

## 内容简介

本书以 Linux 1.0 核心为基础，详细注释了从开机启动到其正常运转的全过程，并且还给出了理解这个过程所需要的基础知识。

本书适合于所有 Linux 操作系统的爱好者。

# 前言

献给和我一样的计算机菜鸟们

献给和我一样的 Linux 爱好者们

为中国的 Linux 事业做点微薄的贡献

先谈一谈我个人的学习经历，为初学者打气（当然我还是菜鸟）。学习核心并不需要高学历，高智商，只要勤奋。正所谓世上无难事，只怕有心人。

我从 99 年 9 月份开始接触计算机（大一），到现在算起来也有好几个年头了。至今还记得第一次上电脑课闹出的笑话（因为我们是非计算机专业的，以前根本没有接触过它，只是听过）。那就是进入机房后看到一位同学在键盘上噼里啪啦的胡乱的敲击着。我忍不住的问了一句，键盘麻手吗（谓之带电吗）。引起全班哄堂大笑。

当时班上有位同学（后来才知道，他哥是学计算机的）懂点 DOS 命令（被我们奉若神明，同学们常买东西犒劳他，因为要玩游戏），也不知道从那搞来的“超级马力”的游戏（也就是这个游戏，我们才觉得上电脑课才有点意思）。只有他懂的如何运行该游戏！说到这大家可能有点不信，都 99 年了，还用什么 DOS 命令啊！唉！谁叫咱们是非计算机专业的，我们只能用用 DOS 和 WPS（计算机课都成了打字课了，可惜我到最后还是没有把五笔的字根背下来），装有 Windows98 的机器只能给计算机专业的用，除非自己花钱买票上机。（不过，现在想起来应该给我们用才对啊！真不知道学校老师是怎么想的）

我们学的是编程语言是 True Basic（第一节课，老师就说了句打击我们的话：“这种语言早就被淘汰了 10 年以上了”，以致到该门课程结束后大家都不知道学了什么，稀里糊涂考完试了事，因为同学们学的都没有兴趣了！淘汰 10 年的东西还有什么好学的。老师的这句实在话现在看来还真的是“误人子弟”）。我学了半学期下来，感觉什么都不会。于是开始去机房揩油（趁人家有上机课时，偷偷溜进机房，也有被撵出来的时候）。到最后还是什么都不会，就是打字速度稍微快了点。

到了 2000 年，新学期开学的时候，我们系里买了 20 台左右的电脑（现在我还记得配置：CPU C433，内存 64M，显卡 8M），开始教授 AutoCAD 课程。这个时候我也在考虑我毕业后所要从事的职业了（该玩的我大一已经玩了，所以建议过了大一后，各位兄弟姐妹们就不要玩了吧“考研的高三就不玩了”）。把时间用在学习上，毕业后找工作的好处我深有体会。第一个想法是肯定不去干机械（因为在做金工实习时，锯一块铁料已经够我受的了，后来才知道真正到厂里不是这样的，厂里有切割工具的，因为后来我们去参观好几个大型机械厂的）。于是就萌发了自学计算机的想法，所以我真正开始学习语言的时间是从 2000 年的下半年开始的。买了本谭浩强老师的《C 程序设计》，并做完了和其配套的习题册，习题册中的每个程序我都调试过（开始用的调试工具是 TC2.0，后来改用 VC6.0。就学习的效率来说我推荐用 VC6.0 来调试，当然现在都已经有 Visual studio .NET 2005 了，调试功能应该更强大了）。就这样搞了半个学期，终于过了语言关。接下来又开始学习严蔚敏，吴伟民两位老师的《数据结构（C 语言版）》，并也做了其对应的习题。感觉有些功力了，于是主动给系里做了些小程序（老师教学用的，也为我后来管理系机房铺平了的道路）后来陆续学习了 C 十十，汇编（不过学的不到位）及 MFC，为了能写个在手机上跑的程序，JAVA 也学了点，

因为当时我感觉手机上跑的程序比较好玩，最终用 J2ME 写了个运行在 Nokia 上的《英语 900 句》算是结束了，没有坚持下去。（现在想想有点后悔，要是坚持下来多好，手机游戏也是个大产业哦，所以在校的学生可以考虑）

接触 Linux 也是在 2000 年，当时看到网上到处都在讲它。非常好奇（因为，我们学校买了台 Sun 的机器，居然没有老师会配置，“当时的老师咋就没有想到搜索互联网呢？”最后还是从南京某个学校请的人过来配的。那是我第一次看到类 Unix 操作系统，给我的印象是很神秘，不像 Windows 来的直观，不知大家是否也有这个想法）。于是从网络上下载了个 Redhat6.2，搞了好几个星期还是安装不了（现在想想搞个虚拟机多简单，那时为什么就想不到呢？也不知道搜索互联网！“当时已经有 Yahoo 了”）。于是决定买张光盘安装（花了 8 块钱），可是系里的机器没有光驱，怎么办呢！只好在下课后不走（我说的是晚自习后）。待夜深人静的时候偷偷把老师机器的光驱拆了用。就这样搞了一个通宵（这也是我熬过的唯一的一个通宵），终于装好了。（不得不说的是，不知是紧张还是什么原因，重新把光驱装回老师机器时，居然把电源线给忘接了。露馅了。不过还好老师什么也没有说，只是叫我帮它重新装了一下，他知道肯定是我干的。我一直都很感激这个老师，他给我的帮助也比较大，指导我该看那些书，偶尔也会请我去吃顿饭。）

写到这我顺便说一说在学校时我和老师的关系（因为我现在看来上学时和老师搞好关系真的是非常重要的）。当时系里的老师对我都蛮照顾的，都认识我。所以有些课我认为可听可不听时就不去了（翘课估计每个人都有过吧，不鼓励翘课），他们也自然不会追问了。学分照给。其他同学就惨了。重点说一下当时我们机械系的系主任，现在恐怕已经做上副校长了。给了我看管系机房的机会，另外每个月还给 200 块的补贴。这个最实际了 ☺——还在读书的朋友一定要深刻体会这点哦。

为了能够看懂核心，需要的知识非常多，还要深入体会这些知识（以 Intel X86 CPU 来说吧，它分为实模式，保护模式，并且在不同的模式下寻址方式也不同，拿着书看都知道，丢了书什么忘了，所以要靠自己体会的）。所以当时学校图书馆关于计算机的书基本都被我看我过了（符合要求的），搞到最后图书馆的老师们都认识我了。最高记录是一学期借了 76 本书，这还是我们班同学告诉我的。

然后在 2002 年时候开始看核心，前前后后看了很多个版本（很泛的看）。刚开始真的很痛苦，看看停停（我想大部分初学者都有这种感觉，过了这道槛就 ok 了）因为看不懂。核心不像一般的应用程序那样，它和硬件结合的比较紧密（如果是学硬件的人，看核心可能会更轻松点）。后来关于核心分析的书籍渐渐多起来了，比如陈莉君的《深入分析 Linux 内核源代码》，也为我读核心提供不小的帮助。到了 2003 年的时候，我在网上看到了赵炯博士的《LINUX 内核完全注释》电子版以及浙江大学毛德操和胡希明的《LINUX 内核源代码情景分析》，才看的快了一点（互联网真是好东西，大家好好利用）。在看了后也就萌发了写点东西的想法，于是便有了它。

该书分 3 章：

第一章为基础知识，概述了理解核心所需要的软件及硬件知识。这也是我们在看核心时，首先应掌握的知识。该章里提到的知识（可能还有其他的重要内容没有加入，如果有要加入的可发 mail 给我，谢谢！☺），毫不夸张的说每一个展开来都是一本甚至几本书的内容，在这里我只是给大家做了一个提纲式的总结。省的读者还要先研究这些知识！当然深入的话题还得靠大家去查看消化，另外该章中提到的硬件知识有些来源于互联网。

第二章为代码分析，也是本书的主角。从我们给计算机加电启动介绍起，直到核心正常运作为止。其中涉及了好多个文件中的函数（有很少数几个的函数，我没有做出注释，还请各位见谅。因为我的主业是在 Windows 平台上做开发的“如果能给我个 Linux 平台上开发

的机会会更好 😊”，所以还有其他工作要做。分析只能在晚上做 1 到 2 个小时，因为我不熬夜的 😊)。我都一一注释，并在调用某个函数时，给出了这个被调用函数的所在的文件位置，从而方便读者查找，为你节省时间。

第三章为其他的话题，内容比较少，原因可能是我没有在 Linux 下做过开发吧，所以不知道大家需要什么，而我要写些什么（等我买了开发板后，我把嵌入式开发补上，哦我的银子啊 🍑）。其中包括了模块的编写，动态及静态添加修改系统调用，动态及静态函数库的编写。

本书的不足：

目前讲的只是从计算机启动到核心真正运作起来的整个过程。其中并没有涉及到核心中数据结构的分析。因为 1.0 太大了，对我个人而言（如果有感兴趣的朋友的话，我们可以一起做分析 😊）。

另外在我对代码的注释过程中肯定会犯有不少的错误，也许是用了错别字，也许是对我对代码的理解不正确从而导致错误的注释，等等！读者在发现上述问题后请毫不犹豫的并且毫不留情的给我指出来，再次感谢！

你们的朋友

[郭大海](#)

2005 年 11 月

# 源代码

本书样例源代码 

Linux1.0 核心源代码 

# 目 录

第一部分 基础知识 (Basic knowledge) .....	11
软件部分 (Software part) .....	12
S1.Makefile简介 .....	12
S1. 1Makefile规则.....	12
S2.汇编简介 .....	17
S2. 1 汇编优缺点.....	17
S2. 2 汇编语法 (AT&T asm VS Intel asm) .....	18
S2. 3 Hello world!示例.....	20
S3.实模式向保护模式切换.....	21
S3.1 切换到保护方式的准备工作.....	21
S3.2 使用段间指令切换进保护模式.....	22
S3.3 打开A20 地址线切换进保护模式.....	23
S4.gcc内嵌汇编 .....	25
S4.1 内嵌汇编格式.....	25
S4.2 内嵌汇编示例.....	26
S5.GDB调试器 .....	28
S5.1 GDB命令 .....	29
S5.2 GDB调试样例 .....	29
S6.系统调用实现详解.....	34
S6.1 核心中提供的宏.....	34
S6.2 系统调用编号.....	37
S6.3 系统调用入口点及函数表.....	40
S6.4 对系统调用调用.....	41
硬件部分 (Hardware part) .....	44
H1.操作系统的引导 .....	44
H1.1 BIOS的工作.....	44
H1.2 操作系统的引导块程序.....	44
H2. X86 CPU 寻址简介 .....	47
H2.1 实模式.....	47
H2.2 实模式方式下物理地址的形成.....	48
H2.3 保护模式.....	49
H2.4 保护模式方式下物理地址的形成.....	49
H3. IDT & GDT & LDT .....	51
H3.1 IDT .....	51
H3.2 GDT & LDT .....	52
H4.8259A可编程中断控制器 .....	53
H4.1 8259A芯片简介 .....	53
H4.2 8259A芯片对的中断处理过程 .....	54
H4.3 8259A编程方式 .....	55
H5.I/O端口及指令 .....	61
H5.1I/O端口 .....	61

H5.2I/O指令 .....	61
H6.获取系统时间 .....	62
H6.1CMOS RAM分配表 .....	62
H6.2 读取CMOS RAM表.....	63
H6.3Linux获取读取CMOS RAM的方式 .....	64
第二部分     代码分析 (Code analyzed) .....	66
引爆点.....	67
整个核心工程Makefile .....	67
zBoot/Makefile.....	76
总结.....	77
内存布局图.....	79
B .....	79
Boot/bootsect.S .....	79
概述.....	79
代码分析.....	81
Boot/setup.S .....	94
概述.....	94
代码分析.....	95
Boot/head.S .....	121
概述.....	121
代码分析.....	122
D .....	134
Drivers/char/console.c (部分代码) .....	134
Drivers/char/serial.c (部分代码) .....	147
Drivers/char/keyboard.c (部分代码) .....	154
Drivers/char/tty_io.c (部分代码) .....	156
Drivers/char/mem.c (部分代码) .....	157
Drivers/block/floppy.c (部分代码) .....	158
Drivers/block/ramdisk.c (部分代码) .....	162
Drivers/block/hd.c .....	165
Drivers/block/genhd.c (部分代码) .....	168
Drivers/block/blk.c (部分代码) .....	173
Drivers/block/xd.c (部分代码) .....	173
Drivers/block/ramdisk.c (部分代码) .....	176
Drivers/block/l1_rw_blk.c (部分代码) .....	177
Devices/net/lance.c (部分代码) .....	185
Devices/net/net_init.c (部分代码) .....	186
F .....	187
Fs/fcntl.c (部分代码) .....	187
Fs/exec.c (部分代码) .....	188
Fs/file_table.c (部分代码) .....	209
Fs/namei.c .....	210
Fs/buffer.c (部分代码) .....	229
Fs/super.c.....	238

Fs/file_table.c (部分代码) .....	252
Fs/inode.c (部分代码) .....	253
Fs/locks.c (部分代码) .....	258
Fs/open.c (部分代码) .....	260
Fs/devices.c (部分代码) .....	264
Fs/minix/inode.c (部分代码) .....	265
I .....	268
Init/main.c .....	268
概述.....	268
代码分析.....	268
Ipc/shm.c (部分代码) .....	286
Ipc/sem.c (部分代码) .....	288
Ibcs/emulate.c .....	290
Include/linux/unistd.h.....	291
Include/linux/sched.h .....	298
K.....	303
Kernel/panic.c .....	303
Kernel/traps.c .....	304
Kernel/irq.c (部分代码) .....	305
Kernel/time.c.....	308
Kernel/sched.c.....	310
Kernel/exit.c(部分代码) .....	330
Kernel/signal.c .....	336
Kernel/printk.c (部分代码) .....	340
Kernel/vsprintf.c (部分代码) .....	343
Kernel/fork.c (部分代码) .....	344
Kernel/sys_call.s .....	350
L .....	362
Lib/_exit.c .....	362
Lib/open.c .....	362
M .....	363
Mm/vmalloc.c (部分代码) .....	363
Mm/kmalloc.c (部分代码) .....	366
Mm/swap.c (部分代码) .....	367
Mm/memory.c (部分代码) .....	372
N.....	383
Net/unix/sock.c (部分代码) .....	383
Net/space.c (部分代码) .....	385
Net/ddi.c (部分代码) .....	385
Net/socket.c (部分代码) .....	386
Z .....	388
zBoot/head.S .....	388
概述.....	388
代码分析.....	388

---

核心游记总结（1.0 核心） .....	391
第三部分 其他话题（Advanced part） .....	395
A1.模块的编写 .....	396
A1-1 模块代码及分析 .....	396
A1-2 模块的加载、注销及查看 .....	398
A2.系统调用的添加 .....	400
A2-1 静态添加系统调用 .....	400
A2-1-1 讨论Linux系统调用的体系 .....	400
A2-1-2 修改代码来添加系统调用 .....	405
A2-2 动态添加系统调用 .....	406
A2-2-1 动态添加系统调用的原理 .....	407
A2-2-2 实现动态添加、修改系统调用 .....	408
A2-2-3 反汇编capturemod.o并分析之 .....	414
A3.函数库的编写 .....	419
A3-1 静态函数库的编写 .....	419
A3-1-1 包含算法的各个文件及Makefile .....	420
A3-1-2 测试静态函数库的程序及Makefile .....	422
A3-1-3 静态库编译情况 .....	423
A3-1-3 主程序与静态库连接 .....	425
A3-2 动态函数库的编写 .....	425
A3-2-1 动态库编译情况 .....	426
A3-2-2 使用动态装载器 .....	428
A3-3 动态/静态函数库优点 .....	429
A3-3-1 静态库优点 .....	429
A3-3-2 动态库优点 .....	430
第四部分 附录（Appendix） .....	431
第五部分 参考资料（Reference） .....	435

# 第一部分 基础知识 (Basic knowledge)

本部分描述了理解核心所需要的基础知识，分为软件和硬件。这里所提到的知识只是提纲式的总结（但是我相信可以为您节省很多的时间）。真正的深入还需读者们根据自己的情况查阅相关资料。

## 软件部分 (Software part)

### S1. Makefile 简介

我们在做大型工程时（比如：Linux、Gcc），通常会在工程中包括成百上千甚至更多的源文件。这么多的源文件如果没有个好的方法去编译它（要你手工的一个一个去编译它们，你会做如何感想，况且只要你改动其中的任何一个源文件，你都要重新编译和连接它们）导致的困难真的是不敢想像的。由于上述原因是人们便做出了一个叫 make 工具（在这里不对 make 工具作解释）。该工具通过一个称为 Makefile 的文件来完成并自动维护源文件的编译及连接工作。Makefile 文件需要按照其规定的语法进行编写，在该文件中定义了如何编译各个源文件并连接生成可执行文件的规则，并定义了源文件之间的依赖关系。当修改了其中某个源文件时，如果其他源文件依赖于该文件，则也要重新编译所有依赖该文件的源文件。

#### S1. 1 Makefile 规则

简单的说来，Makefile 文件就是定义如何编译和连接程序的规则文件。

Makefile的规则：

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

...

target是目标文件，它可以是Object File，也可以是执行文件。还可以是一个标签 (Label)。

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。简单的说就是，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。

下面我们先从一个简单的Makefile看起，我用的测试环境是Redhat 9.0，GNU make 3.79

---

#### S1. 1-1/Makefile

---

```
#Copyright gotop167
#Begin Makefile
one:
```

```

@echo one
two:
@echo two
three:
@echo three
#End Makefile

```

把上面的代码保存进Makefile文件中，执行情况请看下面一系列图示：

当我们只执行 make 命令时，make 命令总是假设在 Makefile 文件中遇到的第一个目标文件是默认目标文件，所以其输出即为“one”。(请看图 S1. 1-1)

```

[root@localhost work]# ls -las
total 12
  4 drwxr-xr-x    2 root      root          4096 Sep 26 22:19 .
  4 drwxr-xr-x    4 root      root          4096 Sep 26 22:19 ..
  4 -rw-----   1 root      ftp           77 Sep 26 22:01 Makefile
[root@localhost work]# make
one
[root@localhost work]#

```

图 S1. 1-1

我们也可以手工调用多个目标文件，此时的输出请看图 S1. 1-2，同时请注意调用目标文件时的顺序。

```

[root@localhost work]# ls -las
total 12
  4 drwxr-xr-x    2 root      root          4096 Sep 26 22:32 .
  4 drwxr-xr-x    4 root      root          4096 Sep 26 22:19 ..
  4 -rw-----   1 root      ftp           79 Sep 26 22:32 Makefile
[root@localhost work]# make two one three
two
one
three
[root@localhost work]#

```

图 S1. 1-2

### S1. 1-2/Makefile (含有宏的)

---

```

#Copyright gotop167
#Begin Makefile
OBJECT=Dog
one:
@echo one $(OBJECT)
two:

```

```

@echo two $(OBJECT)s
three:
@echo three $(OBJECT)s
#End Makefile

```

上述的 Makefile 文件中，“OBJECT”便是定义的宏变量

当我们只执行 make 命令时，make 命令总是假设在 Makefile 文件中遇到的第一个目标文件是默认目标文件，所以其输出即为“one Dog”（请看图 S1.1-3）。

```

[root@localhost work]# ls -las
total 12
4 drwxr-xr-x    2 root      root          4096 Sep 26 22:32 .
4 drwxr-xr-x    4 root      root          4096 Sep 26 22:19 ..
4 -rw-----    1 root      ftp           121 Sep 26 22:42 Makefile
[root@localhost work]# make
one Dog
[root@localhost work]#

```

图 S1.1-3 (含有宏的)

我们也可以手工调用多个目标文件，此时的输出请看图 S1.1-4，同时请注意调用目标文件时的顺序。

```

[root@localhost work]# ls -las
total 12
4 drwxr-xr-x    2 root      root          4096 Sep 26 22:32 .
4 drwxr-xr-x    4 root      root          4096 Sep 26 22:19 ..
4 -rw-----    1 root      ftp           121 Sep 26 22:42 Makefile
[root@localhost work]# make tow one three
two Dogs
one Dog
three Dogs
[root@localhost work]#

```

图 S1.1-4

当然我们也可以在输入 make 命令时，替换掉在 Makefile 文件中定义的宏变量，执行情况详见图 S1.1-5

```
[root@localhost work]# ls -las
total 12
  4 drwxr-xr-x    2 root      root          4096 Sep 26 22:32 .
  4 drwxr-xr-x    4 root      root          4096 Sep 26 22:19 ..
  4 -rw-----    1 root      ftp           121 Sep 26 22:42 Makefile
[root@localhost work]# make OBJECT=apple
one apple
[root@localhost work]# _
```

图 S1.1-5

最后，我们以一个真正编译并连接程序的 Makefile 结束。

## S1. 1-3/a. c

---

```
#include<stdio.h>
int
main(void)
{
    printf( "In main().\n" );
    called(void);
    return 0;
}
```

---

## S1. 1-3/b. c

---

```
#include <stdio.h>
void
called(void)
{
printf("In called.\n");
}
```

---

## S1. 1-3/Makefile

---

```
#Copyright gotop167
CC=gcc
OBJS= a.o b.o

all:test

test:a.o b.o
$(CC) $(OBJS) -o test
```

```
clean:
    rm -f *.o core
clobber:clean
    rm -f test
```

我们把上面的三个文件，分别保存后，放在同一个目录下。接下来的就是我们享受 Makefile 所带来的便利的时候了。

三个文件在目录中的情况（图 S1.1-6）

```
[root@localhost work]# ls -las
total 20
4 drwxr-xr-x  2 root      root          4096 Sep 26 23:48 .
4 drwxr-xr-x  4 root      root          4096 Sep 26 22:19 ..
4 -rw-----  1 root      root           92 Sep 26 23:27 a.c
4 -rw-----  1 root      root          71 Sep 26 23:25 b.c
4 -rw-----  1 root      ftp           163 Sep 26 23:29 Makefile
[root@localhost work]# _
```

图 S1.1-6

执行 make 时情况（图 S1.1-7）

```
[root@localhost work]# make
gcc    -c -o a.o a.c
gcc    -c -o b.o b.c
gcc a.o b.o -o test
[root@localhost work]# _
```

图 S1.1-7

执行被连接成程序 test 时的情况（图 S1.1-8）

```
[root@localhost work]# ./test
In main()
In called.
[root@localhost work]# _
```

图 S1.1-8

执行 make clean 情况（图 S1.1-9）

```
[root@localhost work]# make clean
rm -f *.o core
[root@localhost work]# ls -las
total 32
  4 drwxr-xr-x    2 root      root        4096 Sep 26 23:55 .
  4 drwxr-xr-x    4 root      root        4096 Sep 26 22:19 ..
  4 -rw-----    1 root      root         92 Sep 26 23:27 a.c
  4 -rw-----    1 root      root        71 Sep 26 23:25 b.c
  4 -rw-----    1 root      ftp        163 Sep 26 23:29 Makefile
12 -rwxr-xr-x    1 root      root       11677 Sep 26 23:51 test
[root@localhost work]# _
```

图 S1.1-9

执行 make clobber 情况(图 S1.1-10)

```
[root@localhost work]# make clobber
rm -f *.o core
rm -f test
[root@localhost work]# ls -las
total 20
  4 drwxr-xr-x    2 root      root        4096 Sep 26 23:57 .
  4 drwxr-xr-x    4 root      root        4096 Sep 26 22:19 ..
  4 -rw-----    1 root      root         92 Sep 26 23:27 a.c
  4 -rw-----    1 root      root        71 Sep 26 23:25 b.c
  4 -rw-----    1 root      ftp        163 Sep 26 23:29 Makefile
[root@localhost work]# _
```

图 S1.1-10

怎么样，感觉爽吧！简单的几个命令统统都搞定了！可以想象如果没有 Makefile，对 Linux 核心的编译该是多么折腾人啊！

## S2.汇编简介

汇编语言的优点是速度快，可以直接对硬件进行操作，对时间和空间要求比较高的应用来说是非常重要的。对于编写操作系统更是如此。Linux 核心虽然绝大部分是用 C 语言开发的，但还有很多操作是 C 语言无法完成的。譬如：端口访问、中断调用、访问 CPU 中特殊的寄存器等等。

### S2.1 汇编优缺点

优点：

- 1) 可以根据特定的应用对代码做出最佳的优化，从而提高运行的速度
- 2) 可以直接访问特殊的寄存器、硬件及 I/O 端口

- 3) 可以不受编译器的限制，对生成的二进制代码进行完全的控制
- 4) 可以最大限度地发挥硬件的功能
- 5) 写出的程序尺寸小并且运行效率高
- 6) 可以非常精确的控制代码的执行

缺点：

- 1) 对程序的编写者的要求较高
- 2) 编写的代码难懂，难以维护
- 3) 容易产生 bug，难于调试
- 4) 移植行不好，只能针对特定的体系结构和处理器进行优化
- 5) 开发效率很低，时间长且单调

## S2.2 汇编语法 (AT&T asm VS Intel asm)

1. 在 AT&T 汇编格式中，寄存器名要加上 '%' 作为前缀；在 Intel 汇编格式中，寄存器名不需要加前缀。例如：

AT&T 格式	Intel 格式
movl %eax,%ebx	mov ebx, eax

2. 在 AT&T 汇编格式中，用 '\$' 前缀表示一个立即操作数；在 Intel 汇编格式中，立即数的表示不用带任何前缀。例如：

AT&T 格式	Intel 格式
cmpl \$0x10,%eax	cmp eax,0x10

3. AT&T 和 Intel 格式中的源操作数和目标操作数的位置正好相反。在 Intel 汇编格式中，目标操作数在源操作数的左边；在 AT&T 汇编格式中，目标操作数在源操作数的右边。例如：

AT&T 格式	Intel 格式
addl \$0x10, %eax	add eax, 0x10

4. 在 AT&T 汇编格式中，操作数的字长由操作符的最后一个字母决定，后缀'b'、'w'、'l' 分别表示操作数为字节 (byte, 8 比特)、字 (word, 16 比特) 和长字 (long, 32 比特)；而在 Intel 汇编格式中，操作数的字长是用 "byte ptr" 和 "word ptr" 等前缀来表示的。例如：

AT&T 格式	Intel 格式

movb val, %al	mov al, byte ptr val
---------------	----------------------

5. 在 AT&T 汇编格式中，绝对转移和调用指令（jump/call）的操作数前要加上'\*'作为前缀，而在 Intel 格式中则不需要。
6. 远程转移指令和远程子调用指令的操作码，在 AT&T 汇编格式中为 “ljump” 和 “lcall”，而在 Intel 汇编格式中则为 “jmp far” 和 “call far”，即：

AT&T 格式	Intel 格式
ljump \$section, \$offset	jmp far section:offset
lcall \$section, \$offset	call far section:offset

7. 与之相应的远程返回指令则为：

AT&T 格式	Intel 格式
lret \$stack_adjust	ret far stack_adjust

8. 在 AT&T 汇编格式中，内存操作数的寻址方式是

section:disp(base, index, scale)

而在 Intel 汇编格式中，内存操作数的寻址方式为：

section:[base + index\*scale + disp]

由于 Linux 工作在保护模式下，用的是 32 位线性地址，所以在计算地址时不用考虑段基址和偏移量，而是采用如下的地址计算方法：

disp + base + index \* scale

下面是一些内存操作数的例子：

AT&T 格式	Intel 格式
movl -4(%ebp), %eax	mov eax, [ebp - 4]
movl array(%eax, 4), %eax	mov eax, [eax*4 + array]

movw array(%ebx, %eax, 4), %cx	mov cx, [ebx + 4*eax + array]
movb \$4, %fs:(%eax)	mov fs:eax, 4

## S2.3 Hello world!示例

### S2.3-1/Helloworld.s

---

```
#Helloworld.s
```

```
.data
    msg : .string "Hello, world!\n"
    len = . - msg

.text
.global _start

_start:
    movl $len, %edx
    movl $msg, %ecx
    movl $1, %ebx
    movl $4, %eax
    int  $0x80
    movl $0,%ebx
    movl $1,%eax
    int  $0x80
```

---

### S2.3-1/Makefile

---

```
#Copyright gotop167
```

```
AS=as
```

```
LD=ld
```

```
OBJS=helloworld
```

```
.S.O:
$(AS) -o $*.o $<
```

```
all:$(OBJS)
```

```
helloworld:helloworld.o
```

```

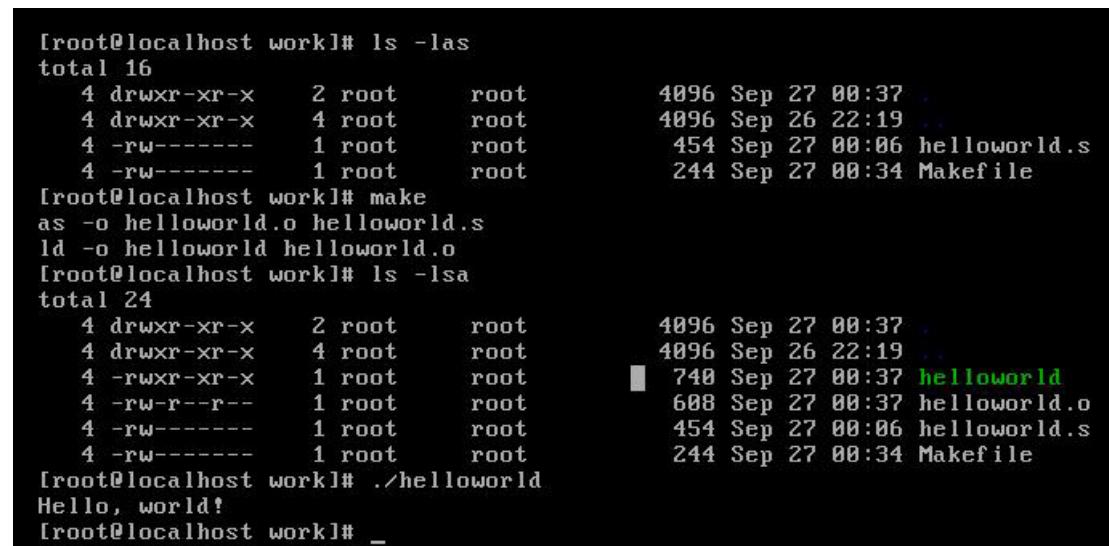
$(LD) -o helloworld helloworld.o

helloworld.o:helloworld.s

clean:
    rm -f *.o core
clobber:clean
    rm -f $(OBJS)

```

执行情况请看图 S2.3-1



```

[root@localhost work]# ls -las
total 16
4 drwxr-xr-x  2 root      root          4096 Sep 27 00:37 .
4 drwxr-xr-x  4 root      root          4096 Sep 26 22:19 ..
4 -rw-----  1 root      root          454 Sep 27 00:06 helloworld.s
4 -rw-----  1 root      root          244 Sep 27 00:34 Makefile
[root@localhost work]# make
as -o helloworld.o helloworld.s
ld -o helloworld helloworld.o
[root@localhost work]# ls -lsa
total 24
4 drwxr-xr-x  2 root      root          4096 Sep 27 00:37 .
4 drwxr-xr-x  4 root      root          4096 Sep 26 22:19 ..
4 -rwxr-xr-x  1 root      root          740 Sep 27 00:37 helloworld
4 -rw-r--r--  1 root      root          608 Sep 27 00:37 helloworld.o
4 -rw-----  1 root      root          454 Sep 27 00:06 helloworld.s
4 -rw-----  1 root      root          244 Sep 27 00:34 Makefile
[root@localhost work]# ./helloworld
Hello, world!
[root@localhost work]#

```

图 S2.3-1

## S3.实模式向保护模式切换

### S3.1 切换到保护方式的准备工作

从实模式切换到保护模式之前，必须作必要的准备。准备工作的内容根据实际而定。最起码的准备工作是建立合适的全局描述符表，并使用 GDTR 指向该 GDT。因为在切换到保护方式时，至少要把代码段的选择子装载到 CS，所以 GDT 中至少含有代码段的描述符。另外还要设置 IDTR。

设置 GDTR：

我们从 linux1.0 核心中取出部分代码来看看！以下代码取之 boot/setup.s 中

```

778 gdt:
779     .word 0,0,0,0      ! dummy

```

```

780
781      .word  0,0,0,0      ! unused
782
783      .word  0x07FF      ! 8Mb - limit=2047 (2048*4096=8Mb)
784      .word  0x0000      ! base address=0
785      .word  0x9A00      ! code read/exec
786      .word  0x00C0      ! granularity=4096, 386
787
788      .word  0x07FF      ! 8Mb - limit=2047 (2048*4096=8Mb)
789      .word  0x0000      ! base address=0
790      .word  0x9200      ! data read/write
791      .word  0x00C0      ! granularity=4096, 386
797 gdt_48:
798      .word  0x800      ! gdt limit=2048, 256 GDT entries
799      .word  512+gdt,0x9  ! gdt base = 0X9xxxx

```

从源程序可见，全局描述符表 GDT 有四个描述符：第一个是空描述符；第二个是空描述符；第三个是代码段描述符（大小是 8M，起始于地址 0 处）；第四个数据段描述符（大小是 8M，起始于地址 0 处）。

由于在切换到保护方式后就要引用 GDT，所以在切换到保护方式前必须装载 GDTR。实例中使用如下指令装载 GDTR：

```
180      lgdt  gdt_48      ! load gdt with whatever appropriate
```

该指令的功能是把存储器中的描述符 gdt\_48 装入到全局描述符表寄存器 GDTR 中。

设置 IDTR：

我们从 linux1.0 核心中取出部分代码来看看！以下代码取之 boot/setup.s 中

```
793 idt_48:
```

```

794      .word  0          ! idt limit=0
795      .word  0,0         ! idt base=0L

```

```
179      lidt  idt_48      ! load idt with 0,0
```

该指令的功能是把存储器中的描述符 idt\_48 装入到中断描述符表寄存器 IDTR 中。通过代码我们可以看到 idt\_48 是空的，什么也没有设置，只是为了在向保护模式切换时不发生错误而已。

## S3.2 使用段间指令切换进保护模式

在做好准备后，从实模式切换到保护模式并不难。原则上只要把控制寄存器 CR0 中的 PE 位置 1 即可。本实例采用如下三条指令设置 PE 位：

```

mov    eax,cr0
or     eax,1
mov    cr0,ea

```

实际情况要比这复杂些。执行上面的三条指令后，处理器转入保护模式，但 CS 中的内容还是实模式下代码段的段值，而不是保护模式下代码段的选择子，所以在取指令之前得把代码段的选择子装入 CS。为此，紧接着这三条指令，安排一条如下所示的段间转移指令：

JMPI Code\_Seg , Offset Address

这条段间转移指令在实模式下被预取并在保护方式下被执行。利用这条段间转移指令可把保护模式下代码段的选择子装入 CS，同时也刷新指令预取队列。从此真正进入保护模式。

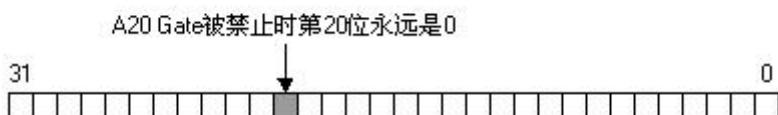
### S3.3 打开 A20 地址线切换进保护模式

打开 A20 地址线也是 linux1.0 核心切换到保护模式的方式！

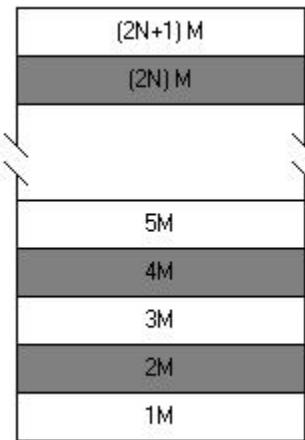
IBM 使用键盘控制器上剩余的一些输出线来管理第 21 根地址线（从 0 开始数是第 20 根），被称为 A20 Gate：如果 A20 Gate 被打开，则当程序员给出 100000H-10FFEFH 之间的地址的时候，系统将真正访问这块内存区域；如果 A20 Gate 被禁止，则当程序员给出 100000H-10FFEFH 之间的地址的时候，系统仍然使用 8086/8088 的方式。绝大多数 IBM PC 兼容机默认的 A20 Gate 是被禁止的。由于在当时没有更好的方法来解决这个问题，所以 IBM 使用了键盘控制器来操作 A20Gate，但这只是一种黑客行为，毕竟 A20 Gate 和键盘操作没有任何关系。在许多新型 PC 上存在着一种通过芯片来直接控制 A20Gate 的 BIOS 功能。从性能上，这种方法比通过键盘控制器来控制 A20 Gate 要稍微高一点。

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。（其实所谓的实模式，就是 8086/8088 的模式，这种模式存在的唯一理由就是为了让旧的程序能够继续正常的运行在新的 PC 体系上）

我们来看一看 A20 的工作原理。A20，从它的名字就可以看出来，其实它就是对于 20-bit（从 0 开始数）的特殊处理（也就是对第 21 根地址线的处理）。如果 A20 Gate 被禁止，对于 80286 来说，其地址为 24bit，其地址表示为 EFFFFF；对于 80386 极其随后的 32-bit 芯片来说，其地址表示为 FFEFFFFF。这种表示的意思是如果 A20 Gate 被禁止，则其第 20-bit 在 CPU 做地址访问的时候是无效的，永远只能被作为 0；如果 A20 Gate 被打开，则其第 20-bit 是有效的，其值既可以是 0，又可以是 1。



所以，在保护模式下，如果 A20 Gate 被禁止，则可以访问的内存只能是奇数 1M 段，即 1M, 3M, 5M…，也就是 00000-FFFFF, 200000-2FFFFF, 300000-3FFFFF…。如果 A20 Gate 被打开，则可以访问的内存则是连续的。



打开 A20 Gate 的方法是通过设置 8042 芯片输出端口（64h）的 2nd-bit，但事实上，当你向 8042 芯片输出端口进行写操作的时候，在键盘缓冲区中，或许还有别的数据尚未处理，因此你必须首先处理这些数据。

流程如下：

1. 禁止中断；
2. 等待，直到 8042 Input buffer 为空为止；
3. 发送禁止键盘操作命令到 8042 Input buffer；
4. 等待，直到 8042 Input buffer 为空为止；
5. 发送读取 8042 Output Port 命令；
6. 等待，直到 8042 Output buffer 有数据为止；
7. 读取 8042 Output buffer，并保存得到的字节；
8. 等待，直到 8042 Input buffer 为空为止；
9. 发送 Write 8042 Output Port 命令到 8042 Input buffer；
10. 等待，直到 8042 Input buffer 为空为止；
11. 将从 8042 Output Port 得到的字节的第 2 位置 1 (OR 2)，然后写入 8042 Input buffer；
12. 等待，直到 8042 Input buffer 为空为止；
13. 发送允许键盘操作命令到 8042 Input buffer；
14. 打开中断。

我们从 boot/setup.s 中取出代码看看

```

184      call    empty_8042    ! 等待 8042 缓冲器空，只有为空时
          ! 才可以写
185      mov     al,#0xD1           ! command write
186      out    #0x64,al
          ! 0xD1 命令码，写到 8042 的 P2 端口，
          ! P2 端口的位 1 用于选通 A20 地址线

187      call    empty_8042    ! 等待 8042 缓冲器空
188      mov     al,#0xDF           ! A20 on
189      out    #0x60,al ! 选通A20 地址线的参数
190      call    empty_8042    ! 等待 8042 缓冲器空

```

```

.....  

260 empty_8042:  

261     call    delay ! 等待 (起延时作用)  

262     in     al,#0x64      ! 8042 status port  

        ! 读 AT 键盘控制器状态寄存器。  

263     test   al,#1       ! output buffer?  

        ! 测试位 1  

264     jz     no_output  

265     call   delay  

266     in     al,#0x60      ! read it  

267     jmp   empty_8042  

268 no_output:      ! 没有输出  

269     test   al,#2       ! is input buffer full?  

        ! 测试位 2, 输入缓冲器满?  

270     jnz   empty_8042      ! yes - loop  

271     ret

```

## S4.gcc 内嵌汇编

### S4.1 内嵌汇编格式

```

asm ( assembler template          #汇编指令部分
      : output operands         #输出寄存器           (可选的)
      : input operands          #输入寄存器           (可选的)
      : list of clobbered registers #操作会被修改的寄存器或内存 (可选的)
);

```

内联汇编的重要性体现在它能够灵活操作，而且可以使其输出通过 C 变量显示出来。因为它具有这种能力，所以 "asm" 可以用作汇编指令和包含它的 C 程序之间的接口。一个非常基本但很重要的区别在于 简单内联汇编只包括指令，而 扩展内联汇编包括操作数。

在 GCC 内联汇编语句的指令部中，加上前缀 '%' 的数字(如%0, %1)表示的就是需要使用寄存器的"样板"操作数。指令部中使用了几个样板操作数，就表明有几个变量需要与寄存器相结合，这样 GCC 和 GAS 在编译和汇编时会根据后面给定的约束条件进行恰当的处理。由于样板操作数也使用 '%' 作为前缀，因此在涉及到具体的寄存器时，寄存器名前面应该加上两个 '%'，以免产生混淆。

紧跟在指令部后面的是输出寄存器，是规定输出变量如何与样板操作数进行结合的条件，每个条件称为一个"约束"，必要时可以包含多个约束，相互之间用逗号分隔开就可以了。每个输出约束都以 '=' 号开始，然后紧跟一个对操作数类型进行说明的字后，最后是如何与变量相结合的约束。凡是与输出部中说明的操作数相结合的寄存器或操作数本身，在执行完嵌入的汇编代码后均不保留执行之前的内容，这是 GCC 在调度寄存器时所使用的依据。

输出部后面是输入寄存器，输入约束的格式和输出约束相似，但不带'='号。如果一个输入约束要求使用寄存器，则GCC在预处理时就会为之分配一个寄存器，并插入必要的指令将操作数装入该寄存器。与输入部中说明的操作数结合的寄存器或操作数本身，在执行完嵌入的汇编代码后也不保留执行之前的内容。

有时在进行某些操作时，除了要用到进行数据输入和输出的寄存器外，还要使用多个寄存器来保存中间计算结果，这样就难免会破坏原有寄存器的内容。在GCC内联汇编格式中的最后一个部分中，可以对将产生副作用的寄存器进行说明，以便GCC能够采用相应的措施。

## S4.2 内嵌汇编示例

S4.2-1/swap.c

---

```
#include <stdio.h>
int
main()
{
    int iValue = 100;
    int jValue = 200;
    printf("\nBefore Swap:a=%d,b=%d\n",iValue,jValue);
    __asm__
    ("movl (%0),%%eax\n\t"
     "movl (%1),%%edx\n\t"
     "movl %%eax,(%1)\n\t"
     "movl %%edx,(%0)\n\t"
     ::"b"(&iValue),"c"(&jValue)
    );
    printf("After Swap:a=%d,b=%d\n\n",iValue,jValue);
    return 0;
}
```

---

S4.2-1/Makefile

---

```
GCC=gcc
OBJS=swap
```

---

```
.c.o:
$(GCC) -c -Wall $<
```

```

all:$(OBJS)

$(OBJS):swap.o
    $(GCC) -o $(OBJS) swap.o

clean:
    rm -f *.o core
clobber:clean
    rm -f $(OBJS)

```

我们取出 swap.c 中嵌入汇编来分析一下：

```

__asm__
("movl (%0),%%eax\n\t"           #把 iValue 送入 eax 中
 "movl (%1),%%edx\n\t"           #把 jValue 送入 edx 中
 "movl %%eax,(%1)\n\t"          #把 eax 中的值送入 jValue
 "movl %%edx,(%0)\n\t"          #把 edx 中的值送入 iValue 中
 ::"b"("&iValue),"c"(&jValue)      #ebx, ecx 为输入部分, 没有输出部分及 clobbered
 registers
);

```

通过以上的嵌入代码便完成了数据的交换，上面的代码类似于如下的 C 代码

```

{
    int iTmp = iValue;
    iValue = jValue;
    iValue = iTmp;
}

```

执行情况请看图 S4.2-1

```

[root@localhost work]# ls -ls
total 8
  4 -rw-r--r--    1 ftp      ftp            154 Sep 27 04:39 Makefile
  4 -rw-r--r--    1 ftp      ftp            311 Sep 27 05:35 swap.c
[root@localhost work]# make
gcc -c -Wall swap.c
gcc -o swap swap.o
[root@localhost work]# ls -ls
total 24
  4 -rw-r--r--    1 ftp      ftp            154 Sep 27 04:39 Makefile
 12 -rwxr-xr-x   1 root     root           11637 Sep 27 05:35 swap
  4 -rw-r--r--    1 ftp      ftp            311 Sep 27 05:35 swap.c
  4 -rw-r--r--    1 root     root           912 Sep 27 05:35 swap.o
[root@localhost work]# ./swap
Before Swap:a=100,b=200
After Swap:a=200,b=100

[root@localhost work]#

```

图 S4.2-1

## S5.GDB 调试器

GDB 调试器是用来调试 Linux 下程序用的，它是一个交互式工具，工作在字符模式。在 X Window 系统中，有一个 gdb 的前端图形工具，称为 ddd。gdb 是功能强大的调试程序，可完成如下的调试任务：

1. 设置断点
2. 监视程序变量的值
3. 程序的单步执行
4. 动态修改变量的值

当我们使用 gdb 调试程序之前，必须使用 -g 选项编译源文件。这样我们便可以直接对源代码进行调试了。比较简单的办法是在 Makefile 中如下定义 CFLAGS 变量：

CFLAGS = -g

来编译程序。

GDB 调试程序时通常使用如下的命令：

`gdb 程序名`

在 gdb 提示符处键入 help，将列出命令的分类，主要的分类有（请看图 S5.1-1）

```

GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb) _

```

图 S5.1-1

- aliases: 命令别名
- breakpoints: 断点定义
- data: 数据查看
- files: 指定并查看文件
- internals: 维护命令
- running: 程序执行
- stack: 调用栈查看
- status: 状态查看
- support: 支持工具
- tracepoints: 跟踪程序执行。

## S5.1 GDB 命令

命令	解释
info local	显示当函数中的局部变量信息
info var	显示所有的全局和静态变量名称
display EXPR	每次程序停止后显示表达式的值。表达式由程序定义的变量组成
file FILE	装载指定的可执行文件进行调试
info prog	显示被调试程序的执行状态
kill	终止正被调试的程序
list	显示源代码段
make	在不退出 gdb 的情况下运行 make 工具
next	在不单步执行进入其他函数的情况下，向前执行一行源代码
print EXPR	显示表达式 EXPR 的值
info func	显示所有的函数名称
info files	显示被调试文件的详细信息
help NAME	显示指定命令的帮助信息
info break	显示当前断点清单，包括到达断点处的次数等
continue	继续执行正在调试的程序。该命令用在程序由于处理信号或断点而导致停止运行
clear	删除设置在特定源文件、特定行上的断点。其用法为： clear FILENAME:NUM。
bt	显示所有的调用栈帧。该命令可用来显示函数的调用顺序
break NUM	在指定的行上设置断点

## S5.2 GDB 调试样例

### S5.2-1/bug.c

```
#include <stdio.h>
#include <unistd.h>
```

```
int
main(void)
{
int scores[10];
int sum;
int i;
int average;

for(i=0; i<10; ++i)
    scores[i] = 85;

for(i=0; i<10; ++i)
    sum += scores[i];

average = sum / 10;

printf("The average score is %d.\n",average);

return 0;
}
```

---

### S5.2-1/Makefile

---

```
GCC=gcc
OBJS=bug.o
CFLAGS = -g

.c.o:
    $(GCC) $(CFLAGS) -c $<

all:$(OBJS)
    $(GCC) $(OBJS) -o bug
clean:
    rm -f *.o core

clobber:clean
    rm -f bug
```

---

```
[root@localhost S5-1]# gdb bug
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) list
3
4     int
5     main(void)
6     {
7         int scores[10];
8         int sum;
9         int i;
10        int average;
11
12        for(i=0; i<10; ++i)
(gdb) _
```

图 S5.2-1

在图 S5.2-1 中，我们可以看到用“gdb bug”来加载这个要被调试的程序的。并且我们用 list 列出 10 行源代码。

```
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) list
3
4     int
5     main(void)
6     {
7         int scores[10];
8         int sum;
9         int i;
10        int average;
11
12        for(i=0; i<10; ++i)
(gdb) break 12
Breakpoint 1 at 0x8048338: file bug.c, line 12.
(gdb) r
Starting program: /var/ftp/pub/S5-1/bug

Breakpoint 1, main () at bug.c:12
12        for(i=0; i<10; ++i)
(gdb) _
```

图 S5.2-2

在图 S5.2-2 中，我们用“break”指令为第 12 行代码加上断点，接着用“r”（其实也就是“run”）指令让程序开始运行，并且在运行到第 12 行时停着。等待这你的调度。

```

Breakpoint 2 at 0x8048359: file bug.c, line 15.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) n
Program not restarted.
(gdb) c
Continuing.

Breakpoint 2, main () at bug.c:15
15      for(i=0; i<10; ++i)
(gdb) print i
$8 = 10
(gdb) n
16      sum += scores[i];
(gdb) print i
$9 = 0
(gdb) print sum
$10 = 1073832832
(gdb) n
15      for(i=0; i<10; ++i)
(gdb) n
16      sum += scores[i];
(gdb) print sum
$11 = 1073832917
(gdb) _

```

图 S5.2-3

在图 S5.2-3 中，我们用首先打印出 `i` 的值，它为 0。接着打印 `sum` 的值，居然是 1073832832，而不是 0，这就让人奇怪了。因为这个时候还未进行任何数据的相加。（因为此时 `i=0`）估计这里发生了什么问题，于是再向下跟踪。

```

13      scores[i] = 85;
14
15      for(i=0; i<10; ++i)
16          sum += scores[i];
17
18      average = sum / 10;
19
(gdb) break 18
Breakpoint 3 at 0x804837b: file bug.c, line 18.
(gdb) c
Continuing.

Breakpoint 3, main () at bug.c:18
18      average = sum / 10;
(gdb) print sum
$12 = 1073833682
(gdb) print average
$13 = 1073790467
(gdb) n
20      printf("The average score is %d.\n",average);
(gdb) print sum
$14 = 1073833682
(gdb) print average
$15 = 107383368
(gdb) _

```

图 S5.2-4

在图 S5.2-4 中，我们看到在 `sum=1073833682`, `average=107383368`, 这个结果肯定是对的。在图 S5.2-3，还未进行任何的累加。`sum=1073832832`。这应该是我们没有对其初始化的原因。于是我们修改之。

```
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) l
3
4     int
5     main(void)
6     {
7         int scores[10];
8         int sum;
9         int i;
10        int average;
11
12        for(i=0; i<10; ++i)
(gdb) break 10
Breakpoint 1 at 0x8048338: file bug.c, line 10.
(gdb) r
Starting program: /var/ftp/pub/S5-1/bug

Breakpoint 1, main () at bug.c:12
12        for(i=0; i<10; ++i)
(gdb) set sum=0
(gdb) n
13        scores[i] = 85;
(gdb) print sum
$1 = 0
(gdb) _
```

图 S5.2-5

在图 S5.2-5 中，我们用“set”指令把 sum 设置为 0

```
10    int average;
11
12    for(i=0; i<10; ++i)
(gdb) break 10
Breakpoint 1 at 0x8048338: file bug.c, line 10.
(gdb) r
Starting program: /var/ftp/pub/S5-1/bug

Breakpoint 1, main () at bug.c:12
12        for(i=0; i<10; ++i)
(gdb) set sum=0
(gdb) n
13        scores[i] = 85;
(gdb) print sum
$1 = 0
(gdb) break 17
Breakpoint 2 at 0x804837b: file bug.c, line 17.
(gdb) c
Continuing.

Breakpoint 2, main () at bug.c:18
18        average = sum / 10;
(gdb) print sum
$2 = 850
(gdb) _
```

图 S5.2-6

在图 S5.2-7 中，我们打印 sum 等于 850，这个是对的了。因为我们确实是对 10 个 85 进行相加的。

```
(gdb) n
13     scores[i] = 85;
(gdb) print sum
$1 = 0
(gdb) break 17
Breakpoint 2 at 0x804837b: file bug.c, line 17.
(gdb) c
Continuing.

Breakpoint 2, main () at bug.c:18
18     average = sum / 10;
(gdb) print sum
$2 = 850
(gdb) n
20     printf("The average score is %d.\n",average);
(gdb) print average
$3 = 85
(gdb) n
The average score is 85.
22     return 0;
(gdb) n
23 }
(gdb) n
0x42015574 in __libc_start_main () from /lib/tls/libc.so.6
(gdb) _
```

图 S5.2-7

在图 S5.2-7 中，我们看到打印出的提示信息是 “The average score is 85.”。根据提示信息我们可以知道程序对了。那么我们也知道要修改的代码是什么了。就是要初始化 sum=0。这里就不在给出修改后的代码了。

## S6.系统调用实现详解

Linux 核心中真正被所有进程都使用的内核通信方式是系统调用。当进程请求内核服务时，就是通过系统调用来达到要求的。因为在用户态方式下，进程是不能够存取系统核心的。它不能存取内核使用的内存段，也不能调用内核函数，CPU 的硬件结构保证了这一点（X86CPU 有 4 个级别，从 Ring0 到 Ring3）。只有系统调用是一个例外（当然在执行真正的调用前，核心会做检查的）。进程使用寄存器中适当的值跳转到内核中事先定义好的代码中执行。目前的 Linux 核心是用 0x80 号做为所有中断的入口点。当然你也可以修改它（不过你还要修改你要使用的应用程序）。

进程可以跳转到的内核中的位置叫做 system\_call。在此位置的过程检查系统调用号，它将告诉内核进程请求的服务是什么。然后，它再查找系统调用表 sys\_call\_table，找到希望调用的内核函数的地址，并调用此函数，最后返回。

### S6.1 核心中提供的宏

在 Linux 核心源代码中，提供了好几个宏来实现系统调用。我们只要好好利用它，调用系统调用将会是件非常容易的事情。以下的宏来源于 1.0 核心。

```

148 #define __syscall0(type,name) \
149 type name(void) \
150 { \
151     long __res; \
152     __asm__ volatile ("int $0x80" \
153                     : "=a" (__res) \
154                     : "")(__NR_##name)); \
155     if (__res >= 0) \
156         return (type) __res; \
157     __errno = -__res; \
158     return -1; \
159 }

```

! 对应于无参数的系统调用,  
! 比如: asmlinkage int sys\_pause(void)

```

160
161 #define __syscall1(type,name,atype,a) \
162 type name(atype a) \
163 { \
164     long __res; \
165     __asm__ volatile ("int $0x80" \
166                     : "=a" (__res) \
167                     : "")(__NR_##name),"b" ((long)(a))); \
168     if (__res >= 0) \
169         return (type) __res; \
170     __errno = -__res; \
171     return -1; \
172 }

```

! 对应于一个参数的系统调用,  
! 比如: asmlinkage int sys\_setup(void \* BIOS)

```

173
174 #define __syscall2(type,name,atype,a,btype,b) \
175 type name(atype a,btype b) \
176 { \
177     long __res; \
178     __asm__ volatile ("int $0x80" \
179                     : "=a" (__res) \
180                     : "")(__NR_##name),"b" ((long)(a)), "c" ((long)(b))); \
181     if (__res >= 0) \
182         return (type) __res; \
183     __errno = -__res; \
184     return -1; \
185 }

```

！对应于二个参数的系统调用，

！比如：asmlinkage int sys\_chmod(const char \* filename, mode\_t mode)

186

```
187 #define syscall3(type,name,atype,a,btype,b,ctype,c) \
188 type name(atype a,btype b,ctype c) \
189 { \
190 long __res; \
191 __asm__ volatile ("int $0x80" \
192         : "=a" (__res) \
193         : "" ("NR_##name),"b" ((long)(a)), "c" ((long)(b)), "d" ((long)(c))); \
194 if (__res>=0) \
195     return (type) __res; \
196 errno=-__res; \
197 return -1; \
198 }
```

！对应于三个参数的系统调用，

！比如 asmlinkage int sys\_write(unsigned int fd,char \* buf,unsigned int count)

199

```
200 #define syscall4(type,name,atype,a,btype,b,ctype,c,dtype,d) \
201 type name (atype a, btype b, ctype c, dtype d) \
202 { \
203 long __res; \
204 __asm__ volatile ("int $0x80" \
205         : "=a" (__res) \
206         : "" ("NR_##name),"b" ((long)(a)), "c" ((long)(b)), \
207         "d" ((long)(c)), "S" ((long)(d))); \
208 if (__res>=0) \
209     return (type) __res; \
210 errno=-__res; \
211 return -1; \
212 }
```

！对应于四个参数的系统调用，

！比如 asmlinkage int

！ sys\_init\_module(char \*module\_name, char \*code, unsigned codesize,  
！ struct mod\_routines \*routines)

