口挂在中断 0x80 上。其中 `SYSCALL_VECTOR` 是定义在 `linux/arch/i386/kernel/irq.h` 中的一个常量 0x80；而 `system_call` 即为中断总控程序的入口地址；中断总控程序用汇编语言定义在 `arch/i386/kernel/entry.S` 中。（相关内容可参见 详解系统调用 部分）

✎✎ 在系统初始化完成后运行的第一个内核程序 `asmlinkage void __init start_kernel(void)`（源码在文件 `init/main.c` 中）中，通过调用 `void init_IRQ(void)` 函数，把地址标号 `interrupt[i]`（*i* 从 1-223）设置到 `idt` 表中的 32-255 号中断向量（0x80 除外），外部硬件 IRQ 的触发，将通过这些地址标号最终进入到各自相应的处理程序。（`init_IRQ(void)` 函数定义在 `arch/i386/kernel/IRQ.c` 中；）

✍ interrupt[i] (i 从 1-223), 是在 arch/i386/kernel/IRQ.c 文件中, 通过一系列嵌套的类似如 BUILD_16_IRQS(0x0) 的宏, 定义的一系列地址标号; (这些定义 interrupt[i] 的宏, 全部定义在文件 arch/i386/kernel/IRQ.c 和 arch/i386/kernel/IRQ.H 中。这些嵌套的宏的使用, 原理很简单, 但很烦, 限于篇幅, 在此省略)

✍ 各以 interrupt[i] 为入口的代码, 在进行一些简单的处理后, 最后都会调用函数 asmlinkage void do_IRQ(struct pt_regs regs), do_IRQ 函数调用 static void do_8259A_IRQ(unsigned int irq, struct pt_regs * regs) 而 do_8259A_IRQ 在进行必要的处理后, 将调用已与此 IRQ 建立联系 irqaction 中的处理函数, 以进行相应的中断处理。最后处理机将跳转到 ret_from_intr 进行必要处理后, 整个中断处理结束返回。(相关源码都在文件 arch/i386/kernel/IRQ.c 和 arch/i386/kernel/IRQ.H 中。Irqaction 结构参见上面的数据结构说明)

三. Bottom_half 处理机制

在此版本(2.2.5)的 Linux 中, 中断处理程序从概念上被分为上半部分 (top half) 和下半部分 (bottom half); 在中断发生时上半部分的处理过程立即执行, 但是下半部分 (如果有的话) 却推迟执行。内核把上半部分和下半部分作为独立的函数来处理, 上半部分决定其相关的下半部分是否需要执行。必须立即执行的部分必须位于上半部分, 而可以推迟的部分可能属于下半部分。

那么为什么这样划分成两个部分呢?

✍ 一个原因是要把中断的总延迟时间最小化。Linux 内核定义了两种类型的中断, 快速的和慢速的, 这两者之间的一个区别是慢速中断自身还可以被中断, 而快速中断则不能。因此, 当处理快速中断时, 如果有其它中断到达——不管是快速中断还是慢速中断——它们都必须等待。为了尽可能快地处理这些其它的中断, 内核就需要尽可能地将处理延迟到下半部分执行。

✍ 另外一个原因是, 当内核执行上半部分时, 正在服务的这个特殊 IRQ 将会被可编程中断控制器禁止, 于是, 连接在同一个 IRQ 上的其它设备就只有等到该该中断处理被处理完毕后果才能发出 IRQ 请求。而采用 Bottom_half 机制后, 不需要立即处理的部分就可以放在下半部分处理, 从而, 加快了处理机对外部设备的中断请求的响应速度。

✍ 还有一个原因就是, 处理程序的下半部分还可以包含一些并非每次中断都必须处理的操作; 对这些操作, 内核可以在一系列设备中断之后集中处理一次就可以了。即在这种情况下, 每次都执行并非必要的操作完全是一种浪费, 而采用 Bottom_half 机制后, 可以稍稍延迟并在后来只执行一次就行了。