213

```
214 #define syscall5(type,name,atype,a,btype,b,ctype,c,dtype,d,etype,e) \
215 type name (atype a,btype b,ctype c,dtype d,etype e) \
216 { \
217 long __res; \
218 __asm__ volatile ("int $0x80" \
```

```

219      : "=a" (__res) \
220      : "" (NR_##name), "b" ((long)(a)), "c" ((long)(b)), \
221      "d" ((long)(c)), "S" ((long)(d)), "D" ((long)(e)); \
222 if (__res>=0) \
223     return (type) __res; \
224 errno=-__res; \
225 return -1; \
226 }

! 对应于五个参数的系统调用,
! 比如 asmlinkage int sys_mount(char * dev_name, char * dir_name, char * type,
unsigned long new_flags, void * data)

227

```

## S6.2 系统调用编号

```

3
4 /*
5 * This file contains the system call numbers and the syscallX
6 * macros
7 */
8
9 #define NR_setup          0 /* used only by init, to get system going */
10 #define NR_exit           1
11 #define NR_fork            2
12 #define NR_read             3
13 #define NR_write            4
14 #define NR_open             5
15 #define NR_close            6
16 #define NR_waitpid         7
17 #define NR_creat            8
18 #define NR_link             9
19 #define NR_unlink           10
20 #define NR_execve           11
21 #define NR_chdir            12
22 #define NR_time             13
23 #define NR_mknod            14
24 #define NR_chmod            15
25 #define NR_chown            16
26 #define NR_break            17
27 #define NR_oldstat          18
28 #define NR_lseek             19
29 #define NR_getpid           20

```

30 #define <u>NR_mount</u>	21
31 #define <u>NR_umount</u>	22
32 #define <u>NR_setuid</u>	23
33 #define <u>NR_getuid</u>	24
34 #define <u>NR_stime</u>	25
35 #define <u>NR_ptrace</u>	26
36 #define <u>NR_alarm</u>	27
37 #define <u>NR_oldfstat</u>	28
38 #define <u>NR_pause</u>	29
39 #define <u>NR_ftime</u>	30
40 #define <u>NR_stty</u>	31
41 #define <u>NR_gtty</u>	32
42 #define <u>NR_access</u>	33
43 #define <u>NR_nice</u>	34
44 #define <u>NR_fsync</u>	35
45 #define <u>NR_sync</u>	36
46 #define <u>NR_kill</u>	37
47 #define <u>NR_rename</u>	38
48 #define <u>NR_mkdir</u>	39
49 #define <u>NR_rmdir</u>	40
50 #define <u>NR_dup</u>	41
51 #define <u>NR_pipe</u>	42
52 #define <u>NR_times</u>	43
53 #define <u>NR_prof</u>	44
54 #define <u>NR_brk</u>	45
55 #define <u>NR_setgid</u>	46
56 #define <u>NR_getgid</u>	47
57 #define <u>NR_signal</u>	48
58 #define <u>NR_geteuid</u>	49
59 #define <u>NR_getegid</u>	50
60 #define <u>NR_acct</u>	51
61 #define <u>NR_phys</u>	52
62 #define <u>NR_lock</u>	53
63 #define <u>NR_ioctl</u>	54
64 #define <u>NR_fcntl</u>	55
65 #define <u>NR_mpx</u>	56
66 #define <u>NR_setpgid</u>	57
67 #define <u>NR_ulimit</u>	58
68 #define <u>NR_oldolduname</u>	59
69 #define <u>NR_umask</u>	60
70 #define <u>NR_chroot</u>	61
71 #define <u>NR_ustat</u>	62
72 #define <u>NR_dup2</u>	63
73 #define <u>NR_getppid</u>	64

74 #define <a href="#">NR_getpgrp</a>	65
75 #define <a href="#">NR_setsid</a>	66
76 #define <a href="#">NR_sigaction</a>	67
77 #define <a href="#">NR_sgetmask</a>	68
78 #define <a href="#">NR_ssetmask</a>	69
79 #define <a href="#">NR_setreuid</a>	70
80 #define <a href="#">NR_setregid</a>	71
81 #define <a href="#">NR_sigsuspend</a>	72
82 #define <a href="#">NR_sigpending</a>	73
83 #define <a href="#">NR_sethostname</a>	74
84 #define <a href="#">NR_setrlimit</a>	75
85 #define <a href="#">NR_getrlimit</a>	76
86 #define <a href="#">NR_getrusage</a>	77
87 #define <a href="#">NR_gettimeofday</a>	78
88 #define <a href="#">NR_settimeofday</a>	79
89 #define <a href="#">NR_getgroups</a>	80
90 #define <a href="#">NR_setgroups</a>	81
91 #define <a href="#">NR_select</a>	82
92 #define <a href="#">NR_symlink</a>	83
93 #define <a href="#">NR_oldlstat</a>	84
94 #define <a href="#">NR_readlink</a>	85
95 #define <a href="#">NR_uselib</a>	86
96 #define <a href="#">NR_swapon</a>	87
97 #define <a href="#">NR_reboot</a>	88
98 #define <a href="#">NR_readdir</a>	89
99 #define <a href="#">NR_mmap</a>	90
100 #define <a href="#">NR_munmap</a>	91
101 #define <a href="#">NR_truncate</a>	92
102 #define <a href="#">NR_ftruncate</a>	93
103 #define <a href="#">NR_fchmod</a>	94
104 #define <a href="#">NR_fchown</a>	95
105 #define <a href="#">NR_getpriority</a>	96
106 #define <a href="#">NR_setpriority</a>	97
107 #define <a href="#">NR_profil</a>	98
108 #define <a href="#">NR_statfs</a>	99
109 #define <a href="#">NR_fstatfs</a>	100
110 #define <a href="#">NR_ioperm</a>	101
111 #define <a href="#">NR_socketcall</a>	102
112 #define <a href="#">NR_syslog</a>	103
113 #define <a href="#">NR_setitimer</a>	104
114 #define <a href="#">NR_getitimer</a>	105
115 #define <a href="#">NR_stat</a>	106
116 #define <a href="#">NR_lstat</a>	107
117 #define <a href="#">NR_fstat</a>	108

<u>118</u> #define <u>NR_olduname</u>	109
<u>119</u> #define <u>NR_iopl</u>	110
<u>120</u> #define <u>NR_vhangup</u>	111
<u>121</u> #define <u>NR_idle</u>	112
<u>122</u> #define <u>NR_vm86</u>	113
<u>123</u> #define <u>NR_wait4</u>	114
<u>124</u> #define <u>NR_swapoff</u>	115
<u>125</u> #define <u>NR_sysinfo</u>	116
<u>126</u> #define <u>NR_ipc</u>	117
<u>127</u> #define <u>NR_fsync</u>	118
<u>128</u> #define <u>NR_sigreturn</u>	119
<u>129</u> #define <u>NR_clone</u>	120
<u>130</u> #define <u>NR_setdomainname</u>	121
<u>131</u> #define <u>NR_uname</u>	122
<u>132</u> #define <u>NR_modify_ldt</u>	123
<u>133</u> #define <u>NR_adjtimex</u>	124
<u>134</u> #define <u>NR_mprotect</u>	125
<u>135</u> #define <u>NR_sigprocmask</u>	126
<u>136</u> #define <u>NR_create_module</u>	127
<u>137</u> #define <u>NR_init_module</u>	128
<u>138</u> #define <u>NR_delete_module</u>	129
<u>139</u> #define <u>NR_get_kernel_syms</u>	130
<u>140</u> #define <u>NR_quotactl</u>	131
<u>141</u> #define <u>NR_getpgid</u>	132
<u>142</u> #define <u>NR_fchdir</u>	133
<u>143</u> #define <u>NR_bdfflush</u>	134

这些编号来源于 1.0 核心，只有 134 个。而 2.4 核心中却已经有了 258 个了。当然有的并没有被使用。

### S6.3 系统调用入口点及函数表

```

161 .align 4
162 _system_call:
163     pushl %eax          # save orig_eax
164     SAVE_ALL
165     movl $-ENOSYS,EAX(%esp)
166     cmpl _NR_syscalls,%eax
167     jae ret_from_sys_call
168     movl _current,%ebx
169     andl $~CF_MASK,EFLAGS(%esp)    # clear carry - assume no errors

```

```

170     movl $0,errno(%ebx)
171     movl %db6,%edx
172     movl %edx,dbggreg6(%ebx) # save current hardware debugging status
173     testb $0x20,flags(%ebx)      # PF_TRACESYS
174     jne 1f
175     call _sys_call_table(%eax,4)
176     movl %eax,EAX(%esp)        # save the return value

```

上面这片代码来源于 1.0 核心，`_system_call` 便是整个系统调用的入口点，在第 163 到 174 行代码中做了一些必要的工作后，便在 175 行处通过一条 `call` 指令调用对应的系统调用函数处理指针。其实是查表得到的，该表是定义在 `Kernel/sched.c` 中函数表。下面给出该表的代码。

```

121 fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
122 sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
123 sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
124 sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
125 sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
126 sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
127 sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
128 sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
129 sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
130 sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
131 sys_olduname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
132 sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
133 sys_setreuid, sys_setregid, sys_sigsuspend, sys_sigpending,
134 sys_sethostname, sys_setrlimit, sys_getrlimit, sys_getrusage,
135 sys_gettimeofday, sys_settimeofday, sys_getgroups, sys_setgroups,
136 sys_select, sys_symlink, sys_lstat, sys_readlink, sys_uselib,
137 sys_swapon, sys_reboot, sys_readdir, sys_mmap, sys_munmap, sys_truncate,
138 sys_ftruncate, sys_fchmod, sys_fchown, sys_getpriority, sys_setpriority,
139 sys_profil, sys_statsfs, sys_fstatfs, sys_ioperm, sys_socketcall,
140 sys_syslog, sys_setitimer, sys_getitimer, sys_newstat, sys_newlstat,
141 sys_newfstat, sys_uname, sys_iopl, sys_vhangup, sys_idle, sys_vm86,
142 sys_wait4, sys_swapoff, sys_sysinfo, sys_ipc, sys_fsync, sys_sigreturn,
143 sys_clone, sys_setdomainname, sys_newuname, sys_modify_ldt,
144 sys_adjtimex, sys_mprotect, sys_sigprocmask, sys_create_module,
145 sys_init_module, sys_delete_module, sys_get_kernel_syms, sys_quotactl,
146 sys_getpgid, sys_fchdir, sys_bdflush };

```

## S6.4 对系统调用调用

测试环境 Vmware5.0+RedHat 9.0

---

S6.4-1/father.c

---

```
#include <linux/unistd.h>
! 在该头文件中包含了我们需要的系统调用宏，就象 S6.1 节列出的，

int
main(void)
{
int i=0;
printf("Now,we begin fork me.\n");
! 打印提示，告诉我们父进程要开始 fork 它自己了
if(!fork())    ! 执行 fork，对应于 sys_fork。
{
    execve("./son", 0, 0);
! fork 成功后，会执行这里的代码，因为返回值是 0，所以这里我们用 execve 系统调用去执行 son 程序，如果 son 执行成功的话，便会用 son 的镜像替换掉自己。
    printf("Sorry! execve a new process failed.\n");
    ! 如果，执行这句函数，则说明我们执行 son 时失败。
}
else
{
    for(i=0; i<5; ++i)
    printf("In father:The i is %d.\n",i);
    ! 对应于父进程。我们打印出提示消息。
}
return 0;
}
```

---



---

S6.4-1/son.c

---

```
#include <stdio.h>
! 子进程，没什么功能，只是告诉我，它已经在执行了。

int
main(void)
{
int i;
int j;
printf("I am son process.\n");
for(i=0; i<5; ++i)
printf("In son:The i is %d.\n",i);
return 0;
}
```

---

---

S6.4-1/Makefile

---

```

GCC=gcc
FATHER=father.o
SON=son.o

.c.o:
$(GCC) -c $<

all:$(FATHER) $(SON)
    $(GCC) $(FATHER) -o father
    $(GCC) $(SON) -o son

clean:
    rm -f *.o core

clobber:clean
    rm -f father son

```

---

请看执行时图 S6.4-1，我们可以看到子进程确实执行成功了，并且打印出了提示消息，父进程也打印出了提示消息。完全符合我们所想的。

```

[root@localhost S6-1]# make
gcc -c father.c
gcc -c son.c
gcc father.o -o father
gcc son.o -o son
[root@localhost S6-1]# ./father
Now,we begin fork me.
I am son process.
In son:The i is 0.
In son:The i is 1.
In son:The i is 2.
In son:The i is 3.
In son:The i is 4.
In father:The i is 0.
In father:The i is 1.
In father:The i is 2.
In father:The i is 3.
In father:The i is 4.
[root@localhost S6-1]#

```

图 S6.4-1

## 硬件部分（Hardware part）

### H1.操作系统的引导

#### H1.1 BIOS 的工作

当我们打开计算机的电源时，便会送一个电信号给主板，主板在收到这个信号后，接下来会将此电信号传给供电系统，于是供电系统开始工作，为整个系统供电，并送出一个电信号给 BIOS，通知 BIOS 供电系统已经准备完毕。随后 BIOS 启动一个程序，进行主机自检，主机自检的主要工作是确保系统的每一个部分都得到了电源支持，内存条、主板上的其它芯片、键盘、鼠标、磁盘控制器及一些 I/O 端口正常可用，此后，自检程序将控制权还给 BIOS。接下来 BIOS 读取 BIOS 中的相关设置，得到引导驱动器的顺序，然后依次检查，直到找到可以用来引导的驱动器（或说可以用来引导的磁盘，包括软盘、硬盘、光盘等），然后调用这个驱动器上磁盘的引导扇区中的引导块程序进行引导。

#### H1.2 操作系统的引导块程序

对于可以引导操作系统的设备来说，操作系统的引导块程序被放在该设备的第 0 磁道，0 磁头，1 扇区中，刚好大小为 512 个字节，并且这 512 个字节的最后两个字节必须是以 0x55AA 来结束的。在系统启动并且初始化成功后，BIOS 会主动的把该扇区中的内容全部读出后放在物理地址 0x0000:0x7c00 处，接下来 BIOS 会跳到该地址处执行。（也就是把控制权交给了操作系统，对于操作系统的编写者来说，他可以根据自己的想法写出自己的引导块程序，当然编写的引导块程序必须符合上面所说的要求）也便完成了操作系统的引导操作！

对于 linux1.0 核心，boot/bootsect.s 便是引导块程序，大小刚好是 512 个字节，并且也是以 0x55AA 为标志结束的。（具体请看 boot/ bootsect.s 代码）

我们可以仿造 linux 核心的 bootsect.s 写个什么也不做的引导块程序，只是不停的打印“Ah, I am a boot process!”。这样我们会有个感性的认识！

请看如下代码：

#### H1.2/boot.s

---

```
.text
entry _start
_start:
    jmp boot,#0x07C0
boot:
    mov ax,cs
    mov es,ax
label2:
    call print_msg
    jmp label2
```

---

```

print_msg:
    mov     ah,#0x03
    xor     bh,bh
    int     0x10
    mov     cx,#29
    mov     bx,#0x0007
    mov     bp,#msg
    mov     ax,#0x1301
    int     0x10
    ret

msg:
    .byte 13,10
    .ascii "Aha,I am a boot process !"
    .byte 13,10

.org 510
    .word 0xAA55

```

---

对 boot.s 的分析：

因为，直接在代码中用中文注释后，编译会有问题，所以放在这里对代码进行分析

entry _start	
_start:	! 这是必须要有进入点，因为 ld86 在连接程序时需要它
jmpi boot,#0x07C0	! 这句执行的目的有两个，一：间接跳转到 boot, ! 二：设置 cs=0x07C0 ! 因为 BIOS 会将 boot.s 加载到 0x0000:0x7c00
print_msg:	! 用于打印在屏幕上消息，就是一些 BIOS 中断的调用！
.org 510	! 用于代码的定位，距离 boot.s 开始处 510 个字节
.word 0xAA55	! 510-511 是 0xAA55

## H1.2/Makefile

---

```

AS86 = as86 -0 -a
LD86 = ld86 -0 -d -s
OBJ  = boot

```

```

.SUFFIXES:
$(AS86) -o $*.o $<

```

```

all:$(OBJ)

$(OBJ):boot.o
    $(LD86) -o $(OBJ) boot.o
boot.o:boot.s

disk:$(OBJ)
    dd if=./boot of=/dev/fd0 seek=0 bs=512 count=1

clean:
    rm -f *.o core
clobber:clean
    rm -f $(OBJ)

```

---

编译及执行情况请看图 H1.2-1, 图 H1.2-2

```

[root@localhost gdh]# ls -l
total 8
-rw-r--r--    1 gdh      gdh          324 Aug 30 19:07 boot.s
-rw-r--r--    1 gdh      gdh          256 Aug 30 18:55 Makefile
[root@localhost gdh]# make disk
as86 -O -a -o boot.o boot.s
ld86 -O -d -s -o boot boot.o
dd if=./boot of=/dev/fd0 seek=0 bs=512 count=1
1+0 records in
1+0 records out
[root@localhost gdh]# ls -l
total 16
-rwxr-xr-x    1 root     root        512 Aug 30 19:09 boot
-rw-r--r--    1 root     root       169 Aug 30 19:09 boot.o
-rw-r--r--    1 gdh      gdh          324 Aug 30 19:07 boot.s
-rw-r--r--    1 gdh      gdh          256 Aug 30 18:55 Makefile
[root@localhost gdh]# _

```

图 H1.2-1

请注意！在执行 make disk 时，要插入一个软盘！写入成功后，用该软盘就可以启动计算机，执行结果可以看下图（图 H1.2-2）

图 H1.2-2

## H2. X86 CPU 寻址简介

## H2.1 实模式

在 8086/8088 X86 CPU 只有 20 根地址线，所以可以寻址的地址空间也只有 1M ( $2^{20}$ ) 字节。它包括四个 16 位数据寄存器，二个 16 位指针寄存器，二个 16 位变址寄存器，一个 16 位指令指针，四个 16 位段寄存器，一个 16 位标志寄存器！

图 H2.1-1 数据寄存器

AH	AL	AX
BH	BL	BX
CH	CL	CX
DH	DL	DX

图 H2.1-1

图 H2.1-2 指针寄存器

SP	堆栈指针
BP	基址指针

图 H2.1-2

图 H2.1-3 变址寄存器



图 H2.1-3

图 H2.1-4 变址寄存器



图 H2.1-4

图 H2.1-5 变址寄存器



图 H2.1-5

通过图 H2.1-1 数据寄存器，可以看到他们都是 16 位的寄存器，所以最大可访问的地址空间只有 64K，那么我们怎么做才可以访问全部的 1M 地址空间呢？答案是通过对存储器分段。

我们可以根据自己的需要来把 1M 字节地址空间划分成若干逻辑段。每个逻辑段必须满足如下两个条件：

- 1) 逻辑段的开始地址必须是 16 的倍数
- 2) 逻辑段的最大长度为 64K

根据这两个条件我们可以得出 1M 地址空间最多可以划分成 64K 个逻辑段，最少也要划分成 16 个逻辑段。划分出的逻辑段可以相连，也可以不相连，甚至还可以部分重叠。这种分段的方法不仅有利于实现寻址 1M 字节空间，而且也十分有利于对 1M 字节存储空间的管理。对实现程序的重定位和浮动，对实现代码数据的隔离，对充分利用存储空间，这种方法都有益。

## H2.2 实模式方式下物理地址的形成

段起始地址必须是 16 的倍数，形如 0xXXXX0。我们可以把 20 位的段起始地址，压缩表示成 16 位的 XXXX 形式。

我们把 20 位段起始地址的高 16 位 XXXX 称为段值。很显然段起始地址便是段值乘以 16（即左移 4 位）。

我们把存储单元的地址与所在段的段起始地址的差值称为段内偏移。在一个段内，通过段内偏移便可以访问存储单元。所以我们可以得到在实模式下，存储单元的物理地址等于段起始地址加上段内偏移。于是我们利用如下形式来表示存储单元的逻辑地址

段起始地址：段内偏移

根据逻辑地址可以得到物理地址的计算方法

物理地址 = 段起始地址 X 16 + 段内偏移

假设逻辑地址为：0x3A56:0x9854，那么根据上面的公式可以计算出物理地址

$$\text{物理地址} = 0x3A56 \times 16 + 0x9854 = 0x3A560 + 0x9854 = 0x43DB4$$

## H2.3 保护模式

80386 CPU 有 32 根地址线，在保护方式下，它们都能发挥作用，所以可寻址的物理地址空间高达 4G 字节。在以 80386 及其以上处理器为 CPU 的 PC 兼容机系统中，把地址在 1M 以下的内存称为常规内存，把地址在 1M 以上的内存称为扩展内存。

80386 还要对实现虚拟存储器提供支持。虽然与 8086 可寻址的 1M 字节物理地址空间相比，80386 可寻址的物理地址空间可谓很大，但实际的微机系统不可能安装如此大的物理内存。所以，为了运行大型程序和真正实现多任务，必须采用虚拟存储器。虚拟存储器是一种软硬件结合的技术，用于提供比在计算机系统中实际可以使用的物理主存储器大得多的存储空间。这样，程序员在编写程序时不用考虑计算机中物理存储器的实际容量。80386 还要对存放在存储器中的代码及数据的共享和保护提供支持。任务甲和任务乙并存，任务甲和任务乙必须隔离，以免相互影响。但它们又可能要共享部分代码和数据。所以，80386 既要支持任务隔离，又要支持可共享代码和数据的共享，还要支持特权保护。

## H2.4 保护模式方式下物理地址的形成

保护方式下的虚拟存储器由大小可变的存储块构成，这样的存储块称为段。80386 采用称为描述符的数据来描述段的位置、大小和使用情况。虚拟存储器的地址(逻辑地址)由指示描述符的选择子和段内偏移两部分构成，这样的地址集合称为虚拟地址空间。80386 支持的虚拟地址空间可达 64T 字节。程序员编写程序时使用的存储地址空间是虚拟地址空间，所以，他们可认为有足够的存储空间可供使用。显然，只有在物理存储器中的程序才能运行，只有在物理存储器中的数据才能访问。因此，虚拟地址空间必须映射到物理地址空间，二维的虚拟地址必须转化成一维的物理地址。由于物理地址空间远小于虚拟地址空间，所以只有虚拟地址空间中的部分可以映射到物理地址空间。由于物理存储器的大小要远小于物理地址空间，所以只有上述部分中的部分才能真正映射到物理存储器。每一个任务有一个虚拟地址空间。为了避免多个并行任务的多个虚拟地址空间直接映射到同一个物理地址空间，采用线性地址空间隔离虚拟地址空间和物理地址空间。线性地址空间由一维的线性地址构成，线性地址空间和物理地址空间对等。线性地址 32 位长，线性地址空间容量为 4G 字节。

80386 分两步实现虚拟地址空间到物理地址空间的映射，也就是分两步实现虚拟地址到物理地址的转换，但第二步是可选的。下图(图 H2.4-1)是地址映射转换的示意图。



分页管理机制是可选的，这取决于系统的编写者

图 H2.4-1

通过描述符表和描述符，分段管理机制实现虚拟地址空间到线性地址空间的映射，实现把二维的虚拟地址转换为一维的线性地址。这一步总是存在的。

分页管理机制把线性地址空间和物理地址空间分别划分为大小相同的块，这样的块称为页。通过在线性地址空间的页与物理地址空间的页建立之间建立的映射表，分页管理机制实现线性地址空间到物理地址空间的映射，实现线性地址到物理地址的转换。分页管理机制是可选的，在不采用分页管理机制时，线性地址空间就等同于物理地址空间，线性地址就等于物理地址。

分段管理机制所使用的可变大小的块，分段管理机制比较适宜处理复杂系统的逻辑分段。存储块的大小可以根据适当的逻辑含义进行定义，而不用考虑固定大小的页所强加的人为限制。每个段可作为独立的单位处理，以简化段的保护及共享。分页机制使用的固定大小的块最适合于管理物理存储器，无论是管理内存还是外存都同样有效。分页管理机制能够有效地支持实现虚拟存储器。

段及分页这两种机制是两种不同的转换机制，是整个地址转换函数的不同的转换级。虽然两种机制都利用存储在主存储器中的转换表，但这些表具有独立的结构。事实上，段表存储在线性地址空间，而页表存储在物理地址空间。因此，段转换表可由分页机制重新进行定位而不需段机制的参与。段转换机制把虚拟地址转换为线性地址，并在线性地址中访问段转换机制的表格，而不会觉察分页机制已把线性地址转换为物理地址。类似地，分页机制对于程序产生的地址所使用的虚拟地址空间一无所知。分页机制只是直接地把线性地址转换为物理地址，并且在物理地址中访问转换表格，并不知道虚拟地址空间的存在，甚至不知道段转换机制的存在。

## H3. IDT & GDT & LDT

### H3.1 IDT

在 IDT 中，可以包含 3 种类型的描述符

- Task-gate descriptor (任务门描述符)
- Interrupt-gate descriptor (中断门描述符)
- Trap-gate descriptor (陷阱门描述符)

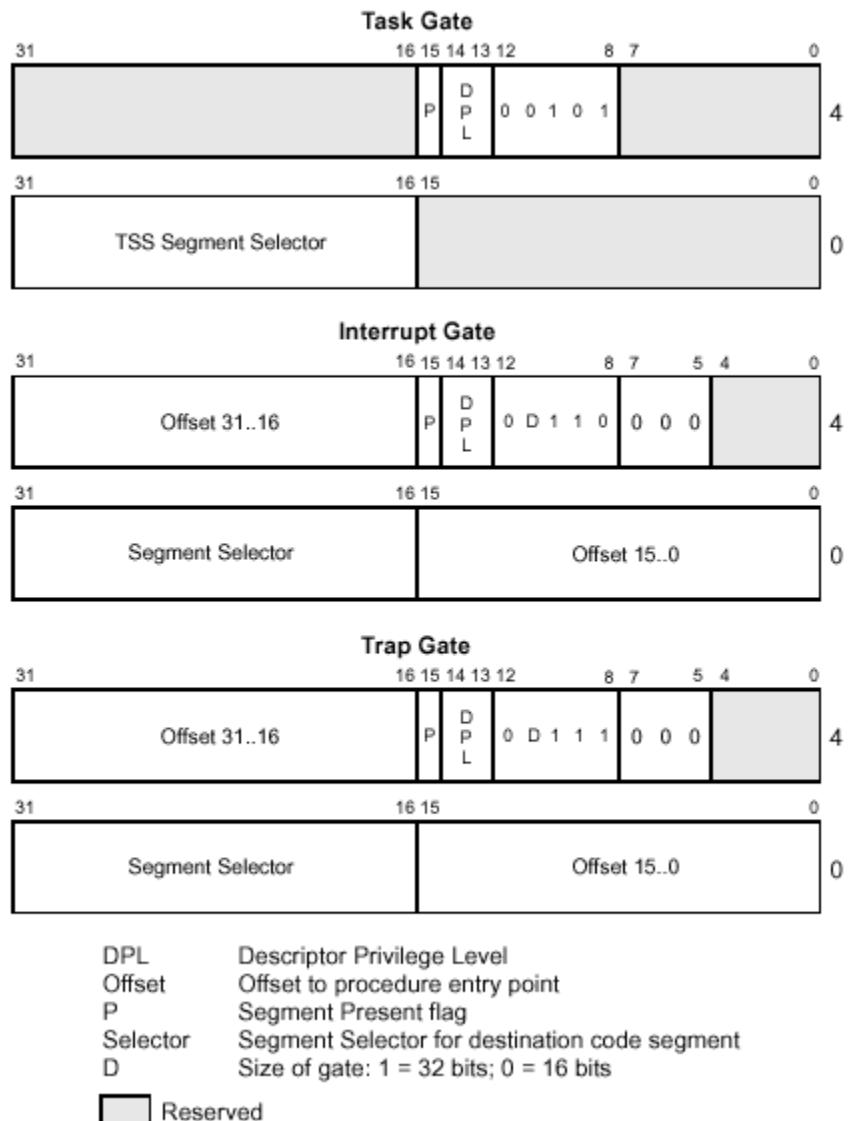


图 H3.1-1

Interrupts/Exceptions 应该使用 Interrupt Gate 和 TrapGate，它们之间的唯一区别就是：

当调用 InterruptGate 时，Interrupt 会被 CPU 自动禁止；而调用 Trap Gate 时，CPU 则不会禁止或打开中断，而是保留它原来的样子。

TaskGate 一种通过硬件实现任务切换，将 ISR 作为一个 Task 的方法，我们在处理

Interrupts/Excetpions 的时候，通常不会用到这种方法。

由于 IDT 最多可拥有 256 个门描述符

我们看看这 256 个中断的分布

异常 常 览 表	向量号	异常名称	异常类型	出错代码	相关指令
	0	除法出错	故障	无	DIV, IDIV
	1	调试异常	故障/陷阱	无	任何指令
	3	单字节 INT3	陷阱	无	INT 3
	4	溢出	陷阱	无	INTO
	5	边界检查	故障	无	BOUNT
	6	非法操作码	故障	无	非法指令编码或操作数
	7	设备不可用	故障	无	浮点指令或 WAIT
	8	双重故障	中止	有	任何指令
	9	协处理器段越界	中止	无	访问存储器的浮点指令
	0AH	无效 TSS 异常	故障	有	JMP、CALL、IRET 或中断
	0BH	段不存在	故障	有	装载段寄存器的指令
	0CH	堆栈段异常	故障	有	装载 SS 寄存器的任何指令、对 SS 寻址的段访问的任何指令
	0DH	通用保护异常	故障	有	任何特权指令、任何访问存储器的指令
	0EH	页异常	故障	有	任何访问存储器的指令
	10H	协处理器出错	故障	无	浮点指令或 WAIT
	11H—OFFH	软中断	陷阱	无	INT n

图 H3.1-2

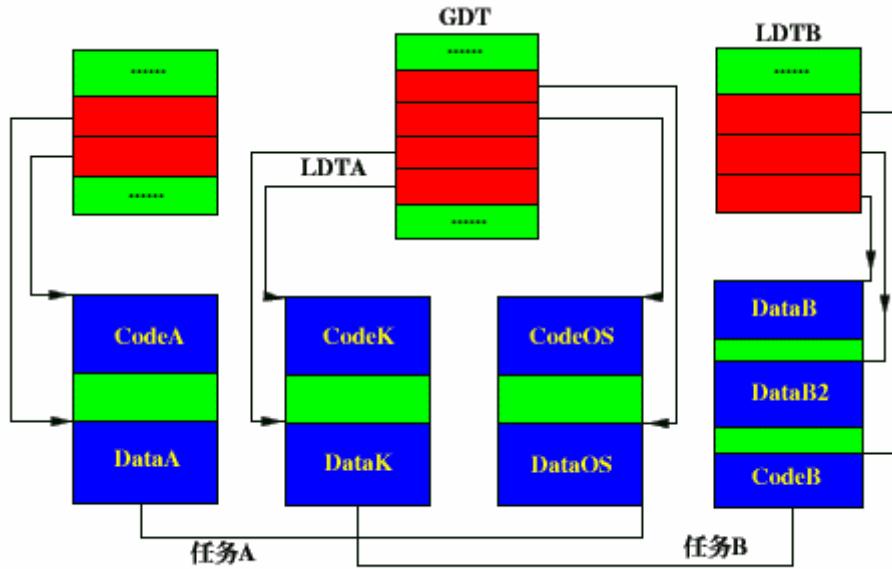
我们可以看到 0x00—0x10 是 Intel 预先保留的中断向量，从 0x11—0xFF，可以被系统的编写者自己去决定用来做些什么事情，linux 1.0 核心中用了第 0x80 号作为其所有系统调用的入口。

## H3.2 GDT & LDT

全局描述符表 GDT 含有每一个任务都可能或可以访问的段的描述符，通常包含描述操作系统所使用的代码段、数据段和堆栈段的描述符，也包含多种特殊数据段描述符，如各个用于描述任务 LDT 的特殊数据段等。在任务切换时，并不切换 GDT。

通过 LDT 可以使各个任务私有的各个段与其它任务相隔离，从而达到受保护的目的。通过 GDT 可以使各任务都需要使用的段能够被共享。下图给出了任务 A 和任务 B 所涉及的有关段既隔离受保护，又合用共享的情况。通过任务 A 的局部描述符表 LDTA 和任务 B 的局部描述符表 LDTB，把任务 A 所私有的代码段 CodeA 及数据段 DataA 与任务 B 所私有的代码段 CodeB 和数据段 DataB 及 DataB2 隔离，但任务 A 和任务 B 通过全局描述符表 GDT

共享代码段 CodeK 及 CodeOS 和数据段 DataK 及 DataOS。



一个任务可使用的整个虚拟地址空间分为相等的两半，一半空间的描述符在全局描述符表中，另一半空间的描述符在局部描述符表中。由于全局和局部描述符表都可以包含多达 8192 个描述符，而每个描述符所描述的段的最大值可达 4G 字节，因此最大的虚拟地址空间可为：  
 $4\text{GB} \times 8192 \times 2 = 64\text{MMB} = 64\text{TB}$

## H4.8259A 可编程中断控制器

### H4.1 8259A 芯片简介

一个 8259A 芯片的可以接最多 8 个中断源，但由于可以将 2 个或多个 8259A 芯片级连 (cascade)，并且最多可以级连到 9 个，所以最多可以接 64 个中断源。早期，IBM PC/XT 只有 1 个 8259A，但设计师们马上意识到这是不够的，于是到了 IBM PC/AT，8259A 被增加到 2 个以适应更多外部设备的需要，其中一个被称作 Master，另外一个被称作 Slave，Slave 以级连的方式连接在 Master 上。如今绝大多数的 PC 都拥有两个 8259A，这样 最多可以接收 15 个中断源

8259A 芯片中有如下几个内部寄存器：

- 1) Interrupt Mask Register (IMR)
- 2) Interrupt Request Register (IRR)
- 3) InService Register (ISR)

IMR 被用作过滤被屏蔽的中断；IRR 被用作暂时放置未被进一步处理的 Interrupt；当一个 Interrupt 正在被 CPU 处理时，此中断被放置在 ISR 中。除了这几个寄存器之外，8259A 还

有一个单元叫做 Priority Resolver，当多个中断同时发生时，PriorityResolver 根据它们的优先级，将高优先级者优先传递给 CPU。

## H4.2 8259A 芯片对的中断处理过程

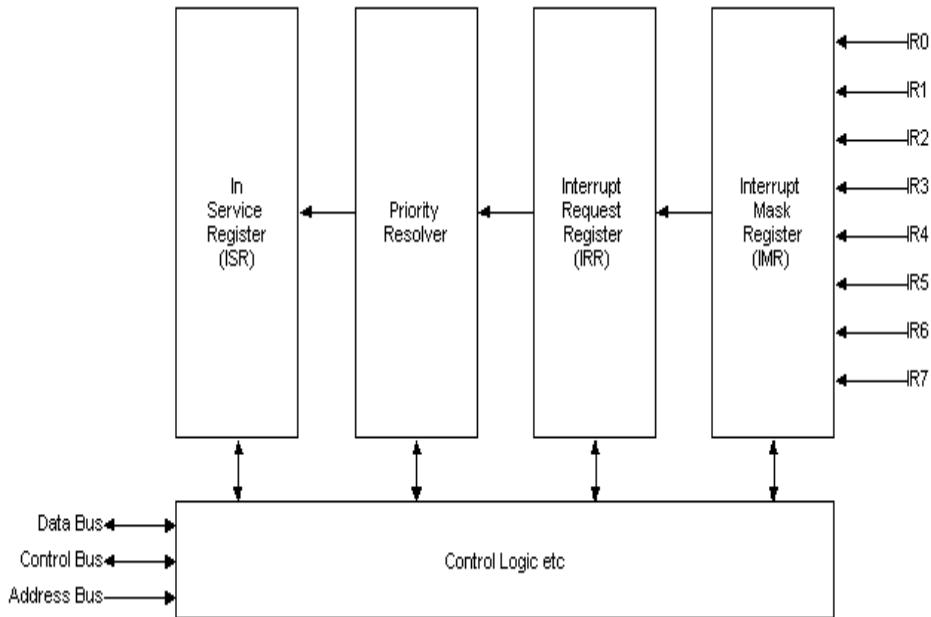


图 H4.2-1

当一个中断请求从 IR0 到 IR7 中的某根线到达 IMR 时，IMR 首先判断此 IR 是否被屏蔽，如果被屏蔽，则此中断请求被丢弃；否则，则将其放入 IRR 中。在此中断请求不能进行下一步处理之前，它一直被放在 IRR 中。一旦发现处理中断的时机已到，Priority Resolver 将从所有被放置于 IRR 中的中断中挑选出一个优先级最高的中断，将其传递给 CPU 去处理。IR 号越低的中断优先级别越高，比如 IR0 的优先级别是最高的。

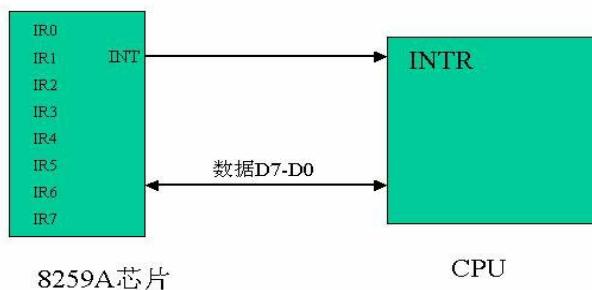


图 H4.2-2

(请看图图 H4.2-2) 8259A 通过发送一个 INTR(Interrupt Request)信号给 CPU, 通知 CPU 有一个中断到达。CPU 收到这个信号后, 会暂停执行下一条指令, 然后发送一个 INTA(Interrupt Acknowledge)信号给 8259A。8259A 收到这个信号之后, 马上将 ISR 中对应此中断请求的 Bit 设置, 同时 IRR 中相应的 bit 会被 reset。比如, 如果当前的中断请求是 IR3 的话, 那么 ISR 中的 bit-3 就会被设置, IRR 中 IR3 对应的 bit 就会被 reset。这表示此中断请求正在被 CPU 处理, 而不是正在等待 CPU 处理。随后, CPU 会再次发送一个 INTA 信号给 8259A, 要求它告诉 CPU 此中断请求的中断向量是什么, 这是一个从 0 到 255 的一个数。8259A 根据被设置的起始向量号 (起始向量号通过中断控制字 ICW2 被初始化) 加上中断请求号计算出中断向量号, 并将其放置在 Data Bus 上。比如被初始化的起始向量号为 8, 当前的中断请求为 IR3, 则计算出的中断向量为  $8+3=11$ 。CPU 从 Data Bus 上得到这个中断向量之后, 就去 IDT 中找到相应的中断服务程序 ISR, 并调用它。如果 8259A 的 End of Interrupt (EOI) 通知被设定位人工模式, 那么当 ISR 处理完该处理的事情之后, 应该发送一个 EOI 给 8259A。8259A 得到 EOI 通知之后, ISR 寄存器中对应于此中断请求的 Bit 会被 Reset。

如果 8259A 的 End of Interrupt (EOI) 通知被设定位自动模式, 那么在第 2 个 INTA 信号收到后, 8259A ISR 寄存器中对应于此中断请求的 Bit 就会被 Reset。在此期间, 如果又有新的中断请求到达, 并被放置于 IRR 中, 如果这些新的中断请求中有比在 ISR 寄存器中放置的所有中断优先级别还高的话, 那么这些高优先级别的中断请求将会被马上按照上述过程进行处理; 否则, 这些中断将会被放在 IRR 中, 直到 ISR 中高优先级别的中断被处理结束, 也就是说知道 ISR 寄存器中高优先级别的 bit 被 Reset 为止。

### H4.3 8259A 编程方式

8259A 芯片都有两个 I/O ports, 系统程序员可以通过它们对 8259A 进行编程, 在目前的 PC 机上, 有两个 8259A 中断控制芯片, 分别称为主控制芯片 (主片: 两个端口地址是 0x20, 0x21), 从控制芯片 (从片: 两个端口地址是 0xA0, 0xA1)。

为了可以对 8259A 编程, 定义了两个专用的指令用来对其初始化和写控制命令。这两个指令是:

- 1) ICW 用来初始化 8259A 芯片
- 2) OCW 用来向 8259A 芯片发布命令, 以对其进行控制。OCW 可以在 8259A 被初始化之后的任何时候被使用。

下表 (表 H4.3-1) 中列出主 8259A 的 I/O 端口地址, 以及通过它们所能操作的寄存器

Address	Read/Write	Function
0x20	Write	Initialization Command Word 1 (ICW1)
	Write	Operation Command Word 2 (OCW2)
	Write	Operation Command Word 3 (OCW3)
	Read	Interrupt Request Register (IRR)
	Read	In-Service Register (ISR)
0x21	Write	Initialization Command Word 2 (ICW2)

	Write	Initialization Command Word 3 (ICW3)
	Write	Initialization Command Word 4 (ICW4)
	Read/Write	Interrupt Mask Register (IMR)

表 H4.3-1

下表 (表 H4.3-2) 中列出从 8259A 的 I/O 端口地址, 以及通过它们所能操作的寄存器

Address	Read/Write	Function
0xA0	Write	Initialization Command Word 1 (ICW1)
	Write	Operation Command Word 2 (OCW2)
	Write	Operation Command Word 3 (OCW3)
	Read	Interrupt Request Register (IRR)
	Read	In-Service Register (ISR)
0xA1	Write	Initialization Command Word 2 (ICW2)
	Write	Initialization Command Word 3 (ICW3)
	Write	Initialization Command Word 4 (ICW4)
	Read/Write	Interrupt Mask Register (IMR)

表 H4.3-2

任何时候, 只要向某一个 8259A 的第一个端口(0x20 for Master, and 0xA0 for Slave)写入的命令的 bit-4(从 0 算起)为 1, 那么这个 8259A 就认为这是一个 ICW1; 而一旦一个 8259A 收到一个 ICW1, 它就认为一个初始化序列开始了。你可以通过对照上边的表和后面的表, 第一端口可写的有 ICW1, OCW2 和 OCW3。而 ICW1 的 bit-4 要求必须是 1, 但 OCW2 和 OCW3 的 bit-4 要求必须是 0。

8259A 的初始化流程协议如下图 H4.3-1 所示, 系统程序员对其进行初始化时必须遵守此协议:

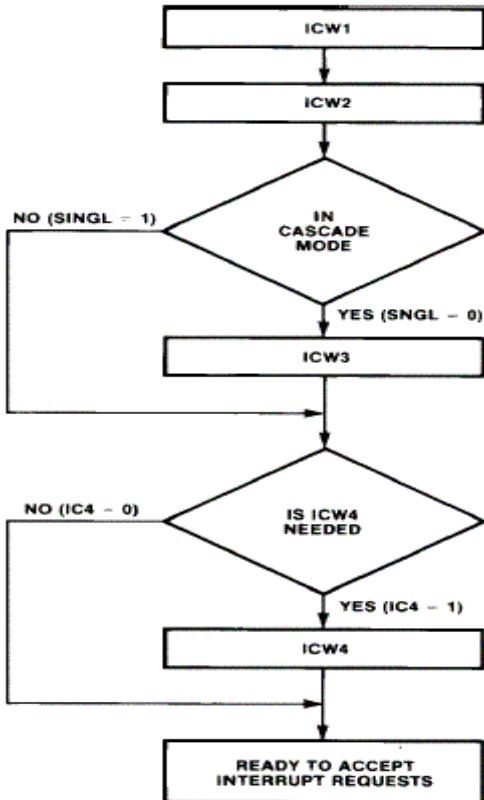


图 H4.3-1

**ICW1**

Bit(s)	Function
7:5	Interrupt Vector Addresses for MCS-80/85 Mode.
4	Must be set to 1 for ICW1
3	1 Level Triggered Interrupts 0 Edge Triggered Interrupts
2	1 Call Address Interval of 4 0 Call Address Interval of 8
1 (SINGL)	1 Single PIC 0 Cascaded PICs
0 (IC4)	1 Will be Sending ICW4 0 Don't need ICW4

Initialization Command Word 1 (ICW1)

对于x86,bit-0必须被设置为1;由于当今的IBM PC上都有两个级连的8259A,所以bit-1应该被设置为0;由于bit-2是为MCS-80/85服务的,我们将其设置为0;bit-3也设置为0;bit-4被要求必须设置为1;bit5:7是为MCS-80/85服务的,对于x86,应将全部将其设为0。

所以，在 x86 系统上，ICW1 应该被设置为二进制  $00010001 = 0x11$ 。

## ICW2

Bit	80x86Mode
7	I7
6	I6
5	I5
4	I4
3	I3
2	0
1	0
0	0

Initialization Command Word 2 (ICW2)

ICW2 被用作指定本 8259A 中的中断请求的起始中断向量，bit0:3 必须被设为 0；所以，其起始中断向量必须是 8 的倍数。比如，我们的 os 的设计讲来自于 Master8259A 的 8 个中断请求放在 IDT 的第 32 (从 0 开始计) 个位置到第 39 个位置，则我们应该将 ICW2 设为 0x20。

这样，当将来此 8259A 上接收到一个 IRQ 时，其低 3 位会被自动填充为 IRQ 号。比如，其收到一个 IRQ6，将 6 自动填充到后 3 位，则生成的向量号为 0x26。8259A 会在收到 CPU 发来的第二个 INTA 信号之后，将生成的向量号放到 DataBus 上。

## ICW3

Master 8259A 和 Slave8259A 有不同的 ICW3 格式。

Bit	Function
7	IR7 is connected to a Slave
6	IR6 is connected to a Slave
5	IR5 is connected to a Slave
4	IR4 is connected to a Slave
3	IR3 is connected to a Slave
2	IR2 is connected to a Slave
1	IR1 is connected to a Slave
0	IR0 is connected to a Slave

Initialization Command Word 3 for Master8259A (ICW3)

Slave 8259A 被接在 Master8259A 的那个 IRQ 上，则相应的位就被设置为 1，其余的位都被设置为 0。在 IBM PC 上，Slave 8259A 被接在 Master8259A 的 IRQ2 上，则此 ICW3 的值应该被设置为二进制  $00000100 = 0x04$ 。

Bit(s)	Function
7	Reserved. Set to0
6	Reserved. Set to0
5	Reserved. Set to0
4	Reserved. Set to0
3	Reserved. Set to0
2:0	<i>Slave ID</i>
000	Slave 0
001	Slave 1
010	Slave 2
011	Slave 3
100	Slave 4
101	Slave 5
110	Slave 6
111	Slave7

Initialization Command Word 3 for Slaves(ICW3)

Slave8259A 的 ICW3 的 bit3:7 被保留，必须被设为 0；而 bit0:2 被设置为此 Slave 8259A 被接在 Master8259A 的哪个 IRQ 上。比如，在 IBM PC 上，Slave 8259A 被接在 Master8259A 的 IRQ2 上，则此 ICW3 应被设为 0x02。

## ICW4

Bit(s)	Function
7	Reserved. Set to0
6	Reserved. Set to0
5	Reserved. Set to0
4	1 Special Fully Nested Mode
	0 Not Special Fully Nested Mode
3:2	0x Non - Buffered Mode
	10 Buffered Mode - Slave
	11 Buffered Mode - Master
1	1 Auto EOI
	0 Normal EOI
0	1 8086/8080 Mode
	0 MCS-80/85

Initialization Command Word 4 (ICW4)

在 80x86 模式下，我们不需要使用 8259A 的特殊功能，因此我们将 bit1:4 都设为 0，这意味使用默认的 Full NestedMode，不使用 Buffer，以及手动 EOI 模式；我们只需要将 bit-0 设为 1，这也正是我们 ICW0 处提到的我们为什么必须要 ICW4 的原因。所以 ICW4 的值应该被设为 0x01。

我们看看linux1.0 核心初始化的代码：(代码取之于boot/setup.s)

```

208     mov     al,#0x11           ! initialization sequence
! 0x11 表示初始化命令开始，是 ICW1 命令字，表示边
! 沿触发、多片 8259 级连、最后要发送 ICW4 命令字。
209     out    #0x20,al           ! send it to 8259A-1
! 发送到 8259A 主芯片。
210     call   delay
211     out    #0xA0,al           ! and to 8259A-2
! 发送到 8259A 从芯片。
212     call   delay
213     mov    al,#0x20           ! start of hardware int's
(0x20)
214     out    #0x21,al
! 送主芯片 ICW2 命令字，起始中断号，要送奇地址。
215     call   delay
216     mov    al,#0x28           ! start of hardware int's 2
(0x28)
217     out    #0xA1,al
! 送从芯片 ICW2 命令字，从芯片的起始中断号。
218     call   delay
219     mov    al,#0x04           ! 8259-1 is master
220     out    #0x21,al
! 送主芯片 ICW3 命令字，主芯片的 IR2 连从芯片 INT。
221     call   delay
222     mov    al,#0x02           ! 8259-2 is slave
223     out    #0xA1,al
! 送从芯片 ICW3 命令字，表示从芯片的 INT 连到主芯
224     call   delay
225     mov    al,#0x01           ! 8086 mode for both
226     out    #0x21,al
! 送主芯片 ICW4 命令字。8086 模式；普通 EOI 方式，
! 需发送指令来复位。初始化结束，芯片就绪。
227     call   delay
228     out    #0xA1,al           ! 送从芯片 ICW4 命令字，内容同上。
229     call   delay
230     mov    al,#0xFF           ! mask off all interrupts for
now
231     out    #0xA1,al ! 屏蔽主芯片所有中断请求
232     call   delay

```

```

233      mov     al,#0xFB           ! mask all irq's but irq2
which
234      out     #0x21,al          ! is cascaded
           ! 屏蔽从芯片所有中断请求, 除了 irq2

```

## H5.I/O 端口及指令

与 PC 相连的外部设备(比如液晶显示器,激光打印机),要想实现自己的功能,都必须接受 PC 的管理,让 PC 来控制其要做的工作.那么 PC 为什么能够控制这些外部设备呢?原因是在这些设备中有一些寄存器,这些寄存器组被称为接口,也被成为 I/O 端口(当然这些接口的具体值是事先规定好的,要不然我们怎么能够写它们呢?).CPU 还专门提供了 I/O 指令来负责对这些接口的存取.所以也就完成了对外部设备的控制了

### H5.1I/O 端口

在 X86CPU 上,I/O 端口地址和系统中的内存地址是分开的,即不会占用系统的内存地址空间, I/O 端口地址空间范围是 64K, 因此可以接 8 位端口 64K 个, 那么 16 位端口 32K 个。

### H5.2I/O 指令

输出指令:

格式为: out 累加器,端口地址

例如: "outb %%al,%%dx", 把 al 中的内容输出到 dx 寄存器所表示的端口中。

输入指令:

格式为: in 端口地址, 累加器

例如: "inb %%dx,%%al", 从 dx 寄存器所表示的端口中取出内容送入 al 中。

另外还有一种直接存储器存取方式 (即 DMA) 的数据交换方法, 该方法是通过 DMA 硬件控制器来实现的。该硬件首先向 CPU 申请占用系统总线, 获得总线的控制权后, 便开始数据的传送, 数据传送结束后再次通知 CPU 结束对系统总线的控制权释放。从而结束了数据的传送。

附 I/O 端口地址分配表

I/O 的地址	对应的功能
00-0F	DMA 控制器 8237
20-3F	8259A
40-5F	可编程中断计时器
60-63	8255A PPI
70-71	CMOS RAM
81-8F	DMA 页表地址寄存器

93-9F	DMA 控制器
A0-A1	可编程中断控制器 2
C0-CE	DMA 通道, 内存、传输地址寄存器
F0-FF	协处理器
170-1F7	硬盘控制器
200-20F	游戏控制端口
278-27A	3 号并行口
2E0-2E0	EGA、VGA
2F8-2FE	2 号串口 (COM2)
320-324	硬盘适配器
366-36F	PC 网络
372-377	软盘适配器
378-37A	2 号并行口
380-38F	SDLC 及 BSC 通信
390-393	Cluster 适配器
3A0-3AF	BSC 通信
3B0-3BF	MDA 视频寄存器
3BC-3BE	1 号并行口
3C0-3CF	EGA/VGA 视频寄存器
3D0-3D7	CGA 视频寄存器
3F0-3F7	软盘控制寄存器
3F8-3FE	1 号串行口 (COM1)

## H6. 获取系统时间

在 Linux 核心中当前的时间，是通过对 CMOS RAM 的存储来获得的。在与 IBM 兼容的 PC 中安装有一块 RT/CMOS RAM 的芯片，全称是互补金属氧化物半导体随机存取存储器，它用于保存系统的配置情况，因为对其的供电是借助于一个纽扣电池，所以即使在我们关闭计算机后，系统的时间还是会继续累加的，这样在我们每次启动系统后都能够得到正确的时间。

### H6.1 CMOS RAM 分配表

CMOS RAM 共有 64 个字节的大小，请看下表对其的描述。

偏移	功能
0x00	秒

0x01	报警秒
0x02	分
0x03	报警分
0x04	时
0x05	报警时
0x06	星期
0x07	日
0x08	月
0x08	年
0x0A-0x0D	状态寄存器 A 到 D
0x0E	诊断状态
0x0F	停止状态
0x10	软盘驱动器类型
0x11	保留
0x12	硬盘驱动器类型
0x13	保留
0x14	设备标志
0x15	常规 RAM 容量低字节
0x16	常规 RAM 容量高字节
0x17	扩展 RAM 容量低字节
0x18	扩展 RAM 容量高字节
0x19-0x1A	硬驱类型扩展字节
0x1B-0x2D	保留
0x2E-0x2F	配置信息字节累加和
0x30-0x3F	其他（含世纪信息）

## H6.2 读取 CMOS RAM 表

读取 CMOS RAM 中的数据时，要分成两个步骤，首先把要读取的单元的地址送入端口 0x70 处，接下再从 0x71 端口读出。这时读出的内容便是我们想要的东西了。对于上表从偏移 0 x00 到偏移 0x0D 用于实时钟，我们直接用该值就行了，但是对于剩下的偏移读取，我们需要在加上 0x80 后，才可以读到。

读秒数代码：

```
movb 0,al      #要读取地址
outb al,$0x70  #送入 0x70
jmp 1f          #延时
1: jmp 1f      #延时
```

```
1:  inb $0x71,al      #读取值
```

## H6.3Linux 获取读取 CMOS RAM 的方式

这里我们看看 1.0 核心的 time\_init.

```
//该函数,用于获取系统的时间,我们看其中读取 CMOS 的方式
void time_init(void)
{
    struct mktm time;
    int i;

/* checking for Update-In-Progress could be done more elegantly
 * (using the "update finished"-interrupt for example), but that
 * would require excessive testing. promise I'll do that when I find
 * the time.          - Torsten
 */
/* read RTC exactly on falling edge of update flag */
for (i = 0 ; i < 1000000 ; i++) /* may take up to 1 second... */
    if (CMOS_READ(RTC_FREQ_SELECT) & RTC_UIP)
        break;
for (i = 0 ; i < 1000000 ; i++) /* must try at least 2.228 ms*/
    if (!(CMOS_READ(RTC_FREQ_SELECT) & RTC_UIP))
        break;
do { /* Isn't this overkill ? UIP above should guarantee consistency */
    time.sec = CMOS_READ(RTC_SECONDS);
    time.min = CMOS_READ(RTC_MINUTES);
    time.hour = CMOS_READ(RTC_HOURS);
    time.day = CMOS_READ(RTC_DAY_OF_MONTH);
    time.mon = CMOS_READ(RTC_MONTH);
    time.year = CMOS_READ(RTC_YEAR);
} while (time.sec != CMOS_READ(RTC_SECONDS));
if (!(CMOS_READ(RTC_CONTROL) & RTC_DM_BINARY) || RTC_ALWAYS_BCD)
{
    BCD_TO_BIN(time.sec);
    BCD_TO_BIN(time.min);
    BCD_TO_BIN(time.hour);
    BCD_TO_BIN(time.day);
    BCD_TO_BIN(time.mon);
    BCD_TO_BIN(time.year);
}
time.mon--;
```

```
xtime.tv_sec = kernel_mktime(&time);
}

#define CMOS_READ(addr) ({ \
outb_p(addr|0x80,0x70); \
inb_p(0x71); \
})

#define outb_p(val,port) \
((__builtin_constant_p((port)) && (port) < 256) ? \
__outbc_p((val),(port)) : \
__outb_p((val),(port)))

#define inb_p(port) \
((__builtin_constant_p((port)) && (port) < 256) ? \
__inbc_p(port) : \
__inb_p(port))
```

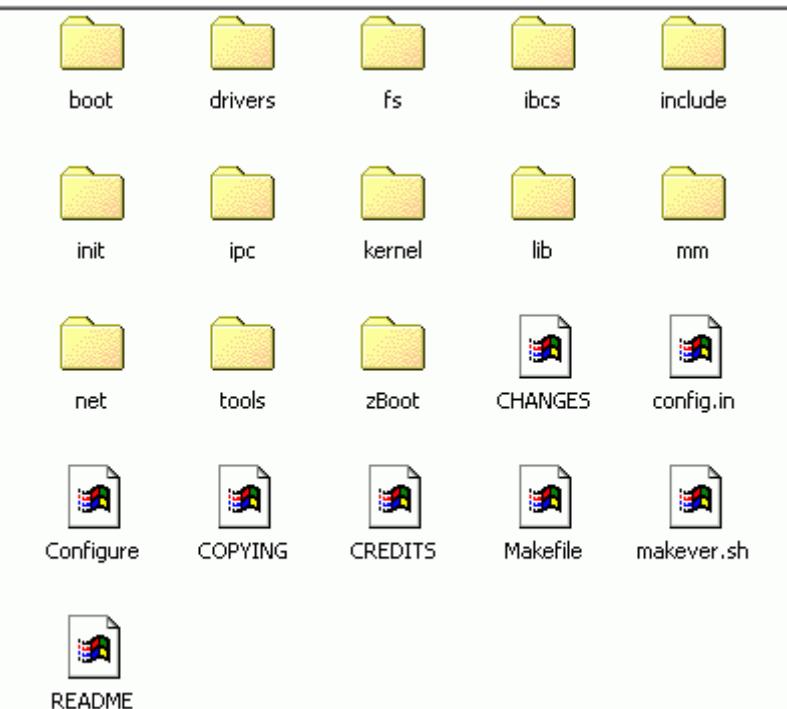
看上面的 CMOS\_READ 宏定义，我们可以看到先把地址送入 0x70 地址处，再从 0x71 处读出，也符合我们上面所说的。  
\_\_outbc\_p、\_\_outb\_p、\_\_inbc\_p、\_\_inb\_p 由 gcc 定义，有兴趣的读者请自己查看 gcc 的代码。（其实无非是包裹了 in、out 指令）

## 第二部分 代码分析 (Code analyzed)

本部分注释了从计算机的加电开始直到核心正常运作的全过程的代码(当然还有很少数的几个函数未做注释)。

## 引爆点

当我们解开 1.0 版本的 Linux 核心后（型如下图），可能会让我们很苦恼，从什么地方开始分析呢！



对于象 Linux 这样的大型工程来说！Makefile 文件非常的重要！可以说如果没有 Makefile 文件写大型工程是很难的，或者可以肯定的说是几乎不可能的。Makefile 文件定义了连接影象的方法！通过分析 Makefile 文件我们可以非常清楚的了解产生核心影象文件的过程！（关于操作系统的引导过程请看基础部分）。

下面我们看看整个工程的 Makefile 文件和 zBoot/Makefile！

## 整个核心工程 Makefile

```

1 VERSION = 1
    ! 版本号
2 PATCHLEVEL = 0
    ! 补丁号
3 ALPHA =
4
5 all: Version zImage
    ! Version 用于生成.config 和.depend 文件
    ! zImage 就是要生成的核心

```

! Version 依赖于 dummy,作用是删除 tools/version.h 和如果系统中没有  
 ! .config 和.depend 文件的话，帮助生成这两个文件  
 ! zImage要生成的影象文件名，1.0 版本的核心不支持非压缩的核心！

6

7.EXPORT\_ALL\_VARIABLES:  
 ! 导出所有的值，用于生成.config文件时，读取config.in文件

8

9 CONFIG\_SHELL := \$(shell if [ -x "\$\$BASH" ]; then echo \$\$BASH; \  
10 else if [ -x /bin/bash ]; then echo /bin/bash; \  
11 else echo sh; fi ; fi)

12

13#

14 # Make "config" the default target if there is no configuration file or  
15 # "depend" the target if there is no top-level dependency information.