由此可见, 没有必要每次中断都调用下半部分; 只有 bh_mask 和 bh_active 的对应位的与为 1 时, 才必须执行下半部分(do_botoom_half)。所以, 如果在上半部分中 (也可能在其他地方) 决定必须执行对应的半部分, 那么可以通过设置 bh_active 的对应位, 来指明下半部分必须执行。当然, 如果 bh_active 的对应位被置位, 也不一定会马上执行下半部分, 因为还必须具备另外两个条件: 首先是 bh_mask 的相应位也必须被置位, 另外, 就是处理的时机, 如果下半部分已经标记过需要执行了, 现在又再次标记, 那么内核就简单地保持这个标记; 当情况允许的时候, 内核就对它进行处理。如果在内核有机会运行其下半部分之前给定的设备就已经发生了 100 次中断, 那么内核的上半部

分就运行 100 次，下半部分运行 1 次。

bh_base 数组的索引是静态定义的，定时器底半处理过程的地址保存在第 0 个元素中，控制台底半处理过程的地址保存在第 1 个元素中，等等。当 bh_mask 和 bh_active 表明第 N 个底半处理过程已被安装且处于活动状态，则调度程序会调用第 N 个底半处理过程，该底半处理过程最终会处理与之相关的任务队列中的各个任务。因为调度程序从第 0 个元素开始依次检查每个底半处理过程，因此，第 0 个底半处理过程具有最高的优先级，第 31 个底半处理过程的优先级最低。

内核中的某些底半处理过程是和特定设备相关的，而其他一些则更一般一些。表二 列出了内核中通用的底半处理过程。

表二、Linux 中通用的底半处理过程

TIMER_BH (定时器)	在每次系统的周期性定时器中断中，该底半处理过程被标记为活动状态，并用来驱动内核的定时器队列机制。
CONSOLE_BH (控制台)	该处理过程用来处理控制台消息。
TQUEUE_BH (TTY 消息队列)	该处理过程用来处理 tty 消息。
NET_BH (网络)	用于一般网络处理，作为网络层的一部分
IMMEDIATE_BH (立即)	这是一个一般性处理过程，许多设备驱动程序利用该过程对自己要在随后处理的任务进行排队。

当某个设备驱动程序，或内核的其他部分需要将任务排队进行处理时，它将任务添加到适当的系统队列中（例如，添加到系统的定时器队列中），然后通知内核，表明需要进行底半处理。为了通知内核，只需将 bh_active 的相应数据位置为 1。例如，如果驱动程序在 immediate 队列中将某任务排队，并希望运行 IMMEDIATE 底半处理过程来处理排队任务，则只需将 bh_active 的第 8 位置为 1。在每个系统调用结束并返回调用进程之前，调度程序要检验 bh_active 中的每个位，如果有任何一位为 1，则相应的底半处理过程被调用。每个底半处理过程被调用时，bh_active 中的相应位被清除。bh_active 中的置位只是暂时的，在两次调用调度程序之间 bh_active 的值才有意义，如果 bh_active 中没有置位，则不需要调用任何底半处理过程。

四．中断处理全过程

由前面的分析可知，对于 0-31 号中断向量，被保留用来处理异常事件；0x80 中断向量用来作为系统调用的总入口点；而其他中断向量，则用来处理外部设备中断；这三者的处理过程都是不一样的。

一、 异常的处理全过程

对这 0-31 号中断向量，保留用来处理异常事件；操作系统提供相应的异常的处理程序，并在初始化时把处理程序的入口等级在对应的中断向量表项中。当产生一个异常时，处理机就会自动把控制转移到相应的处理程序的入口，运行相应的处理程序，

进行相应的处理后，返回原中断处。当然，在前面已经提到，此版本（2.2.5）的 Linux 只提供了 0-17 号中断向量的处理程序。

二、 中断的处理全过程

对于 0-31 号和 0x80 之外的中断向量，主要用来处理外部设备中断；在系统完成初始化后，其中断处理过程如下：

当外部设备需要处理机进行中断服务时，它就会通过中断控制器要求处理机进行中断服务。如果 CPU 这时可以处理中断，CPU 将根据中断控制器提供的中断向量号和中断描述符表（IDT）中的登记的地址信息，自动跳转到相应的 `interrupt[i]` 地址；在进行一些简单的但必要的处理后，最后都会调用函数 `do_IRQ`，`do_IRQ` 函数调用 `do_8259A_IRQ` 而 `do_8259A_IRQ` 在进行必要的处理后，将调用已与此 IRQ 建立联系 `irqaction` 中的处理函数，以进行相应的中断处理。最后处理机将跳转到 `ret_from_intr` 进行必要处理后，整个中断处理结束返回。