16#  
 ! 如果没有 configuration 文件 make config 将会是缺省的目标，如果没有  
 ! 顶层的依赖信息depend将会是缺省的目标

17 ifeq (.config,\$(wildcard .config))  
18 include .config  
 ! 如果既包含了.config 文件，也包含了.depend 文件  
 ! 则 include 它们之。  
 ! 如果只包含了.config 文件，则让 CONFIGURATION=depend  
 ! 如果只包含了.depend文件，则让CONFIGURATION=config

19 ifeq (.depend,\$(wildcard .depend))  
20 include .depend  
21 else  
22 CONFIGURATION = depend  
23 endif  
24 else  
25 CONFIGURATION = config  
26 endif  
27

28 ifdef CONFIGURATION  
29 CONFIGURE = dummy  
30 endif  
31

32#  
33 # ROOT\_DEV specifies the default root-device when making the image.  
34 # This can be either FLOPPY, CURRENT, /dev/xxxx or empty, in which case  
35 # the default of FLOPPY is used by 'build'.  
36#  
 ! ROOT\_DEV 是我们制作影象文件时缺省的根设备。  
 ! 它可以是 FLOPPY, CURRENT  
 ! /dev/xxxx或者空，另外，缺省的FLOPPY被用于”build”

```

37
38 ROOT_DEV = CURRENT
    ! 根设备
39 #
40 #
41 # If you want to preset the SVGA mode, uncomment the next line and
42 # set SVGA_MODE to whatever number you want.
43 # Set it to -DSVGA_MODE=NORMAL_VGA if you just want the EGA/VGA mode.
44 # The number is the same as you would ordinarily press at bootup.
45 #

    ! 如果你想调整 SVGA 模式，注释掉下行并且可以设置成你想要的任何模式
    ! 如果你想要 EGA/VGA 模式的话可用设置-DSVGA_MODE=NORMAL_VGA
    ! 这个数字和你在启动时按下按键一样

46
47 SVGA_MODE= -DSVGA_MODE=NORMAL_VGA
48
49 #
50 # standard CFLAGS
51 #
52
53 CFLAGS = -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer -pipe
    ! 编译时用的选项

54
55 ifdef CONFIG_CPP
    ! 可以c++编译器编译
56 CFLAGS := $(CFLAGS) -x c++
57 endif
58
59 ifdef CONFIG_M486
    ! 处理器的定义，根据用户的选择可以
    ! 让gcc为对应的CPU类型产生优化的代码
60 CFLAGS := $(CFLAGS) -m486
61 else
62 CFLAGS := $(CFLAGS) -m386
63 endif
64
65 #
66 # if you want the ram-disk device, define this to be the
67 # size in blocks.
68 #

    ! 如果你想用虚拟盘，请定义这里的尺寸

69
70 #RAMDISK = -DRAMDISK=512
    ! 定义虚拟盘的大小

```

```

71
72 AS86 =as86 -O -a
      ! 8086 汇编器
73 LD86 =ld86 -O
      ! 8086 连接器
74
75 AS =as
      ! GUN汇编器
76 LD =ld
      ! GNU连接器
77 HOSTCC =gcc
      ! 定义GCC
78 CC =gcc -D__KERNEL__
      ! -D__KERNEL__ 编译核心选项
79 MAKE =make
80 CPP =$(CC) -E
81 AR =ar
82 STRIP =strip
83
      ! 从 84 行开始到 106 行，定义了最终要连接成zImage时要用的各个模块
84 ARCHIVES =kernel/kernel.o mm/mm.o fs/fs.o net/net.o ipc/ipc.o
85 FILESYSTEMS =fs/filesystems.a
86 DRIVERS =drivers/block/block.a \
87           drivers/char/char.a \
88           drivers/net/net.a \
89           ibcs(ibcs.o
90 LIBS =lib/lib.a
91 SUBDIRS =kernel drivers mm fs net ipc ibcs lib
92
93 KERNELHDRS =/usr/src/linux/include
94
95 ifdef CONFIG_SCSI
96 DRIVERS := $(DRIVERS) drivers/scsi/scsi.a
97 endif
98
99 ifdef CONFIG_SOUND
100 DRIVERS := $(DRIVERS) drivers/sound/sound.a
101 endif
102
103 ifdef CONFIG_MATH_EMULATION
104 DRIVERS := $(DRIVERS) drivers/FPU-emu/math.a
105 endif
106
107 .c.s:

```

```

108      $(CC) $(CFLAGS) -S -o $*.s $<
109 .s.o:
110      $(AS) -c -o $*.o $<
111 .c.o:
112      $(CC) $(CFLAGS) -c -o $*.o $<
113
114 Version: dummy
115      rm -f tools/version.h
116
117 config:
118      $(CONFIG_SHELL) Configure $(OPTS) < config.in
119      @if grep -s '^CONFIG_SOUND' .tmpconfig ; then \
120          $(MAKE) -C drivers/sound config; \
121      else : ; fi
122      mv .tmpconfig .config
123
124 linuxsubdirs: dummy
    ! 循环进入子目录编译
125      set -e; for i in $(SUBDIRS); do $(MAKE) -C $$i; done
126
127 tools/.version.h: tools/version.h
    ! 生成./version.h
128
129 tools/version.h: $(CONFIGURE) Makefile
    ! 生成tools/version.h
130      @./makever.sh
131      @echo '#define UTS_RELEASE '\"$(VERSION).$(PATCHLEVEL)$(ALPHA)\" >
tools/version.h
132      @echo '#define UTS_VERSION \"`cat .version` `date`\" >> tools/version.h
133      @echo '#define LINUX_COMPILE_TIME '\"`date +%T`\" >> tools/version.h
134      @echo '#define LINUX_COMPILE_BY '\"`whoami`\" >> tools/version.h
135      @echo '#define LINUX_COMPILE_HOST '\"`hostname`\" >> tools/version.h
136      @echo '#define LINUX_COMPILE_DOMAIN '\"`domainname`\" >> tools/version.h
137
138 tools/build: tools/build.c $(CONFIGURE)
    ! 生成tools/build
139      $(HOSTCC) $(CFLAGS) -o $@ $<
140
141 boot/head.o: $(CONFIGURE) boot/head.s
    ! 生成boot/head.o
142
143 boot/head.s: boot/head.S $(CONFIGURE) include/linux/tasks.h
    ! 生成boot/head.s
144      $(CPP) -traditional $< -o $@

```

```

145
146 tools/version.o: tools/version.c tools/version.h
    ! 生成tools/version.o
147
148 init/main.o: $(CONFIGURE) init/main.c
    ! 生成inti/main.o
149 $(CC) $(CFLAGS) $(PROFILING) -c -o $*.o $<
150
151 tools/system: boot/head.o init/main.o tools/version.o linuxsubdirs
    ! 在 1.0 核心中 tools/system 没有被使用，而是用了下面的 tools/zSystem,
    ! 所以从该行开始到 160 行我们都可不用关心
152 $(LD) $(LDFLAGS) -Ttext 1000 boot/head.o init/main.o tools/version.o \
153     $(ARCHIVES) \
154     $(FILESYSTEMS) \
155     $(DRIVERS) \
156     $(LIBS) \
157     -o tools/system
158 nm tools/zSystem | grep -v '\(compiled\)\|\(\.o$$\)\|\( a \)' | \
159     sort > System.map
160
161 boot/setup: boot/setup.o
162 $(LD86) -s -o $@ $<
163
164 boot/setup.o: boot/setup.s
    ! 生成boot/setup.o
165 $(AS86) -o $@ $<
166
167 boot/setup.s: boot/setup.S $(CONFIGURE) include/linux/config.h Makefile
    ! 生成boot/setup.s
168 $(CPP) -traditional $(SVGA_MODE) $(RAMDISK) $< -o $@
169
170 boot/bootsect: boot/bootsect.o
    ! 生成boot/bootsect
171 $(LD86) -s -o $@ $<
172
173 boot/bootsect.o: boot/bootsect.s
    ! 生成boot/bootset.o
174 $(AS86) -o $@ $<
175
176 boot/bootsect.s: boot/bootsect.S $(CONFIGURE) include/linux/config.h Makefile
    ! 生成boot/bootsect.s
177 $(CPP) -traditional $(SVGA_MODE) $(RAMDISK) $< -o $@
178
179 zBoot/zSystem: zBoot/*.c zBoot/*.S tools/zSystem

```

```

! 生成zBoot/zSystem
180 $(MAKE) -C zBoot
181
    ! 生成 zImage
    ! zImage = boot/bootsect + boot/setup + zBoot/zSystem (当然 zImage 并不是简单
    ! 的由这 3 个模块的叠加，而是由 tools/build 这个应用程序去除了 MINIX 头和
    ! GCC头后才叠加的，最终生成zImage
182 zImage: $(CONFIGURE) boot/bootsect boot/setup zBoot/zSystem tools/build
183     tools/build boot/bootsect boot/setup zBoot/zSystem $(ROOT_DEV) > zImage
184     sync
185
186 zdisk: zImage
    ! 将生成的zImage写入软盘
187     dd bs=8192 if=zImage of=/dev/fd0
188
189 zlilo: $(CONFIGURE) zImage
    ! 使用 lilo 启动核心，首先备份
    ! 系统原有的 vmlinuz 和 zSystem.map
    ! 然后把生成的zImage和zSystem.map，拷贝到“/”下！最后运行lilo安装lilo
190     if [ -f /vmlinuz ]; then mv /vmlinuz /vmlinuz.old; fi
191     if [ -f /zSystem.map ]; then mv /zSystem.map /zSystem.old; fi
192     cat zImage > /vmlinuz
193     cp zSystem.map /
194     if [ -x /sbin/lilo ]; then /sbin/lilo; else /etc/lilo/install; fi
195
    ! 生成 tools/zSystem
    ! tools/zSystem = boot/head.o + init/main.o + tools/version.o
    ! + 各个编译好的子目录下的模块，请注意生成的
    ! tools/zSystem模块代码段从 1M处开始编址
196 tools/zSystem: boot/head.o init/main.o tools/version.o linuxsubdirs
197     $(LD) $(LDFLAGS) -Ttext 100000 boot/head.o init/main.o tools/version.o \
198         $(ARCHIVES) \
199         $(FILESYSTEMS) \
200         $(DRIVERS) \
201         $(LIBS) \
202         -o tools/zSystem
203     nm tools/zSystem | grep -v '^(compiled)\|(.o$$)\|( a )'| \
204         sort > zSystem.map
205
    ! 从这开始到 226 行，是编译各个子目录的下模块的情况
206 fs: dummy
207     $(MAKE) linuxsubdirs SUBDIRS=fs
208
209 lib: dummy

```

```

210      $(MAKE) linuxsubdirs SUBDIRS=lib
211
212 mm: dummy
213      $(MAKE) linuxsubdirs SUBDIRS=mm
214
215 ipc: dummy
216      $(MAKE) linuxsubdirs SUBDIRS=ipc
217
218 kernel: dummy
219      $(MAKE) linuxsubdirs SUBDIRS=kernel
220
221 drivers: dummy
222      $(MAKE) linuxsubdirs SUBDIRS=drivers
223
224 net: dummy
225      $(MAKE) linuxsubdirs SUBDIRS=net
226
    ! 清除编译生成的目标文件

227 clean:
228      rm -f kernel/ksyms.lst
229      rm -f core `find . -name '*.[oas]' -print`
230      rm -f core `find . -name 'core' -print`
231      rm -f zImage zSystem.map tools/zSystem tools/system
232      rm -f Image System.map boot/bootsect boot/setup
233      rm -f zBoot/zSystem zBoot/xtract zBoot/piggyback
234      rm -f .tmp* drivers/sound/configure
235      rm -f init/*.o tools/build boot/*.o tools/*.o
236
    ! 清除编译生成的目标文件，还包括生成的h文件和.config及.depend文件

237 mrproper: clean
238      rm -f include/linux/autoconf.h tools/version.h
239      rm -f drivers/sound/local.h
240      rm -f .version .config* config.old
241      rm -f .depend `find . -name .depend -print`
242
243 distclean: mrproper
244
    ! 备份核心源代码

245 backup: mrproper
246      cd .. && tar cf - linux | gzip -9 > backup.gz
247      sync
248
    ! 产生.depend文件

249 depend dep:

```

```

250 touch tools/version.h
251 for i in init/*.c;do echo -n "init/"$(CPP) -M $$i;done > .tmpdepend
252 for i in tools/*.c;do echo -n "tools/"$(CPP) -M $$i;done >> .tmpdepend
253 set -e; for i in $(SUBDIRS); do $(MAKE) -C $$i dep; done
254 rm -f tools/version.h
255 mv .tmpdepend .depend
256
257 ifdef CONFIGURATION
    ! 如果定义了 CONFIGURATION,则说明上面的
    ! 运行 make config 或者 make depend 来生成
    ! .config 和.depend 文件，只有这两个文件都有了
    ! 才可以编译出核心来
    ! 最后让你重新make
258 ..$(CONFIGURATION):
259     @echo
260     @echo "You have a bad or nonexistent" .$(CONFIGURATION) ": running 'make"
$$(CONFIGURATION)"""
261     @echo
262     $(MAKE) $$(CONFIGURATION)
263     @echo
264     @echo "Successful. Try re-making (ignore the error that follows)"
265     @echo
266     exit 1
267
268 dummy: ..$(CONFIGURATION)
269
270 else
271
272 dummy:
273
274 endif
275
276 #
277 # Leave these dummy entries for now to tell people that they are going away..
278 #
    ! 从这行开始直到文件解释，都不在被 1.0 核心支持，分别被
    ! make zlilo
    ! make zImage
    ! make zdisk
    ! 替代了
279 lilo:
280     @echo
281     @echo Uncompressed kernel images no longer supported. Use
282     @echo \"make zlilo\" instead.

```

```

283      @echo
284      @exit 1
285
286 Image:
287      @echo
288      @echo Uncompressed kernel images no longer supported. Use
289      @echo \"make zImage\" instead.
290      @echo
291      @exit 1
292
293 disk:
294      @echo
295      @echo Uncompressed kernel images no longer supported. Use
296      @echo \"make zdisk\" instead.
297      @echo
298      @exit 1

```

以上是整个工程的 Makefile 文件，但是看了后，我们还有不解的是生成 zBoot/zSystem 时 zBoot 的作用，所以我们还要看看 zBoot 目录下的 Makefile 才可用真正了解生成的影象文件的组成！（下面是从整个工程的 Makefile 中取出的内容）

```

179 zBoot/zSystem: zBoot/*.c zBoot/*.S tools/zSystem          ! 生成zBoot/zSystem
180      $(MAKE) -C zBoot

```

下面我们看看 zBoot/Makefile！

## **zBoot/Makefile**

```

1
2 HEAD = head.o
3 SYSTEM = ../tools/zSystem
4 #LD = gcc
5 #TEST = -DTEST_DRIVER
6
7 zOBJECTS = $(HEAD) inflate.o unzip.o misc.o
8
9 CFLAGS = -O2 -DSTDC_HEADERS $(TEST)
10
11 .c.s:
12      ! 把.c文件编译成.s文件
13      $(CC) $(CFLAGS) -S -o $*.s $<

```

```

13 .S.O:
    ! 把.s文件编译成.o文件
14     $(AS) -c -o $*.o $<
15 .C.O:
    ! 把.c文件编译成.o文件
16     $(CC) $(CFLAGS) -c -o $*.o $<
17
18 all: zSystem
    ! 生成zSystem文件
19
    ! zSystem = $(zOBJECTS) + piggy.o
    ! 即 zSystem = head.o + inflate.o + unzip.o + misc.o + piggy.o
    ! zSystem 代码段被从 4k 地址处开始编址连接
    ! 请注意 piggy.o 是真正的核心，并且已经以 gzip -9 压缩
    ! 请看 28 行
20 zSystem:    piggy.o $(zOBJECTS)
21         $(LD) $(LDFLAGS) -o zSystem -Ttext 1000 $(zOBJECTS) piggy.o
22
23 head.o: head.s
    ! 生成head.o
24
25 head.s: head.S ..../include/linux/tasks.h
    ! 生成head.s
26     $(CPP) -traditional head.S -o head.s
27
28 piggy.o:    $(SYSTEM) xtract piggyback ! 生成piggy.o!
    ! piggy.o 由 tools/zSystem 模块压缩而成
    ! tools/zSystem 在整个工程的 Makefile 被生
    ! 成，这里也会进行更新，请看 31 行
29     ./xtract $(SYSTEM) | gzip -9 | ./piggyback > piggy.o
30
31 $(SYSTEM):
32     $(MAKE) -C .. tools/zSystem

```

## 总结

看了以上的两个 Makefile 文件我们可知

**zImage = boot/bootsect + boot/setup + zBoot/zSystem**

**zBoot/zSystem = zBoot/head.o + zBoot/inflate.o + zBoot/unzip.o +**  
**zBoot/misc.o + zBoot/piggy.o**

**zBoot/piggy.o = tools/zSystem 的 gzip -9 压缩后的文件**

**tools/zSystem = boot/head.o + init/main.o + tools/version.o + linuxsubdirs**

**linuxsubdirs = kernel + drivers + mm + fs + net + ipc + ibcs + lib**

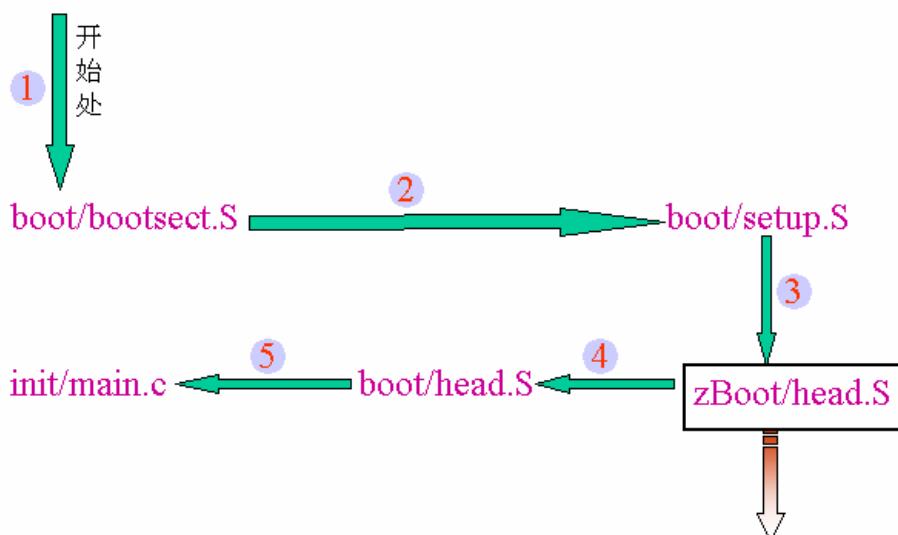
**所以 linuxsubdirs 最终就是各个子目录中文件编译后生成模块的叠加。**

**同时请注意：**

**tools/zSystem 代码段被从 1M 处开始编址！（为什么要被从 1M 编址呢？这个要看 zBoot/head.s 文件了，后面会分析这个文件）**

**zBoot/zSystem 代码段被从 4K 处开始编址！（为什么要被从 4K 编址呢？这个要看 boot/head.s 文件了，后面会分析这个文件）**

接着请按下面顺序阅读文件，这样就可以了解系统是怎么启动并且跑起来的了。



以下说明只针对1.0核心，现在高版本的核心可能有所变化

请注意`zBoot/head.S`调用了`zBoot`下的`inflate.c`、`unzip.c`、`misc.c`中定义的函数，这些函数用于解压缩被用`gzip -9`压缩的核心。并且把解压缩后的核心从物理地址1M开始处放置，这里不再注释这些文件，因为和核心关系不大！

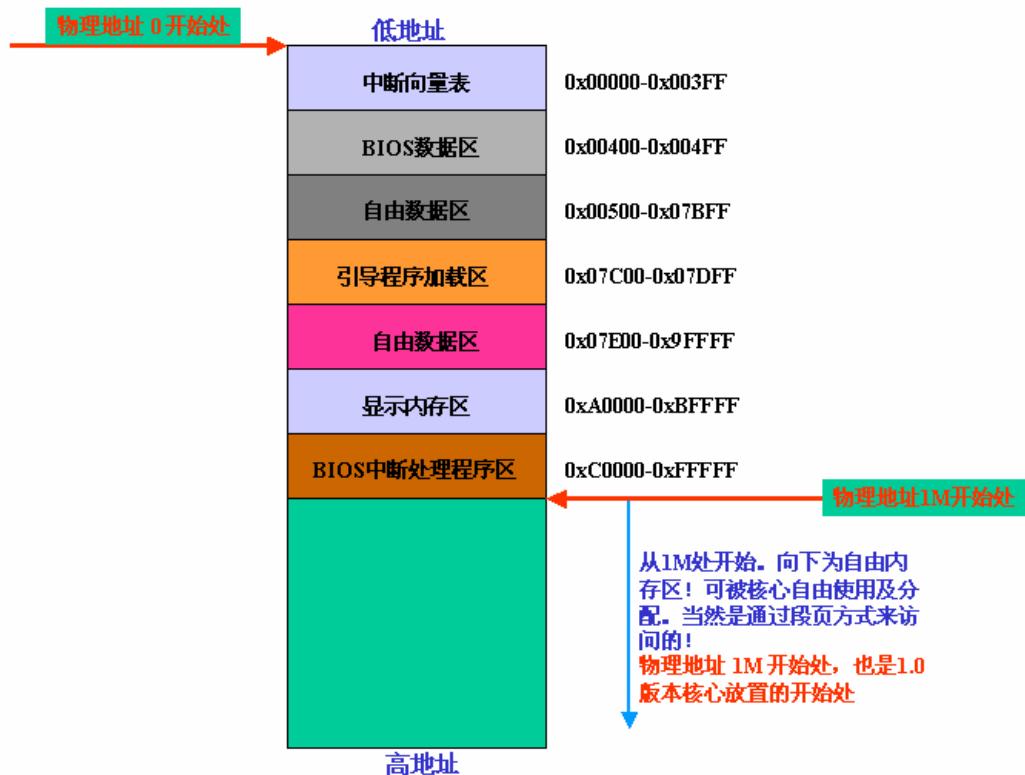
要解释一下的是为什么不从物理地址0处开始放置，这有两个原因

一：从物理地址0开始放置，会覆盖掉中断向量表，因为在有些笔记本的管理程序需要用到它！

二：从物理地址0开始放置，编译出的核心大小有限制，不能大于640k，如果大于640k就不能连续编址！不连续编址核心编写难度会加大

引爆点图 1

## 内存布局图



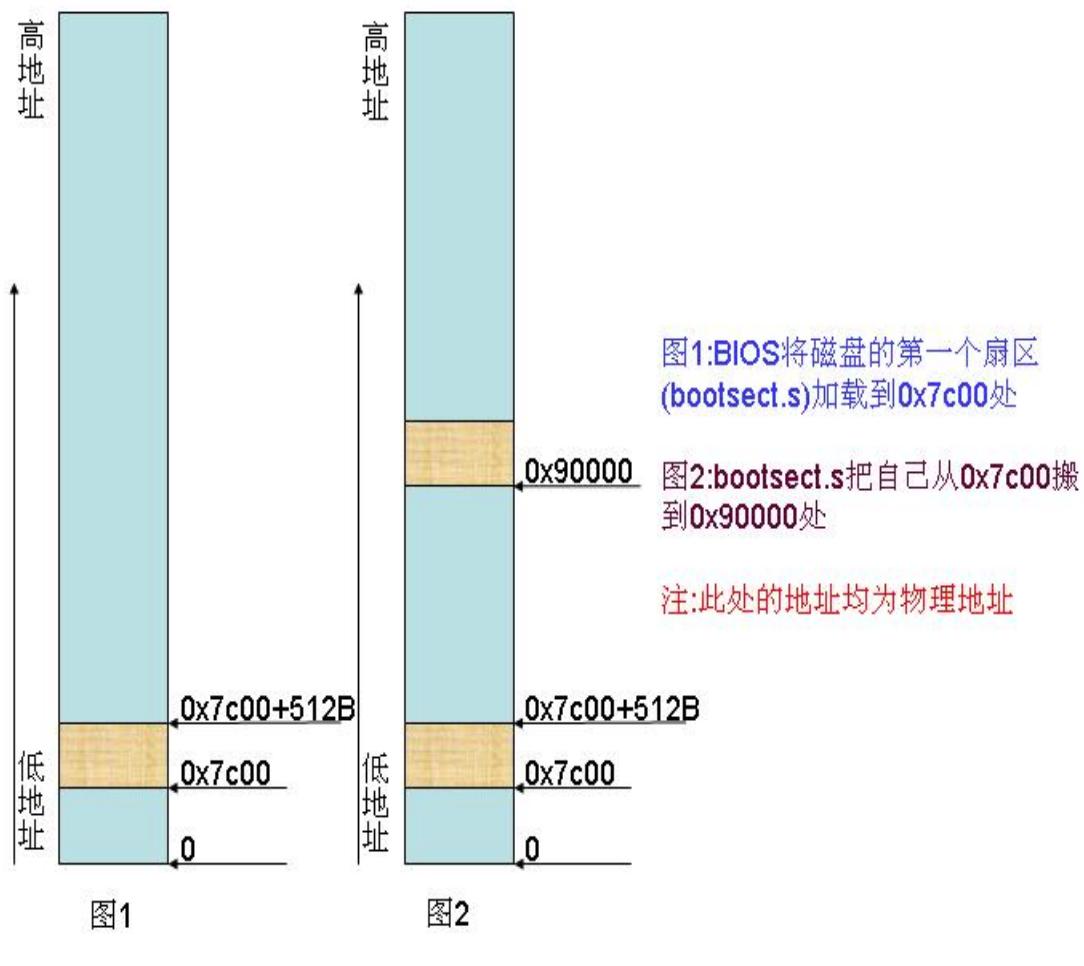
引爆点图 2

## B

### Boot/bootsect.S

#### 概述

bootsect.S 是核心中第一个执行的文件，该文件编译好后，放在磁盘的第一个扇区。当给计算机加电启动时，由 PC 的 BIOS 把它加载到以 0x7c00 物理内存地址开始的 512 个字节大小的地址空间中。在 BIOS 把这 512 个字节加载完后，便会跳到物理地址 0x07c0 处继续执行（意味着进入 bootsect.s 中执行）。Bootsect.s 首先把自己从 0x7c00 处搬到 0x90000 处。请参考图 B-1



在把自己搬运到 0x90000 处后，便会继续加载 `setup` 模块（4 个扇区大小，从 0x90000+512b 处开始放置）。待 `setup` 模块加载完成后，便会加载 `system` 模块（压缩 `linux` 核心，从 0x10000 开始放置，共 508k）。待 `system` 模块也加载成功后，就跳到 `setup` 模块中继续执行。到此为止，整个核心便已加载完成了。这时的内存布局请参看图 B-2

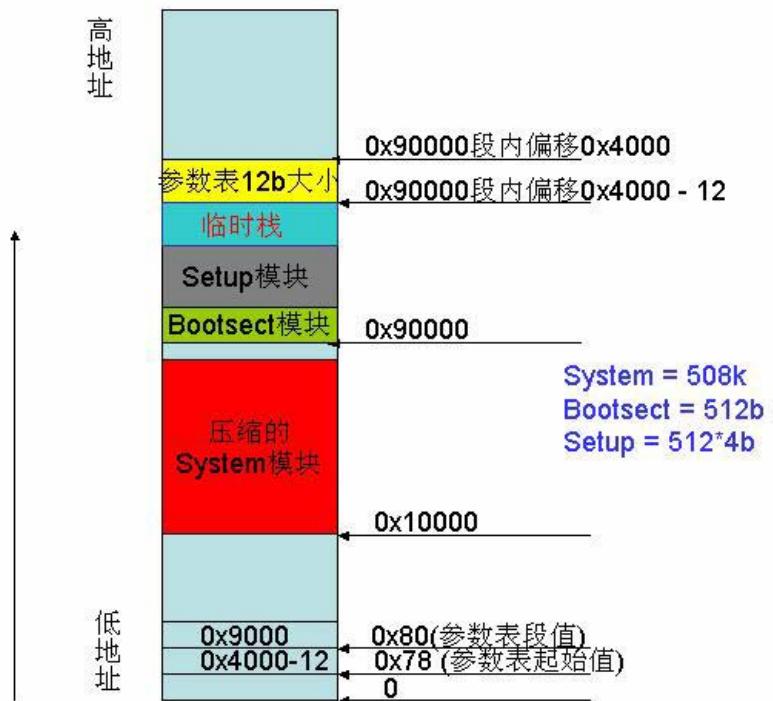


图 B-2

## 代码分析

1!  
2! SYS\_SIZE is the number of clicks (16 bytes) to be loaded.  
3! 0x7F00 is 0x7F000 bytes = 508kB, more than enough for current  
4! versions of linux which compress the kernel  
5!  
 ! SYS\_SIZE 是要加载的节数 (16Byte/节)。0x7F00 是 0x7F000=508kb,  
 ! 对于当前压缩版本的核心已经足够了  
6 #include <linux/config.h>  
7 SYSSIZE = DEF\_SYSSIZE  
8!  
9! bootsect.s Copyright (C) 1991, 1992 Linus Torvalds  
10! modified by Drew Eckhardt  
11! modified by Bruce Evans (bde)  
12!  
13! bootsect.s is loaded at 0x7c00 by the bios-startup routines, and moves  
14! itself out of the way to address 0x90000, and jumps there.  
 ! bootsect.s 被 BIOS 加载到 0x7c00 处，然后自己把自己移到 0x90000，  
 ! 接下来跳转到 0x90000 处。

15!

16 ! bde - should not jump blindly, there may be systems with only 512K low  
17 ! memory. Use int 0x12 to get the top of memory, etc.  
 ! bde - 不可以盲目的跳转，可能有的系统仅仅只有 512k 的少量内存，使  
 ! 用 int 0x12 来得到最大的内存，等等。

18 !  
19 ! It then loads 'setup' directly after itself (0x90200), and the system  
20 ! at 0x10000, using BIOS interrupts.  
 ! 接下来使用 BIOS 的中断，把 setup 加载在它自己 (bootsect) 的后面  
 ! (0x90200), system 加载到 0x10000 处

21 !  
22 ! NOTE! currently system is at most (8\*65536-4096) bytes long. This should  
23 ! be no problem, even in the future. I want to keep it simple. This 508 kB  
24 ! kernel size should be enough, especially as this doesn't contain the  
25 ! buffer cache as in minix (and especially now that the kernel is  
26 ! compressed :-)  
 ! 注意：当前 system 大小是 508k 个字节。即使是将来也没有问题。我想让  
 ! 它保持简单，这 508 个字节足够了，尤其是它没有象 minix 的高速缓冲区  
 ! (并且特别的是现在的核心已经是压缩的了)

27 !  
28 ! The loader has been made as simple as possible, and continuos  
29 ! read errors will result in a unbreakable loop. Reboot by hand. It  
30 ! loads pretty fast by getting whole tracks at a time whenever possible.  
 ! 这个加载程序已经做的够简单了，持续的读写将导致死循环，只能手工  
 ! 重新启动。只要可能，通过一次读取所有的扇区，加载过程可以很快。

31  
32 .text  
33  
34 SETUPSECS = 4 ! nr of setup-sectors  
 ! setup 模块的扇区数  
35 BOOTSEG = 0x07C0 ! original address of boot-sector  
 ! 启动扇区的最初地址  
36 INITSEG = DEF\_INITSEG ! we move boot here - out of the way  
 ! DEF\_INITSEG = 0x9000, 我们将把boot块 (bootsect) 移动到这里  
37 SETUPSEG = DEF\_SETUPSEG ! setup starts here  
 ! DEF\_SETUPSEG = 0x9020, setup模块的起始地址  
38 SYSSEG = DEF\_SYSSEG ! system loaded at 0x10000 (65536).  
 ! DEF\_SYSSEG = 0x1000, system模块加载的地址 (64k)  
39  
40 ! ROOT\_DEV & SWAP\_DEV are now written by "build".  
 ! ROOT\_DEV & SWAP\_DEV 将在build程序中被写  
41 ROOT\_DEV = 0  
42 SWAP\_DEV = 0

```

43 #ifndef SVGA_MODE
44 #define SVGA_MODE ASK_VGA
45 #endif
46 #ifndef RAMDISK
47 #define RAMDISK 0
48 #endif
49 #ifndef CONFIG_ROOT_RDONLY
50 #define CONFIG_ROOT_RDONLY 0
51 #endif
52
    ! 以上为条件编译
53 ! ld86 requires an entry symbol. This may as well be the usual one.
    ! ld86 需要一个进入点。main 就是。
54 .globl _main
55 _main:
56 #if 0 /* hook for debugger, harmless unless BIOS is fussy (old HP) */
57     int    3
58#endif
    ! 条件编译, 如果定义了 0, 则用于 debugger。大部分情况是没有关系的除
    ! 老的 HP 机器。
59     mov    ax,#BOOTSEG    ! ax = 0x07C0
60     mov    ds,ax          ! ds = 0x07C0
61     mov    ax,#INITSEG    ! ax = 0x9000
62     mov    es,ax          ! es = 0x9000
63     mov    cx,#256        ! cx = 256
64     sub    si,si          ! si = 0
65     sub    di,di          ! di = 0
66     cld                ! DF = 0(索引由小变大)
67     rep
68     movsw
69     jmpi   go,INITSEG    ! 重复移动 256Word
70
71 go:    mov    ax,cs          ! 此时 cs = 0x9000, 因为上一句
            ! 的导致
72     mov    dx,#0x4000-12   ! 0x4000 is arbitrary value >= length of
73                           ! bootsect + length of setup + room for stack
74                           ! 12 is disk parm size
    ! 0x4000 是一个武断的值, 它大于等于 bootsect + setup + 堆栈的空间
    ! 12 是磁盘的参数尺寸
75
76 ! bde - changed 0xff00 to 0x4000 to use debugger at 0x6400 up (bde). We
77 ! wouldn't have to worry about this if we checked the top of memory. Also
78 ! my BIOS can be configured to put the wini drive tables in high memory

```

79 ! instead of in the vector table. The old stack might have clobbered the  
80 ! drive table.

! bde - 改变 0xff00 到 0x4000，是为了在 0x6400 以上 debugger(bde)。  
 ! 我们不得不担心这个，假如我们检查最大内存值。并且我的 BIOS 能够  
 ! 被配置成用 wini 驱动表来代替向量表。这样的话老的堆栈可能会破坏  
 ! 这个驱动表。

```

81
82     mov      ds,ax          ! ds = 0x9000
83     mov      es,ax          ! es = 0x9000
84     mov      ss,ax          ! put stack at INITSEG:0x4000-12.
85     mov      sp,dx
    ! ss = 0x9000, sp = dx。这样的话最初 ss:sp 值为 0x9000:0x4000-12
86 /*
87 *      Many BIOS's default disk parameter tables will not
88 *      recognize multi-sector reads beyond the maximum sector number
89 *      specified in the default diskette parameter tables - this may
90 *      mean 7 sectors in some cases.
91 *
92 *      Since single sector reads are slow and out of the question,
93 *      we must take care of this by creating new parameter tables
94 *      (for the first disk) in RAM. We will set the maximum sector
95 *      count to 18 - the most we will encounter on an HD 1.44.
96 *
97 *      High doesn't hurt. Low does.
98 *
99 *      Segments are as follows: ds=es=ss=cs - INITSEG,
100 *                  fs = 0, gs = parameter table segment
101 */
102
    !许多的 BIOS 磁盘参数表不承认通过指定的磁盘参数表的最大扇区数
    !来读。- 这意味着在一些案例中是 7 个扇区。
    !由于有通过单个扇区读比较慢的问题，所以我们必须在内存中创建
    !一个新的参数表（为第一块硬盘）来小心处理这个。我们将设定最大的
    !扇区数为 18 - 用于 HD - 1.44
    !High doesn't hurt. Low does. (不知道该如何翻译)
    !ds = es = ss = cs = 0x9000, fs = 0, gs = 参数表段
103    push    #0          ! 0 入栈
104    pop     fs          ! 出栈后, fs = 0
105    mov     bx,#0x78      ! fs:bx is parameter table address
106    seg    fs          ! fs:bx = 0:0x78 是参数表地址
107    lgs     si,(bx)      ! gs:si is source
                                ! si = fs:[0x78]=0:[0x78]
                                ! gs = 0:[0x80]
```

108

```

109    mov     di,dx          ! es:di is destination
110    mov     cx,#6           ! copy 12 bytes
                  ! 拷贝 12 个字节
111    cld
112
113    rep
114    seg gs             ! es:di = 0x9000:0x4000-12
115    movsw

```

! 从gs:si拷贝 12 个字节到es:di处  
! 拷贝关系请看图 B-3

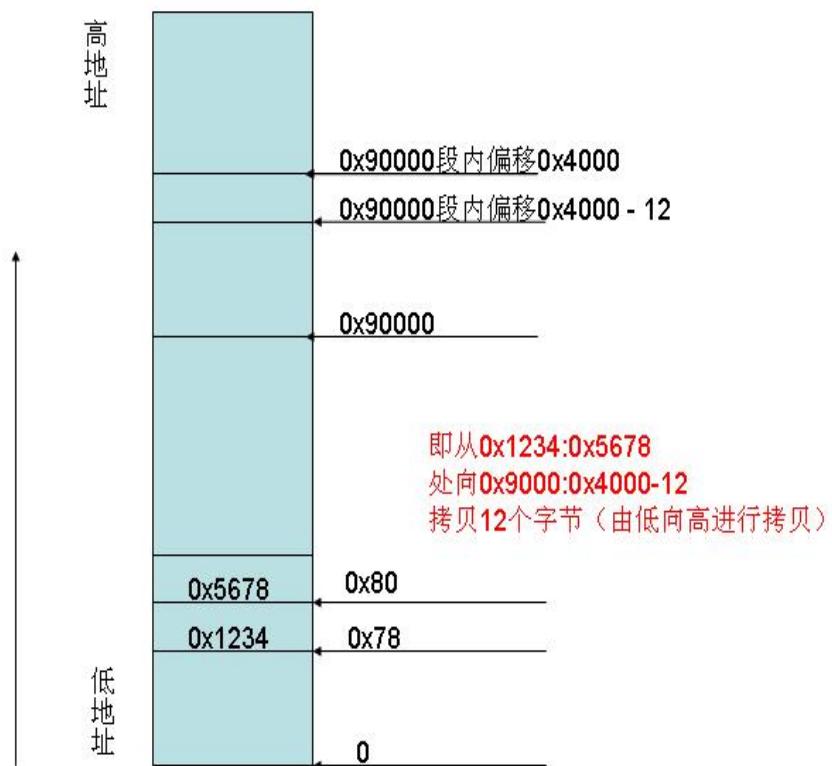


图 B-3

```

116
117    mov     di,dx          ! di = dx = 0x4000 - 12
118    movb   4(di),*18        ! patch sector count
                  ! ds:[di + 4] * 18 再重新写入
119
120    seg fs             ! fs = 0
121    mov     (bx),di        ! fs:[bx] = di (0:[0x78] = di)
122    seg fs             ! fs = 0
123    mov     2(bx),es        ! fs:[bx+2] = es (0:[0x80] = 0x9000)
                  ! 即把新的参数表地址放到图 B-1
                  ! 0x78 和 0x80 中去，这时候这两个

```

```

! 地址中值分别为: 0x4000 - 12
! 和 0x9000

124
125    mov     ax,cs          ! ax = cs = 0x9000
126    mov     fs,ax          ! fs = ax = 0x9000
127    mov     gs,ax          ! gs = ax = 0x9000
128
129    xor     ah,ah          ! reset FDC
130    xor     dl,dl          ! ah = dl = 0
131    int     0x13          ! 复位磁盘
132
133 ! load the setup-sectors directly after the bootblock.
134 ! Note that 'es' is already set up.
      ! 把 setup 模块直接加载在 bootsect 后, 注意这时的 es 已经被修改了

135
136 load_setup:           ! 加载setup模块子程序
137    xor     dx,dx          ! drive 0, head 0
138    mov     cx,#0x0002      ! sector 2, track 0
139    mov     bx,#0x0200      ! address = 512, in INITSEG
140    mov     ax,#0x0200+SETUPSECS ! service 2, nr of sectors
141                  ! (assume all on head 0, track 0)
142    int     0x13          ! read it
143    jnc     ok_load_setup ! ok - continue
      ! 加载 setup 模块成功, 则跳转到
      ! ok_load_setup 继续执行

144
145    push    ax          ! dump error code
      ! 错误码入栈
146    call    print_nl      ! 打印回车换行
147    mov     bp, sp          ! bp = sp
148    call    print_hex      ! 打印错误值的 16 进制编码
149    pop     ax          !! 弹出错误码
150
151    xor     dl, dl          ! reset FDC
152    xor     ah, ah
153    int     0x13
154    jmp     load_setup     ! 重新执行加载setup操作
155
156 ok_load_setup:        ! ok_load_setup子程序
157
158 ! Get disk drive parameters, specifically nr of sectors/track
159 ! 得到磁盘的参数, 特别是每道扇区数和磁道数
160 #if 0
161

```

---

162 ! bde - the Phoenix BIOS manual says function 0x08 only works for fixed  
163 ! disks. It doesn't work for one of my BIOS's (1987 Award). It was  
164 ! fatal not to check the error code.  
165  
 ! bde - Phoenix BIOS 手册说 8 号功能仅仅只为固定的磁盘工作，对于我  
 ! BIOS 不工作（1987 Award），它也没有检查错误代码  
166 xor dl,dl ! dl = 0  
167 mov ah,#0x08 ! AH=8 is get drive parameters  
168 int 0x13 ! 8 号功能调用，为了得到磁盘参数  
169 xor ch,ch ! ch = 0  
170 #else  
171  
172 ! It seems that there is no BIOS call to get the number of sectors. Guess  
173 ! 18 sectors if sector 18 can be read, 15 if sector 15 can be read.  
174 ! Otherwise guess 9.  
 ! 好象没有 BIOS 调用可以达到扇区数，只能猜测如果可以读扇区 18，则  
 ! 是 18，如果可以读扇区 15，则是 15，其他是 9  
175  
176 xor dx,dx ! drive 0, head 0  
177 mov cx,#0x0012 ! sector 18, track 0  
178 mov bx,#0x0200+SETUPSECS\*0x200 ! address after setup (es = cs)  
 ! bx = 512 + 512 \* 4  
179 mov ax,#0x0201 ! service 2, 1 sector  
180 int 0x13  
181 jnc got\_sectors  
 ! 先读第 18 个扇区，若是则跳到  
 ! got\_sectors  
182 mov cl,#0x0f ! sector 15  
183 mov ax,#0x0201 ! service 2, 1 sector  
184 int 0x13  
185 jnc got\_sectors  
 ! 不是 18，则再读第 15 个扇区  
 !，若是则跳到 got\_sectors  
186 mov cl,#0x09  
 ! 第三种情况表示 9 个扇区  
187  
188 #endif  
189  
190 got\_sectors:  
 ! 保存取到的扇区数  
191 seg cs  
192 mov sectors,cx ! 存入sectors中  
193 mov ax,#INITSEG ! ax = 0x9000  
194 mov es,ax ! es = 0x9000  
195  
196 ! Print some inane message

```

197
198     mov     ah,#0x03          ! read cursor pos
199     xor     bh,bh
200     int     0x10          ! 读光标位置
201
202     mov     cx,#9           ! cx = 9 (共 9 个字符)
203     mov     bx,#0x0007      ! page 0, attribute 7 (normal)
204     mov     bp,#msg1        ! 定义在L432
205     mov     ax,#0x1301      ! write string, move cursor
206     int     0x10          ! 写字符串并且移动光标
207
208 ! ok, we've written the message, now
209 ! we want to load the system (at 0x10000)
    ! 好，我们已经写了这个信息，接下来我们加载 system 模块（0x10000）
210
211     mov     ax,#SYSSEG       ! ax = 0x1000
212     mov     es,ax           ! segment of 0x010000
213     call    read_it         ! 调用 read_it子程序
214     call    kill_motor       ! 调用 kill_motor子程序
215     call    print_nl        ! 打印回车换行
216
217 ! After that we check which root-device to use. If the device is
218 ! defined (!= 0), nothing is done and the given device is used.
219 ! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
220 ! on the number of sectors that the BIOS reports currently.
    ! 此后，我们检查使用那个根设备，如果这个设备是定义的（!=0），
    ! 就直接使用这个设备。否则，就需要根据 BIOS 报出来的扇区数来确定
    ! 是使用/dev/PS0 还是 /dev/at0
221
222     seg cs                ! cs = 0x9000
223     mov     ax,root_dev      ! ax = root_dev
224     or      ax,ax           ! ax = 0 表示没有定义
225     jne    root_defined     ! 跳转到root_defined
226     seg cs                ! cs = 0x9000
227     mov     bx,sectors       ! bx = sectors(扇区数)
228     mov     ax,#0x0208      ! /dev/ps0 - 1.2Mb
229     cmp     bx,#15          ! 检查扇区是否等于 15
230     je     root_defined
231     mov     ax,#0x021c      ! /dev/PS0 - 1.44Mb
232     cmp     bx,#18          ! 检查扇区是否等于 18
233     je     root_defined
234     mov     ax,#0x0200      ! /dev/fd0 - autodetect
235 root_defined:
236     seg cs

```

```

237      mov      root_dev,ax          ! 保存真正的根设备
238
239 ! after that (everyting loaded), we jump to
240 ! the setup-routine loaded directly after
241 ! the bootblock:
    ! 最后（所有模块加载完成），我们直接跳转到 setup 模块中去
242
243      jmp     0,SETUPSEG
244
245 ! This routine loads the system at address 0x10000, making sure
246 ! no 64kB boundaries are crossed. We try to load it as fast as
247 ! possible, loading whole tracks whenever we can.
248 !
249 ! in:   es - starting address segment (normally 0x1000)
250 !
    ! 该程序把 system 模块加载到 0x1000 初，确保是以 64k 为边界的。我们
    ! 尽可能快速的加载它，只要有可能我们会加载整个磁道。
    ! es = 0x1000
251 sread: .word 1+SETUPSECS      ! sectors read of current track
                                ! sread 是已经加载的扇区数
252 head:   .word 0              ! current head
253 track:  .word 0              ! current track
254
255 read_it:                      ! read_it子程序
256      mov ax,es                ! ax = es = 0x1000
257      test ax,#0x0fff          ! 测试es是否在 64k边界
258 die:    jne die              ! es must be at 64kB boundary
259      xor bx,bx               ! bx is starting address within segment
260 rp_read:                     ! rp_read子程序
261      mov ax,es                ! ax = es = 0x1000
262      sub ax,#SYSSEG          ! ax = ax - SYSSEG
263      cmp ax,syssize          ! have we loaded all yet?
                                ! 比较是否已经全部加载完成
264      jbe ok1_read            ! 没有继续加载
265      ret
266 ok1_read:                    ! ok_read子程序
267      seg cs                  ! cs = 0x9000 为了取得sectors值
268      mov ax,sectors           ! ax = 扇区数/磁道
269      sub ax,sread             ! ax = ax - sread (还乘未读扇区数/磁道)
270      mov cx,ax                ! cx = ax (剩余未读扇区数/磁道)
271      shl cx,#9               ! cx = cx * 512
272      add cx,bx               ! cx = cx + bx(bx是已读数据存放地址结尾)
273      jnc ok2_read

```

! 若没有超过 64k，则跳转到 ok\_read 处读)

```

274      je ok2_read
275      xor ax,ax          ! 若加上磁道此次未读扇区大于 64k，则计算
                                ! 此次能读的字节数，再转换成要读扇区数
276      sub ax,bx
277      shr ax,#9          ! ax = ax/512
278 ok2_read:
279      call read_track     ! read_track子程序
280      mov cx,ax          ! cx = 该次操作已经读取扇区数
281      add ax,sread        ! 当前磁道已经读取的扇区数
282      seg cs
283      cmp ax,sectors
284      jne ok3_read        ! 若还有未读，则跳转到ok3_read处
285      mov ax,#1            ! ax = 1
286      sub ax,head
287      jne ok4_read        ! 如果是 0 磁头，再去度 1 磁头
288      inc track           ! 磁道数减一（转到下一磁道）
289 ok4_read:
290      mov head,ax          ! 磁头 1 放入head
291      xor ax,ax            ! ax = 0
292 ok3_read:
293      mov sread,ax          ! 保存当前磁道已读扇区数
294      shl cx,#9            ! cx = cx * 512
295      add bx,cx            ! 加上bx，段内最大值
296      jnc rp_read          ! 若小于 64k边界，则跳转到rp_read
297      mov ax,es
298      add ah,#0x10
299      mov es,ax
300      xor bx,bx
301      jmp rp_read          ! 继续读
302
303 read_track:
304      pusha                ! 所有寄存器入栈
305      pusha                ! 所有寄存器入栈
306      mov      ax, #0xe2e    ! loading... message 2e = .
307      mov      bx, #7
308      int      0x10          ! 显示 2e
309      popa                 所有寄存器出栈
310
311      mov      dx,track      ! dx = 当前磁道号
312      mov      cx,sread      ! cx = 当前磁道已读扇区
313      inc      cx            ! cx = cx + 1 开始读扇区
314      mov      ch,dl          ! 当前磁道号
315      mov      dx,head        ! 当前磁头号
316      mov      dh,dl          ! dh = dl

```

```

317      and    dx,#0x0100
318      mov    ah,#2
319
320      push   dx          ! save for error dump
321      push   cx
322      push   bx
323      push   ax          ! 发生错误时, DUMP的数据
324
325      int    0x13
326      jc    bad_rt        ! 错误调用bad_rt
327      add    sp,#8
328      popa
329      ret
330
331 bad_rt: push   ax          ! save error code
332      call   print_all     ! ah = error, al = read
333                      ! 调用print_all dump错误值
334
335      xor ah,ah
336      xor dl,dl
337      int 0x13           ! 复位磁盘
338
339
340      add    sp,#10         ! 退栈
341      popa
342      jmp read_track      ! 继续读
343
344 /*
345 *      print_all is for debugging purposes.
346 *      It will print out all of the registers.  The assumption is that this is
347 *      called from a routine, with a stack frame like
348 *      dx
349 *      cx
350 *      bx
351 *      ax
352 *      error
353 *      ret <- sp
354 *
355 */
356
! print_all 用于 debugging。它将会打印所有的寄存器，调用该函数时
! 栈分布如下
! dx
! cx

```

```

! bx
! ax
! error
! ret <- sp

357 print_all:
358     mov    cx, #5      ! error code + 4 registers
                    ! cx = 5( 一个错误码 + 4 个寄存器)
359     mov    bp, sp      ! 保存栈指针
360

361 print_loop:
362     push   cx          ! save count left
363     call   print_nl     ! nl for readability
364
365     cmp    cl, 5
366     jae   no_reg       ! see if register name is needed
                    ! 当 cl=5 时，表示要打印的值不需要寄存器
                    ! 来描述，直接跳转到 no_reg 执行
367
368     mov    ax, #0xe05 + 'A - 1
369     sub    al, cl
370     int    0x10
371
372     mov    al, #'X
373     int    0x10
374
375     mov    al, #':'
376     int    0x10
                    ! cl<5 时，顺序输出型如：
AX:1234
BX:5678
CX:E2A5
DX:4BC6

377
378 no_reg:
379     add    bp, #2      ! next register
380     call   print_hex   ! print it
381     pop    cx          ! 上面保留的cx出栈
382     loop   print_loop   ! cx = cx - 1>0 ? print_loop : ret
383     ret
384
385 print_nl:           ! print_nl子程序，打印回车换行
386     mov    ax, #0xe0d   ! CR
387     int    0x10
388     mov    al, #0xa   ! LF

```

```

389      int     0x10
390      ret
391
392 /*
393 *      print_hex is for debugging purposes, and prints the word
394 *      pointed to by ss:bp in hexadecmial.
395 */
396
397 print_hex:                      ! 16 进制打印错误码
398      mov     cx, #4          ! 4 hex digits
399      mov     dx, (bp)        ! load word into dx
399                                ! dx = 要打印的栈上的值

400 print_digit:
401      rol     dx, #4          ! rotate so that lowest 4 bits are used
401                                ! 循环左移四位，从最高的四位开始打印
402      mov     ah, #0xe
403      mov     al, dl          ! mask off so we have only next nibble
404      and     al, #0xf
405      add     al, #'0         ! convert to 0-based digit
406      cmp     al, #'9         ! check for overflow
406                                ! 查看是否超过 ‘9’
407      jbe     good_digit
408      add     al, #'A - '0 - 10
409
410 good_digit:
411      int     0x10          ! 打印 16 进制数
412      loop    print_digit   ! 打印下一个（共分打印 4 次）
413      ret
414
415
416 /*
417 * This procedure turns off the floppy drive motor, so
418 * that we enter the kernel in a known state, and
419 * don't have to worry about it later.
420 */
420                                ! 该程序用于关闭驱动器，以至于我们非常清楚进入内核后的状态，
420                                ! 以后就不用担心它了。

421 kill_motor:
422      push   dx
423      mov    dx,#0x3f2        ! 软驱控制卡的驱动端口，只写！
424      xor    al,al           ! al = 0
425      outb

```

```

426      pop dx
427      ret
428
429 sectors:
430      .word 0          ! 磁盘扇区数/磁道
431
432 msg1:
433      .byte 13,10
434      .ascii "Loading"    ! 加载system前的提示
435
436 .org 498
437 root_flags:
438      .word CONFIG_ROOT_RDONLY
439 syssize:
440      .word SYSSIZE
441 swap_dev:
442      .word SWAP_DEV
443 ram_size:
444      .word RAMDISK
445 vid_mode:
446      .word SVGA_MODE
447 root_dev:
448      .word ROOT_DEV
449 boot_flag:
450      .word 0xAA55      ! 引导块签名

```

## Boot/setup.S

### 概述

用 BIOS 的调用来得到系统硬件的配置值，然后把这些得到的值存放在一个安全的地方。开始于 0x90000、止于 0x90000+0x200。具体的这些值，请看下面的表格内容。接下来首先判断是否是否有 VGA 或者 EGA 卡，如果没有的话，直接调用采用系统默认的显示方式。否则，取出定义在 bootsect.s 0x01FA 处的值，来确认是普通的 VGA 显卡还是扩展的 VGA 显卡，或者是让用户自己选择。如果都不是则直接采用 svga 显示模式！（请注意：通过 BIOS 并不是顺序取值后，顺序存放的，具体请看代码的注释）

然后为切换到保护模式做准备！把 System 模块（该模块在 bootsect.s 中被放在从 0x10000 开始处）移动到从 0x1000 开始处，共 508k 个字节。移动完后加载 idt 表和 gdt 表。接下来通过选通 A20 地址线来进入保护模式。真正进入后，初始化 8259A 中断芯片。

段内偏移	值
0x0	当前光标位置
0x2	扩展内存的大小
0x4	当前显示页
0x6	高字节=字符列数 低字节=显示模式
0x8	未知
0x9	安装的显示内存大小
0x10	显示状态(彩色/单色)
0x12	显示卡特性
0x14	显卡类型(EGA/VGA)
0x80	第一块硬盘的参数表
0x90	第二块硬盘的参数表
0x1ff	0 = 没有安装针设备, 0xaa = 安装了

图 Boot/setup.S-1

## 代码分析

```

1 !
2 !      setup.S      Copyright (C) 1991, 1992 Linus Torvalds
3 !
4 ! setup.s is responsible for getting the system data from the BIOS,
5 ! and putting them into the appropriate places in system memory.
6 ! both setup.s and system has been loaded by the bootblock.

    ! setup.s 用 BIOS 获得系统的数据，并且把他们放在正确的系统内存位置。
    ! setup.s 和 system 都由引导块来加载。

7 !
8 ! This code asks the bios for memory/disk/other parameters, and
9 ! puts them in a "safe" place: 0x90000-0x901FF, ie where the
10 ! boot-block used to be. It is then up to the protected mode
11 ! system to read them from there before the area is overwritten
12 ! for buffer-blocks.

13 !
    ! 这里的代码通过 BIOS 调用来获得内存/磁盘/其他的参数，并且把他们放
    ! 一个安全的地方“0x90000 - 0x901FF”，其他的模块会使用它，它然后
    ! 进入保护模式！系统在高速缓冲覆盖它们之前把它们读出来。

14 ! Move PS/2 aux init code to psaux.c
15 ! (troyer@saifr00.cfsat.Honeywell.COM) 03Oct92
    ! 驱动 PS/2 计算机设备的代码在 psaux.c 中

16 !