从数据结构入手，应该说是分析操作系统源码最常用的和最主要的方法。因为操作系统的几大功能部件，如进程管理，设备管理，内存管理等等，都可以通过对其相应的数据结构的分析来弄懂其实现机制。很好的掌握这种方法，对分析 Linux 内核大有裨益。

方法之四、以功能为中心，各个击破

从功能上看，整个 Linux 系统可看作有以下几个部分组成：

1. 进程管理机制部分；
2. 内存管理机制部分；
3. 文件系统部分；
4. 硬件驱动部分；
5. 系统调用部分等；

以功能为中心、各个击破，就是指从这五个功能入手，通过源码分析，找出 Linux 是怎样实现这些功能的。

在这五个功能部件中，系统调用是用户程序或操作调用核心所提供的功能的接口；也是分析 Linux 内核源码几个很好的入口点之一。对于那些在 dos 或 Unix、Linux 下有 C 编程经验的高手尤其如此。又由于系统调用相对其它功能而言，较为简单，所以，我就以它为例，希望通过对系统调用的分析，能使读者体会到这一方法。

I、系统调用的初始化设置和相关数据结构

与系统调用相关的内容主要有：系统调用总控程序，系统调用向量表 `sys_call_table`，以及各系统调用服务程序。下面将对此一一介绍：

1. 保护模式下的初始化过程中，设置并初始化 `idt`，共 256 个入口，服务程序均为 `ignore_int`，该服务程序仅打印 “Unknown interrupt\n”。（源码参见 `/Arch/i386/KERNEL/head.S` 文件；相关内容可参见 [保护模式下的初始化](#) 部分）

2. 在系统初始化完成后运行的第一个内核程序 `start_kernel` 中，通过调用 `trap_init` 函数，把各自陷和中断服务程序的入口地址设置到 `idt` 表中；同时，此函数还通过调用函数 `set_system_gate` 把系统调用总控程序的入口地址挂在中断 0x80 上。其中：

✎ `start_kernel` 的原型为 `void __init start_kernel(void)`，其源码在文件 `init/main.c` 中；

✎ `trap_init` 函数的原型为 `void __init trap_init(void)`，定义在 `arch/i386/kernel/traps.c` 中

✎ 函数 `set_system_gate` 同样定义在 `arch/i386/kernel/traps.c` 中，调用原型为 `set_system_gate(SYSCALL_VECTOR, &system_call)`；

✎ 其中，`SYSCALL_VECTOR` 是定义在 `linux/arch/i386/kernel/irq.h` 中的一个常量 0x80；

✎ 而 `system_call` 即为系统调用总控程序的入口地址；中断总控程序用汇编语言定义在 `arch/i386/kernel/entry.S` 中。

（其它相关内容可参见 [中断和中断处理](#) 部分）

3. 系统调用向量表 `sys_call_table`，是一个含有 `NR_syscalls=256` 个单元的数组。它的每个单元存放着一个系统调用服务程序的入口地址。该数组定

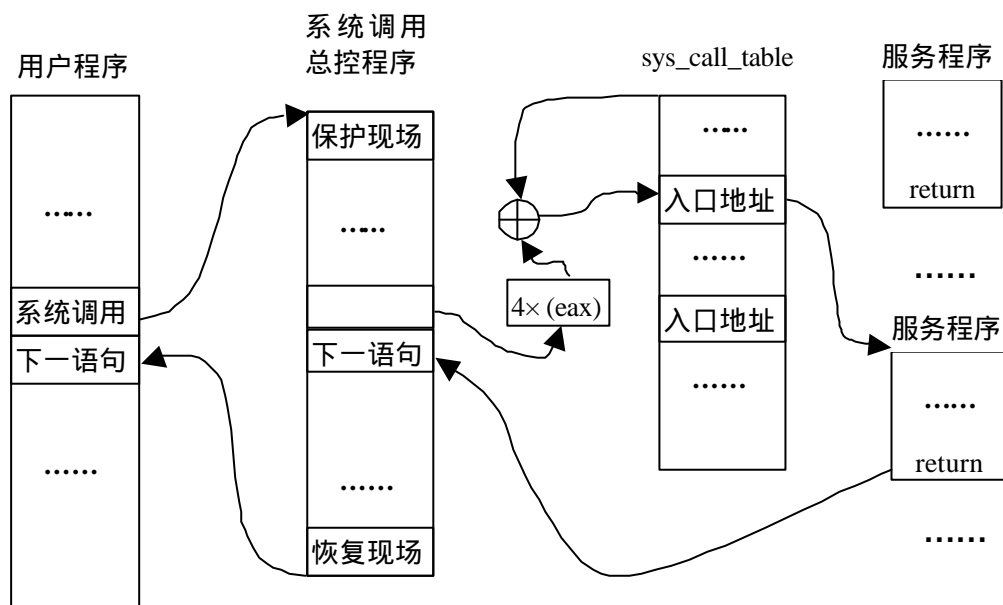
义在 `/arch/i386/kernel/entry.S` 中；而 `NR_syscalls` 则是一个等于 256 的宏，定义在 `include/linux/sys.h` 中。

4. 各系统调用服务程序则分别定义在各个模块的相应文件中；例如 `asmlinkage int sys_time(int * tloc)` 就定义在 `kernel/time.c` 中；另外，在 `kernel/sys.c` 中也有不少服务程序；

11、系统调用过程

我们知道，系统调用是用户程序或操作调用核心所提供的功能的接口；所以系统调用的过程就是从用户程序到系统内核，然后又回到用户程序的过程；在 Linux 中，此过程大体过程可描述如下：

系统调用过程示意图




注：此处语句为：`call *SYMBOL_NAME(sys_call_table)(,%eax,4)`；`eax` 中为系统调用号



整个系统调用进入过程客表示如下：


用户程序 \rightarrow 系统调用总控程序(system_call) \rightarrow 各个服务程序

可见，系统调用的进入课分为“用户程序 \rightarrow 系统调用总控程序”和“系统调用总控程序 \rightarrow 各个服务程序”两部分；下边将分别对这两个部分进行详细说明：

“用户程序 \rightarrow 系统调用总控程序”的实现：在前面已经说过，Linux 的系统调用使用第 0x80 号中断向量项作为总的入口，也即，系统调用总控程序的入口地

址 `system_call` 就挂在中断 `0x80` 上。也就是说,只要用户程序执行 `0x80` 中断 (`int 0x80`), 就可实现“用户程序  系统调用总控程序”的进入;事实上,在 Linux 中,也是这么做的。只是 `0x80` 中断的执行语句 `int 0x80` 被封装在标准 C 库中,用户程序只需用标准系统调用函数就可以了,而不需要在用户程序中直接写 `0x80` 中断的执行语句 `int 0x80`。至于中断的进入的详细过程可参见前面的“中断和中断处理”部分。

 “系统调用总控程序  各个服务程序”的实现:在系统调用总控程序中通过语句“`call * SYMBOL_NAME(sys_call_table)(,%eax,4)`”来调用各个服务程序 (`SYMBOL_NAME` 是定义在 `/include/linux/linkage.h` 中的宏 `#define SYMBOL_NAME_LABEL(X)`), 可以忽略)。当系统调用总控程序执行到此语句时, `eax` 中的内容即是相应系统调用的编号,此编号即为相应服务程序在系统调用向量表 `sys_call_table` 中的编号 (关于系统调用的编号说明在 `/linux/include/asm/unistd.h` 中)。又因为系统调用向量表 `sys_call_table` 每项占 4 个字节,所以由 `%eax` 乘上 4 形成偏移地址,而 `sys_call_table` 则为基址;基址加上偏移所指向的内容就是相应系统调用服务程序的入口地址。所以此 `call` 语句就相当于直接调用对应的系统调用服务程序。