```

```

17 ! some changes and additional features by Christoph Niemann, March
1993
18 ! (niemann@rubdv15.ETDV.Ruhr-Uni-Bochum.De)
19 !
20 ! Christoph Niemann 加了一些其他的特征
21 ! NOTE! These had better be the same as in bootsect.s!
22 ! 注意: 它们最好保持和 bootsect.s 一样
23 #include <linux/config.h>
24 #include <linux/segment.h>
25
26 #ifndef SVGA_MODE
27 #define SVGA_MODE ASK_VGA      ! 如果未定义 SVGA_MODE
28 #endif                      ! 则定义它为 ASK_VGA
29
30 INITSEG = DEF_INITSEG ! we move boot here - out of the way
31 SYSSEG  = DEF_SYSSEG   ! system loaded at 0x10000 (65536).
32 SETUPSEG = DEF_SETUPSEG ! this is the current segment
33
34 .globl begtext, begdata, begbss, endtext, enddata, endbss
35 .text
36 begtext:
37 begdata:
38 .bss
39 begbss:
40 .text
41
42 entry start
43 start:           ! start 为入口点
44
45 ! ok, the read went well so we get current cursor position and save
it for
46 ! posterity.
47
48     mov    ax,#INITSEG    ! this is done in bootsect already,
but...
49     mov    ds,ax      ! ds = ax = 0x9000
50
51 ! Get memory size (extended mem, kB)
52
53     mov    ah,#0x88
54     int    0x15

```

```

55      mov     [2],ax      ! 取的扩展内存的大小，并把它放在
           ! 0x9000:2 处
56
57 ! set the keyboard repeat rate to the max
58
59      ! ah = 0x03 - 设置键入速度和延时
60      ! al = 0x05 - 设置重复速率和延时
61      mov     ax,#0x0305
62      xor     bx,bx      ! clear bx
63      int     0x16      设置键盘速率
64
65      ! check for EGA/VGA and some config parameters
66      ! 检查 EGA/VGA 和其他的一些配置参数
67
68      mov     ah,#0x12      ! ah = 0x12
69      mov     bl,#0x10      ! bl = 0x10
70      int     0x10
71      mov     [8],ax      ! 0x9000:8 = ???
72      mov     [10],bx      ! 0x9000:9 = 安装的显示内存
           ! 0x9000:10 = 显示状态(彩色/单色)
73      cmp     bl,#0x10      ! 若bl=0x10，则跳转到novga
74      je      novga
75      mov     ax,#0x1a00      ! Added check for EGA/VGA
discrimination
           ! 附加的检查为了区别 EGA/VGA
76      int     0x10
           ! 返回值：如果函数支持的话 al = 0x1a
           ! bl = 激活显示码 bh = 预备的显示码
77      mov     bx,ax      ! bx = ax
78      mov     ax,#0x5019      ! ax = 0x5019
79      cmp     bl,#0x1a      ! 1a means VGA, anything else EGA or
lower
           ! 根据上面代码 75 行执行后返回结果
           ! 果，来判断是否是 VGA
80      jne     novga      ! 如果bl!=0x1a，说明不是VGA
           ! 则跳转到 novga
81      call    chsvga      ! 否则调用chsvga来进一步判断
82      novga:  mov     [14],ax      ! 显卡类型放到 0x9000:14 (VGA/EGA)
83
84      mov     ah,#0x03      ! read cursor pos
85      xor     bh,bh      ! clear bh
86      int     0x10      ! save it in known place, con_init

```

```

fetches

85      mov     [0],dx          ! it from 0x90000.
          !0x9000:0 放当前光标位置

86

87 ! Get video-card data:
    ! 获得显卡数据

88

89      mov     ah,#0x0f
90      int     0x10      ! 取当前显示方式
91      mov     [4],bx      ! bh = display page
92      mov     [6],ax      ! al = video mode, ah = window width
93

94 ! Get hd0 data
    ! 得到第一块硬盘的数据

95

96      xor     ax,ax      ! clear ax
97      mov     ds,ax      ! ds=ax=0
98      lds     si,[4*0x41] ! 取中断向量 0x41 的值作
          ! 第一块硬盘参数表的首地址
99      mov     ax,#INITSEG
100     mov    es,ax      ! es = 0x9000
101     mov    di,#0x0080
102     mov    cx,#0x10
103     cld
104     rep
105     movsb      ! 从ds:si 传送 0x10 个字节到
          ! es:di 即 0x9000:0x80

106

107 ! Get hd1 data
    ! 得到第二块硬盘的数据

108

109     xor     ax,ax      ! clear ax
110     mov     ds,ax      ! ds=ax=0
111     lds     si,[4*0x46] ! 取中断向量 0x46 的值
          ! 作为第二块硬盘参数表首地址
112     mov     ax,#INITSEG
113     mov    es,ax
114     mov    di,#0x0090
115     mov    cx,#0x10
116     cld
117     rep
118     movsb      ! 从 ds:si 传送 0x10 个字节到
          ! es:di 即 0x9000:0x90

```

```

119
120 ! Check that there IS a hd1 :-
    ! 检查是否是真正的有第 2 块硬盘，用 BIOS 0x13 号调用得到硬盘类型
    ! dl = 0x81 = 驱动器号（位 7=1，表示硬盘，0x81 表示第二块硬盘）
    ! 返回值： ah = 类型码（00 表示没有这个盘， CF = 1。 01 表示软驱
    没有 change-line 支持。 02 表示软驱，有 change-line 支持。 03 是硬盘）

121
122     mov    ax,#0x01500
123     mov    dl,#0x81
124     int    0x13
125     jc     no_disk1      ! 没有第 2 块硬盘
126     cmp    ah,#3
127     je     is_disk1      ! 有第 2 块硬盘
128 no_disk1:
129     mov    ax,#INITSEG
130     mov    es,ax
131     mov    di,#0x0090
132     mov    cx,#0x10
133     xor    ax,ax
134     cld
135     rep
136     stosb      ! 没有第 2 块硬盘时，
                    ! 将 0x9000:0x90 处开始的 0x10 个
                    ! 字清零
137 is_disk1:
                    ! 有第 2 块硬盘，不清零
138
139 ! check for PS/2 pointing device
    ! 检查 PS/2 针设备
140
141     mov    ax,#INITSEG
142     mov    ds,ax      ! ds = 0x9000
143     mov    [0x1ff],#0      ! default is no pointing device
144     int    0x11      ! int 0x11: equipment determination
145     test   al,#0x04      ! check if pointing device installed
                    ! 检测是否安装了针设备
146     jz     no_psmouse    ! 没有安装鼠标
147     mov    [0x1ff],#0xaa    ! device present
                    ! 设备类型放在 0x9000:0x1FF 处
                    ! int 0x11 设备检验
                    ! AX=返回值 bit0=1, 配有磁盘
                    ! bit1=1, 80287 协处理器 bit4,5=01, 40×25BW(彩色板)
                    =10, 80×25BW(彩色板)
                    =11, 80×25BW(黑白板)

```

```

    ! bit6,7=罗盘驱动器
    ! bit9,10,11=RS-232 板号
    ! bit12=游戏适配器

    ! bit13=串行打印机
    ! bit14,15=打印机号
    !!!!!!!我手里的 BIOS 资料，看不到 bit2 的描述，猜测应该是鼠标有无！
    !!!!!!! bit2=0 没有鼠标
        bit2=1 有鼠标

148 no_psmouse:
149 ! now we want to move to protected mode ...
    ! 现在我们准备进入保护模式

150
151     cli           ! no interrupts allowed !
        ! 关中断
152     mov al,#0x80      ! disable NMI for the bootup sequence

153     out #0x70,al
        ! 禁止非屏蔽中断

154
155 ! first we move the system to its rightful place
    ! 首先我们把 system 模块移动到正确的位置

156
157     mov ax,#0x100      ! start of destination segment
158     mov bx,#0x1000     ! start of source segment
159     cld               ! 'direction'=0, movs moves forward
        ! 目的地址 0x100,源地址 0x10000
        ! DF = 1

160 do_move:
161     mov es,ax          ! destination segment
        ! es = 0x100 (目标段)
162     add ax,#0x100      ! ax = ax+0x100=0x200(第一次时的值)
163     cmp ax,#0x9000    ! 比较是否已经等于 0x9000
164     jz end_move ! 是则结束移动
165     mov ds,bx          ! source segment
        ! ds=bx=0x1000(源段)
166     add bx,#0x100      ! bx=bx+0x100
167     sub di,di ! di=0
168     sub si,si ! si=0
169     mov cx,#0x800      ! 循环记数值
170     rep
171     movsw      ! 移动 0x800WORD/次 (即 4k)
        ! 从 ds;si 到 es;di
172     jmp do_move

```

! 当移动完最后一个 4k (从 0x90000 - 4k 地址开始) 后,  
 ! ax=0x9000, 这表明 system 模块已经全部移动完成  
 ! (方向从 0x1000: 0 向 0x100:0) 一共移动了 576k 个字节,  
 ! 并不是内核就是这么大, 内核最大 508k, 所以有些垃圾。  
 ! 这个时候内存如下图 B-4

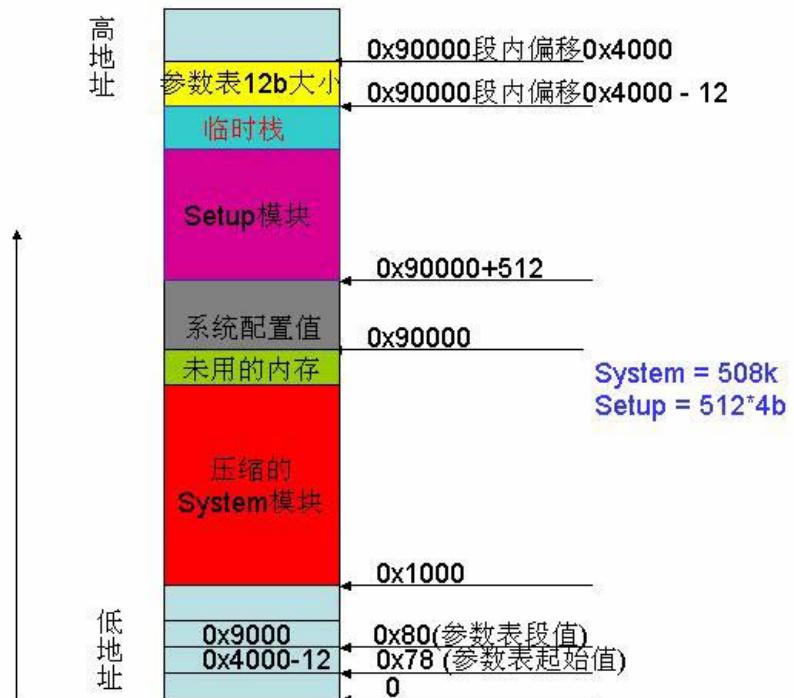


图 B-4

```

173
174 ! then we load the segment descriptors
    ! 接下来我们加载段描述符
175
176 end_move:
177     mov     ax,#SETUPSEG      ! right, forgot this at first.
didn't work :-)
178     mov     ds,ax ! ds = ax = 0x9020
179     lidt    idt_48          ! load idt with 0,0
        ! 加载 idt
180     lgdt    gdt_48          ! load gdt with whatever
appropriate
        ! 加载 gdt
181
182 ! that was painless, now we enable A20
    ! 以上的操作都很简单, 下面我们开启 A20 地址线
183
184     call    empty_8042      ! 等待 8042 缓冲器空, 只有为空时

```

```

    ! 才可以写
185     mov      al,#0xD1           ! command write
186     out      #0x64,al
        ! 0xD1 命令码，写到 8042 的 P2 端口，
        ! P2 端口的位 1 用于选通 A20 地址线

187     call     empty_8042   ! 等待 8042 缓冲器空
188     mov      al,#0xDF           ! A20 on
189     out      #0x60,al ! 选通A20 地址线的参数
190     call     empty_8042   ! 等待 8042 缓冲器空
191
192 ! make sure any possible coprocessor is properly reset..
    ! 确认协处理器
193
194     xor      ax,ax
195     out      #0xf0,al
196     call     delay
197     out      #0xf1,al
198     call     delay

    ! 我找不到 0xF0-0xF1 端口的资料，那位请提供给我，谢谢！
    !!!! 猜测是和协处理器相关的

199
200 ! well, that went ok, I hope. Now we have to reprogram the
  interrupts :-(

201 ! we put them right after the intel-reserved hardware interrupts,
  at

202 ! int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
203 ! messed this up with the original PC, and they haven't been able
  to

204 ! rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
205 ! which is used for the internal hardware interrupts as well. We
  just

206 ! have to reprogram the 8259's, and it isn't fun.

    ! 好，我希望上面是对的，下面我们不得不对中断进行重新编码
    ! 我们把它们放在 INTEL 保留的硬件中断后面，在 0x20-0x2F 之间。在
    ! 那里它们不会引起冲突，不幸的是 IBM 在原 PC 机中搞糟了，在以后
    ! 也没有纠正过来。Bios 把中断放在 0x08-0x0f，内部硬件也使用这些中
    ! 断，所以，我们必须对 8259 重新编程。

207
208     mov      al,#0x11           ! initialization sequence
        ! 0x11 表示初始化命令开始，是 ICW1 命令字，表示边
        ! 沿触发、多片 8259 级连、最后要发送 ICW4 命令字。
209     out      #0x20,al           ! send it to 8259A-1
        ! 发送到 8259A 主芯片。

```

```

210      call    delay
211      out    #0xA0,al           ! and to 8259A-2
! 发送到 8259A 从芯片。
212      call    delay
213      mov    al,#0x20          ! start of hardware int's
(0x20)
214      out    #0x21,al
! 送主芯片 ICW2 命令字，起始中断号，要送奇地址。
215      call    delay
216      mov    al,#0x28          ! start of hardware int's 2
(0x28)
217      out    #0xA1,al
! 送从芯片 ICW2 命令字，从芯片的起始中断号。
218      call    delay
219      mov    al,#0x04          ! 8259-1 is master
220      out    #0x21,al
! 送主芯片 ICW3 命令字，主芯片的 IR2 连从芯片 INT。
221      call    delay
222      mov    al,#0x02          ! 8259-2 is slave
223      out    #0xA1,al
! 送从芯片 ICW3 命令字，表示从芯片的 INT 连到主芯
224      call    delay
225      mov    al,#0x01          ! 8086 mode for both
226      out    #0x21,al
! 送主芯片 ICW4 命令字。8086 模式；普通 EOI 方式，
! 需发送指令来复位。初始化结束，芯片就绪。
227      call    delay
228      out    #0xA1,al          ! 送从芯片 ICW4 命令字，内容同上。
229      call    delay
230      mov    al,#0xFF          ! mask off all interrupts for
now
231      out    #0xA1,al          ! 屏蔽主芯片所有中断请求
232      call    delay
233      mov    al,#0xFB          ! mask all irq's but irq2
which
234      out    #0x21,al          ! is cascaded
! 屏蔽从芯片所有中断请求，除了 irq2
235
236 ! well, that certainly wasn't fun :-(. Hopefully it works, and
we don't
237 ! need no steenkng BIOS anyway (except for the initial loading :-).
238 ! The BIOS-routine wants lots of unnecessary data, and it's less
239 ! "interesting" anyway. This is how REAL programmers do it.
240 !

```

```

241 ! Well, now's the time to actually move into protected mode. To
make
242 ! things as simple as possible, we do no register set-up or anything,
243 ! we let the gnu-compiled 32-bit programs do that. We just jump
to
244 ! absolute address 0x00000, in 32-bit protected mode.
245 !
246 ! Note that the short jump isn't strictly needed, althought there
are
247 ! reasons why it might be a good idea. It won't hurt in any case.
248 !

```

！ 哼，上面这段当然没劲，希望这样能工作，而且我们也不再需要乏味的BIOS 了（除了初始的加载）。BIOS 子程序要求很多不必要的数据，而且它一点都没趣。那是“真正”的程序员所做的事。

！ 好，到这里我们真正进入 32 位保护模式运行。我们尽量使得事情简单化，我们不做任何寄存器设置，我们让 gun 的 32 位编译程序做那事情，我们跳转到 32 位保护方式下绝对地址 0（其实并不是跳转到 0 地址处，而是跳转到 0x1000 处，我想这样写的原因是要告诉我们，目前保护方式下的地址等于物理地址）。

！注意：虽然短跳转不是绝对的，但是这样看来是个好主意，因为我们不想有任何错误发生。

```

249     mov      ax,#0x0001      ! protected mode (PE) bit
250     lmsw
251     ax          ! This is it!
252     jmp      flush_instr ! 跳转到flush_instr
253     flush_instr:
254     jmpi    0x1000,KERNEL_CS      ! jmp offset 1000 of segment
0x10 (cs)           ! 间接跳转到cs=0x10
                           ! offset=0x1000 位置处
                           ! 也就是 system 模块的 head.s 中
                           ! 已经进入保护模式，此时跳到 0x10:0x1000 地址处
                           ! 到这里为止没有开启分页，并且 gdt 是从 0 开始为基地址的
                           ! 所以这里的偏移地址 0x1000 就是物理地址 0x1000
                           ! 所以这里也就跳到了 zBoot/head.s 中去了
                           !!! 特别要注意的是这里的 cs 已经实模式下的段基址了，
                           !!! 而是段选择符号了。具体描述请看基础部分。

```

```

255 ! This routine checks that the keyboard command queue is empty
256 ! (after emptying the output buffers)
257 !
258 ! No timeout is used - if this hangs there is something wrong with
259 ! the machine, and we probably couldn't proceed anyway.
                           ! 下面这个子程序检查键盘命令队列是否为空。这里不使用超时方法 - 如果这里死机，则
                           ! 说明PC 机有问题，我们就没有办法再处理下去了。
                           ! 只有当输入缓冲器为空时（状态寄存器位2 = 0）才可以对其进行写命令。
260 empty_8042:

```

```

261      call    delay ! 等待 (起延时作用)
262      in     al,#0x64      ! 8042 status port
          ! 读 AT 键盘控制器状态寄存器。
263      test   al,#1       ! output buffer?
          ! 测试位 1
264      jz     no_output
265      call    delay
266      in     al,#0x60      ! read it
267      jmp    empty_8042
268 no_output:        ! 没有输出
269      test   al,#2       ! is input buffer full?
          ! 测试位 2, 输入缓冲器满?
270      jnz   empty_8042    ! yes - loop
271      ret
272 !
273 ! Read a key and return the (US-)ascii code in al, scan code in
ah
274 !
275 getkey:
276      xor    ah,ah
277      int    0x16      ! 从键盘读字符
          ! al = 键盘状态字节
278      ret
279
280 !
281 ! Read a key with a timeout of 30 seconds. The cmos clock is used
to get
282 ! the time.
283 !
          ! 30 秒内读一个按键, 从 CMOS 时钟中得到时间 (该子程序, 首先获得当前
CMOS 中的 tick 数, 然后加上 30 秒后得到超时的值, 当在 30 秒中没有键盘
被按下时, 将会返回默认的字符空格, 否则会直接调用 getkey, 得到
被按下的键码后, 返回到调用该子程序处)
284 getkt:
285      call    gettimeofday ! 从coms中获取当前ticks
286      add    al,#30       ! wait 30 seconds
          ! 把获得的 ticks 加上 30
          ! 即是超时的时间
287      cmp    al,#60
288      jl     lminute    ! 小于 60, 则跳到lminute
289      sub    al,#60      ! 否则, 减去 60
290 lminute:
291      mov    cl,al ! 秒数 (小于 60)
292 again:  mov    ah,#0x01

```

```

293     int      0x16      ! 读键盘缓冲区字符
294     jnz      getkey      ! key pressed, so get it
          ! 缓冲中有字符，则读取
          ! 该字符
295     call     gettimeofday ! 如果缓冲中没有字符，则
          ! 会调用 gettimeofday 来从 CMOS 中
          ! 得到时间
296     cmp      al,cl      ! 看再次得到时间是否和上面保存的相等
297     jne      again      ! 不等，则继续读键盘缓冲
298     mov      al,#0x20    ! timeout, return default char
`space'
299     ret      ! 超时，返回缺省的值 0x20(空格字符)
300
301 !
302 ! Flush the keyboard buffer
303 !
          ! 清空键盘缓冲
304 flush:  mov      ah,#0x01    ! ah = 0x01
305     int      0x16      ! 调用bios 16 中断,
          ! 读键盘缓冲区字符
          ! ZF=0 AL=字符码

```

AH=扫描码

```

          ZF=1 缓冲区空
306     jz      empty      ! ZF=1 缓冲空，意味键盘缓冲空
307     xor      ah,ah      ! 否则，ah = 0
308     int      0x16      ! 从键盘读字符
309     jmp      flush      ! 继续读
310 empty:  ret
311
312 !
313 ! Read the cmos clock. Return the seconds in al
314 !
          ! 读取 CMOS 时钟，结果放在 AL 中
315 gettimeofday:
316     push     cx      ! cx入栈
317     mov      ah,#0x02    ! ah = 0x02
318     int      0x1a      ! 读实时钟
          CH:CL=时:分(BCD)
          DH:DL=秒:1/100 秒(BCD)
319     mov      al,dh      ! dh contains the seconds
          ! 秒值付给 al
320     and      al,#0x0f    ! 清al中的高 4 位

```

```

321      mov    ah,dh      ! 秒值付给ah
322      mov    cl,#0x04    ! cl = 4
323      shr    ah,cl      ! ah = ah/16
324      aad             ! al = 秒数, ah = 0
325      pop    cx         ! 弹出入栈的cx
326      ret              ! 返回调用处
327
328 !
329 ! Delay is needed after doing i/o
330 !
! 做完 I/O 后, 延时是必要的
331 delay:
332     .word   0x00eb          ! jmp $+2
! 起延时作用, jmp $+2 的机器码
333     ret                 ! 返回
334
335 ! Routine trying to recognize type of SVGA-board present (if any)
336 ! and if it recognize one gives the choices of resolution it offers.
337 ! If one is found the resolution chosen is given by al,ah
(rows,cols).
338
! 该子程序尝试给出确认 SVGA 板的类型的提示 (如果可能)。最终
! 确认用户的选择, 假如发现了一个选择将会在 al,ah 中给出 (行, 列)
339 chsvga: cld          ! DF = 0
340     push   ds           ! 此时 ds = 0x9000,cs = 0x9020
341     push   cs           ! ds,cs入栈
342     mov    ax,[0x01fa]   ! 取得 0x9000:0x1FA处定义的值
! 该值定义在 bootsect.s 中的
! vid_mode
343     pop    ds           ! ds 出栈,此时ds=0x9020
! 为的是可以访问以此为地址的
! 段
344     mov    modesave,ax  ! 定义在第 861 行 (2 个字节)
345     mov    ax,#0xc000    !
346     mov    es,ax        ! es = ax = 0xC000
347     mov    ax,modesave  ! 取的上面保存的值
348     cmp    ax,#NORMAL_VGA ! 与NORMAL_VGA比较
349     je     defvga       ! 相等则转到defvga
350     cmp    ax,#EXTENDED_VGA ! 否则和EXTENDED_VGA比较
351     je     vga50        ! 是, 跳转到vga50
352     cmp    ax,#ASK_VGA  ! 和ASK_VGA比较
353     jne    svga         ! 否, 则跳转到svga
354     lea    si,msg1      ! si = msg1
355     call   prtstr       ! 调用 prtstr, 打印msg1

```

```

356      call    flush      ! 等待键盘缓冲空
357 nokey:  call    getkt      ! 调用getkt
358      cmp     al,#0x0d      ! enter ?
359      je     svga        ! yes - svga selection
360      cmp     al,#0x20      ! space ?
361      je     defvga     ! 是, 则跳转到 defvga
362      call    beep
363      jmp    nokey     ! 跳转到nokey
364 defvga: mov    ax,#0x5019    ! ax = 0x5019
365      pop    ds        ! 弹出栈上内容到ds, 令ds=0x9000
366      ret
367 /* extended vga mode: 80x50 */
368      ! 扩展vga的模式
368 vga50:
369      mov    ax,#0x1112    ! ax = 0x1112
370      xor    bl,bl      ! bl = 0
371      int    0x10        ! use 8x8 font set (50 lines on VGA)
371      ! 调用 0x10 中断
372      mov    ax,#0x1200    ! ax = 0x1200
373      mov    bl,#0x20      ! bl = 0x20
374      int    0x10        ! use alternate print screen
375      mov    ax,#0x1201    ! ax = 0x1201
376      mov    bl,#0x34      ! bl = 0x34
377      int    0x10        ! turn off cursor emulation
377      ! 关闭鼠标
378      mov    ah,#0x01      ! ah = 0x01
379      mov    cx,#0x0607    ! cx = 0x0607
380      int    0x10        ! turn on cursor (scan lines 6 to 7)
380      ! 打开鼠标
381      pop    ds        ! 出栈到ds
382      mov    ax,#0x5032    ! return 80x50
383      ret
384 /* extended vga mode: 80x28 */
384      ! 扩展vga模式
385 vga28:
386      pop    ax        ! clean the stack
387      mov    ax,#0x1111    ! ax = 0x1111
388      xor    bl,bl      ! bl = 0
389      int    0x10        ! use 9x14 fontset (28 lines on VGA)
390      mov    ah, #0x01      ! ah = 0x01

```

```

391      mov    cx,#0x0b0c      ! cx = 0x0b0c
392      int    0x10      ! turn on cursor (scan lines 11 to 12)
                  ! 打开鼠标
393      pop    ds      ! 出栈到ds
394      mov    ax,#0x501c      ! return 80x28
395      ret
396 /* svga modes */
                  ! svga模式
      ! 请注意！从这里开始到第 711 行，主要用于 svga 显示模式的确认！我找不到其硬件资料，所以这一段我几乎没有注释！不过这对理解核心是没有影响的，读者可以完全忽略这一段。（如果有朋友可以提供给我详细的资料，我非常乐意的补上这段，在这先谢了）
397 svga:   cld      ! DF=0
398         lea    si,id9GXE      ! Check for the #9GXE
(jyanowit@orixa.mtholyoke.edu, thanks dlm40629@uxa.cso.uiuc.edu)
                  ! 把id9GXE地址送给si
399      mov    di,#0x49      ! id string is at c000:049
                  ! id 字符串存放在 0xC000:0x49
400      mov    cx,#0x11      ! length of "Graphics Power By"
401      repe      ! cx = "Graphics Power By"的长度
402      cmpsb      ! 查找es:[di] ds:[si]中不同的字节,
403      jne    of1280      ! 有不同的（请注意es在上面的第
                  ! 345行设置）
                  ! 则 ZF=0,便跳转到 of1280
404 is9GXE: lea    si,dsc9GXE      ! table of descriptions of
                  ! video modes for BIOS
405         lea    di,mo9GXE      ! table of sizes of video
                  ! modes for my BIOS
406         br    selmod      ! go ask for video mode
407 of1280: cld
408         lea    si,idf1280      ! Check for Orchid F1280
(dingbat@diku.dk)
                  ! si = idf1280
409      mov    di,#0x10a      ! id string is at c000:010a
410      mov    cx,#0x21      ! length
411      repe
412      cmpsb
413      jne    nf1280      ! 见上述注释（雷同 398-403）
414 isVRAM: lea    si,dscf1280      !!!!  

415         lea    di,mof1280      !!!!  

416         br    selmod      ! 这段不明白，没有资料
417 nf1280: lea    si,idVRAM
418         mov    di,#0x10a
419         mov    cx,#0x0c

```

```

420      repe
421      cmpsb
422      je      isVRAM      ! 见上述注释 (雷同 398-403)
423      cld
424      lea      si,idati      ! Check ATI 'clues'
425      mov      di,#0x31      ! 检查是否是Ati线索
                           ! 找不到资料
426      mov      cx,#0x09
427      repe
428      cmpsb
429      jne      noati
430      lea      si,dscati
431      lea      di,moati
432      br      selmod      ! 见上述注释 (雷同 398-403)
433 noati:  mov      ax,#0x200f      ! Check Ahead 'clues'
434      mov      dx,#0x3ce      ! 检查是否是Ati线索
                           ! 找不到资料
435      out     dx,ax
436      inc      dx
437      in      al,dx
438      cmp      al,#0x20
439      je      isahed
440      cmp      al,#0x21
441      jne      noahed
442 isahed: lea      si,dscahead
443      lea      di,moahead
444      br      selmod
445 noahed: mov      dx,#0x3c3      ! Check Chips & Tech. 'clues'
                           ! 检查 Chips & Tech
                           ! 找不到资料
446      in      al,dx
447      or      al,#0x10
448      out     dx,al
449      mov      dx,#0x104
450      in      al,dx
451      mov      bl,al
452      mov      dx,#0x3c3
453      in      al,dx
454      and     al,#0xef
455      out     dx,al
456      cmp      bl,[idcandt]
457      jne      nocant
458      lea      si,dscinandt
459      lea      di,mocandt

```

```

460      br      selmod
461  nocant: mov     dx,#0x3d4          ! Check Cirrus 'clues'
462      mov     al,#0x0c           ! 检查是否是Cirrus线索
                                         ! 找不到资料
463      out    dx,al
464      inc    dx
465      in     al,dx
466      mov    bl,al
467      xor    al,al
468      out    dx,al
469      dec    dx
470      mov    al,#0x1f
471      out    dx,al
472      inc    dx
473      in     al,dx
474      mov    bh,al
475      xor    ah,ah
476      shl    al,#4
477      mov    cx,ax
478      mov    al,bh
479      shr    al,#4
480      add    cx,ax
481      shl    cx,#8
482      add    cx,#6
483      mov    ax,cx
484      mov    dx,#0x3c4
485      out    dx,ax
486      inc    dx
487      in     al,dx
488      and    al,al
489      jnz    nocirr
490      mov    al,bh
491      out    dx,al
492      in     al,dx
493      cmp    al,#0x01
494      jne    nocirr
495      call   rst3d4
496      lea    si,dsccirrus
497      lea    di,mocirrus
498      br     selmod
499  rst3d4: mov     dx,#0x3d4
500      mov    al,bl
501      xor    ah,ah
502      shl    ax,#8

```

```

503      add    ax,#0x0c
504      out    dx,ax
505      ret
506 nocirr: call    rst3d4           ! Check Everex 'clues'
               ! 检查是否是 Everex
               ! 找不到资料

507      mov    ax,#0x7000
508      xor    bx,bx
509      int    0x10
510      cmp    al,#0x70
511      jne    noevrx
512      shr    dx,#4
513      cmp    dx,#0x678
514      je     istridd
515      cmp    dx,#0x236
516      je     istridd
517      lea    si,dsceverex
518      lea    di,moeverex
519      br    selmod
520 istridd: lea    cx,ev2tri
521      jmp    cx
522 noevrx: lea    si,idgenoa        ! Check Genoa 'clues'
               ! 检查是否是 Genoa
               ! 找不到资料

523      xor    ax,ax
524      seg    es
525      mov    al,[0x37]
526      mov    di,ax
527      mov    cx,#0x04
528      dec    si
529      dec    di
530 l1:   inc    si
531      inc    di
532      mov    al,(si)
533      test   al,al
534      jz     l2
535      seg    es
536      cmp    al,(di)
537 l2:   loope  l1
538      cmp    cx,#0x00
539      jne    nogen
540      lea    si,dscgenoa

```

```

541      lea      di,mogenoa
542      br       selmod
543 nogen: cld
544      lea      si,idoakvga
545      mov      di,#0x08
546      mov      cx,#0x08
547      repe
548      cmpsb
549      jne      nooak
550      lea      si,dscoakvga
551      lea      di,mooakvga
552      br       selmod
553 nooak: cld
554      lea      si,idparadise           ! Check Paradise 'clues'
                                         ! 检查是否是 Paradise
                                         ! 找不到资料

555      mov      di,#0x7d
556      mov      cx,#0x04
557      repe
558      cmpsb
559      jne      nopara
560      lea      si,dscparadise
561      lea      di,moparadise
562      br       selmod
563 nopara: mov      dx,#0x3c4           ! Check Trident 'clues'
                                         ! 检查是否是 Trident
                                         ! 找不到资料
564      mov      al,#0x0e
565      out     dx,al
566      inc      dx
567      in      al,dx
568      xchg   ah,al
569      xor      al,al
570      out     dx,al
571      in      al,dx
572      xchg   al,ah
573      mov      bl,al           ! Strange thing ... in the book this
wasn't
574      and     bl,#0x02           ! necessary but it worked on my card
which
575      jz      setb2           ! is a trident. Without it the screen
goes
576      and     al,#0xfd           ! blurred ...

```

```

577      jmp    clrb2          !
578 setb2: or     al,#0x02      !
579 clrb2: out   dx,al
580      and   ah,#0x0f
581      cmp   ah,#0x02
582      jne   notrid
583 ev2tri: lea    si,dsctrident
584      lea    di,motrident
585      jmp    selmod
586 notrid: mov   dx,#0x3cd      ! Check Tseng 'clues'
587      in    al,dx          ! Could things be this
simple ! :-)
588      mov    bl,al
589      mov    al,#0x55
590      out   dx,al
591      in    al,dx
592      mov    ah,al
593      mov    al,bl
594      out   dx,al
595      cmp   ah,#0x55
596      jne   notsen
597      lea    si,dsctseng
598      lea    di,motseng
599      jmp    selmod
600 notsen: mov   dx,#0x3cc      ! Check Video7 'clues'
                           ! 检查是否是 Video7
                           ! 找不到资料
601      in    al,dx
602      mov   dx,#0x3b4
603      and   al,#0x01
604      jz    even7
605      mov   dx,#0x3d4
606 even7: mov   al,#0x0c
607      out   dx,al
608      inc   dx
609      in    al,dx
610      mov   bl,al
611      mov   al,#0x55
612      out   dx,al
613      in    al,dx
614      dec   dx
615      mov   al,#0x1f
616      out   dx,al
617      inc   dx

```

```
618      in     al,dx
619      mov    bh,al
620      dec    dx
621      mov    al,#0x0c
622      out   dx,al
623      inc    dx
624      mov    al,bl
625      out   dx,al
626      mov    al,#0x55
627      xor    al,#0xea
628      cmp    al,bh
629      jne   novid7
630      lea    si,dscvideo7
631      lea    di,movideo7
632      jmp   selmod
633 novid7: lea    si,dsunknown
634      lea    di,mounknown
635 selmod: xor   cx,cx
636      mov    cl,(di)
637      mov    ax,modesave
638      cmp    ax,#ASK_VGA
639      je    askmod
640      cmp    ax,#NORMAL_VGA
641      je    askmod
642      cmp    al,cl
643      jl    gotmode
644      push   si
645      lea    si,msg4
646      call   prtstr
647      pop    si
648 askmod: push   si
649      lea    si,msg2
650      call   prtstr
651      pop    si
652      push   si
653      push   cx
654 tbl:   pop    bx
655      push   bx
656      mov    al,bl
657      sub    al,cl
658      call   modepr
659      lodsw
660      xchg   al,ah
661      call   dprnt
```

```

662      xchg    ah,al
663      push    ax
664      mov     al,#0x78
665      call    prnt1
666      pop     ax
667      call    dprnt
668      push    si
669      lea     si,crlf      ! print CR+LF
670      call    prtstr
671      pop     si
672      loop    tbl
673      pop     cx
674      lea     si,msg3
675      call    prtstr
676      pop     si
677      add    cl,#0x30
678      jmp    nonum
679  nonumb: call    beep
680  nonum:  call    getkey
681      cmp    al,#0x30      ! ascii `0'
682      jb     nonumb
683      cmp    al,#0x3a      ! ascii `9'
684      jbe    number
685      cmp    al,#0x61      ! ascii `a'
686      jb     nonumb
687      cmp    al,#0x7a      ! ascii `z'
688      ja     nonumb
689      sub    al,#0x27
690      cmp    al,cl
691      jae    nonumb
692      sub    al,#0x30
693      jmp    gotmode
694  number: cmp    al,cl
695      jae    nonumb
696      sub    al,#0x30
697  gotmode: xor    ah,ah
698      or     al,al
699      beq    vga50
700      push   ax
701      dec    ax
702      beq    vga28
703      add    di,ax
704      mov    al,(di)
705      int    0x10

```

```

706      pop     ax
707      shl     ax,#1
708      add     si,ax
709      lodsw
710      pop     ds
711      ret
712
713 ! Routine to print asciiz-string at DS:SI
714
    ! 该子程序打印 asciiz 码，来之于 ds:si
715 prtstr: lodsb      ! 从ds:si中取一个字符送到al中
716      and     al,al ! 检查取得字符是否是 0
717      jz      fin   ! 是则结束打印
718      call    prnt1 ! 否，调用prnt1打印
719      jmp     prtstr ! 跳转到prtstr继续打印
720 fin:   ret      ! 该子程序结束
721
722 ! Routine to print a decimal value on screen, the value to be
723 ! printed is put in al (i.e 0-255).
    ! 该子程序在屏幕上打印 10 进制数，值存放在 al 中，大小在 0-255
724
725 dprnt: push    ax
726      push    cx      ! ax,cx入栈
727      xor     ah,ah   ! Clear ah
    ! 高字节清零，要打印的字节在 al 中
728      mov     cl,#0x0a  ! cl = 0x0A
729      idiv   cl      ! ax/cl,ah=余数, al=商
730      cmp     al,#0x09  ! 商 - 0x09,如果al小于或者等于
    ! 0，则跳转到 lt100, 打印之，否则
    ! 跳转到 dprnt
731      jbe    lt100
732      call   dprnt
733      jmp    skip10
734 lt100: add    al,#0x30  ! 打印高 4 位
735      call   prnt1
736 skip10: mov    al,ah   ! 打印低 4 位
737      add    al,#0x30
738      call   prnt1
739      pop    cx
740      pop    ax
741      ret
742
743 !
744 ! Routine to print the mode number key on screen. Mode numbers

```

```

745 ! 0-9 print the ascii values `0' to '9', 10-35 are represented
by
746 ! the letters `a' to `z'. This routine prints some spaces around
the
747 ! mode no.
748 !
749
    ! 该子程序在屏幕上打印模数键，模数 0-9 对应于 ascii 的 ‘0’ - ‘9’ ,
    ! 10 - 35 对应于 ascii 的 ‘a’ - ‘z’， 并且还打印一些空格符。
750 modepr: push    ax          ! ax入栈
751         cmp     al,#0x0a      ! 与 0x0a比较，如果大于 0x0a则跳
    ! 转到 digit, 如果小于 0x0a 则会
    ! 先加上 0x27 后，在加上 0x30 转换
    ! 成 ascii 码
752         jb      digit       ! Here is no check for number > 35
753         add     al,#0x27      ! 加上 0x27 (转换成ascii码)
754 digit:  add     al,#0x30
755         mov     modenr, al      ! 把结果保存在modenr中
756         push    si
757         lea     si, modestring
758         call    prtstr       ! 打印modestring
759         pop     si
760         pop     ax
761         ret      ! 返回
762
763 ! Part of above routine, this one just prints ascii al
    ! prnt1 子程序用来打印 al 中的 ascii 码
764
765 prnt1:  push    ax
766         push    cx          ! ax,cx分别入栈
767         xor    bh,bh ! bh = 0
768         mov     cx,#0x01 ! cx = 1
769         mov     ah,#0x0e ! ah = 0xE
770         int    0x10 ! 调用bios的 10 号中断打印al
771         pop     cx
772         pop     ax          ! cx,ax出栈
773         ret      ! 返回
774
775 beep:   mov     al,#0x07      ! al = 0x07(查不到 0x7 的作用)
776         jmp     prnt1 ! 请看prnt1注释
777
    ! 设置全局描述符表，每一项是 8 个字节
778 gdt:
779         .word   0,0,0,0          ! dummy

```

! intel 规定第 0 项不用，为了防止  
系统没有正确的初始化而进入保护模式

```

780
781     .word  0,0,0,0          ! unused
           ! 第一项也没有使用
782
783     .word  0x07FF          ! 8Mb - limit=2047 (2048*4096=8Mb)
784     .word  0x0000          ! base address=0
785     .word  0x9A00          ! code read/exec
786     .word  0x00C0          ! granularity=4096, 386
           ! 第二项用于代码段，大小是 8M,
           ! 起始于地址 0!
787
788     .word  0x07FF          ! 8Mb - limit=2047 (2048*4096=8Mb)
789     .word  0x0000          ! base address=0
790     .word  0x9200          ! data read/write
791     .word  0x00C0          ! granularity=4096, 386
           ! 第三项用于数据段，大小是 8M,
           ! 起始于地址 0!
792
793 idt_48:
794     .word  0                  ! idt limit=0
795     .word  0,0                ! idt base=0L
           ! 用于设置 idt，其实现在设置
           ! 只是为了能够顺利的切换到
           ! 保护模式。所以这里将其基址
           ! 设为 0，大小也是 0。
796
797 gdt_48:
798     .word  0x800             ! gdt limit=2048, 256 GDT entries
799     .word  512+gdt,0x9      ! gdt base = 0X9xxxx
           ! 就如同上面的 idt_48 一样，这里
           ! 设的 gdt_48，也是为了切换的方便
           ! 后面会重新在设置
           ! 从下面的内容开始直到结尾，是要直接打印在屏幕上的 ascii 码，
           ! 和一些常数值！
800
801 msg1:          .ascii  "Press <RETURN> to see SVGA-modes available,
<SPACE> to continue or wait 30 secs."
802                 db      0x0d, 0x0a, 0x0a, 0x00
803 msg2:          .ascii  "Mode: COLSxROWS:"
804                 db      0x0d, 0x0a, 0x0a, 0x00
805 msg3:          db      0x0d, 0x0a
806                 .ascii  "Choose mode by pressing the corresponding"

```

```

number or letter."
807 crlf:          db      0x0d, 0x0a, 0x00
808 msg4:          .ascii  "You passed an undefined mode number to
setup. Please choose a new mode."
809                 db      0x0d, 0x0a, 0x0a, 0x07, 0x00
810 modestring:    .ascii  "
811 modenr:        db      0x00      ! mode number
812                 .ascii  ":"      "
813                 db      0x00
814
815 idati:         .ascii  "761295520"
816 idcandt:       .byte   0xa5
817 idgenoa:       .byte   0x77, 0x00, 0x99, 0x66
818 idparadise:   .ascii  "VGA="
819 idoakvga:      .ascii  "OAK VGA "
820 idf1280:        .ascii  "Orchid Technology Fahrenheit 1280"
821 id9GXE:         .ascii  "Graphics Power By"
822 idVRAM:        .ascii  "Stealth VRAM"
823
824 ! Manufacturer: Numofmodes+2: Mode:
825 ! Number of modes is the number of chip-specific svga modes plus
the extended
826 ! modes available on any vga (currently 2)
827
828 moati:         .byte   0x04, 0x23, 0x33
829 moahead:        .byte   0x07, 0x22, 0x23, 0x24, 0x2f, 0x34
830 mocandt:        .byte   0x04, 0x60, 0x61
831 mocirrus:      .byte   0x06, 0x1f, 0x20, 0x22, 0x31
832 moeverex:      .byte   0x0c, 0x03, 0x04, 0x07, 0x08, 0x0a, 0x0b,
0x16, 0x18, 0x21, 0x40
833 mogenoa:       .byte   0x0c, 0x58, 0x5a, 0x60, 0x61, 0x62, 0x63,
0x64, 0x72, 0x74, 0x78
834 moparadise:    .byte   0x04, 0x55, 0x54
835 motrident:     .byte   0x09, 0x50, 0x51, 0x52, 0x57, 0x58, 0x59,
0x5a
836 motseng:       .byte   0x07, 0x26, 0x2a, 0x23, 0x24, 0x22
837 movideo7:      .byte   0x08, 0x40, 0x43, 0x44, 0x41, 0x42, 0x45
838 mooakvga:      .byte   0x08, 0x00, 0x07, 0x4e, 0x4f, 0x50, 0x51
839 mo9GXE:         .byte   0x04, 0x54, 0x55
840 mof1280:        .byte   0x04, 0x54, 0x55
841 mouknown:      .byte   0x02
842
843 !           msb = Cols lsb = Rows:
844 ! The first two modes are standard vga modes available on any vga.

```

```

845 ! mode 0 is 80x50 and mode 1 is 80x28
846
847 dscati:      .word 0x5032, 0x501c, 0x8419, 0x842c
848 dscahead:     .word 0x5032, 0x501c, 0x842c, 0x8419, 0x841c,
0xa032, 0x5042
849 dsccandt:     .word 0x5032, 0x501c, 0x8419, 0x8432
850 dsccirrus:    .word 0x5032, 0x501c, 0x8419, 0x842c, 0x841e,
0x6425
851 dsceverex:    .word 0x5032, 0x501c, 0x5022, 0x503c, 0x642b,
0x644b, 0x8419, 0x842c, 0x501e, 0x641b, 0xa040, 0x841e
852 dscgenoa:     .word 0x5032, 0x501c, 0x5020, 0x642a, 0x8419,
0x841d, 0x8420, 0x842c, 0x843c, 0x503c, 0x5042, 0x644b
853 dscparadise:  .word 0x5032, 0x501c, 0x8419, 0x842b
854 dsctrident:   .word 0x5032, 0x501c, 0x501e, 0x502b, 0x503c,
0x8419, 0x841e, 0x842b, 0x843c
855 dsctseng:     .word 0x5032, 0x501c, 0x503c, 0x6428, 0x8419,
0x841c, 0x842c
856 dscvideo7:    .word 0x5032, 0x501c, 0x502b, 0x503c, 0x643c,
0x8419, 0x842c, 0x841c
857 dscoakvga:    .word 0x5032, 0x501c, 0x2819, 0x5019, 0x503c,
0x843c, 0x8419, 0x842b
858 dscf1280:     .word 0x5032, 0x501c, 0x842b, 0x8419
859 dsc9GXE:      .word 0x5032, 0x501c, 0x842b, 0x8419
860 dsunknown:    .word 0x5032, 0x501c
861 modesave:     .word SVGA_MODE
862
863
864 .text
865 endtext:
866 .data
867 enddata:
868 .bss
869 endbss:

```

## Boot/head.S

### 概述

head.s 完全是 32 位的汇编代码，该汇编代码是由 AT&T 格式写成。AT&T 格式汇

编和我们比较熟悉 Intel 汇编有点区别，具体请看基础部分。

该代码首先清 BSS 段，然后设置临时的 IDT 中断描述符，具体的中断描述符会在后面设置。然后检查是否真的进入了保护模式，如果没有，则死机。

把在 setup.s 用 BIOS 获得的一些参数放在 empty\_zero\_page 前 2k 中，后 2k 放命令行参数(即启动参数)

检测 CPU 的类型和是否有协处理器。接下来设置分页（注意：只设置 4M 的内存，将在后面根据内存的实际大小设置剩余的内存），分页好后，设置 gdt（全局描述符表）和 idt（中断描述符表），并设置 CS=0x10,DS=ES=FS=GS=0x18,ESP=statck\_start, 最后跳到 start\_kernel 中执行，start\_kernel 是用 C 语言写的，start\_kernel 定义于 init/main.c 中

## 代码分析

```

1  /*
2   *  linux/boot/head.S
3   *
4   *  Copyright (C) 1991, 1992 Linus Torvalds
5   */
6
7  /*
8   *  head.S contains the 32-bit startup code.
9   */
! head.s 中包含 32 位启动代码

10
11 .text
12 .globl _idt,_gdt,
13 .globl _swapper_pg_dir,_pg0
14 .globl _empty_bad_page
15 .globl _empty_bad_page_table
16 .globl _empty_zero_page
17 .globl _tmp_floppy_area,_floppy_track_buffer
18
19 #include <linux/tasks.h>
20 #include <linux/segment.h>
21
22 #define CL_MAGIC_ADDR    0x90020
23 #define CL_MAGIC          0xA33F
24 #define CL_BASE_ADDR      0x90000
25 #define CL_OFFSET         0x90022
26
27 /*
28  * swapper_pg_dir is the main page directory, address 0x00001000
29  * (or at
30  * address 0x00101000 for a compressed boot).
31  */

```

! swapper\_pg\_dir 是主页目录，地址是 0x00001000  
 ! (或者对于压缩内核启动时则在 0x00101000 的地址处)  
 ! 请注意 AT&T 汇编的源地址和目的地址与 Intel 是恰恰相反的。  
 ! 还有从这里开始，段寄存器中的值与实模式下的意义是不同的。  
 ! 在这里表示是段选择符！

```

31  startup_32:
32      cld
33      movl $(KERNEL_DS),%eax      ! eax = 0x18
34      mov %ax,%ds
35      mov %ax,%es
36      mov %ax,%fs
37      mov %ax,%gs      ! ds=es=fs=gs=0x18
                           ! 即 gdt 中的第三项
                           ! 请看在 setup.s 中
的
                           ! gdt_48 注释！
38      lss _stack_start,%esp      ! esp = _stack_start
                           ! 该 esp 即为内核堆栈
                           ! 见 linux\kernel\sched.c 中定义
39  /*
40  * Clear BSS first so that there are no surprises...
41  */
                           ! 首先清 bss，以至没有令人不愉快的事情发生

```

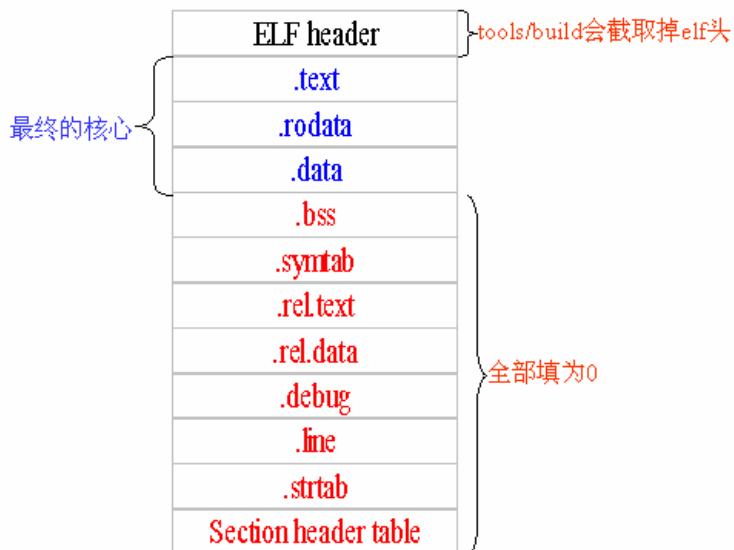


图 B-5

! 1.0 核心编译出来的格式是 ELF, 图 B-5 表示的即为标准的 ELF  
 ! 格式!  
 ! \_\_edata 和 \_\_end 是由 ld 产生的, \_\_edata 表示数据段的结束, \_\_end  
 ! 表示编译出的核心大小。

```

42      xorl %eax,%eax          ! eax = 0
43      movl $__edata,%edi     ! edi = edata地址
           ! edata 由 ld 联结时产生
44      movl $__end,%ecx      ! ecx = end, 同样由
           ! ld 产生
45      subl %edi,%ecx        ! ecx = ecx - edi
46      cld                   ! df = 0
47      rep
48      stosb                ! 1. 重复把零写到es:edi,
           ! 2. edi = edi + 1
           ! 3. 重复执行, 直到 ecx=0
  
```

! 在做完上面这段程序后, 图 B-5 中红色所标的地方, 会全部被  
 ! 清为 0。而 ELF header 会被 tools/build 程序去掉

```

49  /*
50  * start system 32-bit setup. We need to re-do some of the things
done
51  * in 16-bit mode for the "real" operations.
52  */
  
```

! 开始 32 位系统的设置, 我们需要重新做一遍在 16 位实方式下的事情

```

53      call setup_idt          ! 调用setup_idt
54      xorl %eax,%eax        ! eax = 0
  
```

! 下面用于测试 A20 地址线是否真的已经开启, 采用的方法是把数值写到  
 ! 0x0000000 处, 然后查看 0x100000 处是否是这个值, 如果是则表示 A20  
 ! 地址线并没有真正的被打开。所以一直循环下去, 死机。(因为, 在  
 ! 实方式下, 对向高于 1M 地址处写入内容后, 会自动环绕, 即对 1M 取模)

```

55 1:   incl %eax            # check that A20 really IS enabled
56      movl %eax,0x0000000    # loop forever if it isn't
57      cmpl %eax,0x100000
58      je 1b
59  /*
60  * Initialize eflags. Some BIOS's leave bits like NT set. This
would
61  * confuse the debugger if this code is traced.
  