 参数传递的实现:在 Linux 中所有系统调用服务例程都使用了 `asmlinkage` 标志。此标志是一个定义在 `/include/linux/linkage.h` 中的一个宏:

```
#if defined __i386__ && (__GNUC__ > 2 || __GNUC_MINOR__ > 7)
#define asmlinkage CPP_ASMLINKAGE__attribute__((regparm(0)))
#else
#define asmlinkage CPP_ASMLINKAGE
#endif
```


其中涉及到了 gcc 的一些约定,总之,这个标志它可以告诉编译器该函数不需要从寄存器中获得任何参数,而是从堆栈中取得参数;即参数在堆栈中传递,而不是直接通过寄存器;

堆栈参数如下:

EBX	= 0x00
ECX	= 0x04
EDX	= 0x08
ESI	= 0x0C
EDI	= 0x10
EBP	= 0x14
EAX	= 0x18
DS	= 0x1C
ES	= 0x20
ORIG_EAX	= 0x24
EIP	= 0x28
CS	= 0x2C
EFLAGS	= 0x30

在进入系统调用总控程序前,用户按照以上的对应顺序将参数放到对应寄存器中,在系统调用总控程序一开始就将这些寄存器压入堆栈;在退出总控程序前又按如上顺序堆栈;用户程序则可以直接从寄存器中复得被服务程序加工过了的参数。而对于系统调用服务程序而言,参数就可以直接从总控程序压入的堆栈中复得;对参数的

修改一可以直接在堆栈中进行；其实，这就是 `asm linkage` 标志的作用。所以在进入和退出系统调用总控程序时，“保护现场”和“恢复现场”的内容并不一定会相同。

 **特殊的服务程序：**在此版本(2.2.5)的 Linux 内核中，有好几个系统调用的服务程序都是定义在 `/usr/src/linux/kernel/sys.c` 中的同一个函数：

```
asm linkage int sys_ni_syscall(void)

{

    return -ENOSYS;

}
```

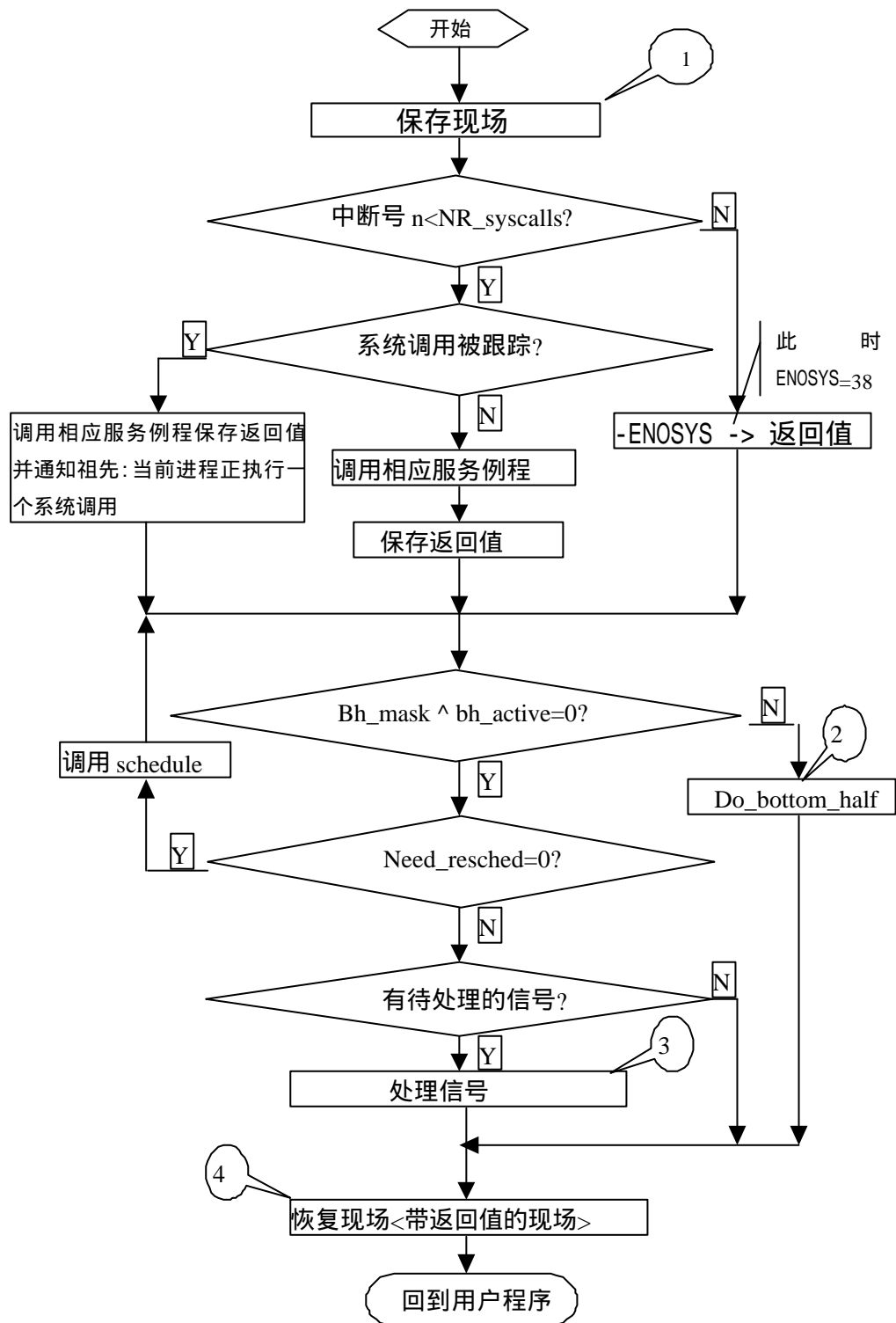
此函数除了返回错误号之外，什么都没干。那他有什么作用呢？归结起来有如下三种可能：