```

! 初始化 eflags, 设置 BIOS 的一些位象 NT 位,  
 ! 这些代码不容易被调试

```

62  * XXX - best to initialize before switching to protected mode.
63  */
  
```

! 切换到保护模式前最好的的初始化

```

64      pushl $0
  
```

```

65      popfl          ! eflags = 0

66  /*
67  * Copy bootup parameters out of the way. First 2kB of
68  * _empty_zero_page is for boot parameters, second 2kB
69  * is for the command line.
70  */
! 拷贝启动的参数放在安全的地方。_empty_zero_page 前 2kb 是专门
! 用于存放启动参数的。后 2kb 是用于命令行参数

71      movl $0x90000,%esi    ! esi = 0x90000
72      movl $_empty_zero_page,%edi
73      movl $512,%ecx        ! ecx = 512
74      cld                  ! df = 0
75      rep
76      movsl                ! 把从 0x90000 开始的
! 512*4 个字节拷贝到
! _empty_zero_page
77      xorl %eax,%eax
78      movl $512,%ecx
79      rep
80      stosl                ! 接下来把从 0x90000+512
! 处开始的 512*4 个字节清零
81      cmpw $(CL_MAGIC),CL_MAGIC_ADDR    ! 比较 0x90020 处的值是否
! 是 0xA33F, 如果不是则
! 跳转到 1 处
82      jne 1f
83      movl $_empty_zero_page+2048,%edi
84      movzwl CL_OFFSET,%esi      ! 取得在 0x90022 处的值后
! 放入 esi 中
85      addl $(CL_BASE_ADDR),%esi ! 上面取得的值加上 0x90000
86      movl $2048,%ecx        ! ecx = 2048
87      rep
88      movsb                ! 把命令行参数, 放在后
! 2kb 中

89 1:
90  /* check if it is 486 or 386. */
91  /*
92  * XXX - this does a lot of unnecessary setup. Alignment checks
don't
93  * apply at our cpl of 0 and the stack ought to be aligned already,
and
94  * we don't need to preserve eflags.
95  */
! 检查 486 或 386

```

```

! xxx - 该段包含许多不必要的设置。
96    movl %esp,%edi          # save stack pointer
      ! 保存堆栈指针
97    andl $0xfffffffffc,%esp # align stack to avoid AC fault
      ! 使堆栈指针以 4 字节为边界寻址
98    movl $3,_x86           ! x86 定义于kernel/sched.c中
99    pushfl                 # push EFLAGS
100   popl %eax              # get EFLAGS
101   movl %eax,%ecx         # save original EFLAGS
102   xorl $0x40000,%eax     # flip AC bit in EFLAGS
      ! 反转标志寄存器中的 AC 位
103   pushl %eax              # copy to EFLAGS
104   popfl                 # set EFLAGS
      ! 把新的 EFLAGS 弹入到
      ! EFLAGS 标志寄存器中
105   pushfl                 # get new EFLAGS
106   popl %eax              # put it in eax
107   xorl %ecx,%eax         # change in flags
      ! 改变 AC 位前和改变后的值, 异或
108   andl $0x40000,%eax     # check if AC bit changed
      ! 看看 AC 位是否改变, 如果改变跳转
      ! is386 处
109   je is386
110   movl $4,_x86           ! 否则, 将 4 送入x86
111   movl %ecx,%eax         ! ecx=改变前的标志值
112   xorl $0x200000,%eax    # check ID flag
      ! 检查 ID 位, 如果可以设置或者清除
      ! 该值, 则说明 CPU 支持 CPUID
113   pushl %eax
114   popfl                 # if we are on a straight 486DX, SX,
or
115   pushfl                 # 487SX we can't change it
116   popl %eax
117   xorl %ecx,%eax
118   andl $0x200000,%eax    ! 查看ID位, 是则是is486
119   je is486
120 isnew: pushl %ecx        # restore original EFLAGS
121   popfl
122   movl $1, %eax          # Use the CPUID instruction to
123   .byte 0x0f, 0xa2        # check the processor type
124   andl $0xf00, %eax       # Set _x86 with the family
125   shr $8, %eax           # returned.
126   movl %eax, _x86
127   movl %edi,%esp         # restore esp

```

```

128      movl %cr0,%eax          # 486+
129      andl $0x80000011,%eax  # Save PG,PE,ET
130      orl $0x50022,%eax     # set AM, WP, NE and MP
131      jmp 2f
132  is486: pushl %ecx          # restore original EFLAGS
133      popfl               ! 恢复为原来的eflags
134      movl %edi,%esp        # restore esp
135      movl %cr0,%eax        # 486
136      andl $0x80000011,%eax  # Save PG,PE,ET
137      orl $0x50022,%eax     # set AM, WP, NE and MP
138      jmp 2f
139  is386: pushl %ecx          # restore original EFLAGS
140      popfl
141      movl %edi,%esp        # restore esp
142      movl %cr0,%eax        # 386
143      andl $0x80000011,%eax  # Save PG,PE,ET
144      orl $2,%eax           # set MP
145  2:   movl %eax,%cr0       ! cr0 等于新的值
146      call check_x87
147      call setup_paging
148      lgdt gdt_descr
149      lidt idt_descr
150      ljmp $(KERNEL_CS),$1f  ! 因为，上面设置了分页，所以这里
                                ! 要一个长跳转，才可以到 1 处
                                ! 同时设置 cs=0x10
151  1:   movl $(KERNEL_DS),%eax  # reload all the segment registers
152      mov %ax,%ds             # after changing gdt.
                                ! 在改变了 gdt 后，重新加载段
                                ! 寄存器为核心数据段
153      mov %ax,%es
154      mov %ax,%fs
155      mov %ax,%gs            ! ds=es=fs=gs=0x18
156      lss _stack_start,%esp   ! 重新设置esp
157      xorl %eax,%eax         ! eax = 0
158      lldt %ax                ! 将 0 加载到局部描述符表寄存器
                                ! ldtr 中
159      pushl %eax              # These are the parameters to main :-)
160      pushl %eax
161      pushl %eax              ! eax=0 入栈 3 次，这里入栈的参数
                                ! 并没有函数取用，可能是为了调式
                                ! 代码用的。
162      cld                     # gcc2 wants the direction flag
cleared at all times           ! df = 0, gcc的需要
163      call _start_kernel      ! 调用start_kernel, 定义于

```

```

        ! init/main.c中

164 L6:
165     jmp L6                      # main should never return here, but
166                                # just in case, we know what happens.
                                ! 正如上面的注释所说的, 如果执行
                                ! 到这里, 肯定是内核出问题了
                                ! 正常情况是不可能执行到这里的

167
168 /*
169 * We depend on ET to be correct. This checks for 287/387.
170 */
                                ! 我们依赖于 ET 的正确与否来检测 287/387 是否存在

171 check_x87:
172     movl $0,_hard_math          ! 定义于kernel/sched.c
173     clts                      ! 清除任务切换标志
174     fninit
175     fstsw %ax
176     cmpb $0,%al
177     je 1f
178     movl %cr0,%eax           /* no coprocessor: have to set bits
*/
179     xorl $4,%eax             /* set EM */
180     movl %eax,%cr0
181     ret                      ! 如果存在, 则跳转到 1 处, 否则改
                                ! 写 cr0
182 .align 2
183 1:    movl $1,_hard_math
184     .byte 0xDB,0xE4          /* fsetpm for 287, ignored by 387
*/
185     ret
186
187 /*
188 * setup_idt
189 *
190 * sets up a idt with 256 entries pointing to
191 * ignore_int, interrupt gates. It doesn't actually load
192 * idt - that can be done only after paging has been enabled
193 * and the kernel moved to 0xC0000000. Interrupts
194 * are enabled elsewhere, when we can be relatively
195 * sure everything is ok.
196 */
                                ! setup_idt
                                ! 在 idt 表中设置 256 个中断门 (ignore_int), 这里设置的并不会被真正加载。
                                ! idt - 在分页启动后, 才可以被使用并且核心会被移到 0xC0000000 处,

```

```

    ! 当相关的设置都 ok 时，会在其适当的地方开启中断。
197 setup_idt:
198     lea ignore_int,%edx      ! edx等于ignore_int地址
199     movl $(KERNEL_CS << 16),%eax      ! eax = 0x10 X 216
200     movw %dx,%ax           /* selector = 0x0010 = cs */
        ! 将偏移地址的低 16 位放
        ! 入 eax 低 16 位中，这个
        ! 时候 eax 含有门描述符的
        ! 低 4 个字节
201     movw $0x8E00,%dx       /* interrupt gate - dpl=0, present
*/
        ! 此时edx含有门描述符的
        ! 高四个字节
202
203     lea _idt,%edi          ! idt是中断描述符表的地址
204     mov $256,%ecx          ! ecx = 256
205 rp_sidt:
206     movl %eax,(%edi)
207     movl %edx,4(%edi)      ! 将中断描述放入表中
208     addl $8,%edi          ! edi指向下一项
209     dec %ecx
210     jne rp_sidt
211     ret                   ! 循环填充 256 个中断描述
        ! 后，退出。
212
213
214 /*
215 * Setup_paging
216 *
217 * This routine sets up paging by setting the page bit
218 * in cr0. The page tables are set up, identity-mapping
219 * the first 4MB. The rest are initialized later.
220 *
221 * (ref: added support for up to 32mb, 17Apr92) -- Rik Faith
222 * (ref: update, 25Sept92) -- croutons@crunchy.uucp
223 * (ref: 92.10.11 - Linus Torvalds. Corrected 16M limit - no upper
memory limit)
224 */
    ! 该子程序通过设置 cr0 中的分页位，来启用分页。这些页表只设置了前 4M。
    ! 剩余将在后面初始化。
225 .align 2                  ! 按 4 个字节对齐
226 setup_paging:
227     movl $1024*2,%ecx      /* 2 pages -
swapper_pg_dir+1 page table */
        ! 2 页-页目录+页表

```

```

228         xorl %eax,%eax          ! eax = 0
229         movl $_swapper_pg_dir,%edi      /* swapper_pg_dir is at
0x1000 */
230             ! edi=swapper_pg_dir
230         cld;rep;stosl          ! 清 0 一页目录和一页页表
231 /* Identity-map the kernel in low 4MB memory for ease of transition
*/
231     ! 映射内存的低 4M
232         movl $_pg0+7,_swapper_pg_dir      /* set present
bit/user r/w */
232     ! pg0 是第一页页表的地址, 7 是表示该页可以读/写/存在
233 /* But the real place is at 0xC0000000 */
233     ! 但是真正的地址在 0xC0000000
234         movl $_pg0+7,_swapper_pg_dir+3072      /* set present
bit/user r/w */
234     ! 在把第一个页表的地址放在 swapper_pg_dir+3072 处
235         movl $_pg0+4092,%edi    ! edi指向页表的最后一项
236         movl $0x03ff007,%eax      /* 4Mb - 4096 + 7 (r/w
user,p) */
236     ! 4M-4096+7 送给 eax
237         std          ! 方向位置位
238 1:      stosl      /* fill the page backwards - more
efficient :- */
239         subl $0x1000,%eax      ! 每填好一页, 物理地址就减 0x1000
240         jge 1b
241         cld
242         movl $_swapper_pg_dir,%eax  ! swapper_pg_dir在 0x1000 处
243         movl %eax,%cr3      /* cr3 - page directory start
*/
244         movl %cr0,%eax
245         orl $0x80000000,%eax
246         movl %eax,%cr0      /* set paging (PG) bit */
246     ! 设置分页位
247         ret      /* this also flushes the
prefetch-queue */
248
249 /*
250 * page 0 is made non-existent, so that kernel NULL pointer
references get
251 * caught. Thus the swapper page directory has been moved to 0x1000
252 *
253 * XXX Actually, the swapper page directory is at 0x1000 plus 1
megabyte,
254 * with the introduction of the compressed boot code.

```

```

Theoretically,
255 * the original design of overlaying the startup code with the
swapper
256 * page directory is still possible --- it would reduce the size
of the kernel
257 * by 2-3k. This would be a good thing to do at some point.....
258 */
! 第 0 页是不存在的，以至于核心 NULL 指针参考它。尽管交换页目录
! 被放在 0x1000 处，对于压缩的内核实际上交换页目录在 1M
! 加 0x1000 处。
! 理论上，最初设计的启动代码将会尽可能的放在交换页目录中
!, 它将减少 2-3k 的内核尺寸，这是好的事情将用于做其它事情.....
259 .org 0x1000
260 _swapper_pg_dir:
261 /*
262 * The page tables are initialized to only 4MB here - the final
page
263 * tables are set up later depending on memory size.
264 */
! 这个页表只初始化 4M 的内存，后来的页表设置将依靠内存的大小
! 来设置
265 .org 0x2000
266 _pg0:
267
268 .org 0x3000
269 _empty_bad_page:
270
271 .org 0x4000
272 _empty_bad_page_table:
273
274 .org 0x5000
275 _empty_zero_page:
276
277 .org 0x6000
278 /*
279 * tmp_floppy_area is used by the floppy-driver when DMA cannot
280 * reach to a buffer-block. It needs to be aligned, so that it isn't
281 * on a 64kB border.
282 */
! 当 DMA 不能直接访问缓冲块时，下面的 tmp_floppy_area 可以供软盘
! 驱动器直接使用，其地址需要对齐，这样就不会跨 64k 边界。
283 _tmp_floppy_area:
284     .fill 1024,1,0
285 /*

```

```

286 * floppy_track_buffer is used to buffer one track of floppy data:
it
287 * has to be separate from the tmp_floppy area, as otherwise a
single-
288 * sector read/write can mess it up. It can contain one full track
of
289 * data (18*2*512 bytes).
290 */
! floppy_track_buffer 被用来缓冲一个软盘磁道的数据：它和
! tmp_floppy_area 之间有个分隔，另外的，一个扇区的读/写会搞糟它。
! 它包含满满一个磁道的数据。
291 _floppy_track_buffer:
292     .fill 512*2*18,1,0
293
294 /* This is the default interrupt "handler" :-) */
! 这是缺省的中断句柄
295 int_msg:
296     .asciz "Unknown interrupt\n"
297 .align 2
298 ignore_int:
299     cld
300     pushl %eax
301     pushl %ecx
302     pushl %edx
303     push %ds
304     push %es
305     push %fs
306     movl $(KERNEL_DS),%eax
307     mov %ax,%ds
308     mov %ax,%es
309     mov %ax,%fs      ! ds=es=fs=0x18
310     pushl $int_msg
311     call _printk      ! 打印int_msg消息
! printk 定义于 kernel/printk.c
312     popl %eax
313     pop %fs
314     pop %es
315     pop %ds
316     popl %edx
317     popl %ecx
318     popl %eax
319     iret      ! 中断返回
320
321 /*

```

```

322 * The interrupt descriptor table has room for 256 idt's
323 */
      ! 中断描述符表中包含 256 个中断
324 .align 4          ! 8 个字节对齐
325 .word 0
326 idt_descr:
327     .word 256*8-1      # idt contains 256 entries
328     .long 0xc0000000+_idt    ! 基地址在 0xC0000000+idt
329
330 .align 4
331 _idt:
332     .fill 256,8,0      # idt is uninitialized
                  ! 填 256 项, 8byte/项, 每项填 0
333
334 .align 4          ! 8 个字节对齐
335 .word 0
336 gdt_descr:
337     .word (8+2*NR_TASKS)*8-1
338     .long 0xc0000000+_gdt    ! 基地址在 0xC0000000+gdt
339
340 /*
341 * This gdt setup gives the kernel a 1GB address space at virtual
342 * address 0xC0000000 - space enough for expansion, I hope.
343 */
344 .align 4
345 _gdt:
346     .quad 0x0000000000000000      /* NULL descriptor */
                  ! NULL 描述符
347     .quad 0x0000000000000000      /* not used */
                  ! 没有使用
348     .quad 0xc0c39a00000fffff      /* 0x10 kernel 1GB code at
0xC0000000 */
349     .quad 0xc0c392000000fffff      /* 0x18 kernel 1GB data at
0xC0000000 */
350     .quad 0x00cbfa000000fffff      /* 0x23 user    3GB code at
0x00000000 */
351     .quad 0x00cbf2000000fffff      /* 0x2b user    3GB data at
0x00000000 */
      ! 0x10 表示核心代码段, 大小 1GB
      ! 0x18 表示核心数据段, 大小 1GB
      ! 0x23 表示用户代码段, 大小 3GB
      ! 0x2b 表示用户数据段, 大小 3GB
352     .quad 0x0000000000000000      /* not used */
353     .quad 0x0000000000000000      /* not used */

```

```

    ! 未用
354     .fill 2*NR_TASKS, 8, 0           /* space for LDT's and TSS's
etc */
    ! 剩余的留给 LDT 和 TSS 使用

```

## D

### Drivers/char/console.c (部分代码)

```

300 /*
301 * gotoxy() must verify all boundaries, because the arguments
302 * might also be negative. If the given position is out of
303 * bounds, the cursor is placed at the nearest margin.
304 */
    ! gotoxy()必须校验所有的边界，因为这些参数可能是负数，假如
    ! 给的位置超过了边界，必须把光标设置在最近的页空白处
    ! 把光标设置到新的坐标
    × new_x=光标所在列号
    × new_y=光标所在行号
305 static void gotoxy(int currcons, int new_x, int new_y)
306 {
307     int max_y;
308
309     if (new_x < 0)
310         x = 0;
    ! 如果x坐标<0,则重新设置为0
311     else
312         if (new_x >= video_num_columns)
    ! 如果 x坐标大于等于显示列数
313         x = video_num_columns - 1;
    ! 设置为最后一列
314     else
315         x = new_x;
    ! 不然就设置为new_x
316     if (decom) {

```

```

    ! 如果定义了模式
317     new_y += top;
318     max_y = bottom;
    ! max_y=模式中定义的行数
319 } else
320     max_y = video_num_lines;
    ! 否则让max_y=显示器可显示行数
321 if (new_y < 0)
322     y = 0;
323 else
324     if (new_y >= max_y)
325         y = max_y - 1;
326     else
327         y = new_y;
    ! 类似于x坐标
328 pos = origin + y*video_size_row + (x<<1);
    ! 重新在显存中位置pos
329 need_wrap = 0;
330 }
331

```

! 根据真实的起始位置，写对应的端口  
 ×offset=卷屏起始地址值

```

338 static inline void __set_origin(unsigned short offset)
339 {
340     unsigned long flags;
341 #ifdef CONFIG_SELECTION
342     clear_selection();
343 #endif /* CONFIG_SELECTION */
344     save_flags(flags); cli();
    ! 保存标志寄存器值并且关闭中断
345     origin = offset;
346     outb_p(12, video_port_reg);
    ! 选择显示控制寄存器R12
347     outb_p(offset >> 8, video_port_val);
    ! 写入起始地址的高字节
348     outb_p(13, video_port_reg);
    ! 选择显示控制寄存器R13
349     outb_p(offset, video_port_val);
    ! 写入起始地址的低字节
350     restore_flags(flags);
    ! 恢复标志寄存器
351 }

```

352

! 根据 currcons, 设置控制台的起始位置  
 ✕ currcons=虚拟控制台号

```

375 static void set_origin(int currcons)
376 {
377     if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
378         return;
  ! 如果是单色显卡, 直接退出
379     if (currcons != fg_console || console_blanked || vcmode == KD_GRAPHICS)
380         return;
381     real_origin = (origin-video_mem_base) >> 1;
  ! 真实的起始位置
382     set_origin(real_origin);
  ! 定义于本文件中
383 }
```

385 /\*
386 \* *Put the cursor just beyond the end of the display adaptor memory.*
387 \*/

! 把光标放在显卡内存的结尾

```

388 static inline void hide_cursor(void)
389 {
390     /* This is inefficient, we could just put the cursor at 0xffff,
391 but perhaps the delays due to the inefficiency are useful for
392 some hardware... */
  ! 这段代码的效率比较低, 我们把光标放在 0xFFFF 处, 但是可能由于延时
  ! 而导致在某些硬件上效率低下。
393     outb_p(14, video_port_reg);
  ! 使用索引寄存器端口选择显示控制数据寄存器 r14
  ! (光标当前显示位置高字节), 然后写入光标当前位置高字节
394     outb_p(0xff&((video_mem_term-video_mem_base)>>9), video_port_val);
395     outb_p(15, video_port_reg);
  ! 光标当前位置低字节写入r15 中
396     outb_p(0xff&((video_mem_term-video_mem_base)>>1), video_port_val);
397 }
398
```

! 设置对应控制台的光标位置  
 ✕ currcons=对应控制台

```

399 static inline void set_cursor(int currcons)
400 {
401     unsigned long flags;
402
403     if (currcons != fg_console || console_blanked || vemode == KD_GRAPHICS)
404         return;
        ! 如果 currcons != 当前的虚拟控制台号，或者控制台是断开的，或者虚拟终端
        ! 是图形显示模式，则直接退出。
405     if (_real_origin != _origin)
406         set_origin(_real_origin);
        ! 设置控制台的起始位置，定义于本文件
407     save_flags(flags); cli();
        ! 保存寄出器值，关闭中断
408     if (deccm) {
409         outb_p(14, video_port_reg);
        ! 使用索引寄存器端口选择显示控制数据寄存器 r14
        ! (光标当前显示位置高字节)，然后写入光标当前位置高字节
410         outb_p(0xff&((pos-video_mem_base)>>9), video_port_val);
411         outb_p(15, video_port_reg);
        ! 光标当前位置低字节写入r15 中
412         outb_p(0xff&((pos-video_mem_base)>>1), video_port_val);
413     } else
414         hide_cursor();
        ! 隐藏光标，定义于本文件
415     restore_flags(flags);
        ! 恢复标志寄存器
416 }

```

! 对应的虚拟控制台中显示的内容，向上滚动一行  
 × currcons=虚拟控制台号  
 × t=顶行行号  
 × b=底行行号

```

418 static void scrup(int currcons, unsigned int t, unsigned int b)
419 {
420     int hardscroll = 1;
421
422     if (b > video_num_lines || t >= b)
423         return;
        ! 如果b大于屏幕所能显示的行数或者t大于等于b，则返回。
424     if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
425         hardscroll = 0;
        ! 如果显卡类型不是 EGA/VGA 彩色模式并且也不是 EGA/VGA 单色模式
        ! 则让hardscroll=0

```

```

426     else if (t || b != video_num_lines)
427         hardscroll = 0;
    ! 如果如果 b 大于屏幕所能显示的行数或者 t 不为 0
    ! 则让hardscroll=0
428     if (hardscroll) {
429         origin += video_size_row;
430         pos += video_size_row;
431         scr_end += video_size_row;
    ! origin 为向下移一下屏幕字符对应的内存位置
    ! pos 为调整后光标的内存位置
    ! scr_end为屏幕最后一行最后一个字符的位置
432         if (scr_end > video_mem_end) {
    ! 如果屏幕最后一行最后一个字符的所对应的内存大小超过了物理显存的大小
    ! 则将屏幕内容内存数据移动到显示内存的起始位置 video_mem_start 处,
    ! 并在新行上填入空格字符。
433         __asm__("cld\n\t"
    ! 清方向位
434         "rep\n\t"
435         "movsl\n\t"
    ! 将当前屏幕内存数据移动到显示内存起始处
436         "movl _video_num_columns,%1\n\t"
437         "rep\n\t"
438         "stosw"
    ! 新行填入空格字符
439         : /* no output */
440         :"a" (video_erase_char),
441         "c" ((video_num_lines-1)*video_num_columns>>1),
442         "D" (video_mem_start),
443         "S" (origin)
444         :"cx","di","si");
445         scr_end -= origin-video_mem_start;
446         pos -= origin-video_mem_start;
447         origin = video_mem_start;
    ! 根据屏幕移动后的情况，重新调整scr_end, pos, origin
448     } else {
    ! 反之，屏幕最后一行最后一个字符的所对应的内存大小
    ! 没有超过了物理显存的大小，则在新行上填入空格字符
449         __asm__("cld\n\t"
    ! 清方向位
450         "rep\n\t"
451         "stosw"
    ! 在新行上填入空格字符
452         : /* no output */
453         :"a" (video_erase_char),

```

```

454         "c" (video_num_columns),
455         "D" (scr_end-video_size_row)
456         :"cx","di");
457     }
458     set_origin(currcons);
! 向显示控制器中写入新的屏幕内容对应的内存起始位置值
! 定义于本文件中
459 } else {
! 否则
460     __asm__("cld\n\t"
! 清方向位
461     "rep\n\t"
462     "movsl\n\t"
463     "movl _video_num_columns,%%ecx\n\t"
! 将t+1 到b的所对应内容移动到t行开始处
464     "rep\n\t"
465     "stosw"
! 填入空格字符
466     :/* no output */
467     :"a" (video_erase_char),
468     "c" ((b-t-1)*video_num_columns>>1),
469     "D" (origin+video_size_row*t),
470     "S" (origin+video_size_row*(t+1))
471     :"cx","di","si");
472 }
473 }
474

```

! 虚拟控制台的光标下移一行,  
! currcons 对应虚拟控制台

```

495 static void lf(int currcons)
496 {
497     if (y+1<bottom) {
498         ypos += video_size_row;
500         return;
! 如果光标小于屏幕所能显示的行数数，则直接让 y 自增一，即跳到下面一行。
! 并且同时修改其所在的显示内存值。
501     } else
502         scrup(currcons,top,bottom);
! 不然，对应的虚拟控制台显示的内容向上滚动一行
503     need_wrap = 0;
! 取消限制

```

504 }505

! 送光标到第 0 列  
 × currcons=虚拟终端号

517 static inline void cr(int currcons)518 {

519     pos -= x<<1;  
     ! 把pos内存值设置为第 0 列所在的位置  
520     need\_wrap = x = 0;  
     ! 取消打印限制

521 }

! 向屏幕上写一个字符  
 × b=要写的字符

1344 void console\_print(const char \* b)1345 {1346     int currcons = fg\_console;1347     unsigned char c;13481349 if (!printable || currcons<0 || currcons>=NR\_CONSOLES)1350       return;

! 如果当前不可打印，或者虚拟控制台号小于 0，

! 或者大于系统支持的最大控制台数则返回

1351       while ((c = \*(b++)) != 0) {

! 要写的字符不为 0

1352       if (c == 10 || c == 13 || need\_wrap) {

! 如果要打印的字符是换行或者回车或者有所限制

1353           if (c != 13)

! 如果字符不是回车，即换行

1354           lf(currcons);

! 则把对应虚拟控制台的光标下移一行，定义于本文件

1355           cr(currcons);

送光标到第 0 列，定义于本文件

1356       if (c == 10 || c == 13)

continue;

! 如果字符是换行或者回车，则跳过下面代码，即做了换行

1358       }1359       \*(unsigned short \*) pos = (attr << 8) + c;

! 把要输出的字符加上属性后，放入显存中（即显示在屏幕上）

```

1360         if (x == video_num_columns - 1) {
    ! 如果x坐标值（即所在列号）大于屏幕所能显示的最大值
1361             need_wrap = 1;
    ! 则限制屏幕打印
1362             continue;
    ! 跳过下面的代码
1363         }
1364         x++;
1365         pos+=2;
    ! x 坐标增一
    ! 对应的内存位置加 2 (因为一个字符占两个字节)
1366     }
1367     set_cursor(currcons);
    ! 设置虚拟控制台的光标位置，定义于本文件
1368     if (vt_cons[fg_console].vc_mode == KD_GRAPHICS)
1369         return;
    ! 如果当前控制台显示模式是图形方式，则退出！
1370     timer_active &= ~(1<<BLANK_TIMER);
    ! 清零对应的控制台屏幕时器
1371     if (console_blanked) {
        ! 如果控制台已经断开
        timer_table[BLANK_TIMER].expires = 0;
1373     timer_active |= 1<<BLANK_TIMER;
        ! 设置定时器终止值为 0，把对应的控制台号激活位置位
1374     } else if (blankinterval) {
        timer_table[BLANK_TIMER].expires = jiffies + blankinterval;
1376     timer_active |= 1<<BLANK_TIMER;
        ! 设置定时器终止值，把对应的控制台号激活位置位
1377     }
1378 }
1379
1380 */

```

```

1381 * long con_init(long);
1382 *
1383 * This routine initializes console interrupts, and does nothing
1384 * else. If you want the screen to clear, call tty_write with
1385 * the appropriate escape-sequence.
1386 *
1387 * Reads the information preserved by setup.s to determine the current display
1388 * type and sets everything accordingly.
1389 */

```

```

! long con_init(long)
! 该程序初始化控制台中断，不做其他事情。如果你想屏幕干净，就使用适当的
! 转义字符序列来调用 tty_write
! 读取setup.s程序保存的信息，用于确定当前显示器的类型，并且设置所有相关参数

1390 long con_init(long kmem_start)
1391 {
1392     char *display_desc = "????";
1393     int currcons = 0;
1394     long base;
1395     int orig_x = ORIG_X;
1396     int orig_y = ORIG_Y;
1397
1398     vc_scrmembuf = (unsigned short *) kmem_start;
        ! vc_scrmembuf指向可用内存起始处
1399     video_num_columns = ORIG_VIDEO_COLS;
        ! 显示器显示列数
1400     video_size_row = video_num_columns * 2;
        ! 每行使用字节数
1401     video_num_lines = ORIG_VIDEO_LINES;
        ! 显示器可显示字符行数
1402     video_page = ORIG_VIDEO_PAGE;
        ! 当前显示页数
1403     screen_size = (video_num_lines * video_size_row);
        ! 满屏占用字节数
1404     kmem_start += NR_CONSOLES * screen_size;
        ! 给NR_CONSOLES个控制台留出内存
1405     timer_table[BLANK_TIMER].fn = blank_screen;
        ! 设置控制台屏幕保存定时处理程序
1406     timer_table[BLANK_TIMER].expires = 0;
1407     if (blankinterval) {
            ! 设置终止值
1408         timer_table[BLANK_TIMER].expires = jiffies+blankinterval;
1409         timer_active |= 1<<BLANK_TIMER;
1410     }
1411
1412     if (ORIG_VIDEO_MODE == 7)      /* Is this a monochrome display? */
1413     {
            ! 单色显示器
1414         video_mem_base = 0xb0000;
            ! 单显内存起址
1415         video_port_reg = 0x3b4;
            ! 单显索引寄存器端口
1416         video_port_val = 0x3b5;
            ! 单显数据寄存器端口

```

```

1417         if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
1418         {
1419             video_type = VIDEO_TYPE_EGAM;
! 设置显示类型 (EGA单色)
1420             video_mem_term = 0xb8000;
! 显示内存末端地址
1421             display_desc = "EGA+";
! 将会在屏幕显示的字符串
1422         }
1423     else
1424     {
1425         video_type = VIDEO_TYPE_MDA;
! 设置显示类型 (MDA单色)
1426         video_mem_term = 0xb2000;
! 显示内存末端地址
1427         display_desc = "*MDA";
! 将会在屏幕显示的字符串
1428     }
1429     }
1430   else /* If not, it is color. */
1431   {
1432     can_do_color = 1;
! 可显示彩色
1433     video_mem_base = 0xb8000;
! 单显内存起址
1434     video_port_reg = 0x3d4;
! 单显索引寄存器端口
1435     video_port_val = 0x3d5;
! 单显数据寄存器端口
1436     if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
1437     {
1438         video_type = VIDEO_TYPE_EGAC;
! 设置显示类型 (EGA单色)
1439         video_mem_term = 0xc0000;
! 显示内存末端地址
1440         display_desc = "EGA+";
! 将会在屏幕显示的字符串
1441     }
1442     else
1443     {
1444         video_type = VIDEO_TYPE_CGA;
! 设置显示类型 (CGA单色)
1445         video_mem_term = 0xba000;
! 显示内存末端地址

```

```

1446         display_desc = "*CGA";
! 将会在屏幕显示的字符串
1447     }
1448 }
1449
1450 /* Initialize the variables used for scrolling (mostly EGA/VGA) */
1451
! 初始化滚屏用的的值
1452 base = (long)vc_scrmembuf;
1453 for (currcons = 0; currcons<NR_CONSOLES; currcons++) {
! 循环设置控制台及滚屏结束处
1454     pos = origin = video_mem_start = base;
1455     scr_end = video_mem_end = (base += screen_size);
1456     vc_scrbuf[currcons] = (unsigned short *) origin;
1457     vcmode      = KD_TEXT;
1458     vtmode.mode   = VT_AUTO;
1459     vtmode.waitv   = 0;
1460     vtmode.relsig   = 0;
1461     vtmode.acqsig   = 0;
1462     vtmode.frsig   = 0;
1463     vtpid        = -1;
1464     vtnewvt       = -1;
1465     clr_kbd(kbdraw);
1466     def_color      = 0x07; /* white */
1467     ulcolor        = 0x0f; /* bold white */
1468     halfcolor      = 0x08; /* grey */
1469     reset_terminal(currcons, currcons);

! 设置对应的虚拟终端
1470 }
1471 currcons = fg_console = 0;
1472
1473 video_mem_start = video_mem_base;
1474 video_mem_end = video_mem_term;
1475 origin = video_mem_start;
1476 scr_end = video_mem_start + video_num_lines * video_size_row;
! 滚屏结束地址
1477 gotoxy(currcons,0,0);
! 初始化光标位置, 左上角 (0, 0) 处定义于本文件中
1478 save_cur(currcons);
1479 gotoxy(currcons,orig_x,orig_y);
! 定义于本文件
1480 update_screen(fg_console);
! 更新对应的控制台屏幕, 定义于本文件
1481 printable = 1;

```

```

1482     printk("Console: %s %s %ldx%ld, %d virtual consoles\n",
1483             can_do_color?"colour":"mono",
1484             display_desc,
1485             video_num_columns,video_num_lines,
1486             NR_CONSOLES);
    ! 打印控制台的配置情况, 定义于Kernel/printk.c
1487     register_console(console_print);
    ! 注册屏幕打印函数
    ! register_console 定义于 Kernel/printk.c
    ! console_print 定义于Drivers/char/console.c
1488     return kmem_start;
    ! 返回可用内存的开始 (注意: ! 可用内存已经被虚拟控制台
    ! 占用了一些大小是NR_CONSOLES * screen_size
1489 }

```

! 得到对应于 currcons 控制台的滚屏位置  
 × currcons=当前虚拟控制台号

```

1499 static void get_scremem(int currcons)
1500 {
1501     memcpy((void *)vc_scrbuf[currcons],(void *)origin,screen_size);
    ! 从origin开始拷贝screen_size个字节到vc_scrbuf[currcons]中
1502     video_mem_start = (unsigned long)vc_scrbuf[currcons];
1503     origin = video_mem_start;
    ! 重新设置显示内存起始位置
1504     scr_end = video_mem_end = video_mem_start+screen_size;
    ! 设置滚屏结束位置
1505     pos = origin + y*video_size_row + (x<<1);
    ! 重新在显存中位置pos
1506 }
1507

```

! 设置对应控制台的屏幕内存  
 × currcons=当前虚拟控制台号

```

1508 static void set_scremem(int currcons)
1509 {
1510 #ifdef CONFIG_HGA
1511     /* This works with XFree86 1.2, 1.3 and 2.0
1512 This code could be extended and made more generally useful if we could
1513 determine the actual video mode. It appears that this should be
1514 possible on a genuine Hercules card, but I (WM) haven't been able to
1515 read from any of the required registers on my clone card.

```

```

1516 */
    ! 这个工作是为 Xfree861.2, 1.3, 2.0 做的
    ! 假如我们能够确定真正的显卡模式的话这块代码就能够被扩展并且用于大
    ! 多数的情况。看起来我们要一个真正的 Hercules 卡，但是我不可能读我的卡
    ! 上的所有需要的寄存器。
1517 /* This code should work with Hercules and MDA cards. */
    ! 该代码只能用于Hercules卡和单色显卡
1518 if (video_type == VIDEO_TYPE_MDA)
1519 {
    ! 如果显卡类型是单色的
1520     if (vcmode == KD_TEXT)
1521     {
        ! 显示模式为文本方式
1522     /* Ensure that the card is in text mode. */
        ! 确保该卡为文本显示方式
1523         int i;
1524         static char herc_txt_tbl[12] = {
1525             0x61,0x50,0x52,0x0f,0x19,6,0x19,0x19,2,0x0d,0x0b,0x0c };
1526         outb_p(0, 0x3bf); /* Back to power-on defaults */
1527         outb_p(0, 0x3b8); /* Blank the screen, select page 0, etc */
1528         for (i = 0 ; i < 12 ; i++)
1529         {
1530             outb_p(i, 0x3b4);
1531             outb_p(herc_txt_tbl[i], 0x3b5);
1532         }
1533     }
1534 #define HGA_BLINKER_ON 0x20
1535 #define HGA_SCREEN_ON 8
1536     /* Make sure that the hardware is not blanked */
1537     outb_p(HGA_BLINKER_ON | HGA_SCREEN_ON, 0x3b8);
1538 }
    ! 以上代码写M D A 的寄存器
1539#endif CONFIG_HGA
1540
1541     video_mem_start = video_mem_base;
1542     video_mem_end = video_mem_term;
    ! 分别对应显卡基址及显卡地址结尾
1543     origin = video_mem_start;
1544     scr_end = video_mem_start + screen_size;
    ! 滚屏值
1545     pos = origin + y*video_size_row + (x<<1);
    ! pos=内存位置
1546     memcpy((void *)video_mem_base, (void *)vc_scrbuf[fg_console], screen_size);
    ! 复制一屏内容到显存开始处

```

1547 }1548

```

    ! 更新控制台的显示
    × new_console=控制台号

1575 void update_screen(int new_console)
1576 {
1577     static int lock = 0;
1578
1579     if (new_console == fg_console || lock)
1580         return;
    ! 如果是当前控制台或者lock不为 0，则直接退出
1581     lock = 1;
1582     kbdsave(new_console);
    ! 空函数
1583     get_scrmem(fg_console);
    ! 得到对应于currcons控制台的滚屏位置，定义于本文件
1584     fg_console = new_console;
    ! 重新设置当前的控制台
1585     set_scrmem(fg_console);
    ! 根据get_scrmem中设置来设置新的控制台，定义于本文件中
1586     set_origin(fg_console);
    ! 定义于本文件中
1587     set_cursor(new_console);
    ! 设置光标，定义于本文件中
1588     set_leds();
    ! 激活键盘的中断
1589     compute_shiftstate();
    ! 计算shift姿态，定义于本文件
1590     lock = 0;
    ! 改回原值 0
1591 }
```

## Drivers/char/serial.c (部分代码)

```

616 /*
617 * This subroutine is called when the RS_TIMER goes off. It is used
618 * by the serial driver to run the rs_interrupt routine at certain
```

---

619 \* intervals, either because a serial interrupt might have been lost,  
620 \* or because (in the case of IRQ=0) the serial port does not have an  
621 \* interrupt, and is being checked only via the timer interrupts.  
622 \*/

! 当 RS\_TIME 结束后将会调用该子程序。在 rs\_interrupt 程序间隔会被  
 ! 串行驱动调用。可能一个串行中断可能会丢失，也可能串行端口没有  
 ! 中断，并且它仅在时间中断中被检查。

```

623 static void rs_timer(void)
624 {
625     int i, mask;
626     int timeout = 0;
627
628     for (i = 0, mask = 1; mask <= IRQ_active; i++, mask <<= 1) {
629         if ((mask & IRQ_active) && (IRQ_timer[i] <= jiffies)) {
630             IRQ_active &= ~mask;
631             cli();
632 #ifdef SERIAL_DEBUG_TIMER
633             printk("rs_timer: rs_interrupt(%d)...", i);
634 #endif
635             rs_interrupt(i);
636             sti();
637         }
638         if (mask & IRQ_active) {
639             if (!timeout || (IRQ_timer[i] < timeout))
640                 timeout = IRQ_timer[i];
641         }
642     }
643     if (timeout) {
644         timer_table[RS_TIMER].expires = timeout;
645         timer_active |= 1 << RS_TIMER;
646     }
647 }

```

1776

1777 /\*

1778 \* This routine prints out the appropriate serial driver version  
1779 \* number, and identifies which options were configured into this  
1780 \* driver.  
1781 \*/

! 该例程打印正确的串口驱动版本号，并且识别该驱动在配置时  
 ! 的选项。

1782 static void show\_serial\_version(void)

```

1783 {
1784     printk("Serial driver version 3.99a with");
        ! 打印提示信息, 定义于Kernel/printk.c
1785 #ifdef CONFIG_AST_FOURPORT
1786     printk(" AST_FOURPORT");
        ! 打印提示信息, 定义于Kernel/printk.c
1787 #define SERIAL_OPT
1788 #endif
1789 #ifdef CONFIG_ACCEENT_ASYNC
1790     printk(" ACCEENT_ASYNC");
        ! 打印提示信息, 定义于Kernel/printk.c
1791 #define SERIAL_OPT
1792 #endif
1793 #ifdef CONFIG_HUB6
1794     printk(" HUB-6");
        ! 打印提示信息, 定义于Kernel/printk.c
1795 #define SERIAL_OPT
1796 #endif
1797 #ifdef CONFIG_AUTO_IRQ
1798     printk (" AUTO_IRQ");
        ! 打印提示信息, 定义于Kernel/printk.c
1799 #define SERIAL_OPT
1800 #endif
1801 #ifdef SERIAL_OPT
1802     printk(" enabled\n");
        ! 打印提示信息, 定义于Kernel/printk.c
1803 #else
1804     printk(" no serial options enabled\n");
        ! 打印提示信息, 定义于Kernel/printk.c
1805 #endif
1806 #undef SERIAL_OPT
1807 }
1808

1900
1901 /*
1902 * This routine is called by rs_init() to initialize a specific serial
1903 * port. It determines what type of UART ship this serial port is
1904 * using: 8250, 16450, 16550, 16550A. The important question is
1905 * whether or not this UART is a 16550A or not, since this will
1906 * determine whether or not we can use its FIFO features or not.
1907 */

```

！该置程序在 rs\_init() 中被调用来初始化一个特殊的串口。它决定  
 ! UART 的串口类型是什么：8250, 16450, 16550, 16550A。这个  
 ! 重要的问题是 UART 是否是 16550A，因为它将决定是否可用 FIFO 特征。  
 ✗ info=结构指针

```

1908 static void autoconfig(struct async_struct * info)
1909 {
1910     unsigned char status1, status2, scratch, scratch2;
1911     unsigned port = info->port;
1912     unsigned long flags;
1913
1914     info->type = PORT_UNKNOWN;
1915
1916     if (!port)
1917         ! 如果端口为 0，则直接返回。
1918         return;
1919     save_flags(flags); cli();
1920         ! 保存标志寄存器值及关闭终端
1921
1922     /* * Do a simple existence test first; if we fail this, there's
1923     * no point trying anything else.
1924 */
1925         ! 首先做一个简单的存在测试；假如我们失败了，就没有必要在测试了
1926     scratch = serial_inp(info, UART_IER);
1927     serial_outp(info, UART_IER, 0);
1928     scratch2 = serial_inp(info, UART_IER);
1929     serial_outp(info, UART_IER, scratch);
1930     if (scratch2) {
1931         restore_flags(flags);
1932         return;      /* We failed; there's nothing here */
1933     }
1934
1935     /* * Check to see if a UART is really there. Certain broken
1936     * internal modems based on the Rockwell chipset fail this
1937     * test, because they apparently don't implement the loopback
1938     * test mode. So this test is skipped on the COM 1 through
1939     * COM 4 ports. This *should* be safe, since no board
1940     * manufacturer would be stupid enough to design a board
1941     * that conflicts with COM 1-4 --- we hope!
1942 */

```

！ 检查这里是否有UART。某些内部调制解调器的错误测试基于Rockwell芯片。

！因为它们显然不知道如何实现 loopback（回绕）测试模式。因此该测试跳过了  
！COM1 到 COM4 的端口。这个是“安全”的。因为没有那家主板厂商会做出  
！如此愚蠢的设计，也就是让 COM1 到 COM4 有冲突—我们希望。

```

1943 if (!(info->flags & ASYNC_SKIP_TEST)) {
1944     scratch = serial_inp(info, UART_MCR);
1945     serial_outp(info, UART_MCR, UART_MCR_LOOP | scratch);
1946     scratch2 = serial_inp(info, UART_MSR);
1947     serial_outp(info, UART_MCR, UART_MCR_LOOP | 0x0A);
1948     status1 = serial_inp(info, UART_MSR) & 0xF0;
1949     serial_outp(info, UART_MCR, scratch);
1950     serial_outp(info, UART_MSR, scratch2);
1951     if (status1 != 0x90) {
1952         restore_flags(flags);
1953         return;
1954     }
1955 }
1956
1957 /*
1958 * If the AUTO_IRQ flag is set, try to do the automatic IRQ
1959 * detection.
1960 */

```

！如果这个AUTO\_IRQ标志被置位，则试着让其自动侦测IRQ

```

1961 if (info->flags & ASYNC_AUTO_IRQ)
1962     info->irq = do_auto_irq(info);
1963
1964 serial_outp(info, UART_FCR, UART_FCR_ENABLE_FIFO);
1965 scratch = serial_in(info, UART_IIR) >> 6;
1966 info->xmit_fifo_size = 1;
1967 switch (scratch) {
1968     case 0:
1969         info->type = PORT_16450;
1970         break;
1971     case 1:
1972         info->type = PORT_UNKNOWN;
1973         break;
1974     case 2:
1975         info->type = PORT_16550;
1976         break;
1977     case 3:
1978         info->type = PORT_16550A;
1979         info->xmit_fifo_size = 16;
1980         break;
1981 }
1982 if (info->type == PORT_16450) {

```

```

1983     scratch = serial_in(info, UART_SCR);
1984     serial_outp(info, UART_SCR, 0xa5);
1985     status1 = serial_in(info, UART_SCR);
1986     serial_outp(info, UART_SCR, 0x5a);
1987     status2 = serial_in(info, UART_SCR);
1988     serial_outp(info, UART_SCR, scratch);
1989
1990     if ((status1 != 0xa5) || (status2 != 0x5a))
1991         info->type = PORT_8250;
1992     }
1993
1994     /*
1995 * Reset the UART.
1996 */
1997     ! 复位UART
1998     serial_outp(info, UART_MCR, 0x00);
1999     serial_outp(info, UART_FCR, (UART_FCR_CLEAR_RCVR |
2000                                     UART_FCR_CLEAR_XMIT));
2001     (void)serial_in(info, UART_RX);
2002     restore_flags(flags);
2003 }
2004

```

**2006 \* The serial driver boot-time initialization code!**

**2007 \*/**

！串行驱动启动时初始化代码  
× kmem\_start=可用内存的开始值

```

2008 long rs_init(long kmem_start)
2009 {
2010     int i;
2011     struct async_struct * info;
2012
2013     memset(&rs_event, 0, sizeof(rs_event));
2014     ! rs_event清零
2015     bh_base[SERIAL_BH].routine = do_softint;
2016     ! 初始化对应bottom half例程
2017     timer_table[RS_TIMER].fn = rs_timer;
2018     ! 定义处理例程
2019     timer_table[RS_TIMER].expires = 0;
2020     IRQ_active = 0;
2021 #ifdef CONFIG_AUTO_IRQ
2022     ! 忽略自动决定 I R Q 号

```

```

2019      rs_wild_int_mask = check_wild_interrupts(1);
2020 #endif
2021
2022      for (i = 0; i < 16; i++) {
2023          IRQ_ports[i] = 0;
2024          IRQ_timeout[i] = 0;
2025          ! 清零, IRQ_timeout是中断的超时大小
2026      }
2027      show_serial_version();
2028      ! 打印正确的串口驱动程序版本号, 定义于本文件
2029      for (i = 0, info = rs_table; i < NR_PORTS; i++, info++) {
2030          ! 填充结构信息
2031          info->line = i;
2032          info->tty = 0;
2033          info->type = PORT_UNKNOWN;
2034          info->custom_divisor = 0;
2035          info->close_delay = 50;
2036          info->x_char = 0;
2037          info->event = 0;
2038          info->count = 0;
2039          info->blocked_open = 0;
2040          memset(&info->callout_termios, 0, sizeof(struct termios));
2041          memset(&info->normal_termios, 0, sizeof(struct termios));
2042          info->open_wait = 0;
2043          info->xmit_wait = 0;
2044          info->close_wait = 0;
2045          info->next_port = 0;
2046          info->prev_port = 0;
2047          if (info->irq == 2)
2048              info->irq = 9;
2049          if (!(info->flags & ASYNC_BOOT_AUTOCONF))
2050              continue;
2051          autoconfig(info);
2052          ! 初始化串行端口, 定义于本文件
2053          if (info->type == PORT_UNKNOWN)
2054              continue;
2055          printk("tty%02d% at 0x%04x (irq = %d)", info->line,
2056                  (info->flags & ASYNC_FOURPORT) ? "FourPort" : "",
2057                  info->port, info->irq);
2058          switch (info->type) {
2059              case PORT_8250:
2060                  printk(" is a 8250\n");
2061                  break;

```

```

2059         case PORT_16450:
2060             printk(" is a 16450\n");
2061             break;
2062         case PORT_16550:
2063             printk(" is a 16550\n");
2064             break;
2065         case PORT_16550A:
2066             printk(" is a 16550A\n");
2067             break;
2068     default:
2069         printk("\n");
2070         break;
    ! 打印提示信息
2071     }
2072 }
2073 return kmem_start;
    ! 返回可用内存起始值
2074 }
2075
2076

```

## Drivers/char/keyboard.c (部分代码)

```

708 /* called after returning from RAW mode or when changing consoles -
709 recompute k_down[] and shift_state from key_down[] */
    ! 在RAW模式或者改变控制台后调用，重新计算k_dwon和shift_start
710 void compute_shiftstate(void)
711 {
712     int i, j, k, sym, val;
713
714     shift_state = 0;
715     for(i=0; i < SIZE(k_down); i++)
716         k_down[i] = 0;
    ! k_down[]清零
717
718     for(i=0; i < SIZE(key_down); i++)
719         if(key_down[i]) { /* skip this word if not a single bit on */
    ! 测试信号位是否置位
720             k = (i<<5);
    ! 右移 5 位
721             for(j=0; j<32; j++,k++)

```

```

722         if(test_bit(k, key_down)) {
    ! 测试第k位是否为 1
723             sym = key_map[0][k];
724             if(KTYP(sym) == KT_SHIFT) {
725                 val = KVAL(sym);
726                 k_down[val]++;
727                 shift_state |= (1<<val);
728             }
729         }
730     }
731 }
```

! 键盘初始化  
 ✕ kmem\_start=可用内存起始值

```

875 unsigned long kbd_init(unsigned long kmem_start)
876 {
877     int i;
878     struct kbd_struct * kbd;
879
880     kbd = kbd_table + 0;
    ! 让kbd指向全局的kdb_table
881     for (i = 0 ; i < NR_CONSOLES ; i++,kbd++) {
882         kbd->ledstate = KBD_DEFLEDS;
        ! 关闭Num Lock
883         kbd->default_ledstate = KBD_DEFLEDS;
        ! 关闭Num Lock
884         kbd->lockstate = KBD_DEFLOCK;
        ! shift锁定模式
885         kbd->modeflags = KBD_DEFMODE;
        ! 应用程序键模式
886     }
887
888     bh_base[KEYBOARD_BH].routine = kbd_bh;
    ! 初始化bottom half
889     request_irq(KEYBOARD_IRQ,keyboard_interrupt);
    ! 注册键盘中断处理程序并开启键盘中断, 定义于Kernel/irq.c
890     mark_bh(KEYBOARD_BH);
    ! 标记键盘的bottom half
891     return kmem_start;
    ! 返回可用内存值
892 }
```

## Drivers/char/tty\_io.c (部分代码)

```

! 设置 tty 终端的处理例程
× disc=tty 编号
× new_ldisc=该结构中包括了 tty 终端处理例程
90 int tty_register_ldisc(int disc, struct tty_ldisc *new_ldisc)
91 {
92     if (disc < N_TTY || disc >= NR_LDISCS)
93         return -EINVAL;
94
95     if (new_ldisc) {
! 如果, 传入的结构不为空, 则设置之
96         ldiscs[disc] = *new_ldisc;
97         ldiscs[disc].flags |= LDISC_FLAG_DEFINED;
98     } else
99         memset(&ldiscs[disc], 0, sizeof(struct tty_ldisc));
! 否则, 清零之。
100
101     return 0;
102 }

```

```

! tty 终端初始化
× kmem_start=可用内存的起始值
1820 long tty_init(long kmem_start)
1821 {
1822     int i;
1823
1824     if (sizeof(struct tty_struct) > PAGE_SIZE)
1825         panic("size of tty structure > PAGE_SIZE!");
! 检查tty_struct结构大小, 大于页尺寸则死机。
1826     if (register_chrdev(TTY_MAJOR, "tty", &tty_fops))
! 注册TTY, 定义于fs/devices.c
1827         panic("unable to get major %d for tty device", TTY_MAJOR);
! 失败则打印错误信息后, 死机。
1828     if (register_chrdev(TTYAUX_MAJOR, "tty", &tty_fops))
1829         panic("unable to get major %d for tty device", TTYAUX_MAJOR);
! 注册TTYAUX, 失败则打印错误信息后, 死机。定义于fs/devices.c
1830     for (i=0 ; i< MAX_TTYS ; i++) {
1831         tty_table[i] = 0;

```

```

1832         tty_termios[i] = 0;
    ! 定义tty表和对应的终端io属性及控制字符数据结构
1833     }
1834     memset(tty_check_write, 0, sizeof(tty_check_write));
    ! 和标准C库功能相同所以不在注释
1835     bh_base[TTY_BH].routine = tty_bh_routine;
    ! 初始化tty对应的bottom half中断处理例程
1836
1837     /* Setup the default TTY line discipline. */
    ! 设置缺省的TTY线路处理程序
1838     memset(ldiscs, 0, sizeof(ldiscs));
    ! 和标准C库功能相同所以不在注释
1839     (void) tty_register_ldisc(N_TTY, &tty_ldisc_N_TTY);
    ! 设置tty终端的处理例程, 定义于本文件
1840
1841     kmem_start = kbd_init(kmem_start);
    ! 键盘初始化, 定义于Drivers/char/keyboard.c
1842     kmem_start = con_init(kmem_start);
    ! 初始化控制台, 定义于Drivers/char/console.c
1843     kmem_start = rs_init(kmem_start);
    ! 串口初始化, 定义于 Drivers/char/serial.c
    ! (对UART的注释并不完全, 先放着)
1844     return kmem_start;
    ! 返回可用内存值
1845 }

```

## Drivers/char/mem.c (部分代码)

```

    ! 注册字符设备
    × mem_start=可用内存起始值
    × mem_end=可用内存的结束值
396 long chr_dev_init(long mem_start, long mem_end)
397 {
398     if (register_chrdev(MEM_MAJOR,"mem",&memory_fops))
        ! 注册字符设备, 定义于Fs/devices.c。
399     printk("unable to get major %d for memory devs\n", MEM_MAJOR);
        ! 如果失败的话, 打印错误提示信息。
400     mem_start = tty_init(mem_start);
        ! tty终端初始化, 定义于Drivers/char/tty_io.c

```

```

401 #ifdef CONFIG_PRINTER          ! 忽略打印机的配置
402     mem_start = lp_init(mem_start);
403 #endif
404 #if defined(CONFIG_BUSMOUSE) || defined(CONFIG_82C710_MOUSE) || \
405     defined(CONFIG_PSMOUSE) || defined(CONFIG_MS_BUSMOUSE) || \
406     defined(CONFIG_ATIXL_BUSMOUSE)
407     mem_start = mouse_init(mem_start); ! 忽略鼠标的配置
408 #endif
409 #ifdef CONFIG_SOUND              ! 忽略声卡的配置
410     mem_start = soundcard_init(mem_start);
411 #endif
412 #if CONFIG_TAPE_QIC02           ! 忽略磁带机的配置
413     mem_start = tape_qic02_init(mem_start);
414 #endif
415 /*
416 * Rude way to allocate kernel memory buffer for tape device
417 */
418 #ifdef CONFIG_FTAPE              ! 忽略磁带机的配置
419     /* allocate NR_FTAPE_BUFFERS 32Kb buffers at aligned address */
420     ftape_big_buffer= (char*) ((mem_start + 0x7fff) & ~0x7fff);
421     printk( "ftape: allocated %d buffers alligned at: %p\n",
422             NR_FTAPE_BUFFERS, ftape_big_buffer);
423     mem_start = (long) ftape_big_buffer + NR_FTAPE_BUFFERS * 0x8000;
424 #endif
425     return mem_start;
426 }

```

## Drivers/block/floppy.c (部分代码)

```

164 static struct floppy_struct floppy_types[] = {
165     { 720, 9, 2, 40, 0, 0x2A, 0x02, 0xDF, 0x50, "360k/PC" }, /* 360kB PC diskettes */
166     { 720, 9, 2, 40, 0, 0x2A, 0x02, 0xDF, 0x50, "360k/PC" }, /* 360kB PC diskettes */
167     { 2400, 15, 2, 80, 0, 0x1B, 0x00, 0xDF, 0x54, "1.2M" }, /* 1.2 MB AT-diskettes */
168     { 720, 9, 2, 40, 1, 0x23, 0x01, 0xDF, 0x50, "360k/AT" }, /* 360kB in 1.2MB drive */
169     { 1440, 9, 2, 80, 0, 0x2A, 0x02, 0xDF, 0x50, "720k" }, /* 3.5" 720kB diskette */
170     { 1440, 9, 2, 80, 0, 0x2A, 0x02, 0xDF, 0x50, "720k" }, /* 3.5" 720kB diskette */
171     { 2880, 18, 2, 80, 0, 0x1B, 0x00, 0xCF, 0x6C, "1.44M" }, /* 1.44MB diskette */
172     { 1440, 9, 2, 80, 0, 0x2A, 0x02, 0xDF, 0x50, "720k/AT" }, /* 3.5" 720kB diskette */

```

[173](#) };

[471](#)

```

    ! 得到 FDC 版本
    ✘ byte=得到版本代码
472 static void output_byte(char byte)
473 {
474     int counter;
475     unsigned char status;
476
477     if (reset)
478         return;
479     for(counter = 0 ; counter < 10000 ; counter++) {
480         status = inb_p(FD_STATUS) & (STATUS_READY | STATUS_DIR);
    ! 循环 10000 次查询软驱的状态，直到返回值是STATUS_READY，再向指定
    ! 端口写入byte
481         if (status == STATUS_READY) {
482             outb(byte,FD_DATA);
483             return;
    ! 正确返回
484         }
485     }
486     current_track = NO_TRACK;
487     reset = 1;
488     printf("Unable to send byte to FDC\n");
    ! 否则打印错误提示
489 }
```

[490](#)

```

491 static int result(void)
492 {
493     int i = 0, counter, status;
494
495     if (reset)
496         return -1;
497     for (counter = 0 ; counter < 10000 ; counter++) {
498         status = inb_p(FD_STATUS)&(STATUS_DIR|STATUS_READY|STATUS_BUSY);
499         if (status == STATUS_READY) {
500             return i;
    ! 循环 10000 次查询软驱的状态，直到返回值是STATUS_READY，便返回 0
501     }
```

```

502         if (status == (STATUS_DIR|STATUS_READY|STATUS_BUSY)) {
503             if (i >= MAX_REPLIES) {
504                 printk("floppy_stat reply overrun\n");
505                 break;
506             }
507             reply_buffer[i++] = inb_p(FD_DATA);
508         }
509     }
510     reset = 1;
511     current_track = NO_TRACK;
512     printk("Getstatus times out\n");
513     return -1;
514     ! -1 表示由超时
515

```

```

1230
    ! 该宏用于读取CMOS中的信息
1231 #define CMOS_READ(addr) ({ \
1232     outb_p(addr,0x70); \
        ! 0x70 是写端口
1233     inb_p(0x71); \
        ! 0x71 是读端口
1234 })
1235