1. 处理边界错误，0 号系统调用就是用的此特殊的服务程序；
2. 用来替换旧的已淘汰了的系统调用，如：Nr 17, Nr 31, Nr 32, Nr 35, Nr 44, Nr 53, Nr 56, Nr 58, Nr 98；
3. 用于将要扩展的系统调用，如：Nr 137, Nr 188, Nr 189；

III、系统调用总控程序(system_call)

系统调用总控程序(system_call)可参见 `arch/i386/kernel/entry.S` 其执行流程如下图：

系统调用总控程序(system_call) 流程图



注： 2 和 3 处都有关于 V86 的处理，在此流程图中都省略了；

1 和 4 处的寄存器值不一定相等，因为系统调用的入口参数是通过栈传递的，即总控程序通过“恢复现场”大修改过了的参数又传递给用户程序；

IV、实例：增加一个系统调用

由以上的分析可知，增加系统调用由于下两种方法：

i . 编一个新的服务例程，将它的入口地址加入到 `sys_call_table` 的某一项，只要该项的原服务例程是 `sys_ni_syscall` 并且是 `sys_ni_syscall` 的作用属于第三种的项，也即 Nr 137, Nr 188, Nr 189。

ii . 直接增加：

1. 编一个新的服务例程；
2. 在 `sys_call_table` 中添加一个新项，并把的新增加的服务例程的入口地址加到 `sys_call_table` 表中的新项中；
3. 把增加的 `sys_call_table` 表项所对应的向量，在 `include/asm-386/unistd.h` 中进行必要申明，以供用户进程和其他系统进程查询或调用。
4. 由于在标准的 c 语言库中没有新系统调用的承接段，所以，在测试程序中，除了要 `#include<linux/unistd.h>`，还要申明如下 `_syscall1(int,additionSysCall,int, num)`。

下面将对第 ii 种情况列举一个我曾经实现过了的一个增加系统调用的实例：

1.) 在 `kernel/sys.c` 中增加新的系统服务例程如下：

```
asmlinkage int sys_addtotal(int numdata)
{
    int i=0, enddata=0;

    while(i<=numdata)

        enddata+=i++;

    return enddata;
}
```