```

```

    ! 根据 code 码从系统定义的 floppy_types 查询到结构
    × drive=驱动器号
    × code=查询码
1236 static struct floppy_struct *find_base(int drive,int code)
1237 {
1238     struct floppy_struct *base;
1239
1240     if (code > 0 && code < 5) {
1241         base = &floppy_types[(code-1)*2];
        ! floppy_types 定义于本文件
1242         printk("fd%od is %s",drive,base->name);
1243         return base;
    ! 根据code查询系统定义的floppy_types中定义的内容，并打印后返回
1244     }

```

```

1245     printk("fd%d is unknown type %d",drive,code);
1246     return NULL;
    ! 否则打印没有该类型磁盘的提示，并返回NULL
1247 }
1248

```

```

1249 static void config_types(void)
1250 {
1251     printk("Floppy drive(s): ");
    ! 打印提示消息
1252     base_type[0] = find_base(0,(CMOS_READ(0x10)>>4) & 15);
    ! CMOS中偏移地址 0x10 处保存的是磁盘驱动器的类型，定义于本文件
1253     if (((CMOS_READ(0x14)>>6) & 1) == 0)
    ! CMOS中偏移地址 0x14 处保存的是设备字节
1254     base_type[1] = NULL;
1255     else {
1256         printk(",");
1257         base_type[1] = find_base(1,CMOS_READ(0x10) & 15);
1258     }
1259     base_type[2] = base_type[3] = NULL;
    ! 我们可以看到 1.0 核心只最多支持两个软驱
1260     printk("\n");
1261 }
1262

```

```

    ! 软驱初始化
1348 void floppy_init(void)
1349 {
1350     outb(current_DOR,FD_DOR);
    ! 把current_DOR输出到FDC数字输出端口
1351     if (register_blkdev(MAJOR_NR,"fd",&floppy_fops)) {
1352         printk("Unable to get major %d for floppy\n",MAJOR_NR);
1353         return;
    ! 注册软驱，如果失败则打印错误提示后，退出！
    ! register_blkdev 定义于Fs/devices.c
1354     }
1355     blk_size[MAJOR_NR] = floppy_sizes;
1356     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
1357     timer_table[FLOPPY_TIMER].fn = floppy_shutdown;
    ! 设置floppy的处理例程
1358     timer_active &= ~(1 << FLOPPY_TIMER);
    ! 激活其定时器

```

```

1359     config_types();
    ! 根据CMOS操作结果配置软驱，定义于本文件
1360     if (irqaction(FLOPPY_IRQ,&floppy_sigaction))
    ! 定义于Kernel/irq.c
1361         printk("Unable to grab IRQ%d for the floppy driver\n", FLOPPY_IRQ);
    ! 注册对应的IRQ请求，出错则打印提示消息。
1362     if (request_dma(FLOPPY_DMA))
1363         printk("Unable to grab DMA%d for the floppy driver\n", FLOPPY_DMA);
    ! 看是否可以打开DMA通道不能则打印错误信息
1364     /* Try to determine the floppy controller type */
1365     DEVICE_INTR = ignore_interrupt; /* don't ask ... */
1366     output_byte(FD_VERSION);      /* get FDC version code */
    ! 得到FDC版本，定义于本文件
1367     if (result() != 1) {
1368         printk(DEVICE_NAME ": FDC failed to return version byte\n");
1369         fdc_version = FDC_TYPE_STD;
1370     } else
1371         fdc_version = reply_buffer[0];
1372     if (fdc_version != FDC_TYPE_STD)
1373         printk(DEVICE_NAME ": FDC version 0x%0x\n", fdc_version);
1374 #ifndef FDC_FIFO_UNTESTED
1375     fdc_version = FDC_TYPE_STD; /* force std fdc type; can't test other. */
1376 #endif
1377
1378     /* Not all FDCs seem to be able to handle the version command
1379 * properly, so force a reset for the standard FDC clones,
1380 * to avoid interrupt garbage.
1381 */
1382
1383     if (fdc_version == FDC_TYPE_STD) {
1384         initial_reset_flag = 1;
1385         reset_floppy();
    ! 如果fdc_version == FDC_TYPE_STD, 复位磁盘！
1386     }
1387 }

```

## Drivers/block/ramdisk.c (部分代码)

97 /\*

98 \* If the root device is the RAM disk, try to load it.

```

99 * In order to do this, the root device is originally set to the
100 * floppy, and we later change it to be RAM disk.
101 */
    ! 假如根文件系统设备是虚拟盘的话，则尝试加载它。Root_device 原先是指定软盘的
    ! 我们将它改成指向ramdisk

102 void rd_load(void)
103 {
104     struct buffer_head *bh;
105     struct minix_super_block s;
106     int      block, tries;
107     int      i = 1;
108     int      nblocks;
109     char     *cp;
110
111     /* If no RAM disk specified, give up early. */
    ! 如果ramdisk没有被指定，则早点放弃。
112     if (!rd_length) return;
113     printk("RAMDISK: %d bytes, starting at 0x%lx\n",
114           rd_length, (int) rd_start);
    ! 打印提示信息，定义于Kernel/printk.c

115
116     /* If we are doing a diskette boot, we might have to pre-load it. */
    ! 如果我们从软盘启动，我们不得不先加载它。
117     if (MAJOR(ROOT_DEV) != FLOPPY_MAJOR) return;
    ! 如果主设备号，不是软盘的话，则返回！

118
119     /*
120 * Check for a super block on the diskette.
121 * The old-style boot/root diskettes had their RAM image
122 * starting at block 512 of the boot diskette. LINUX/Pro
123 * uses the entire diskette as a file system, so in that
124 * case, we have to look at block 0. Be intelligent about
125 * this, and check both... - FvK
126 */
    ! 检查软盘上的超级块。
    ! 对于老的类型的 boot/root 盘它们 RAM image 从软盘的第 512 块开始存放。
    ! LINUX/pro 使用软盘的入口点作为文件系统，因此我们不得不看看对 0 块
    ! 请考虑这些，并且检查它... -FvK

127     for (tries = 0; tries < 1000; tries += 512) {
128         block = tries;
129         bh = breada(ROOT_DEV, block+1, block, block+2, -1);
    ! 两个情况：
    ! 1: 读软盘块的 1, 0, 2
    ! 2: 读软盘块的 513, 512, 514

```

- ! 为什么会有两种情况呢?
- ! 因为, 如果把根文件系统放在另一张软盘上的话, 就是第 1 种情况
- ! 根文件系统和核心放在一起的话, 就是第 2 种的情况
- ! 请注意block+1 是超级块, 具体的请看基础部分的文件系统解释

```

130     if (!bh) {
131         printk("RAMDISK: I/O error while looking for super block!\n");
132         return;
133     }
134
135     /* This is silly- why do we require it to be a MINIX FS? */
136     *((struct minix_super_block *) &s) =
137         *((struct minix_super_block *) bh->b_data);
    ! 让s指向缓冲区中的超级块
138     brelse(bh);
    ! 释放之, 定义于Fs/buffer.c
139     nblocks = s.s_nzones << s.s_log_zone_size;
    ! 块数=逻辑块数×2^(每区段块数的)
140     if (s.s_magic != MINIX_SUPER_MAGIC &&
141         s.s_magic != MINIX_SUPER_MAGIC2) {
142         printk("RAMDISK: trying old-style RAM image.\n");
143         continue;
144     }
    ! 如果超级块的幻数不为上述两种之一的话, 则打印提示消息, 跳过下面代码
145
146     if (nblocks > (rd_length >> BLOCK_SIZE_BITS)) {
147         printk("RAMDISK: image too big! (%d/%d blocks)\n",
148               nblocks, rd_length >> BLOCK_SIZE_BITS);
149         return;
150     }
    ! 如果数据块大于内存中虚拟盘所能容纳的块数, 则不能加载, 显示
    ! 错误提示并返回。
151     printk("RAMDISK: Loading %d blocks into RAM disk", nblocks);
    ! 打印提示消息, 定义于Kernel/printk.c
152
153     /* We found an image file system. Load it into core! */
154     cp = rd_start;
    ! cp指向虚拟盘起始处
155     while (nblocks) {
156         if (nblocks > 2)
157             bh = breada(ROOT_DEV, block, block+1, block+2, -1);
    ! 如果要读取的块数多于 3 块则采用超前预读方式读数据
158         else
159             bh = bread(ROOT_DEV, block, BLOCK_SIZE);
    ! 否则就单块读取

```

```

160         if (!bh) {
161             printk("RAMDISK: I/O error on block %d, aborting!\n",
162                   block);
163             return;
164         }
165         (void) memcpy(cp, bh->b_data, BLOCK_SIZE);
    ! 将缓冲区中的数据复制到cp处
166         brelse(bh);
    ! 拷贝完后，便释放该缓冲，定义于Fs/buffer.c
167         if (!(nblocks-- & 15)) printk(".");
168         cp += BLOCK_SIZE;
    ! 调整指针
169         block++;
170         i++;
171     }
172     printk("\ndone\n");
    ! 拷贝完成，打印提示信息。
173
174     /* We loaded the file system image. Prepare for mounting it.*/
    ! 我们已经加载了文件系统img，为安装它做准备。
175     ROOT_DEV = ((MEM_MAJOR << 8) | RAMDISK_MINOR);
    ! 修改ROOT_DEV使其指向虚拟盘ramdisk
176     return;
177 }
178 }
179

```

## Drivers/block/hd.c

```

636
    ! 硬盘的初始化
637 static void hd_geninit(void)
638 {
639     int drive, i;
640     extern struct drive_info drive_info;
641     unsigned char *BIOS = (unsigned char *) &drive_info;
642     int cmos_disks;
643

```

```

644     if (!NR HD) {
    ! 如果没有定义硬盘的类型，就从 PARAM+0x80 处开始读在 setup.s 中探测到
    ! 的关于硬盘的配置参数
645         for (drive=0 ; drive<2 ; drive++) {
646             hd_info[drive].cyl = *(unsigned short *) BIOS;
    ! 柱面数
647             hd_info[drive].head = *(2+BIOS);
    ! 磁头数
648             hd_info[drive].wpcom = *(unsigned short *) (5+BIOS);
    ! 写前预补偿柱面号
649             hd_info[drive].ctl = *(8+BIOS);
    ! 控制字节
650             hd_info[drive].lzone = *(unsigned short *) (12+BIOS);
    ! 磁头着陆区柱面号
651             hd_info[drive].sect = *(14+BIOS);
    ! 每磁道扇区数
652             BIOS += 16;
    ! 因为硬盘的参数表大小为 16 个字节，所以加 16 指向下一个表
653         }
654
655     /*
656 We query CMOS about hard disks : it could be that
657 we have a SCSI/ESDI/etc controller that is BIOS
658 compatable with ST-506, and thus showing up in our
659 BIOS table, but not register compatable, and therefore
660 not present in CMOS.
661
662 Furthermore, we will assume that our ST-506 drives
663 <if any> are the primary drives in the system, and
664 the ones reflected as drive 1 or 2.
665
666 The first drive is stored in the high nibble of CMOS
667 byte 0x12, the second in the low nibble. This will be
668 either a 4 bit drive type or 0xf indicating use byte 0x19
669 for an 8 bit type, drive 1, 0x1a for drive 2 in CMOS.
670
671 Needless to say, a non-zero value means we have
672 an AT controller hard disk for that drive.
    ! 我们对 CMOS 有关硬盘的信息有些怀疑：可能会有这样的情况，我们有一块
    ! SCSI/ESDI 的控制器，它是以 ST-506 方式于 BIOS 兼容的，因为会出现在我们
    ! 的 BIOS 参数表中，但又不是寄存器兼容的，因此这些参数在 CMOS 中不存在。
    ! 另外，我们假设 ST-506 驱动器（如果有的话）是系统中的基本驱动器，即以驱动器
    ! 1 或者 2 出现的驱动器。
    ! 第一个驱动器参数存在 CMOS 字节的 0x12 的高半个字节中，第二个放在低半个字节

```

！中。该4位字节信息可以是驱动器类型，也可能仅是0x0F。0x0F表示使用CMOS  
 ! 中0x19字节为驱动器1的8位类型字节，使用CMOS中的0x1A字节作为驱动器  
 ! 2的类型字节。  
 ! 总之，一个非零值意味着我们有一个AT控制器硬盘兼容的驱动器。

```

673
674
675 */
676
677     if ((cmos_disks = CMOS_READ(0x12)) & 0xf0)
678         if (cmos_disks & 0x0f)
679             NR_HD = 2;
680         else
681             NR_HD = 1;
682     }
683     i = NR_HD;
684     while (i-- > 0) {
685         hd[i<<6].nr_sects = 0;
686         if (hd_info[i].head > 16) {
687             printk("hd.c: ST-506 interface disk with more than 16 heads detected,\n");
688             printk(" probably due to non-standard sector translation. Giving up.\n");
689             printk("(disk %d: cyl=%d, sect=%d, head=%d)\n", i,
690                   hd_info[i].cyl,
691                   hd_info[i].sect,
692                   hd_info[i].head);
693             if (i+1 == NR_HD)
694                 NR_HD--;
695             continue;
696         }
697         hd[i<<6].nr_sects = hd_info[i].head*
698                         hd_info[i].sect*hd_info[i].cyl;
699     }
700     if (NR_HD) {
701         if (irqaction(HD_IRQ,&hd_sigaction)) {
702             printk("hd.c: unable to get IRQ%d for the harddisk driver\n", HD_IRQ);
703             NR_HD = 0;
704         }
705     }
706     hd_gendisk.nr_real = NR_HD;
707     ! 设置系统所拥有的真正硬盘计数
708     for(i=0;i<(MAX_HD << 6);i++) hd_blocksizes[i] = 1024;

```

---

```

709     blksize_size[MAJOR_NR] = hd_blocksizes;
710 }
```

## Drivers/block/genhd.c (部分代码)

```

25 /*
26 * Create devices for each logical partition in an extended partition.
27 * The logical partitions form a linked list, with each entry being
28 * a partition table with two entries. The first entry
29 * is the real data partition (with a start relative to the partition
30 * table start). The second is a pointer to the next logical partition
31 * (with a start relative to the entire extended partition).
32 * We do not create a Linux partition for the partition tables, but
33 * only for the actual data partitions.
34 */
35
36 static void extended_partition(struct gendisk *hd, int dev)
37 {
38     struct buffer_head *bh;
39     struct partition *p;
40     unsigned long first_sector, this_sector;
41     int mask = (1 << hd->minor_shift) - 1;
42
43     first_sector = hd->part[MINOR(dev)].start_sect;
44     this_sector = first_sector;
45
46     while (1) {
47         if ((current_minor & mask) >= (4 + hd->max_p))
48             return;
49         if (!(bh = bread(dev,0,1024)))
50             return;
51         /*
52 * This block is from a device that we're about to stomp on.
53 * So make sure nobody thinks this block is usable.
54 */
55         bh->b_dirt=0;
56         bh->b_uptodate=0;
57         if (*(unsigned short *) (bh->b_data+510) == 0xAA55) {
58             p = (struct partition *) (0x1BE + bh->b_data);
```

```

59         /*
60 * Process the first entry, which should be the real
61 * data partition.
62 */
63     if (p->sys_ind == EXTENDED_PARTITION ||
64         !(hd->part[current_minor].nr_sects = p->nr_sects))
65         goto done; /* shouldn't happen */
66     hd->part[current_minor].start_sect = this_sector + p->start_sect;
67     printk("%s%c%d", hd->major_name,
68           'a'+(current_minor >> hd->minor_shift),
69           mask & current_minor);
70     current_minor++;
71     p++;
72     /*
73 * Process the second entry, which should be a link
74 * to the next logical partition. Create a minor
75 * for this just long enough to get the next partition
76 * table. The minor will be reused for the real
77 * data partition.
78 */
79     if (p->sys_ind != EXTENDED_PARTITION ||
80         !(hd->part[current_minor].nr_sects = p->nr_sects))
81         goto done; /* no more logicals in this partition */
82     hd->part[current_minor].start_sect = first_sector + p->start_sect;
83     this_sector = first_sector + p->start_sect;
84     dev = ((hd->major) << 8) | current_minor;
85     brelse(bh);
86 } else
87     goto done;
88 }
89 done:
90 brelse(bh);
! 释放对应的缓冲区，定义于Fs/buffer.c
91 }
92

```

！校验分区的正确与否

× hd=硬盘结构指针

× dev=硬盘的编号

93 static void check\_partition(struct gendisk \*hd, unsigned int dev)

```

94 {
95     static int first_time = 1;
96     int i, minor = current_minor;

```

```

97     struct buffer_head *bh;
98     struct partition *p;
99     unsigned long first_sector;
100    int mask = (1 << hd->minor_shift) - 1;
101
102    if (first_time)
103        printk("Partition check:\n");
! 打印提示信息, 定义于Kernel/printk.c
104    first_time = 0;
105    first_sector = hd->part[MINOR(dev)].start_sect;
! 获取对应的硬盘起始扇区号
106    if (!(bh = bread(dev, 0, 1024))) {
107        printk(" unable to read partition table of device %04x\n", dev);
108        return;
! 读取硬盘上第一块数据, bread定义于Fs/buffer.c
109    }
110    printk("%s%c:", hd->major_name, 'a' + (minor >> hd->minor_shift));
! 打印提示信息
111    current_minor += 4; /* first "extra" minor */
112    if (*((unsigned short *) (bh->b_data + 510)) == 0xAA55) {
! 如果从第 510 个字节开始的两个字节是 0xAA55 的话,
113        p = (struct partition *) (0x1BE + bh->b_data);
! 让p指向分区表
114        for (i=1 ; i<=4 ; minor++, i++, p++) {
115            if (!(hd->part[minor].nr_sects = p->nr_sects))
116                continue;
! 检查分区表 (共 4 个分区), 如果该分区存在, 则跳过下面的代码
117            hd->part[minor].start_sect = first_sector + p->start_sect;
! 否则, 设置对于的起始扇区
118            printk("%s%c%d", hd->major_name, 'a' + (minor >> hd->minor_shift), i);
119            if ((current_minor & 0x3f) >= 60)
120                continue;
121            if (p->sys_ind == EXTENDED_PARTITION) {
122                printk("<");
123                extended_partition(hd, (hd->major << 8) | minor);
124                printk(">");
125            }
126        }
127    /*
128 * check for Disk Manager partition table
129 */
! 检查磁盘管理器分区表
130    if (*((unsigned short *) (bh->b_data + 0xfc)) == 0x55AA) {
131        p = (struct partition *) (0x1BE + bh->b_data);

```

```

132     for (i = 4 ; i < 16 ; i++, current_minor++) {
133         p--;
134         if ((current_minor & mask) >= mask-2)
135             break;
136         if (!(p->start_sect && p->nr_sects))
137             continue;
138         hd->part[current_minor].start_sect = p->start_sect;
139         hd->part[current_minor].nr_sects = p->nr_sects;
140         printk(" %s%c%d", hd->major_name,
141                 'a'+(current_minor >> hd->minor_shift),
142                 current_minor & mask);
143     }
144 }
145 } else
146     printk(" bad partition table");
147     printk("\\n");
148     brelse(bh);
149 }
```

! 设置对应的硬盘  
! dev=硬盘结构指针

```

174 static void setup_dev(struct gendisk *dev)
175 {
176     int i;
177     int j = dev->max_nr * dev->max_p;
        ! 计算最大的分区数
178     int major = dev->major << 8;
        ! 取得主设备号
179     int drive;
180
181
182     for (i = 0 ; i < j; i++) {
183         dev->part[i].start_sect = 0;
184         dev->part[i].nr_sects = 0;
        ! 按顺序初始化对应分区的开始扇区以及扇区数
185     }
186     dev->init();
        ! 调用对应的初始化函数，这些函数对应于特定的块设备
        ! 对于普通的 IDE 硬盘，是 hd_geninit，定义于 Drivers/block/hd.c
        ! 其他的有兴趣的读者可以自己看看
187     for (drive=0 ; drive<dev->nr_real ; drive++) {
188         current_minor = 1+(drive<<dev->minor_shift);
        ! 当前设备的子设备号

```

```

189      check_partition(dev, major+(drive<<dev->minor_shift));
    ! 检查分区
    ! major=主设备号 (计算方法: major = dev->major << 8, 对于 IDE 的硬盘,
    ! dev->major=3, 所以major=3*28=0x300)
    ! 第一块硬盘是 0x300
    ! 第二块硬盘是 0x340
    ! check_partition 定义于本文件

190      }
191      for (i=0 ; i < j ; i++)
192          dev->sizes[i] = dev->part[i].nr_sects >> (BLOCK_SIZE_BITS - 9);
193      blk_size[dev->major] = dev->sizes;
194  }
195
196 /* This may be used only once, enforced by 'static int callable' */
    ! 该函数仅仅只能执行一次, 通过静态的callable来强制之
197 asmlinkage int sys_setup(void * BIOS)
198 {
199     static int callable = 1;
200     struct gendisk *p;
201     int nr=0;
202
203     if (!callable)
204         return -1;
205     callable = 0;
    ! 让callable=0, 以达到只能被调用一次的目的。

206
207     for (p = gendisk_head ; p ; p=p->next) {
208         setup_dev(p);
209         nr += p->nr_real;
    ! gendisk_head 列表在 blk_dev_init 中被初始化成
    ! 循环初始化硬盘。定义于本文件

210     }
211
212     if (ramdisk_size)
213         rd_load();
    ! 如果定义了虚拟盘, 则加载之Drivers/block/ramdisk.c
214     mount_root();
    ! 安装根文件系统, 定义于Fs/super.c
215     return (0);
216 }
217

```

## Drivers/block/hd.c (部分代码)

```

! 硬盘的初始化
× mem_start=可用内存起始值
× mem_end=可用内存结束值

725 unsigned long hd_init(unsigned long mem_start, unsigned long mem_end)
726 {
    ! 定义于Fs/devices.c
727     if (register_blkdev(MAJOR_NR,"hd",&hd_fops)) {
        ! 如果注册失败，打印错误信息后，直接返回可用内存开始处值
728         printk("Unable to get major %d for harddisk\n",MAJOR_NR);
729         return mem_start;
730     }
731     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
    ! 填写请求函数
732     read_ahead[MAJOR_NR] = 8;           /* 8 sector (4kB) read-ahead */
    ! 块设备预读缓冲大小为 8k个字节
733     hd_gendisk.next = gendisk_head;
734     gendisk_head = &hd_gendisk;
735     timer_table[HD_TIMER].fn = hd_times_out;
    ! 设置错误处理例程
736     return mem_start;
    ! 返回可用内存开始值
737 }

```

## Drivers/block/xd.c (部分代码)

```

83 /* xd_init: grab the IRQ and DMA channel and initialise the drives */
    ! xd_init:获取IRQ和DMA通道并且初始化XT硬盘
84 u_long xd_init (u_long mem_start,u_long mem_end)
85 {
86     u_char i,controller,*address;
87     ! 注册XT硬盘驱动
88     if (register_blkdev(MAJOR_NR,"xd",&xd_fops)) {
        ! 定义于 Fs/devices.c
        ! 注册失败，打印出错提示后，返回可用内存开始值

```

```

89         printk("xd_init: unable to get major number %d\n",MAJOR_NR);
90         return (mem_start);
91     }
92     blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
! DEVICE_REQUEST = do_xd_request
93     read_ahead[MAJOR_NR] = 8;      /* 8 sector (4kB) read ahead */
! 预留 8k字节缓冲为XT硬盘驱动
94     xd_gendisk.next = gendisk_head;
95     gendisk_head = &xd_gendisk;
! 填写XT驱动的结构
96
97     if (xd_detect(&controller,&address)) {
! 定义于本文件中
98
99     printk("xd_init: detected a%ss controller (type %d)
at address %p\n",xd_sigs[controller].name,controller,address);
! 如果探测到了则打印探测到的数据
100    if (controller)
! 如果找到了控制器，调用对应的控制器初始化函数
101        xd_sigs[controller].init_controller(address);
102        xd_drives = xd_initdrives(xd_sigs[controller].init_drive);
! 定义于本文件中
103
104        printk("xd_init: detected %d hard drive%ss (using IRQ%d &
DMA%d)\n",xd_drives,xd_drives == 1 ? "" : "s",xd_irq,xd_dma);
! 打印探测到的硬盘及占用了那个IRQ
105        for (i = 0; i < xd_drives; i++)
106            printk("xd_init: drive %d geometry - heads = %d, cylinders = %d, sectors =
%d\n",i_xd_info[i].heads,xd_info[i].cylinders,xd_info[i].sectors);
! 打印找到的硬盘信息
107
108        if (!request_irq(xd_irq,xd_interrupt_handler)) {
! 设置对应xd_irq号的IRQ， 定义于Kernel/irq.c
109            if (request_dma(xd_dma)) {
! 看是否可以打开DMA通道不能则打印错误信息同时释放占用的IRQ请求项
110                printk("xd_init: unable to get DMA%d\n",xd_dma);
111                free_irq(xd_irq);
112            }
113        }
114        else
! 不能得到IRQ中断， 打印错误提示
115        printk("xd_init: unable to get IRQ%d\n",xd_irq);
116    }
117    return mem_start;

```

```

    ! 返回可用内存开始值
118 }

119
120 */* xd_detect: scan the possible BIOS ROM locations for the signature strings */
    ! xd_detect:尽可能的扫描BIOS ROM, 以得到签名串
121 static u_char xd_detect (u_char *controller,u_char **address)
122 {
123     u_char i,j,found = 0;
124
125     if (xd_override)
        ! 如果, xd_override=1, 即从命令行设置了xd_override
126     {
        ! 参见代码中xd_setup
127         *controller = xd_type;
128         *address = NULL;
129         return(1);
    ! 返回 1
130     }
131
132     for (i = 0; i < (sizeof(xd_bases) / sizeof(xd_bases[0])) && !found; i++)
133         for (j = 1; j < (sizeof(xd_sigs) / sizeof(xd_sigs[0])) && !found; j++)
134     if(!memcmp(xd_bases[i] + xd_sigs[j].offset,xd_sigs[j].string,strlen(xd_sigs[j].string)))
    {
        ! 从 xd_bases 数组设置的地址中, 逐个查找 xd_sigs 数组中
        ! 的字符签名, 如果有的话设置! 把索引赋给 controller, 同
        ! 时让adderss指向被找到的地址
135         *controller = j;
136         *address = xd_bases[i];
137         found++;
138     }
139     return (found);
    ! 返回是否已经找到, 0 未找到, 1 已经找到!
140 }

```

! 初始化 xd 驱动的操作函数  
 × init\_drive=对应驱动器的读写函数

```

457 static u_char xd_initdrives (void (*init_drive)(u_char drive))
458 {
459     u_char cmdblk[6],i,count = 0;
460

```

```

461     for (i = 0; i < XD_MAXDRIVES; i++) {
        ! XD_MAXDRIVES=最大驱动器数
462         xd_build(cmdblk,CMD_TESTREADY,i,0,0,0,0,0);
        ! xd_build根据硬盘的特性填写命令块
463         if (!xd_command(cmdblk,PIO_MODE,0,0,0,XD_TIMEOUT * 2)) {
            ! xd_command执行cmdblk中命令
464             init_drive(count);
465             count++;
466         }
467     }
468     return (count);
        ! 返回驱动器个数
469 }

```

## Drivers/block/ramdisk.c (部分代码)

```

72 /*
73 * Returns amount of memory which needs to be reserved.
74 */
    ! 返回还有多少内存可用
75 long rd_init(long mem_start, int length)
76 {
77     int i;
78     char *cp;
79
80     if (register_blkdev(MEM_MAJOR,"rd",&rd_fops)) {
        ! 定义于Fs/devices.c注册虚拟盘
81         printk("RAMDISK: Unable to get major %d.\n", MEM_MAJOR);
82         return 0;
83     }
84     blk_dev[MEM_MAJOR].request_fn = DEVICE_REQUEST;
85     rd_start = (char *) mem_start;
86     rd_length = length;
87     cp = rd_start;
88     for (i=0; i < length; i++)
89         *cp++ = '\0';
        ! 清零虚拟盘
90
91     for(i=0;i<2;i++) rd_blocksizes[i] = 1024;

```

```

! 虚拟盘 1 k / 块
92     blksize_size[MAJOR_NR] = rd_blocksizes;
93
94     return(length);
! 返回可用内存开始值
95 }

```

## Drivers/block/l1\_rw\_blk.c (部分代码)

```

122 /* RO fail safe mechanism */
123
124 static long ro_bits[MAX_BLKDEV][8];
125
! 测试对应的设备是否为只读
× dev=设备值
126 int is_read_only(int dev)
127 {
128     int minor,major;
129
130     major = MAJOR(dev);
131     minor = MINOR(dev);
! major=主设备号
! minor=次设备号
132     if (major < 0 || major >= MAX_BLKDEV) return 0;
! 如果主设备号小于 0 或者大于 1.0 核心所能支持的最大块设备数，则返回 0
133     return ro_bits[major][minor >> 5] & (1 << (minor & 31));
! 否则，返回对应的块设备被安装时所设的值（只读、只写，读写方式）
134 }
135

```

! 创建请求项并插入请求队列  
× major=主设备号  
× rw=读、写命令  
× bh=存放数据缓冲区的头指针

```

182 static void make_request(int major,int rw, struct buffer_head * bh)
183 {
184     unsigned int sector, count;
185     struct request * req;
186     int rw_ahead, max_req;

```

```

187
188 /* WRITEA/READA is special case - it is not really needed, so if the */
189 /* buffer is locked, we just forget about it, else it's a normal read */
    ! WRITEA/READA是一种特殊情况，它们并非真正需要，所以
    ! 如果缓冲被锁定的话，我们就会退出而不管它，不然它们就是
    ! 正常的读写。
190     rw_ahead = (rw == READA || rw == WRITEA);
        ! 取得是否是延迟读、写命令
191     if (rw_ahead) {
192         if (bh->b_lock)
193             return;
        ! 如果是延迟读、写命令并且缓冲已经被锁定，则直接返回。
194         if (rw == READA)
195             rw = READ;
196         else
197             rw = WRITE;
        ! 否则，就设置为普通的读写。
198     }
199     if (rw!=READ && rw!=WRITE) {
200         printk("Bad block dev command, must be R/W/RA/WA\n");
201         return;
        ! 如果，既不是读也不是写，则打印错误提示后返回（好像是不可能发生的）
202     }
203     count = bh->b_size >> 9;
        ! 计算扇区个数（扇区个数=缓冲的大小/512）
204     sector = bh->b_blocknr * count;
        ! 计算起始扇区数
205     if (blk_size[major])
206         if (blk_size[major][MINOR(bh->b_dev)] < (sector + count)>>1) {
207             bh->b_dirt = bh->b_uptodate = 0;
208             return;
209         }
210         lock_buffer(bh);
        ! 锁定缓冲
211         if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
212             unlock_buffer(bh);
213             return;
        ! 如果命令是写，并且该缓冲是干净的。或者命令是读，并且该缓冲已经被修改
        ! 则解锁缓冲后，退出。
214     }
215
216 /* we don't allow the write-requests to fill up the queue completely:
217 * we want some room for reads: they take precedence. The last third
218 * of the requests are only for reads.

```

```

219 */
    ! 我们不允许写请求填满整个队列；我们想留下一些空间给对请求：
    ! 它们（读请求）是优先的。最后的 1/3 之一是留给读请求的。
220     max_req = (rw == READ) ? NR_REQUEST : ((NR_REQUEST*2)/3);
    ! 如果命令是读，则让max_req=为请求队列尾，否则为请求队列的 2/3 处
221
222 /* big loop: look for a free request. */
    ! 大的循环：查找一个空闲的请求项。
223
224 repeat:
225     cli();
    ! 关中断
226
227 /* The scsi disk drivers completely remove the request from the queue when
228 * they start processing an entry. For this reason it is safe to continue
229 * to add links to the top entry for scsi devices.
230 */
    ! 当 SCSI 磁盘驱动程序开始处理入口点时，便会完全的从请求队列中
    ! 移除请求项。由于这个原因把一个请求加到SCSI驱动入口点的顶部是安全的。
231     if ((major == HD_MAJOR
232         || major == SCSI_DISK_MAJOR
233         || major == SCSI_CDROM_MAJOR)
234         && (req = blk_dev[major].current_request))
235     {
        ! 如果 major=HD_MAJOR，或者等于 SCSI_DISK_MAJOR 或者等于
        ! SCSI_CDROM_MAJOR，并且对应major号的请求项存在
236         if (major == HD_MAJOR)
237             req = req->next;
238         while (req) {
239             if (req->dev == bh->b_dev &&
240                 !req->waiting &&
241                 req->cmd == rw &&
242                 req->sector + req->nr_sectors == sector &&
243                 req->nr_sectors < 254)
244             {
245                 req->bhtail->b_reqnext = bh;
246                 req->bhtail = bh;
247                 req->nr_sectors += count;
248                 bh->b_dirt = 0;
249                 sti();
250                 return;
251             }
252
253         if (req->dev == bh->b_dev &&

```

```

254             !req->waiting &&
255             req->cmd == rw &&
256             req->sector - count == sector &&
257             req->nr_sectors < 254)
258     {
259         req->nr_sectors += count;
260         bh->b_reqnext = req->bh;
261         req->buffer = bh->b_data;
262         req->current_nr_sectors = count;
263         req->sector = sector;
264         bh->b_dirt = 0;
265         req->bh = bh;
266         sti();
267         return;
268     }
269
270     req = req->next;
271 }
272 }
273
274 /* find an unused request. */
275     req = get_request(max_req, bh->b_dev);
276
277 /* if no request available: if rw_ahead, forget it; otherwise try again. */
278     if (!req) {
279         if (rw_ahead) {
280             sti();
281             unlock_buffer(bh);
282             return;
283         }
284         sleep_on(&wait_for_request);
285         sti();
286         goto repeat;
287     }
288
289 /* we found a request. */
290     sti();
291
292 /* fill up the request-info, and add it to the queue */
293     req->cmd = rw;
294     req->errors = 0;
295     req->sector = sector;
296     req->nr_sectors = count;
297     req->current_nr_sectors = count;

```

```

298     req->buffer = bh->b_data;
299     req->waiting = NULL;
300     req->bh = bh;
301     req->bhtail = bh;
302     req->next = NULL;
303     add_request(major+blk_dev,req);
304 }
305

```

339 /\* This function can be used to request a number of buffers from a block  
340 device. Currently the only restriction is that all buffers must belong to  
341 the same device \*/

! 该函数可以被块设备用于请求一些缓冲，目前的限制是所有的缓冲必须  
 ! 来源于同一个设备。  
 ! 读写块设备数据函数  
 × rw=读/写命令  
 × nr=缓冲数量  
 × bh=缓冲区指针数组（即要被读/写缓冲的指针数组）

```

342
343 void ll_rw_block(int rw, int nr, struct buffer_head * bh[])
344 {
345     unsigned int major;
346     struct request plug;
347     int plugged;
348     int correct_size;
349     struct blk_dev_struct * dev;
350     int i;
351
352     /* Make sure that the first block contains something reasonable */
353     ! 确信首缓冲中包含一些正确的东西
354     while (!*bh) {
355         bh++;
356         if (--nr <= 0)
357             return;
358
359         dev = NULL;
360         if ((major = MAJOR(bh[0]->b_dev)) < MAX_BLKDEV)
361             dev = blk_dev + major;
362
363         ! 让 dev 指向对应于主设备号的处理结构（即该结构中含有对应于主设备号 major
364         ! 的函数处理指针）

```

```

362     if (!dev || !dev->request_fn) {
        ! 如果 dev=null 或者该设备的请求函数指针为 null，则打印错误提示后,
        ! 跳转到sorry
363         printk(
364             "ll_rw_block: Trying to read nonexistent block-device %04lX (%ld)\n",
365             (unsigned long) bh[0]->b_dev, bh[0]->b_blocknr);
        ! 定义于Kernel/printk.c
366         goto sorry;
367     }
368
369     /* Determine correct block size for this device. */
        ! 获取当前设备的正确的块大小
370     correct_size = BLOCK_SIZE;
        ! 先定义correct_size为系统默认的大小
371     if (blksize_size[major]) {
372         i = blksize_size[major][MINOR(bh[0]->b_dev)];
            if (i)
374             correct_size = i;
        ! 如果对应 major (主设备号) 的设备自己定义了每块的大小，则用定义的大小
        ! 来代替系统的默认大小
375     }
376
377     /* Verify requested block sizes. */
        ! 校验被请求的块大小
378     for (i = 0; i < nr; i++) {
379         if (bh[i] && bh[i]->b_size != correct_size) {
380             printk(
381                 "ll_rw_block: only %d-char blocks implemented (%lu)\n",
382                 correct_size, bh[i]->b_size);
        ! 定义于Kernel/printk.c
383             goto sorry;
384         }
        ! 根据要读、写的缓冲数量，去循环校验这些缓冲指针中所描述的每块大小
        ! 是否与上面取得的大小相等，如果不等则打印错误提示，跳转到sorry去。
385     }
386
387     if ((rw == WRITE || rw == WRITEA) && is_read_only(bh[0]->b_dev)) {
        ! 定义于本文件
388         printk("Can't write to read-only device 0x%X\n", bh[0]->b_dev);
        ! 定义于Kernel/printk.c
389         goto sorry;
        ! 如果命令是写或者延迟写，但是拥有该缓冲的设备却是只读的话,
        ! 则打印错误提示后，跳转到sorry
390     }

```

```

391
392      /* If there are no pending requests for this device, then we insert
393 a dummy request for that device. This will prevent the request
394 from starting until we have shoved all of the blocks into the
395 queue, and then we let it rip. */
396
397     plugged = 0;
398     cli();
399     ! 关中断
400     if (!dev->current_request && nr > 1) {
401         dev->current_request = &plug;
402         plug.dev = -1;
403         plug.next = NULL;
404         plugged = 1;
405         ! 如果当前设备的请求项指针为空并且有要读、写的缓冲，那么
406         ! 让当前设备的请求项指针指向plug，并设置plug中的数据。
407     }
408     sti();
409     ! 开中断
410     for (i = 0; i < nr; i++) {
411         if (bh[i]) {
412             bh[i]->b_req = 1;
413             make_request(major, rw, bh[i]);
414         }
415         ! 定义于本文件，创建请求项并插入请求队列
416         if (rw == READ || rw == READA)
417             kstat.pgpgin++;
418         else
419             kstat.pgpgout++;
420     }
421     ! 根据要读、写的缓冲数量，如果对应的缓冲指针不为 null（即确实有数据要读、
422     ! 写），便创建请求项并插入请求队列中。同时置核心作态
423
424     if (plugged) {
425         ! 如果，真的插入了个没有用请求
426         cli();
427         ! 关中断
428         dev->current_request = plug.next;
429         ! 跳过plug（即跳过上面插入的plug）
430         (dev->request_fn)();
431         ! 调用处理函数（该处理函数，根据块设备的不同而不同）
432         sti();

```

```

    ! 开中断
421     }
422     return;
    ! 返回

423
424     sorry:
425     for (i = 0; i < nr; i++) {
426         if (bh[i])
427             bh[i]->b_dirt = bh[i]->b_uptodate = 0;
    ! 执行这里的代码，说明有问题发生，修改缓冲的标志！
428     }
429     return;
    ! 返回
430 }
431

```

! 块设备的初始化，这些块设备有硬盘、光盘及虚拟盘！  
! 初始化的情况是根据用户在配置内核时的选择值  
× mem\_start=可用内存起始值  
× mem\_end=可用内存结束值

```

475 long blk_dev_init(long mem_start, long mem_end)
476 {
477     struct request * req;
478
479     req = all_requests + NR_REQUEST;
    ! req指向请求项的最后一项
480     while (--req >= all_requests) {
481         req->dev = -1;
482         req->next = NULL;
    ! 清零请求项
483     }
484     memset(ro_bits,0,sizeof(ro_bits));
    ! 清零ro_bits
485 #ifdef CONFIG_BLK_DEV_HD
486     mem_start = hd_init(mem_start,mem_end);
    ! 定义于Drivers/block/hd.c
487 #endif
488 #ifdef CONFIG_BLK_DEV_XD
    ! 定义于Drivers/block/xd.c
489     mem_start = xd_init(mem_start,mem_end);
490 #endif
    ! 忽略CDU31A, MCS, SBPCD
491 #ifdef CONFIG_CDU31A
492     mem_start = cdu31a_init(mem_start,mem_end);

```

```

493 #endif
494 #ifdef CONFIG_MCD
495     mem_start = mcd_init(mem_start,mem_end);
496 #endif
497 #ifdef CONFIG_SPCD
498     mem_start = spsc_init(mem_start, mem_end);
499 #endif CONFIG_SPCD
500     if (ramdisk_size)
501         ! 如果定义了虚拟盘
502             mem_start += rd_init(mem_start, ramdisk_size*1024);
503             ! 定义于Drivers/block/ramdisk.c
504             return mem_start;
505             ! 返回可用内存的开始值
506 }
507

```

## Devices/net/lance.c (部分代码)

```

    ! 初始化 AMD 的一种网卡，该网卡和 NE2000 兼容，
    ! 所以该驱动也可以用于 NE2000 的网卡
222 unsigned long lance_init(unsigned long mem_start, unsigned long mem_end)
223 {
224     int *port, ports[] = {0x300, 0x320, 0x340, 0x360, 0};
225     ! ports[]中定义了要检查的端口
226     for (port = &ports[0]; *port; port++) {
227         int ioaddr = *port;
228
229         if ( check_region(ioaddr, LANCE_TOTAL_SIZE) == 0
230             && inb(ioaddr + 14) == 0x57
231             && inb(ioaddr + 15) == 0x57) {
232             ! 检查从 ioaddr 开始的 LANCE_TOTAL_SIZE 地址，如果该范围的地址存在，并且
233             ! 读基址偏移 14 和 15 处的值都为 0x57，则可以探测该地址处是否真正被网卡占用
234             mem_start = lance_probe1(ioaddr, mem_start);
235             ! lance_probe1 中包括了对占用该地
236             ! 址开始的网卡的特性探测，主要就是对其端口进行尝试行的读写。
237             ! 我手中没有对应的硬件资料，该处忽略，不影响理解核心！
238         }
239     }
240

```

```

236     return mem_start;
    ! 返回可用内存的开始值
237 }
238

```

## Devices/net/net\_init.c (部分代码)

```

63 /*
64 net_dev_init() is our network device initialization routine.
65 It's called from init/main.c with the start and end of free memory,
66 and returns the new start of free memory.
67 */
68
    ! net_dev_init()是我们的网络设备初始化程序，它在 init/main.c 中以 start 和 end
    ! 空闲内存作为参数被调用并且返回新的可用内存的开始值
    ! 初始化网络设备
    × mem_start=可用内存起始值
    × mem_end=可用内存的结束值
69 unsigned long net_dev_init (unsigned long mem_start, unsigned long mem_end)
70 {
71
72 #ifdef NET_MAJOR_NUM
    ! 如果定义了网络设备的主设备号则注册之
73     if (register_chrdev(NET_MAJOR_NUM, "network", &netcard_fops))
    ! 定义于Fs/devices.c
74         printk("WARNING: Unable to get major %d for the network devices.\n",
75             NET_MAJOR_NUM);
    ! 注册失败时，打印出错消息
76 #endif
77
78 #if defined(CONFIG_LANCE)          /* Note this is _not_ CONFIG_AT1500. */
    ! 注意这个不对应 CONFIG_AT1500
    ! lance_init 兼容 NE2000 类型的网卡
    ! 定义于Drivers/net/lance.c
79     mem_start = lance_init(mem_start, mem_end);
80 #endif
81
82     return mem_start;

```

! 返回可用内存的开始值  
83 }

## F

### Fs/fcntl.c (部分代码)

! 复制文件句柄  
 × fd=被复制的文件句柄  
 × arg=新文件句柄的最小值  
20 static int dupfd(unsigned int fd, unsigned int arg)  
21 {  
22 if (fd >= NR\_OPEN || !current->filp[fd])  
23 return -EBADF;  
 ! 如果超出范围，则返回-EBADF  
24 if (arg >= NR\_OPEN)  
25 return -EINVAL;  
 ! 如果传入的新文件句柄的最小值，大于NR\_OPEN，则返回-EINVAL  
26 while (arg < NR\_OPEN)  
27 if (current->filp[arg])  
28 arg++;  
29 else  
30 break;  
 ! 从当前进程中文件描述符指针数组中查找到大于等于 arg,  
 ! 但还没有被使用的项  
31 if (arg >= NR\_OPEN)  
32 return -EMFILE;  
 ! 如果找到的新文件句柄值，大于NR\_OPEN，则返回-EINVAL  
33 FD\_CLR(arg, &current->close\_on\_exec);  
 ! 复位执行时关闭标志位图中得对应位  
34 (current->filp[arg] = current->filp[fd])->f\_count++;  
 ! 引用计数加一  
35 return arg;  
 ! 返回找到得新文件句柄  
36 }  
37

! 相同调用 sys\_dup, 用于复制文件句柄

× fildes=被复制的文件句柄

```

61 asmlinkage int sys_dup(unsigned int fildes)
62 {
63     return dupfd(fildes,0);
    ! 定义于本文件
64 }
65

```

## Fs/exec.c (部分代码)

! 以 mode 方式打开 inode，并返回文件描述符

× inode=要打开的 inode

× mode=打开方式

```

56 int open_inode(struct inode * inode, int mode)
57 {
58     int error, fd;
59     struct file *f, **fpp;
60
61     if (!inode->i_op || !inode->i_op->default_file_ops)
62         return -EINVAL;
    ! 如果函数操作指针不存在，则返回-EINVAL
63     f = get_empty_filp();
    ! 从全局文件描述符表取得一个空的描述符
64     if (!f)
65         return -EMFILE;
    ! 如果为空，则返回-EMFILE
66     fd = 0;
67     fpp = current->filp;
    ! 让fpp指向文件描述符指针
68     for (;;) {
69         if (!*fpp)
70             break;
    ! 找到空的指针后，跳出循环
71         if (++fd > NR_OPEN)
72             return -ENFILE;
    ! 如果文件描述符指针大于一个进程所能打开的文件数，则返回-ENFILE
73         fpp++;
    ! fpp自增一
74     }

```

```

75     *fpp = f;
76     f->f_flags = mode;
    ! 设置上打开模式
77     f->f_mode = (mode+1) & O_ACCMODE;
78     f->f_inode = inode;
79     f->f_pos = 0;
80     f->f_reada = 0;
81     f->f_op = inode->i_op->default_file_ops;
    ! 设置f的各个值
82     if (f->f_op->open) {
83         error = f->f_op->open(inode,f);
84         if (error) {
85             *fpp = NULL;
86             f->f_count--;
87             return error;
88         }
    ! 如果打开函数存在，则调用打开之。如果出错，
    ! 则置空一些内容后返回错误值
89     }
90     inode->i_count++;
    ! 文件的引用计数加一
91     return fd;
    ! 返回文件描述符值
92 }

```

```

263 /*
264 * create_tables() parses the env- and arg-strings in new user
265 * memory and creates the pointer tables from them, and puts their
266 * addresses on the "stack", returning the new stack pointer value.
267 */
    ! create_table () 在新用户内存空间中解析环境变量和参数字符串，并且
    ! 创建指针表，同时把它们的地址放到堆栈上，最后返回新的堆栈的指针
    ✗ p=以数据段为起点的参数和环境信息偏移指针
    ✗ argc=参数个数
    ✗ envc=环境变量的个数
    ✗ ibcs=不知
268 unsigned long * create_tables(char * p,int argc,int envc,int ibcs)
269 {
270     unsigned long *argv,*envp;
271     unsigned long * sp;
272     struct vm_area_struct *mpnt;
273
274     mpnt = (struct vm_area_struct *)kmalloc(sizeof(*mpnt), GFP_KERNEL);

```

```

    ! 分配一页内存
275     if (mpnt) {
        ! 如果分配成功
276         mpnt->vm_task = current;
277         mpnt->vm_start = PAGE_MASK & (unsigned long) p;
278         mpnt->vm_end = TASK_SIZE;
279         mpnt->vm_page_prot = PAGE_PRIVATE|PAGE_DIRTY;
280         mpnt->vm_share = NULL;
281         mpnt->vm_inode = NULL;
282         mpnt->vm_offset = 0;
283         mpnt->vm_ops = NULL;
284         insert_vm_struct(current, mpnt);
285         current->stk_vma = mpnt;
        ! 则把一些必要的数据与之绑定或者设置上一些值
286     }
287     sp = (unsigned long *) (0xfffffffffc & (unsigned long) p);
        ! 调整指针，让其按 4 个字节对齐
288     sp -= envc+1;
        ! sp向下移动，是为了给环境参数留出空间
289     envp = sp;
290     sp -= argc+1;
        ! sp向下移动，是为了给参数留出空间
291     argv = sp;
        ! 让 argv指向sp
292     if (!ibcs) {
        ! 如果ibcs为 0，则将环境参数指针和命令参数指针压入堆栈
293         put_fs_long((unsigned long)envp,--sp);
294         put_fs_long((unsigned long)argv,--sp);
295     }
296     put_fs_long((unsigned long)argc,--sp);
        ! 命令行参数个数压入堆栈
297     current->arg_start = (unsigned long) p;
        ! 让 arg_start=参数和环境信息偏移值（这样我们可以通过该值取的当前进程
        ! 参数和环境信息了）
298     while (argc-->0) {
        ! 根据命令行参数的个数，把命令行参数的指针放到前面为其空出的空间中
299         put_fs_long((unsigned long) p,argv++);
300         while (get_fs_byte(p++) /* nothing */);
        ! 调整指针，让其指向下一个程序参数处
301     }
302     put_fs_long(0,argv);
        ! 放入NULL指针
303     current->arg_end = current->env_start = (unsigned long) p;
        ! 设置参数结束值及环境参数的开始值

```

```

304     while (envc-->0) {
    ! 根据环境参数的个数，把环境参数的指针放到前面为其空出的空间中
305         put_fs_long((unsigned long) p.envp++);
306         while (get_fs_byte(p++) /* nothing */;
    ! 调整指针，让其指向下一个程序参数处
307     }
308     put_fs_long(0,envp);
    ! 放入NULL指针
309     current->env_end = (unsigned long) p;
    ! 设置环境参数的结束值
310     return sp;
    ! 返回当前进程的堆栈指针
311 }

```

```

312
313 /*
314 * count() counts the number of arguments/envelopes
315 */
    ! count()计算命令行参数、环境变量的个数
    × argv=参数指针的指针
316 static int count(char ** argv)
317 {
318     int i=0;
319     char ** tmp;
320
321     if ((tmp = argv) != 0)
    ! 当指针的指针不为空时
322         while (get_fs_long((unsigned long *) (tmp++)))
323             i++;
    ! i中放着个数（请注意tmp是指针的指针）
324
325     return i;
    ! 返回计算得到的个数
326 }
327
328 */

```

```

329 * 'copy_string()' copies argument/envelope strings from user
330 * memory to free pages in kernel mem. These are in a format ready
331 * to be put directly into the top of new user memory.
332 *

```

! copy\_string()函数从用户空间中复制参数和环境字符串到内核的  
 ! 空闲页面内存中  
 ! 这些已经具有直接放到新用户内存中的格式

333 \* *Modified by TYT, 11/24/91 to add the from\_kmem argument, which specifies*

334 \* *whether the string and the string array are from user or kernel segments:*

335 \*

! TYT修改，11/24/91。增加了from\_kmem参数，这个参数说明了字符串或者字符  
 ! 串数组来自于用户段还是核心段

336 \* *from\_kmem argv \* argv \*\**

337 \* *0 user space user space*

338 \* *1 kernel space user space*

339 \* *2 kernel space kernel space*

! from_kmem	argv*	argv**
!	0	用户空间
!	1	核心空间
!	2	内核空间

340 \*

341 \* *We do this by playing games with the fs segment register. Since it*

342 \* *it is expensive to load a segment register, we try to avoid calling*

343 \* *set\_fs() unless we absolutely have to.*

344 \*/

! 我们是通过巧妙处理fs段寄存器来操作的，因为如果加载一个段寄存器的话  
 ! 代价太大，所以我们应该尽量避免调用set\_fs()，除非迫不得已。

! 复制指定个数的参数串到参数和环境空间。

- × argc=参数的个数
- × argv=参数指针数组
- × page=参数和环境空间页面指针数组
- × p=在参数表空间中的偏移指针
- × from\_kmem=字符串来源标志

```

345 unsigned long copy_strings(int argc,char ** argv,unsigned long *page,
346           unsigned long p, int from_kmem)
347 {
348     char *tmp, *pag = NULL;
349     int len, offset = 0;
350     unsigned long old_fs, new_fs;
351
352     if (!p)
        ! 如果偏移指针为空，则直接返回
353         return 0; /* bullet-proofing */
        ! 偏移指针验证
354     new_fs = get_ds();
        ! 把段寄存器ds的值保存在new_fs中

```

```

355     old_fs = get_fs();
    ! 取段寄存器fs的值到old_fs中
356     if (from_kmem==2)
    ! 如果字符串和字符串数组来源于核心空间
357         set_fs(new_fs);
    ! 则让 fs 指向核心数据段, (因为我们在字符串操作时, 默认的段寄存器是 fs, 所以
    ! 这里我们把 fs 的值设置为核心数据段的话, 那么其对字符串的操作就是在核心态
    ! 做的了)
358     while (argc-- > 0) {
        ! 让参数个数作为要重复执行的次数。来做循环。
359         if (from_kmem == 1)
        ! 如果字符串在用户空间但是字符串数组在核心空间
360             set_fs(new_fs);
        ! 则让段寄存器fs的值指向核心数据段
361             if (!(tmp = (char *)get_fs_long((unsigned long *)argv) + argc)))
        ! 从最后一个参数开始逆向操作, 让 tmp 指向 fs 段中的最后一个参数处,
        ! 如果为空, 则打印错误提示后, 死机!
362             panic("VFS: argc is wrong");
        ! 定义于Kernel/panic.c
363         if (from_kmem == 1)
        ! 如果字符串在用户空间但是字符串数组在核心空间, 则恢复老的fs值
364             set_fs(old_fs);
365             len=0;      /* remember zero-padding */
                ! 请记着字符串是以 0 结尾的
366             do {
367                 len++;
                    ! 计算得到字符串的长度
368             } while (get_fs_byte(tmp++));
369             if (p < len) { /* this shouldn't happen - 128kB */
                ! 这个情况是不可能发生的, 因为我们有 128k 大小的空间
                ! 如果我们在上面计算得到的字符串长度大于这时参数和环境空间中的剩余空
                ! 闲长度的话, 则恢复老的fs段寄存器值, 并且返回 0
370                 set_fs(old_fs);
371                 return 0;
372             }
            ! 复制fs段寄存器指定的参数字符串, 请注意是从串尾开始的
373             while (len) {
374                 --p; --tmp; --len;
375                 if (--offset < 0) {
                    ! 当函数刚开始执行时, 偏移量 offset 被初始化为 0, 因此当 offset-1<0 的话, 则
                    ! 说明是第一次复制字符串。
                    ! 并让它等于p指针在页面内的偏移值, 同时还申请空闲的页面
376                 offset = p % PAGE_SIZE;
377                 if (from_kmem==2)

```

```

    ! 如果字符串和字符串数组在核心空间，则恢复fs段寄存器值
378         set_fs(old_fs);
379         if (!(pag = (char *) page[p/PAGE_SIZE]) &&
380             !(pag = (char *) page[p/PAGE_SIZE] =
381               (unsigned long *) get_free_page(GFP_USER)))
382             return 0;
    ! 如果当前偏移 p 所在串的空间页面指针数组项 page[p/PAGE_SIZE]==0 的话,
    ! 则表示相应的页面并不存在，这时就要申请新的空闲页面，将该页面指针填
    ! 入指针数组，并且让 pag 指向该申请的新的页面，倘若申请空闲内存页失败
    ! 则直接返回 0
383     if (from_kmem==2)
    ! 如果字符传和字符串数组来源于核心空间，则让段寄存器fs指向核心数据段
384         set_fs(new_fs);
385
386     }
387     *(pag + offset) = get_fs_byte(tmp);
    ! 从fs段中复制一个字符到pag+offset处
388     }
389     }
390     if (from_kmem==2)
    ! 如果字符和字符串数组在核心空间，则恢复老的fs值
391         set_fs(old_fs);
392     return p;
    ! 返回参数和环境空间中已经复制参数信息的头部偏移值
393 }
394

```

```

415 /*
416 * Read in the complete executable. This is used for "-N" files
417 * that aren't on a block boundary, and for files on filesystems
418 * without bmap support.
419 */
    ! 把可执行文件完全的读入。它被用于“-N”没有按块对齐的文件，并且
    ! 用于没有bmap支持的文件
420 int read_exec(struct inode *inode, unsigned long offset,
421             char * addr, unsigned long count)
422 {
423     struct file file;
424     int result = -ENOEXEC;
425
426     if (!inode->i_op || !inode->i_op->default_file_ops)
427         goto end_readexec;

```

```

    ! 如果 i 节点的操作函数指针为空或者其对应的文件操作函数指针为空，则
    ! 跳转到end_readexec

428     file.f_mode = 1;
429     file.f_flags = 0;
430     file.f_count = 1;
431     file.f_inode = inode;
432     file.f_pos = 0;
433     file.f_reada = 0;
434     file.f_op = inode->i_op->default_file_ops;

    ! 以上到这，设置文件结构中的值

435     if (file.f_op->open)
436         if (file.f_op->open(inode,&file))
437             goto end_readexec;

    ! 如果文件的打开函数存在，则调用之

438     if (!file.f_op || !file.f_op->read)
439         goto close_readexec;

    ! 如果对应的文件函数操作指针为空或者读函数指针为空，则跳转到
    ! close_readexec处。

440     if (file.f_op->lseek) {
441         if (file.f_op->lseek(inode,&file,offset,0) != offset)
442             goto close_readexec;

    ! 如果 lseek 不为空，并且可以 seek 到 offset 个字节，则跳转到
    ! close_readexec处执行

443     } else
444         file.f_pos = offset;
    ! 否则，直接设置文件偏移位置

445     if (get_fs() == USER_DS) {
        ! 如果当前段寄存器fs值等于用户态数据值

446         result = verify_area(VERIFY_WRITE, addr, count);
        ! 则验证addr处的count个字节是否可以写

447         if (result)
448             goto close_readexec;

        ! 如果不可以写（说明用户态的该块内存区域已经不在内存中了
        ! 可能已经被换入到硬盘上了），则跳转到close_readexec执行

449     }
450     result = file.f_op->read(inode, &file, addr, count);
        ! 读取 file 的 count 个字节，这里的 read 指针，我这里就不作跟踪了。感兴趣的
        ! 可以自己研究

451 close_readexec:
452     if (file.f_op->release)
453         file.f_op->release(inode,&file);
        ! 关闭对应的文件

454 end_readexec:
455     return result;

```

```

    ! 返回读入的字节数
456 }
457

459 /*
460 * This function flushes out all traces of the currently running executable so
461 * that a new one can be started
462 */
463

    ! 该函数清空当前可执行文件所有陷阱，以致于它可以作为一个新的程序
    ! 执行。
    × bprm=二进制文件头
464 void flush_old_exec(struct linux_binprm * bprm)
465 {
466     int i;
467     int ch;
468     char * name;
469     struct vm_area_struct * mpnt, *mpnt1;
470
471     current->dumpable = 1;
472     name = bprm->filename;
473     for (i=0; (ch = *(name++)) != '\0';) {
474         if (ch == '/')
475             i = 0;
476         else
477             if (i < 15)
478                 current->comm[i++] = ch;
        ! 把当前文件的名字放入，当前进程的 comm 数组中，请注意名字最大只有
        ! 15 个字节
479     }
480     current->comm[i] = '\0';
        ! 置 0 最后一个字节
481     if (current->shm)
482         shm_exit();
        ! 如果当前shm不为空，释放共享内存，定义于Ipc/shm.c
483     if (current->executable) {
484         iput(current->executable);
485         current->executable = NULL;
        ! 如果当前文件（其实是原程序）可以执行，则释放该I节点后，让其指向NULL
486     }
487     /* Release all of the old mmap stuff. */
488

```

```

    ! 释放所有的老的mmap内容
489     mpnt = current->mmap;
    ! 让mpnt指向mmap
490     current->mmap = NULL;
491     current->stk_vma = NULL;
    ! 置空两个字段
492     while (mpnt) {
        ! 如果虚拟内存管理器指针不为空
493         mpnt1 = mpnt->vm_next;
494         if (mpnt->vm_ops && mpnt->vm_ops->close)
495             mpnt->vm_ops->close(mpnt);
496         kfree(mpnt);
497         mpnt = mpnt1;
        ! 则循环释放之
498     }
499
500     /* Flush the old ldt stuff... */
    ! 释放老的ldt内容
501     if (current->ldt) {
        ! 如果当前进程的ldt表存在
502         free_page((unsigned long) current->ldt);
        ! 则释放之, 定义于Mm/swap.c
503         current->ldt = NULL;
        ! 置空ldt
504         for (i=1 ; i<NR_TASKS ; i++) {
            ! 遍历所有的任务
505             if (task[i] == current) {
                ! 如果找到了
506                 set_ldt_desc(gdt+(i<<1)+
507                             FIRST_LDT_ENTRY,&default_ldt, 1);
                ! 则把default_ldt设置到对应的gdt表中
508                 load_ldt(i);
                ! 加载对应的ldt
509             }
510         }
511     }
512
513     for (i=0 ; i<8 ; i++) current->debugreg[i] = 0;
    ! 清空调试用寄存器
514
515     if (bprm->e_uid != current->euid || bprm->e_gid != current->egid ||
516         !permission(bprm->inode,MAY_READ)
517         current->dumpable = 0;
    ! 根据权限的判断来设置dumpable

```

```

518     current->signal = 0;
    ! 置空信号
519     for (i=0 ; i<32 ; i++) {
520         current->sigaction[i].sa_mask = 0;
521         current->sigaction[i].sa_flags = 0;
    ! 清空所有信号处理句柄
522         if (current->sigaction[i].sa_handler != SIG_IGN)
523             current->sigaction[i].sa_handler = NULL;
    ! 但是当信号处理句柄为SIG_IGN时，则不能置空
524     }
525     for (i=0 ; i<NR_OPEN ; i++)
526         if (FD_ISSET(i,&current->close_on_exec))
527             sys_close(i);
    ! 根据执行时关闭文件句柄位图标志，关闭指定的打开文件
528     FD_ZERO(&current->close_on_exec);
    ! 复位该标志
529     clear_page_tables(current);
    ! 清空当前用户的页表，定义于Mm/memory.c
530     if (last_task_used_math == current)
531         last_task_used_math = NULL;
    ! 如果上次的任务使用的的化，置空之。
532     current->used_math = 0;
    ! 复位协处理器使用标志
533     current->elf_executable = 0;
    ! elf_executable置 0
534 }

```

```

536 /*
537 * sys_execve() executes a new program.
538 */
    ! sys_execve()执行一个新程序
    ! 该函数是 sys_execve () 的主程序
    × filename=要执行文件指针
    × argv=传入的参数
    × envp=环境变量值
    × regs=堆栈指针
539 static int do_execve(char * filename, char ** argv, char ** envp, struct pt_regs * regs)
540 {
541     struct linux_binprm bprm;
    ! linux_binprm结构用于存放加载二进制文件时用的参数
542     struct linux_binfmt * fmt;
    ! linux_binfmt结构存放了对应于真正二进制文件的执行函数

```

```

543     unsigned long old_fs;
544     int i;
545     int retval;
546     int sh_bang = 0;
547
548     if (regs->cs != USER_CS)
549         return -EINVAL;
    ! 如果regs所指的堆栈内容中的cs不是用户态时的值，则直接返回-EINVAL
550     bprm.p = PAGE_SIZE*MAX_ARG_PAGES-4;
    ! 参数和环境字符串空间中的偏移指针，初始化为指向该空间的最后一个长字处
551     for (i=0 ; i<MAX_ARG_PAGES ; i++) /* clear page-table */
552         bprm.page[i] = 0;
    ! 清页表
    ! 初始化参数和环境串空间的页面指针数组
553     retval = open_namei(filename, 0, 0, &bprm.inode, NULL);
    ! 打开文件，定义于Fs/namei.c
554     if (retval)
555         return retval;
556     bprm.filename = filename;
557     bprm.argv = count(argv);
    ! 计算执行程序时参数个数
558     bprm.envc = count(envp);
    ! 计算环境变量个数，定义于本文件
559
560 restart_interp:
561     if (!S_ISREG(bprm.inode->i_mode)) { /* must be regular file */
562         retval = -EACCES;
    ! 必须是正规文件
563         goto exec_error2;
    ! 检查文件是否是正规文件，不是则跳转到exec_error2 处执行
564     }
565     if (IS_NOEXEC(bprm.inode)) { /* FS mustn't be mounted noexec */
566         retval = -EPERM;
567         goto exec_error2;
    ! 如果程序不可以被执行，则跳转到exec_error2 处执行
568     }
569     if (!bprm.inode->i_sb) {
570         retval = -EACCES;
571         goto exec_error2;
572     }
573     i = bprm.inode->i_mode;
574     if (IS_NOSUID(bprm.inode) && (((i & S_ISUID) && bprm.inode->i_uid != current->
575         euid) || ((i & S_ISGID) && !in_group_p(bprm.inode->i_gid))) &&
576         !suser()) {
577         retval = -EPERM;
578         goto exec_error2;

```

```

    ! 比较各种条件，不满足则跳转到exec_error2 处执行
579      }
580      /* make sure we don't let suid, sgid files be ptraced. */
    ! 确信我们不能让suid, sgid文件被跟踪
581      if (current->flags & PF_PTRACED) {
582          bprm.e_uid = current->euid;
583          bprm.e_gid = current->egid;
    ! 如果当前进程被置 PF_PTRACED 位，则设置结构中的 e_uid 及 e_gid 为当前
    ! 进程中的值
584      } else {
585          bprm.e_uid = (i & S_ISUID) ? bprm.inode->i_uid : current->euid;
586          bprm.e_gid = (i & S_ISGID) ? bprm.inode->i_gid : current->egid;
    ! 否则，根据第 573 行代码取得的文件类型和属性值去设置结构中的e_uid及e_gid
587      }
588      if (current->euid == bprm.inode->i_uid)
589          i >>= 6;
590      else if (in_group_p(bprm.inode->i_gid))
591          i >>= 3;
    ! 检查将要执行文件的执行权限
592      if (!(i & 1) &&
593          !((bprm.inode->i_mode & 0111) && suser()) {
594          retval = -EACCES;
595          goto exec_error2;
    ! 根据上面取的i值，判断文件是否具有执行权限，没有则跳转到exec_error2 处执行
596      }
597      memset(bprm.buf, 0, sizeof(bprm.buf));
    ! 清空结构中的缓冲区
598      old_fs = get_fs();
    ! 取得fs寄存器值
599      set_fs(get_ds());
    ! 重置fs寄存器的内容为内核数据段
600      retval = read_exec(bprm.inode, 0, bprm.buf, 128);
    ! 读取可执行文件，定义于本文件
601      set_fs(old_fs);
    ! 设置fs为老的保留的值
602      if (retval < 0)
603          goto exec_error2;
604      if ((bprm.buf[0] == '#') && (bprm.buf[1] == '!') && (!sh_bang)) {
    ! 如果执行文件开始的两个字节为#!， 并且sh_bang为 0，则处理脚本文件的执行
605          /*
606          * This section does the #! interpretation.
607          * Sorta complicated, but hopefully it will work. -TYT
608          */
    ! 这部分处理对#! 的解释，有些复杂但希望可以工作—TYT

```

```

609
610     char *cp, *interp, *i_name, *i_arg;
611
612     iput(bprm.inode);
! 释放该执行文件的I节点, 定义于Fs/inode.c
613     bprm.buf[127] = '\0';
! 设置缓冲最后一个字节为 0
614     if ((cp = strchr(bprm.buf, '\n')) == NULL)
615         cp = bprm.buf+127;
! 查找bprm.buf中是否有换行符, 如果没有则让cp指向最后一个字符处
616     *cp = '\0';
! 把找到的位置处设置为 0
617     while (cp > bprm.buf) {
618         cp--;
619         if ((*cp == ' ') || (*cp == '\t'))
620             *cp = '\0';
621         else
622             break;
623     }
! 删除该行的空格以及制表符
624     for (cp = bprm.buf+2; (*cp == ' ') || (*cp == '\t'); cp++);
! 检查文件名
625     if (!cp || *cp == '\0') {
626         retval = -ENOEXEC; /* No interpreter name found */
627         goto exec_error1;
! 如果文件名为空, 则跳转到exec_error1 处
628     }
629     interp = i_name = cp;
! 不然就得到解释程序的名称
630     i_arg = 0;
631     for (; *cp && (*cp != ' ') && (*cp != '\t'); cp++) {
632         if (*cp == '/')
633             i_name = cp+1;
! 让i_name指向程序名。例如是/bin/sh时, 则i_name指向sh
634     }
635     while ((*cp == ' ') || (*cp == '\t'))
636         *cp++ = '\0';
637     if (*cp)
638         i_arg = cp;
! 如果文件名后面还有字符的话, 则应该是参数, 让i_arg指向它
639     /*
640 * OK, we've parsed out the interpreter name and
641 * (optional) argument.
642 */

```

！好，我们已经解析出了解释程序的名字及参数（可以选）

```

643     if (sh_bang++ == 0) {
    ! 如果 sh_bang 标志没有被设置，则加一设置它，并复制指定个数的环境及参数串到
    ! 环境和参数空间中
644         bprm.p = copy_strings(bprm.envc, envp, bprm.page, bprm.p, 0);
645         bprm.p = copy_strings(--bprm.argv, argv+1, bprm.page, bprm.p, 0);
    ! 定于本文件
646     }
647     /*
648 * Splice in (1) the interpreter's name for argv[0]
649 * (2) (optional) argument to interpreter
650 * (3) filename of shell script
651 *
652 * This is done in reverse order, because of how the
653 * user environment and arguments are stored.
654 */
    ! 拼接
    ! (1) argv[0]中放这解释程序的名字
    ! (2) (可以选择的) 解释程序的参数
    ! (3) 脚本程序的名称
    ! 这是以逆序进行处理的，由用户环境和参数的放置方式造成
655     bprm.p = copy_strings(1, &bprm.filename, bprm.page, bprm.p, 2);
    ! 拷贝脚本程序文件名到参数和环境空间中，定义于本文件
656     bprm.argv++;
657     if (i_arg) {
        !如果有参数存在的话，则拷贝参数到参数和环境空间中
658         bprm.p = copy_strings(1, &i_arg, bprm.page, bprm.p, 2);
    ! 定义于本文件
659         bprm.argv++;
660     }
661     bprm.p = copy_strings(1, &i_name, bprm.page, bprm.p, 2);
    ! 复制解释程序的文件名到参数和环境空间中
662     bprm.argv++;
663     if (!bprm.p) {
664         retval = -E2BIG;
665         goto exec_error1;
    !如果有错误发生，则跳转到exec_error1 处执行
666     }
667     /*
668 * OK, now restart the process with the interpreter's inode.
669 * Note that we use open_namei() as the name is now in kernel
670 * space, and we don't need to copy it.
671 */
    ! 好的，现在使用解释程序的 I 节点重新启动进程。

```

```

    ! 注意, open_namei 使用的名字现在已经在核心空间中了, 我们
    ! 不需要在拷贝它了。
672     retval = open_namei(interp, 0, 0, &bprm.inode, NULL);
    ! 打开文件, 定义于Fs/namei.c
673     if (retval)
674         goto exec_error1;
    ! 如果, 打开文件失败, 则跳转到exec_error1 处执行
675     goto restart_interp;
    ! 否则, 跳转到restart_interp处重新执行
676 }
677     if (!sh_bang) {
    ! 如果 sh_bang 标志没有被设置, 则复制指定个数的环境变量字符串到参数
    ! 和环境空间中了。
678     bprm.p = copy_strings(bprm.envc, envp, bprm.page, bprm.p, 0);
679     bprm.p = copy_strings(bprm.argvc, argv, bprm.page, bprm.p, 0);
    ! 定义于本文件
680     if (!bprm.p) {
681         retval = -E2BIG;
682         goto exec_error2;
    ! 拷贝失败后, 跳转到exec_error2 处
683     }
684 }
    ! 反之如果 sh 标志已经被设置了, 则表明该程序是脚本程序, 所以这个时候的
    ! 环境变量页面已经被复制了, 这里便不用在复制了。
685
686     bprm.sh_bang = sh_bang;
687     fmt = formats;
    ! 让fmt执行对应的格式
688     do {
689         int (*fn)(struct linux_binprm *, struct pt_regs *) = fmt->load_binary;
    ! 让 fn 指向对应的加载二进制函数, 目前在 1.0 核心中只支持 3 中二进制文件格式
    ! (1) a.out
    ! (2) elf
    ! (3) coff
690         if (!fn)
691             break;
    ! 如果没有对应的二进制加载函数, 则跳出该循环
692         retval = fn(&bprm, regs);
    ! 否则则调用之, 作为例子, 我们这里讨论一种二进制文件的加载操作, 那就是
    ! a.out的格式, 定义于本文件!
693         if (retval == 0) {
694             iput(bprm.inode);
695             current->did_exec = 1;
696             return 0;

```

```

    ! 如果加载成功了，释放该I节点后返回
697         }
698         fmt++;
! 否则，遍历下个格式
699     } while (retval == -ENOEXEC);
700 exec_error2:
701     iput(bprm.inode);
! 释放对应的I节点
702 exec_error1:
703     for (i=0 ; i<MAX_ARG_PAGES ; i++)
704         free_page(bprm.page[i]);
! 释放申请的页表
705     return(retval);
! 返回值说明有错误发生了
706 }

708 /*
709 * sys_execve() executes a new program.
710 */
! sys_execve()执行一个新程序
×regs=指向进入核心态时的堆栈指针
711 asmlinkage int sys_execve(struct pt_regs regs)
712 {
713     int error;
714     char * filename;
715
716     error = getname((char *)regs.ebx, &filename);
! 把文件名拷贝到核心空间，让filename指向它，定义于Fs/namei.c
717     if (error)
718         return error;
! 有错误发生，则直接返回错误
719     error = do_execve(filename, (char **)regs.ecx, (char **)regs.edx, &regs);
! 执行文件，定义于本文件
720     putname(filename);
! 释放拷贝时，所占用的内存空间
721     return error;
! 返回错误码（如果执行文件失败的话，就会执行到这里）
722 }
757 /*
758 * These are the functions used to load a.out style executables and shared
759 * libraries. There is no binary dependent code anywhere else.
760 */
761
! 该函数用于加载 a.out 类型的二进制文件格式和共享库。没有任何的

```

! 二进制依赖代码  
 ✗ bprm=可执行文件头指针  
 ✗ regs=各个寄存器值的结构指针

```

762 int load_aout_binary(struct linux_binprm * bprm, struct pt_regs * regs)
763 {
764     struct exec ex;
765     struct file * file;
766     int fd, error;
767     unsigned long p = bprm->p;
768
769     ex = *((struct exec *) bprm->buf);           /* exec-header */
    ! 取得执行文件的头
770     if ((N_MAGIC(ex) != ZMAGIC && N_MAGIC(ex) != OMAGIC &&
771         N_MAGIC(ex) != QMAGIC) ||
772         ex.a_trsize || ex.a_drsizel ||
773         bprm->inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
774         return -ENOEXEC;
    ! 对可执行文件的头信息进行判断，这里不做具体的分析了有兴趣的读者可以
    ! 自己分析
775     }
776
777     if (N_MAGIC(ex) == ZMAGIC &&
778         (N_TXTOFF(ex) < bprm->inode->i_sb->s_blocksize)) {
779         printk("N_TXTOFF < BLOCK_SIZE. Please convert binary.");
780         return -ENOEXEC;
    ! 如果可以执行文件头部长度小于 I 节点的中定义的尺寸大小，则打印
    ! 错误提示后，返回-ENOEXEC
781     }
782
783     if (N_TXTOFF(ex) != BLOCK_SIZE && N_MAGIC(ex) == ZMAGIC) {
784         printk("N_TXTOFF != BLOCK_SIZE. See a.out.h.");
    ! 如果执行文件头部长度不等于 1K 并且格式标志也不对，则打印错误提示
    ! 后，返回-ENOEXEC
785         return -ENOEXEC;
786     }
787
788     /* OK, This is the point of no return */
    ! 好的，该函数没有返回
789     flush_old_exec(bprm);
790
791     current->end_code = N_TXTADDR(ex) + ex.a_text;
    ! 设置当前进程代码段的大小
792     current->end_data = ex.a_data + current->end_code;
    ! 设置当前进程数据段的大小

```

```

793     current->start_brk = current->brk = current->end_data;
    ! 设置当前进程brk段的大小
794     current->start_code += N_TXTADDR(ex);
    ! 设置start_code
795     current->rss = 0;
796     current->suid = current->euid = bprm->e_uid;
797     current->mmap = NULL;
798     current->executable = NULL; /* for OMAGIC files */
799     current->sgid = current->egid = bprm->e_gid;
    ! 设置各个值
800     if (N_MAGIC(ex) == OMAGIC) {
    ! 如果代码和数据段是紧跟在头部的后面的则做下面的代码
801         do_mmap(NULL, 0, ex.a_text+ex.a_data,
802             PROT_READ|PROT_WRITE|PROT_EXEC,
803             MAP_FIXED|MAP_PRIVATE, 0);
804
804         read_exec(bprm->inode, 32, (char *) 0, ex.a_text+ex.a_data);
805     } else {
    ! 否则作如下的代码
806         if (ex.a_text & 0xffff || ex.a_data & 0xffff)
807             printf("%s: executable not page aligned\n", current->comm);
    ! 如果，代码段和数据段没有按页对其，则打印提示消息
808
809         fd = open_inode(bprm->inode, O_RDONLY);
    ! 以只读方式打开节点，定义于本文件
810
811         if (fd < 0)
812             return fd;
    ! 如果返回的fd小于0（说明没有找到空闲的文件描述符值），则直接返回
813         file = current->filp[fd];
    ! 让file指向文件描述符指针
814         if (!file->f_op || !file->f_op->mmap) {
815             sys_close(fd);
    ! 如果文件操作函数为空或者 mmap 为空，则关闭上面打开的文件描述符值
    ! 定义于Fs/open.c
816             do_mmap(NULL, 0, ex.a_text+ex.a_data,
817                 PROT_READ|PROT_WRITE|PROT_EXEC,
818                 MAP_FIXED|MAP_PRIVATE, 0);
819             read_exec(bprm->inode, N_TXTOFF(ex),
820                 (char *) N_TXTADDR(ex), ex.a_text+ex.a_data);
821             goto beyond_if;
    ! 跳转到beyond_if
822         }
823         error = do_mmap(file, N_TXTADDR(ex), ex.a_text,

```

```

824             PROT_READ | PROT_EXEC,
825             MAP_FIXED | MAP_SHARED, N_TXTOFF(ex));
     ! 重新映射该文件
826
827         if (error != N_TXTADDR(ex)) {
     ! 如果返回值不等于文件代码段加载到内存后的地址
828             sys_close(fd);
     ! 则关闭描述符, 定义于Fs/open.c
829             send_sig(SIGSEGV, current, 0);
     ! 给当前进程发送SIGSEGV信号, 定义Kernel/exit.c
830             return 0;
     ! 返回 0
831         };
832
833         error = do_mmap(file, N_TXTADDR(ex) + ex.a_text, ex.a_data,
834             PROT_READ | PROT_WRITE | PROT_EXEC,
835             MAP_FIXED | MAP_PRIVATE, N_TXTOFF(ex) + ex.a_text);
     ! 否则, 重新映射
836         sys_close(fd);
     ! 关闭打开的文件描述符, 定义于Fs/open.c
837         if (error != N_TXTADDR(ex) + ex.a_text) {
838             send_sig(SIGSEGV, current, 0);
     ! 如果返回值不等于文件代码段加载到内存后的地址加上可执行程序代码段的
     ! 大小, 则发送信号后, 返回
839         return 0;
840     };
841         current->executable = bprm->inode;
     ! 让executable指向bprm->inode
842         bprm->inode->i_count++;
     ! 让bprm->inode计数加一
843     }
844 beyond_if:
845     sys_brk(current->brk+ex.a_bss);
846
847     p += change_ldt(ex.a_text,bprm->page);
     ! 修改ldt表中描述符基地址和段限长
848     p -= MAX_ARG_PAGES*PAGE_SIZE;
     ! 将参数和环境空间页面放置在数据段末段。
849     p = (unsigned long) create_tables((char *)p,bprm->argc,bprm->envc,0);
     ! 在新的用户堆栈中创建环境和参数变量指针表, 并返回该堆栈指针, 定义
     ! 于本文件
850     current->start_stack = p;
     ! 让新任务的堆栈指向p
851     regs->eip = ex.a_entry;      /* eip, magic happens :-) */

```

```

        ! eip, 魔法起作用了

    ! 设置eip为可执行文件的入口点
852     regs->esp = p;           /* stack pointer */
        ! 栈指针

    ! 设置堆栈指针为p
853     if (current->flags & PF_PTRACED)
854         send_sig(SIGTRAP, current, 0);
        ! 如果当前进程被置了 PF_PTRACED 位，则发送 SIGTRAP 信号，定义
        ! 于Kernel/exit.c
855     return 0;
        ! 返回 0
856 }
857
858
859 int load_aout_library(int fd)
860 {
861     struct file * file;
862     struct exec ex;
863     struct inode * inode;
864     unsigned int len;
865     unsigned int bss;
866     unsigned int start_addr;
867     int error;
868
869     file = current->filp[fd];
870     inode = file->f_inode;
871
872     set_fs(KERNEL_DS);
873     if (file->f_op->read(inode, file, (char *) &ex, sizeof(ex)) != sizeof(ex)) {
874         return -EACCES;
875     }
876     set_fs(USER_DS);
877
878     /* We come in here for the regular a.out style of shared libraries */
879     if ((N_MAGIC(ex) != ZMAGIC && N_MAGIC(ex) != QMAGIC) || ex.a_trsize ||
880         ex.a_drsiz || ((ex.a_entry & 0xffff) && N_MAGIC(ex) == ZMAGIC) ||
881         inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
882         return -ENOEXEC;
883     }
884     if (N_MAGIC(ex) == ZMAGIC && N_TXTOFF(ex) &&
885         (N_TXTOFF(ex) < inode->i_sb->s_blocksize)) {
886         printk("N_TXTOFF < BLOCK_SIZE. Please convert library\n");
887         return -ENOEXEC;
888     }

```

```

889
890     if (N_FLAGS(ex)) return -ENOEXEC;
891
892     /* For QMAGIC, the starting address is 0x20 into the page. We mask
893 this off to get the starting address for the page */
894
895     start_addr = ex.a_entry & 0xfffff000;
896
897     /* Now use mmap to map the library into memory. */
898     error = do_mmap(file, start_addr, ex.a_text + ex.a_data,
899                           PROT_READ | PROT_WRITE | PROT_EXEC, MAP_FIXED |
900                           MAP_PRIVATE,
901                           N_TXTOFF(ex));
902     if (error != start_addr)
903         return error;
904     len = PAGE_ALIGN(ex.a_text + ex.a_data);
905     bss = ex.a_text + ex.a_data + ex.a_bss;
906     if (bss > len)
907         do_mmap(NULL, start_addr + len, bss - len,
908                   PROT_READ|PROT_WRITE|PROT_EXEC,
909                   MAP_PRIVATE|MAP_FIXED, 0);
910     return 0;
911 }

```

## Fs/file\_table.c (部分代码)

! 从全局描述符表取个新的描述符

```

68 struct file * get_empty_file(void)
69 {
70     int i;
71     struct file * f;
72
73     if (!first_file)
74         grow_files();

```

! 如果全局描述符表为空，则分配之（这里不再注释）

```

75 repeat:
76     for (f = first_file, i=0; i < nr_files; i++, f = f->f_next)
77         if (!f->f_count) {
78             remove_file_free(f);

```

```

79         memset(f,0,sizeof(*f));
80         put_last_free(f);
81         f->f_count = 1;
82         return f;
! 描述符的引用计数赋 1 后，返回新描述符
83     }
84     if (nr_files < NR_FILE) {
85         grow_files();
86         goto repeat;
87     }
88     return NULL;
89 }
90

```

## Fs/namei.c

```

1 /*
2  *  linux/fs/namei.c
3  *
4  *  Copyright (C) 1991, 1992  Linus Torvalds
5  */
6
7 /*
8  *  Some corrections by tytso.
9  */
10
11 #include <asm/segment.h>
12
13 #include <linux/errno.h>
14 #include <linux/sched.h>
15 #include <linux/kernel.h>
16 #include <linux/string.h>
17 #include <linux/fcntl.h>
18 #include <linux/stat.h>
19
20 #define ACC_MODE(x) ("\\000\\004\\002\\006"[x)&O_ACCMODE])
21
22 /*
23  * In order to reduce some races, while at the same time doing additional

```

```

24 * checking and hopefully speeding things up, we copy filenames to the
25 * kernel data space before using them..
26 *
27 * POSIX.1 2.4: an empty pathname is invalid (ENOENT).
28 */
    ! 为了减少一些竞争条件，我们做了一些附加的检查并且希望它的执行
    ! 速度得到提升，我们在使用前把文件名拷贝到核心空间。
    ! 把文件名拷贝到核心空间
    × filename=被拷贝文件名的指针
    × result=返回结果的指针的指针
29 int getname(const char * filename, char **result)
30 {
31     int error;
32     unsigned long i, page;
33     char * tmp, c;
34
35     i = (unsigned long) filename;
    ! 把 filename 的地址强转成无符号整型后给 i
36     if (i || i >= TASK_SIZE)
37         return -EFAULT;
    ! 如果 i 为 0 或者大于 TASK_SIZE (即本身已经在核心空间了) 返回
    ! -EFAULT。
38     i = TASK_SIZE - i;
39     error = -EFAULT;
40     if (i > PAGE_SIZE) {
41         i = PAGE_SIZE;
42         error = -ENAMETOOLONG;
43     }
    ! 如果 i 大于 4K，则设置 i=4K
44     c = get_fs_byte(filename++);
    ! 从用户空间取一个字符到 c 中，请注意 (fs=用户空间选择子)
45     if (!c)
46         return -ENOENT;
    ! 如果为 0，则返回 -ENOENT
47     if (!(page = get_free_page(GFP_KERNEL)))
48         return -ENOMEM;
    ! 否则，申请一页内存，定义于 MM/swap.c
49     *result = tmp = (char *) page;
    ! 让 result 指向申请到的内存，这样该函数返回后，调用方可以通过 result
    ! 读取申请的内存
50     while (--i) {
51         *(tmp++) = c;
52         c = get_fs_byte(filename++);
53         if (!c) {

```

```

54                     *tmp = '\0';
55                     return 0;
56                 }
! 循环读取用户空间中的文件名字符，最后返回 0
57             }
58             free_page(page);
59             return error;
! 否则，释放上面申请的内存后，返回错误值
! 定义于 MM/swap.c
60 }
61

```

! 释放为存放文件名分配的内存  
 ✗ name=指向要释放内存的指针

```

62 void putname(char * name)
63 {
64     free_page((unsigned long) name);
! 定义于 Mm/swap.c
65 }
66
67 /*
68 *      permission()
69 *
70 * is used to check for read/write/execute permissions on a file.
71 * I don't know if we should look at just the euid or both euid and
72 * uid, but that should be easily changed.
73 */
! permission()
! 该函数用于检测一个文件的读、写以及执行权限，我不知道是否需要
! 检查 euid，还是需要检查 euid 和 uid 两者，不过这很容易修改。
! 检测文件的访问许可权限
✗ inode=文件对应的 I 节点
✗ mask=属性屏蔽码
```

74 int permission(struct inode \* inode,int mask)

```

75 {
76     int mode = inode->i_mode;
77
78     if (inode->i_op && inode->i_op->permission)
79         return inode->i_op->permission(inode, mask);
! 如果 I 节点的操作函数指针不为空并且其 permission 不为空，则调用真正的
! 文件访问许可权限，有兴趣的读者可以自己看看
80     else if (current->euid == inode->i_uid)
81         mode >>= 6;
```

```

    ! 如果进程的 euid 等于该 I 节点的 uid，则取文件宿主的用户访问权限
82        else if (in_group_p(inode->i_gid))
83            mode >>= 3;
    ! 如果进程的有效组 id 于 I 节点的组 id 相同，则去组用户的访问权限
84        if (((mode & mask & 0007) == mask) || suser())
85            return 1;
    ! 如果上面所取的访问权限于屏蔽码相同或者是超级用户，则返回 1
86        return 0;
    ! 否则返回 0
87    }
88
89/*
90 * lookup() looks up one part of a pathname, using the fs-dependent
91 * routines (currently minix_lookup) for it. It also checks for
92 * fathers (pseudo-roots, mount-points)
93 */
    ! lookup()查找 pathname 中的一部分，它使用了 fs-dependent 程序（当前是
    ! minix_lookup）它也检查父进程
    × dir=查找时的目录起始 I 节点
    × name=原始的文件名（即没有跳过 ‘/’ 前）
    × len=为查找 ‘/’ 时，所跳过的长度
    × result=函数返回值放在里面
94 int lookup(struct inode * dir,const char * name, int len,
95             struct inode ** result)
96 {
97     struct super_block * sb;
98     int perm;
99
100    *result = NULL;
101    if (!dir)
102        return -ENOENT;
    ! 如果 dir 为空，则返回-ENOENT
103 /* check permissions before traversing mount-points */
104     perm = permission(dir,MAY_EXEC);
    ! 检查文件许可权限，定义于本文件
105    if (len==2 && name[0] == '.' && name[1] == '.') {
106        if (dir == current->root) {
107            *result = dir;
108            return 0;
    ! 如果长度为 2，并且形如 ‘..’，并且等于当前进程的根目录的 I 节点
    ! 则设置后，返回
109    } else if ((sb = dir->i_sb) && (dir == sb->s_mounted)) {
110        sb = dir->i_sb;
111        iput(dir);

```

```

112                     dir = sb->s_covered;
113                     if (!dir)
114                         return -ENOENT;
115                     dir->i_count++;
! 如果当前 I 节点的超级块存在并且是根文件相同安装点，则设置之
116                 }
117             }
118             if (!dir->i_op || !dir->i_op->lookup) {
119                 iput(dir);
120                 return -ENOTDIR;
121             }
122             if (!perm) {
123                 iput(dir);
124                 return -EACCES;
125             }
126             if (!len) {
127                 *result = dir;
128                 return 0;
129             }
130             return dir->i_op->lookup(dir, name, len, result);
! 调用真正的查找函数，这里不做注释
131 }
132

```

```

133 int follow_link(struct inode * dir, struct inode * inode,
134                  int flag, int mode, struct inode ** res_inode)
135 {
136     if (!dir || !inode) {
137         iput(dir);
138         iput(inode);
139         *res_inode = NULL;
140         return -ENOENT;
141     }
142     if (!inode->i_op || !inode->i_op->follow_link) {
143         iput(dir);
144         *res_inode = inode;
145         return 0;
146     }
147     return inode->i_op->follow_link(dir, inode, flag, mode, res_inode);
148 }
149
150 /*
151 *      dir_namei()

```

```

152  *
153  * dir_namei() returns the inode of the directory of the
154  * specified name, and the name within that directory.
155  */
! dir_namei()
! dir_namei()函数返回指定目录名的 I 节点指针，以及在最顶层目录的名称
× pathname=目录路径名
× namelen=路径名长度
× name=指向要返回的顶层目录名
× base=查询时的起始 I 节点
× res_inode=指向要返回最顶层目录 I 节点指针
156 static int dir_namei(const char * pathname, int * namelen, const char ** name,
157                     struct inode * base, struct inode ** res_inode)
158 {
159     char c;
160     const char * thisname;
161     int len,error;
162     struct inode * inode;
163
164     *res_inode = NULL;
165     if (!base) {
166         base = current->pwd;
167         base->i_count++;
!
! 如果查询时的起始 I 节点为空，则设置 base 指向当前进程的工作目录
! (即从当前进程开始查询，如果传入的 pathname 不是从"/"开始的话)
!, 同时！让其引用计数加一
168 }
169     if ((c = *pathname) == '/') {
170         iput(base);
171         base = current->root;
172         pathname++;
173         base->i_count++;
174     }
!
! 如果当前传入的文件名中有 ‘/’ 的话，则重新设置查询起始值，同时释放
! 有可能在 165 行到 167 设置的 I 节点 (iput 定义于 Fs/inode.c)
175     while (1) {
176         thisname = pathname;
177         for(len=0;(c = *(pathname++))&&(c != '/');len++)
178             /* nothing */;
!
! 从 pathname 中找到第一个'/'，并让 len=跳过的字符个数
179         if (!c)
180             break;
!
! 如果在找到后字符 c 为 0，则跳出循环(路径形如： /usr/src/)
181         base->i_count++;

```

```

182             error = lookup(base,thisname,len,&inode);
    ! 查找 thisname 的部分，通过长度 len，定义于本文件
183             if (error) {
184                 iput(base);
185                 return error;
    ! 如果有错误发生，则释放该 I 节点后返回错误码
186             }
187             error = follow_link(base,inode,0,0,&base);
188             if (error)
189                 return error;
190         }
191         if (!base->i_op || !base->i_op->lookup) {
192             iput(base);
193             return -ENOTDIR;
194         }
195         *name = thisname;
196         *namelen = len;
197         *res_inode = base;
    ! 设置要返回的值
198         return 0;
    ! 返回 0
199     }
200

```

```

201 static int namei(const char * pathname, struct inode * base,
202                     int follow_links, struct inode ** res_inode)
203 {
204     const char * basename;
205     int namelen,error;
206     struct inode * inode;
207
208     *res_inode = NULL;
209     error = dir_namei(pathname,&namelen,&basename,base,&base);
210     if (error)
211         return error;
212     base->i_count++; /* lookup uses up base */
213     error = lookup(base,basename,namelen,&inode);
214     if (error) {
215         iput(base);
216         return error;
217     }
218     if (follow_links) {
219         error = follow_link(base,inode,0,0,&inode);

```

```

220         if (error)
221                 return error;
222     } else
223         input(base);
224     *res_inode = inode;
225     return 0;
226 }
227
228 int lnamei(const char * pathname, struct inode ** res_inode)
229 {
230     int error;
231     char * tmp;
232
233     error = getname(pathname,&tmp);
234     if (!error) {
235         error = namei(tmp,NULL,0,res_inode);
236         putname(tmp);
237     }
238     return error;
239 }
240
241 /*
242 *      namei()
243 *
244 * is used by most simple commands to get the inode of a specified name.
245 * Open, link etc use their own routines, but this is enough for things
246 * like 'chmod' etc.
247 */
248 int namei(const char * pathname, struct inode ** res_inode)
249 {
250     int error;
251     char * tmp;
252
253     error = getname(pathname,&tmp);
254     if (!error) {
255         error = namei(tmp,NULL,1,res_inode);
256         putname(tmp);
257     }
258     return error;
259 }
260
261 /*
262 *      open_namei()
263 *

```

```

264 * namei for open - this is in fact almost the whole open-routine.
265 *
266 * Note that the low bits of "flag" aren't the same as in the open
267 * system call - they are 00 - no permissions needed
268 * 01 - read permission needed
269 * 10 - write permission needed
270 * 11 - read/write permissions needed
271 * which is a lot more logical, and also allows the "no perm" needed
272 * for symlinks (where the permissions are checked later).
273 */
!
```

! open\_namei ()

! open 函数使用的函数—这其实几乎是完整的打开文件程序

! 注意标志位的低位与 open 系统函数中标志位不同—00 不需要权限

! 01—读权限

! 10—写权限

! 11—读写权限

```

274 int open_namei(const char * pathname, int flag, int mode,
275         struct inode ** res_inode, struct inode * base)
276 {
277     const char * basename;
278     int namelen,error;
279     struct inode * dir, *inode;
280     struct task_struct ** p;
281
282     mode &= S_IALLUGO & ~current->umask;
283     mode |= S_IFREG;
284     error = dir_namei(pathname,&namelen,&basename,base,&dir);
    ! 根据路径名寻找对应的 I 节点，以及最顶端文件名及其长度，定义于本文件
285     if (error)
286         return error;
    ! 如果有错误发生，则直接返回错误
287     if (!namelen) { /* special case: '/usr/' etc */
288         if (flag & 2) {
289             iput(dir);
290             return -EISDIR;
    ! 如果是权限只写的，则释放该 I 节点，返回-EISDIR
291     }
292     /* thanks to Paul Pluzhnikov for noticing this was missing.. */
    ! 谢谢 Paul Pluzhnikov 注意到了下面的这些
293     if (!permission(dir,ACC_MODE(flag))) {
294         iput(dir);
295         return -EACCES;
296     }
    *res_inode=dir;

```

```

! 设置要返回的 I 节点
298         return 0;
! 返回 0
299     }
300     dir->i_count++;           /* lookup eats the dir */
301     if (flag & O_CREAT) {
! 创建文件标志
302         down(&dir->i_sem);
303         error = lookup(dir, basename, namelen, &inode);
! 定义于本文件
304         if (!error) {
305             if (flag & O_EXCL) {
306                 iput(inode);
307                 error = -EEXIST;
! 如果标志是 O_EXCL 的，则返回-EEXIST
308             }
309             } else if (!permission(dir, MAY_WRITE | MAY_EXEC))
310                 error = -EACCES;
311             else if (!dir->i_op || !dir->i_op->create)
312                 error = -EACCES;
313             else if (IS_RDONLY(dir))
314                 error = -EROFS;
! 从 308 行到这检测不同的情况
315         else {
316             dir->i_count++;           /* create eats the dir */
317             error =
dir->i_op->create(dir, basename, namelen, mode, res_inode);
318             up(&dir->i_sem);
319             iput(dir);
320             return error;
321         }
322         up(&dir->i_sem);
323     } else
324         error = lookup(dir, basename, namelen, &inode);
325     if (error) {
326         iput(dir);
327         return error;
328     }
329     error = follow_link(dir, inode, flag, mode, &inode);
330     if (error)
331         return error;
332     if (S_ISDIR(inode->i_mode) && (flag & 2)) {
333         iput(inode);
334         return -EISDIR;

```

```

335     }
336     if (!permission(inode,ACC_MODE(flag))) {
337         iput(inode);
338         return -EACCES;
339     }
340     if (S_ISBLK(inode->i_mode) || S_ISCHR(inode->i_mode)) {
341         if (IS_NODEV(inode)) {
342             iput(inode);
343             return -EACCES;
344         }
345     } else {
346         if (IS_RDONLY(inode) && (flag & 2)) {
347             iput(inode);
348             return -EROFS;
349         }
350     }
351     if ((inode->i_count > 1) && (flag & 2)) {
352         for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
353             struct vm_area_struct * mpnt;
354             if (!*p)
355                 continue;
356             if (inode == (*p)->executable) {
357                 iput(inode);
358                 return -ETXTBSY;
359             }
360             for(mpnt = (*p)->mmap; mpnt; mpnt = mpnt->vm_next) {
361                 if (mpnt->vm_page_prot & PAGE_RW)
362                     continue;
363                 if (inode == mpnt->vm_inode) {
364                     iput(inode);
365                     return -ETXTBSY;
366                 }
367             }
368         }
369     }
370     if (flag & O_TRUNC) {
371         inode->i_size = 0;
372         if (inode->i_op && inode->i_op->truncate)
373             inode->i_op->truncate(inode);
374         if ((error = notify_change(NOTIFY_SIZE, inode))) {
375             iput(inode);
376             return error;
377         }
378         inode->i_dirt = 1;

```

```

379         }
380         *res_inode = inode;
381         return 0;
382     }
383
384 int do_mknod(const char * filename, int mode, dev_t dev)
385 {
386     const char * basename;
387     int namelen, error;
388     struct inode * dir;
389
390     mode &= ~current->umask;
391     error = dir_namei(filename, &namelen, &basename, NULL, &dir);
392     if (error)
393         return error;
394     if (!namelen) {
395         iput(dir);
396         return -ENOENT;
397     }
398     if (IS_RDONLY(dir)) {
399         iput(dir);
400         return -EROFS;
401     }
402     if (!permission(dir, MAY_WRITE | MAY_EXEC)) {
403         iput(dir);
404         return -EACCES;
405     }
406     if (!dir->i_op || !dir->i_op->mknod) {
407         iput(dir);
408         return -EPERM;
409     }
410     down(&dir->i_sem);
411     error = dir->i_op->mknod(dir, basename, namelen, mode, dev);
412     up(&dir->i_sem);
413     return error;
414 }
415
416 asmlinkage int sys_mknod(const char * filename, int mode, dev_t dev)
417 {
418     int error;
419     char * tmp;
420
421     if (S_ISDIR(mode) || (!S_ISFIFO(mode) && !suser()))
422         return -EPERM;

```

```

423     switch (mode & S_IFMT) {
424         case 0:
425             mode |= S_IFREG;
426             break;
427         case S_IFREG: case S_IFCHR: case S_IFBLK: case S_IFIFO:
428             break;
429         default:
430             return -EINVAL;
431     }
432     error = getname(filename,&tmp);
433     if (!error) {
434         error = do_mknod(tmp,mode,dev);
435         putname(tmp);
436     }
437     return error;
438 }
439
440 static int do_mkdir(const char * pathname, int mode)
441 {
442     const char * basename;
443     int namelen, error;
444     struct inode * dir;
445
446     error = dir_namei(pathname,&namelen,&basename,NULL,&dir);
447     if (error)
448         return error;
449     if (!namelen) {
450         iput(dir);
451         return -ENOENT;
452     }
453     if (IS_RDONLY(dir)) {
454         iput(dir);
455         return -EROFS;
456     }
457     if (!permission(dir,MAY_WRITE | MAY_EXEC)) {
458         iput(dir);
459         return -EACCES;
460     }
461     if (!dir->i_op || !dir->i_op->mkdir) {
462         iput(dir);
463         return -EPERM;
464     }
465     down(&dir->i_sem);
466     error = dir->i_op->mkdir(dir,basename,namelen,mode);

```

```

467         up(&dir->i_sem);
468         return error;
469     }
470
471 asmlinkage int sys_mkdir(const char * pathname, int mode)
472 {
473     int error;
474     char * tmp;
475
476     error = getname(pathname,&tmp);
477     if (!error) {
478         error = do_mkdir(tmp,mode);
479         putname(tmp);
480     }
481     return error;
482 }
483
484 static int do_rmdir(const char * name)
485 {
486     const char * basename;
487     int namelen, error;
488     struct inode * dir;
489
490     error = dir_namei(name,&namelen,&basename,NULL,&dir);
491     if (error)
492         return error;
493     if (!namelen) {
494         iput(dir);
495         return -ENOENT;
496     }
497     if (IS_RDONLY(dir)) {
498         iput(dir);
499         return -EROFS;
500     }
501     if (!permission(dir,MAY_WRITE | MAY_EXEC)) {
502         iput(dir);
503         return -EACCES;
504     }
505     if (!dir->i_op || !dir->i_op->rmdir) {
506         iput(dir);
507         return -EPERM;
508     }
509     return dir->i_op->rmdir(dir, basename, namelen);
510 }

```

```

511
512 asmlinkage int sys_rmdir(const char * pathname)
513 {
514     int error;
515     char * tmp;
516
517     error = getname(pathname,&tmp);
518     if (!error) {
519         error = do_rmdir(tmp);
520         putname(tmp);
521     }
522     return error;
523 }
524
525 static int do_unlink(const char * name)
526 {
527     const char * basename;
528     int namelen, error;
529     struct inode * dir;
530
531     error = dir_namei(name,&namelen,&basename,NULL,&dir);
532     if (error)
533         return error;
534     if (!namelen) {
535         iput(dir);
536         return -EPERM;
537     }
538     if (IS_RDONLY(dir)) {
539         iput(dir);
540         return -EROFS;
541     }
542     if (!permission(dir,MAY_WRITE | MAY_EXEC)) {
543         iput(dir);
544         return -EACCES;
545     }
546     if (!dir->i_op || !dir->i_op->unlink) {
547         iput(dir);
548         return -EPERM;
549     }
550     return dir->i_op->unlink(dir,basename,namelen);
551 }
552
553 asmlinkage int sys_unlink(const char * pathname)
554 {

```

```

555     int error;
556     char * tmp;
557
558     error = getname(pathname,&tmp);
559     if (!error) {
560         error = do_unlink(tmp);
561         putname(tmp);
562     }
563     return error;
564 }
565
566 static int do_symlink(const char * oldname, const char * newname)
567 {
568     struct inode * dir;
569     const char * basename;
570     int namelen, error;
571
572     error = dir_namei(newname,&namelen,&basename,NULL,&dir);
573     if (error)
574         return error;
575     if (!namelen) {
576         iput(dir);
577         return -ENOENT;
578     }
579     if (IS_RDONLY(dir)) {
580         iput(dir);
581         return -EROFS;
582     }
583     if (!permission(dir,MAY_WRITE | MAY_EXEC)) {
584         iput(dir);
585         return -EACCES;
586     }
587     if (!dir->i_op || !dir->i_op->symlink) {
588         iput(dir);
589         return -EPERM;
590     }
591     down(&dir->i_sem);
592     error = dir->i_op->symlink(dir,basename,namelen,oldname);
593     up(&dir->i_sem);
594     return error;
595 }
596
597 asmlinkage int sys_symlink(const char * oldname, const char * newname)
598 {

```

```

599     int error;
600     char * from, * to;
601
602     error = getname(oldname,&from);
603     if (!error) {
604         error = getname(newname,&to);
605         if (!error) {
606             error = do_symlink(from,to);
607             putname(to);
608         }
609         putname(from);
610     }
611     return error;
612 }
613
614 static int do_link(struct inode * oldinode, const char * newname)
615 {
616     struct inode * dir;
617     const char * basename;
618     int namelen, error;
619
620     error = dir_namei(newname,&namelen,&basename,NULL,&dir);
621     if (error) {
622         iput(oldinode);
623         return error;
624     }
625     if (!namelen) {
626         iput(oldinode);
627         iput(dir);
628         return -EPERM;
629     }
630     if (IS_RDONLY(dir)) {
631         iput(oldinode);
632         iput(dir);
633         return -EROFS;
634     }
635     if (dir->i_dev != oldinode->i_dev) {
636         iput(dir);
637         iput(oldinode);
638         return -EXDEV;
639     }
640     if (!permission(dir,MAY_WRITE | MAY_EXEC)) {
641         iput(dir);
642         iput(oldinode);

```

```

643             return -EACCES;
644         }
645         if (!dir->i_op || !dir->i_op->link) {
646             iput(dir);
647             iput(oldinode);
648             return -EPERM;
649         }
650         down(&dir->i_sem);
651         error = dir->i_op->link(oldinode, dir, basename, namelen);
652         up(&dir->i_sem);
653         return error;
654     }
655
656 asmlinkage int sys_link(const char * oldname, const char * newname)
657 {
658     int error;
659     char * to;
660     struct inode * oldinode;
661
662     error = namei(oldname, &oldinode);
663     if (error)
664         return error;
665     error = getname(newname,&to);
666     if (error) {
667         iput(oldinode);
668         return error;
669     }
670     error = do_link(oldinode,to);
671     putname(to);
672     return error;
673 }
674
675 static int do_rename(const char * oldname, const char * newname)
676 {
677     struct inode * old_dir, * new_dir;
678     const char * old_base, * new_base;
679     int old_len, new_len, error;
680
681     error = dir_namei(oldname,&old_len,&old_base,NULL,&old_dir);
682     if (error)
683         return error;
684     if (!permission(old_dir,MAY_WRITE | MAY_EXEC)) {
685         iput(old_dir);
686         return -EACCES;

```

```

687      }
688      if (!old_len || (old_base[0] == '.' &&
689          (old_len == 1 || (old_base[1] == '.' &&
690              old_len == 2)))) {
691          iput(old_dir);
692          return -EPERM;
693      }
694      error = dir_namei(newname, &new_len, &new_base, NULL, &new_dir);
695      if (error) {
696          iput(old_dir);
697          return error;
698      }
699      if (!permission(new_dir, MAY_WRITE | MAY_EXEC)) {
700          iput(old_dir);
701          iput(new_dir);
702          return -EACCES;
703      }
704      if (!new_len || (new_base[0] == '.' &&
705          (new_len == 1 || (new_base[1] == '.' &&
706              new_len == 2)))) {
707          iput(old_dir);
708          iput(new_dir);
709          return -EPERM;
710      }
711      if (new_dir->i_dev != old_dir->i_dev) {
712          iput(old_dir);
713          iput(new_dir);
714          return -EXDEV;
715      }
716      if (IS_RDONLY(new_dir) || IS_RDONLY(old_dir)) {
717          iput(old_dir);
718          iput(new_dir);
719          return -EROFS;
720      }
721      if (!old_dir->i_op || !old_dir->i_op->rename) {
722          iput(old_dir);
723          iput(new_dir);
724          return -EPERM;
725      }
726      down(&new_dir->i_sem);
727      error = old_dir->i_op->rename(old_dir, old_base, old_len,
728          new_dir, new_base, new_len);
729      up(&new_dir->i_sem);
730      return error;

```

```

731 }
732
733 asmlinkage int sys_rename(const char * oldname, const char * newname)
734 {
735     int error;
736     char * from, * to;
737
738     error = getname(oldname,&from);
739     if (!error) {
740         error = getname(newname,&to);
741         if (!error) {
742             error = do_rename(from,to);
743             putname(to);
744         }
745         putname(from);
746     }
747     return error;
748 }
749

```

## Fs/buffer.c (部分代码)

88 /\* Call sync\_buffers with wait!=0 to ensure that the call does not  
 89 return until all buffer writes have completed. Sync() may return  
 90 before the writes have finished; fsync() may not. \*/

91

! wait!=0 时调用 sync\_buffers 直到所有的缓冲被写完才返回，Sync()或许会在  
! 写完前返回；fsync()却不是。

✗ dev=设备号

✗ wait=是否请求

92 static int sync\_buffers(dev\_t dev, int wait)

93 {

94 int i, retry, pass = 0, err = 0;

95 struct buffer\_head \* bh;

96

97 /\* One pass for no-wait, three for wait:

98 0} write out all dirty, unlocked buffers;

99 1} write out all dirty buffers, waiting if locked;

100 2} wait for completion by waiting for all buffers to unlock.