该函数有一个 `int` 型入口参数 `numdata`，并返回从 0 到 `numdata` 的累加值；当然也可以把系统服务例程放在一个自己定义的文件或其他文件中，只是要在相应文件中作必要的说明；

2.) 把 `asmlinkage int sys_addtotal(int)` 的入口地址加到 `sys_call_table` 表中：

`arch/i386/kernel/entry.S` 中的最后几行源代码修改前为：

```

...    ...

.long SYMBOL_NAME(sys_sendfile)

.long SYMBOL_NAME(sys_ni_syscall)    /* streams1 */

.long SYMBOL_NAME(sys_ni_syscall)    /* streams2 */

.long SYMBOL_NAME(sys_vfork)          /* 190 */

.rept NR_syscalls-190

    .long SYMBOL_NAME(sys_ni_syscall)

.endr

```

修改后为:

```

.long SYMBOL_NAME(sys_sendfile)

.long SYMBOL_NAME(sys_ni_syscall)    /* streams1 */

.long SYMBOL_NAME(sys_ni_syscall)    /* streams2 */

.long SYMBOL_NAME(sys_vfork)          /* 190 */

/* add by I */

.long SYMBOL_NAME(sys_addtotal)

.rept NR_syscalls-191

    .long SYMBOL_NAME(sys_ni_syscall)

.endr

```

3.) 把增加的 sys_call_table 表项所对应的向量,在 include/asm-386/unistd.h 中进行必要申明,以供用户进程和其他系统进程查询或调用:

增加后的部分 /usr/src/linux/include/asm-386/unistd.h 文件如下:

```

...    ...

#define __NR_sendfile    187

```



```
    #define __NR_getpmsg 188

    #define __NR_putpmsg 189

    #define __NR_vfork 190

/* add by I */

    #define __NR_addtotal 191
```

4. 测试程序(test.c)如下:

```
#include<linux/unistd.h>

#include<stdio.h>

_syscall1(int,addtotal,int, num)

main()
{
    int i,j;

    do

        printf("Please input a number\n");
    while(scanf("%d",&i)==EOF);

    if((j=addtotal(i))==-1)

        printf("Error occurred in syscall-addtotal();\n");

    printf("Total from 0 to %d is %d \n",i,j);

}
```

对修改后的新的内核进行编译，并引导它作为新的操作系统，运行几个程序后可以发现一切正常；在新的系统下对测试程序进行编译(*注：由于原内核并未提供此系统调用，所以只有在编译后的新内核下，此测试程序才能可能被编译通过)，运行情况如下：

```
$gcc -o test test.c

$./test

Please input a number

36

Total from 0 to 36 is 666
```

可见，修改成功。

综上所述，由于操作系统内核源码的特殊性：体系庞大，结构复杂，代码冗长，代码间联系错综复杂。所以要把内核源码分析清楚，也是一个很艰难，很需要毅力的事。尤其需要交流和讲究方法；只有方法正确，才能事半功倍。

在上面的论述中，一共列举了两个内核分析的入口、和三种分析源码的方法：以程序流程为线索，一线串珠；以数据结构为基点，触类旁通；以功能为中心，各个击破。三种方法各有特点，适合于分析不同部分的代码：

1. 以程序流程为线索，适合于分析系统的初始化过程：系统引导、实模式下的初始化、保护模式下的初始化三个部分，和分析应用程序的执行流程：从程序的装载，到运行，一直到程序的退出。而流程图则是这种分析方法最合适的表达工具。

2. 以数据结构为基点、触类旁通，这种方法是分析操作系统源码最常用的和最主要的方法。对分析进程管理，设备管理，内存管理等等都是很有效的。

3. 以功能为中心、各个击破，是把整个系统分成几个相对独立的功能模块，然后分别对各个功能进行分析。这样带来的一个好处就是，每次只以一个功能为中心，涉及到其他部分的内容，可以看作是其它功能提供的服务，而无需急着追究这种服务的实现细节；这样，在很大程度上减轻了分析的复杂度。

三种方法，各有其长，只要合理的综合运用这些方法，相信对减轻分析的复杂度还是有所帮组的。