101 \*/

! 1 没有等待，3 有等待  
 0} 写所有的脏数据及被解锁的缓冲；  
 1} 写所有的脏的缓冲，等待其锁定；  
 2} 等待其完成，直到所有的缓冲解锁

102 repeat:

```
103     retry = 0;
104     bh = free_list;
105     for (i = nr_buffers*2 ; i-- > 0 ; bh = bh->b_next_free) {
106         if (dev && bh->b_dev != dev)
107

```

! 如果设备号存在并且对应的缓冲区所在的设备号于传入的设备号  
 ! 不等（即并不是要同步的设备），则跳过下面代码

108 #ifdef 0 /\* Disable bad-block debugging code \*/

! 从该行到 113，用于调试时用

```
109     if (bh->b_req && !bh->b_lock &&
110         !bh->b_dirt && !bh->b_uptodate)
111         printk ("Warning (IO error) - orphaned block %08x on %04x\n",
112             bh->b_blocknr, bh->b_dev);
113 #endif
```

```
114     if (bh->b_lock)
115     {
```

! 如果，缓冲区被锁

```
116         /* Buffer is locked; skip it unless wait is
117 requested AND pass > 0. */
```

! 缓冲被锁；跳过下面代码除非请求&pass>0

```
118         if (!wait || !pass) {
119             retry = 1;
120             continue;
121         }
122         wait_on_buffer (bh);
```

! 等待缓冲解锁

```
123     }
124     /* If an unlocked buffer is not uptodate, there has been
125 an IO error. Skip it. */
126     if (wait && bh->b_req && !bh->b_lock &&
```

```
u127         !bh->b_dirt && !bh->b_uptodate)
u128     {
u129         err = 1;
u130         continue;
u131     }
u132     /* Don't write clean buffers. Don't write ANY buffers
u133 on the third pass. */
```

! 不写干净的缓冲，不写任何的缓冲在第 3 种情况下

```

134         if (!bh->b_dirt || pass>=2)
135             continue;
136         bh->b_count++;
137         ll_rw_block(WRITE, 1, &bh);
    ! 将高速缓冲写入块设备，定义于Drivers/block/ll_rw_blk.c
138         bh->b_count--;
139         retry = 1;
140     }
141     /* If we are waiting for the sync to succeed, and if any dirty
142    blocks were written, then repeat; on the second pass, only
143    wait for buffers being written (do not pass to write any
144    more buffers on the second pass). */
    ! 假如我们等待sync成功，并且任何脏数据已经被写，那么跳到repeat处；在第二种
    ! 情况，仅仅等待缓冲被写。
145     if (wait && retry && ++pass<=2)
146         goto repeat;
147     return err;
    ! 返回错误值
148 }
149
    ! 对指定的设备进行高速缓冲数据与设备上数据进行同步
150 void sync_dev(dev_t dev)
151 {
152     sync_buffers(dev, 0);
    ! 同步高速缓冲，定义于本文件
153     sync_supers(dev);
    ! 同步超级块，定义于Fs/super.c
154     sync_inodes(dev);
    ! 同步I节点定义于Fs/inode.c
155     sync_buffers(dev, 0);
    ! 同步高速缓冲，定义于本文件
156 }
157
    ! 同步设备和内存高速缓冲中数据
166 asmlinkage int sys_sync(void)
167 {
168     sync_dev(0);
    ! 定义于本文件
169     return 0;
170 }
171
    ! 在高速缓冲种查找给定块设备和指定块的缓冲区
    × dev=块设备
    × block=块编号

```

```

    × size=大小

357 static struct buffer_head * find_buffer(dev_t dev, int block, int size)
358 {
359     struct buffer_head * tmp;
360
361     for (tmp = hash(dev,block) ; tmp != NULL ; tmp = tmp->b_next)
362         if (tmp->b_dev==dev && tmp->b_blocknr==block)
363             if (tmp->b_size == size)
364                 return tmp;
365
    ! 返回找到的缓冲
366
367     else {
        ! 否则，打印错误提示并且返回NULL
368         printk("VFS: Wrong blocksize on device %d/%d\n",
369             MAJOR(dev), MINOR(dev));
370         return NULL;
371     }
372
373 /*
374 * Why like this, I hear you say... The reason is race-conditions.
375 * As we don't lock buffers (unless we are readint them, that is),
376 * something might happen to it while we sleep (ie a read-error
377 * will force it bad). This shouldn't really happen currently, but
378 * the code is ready.
379 */
    ! 为什么会是这个样子？我听见你问...原因是竞争条件。由于我们没有
    ! 对缓冲区上锁（除非我们正在读取他么中的数据），那么当我们（进程）
    ! 睡眠时缓冲区可能会发生一些问题（例如一个读错误讲导致缓冲区出错）。
    ! 目前这种情况是不可能发生的。但是代码已经准备好了。

380 struct buffer_head * get_hash_table(dev_t dev, int block, int size)
381 {
382     struct buffer_head * bh;
383
384     for (;;) {
385         if (!(bh=find_buffer(dev,block,size)))
386             return NULL;
    ! 在高速缓冲种查找给定块设备和指定块的缓冲区，如果没有找到则返回 NULL
    ! 定义于本文件
387         bh->b_count++;
    ! 引用计数增加一
388         wait_on_buffer(bh);
    ! 等待缓冲解锁
389         if (bh->b_dev == dev && bh->b_blocknr == block && bh->b_size == size)

```

```

390         return bh;
    ! 解锁后，检查该缓冲的正确性，如果正确则返回
391         bh->b_count--;
    ! 否则从新再来一遍
392     }
393 }

```

```

436 /*
437 * Ok, this is getblk, and it isn't very clear, again to hinder
438 * race-conditions. Most of the code is seldom used, (ie repeating),
439 * so it should be much more efficient than it looks.
440 *
441 * The algorithm is changed: hopefully better, and an elusive bug removed.
442 *
443 * 14.02.92: changed it to sync dirty buffers a bit: better performance
444 * when the filesystem starts to get full of dirty blocks (I hope).
445 */

```

！好，下面是 getblk 函数，它的逻辑并不是很清楚，同样也要考虑竞争的条件问题。  
 ！其中的大部分代码很少用到（例如重复操作），因此它应该比看上去的样子有效的多  
 ！算法已经做了改变：希望可以做的更好，并且一个难以捉摸的错误已经去除。  
 ！14.02.92:该块代码已经被修改用于同步脏的缓冲数据：当文件系统开始取的  
 ！所有的脏数据时会更好的性能（我希望）

```

    ! 用于判断缓冲区的修改标志和锁定标志，并且定义修改标志的权要比锁定标志大
446 #define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)
    ! 取高速缓冲中指定的缓冲。
    × dev=块设备
    × block=起始块数
    × size=大小
447 struct buffer_head * getblk(dev_t dev, int block, int size)
448 {
449     struct buffer_head * bh, * tmp;
450     int buffers;
451     static int grow_size = 0;
452
453 repeat:
454     bh = get_hash_table(dev, block, size);
455     if (bh) {
456         if (bh->b_uptodate && !bh->b_dirt)
457             put_last_free(bh);
458         return bh;
459     }

```

！查找 hash 表，如果指定块已经在高速缓冲中，则返回对应的缓冲区指针

```

! 定义于本文件

460     grow_size -= size;
461     if (nr_free_pages > min_free_pages && grow_size <= 0) {
462         if (grow_buffers(GFP_BUFFER, size))
463             grow_size = PAGE_SIZE;
464     }
465     buffers = nr_buffers;
466     bh = NULL;
467
468     for (tmp = free_list; buffers-- > 0 ; tmp = tmp->b_next_free) {
469         if (tmp->b_count || tmp->b_size != size)
470             continue;
        ! 如果高速缓冲区的引用计数不为 0 或者其尺寸并不于传入的值相等，则
        ! 跳过下面的代码
471         if (mem_map[MAP_NR((unsigned long)tmp->b_data)] != 1)
472             continue;
        ! 如果缓冲数据在内存映射位图中并没有被置一，则跳过下面的代码
473         if (!bh || BADNESS(tmp)<BADNESS(bh)) {
474             bh = tmp;
475             if (!BADNESS(tmp))
476                 break;
477         }
        ! 如果缓冲头指针为空，或者 tmp 所指缓冲头的标志权重小于 bh 头标志的权重，则让
        ! bh 指向该 tmp 缓冲区头。如果该 tmp 缓冲头表明缓冲区既没有修改又没有锁定标志
        ! 置位，则说明已为指定设备上的块取得了对应的高速缓冲区，退出循环。
478 #if 0
479     if (tmp->b_dirt) {
480         tmp->b_count++;
481         ll_rw_block(WRITEA, 1, &tmp);
482         tmp->b_count--;
483     }
        ! 从第 478 到这，是用于调试，所以忽略之。
484 #endif
485     }
486
487     if (!bh) {
488         if (nr_free_pages > 5)
489             if (grow_buffers(GFP_BUFFER, size))
490                 goto repeat;
        ! 如果缓冲区正被使用，并且 nr_free_pages>5，则增加可用的空闲高速缓冲。
        ! 跳到repeat，再来一次。
491         if (!grow_buffers(GFP_ATOMIC, size))
492             sleep_on(&buffer_wait);
493         goto repeat;

```

```

494     }
495
496     wait_on_buffer(bh);
    ! 等待缓冲解锁
497     if (bh->b_count || bh->b_size != size)
498         goto repeat;
    ! 如果高速缓冲区的引用计数不为 0 或者其尺寸并不于传入的值相等，则
    ! 跳转到repeat，再来一遍。
499     if (bh->b_dirt) {
500         sync_buffers(0,0);
    ! 同步高速缓冲,定义于本文件
501         goto repeat;
502     }
503 /* NOTE!! While we slept waiting for this block, somebody else might */
504 /* already have added "this" block to the cache. check it */
    ! 注意：当进程为了等待该缓冲块而睡眠时，其他进程可能已经将
    ! 该缓冲块加入高速缓冲中，所以要对此进行检查。
505     if (find_buffer(dev,block,size))
506         goto repeat;)
    ! 在高速缓 hash 表中检查指定缓冲区是否已经被加入。如果是的话，
    ! 就再来一遍。
507 /* OK, FINALLY we know that this buffer is the only one of its kind, */
508 /* and that it's unused (b_count=0), unlocked (b_lock=0), and clean */
    ! 好的，最终我们知道该缓冲区是指定参数的唯一一块，而是还没有
    ! 被使用 (b_count=0) 未被上锁 (b_lock=0) 并且是干净的。
509     bh->b_count=1;
510     bh->b_dirt=0;
511     bh->b_uptodate=0;
512     bh->b_req=0;
    ! 引用计数加一，及修改该缓冲的标志。
513     remove_from_queues(bh);
    ! 从 hash 队列和空闲块链表中移出该缓冲头，让该缓冲区用于指定设备和其上的
    ! 指定块。
514     bh->b_dev=dev;
515     bh->b_blocknr=block;
516     insert_into_queues(bh);
    ! 然后根据新的设备号和块号重新插入空闲链表和 hash 队列新位置处。并
    ! 返回缓冲头指针。
517     return bh;
518 }
    ! 释放指导的缓冲区
    × buf=要释放的缓存区
520 void brelse(struct buffer_head * buf)
521 {

```

```

522     if (!buf)
523         return;
    ! 如果缓冲为空，则返回！
524     wait_on_buffer(buf);
    ! 等待该缓冲解锁
525     if (buf->b_count) {
526         if (--buf->b_count)
527             return;
528         wake_up(&buffer_wait);
529     }
    ! 如果其引用计数不为 0，则自减一后返回！否则等待该缓冲解锁后
    ! 返回！
530 }
531     printk("VFS: brelse: Trying to free free buffer\n");
    ! 否则打印提示消息。
532 }

534 /*
535 * bread() reads a specified block and returns the buffer that contains
536 * it. It returns NULL if the block was unreadable.
537 */
    ! bread()从设备上读取指定的数据块并返回含有数据的缓冲区，如果指定的块
    ! 不存在则返回NULL
538 struct buffer_head * bread(dev_t dev, int block, int size)
539 {
540     struct buffer_head * bh;
541
542     if (!(bh = getblk(dev, block, size))) {
543         printk("VFS: bread: READ error on device %d/%d\n",
544                 MAJOR(dev), MINOR(dev));
    ! 在高速缓冲中申请一块高速缓冲，定义于本文件
545         return NULL;
546     }
547     if (bh->b_uptodate)
548         return bh;
    ! 如果该缓冲区已经被更新则直接返回
549     ll_rw_block(READ, 1, &bh);
    ! 读设备，定义于Drivers/block/ll_rw_blk.c
550     wait_on_buffer(bh);
    ! 等待缓冲解锁
551     if (bh->b_uptodate)
552         return bh;
    ! 如果该缓冲区已经被更新则直接返回
553     brelse(bh);

```

```

554     return NULL;
      ! 执行到这里表示由错误发生，于是返回NULL
555 }
560 /*
561 * Try to increase the number of buffers available: the size argument
562 * is used to determine what kind of buffers we want.
563 */
564 static int grow_buffers(int pri, int size)
565 {
566     unsigned long page;
567     struct buffer_head *bh, *tmp;
568
569     if ((size & 511) || (size > PAGE_SIZE)) {
570         printk("VFS: grow_buffers: size = %d\n", size);
571         return 0;
572     }
573     if(!(page = get_free_page(pri)))
574         return 0;
575     bh = create_buffers(page, size);
576     if (!bh) {
577         free_page(page);
578         return 0;
579     }
580     tmp = bh;
581     while (1) {
582         if (free_list) {
583             tmp->b_next_free = free_list;
584             tmp->b_prev_free = free_list->b_prev_free;
585             free_list->b_prev_free->b_next_free = tmp;
586             free_list->b_prev_free = tmp;
587         } else {
588             tmp->b_prev_free = tmp;
589             tmp->b_next_free = tmp;
590         }
591         free_list = tmp;
592         ++nr_buffers;
593         if (tmp->b_this_page)
594             tmp = tmp->b_this_page;
595         else
596             break;
597     }
598     tmp->b_this_page = bh;
599     buffermem += PAGE_SIZE;
600     return 1;

```

[901 }](#)

```

1003 /*
1004 * This initializes the initial buffer free list. nr_buffers is set
1005 * to one less than the actual number of buffers, as a sop to backwards
1006 * compatibility --- the old code did this (I think unintentionally,
1007 * but I'm not sure), and programs in the ps package expect it.
1008 * - TYT 8/30/92
1009 */
    ! 该初始化程序初始化空闲缓冲列表，nr_buffers 中保存真正的缓冲数量，并且向后
    ! 兼容—这是老的代码做的（我想它是不能被替换的,但是我不能确认）并且这个程序
    ! ps程序包需要它-TYT 8/30/92
1010 void buffer_init(void)
1011 {
1012     int i;
1013
1014     if (high_memory >= 4*1024*1024)
        ! 如果high_memory是 4M以上
1015         min_free_pages = 200;
        ! 则设置min_free_pags=200
1016     else
1017         min_free_pages = 20;
        ! 否则设置为 20
1018     for (i = 0 ; i < NR_HASH ; i++)
1019         hash_table[i] = NULL;
        ! 清零hash表
1020     free_list = 0;
1021     grow_buffers(GFP_KERNEL, BLOCK_SIZE);
        ! 定义于本文件中
1022     if (!free_list)
        ! 如果设置空闲缓冲失败
1023         panic("VFS: Unable to initialize buffer free list!");
        ! 死机
1024     return;
1025 }

```

## Fs/super.c

[1 /\\*](#)

```

2  * linux/fs/super.c
3  *
4  * Copyright (C) 1991, 1992 Linus Torvalds
5  */
6
7 /*
8  * super.c contains code to handle the super-block tables.
9  */
10 #include <linux/config.h>
11 #include <linux/sched.h>
12 #include <linux/kernel.h>
13 #include <linux/major.h>
14 #include <linux/stat.h>
15 #include <linux/errno.h>
16 #include <linux/string.h>
17 #include <linux/locks.h>
18
19 #include <asm/system.h>
20 #include <asm/segment.h>
21
22
23 /*
24  * The definition of file_systems that used to be here is now in
25  * filesystems.c. Now super.c contains no fs specific code. --jrs
26  */
27
28 extern struct file_system_type file_systems[];
29 extern struct file_operations * get_blkfops(unsigned int);
30 extern struct file_operations * get_chrfops(unsigned int);
31
32 extern void wait_for_keypress(void);
33 extern void fcntl_init_locks(void);
34
35 extern int root_mountflags;
36
37 struct super_block super_blocks[NR_SUPER];
38
39 static int do_remount_sb(struct super_block *sb, int flags, char * data);
40
41 /* this is initialized in init/main.c */
42 dev_t ROOT_DEV = 0;
43

```

! 取得对应于文件系统类型名的文件系统类型

× name=文件系统类型名

```

44 struct file_system_type *get_fs_type(char *name)
45 {
46     int a;
47
48     if (!name)
49         return &file_systems[0];
50     for(a = 0 ; file_systems[a].read_super ; a++)
51         if (!strcmp(name,file_systems[a].name))
52             return(&file_systems[a]);
! 查找 file_systems 数组，找到则返回之
53     return NULL;
! 否则，返回 NULL
54 }
55
56 void __wait_on_super(struct super_block * sb)
57 {
58     struct wait_queue wait = { current, NULL };
59
60     add_wait_queue(&sb->s_wait, &wait);
61 repeat:
62     current->state = TASK_UNINTERRUPTIBLE;
63     if (sb->s_lock) {
64         schedule();
65         goto repeat;
66     }
67     remove_wait_queue(&sb->s_wait, &wait);
68     current->state = TASK_RUNNING;
69 }
70
71 void sync_supers(dev_t dev)
72 {
73     struct super_block * sb;
74
75     for (sb = super_blocks + 0 ; sb < super_blocks + NR_SUPER ; sb++) {
76         if (!sb->s_dev)
77             continue;
78         if (dev && sb->s_dev != dev)
79             continue;
! 如果超级块没有被占用，或者被占用但是并不是指定设备的的超级块，则跳过
! 下面代码
80         __wait_on_super(sb);
! 等待超级块解锁
81         if (!sb->s_dev || !sb->s_dirt)
82             continue;

```

```

! 如果超级块没有被占用或者没有脏数据（即被安装了块设备）,
! 则跳过下面代码
83         if (dev && (dev != sb->s_dev))
84             continue;
! 如果超级块没有被占用, 或者被占用但是并不是指定设备的的超级块, 则跳过
! 下面代码
85         if (sb->s_op && sb->s_op->write_super)
86             sb->s_op->write_super(sb);
87     }
88 }
89
! 取指定设备的超级块
90 static struct super_block * get_super(dev_t dev)
91 {
92     struct super_block * s;
93
94     if (!dev)
95         return NULL;
! 设备为 0, 则返回 NULL
96     s = 0+super_blocks;
97     while (s < NR_SUPER+super_blocks)
! 搜索系统的所有超级块
98     if (s->s_dev == dev) {
99         wait_on_super(s);
! 如果搜索到的超级块是对应的设备, 则等待其解锁
100        if (s->s_dev == dev)
101            return s;
! 返回超级块
102        s = 0+super_blocks;
! 否则, 从起始位置重新搜索
103    } else
104        s++;
! 不等, 则搜索下一项
105    return NULL;
106 }
107
108 void put_super(dev_t dev)
109 {
110     struct super_block * sb;
111
112     if (dev == ROOT_DEV) {
113         printk("VFS: Root device %d/%d: prepare for armageddon\n",
114                                         MAJOR(dev),

```

```

MINOR(dev));
115         return;
116     }
117     if (!(sb = get_super(dev)))
118         return;
119     if (sb->s_covered) {
120         printk("VFS: Mounted device %d/%d - tssk, tssk\n",
121                         MAJOR(dev), MINOR(dev));
122         return;
123     }
124     if (sb->s_op && sb->s_op->put_super)
125         sb->s_op->put_super(sb);
126 }
127

! 读超级块
× dev=根设备
× name=文件系统类型名
! (1.0 支持文件系统有"minix", "ext", "ext2", "xafs", "msdos",
"proc".....等等)
× flags=文件系统读写方式 (MS_RDONLY 等等)

128 static struct super_block * read_super(dev_t dev, char *name, int flags,
129                                         void *data, int silent)
130 {
131     struct super_block * s;
132     struct file_system_type *type;
133
134     if (!dev)
135         return NULL;
! 如果根设备为 0, 则返回 NULL
136     check_disk_change(dev);
! 检查对应的 dev 设备是否已经更换, 如果已经更换就使对应的高速
! 缓冲无效, 该函数不注释, 定义于 Fs/buffer.c
137     s = get_super(dev);
! 定义于 Fs/super.c
138     if (s)
139         return s;
! 取对应设备的超级块, 取到则直接返回该超级块。
140     if (!(type = get_fs_type(name))) {
141         printk("VFS: on device %d/%d: get_fs_type(%s) failed\n",
142                         MAJOR(dev), MINOR(dev),
name);
143         return NULL;
! 如果不能取得文件系统类型指针, 则打印错误提示后, 返回 NULL
! get_fs_type 定义于本文件

```

```

! printk 定义于 Kernel/printk.c
144      }
145      for (s = 0+super_blocks ;; s++) {
146          if (s >= NR_SUPER+super_blocks)
147              return NULL;
148          if (!s->s_dev)
149              break;
! 如果超级块大于系统所拥有的超级块数，则返回 NULL
! 或者对应的块设备为空，则跳出循环
150      }
151      s->s_dev = dev;
152      s->s_flags = flags;
153      if (!type->read_super(s,data, silent)) {
! 对应 Minix 是 minix_read_super， 定义在 Fs/minix/inode.c
154          s->s_dev = 0;
155          return NULL;
156      }
157      s->s_dev = dev;
158      s->s_covered = NULL;
159      s->s_rd_only = 0;
160      s->s_dirt = 0;
! 设置超级块中的数据
161      return s;
! 返回超级块指针
162 }
163
164 /*
165 * Unnamed block devices are dummy devices used by virtual
166 * filesystems which don't use real block-devices. --jrs
167 */
168
169 static char unnamed_dev_in_use[256];
170
171 static dev_t get_unnamed_dev(void)
172 {
173     static int first_use = 0;
174     int i;
175
176     if (first_use == 0) {
177         first_use = 1;
178         memset(unnamed_dev_in_use, 0, sizeof(unnamed_dev_in_use));
179         unnamed_dev_in_use[0] = 1; /* minor 0 (nodev) is special */
180     }
181     for (i = 0; i < sizeof(unnamed_dev_in_use)/sizeof(unnamed_dev_in_use[0]; i++) {

```

```

182         if (!unnamed_dev_in_use[i]) {
183             unnamed_dev_in_use[i] = 1;
184             return (UNNAMED_MAJOR << 8) | i;
185         }
186     }
187     return 0;
188 }
189
190 static void put_unnamed_dev(dev_t dev)
191 {
192     if (!dev)
193         return;
194     if (!unnamed_dev_in_use[dev]) {
195         printk("VFS: put_unnamed_dev: freeing unused device %d/%d\n",
196                 MAJOR(dev),
197                 MINOR(dev));
198     }
199     unnamed_dev_in_use[dev] = 0;
200 }
201
202 static int do_umount(dev_t dev)
203 {
204     struct super_block * sb;
205     int retval;
206
207     if (dev==ROOT_DEV) {
208         /* Special case for "unmounting" root. We just try to remount
209          it readonly, and sync() the device. */
210         if (!(sb=get_super(dev)))
211             return -ENOENT;
212         if (!(sb->s_flags & MS_RDONLY)) {
213             fsync_dev(dev);
214             retval = do_remount_sb(sb, MS_RDONLY, 0);
215             if (retval)
216                 return retval;
217         }
218         return 0;
219     }
220     if (!(sb=get_super(dev)) || !(sb->s_covered))
221         return -ENOENT;
222     if (!sb->s_covered->i_mount)
223         printk("VFS: umount(%d/%d): mounted inode has i_mount=NULL\n",
224                 MAJOR(dev),

```

```

MINOR(dev));
225      if (!fs_may_umount(dev, sb->s_mounted))
226          return -EBUSY;
227      sb->s_covered->i_mount = NULL;
228      iput(sb->s_covered);
229      sb->s_covered = NULL;
230      iput(sb->s_mounted);
231      sb->s_mounted = NULL;
232      if (sb->s_op && sb->s_op->write_super && sb->s_dirt)
233          sb->s_op->write_super(sb);
234      put_super(dev);
235      return 0;
236 }
237 /*
238 */
239 * Now umount can handle mount points as well as block devices.
240 * This is important for filesystems which use unnamed block devices.
241 *
242 * There is a little kludge here with the dummy_inode. The current
243 * vfs release functions only use the r_dev field in the inode so
244 * we give them the info they need without using a real inode.
245 * If any other fields are ever needed by any block device release
246 * functions, they should be faked here. -- jrs
247 */
248
249 asmlinkage int sys_umount(char * name)
250 {
251     struct inode * inode;
252     dev_t dev;
253     int retval;
254     struct inode dummy_inode;
255     struct file_operations * fops;
256
257     if (!suser())
258         return -EPERM;
259     retval = namei(name,&inode);
260     if (retval) {
261         retval = lnamei(name,&inode);
262         if (retval)
263             return retval;
264     }
265     if (S_ISBLK(inode->i_mode)) {
266         dev = inode->i_rdev;
267         if (IS_NODEV(inode)) {

```

```

268             iput(inode);
269             return -EACCES;
270         }
271     } else {
272         if (!inode || !inode->i_sb || inode != inode->i_sb->s_mounted) {
273             iput(inode);
274             return -EINVAL;
275         }
276         dev = inode->i_sb->s_dev;
277         iput(inode);
278         memset(&dummy_inode, 0, sizeof(dummy_inode));
279         dummy_inode.i_rdev = dev;
280         inode = &dummy_inode;
281     }
282     if (MAJOR(dev) >= MAX_BLKDEV) {
283         iput(inode);
284         return -ENXIO;
285     }
286     if (!(retval = do_umount(dev)) && dev != ROOT_DEV) {
287         fops = get_blkfops(MAJOR(dev));
288         if (fops && fops->release)
289             fops->release(inode, NULL);
290         if (MAJOR(dev) == UNNAMED_MAJOR)
291             put_unnamed_dev(dev);
292     }
293     if (inode != &dummy_inode)
294         iput(inode);
295     if (retval)
296         return retval;
297     fsync_dev(dev);
298     return 0;
299 }
300
301 /*
302  * do_mount() does the actual mounting after sys_mount has done the ugly
303  * parameter parsing. When enough time has gone by, and everything uses the
304  * new mount() parameters, sys_mount() can then be cleaned up.
305  *
306  * We cannot mount a filesystem if it has active, used, or dirty inodes.
307  * We also have to flush all inode-data for this device, as the new mount
308  * might need new info.
309 */
310 static int do_mount(dev_t dev, const char * dir, char * type, int flags, void * data)
311 {

```

```

312     struct inode * dir_i;
313     struct super\_block * sb;
314     int error;
315
316     error = namei(dir,&dir_i);
317     if (error)
318         return error;
319     if (dir_i->i_count != 1 || dir_i->i_mount) {
320         iput(dir_i);
321         return -EBUSY;
322     }
323     if (!S\_ISDIR(dir_i->i_mode)) {
324         iput(dir_i);
325         return -EPERM;
326     }
327     if (!fs\_may\_mount(dev)) {
328         iput(dir_i);
329         return -EBUSY;
330     }
331     sb = read\_super(dev,type,flags,data,0);
332     if (!sb || sb->s_covered) {
333         iput(dir_i);
334         return -EBUSY;
335     }
336     sb->s_covered = dir_i;
337     dir_i->i_mount = sb->s_mounted;
338     return 0;           /* we don't iput(dir_i) - see umount */
339 }
340
341
342 /*
343  * Alters the mount flags of a mounted file system. Only the mount point
344  * is used as a reference - file system type and the device are ignored.
345  * FS-specific mount options can't be altered by remounting.
346  */
347
348 static int do\_remount\_sb(struct super\_block *sb, int flags, char *data)
349 {
350     int retval;
351
352     /* If we are remounting RONLY, make sure there are no rw files open */
353     if ((flags & MS\_RDONLY) && !(sb->s_flags & MS\_RDONLY))
354         if (!fs\_may\_remount\_ro(sb->s_dev))
355             return -EBUSY;

```

```

356         if (sb->s_op && sb->s_op->remount_fs) {
357             retval = sb->s_op->remount_fs(sb, &flags, data);
358             if (retval)
359                 return retval;
360         }
361         sb->s_flags = (sb->s_flags & ~MS_RMT_MASK) |
362             (flags & MS_RMT_MASK);
363         return 0;
364     }
365
366 static int do_remount(const char *dir,int flags,char *data)
367 {
368     struct inode *dir_i;
369     int retval;
370
371     retval = namei(dir,&dir_i);
372     if (retval)
373         return retval;
374     if (dir_i != dir_i->i_sb->s_mounted) {
375         iput(dir_i);
376         return -EINVAL;
377     }
378     retval = do_remount_sb(dir_i->i_sb, flags, data);
379     iput(dir_i);
380     return retval;
381 }
382
383 static int copy_mount_options (const void * data, unsigned long *where)
384 {
385     int i;
386     unsigned long page;
387     struct vm_area_struct * vma;
388
389     *where = 0;
390     if (!data)
391         return 0;
392
393     for (vma = current->mmap ; ; ) {
394         if (!vma ||
395             (unsigned long) data < vma->vm_start) {
396             return -EFAULT;
397         }
398         if ((unsigned long) data < vma->vm_end)
399             break;

```

```

400             vma = vma->vm_next;
401         }
402         i = vma->vm_end - (unsigned long) data;
403         if (PAGE_SIZE <= (unsigned long) i)
404             i = PAGE_SIZE-1;
405         if (!(page = __get_free_page(GFP_KERNEL))) {
406             return -ENOMEM;
407         }
408         memcpy_fromfs((void *) page,data,i);
409         *where = page;
410         return 0;
411     }
412
413 /*
414  * Flags is a 16-bit value that allows up to 16 non-fs dependent flags to
415  * be given to the mount() call (ie: read-only, no-dev, no-suid etc).
416  *
417  * data is a (void *) that can point to any structure up to
418  * PAGE_SIZE-1 bytes, which can contain arbitrary fs-dependent
419  * information (or be NULL).
420  *
421  * NOTE! As old versions of mount() didn't use this setup, the flags
422  * has to have a special 16-bit magic number in the hight word:
423  * 0xC0ED. If this magic word isn't present, the flags and data info
424  * isn't used, as the syscall assumes we are talking to an older
425  * version that didn't understand them.
426 */
427 asmlinkage int sys_mount(char * dev_name, char * dir_name, char * type,
428                           unsigned long new_flags, void * data)
429 {
430     struct file_system_type * fstype;
431     struct inode * inode;
432     struct file_operations * fops;
433     dev_t dev;
434     int retval;
435     char * t;
436     unsigned long flags = 0;
437     unsigned long page = 0;
438
439     if (!suser())
440         return -EPERM;
441     if ((new_flags &
442          (MS_MGC_MSK | MS_REMOUNT)) == (MS_MGC_VAL |
443 MS_REMOUNT)) {

```

```

443         retval = copy_mount_options(data, &page);
444         if (retval < 0)
445             return retval;
446         retval = do_remount(dir_name,
447                               new_flags & ~MS_MGC_MSK &
448                               ~MS_REMOUNT,
449                               (char *) page);
450         free_page(page);
451         return retval;
452     }
453     retval = copy_mount_options(type, &page);
454     if (retval < 0)
455         return retval;
456     fstype = get_fs_type((char *) page);
457     free_page(page);
458     if (!fstype)
459         return -ENODEV;
460     t = fstype->name;
461     if (fstype->requires_dev) {
462         retval = namei(dev_name,&inode);
463         if (retval)
464             return retval;
465         if (!S_ISBLK(inode->i_mode)) {
466             iput(inode);
467             return -ENOTBLK;
468         }
469         if (IS_NODEV(inode)) {
470             iput(inode);
471             return -EACCES;
472         }
473         dev = inode->i_rdev;
474         if (MAJOR(dev) >= MAX_BLKDEV) {
475             iput(inode);
476             return -ENXIO;
477         }
478     } else {
479         if (!(dev = get_unnamed_dev()))
480             return -EMFILE;
481         inode = NULL;
482     }
483     fops = get_blkfops(MAJOR(dev));
484     if (fops && fops->open) {
485         retval = fops->open(inode,NULL);
486         if (retval) {

```

```

486             iput(inode);
487             return retval;
488         }
489     }
490     page = 0;
491     if ((new_flags & MS_MGC_MSK) == MS_MGC_VAL) {
492         flags = new_flags & ~MS_MGC_MSK;
493         retval = copy_mount_options(data, &page);
494         if (retval < 0) {
495             iput(inode);
496             return retval;
497         }
498     }
499     retval = do_mount(dev, dir_name, t, flags, (void *) page);
500     free_page(page);
501     if (retval && fops && fops->release)
502         fops->release(inode, NULL);
503     iput(inode);
504     return retval;
505 }
506
! 安装根文件系统
507 void mount_root(void)
508 {
509     struct file_system_type * fs_type;
510     struct super_block * sb;
511     struct inode * inode;
512
513     memset(super_blocks, 0, sizeof(super_blocks));
! 清零超级块，该函数不做注释，定义在 Include/linux/string.h
! 本书没有列出
514     fcntl_init_locks();
! 初始化文件锁定表，定义于 Fs/locks.c
515     if (MAJOR(ROOT_DEV) == FLOPPY_MAJOR) {
516         printk(KERN_NOTICE "VFS: Insert root floppy and press ENTER\n");
517         wait_for_keypress();
! 如果根文件系统设备是软盘，则打印提示信息，并等待用户按下键盘回车键
! wait_for_keypress () 定义于 Drivers/char/tty_io.c，这里不做注释。
518     }
519     for (fs_type = file_systems; fs_type->read_super; fs_type++) {
520         if (!fs_type->requires_dev)
521             continue;
522         sb = read_super(ROOT_DEV, fs_type->name, root_mountflags, NULL, 1);
! 读根设备上超级块，定义于 Fs/super.c

```

```

    ! 请注意此函数是通用的，会在其里面调用对应文件系统类型的超级块读写函数
523         if (sb) {
    ! 超级块被读出
524             inode = sb->s_mounted;
    ! 取得被安装的根文件系统I节点
525             inode->i_count += 3 ;
                /* NOTE! it is logically used 4
times, not 1 */
    ! 注意：从逻辑上讲，它被引用
了 4 次，而
    ! 不是一次
    !
526             sb->s_covered = inode;
527             sb->s_flags = root_mountflags;
    ! 置根文件系统的被安装标志
528             current->pwd = inode;
529             current->root = inode;
    ! 置当前进程的当前工作目录及根目录的 I 节点
530             printk ("VFS: Mounted root (%s filesystem)%s.\n",
531                     fs_type->name,
532                     (sb->s_flags & MS_RDONLY) ? " readonly" : "");
533             return;
    ! 打印提示消息后返回
534         }
535     }
536     panic("VFS: Unable to mount root");
    ! 否则，打印提示消息后死机
537 }
538

```

## Fs/file\_table.c (部分代码)

```

    ! 文件描述符表数组的初始化
62 unsigned long file_table_init(unsigned long start, unsigned long end)
63 {
64     first_file = NULL;
    ! 让first_file指向空
65     return start;
    ! 返回可用内存的开始值
66 }

```

## Fs/inode.c (部分代码)

```

! I 节点 hash 表的初始化
108 unsigned long inode_init(unsigned long start, unsigned long end)
109 {
110     memset(hash_table, 0, sizeof(hash_table));
    ! 清零用于查找I节点的hash表
111     first_inode = NULL;
    ! first_inode指向空
112     return start;
    ! 返回可用内存的开始值
113 }

! 等待 I 节点解锁
× inode=要解锁的节点
117 static inline void wait_on_inode(struct inode * inode)
118 {
119     if (inode->i_lock)
120         __wait_on_inode(inode);
    ! 如果，该I节点被锁的话，则调用__wait_on_inode释放之
    ! 定义于本文件
121 }

! 解锁 I 节点
× inode=要解锁的节点
129 static inline void unlock_inode(struct inode * inode)
130 {
131     inode->i_lock = 0;
132     wake_up(&inode->i_wait);
    ! 唤醒等待队列
133 }

211
    ! I节点写入块设备
    X inode=要写入设备的I节点
212 static void write_inode(struct inode * inode)
213 {
214     if (!inode->i_dirt)
215         return;

```

```

    ! 如果I节点没有被占用，则返回
216     wait_on_inode(inode);
    ! 等待I节点解锁
217     if (!inode->i_dirt)
218         return;
    ! 如果I节点没有被占用，则返回
219     if (!inode->i_sb || !inode->i_sb->s_op || !inode->i_sb->s_op->write_inode) {
220         inode->i_dirt = 0;
221         return;
222     }
223     inode->i_lock = 1;
224     inode->i_sb->s_op->write_inode(inode);
    ! 写I节点到块设备
225     unlock_inode(inode);
    ! 解锁块I节点
226 }

```

```

    ! 同步 I 节点
288 void sync_inodes(dev_t dev)
289 {
290     int i;
291     struct inode * inode;
292
293     inode = first_inode;
    ! inode指向首节点
294     for(i = 0; i < nr_inodes*2; i++, inode = inode->i_next) {
295         if (dev && inode->i_dev != dev)
296             continue;
    ! 如果I节点的设备并不是传入的设备，则跳过下面代码
297         wait_on_inode(inode);
    ! 否则等待该节点解锁
298         if (inode->i_dirt)
299             write_inode(inode);
    ! 如果该I节点被修改（有脏数据），则把该节点写入块设备，定义于本文件
300     }
301 }
302
    ! 释放 I 节点
    × inode=要释放的节点
303 void put(struct inode * inode)
304 {
305     if (!inode)
306         return;
307     wait_on_inode(inode);

```

```

    ! 如果, I节点上锁的话, 则解锁, 定义于本文件
308     if (!inode->i_count) {
        ! 如果I节点被使用次数为 0 (即该I节点空闲), 打印提示消息后退出
309         printk("VFS: iput: trying to free free inode\n");
310         printk("VFS: device %d/%d, inode %lu, mode=0%07o\n",
311             MAJOR(inode->i_rdev), MINOR(inode->i_rdev),
312             inode->i_ino, inode->i_mode);
313         return;
314     }
315     if (inode->i_pipe)
316         wake_up_interruptible(&PIPE_WAIT(*inode));
        ! 如果当前的I节点是管道型的, 调用wake_up_interruptible, 定义于Kernel/sched.c
317 repeat:
318     if (inode->i_count>1) {
319         inode->i_count--;
320         return;
        ! 如果I节点被使用次数大于 1, 则主动减一后, 返回!
321     }
322     wake_up(&inode_wait);
        ! 唤醒等待队列, 定义于Kernel/sched.c
323     if (inode->i_pipe) {
        ! 如果当前的I节点是管道型的
324         unsigned long page = (unsigned long) PIPE_BASE(*inode);
        ! 获取该pipe类型I节点的基址
325         PIPE_BASE(*inode) = NULL;
        ! 置空基地址
326         free_page(page);
        ! 释放该页, 定义于Mm/swap.c
327     }
328     if (inode->i_sb && inode->i_sb->s_op && inode->i_sb->s_op->put_inode) {
329         inode->i_sb->s_op->put_inode(inode);
330         if (!inode->i_nlink)
331             return;
332     }
333     if (inode->i_dirt) {
334         write_inode(inode); /* we can sleep - so do again */
335         wait_on_inode(inode);
336         goto repeat;
        ! 如果该 I 节点已经被修改, 首先将 I 节点信息写入设备, 调整到 repeat 处
        ! 定义于本文件,
337     }
338     inode->i_count--;
        ! I节点计数自减一
339     nr_free_inodes++;

```

```

    ! 可使用I节点数自增一
420     return;
    ! 返回
421 }

    ! 获取 I 节点
    ✗ sb=超级块指针
    ✗ nr=I 节点号
422 struct inode * iget(struct super_block * sb,int nr)
423 {
424     return _iget(sb,nr,1);
    ! 真正所调用的函数, 定义于本文件
425 }
426
427 struct inode * _iget(struct super_block * sb, int nr, int crossmntp)
428 {
429     static struct wait_queue * update_wait = NULL;
430     struct inode_hash_entry * h;
431     struct inode * inode;
432     struct inode * empty = NULL;
433
434     if (!sb)
435         panic("VFS: iget with sb==NULL");
    ! 如果超级块为空, 打印提示消息。
436     h = hash(sb->s_dev, nr);
    ! 根据超级块中设备号, 从全局的hash_table表中取的I节点在hash表中的指针
437 repeat:
438     for (inode = h->inode; inode ; inode = inode->i_hash_next)
439         if (inode->i_dev == sb->s_dev && inode->i_ino == nr)
440             goto found_it;
    ! 根据 I 节点及超级块中设备号及 I 节点号, 检查是否已经找到了, 如果找到了
    ! 调转到found_it
441     if (!empty) {
442         h->updating++;
443         empty = get_empty_inode();
444         if (!--h->updating)
445             wake_up(&update_wait);
446         if (empty)
447             goto repeat;
448         return (NULL);
449     }
    ! 否则, 获取一个空闲的 I 节点后, 重新跳到 repeat, 再来一遍
    ! get_empty_inode这里不做注释
450     inode = empty;

```

```

451     inode->i_sb = sb;
452     inode->i_dev = sb->s_dev;
453     inode->i_ino = nr;
454     inode->i_flags = sb->s_flags;
    ! 设置取的的I节点中的内容
455     put_last_free(inode);
456     insert_inode_hash(inode);
457     read_inode(inode);
    ! 首先把该 I 节点加入 I 节点的链表中，并插入 I 节点的 hash 表中，并调用
    ! 对应文件系统的真实的读 I 节点函数
    ! 以上 3 个函数未作注释
458     goto return_it;
459
460 found_it:
461     if (!inode->i_count)
462         nr_free_inodes--;
463     inode->i_count++;
464     wait_on_inode(inode);
465     if (inode->i_dev != sb->s_dev || inode->i_ino != nr) {
466         printk("Whee.. inode changed from under us. Tell Linus\n");
467         iput(inode);
468         goto repeat;
469     }
    ! 检查 I 节点是否与超级块中定义的相同，不同则打印提示消息后，跳转到
    ! repeat处执行
470     if (crossmnt && inode->i_mount) {
471         struct inode * tmp = inode->i_mount;
472         tmp->i_count++;
473         iput(inode);
474         inode = tmp;
475         wait_on_inode(inode);
476     }
477     if (empty)
478         iput(empty);
    ! 释放临时的I节点，定义于本文件
479
480 return_it:
481     while (h->updating)
482         sleep_on(&update_wait);
    ! 如果，I节点被更新了，则循环等待其解锁
483     return inode;
    ! 返回找到的I节点
484 }
485

```

```

486 /*
487 * The "new" scheduling primitives (new as of 0.97 or so) allow this to
488 * be done without disabling interrupts (other than in the actual queue
489 * updating things: only a couple of 386 instructions). This should be
490 * much better for interrupt latency.
491 */
    ! 这个新的调度程序（为 0.97 或更新）允许不用关中断调用它（另外在
    ! 实际的队列中更新这些事情：仅仅只有几个 386 指令），它在中断期间
    ! 会做的更好
492 static void __wait_on_inode(struct inode *inode)
493 {
494     struct wait_queue wait = { current, NULL };
495
496     add_wait_queue(&inode->i_wait, &wait);
    ! 定义于Include/linux/sched.h
497 repeat:
498     current->state = TASK_UNINTERRUPTIBLE;
    ! 置当前进程的状态为不可中断
499     if (inode->i_lock) {
        ! 如果该I节点仍然被锁，则重新调度，直到解锁
500         schedule();
501         goto repeat;
502     }
503     remove_wait_queue(&inode->i_wait, &wait);
    ! 从等待队列中删除I节点等待队列
504     current->state = TASK_RUNNING;
    ! 置当前进程状态为可调度状态
505 }

```

## Fs/locks.c (部分代码)

```

37 /*
38 * Called at boot time to initialize the lock table ...
39 */
    ! 在系统启动时调用用于初始化锁定表
40
    ! 初始化文件锁定表
41 void fcntl_init_locks(void)
42 {
43     struct file_lock *fl;

```

```

44
45     for (fl = &file_lock_table[0]; fl < file_lock_table + NR_FILE_LOCKS - 1; fl++) {
46         fl->fl_next = fl + 1;
47         fl->fl_owner = NULL;
48     }
49     file_lock_table[NR_FILE_LOCKS - 1].fl_next = NULL;
50     file_lock_table[NR_FILE_LOCKS - 1].fl_owner = NULL;
51     file_lock_free_list = &file_lock_table[0];
! 初始化 NR_FILE_LOCKS 个文件锁定表，并且让 file_lock_free_list
! 指向文件锁定表的第 0 项（实际上就是第一个）
52 }
169 /*
170 * This function is called when the file is closed.
171 */
172
! 该函数在文件关闭时调用
173 void fcntl_remove_locks(struct task_struct *task, struct file *filp,
174                         unsigned int fd)
175 {
176     struct file_lock *fl;
177     struct file_lock **before;
178
179     /* Find first lock owned by caller ... */
! 通过调用者查找自己的第一个锁
180
181     before = &filp->f_inode->i_flock;
182     while ((fl = *before) && (task != fl->fl_owner || fd != fl->fl_fd))
183         before = &fl->fl_next;
184
185     /* The list is sorted by owner and fd ... */
186
! 这个通过自己和文件描述符存储列表
187     while ((fl = *before) && task == fl->fl_owner && fd == fl->fl_fd)
188         free_lock(before);
! 定义于本文件
189 }

435 /*
436 * Add a lock to the free list ...
437 */
! 把锁加入释放列表
438
439 static void free_lock(struct file_lock **fl_p)
440 {

```

```

441     struct file_lock *fl;
442
443     fl = *fl_p;
444     if (fl->fl_owner == NULL)      /* sanity check */
445         panic("free_lock: broken lock list\n");
    ! 如果对应的文件锁没有和任务绑定，则死机
446
447     *fl_p = (*fl_p)->fl_next;
448
449     fl->fl_next = file_lock_free_list;    /* add to free list */
450     file_lock_free_list = fl;
451     fl->fl_owner = NULL;                /* for sanity checks */
452
453     wake_up(&fl->fl_wait);
    ! 唤醒等待队列
454 }

```

## Fs/open.c (部分代码)

```

361
362 /*
363 * Note that while the flag value (low two bits) for sys_open means:
364 * 00 - read-only
365 * 01 - write-only
366 * 10 - read-write
367 * 11 - special
368 * it is changed into
369 * 00 - no permissions needed
370 * 01 - read-permission
371 * 10 - write-permission
372 * 11 - read-write
373 * for the internal routines (ie open_namei()/follow_link() etc). 00 is
374 * used by symlinks.
375 */
    ! 注意标志（低两位）有不同的意义
    ! 00—只读
    ! 01—只写
    ! 10—读写
    ! 11—专用的

```

```

! .....
! filename=文件名 (注意此时已经在核心空间了)
! flags=打开标志
! mode=许可属性

376 int do_open(const char * filename,int flags,int mode)
377 {
378     struct inode * inode;
379     struct file * f;
380     int flag,error,fd;
381
382     for(fd=0 ; fd<NR_OPEN ; fd++)
383         if (!current->filp[fd])
384             break;
    ! 搜索空闲项
385     if (fd>=NR_OPEN)
386         return -EMFILE;
    ! 如果找到的fd大于NR_OPEN，则返回-EMFILE
387     FD_CLR(fd,&current->close_on_exec);
    ! 设置执行时关闭文件句柄位图，复位对应比特位
388     f = get_empty_filp();
    ! 取得空闲的文件结构项，定义于Fs/file_table.c
389     if (!f)
390         return -ENFILE;
    ! 如果找不到，则返回-ENFILE
391     current->filp[fd] = f;
    ! 当前进程文件句柄的文件结构指针指向搜索到文件结构
392     f->f_flags = flag = flags;
393     f->f_mode = (flag+1) & O_ACCMODE;
    ! 设置对应的标志及许可模式
394     if (f->f_mode)
395         flag++;
396     if (flag & (O_TRUNC | O_CREAT))
397         flag |= 2;
398     error = open_namei(filename,flag,mode,&inode,NULL);
    ! 打开文件，定义于Fs/namei.c
399     if (error) {
400         current->filp[fd]=NULL;
401         f->f_count--;
402         return error;
403     }
    ! 返回值为非 0，则置空对应的文件描述符指针
404
405     f->f_inode = inode;
406     f->f_pos = 0;

```

```

407     f->f_reada = 0;
408     f->f_op = NULL;
    ! 设置文件描述符中的内容
409     if (inode->i_op)
410         f->f_op = inode->i_op->default_file_ops;
411     if (f->f_op && f->f_op->open) {
412         error = f->f_op->open(inode,f);
413         if (error) {
414             iput(inode);
415             f->f_count--;
416             current->filp[fd]=NULL;
417             return error;
418         }
419     }
    ! 如果对应的文件描述符指针中文件操作指针不为空，并且其 open 指针不是空
    ! 则调用之
420     f->f_flags &= ~(O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);
    ! 标志位置位
421     return (fd);
    ! 返回文件文件描述符的编号（即文件句柄）
422 }

```

! 系统调用，打开、创建文件  
 ✗ filename=文件名  
 ✗ flags=打开文件标志  
 ✗ mode=文件的许可属性

```

424 asmlinkage int sys_open(const char * filename,int flags,int mode)
425 {
426     char * tmp;
427     int error;
428
429     error = getname(filename, &tmp);
    ! 拷贝用户名到核心空间，定义于Fs/namei.c
430     if (error)
431         return error;
    ! 如果error不为 0，则直接返回错误值。
432     error = do_open(tmp,flags,mode);
    ! 否则打开或创建对应的文件，定义于本文件
433     putname(tmp);
    ! 释放为存放文件名分配的内存，定义于Fs/namei.c
434     return error;
    ! 相同调用的返回值

```

435 }436

! 关闭 filp

× filp=文件指针

× fd=文件描述符

442 int close\_fp(struct file \*filp, unsigned int fd)443 {    struct inode \*inode;445

if (filp-&gt;f\_count == 0) {

! 如果对应的文件描述符数为 0，则提示后退出

printk("VFS: Close: file count is 0\n");

return 0;

}

inode = filp->f\_inode;

! 取得对应的i节点

        if (inode && S\_ISREG(inode->i\_mode))            fctl\_remove\_locks(current, filp, fd);

! 如果取得的文件是正规文件，

if (filp-&gt;f\_count &gt; 1) {

filp-&gt;f\_count--;

return 0;

! 对应的文件描述符数减一，直接返回 0

}

        if (filp->f\_op && filp->f\_op->release)            filp->f\_op->release(inode,filp);

! 如果对应的文件操作函数指针存在并且存在release函数，则调用之

filp-&gt;f\_count--;

        filp->f\_inode = NULL;

! 置空文件的I节点

iput(inode);

! 释放I节点，定义于Fs/inode.c

return 0;

! 返回 0

463 }

! 关闭文件描述符

× fd=要关闭的文件描述符

465 asm linkage int sys\_close(unsigned int fd)

```

466 {
467     struct file * filp;
468
469     if (fd >= NR_OPEN)
470         return -EBADF;
    ! 如果文件描述符大于一个进程所能打开的最大文件数，则返回-EBADF
471     FD_CLR(fd, &current->close_on_exec);
    ! 设置执行时关闭句柄位图，并复位对应的bit位
472     if (!(filp = current->filp[fd]))
473         return -EBADF;
    ! 如果，当前进程中的对应的文件描述符没有被使用，直接返回-EBADF
474     current->filp[fd] = NULL;
    ! 设置对应的文件描述符指针为NULL
475     return (close_fp (filp, fd));
    ! 定义于本文件，关闭对应的文件描述符
476 }

```

## Fs/devices.c (部分代码)

```

! 注册字符设备
× major=主字符设备号
× name=对应的名字
× fops=对应 major 号字符设备的操作函数结构
45 int register_chrdev(unsigned int major, const char * name, struct file_operations *fops)
46 {
47     if (major >= MAX_CHRDEV)
48         return -EINVAL;
    ! 如果主设备号大于MAX_CHRDEV，则返回-EINVAL
49     if (chrdevs[major].fops)
50         return -EBUSY;
    ! 如果对应于该主设备号的文件操作指针被占用，则直接返回 -EBUSY
51     chrdevs[major].name = name;
52     chrdevs[major].fops = fops;
    ! 设置处理程序
53     return 0;
    ! 成功返回 0
54 }
55

```

! 注册块设备  
 × major=主设备号  
 × name=设备名  
 × fops=对应于设备操作的函数操作结构指针

```

56 int register_blkdev(unsigned int major, const char * name, struct file_operations *fops)
57 {
58     if (major >= MAX_BLKDEV)
      ! 如果主设备号大于MAX_BLKDEV
59         return -EINVAL;
      ! 则返回-EINVAL
60     if (blkdevs[major].fops)
      ! 如果对应于该主设备号的文件操作指针
61         return -EBUSY;
      ! 被占用，则直接返回 -EBUSY
62     blkdevs[major].name = name;
63     blkdevs[major].fops = fops;
      ! 设置处理程序
64     return 0;
      ! 成功返回 0
65 }
```

## Fs/minix/inode.c (部分代码)

! 读 Minix 文件系统的超级块  
 × s=超级块指针

```

115 struct super_block *minix_read_super(struct super_block *s,void *data,
116                                         int silent)
117 {
118     struct buffer_head *bh;
119     struct minix_super_block *ms;
120     int i,dev=s->s_dev,block;
121
122     if (32 != sizeof (struct minix_inode))
123         panic("bad i-node size");
      ! 如果minix节点头大小不为 32 则死记。定义于Kernel/panic.c
124     lock_super(s);
      ! 为超级缓冲加锁
125     if (!(bh = bread(dev,1,BLOCK_SIZE)))
126         s->s_dev=0;
127         unlock_super(s);
```

```

128         printk("MINIX-fs: unable to read superblock\n");
129         return NULL;
    ! 读 dev 设备上的超级块（第一块），放入高速缓冲区中，如果失败
    ! 则返回NULL
130     }
131     ms = (struct minix_super_block *) bh->b_data;
    ! 把所取得的缓冲区数据强转后放入ms中
132     s->u.minix_sb.s_ms = ms;
133     s->u.minix_sb.s_sbh = bh;
134     s->u.minix_sb.s_mount_state = ms->s_state;
135     s->s_blocksize = 1024;
136     s->s_blocksize_bits = 10;
137     s->u.minix_sb.s_ninodes = ms->s_ninodes;
138     s->u.minix_sb.s_nzones = ms->s_nzones;
139     s->u.minix_sb.s_imap_blocks = ms->s_imap_blocks;
140     s->u.minix_sb.s_zmap_blocks = ms->s_zmap_blocks;
141     s->u.minix_sb.s_firstdatazone = ms->s_firstdatazone;
142     s->u.minix_sb.s_log_zone_size = ms->s_log_zone_size;
143     s->u.minix_sb.s_max_size = ms->s_max_size;
144     s->s_magic = ms->s_magic;
    ! 从第 132 行到这把从文件系统超级块读出的数据放入对应于传入的超级块中
145     if (s->s_magic == MINIX_SUPER_MAGIC) {
146         s->u.minix_sb.s_dirsize = 16;
147         s->u.minix_sb.s_nameLEN = 14;
    ! 对应于老的Minix文件系统
148     } else if (s->s_magic == MINIX_SUPER_MAGIC2) {
149         s->u.minix_sb.s_dirsize = 32;
150         s->u.minix_sb.s_nameLEN = 30;
    ! 对应于新的Minix文件系统
151     } else {
152         s->s_dev = 0;
153         unlock_super(s);
154         brelse(bh);
155         if (!silent)
156             printk("VFS: Can't find a minix filesystem on dev 0x%04x.\n", dev);
157         return NULL;
    ! 否则解锁该超级块，打印错误提示，返回NULL
158     }
159     for (i=0;i < MINIX_I_MAP_SLOTS;i++)
160         s->u.minix_sb.s_imap[i] = NULL;
    ! 初始化I节点位图缓冲块指针数组
161     for (i=0;i < MINIX_Z_MAP_SLOTS;i++)
162         s->u.minix_sb.s_zmap[i] = NULL;
    ! 初始化逻辑块位图缓冲块指针数组

```

```

163     block=2;
164     for (i=0 ; i < s->u.minix_sb.s_imap_blocks ; i++)
165         if ((s->u.minix_sb.s_imap[i]=bread(dev,block,BLOCK_SIZE)) != NULL)
166             block++;
167         else
168             break;
! 读出对应的block块，放入I节点位图
169     for (i=0 ; i < s->u.minix_sb.s_zmap_blocks ; i++)
170         if ((s->u.minix_sb.s_zmap[i]=bread(dev,block,BLOCK_SIZE)) != NULL)
171             block++;
172         else
173             break;
! 读出对应的block块，放入逻辑位图
174     if (block != 2+s->u.minix_sb.s_imap_blocks+s->u.minix_sb.s_zmap_blocks) {
175         for(i=0;i<MINIX_I_MAP_SLOTS;i++)
176             brelse(s->u.minix_sb.s_imap[i]);
177         for(i=0;i<MINIX_Z_MAP_SLOTS;i++)
178             brelse(s->u.minix_sb.s_zmap[i]);
179         s->s_dev=0;
180         unlock_super(s);
181         brelse(bh);
182         printk("MINIX-fs: bad superblock or unable to read bitmaps\n");
183         return NULL;
! 如果读出的位图逻辑块数不等于位图应该占有的逻辑块数，说明文件系
! 统位图信息有问题，超级块初始化失败，因此便释放申请的内存后返回 NULL
! brelse定义于Fs/buffer.c
184     }
185     set_bit(0,s->u.minix_sb.s_imap[0]->b_data);
186     set_bit(0,s->u.minix_sb.s_zmap[0]->b_data);
! 置 0, I节点位图以及逻辑位图的比特位
187     unlock_super(s);
188     /* set up enough so that it can read an inode */
189     s->s_dev = dev;
190     s->s_op = &minix_sops;
191     s->s_mounted = iget(s,MINIX_ROOT_INO);
! 读取 I 节点后，赋给 s_mounted，即该文件系统被安装到的 I 节点。
! 定义于Fs/inode.c本文件
192     if (!s->s_mounted) {
! 如果该文件系统没有被安装，则打印错误提示后返回NULL
193         s->s_dev = 0;
194         brelse(bh);
195         printk("MINIX-fs: get root inode failed\n");
196         return NULL;
197     }

```

```

198     if (!(s->s_flags & MS_RDONLY)) {
199         ms->s_state &= ~MINIX_VALID_FS;
200         bh->b_dirt = 1;
201         s->s_dirt = 1;
202     }
    ! 根据文件系统的被安装类型，设置其状态
203     if (!(s->u.minix_sb.s_mount_state & MINIX_VALID_FS))
204         printk ("MINIX-fs: mounting unchecked file system, "
205                 "running fsck is recommended.\n");
206     else if (s->u.minix_sb.s_mount_state & MINIX_ERROR_FS)
207         printk ("MINIX-fs: mounting file system with errors, "
208                 "running fsck is recommended.\n");
209     return s;
    ! 返回超级块
210 }
211

```

# I

## Init/main.c

### 概述

请参看“核心游记总结（1.0 核心）”这节！

### 代码分析

```

1 /*
2 * linux/init/main.c
3 *
4 * Copyright (C) 1991, 1992 Linus Torvalds
5 */
6
7 #include <stdarg.h>
8
9 #include <asm/system.h>

```

```

10 #include <asm/io.h>
11
12 #include <linux/types.h>
13 #include <linux/fcntl.h>
14 #include <linux/config.h>
15 #include <linux/sched.h>
16 #include <linux/tty.h>
17 #include <linux/head.h>
18 #include <linux/unistd.h>
19 #include <linux/string.h>
20 #include <linux/timer.h>
21 #include <linux/fs.h>
22 #include <linux/ctype.h>
23 #include <linux/delay.h>
24 #include <linux/utsname.h>
25 #include <linux/ioport.h>
26
27 extern unsigned long * prof_buffer;
28 extern unsigned long prof_len;
29 extern char edata, end;
30 extern char *linux_banner;
31 asm linkage void lcall7(void);
32 struct desc_struct default_ldt;
33
34 /*
35 * we need this inline - forking from kernel space will result
36 * in NO COPY ON WRITE (!!!), until an execve is executed. This
37 * is no problem, but for the stack. This is handled by not letting
38 * main() use the stack at all after fork(). Thus, no function
39 * calls - which means inline code for fork too, as otherwise we
40 * would use the stack upon exit from 'fork()'!
41 *
42 * Actually only pause and fork are needed inline, so that there
43 * won't be any messing with the stack from main(), but we define
44 * some others too.
45 */
46 #define NR_exit NR_exit
47 static inline syscall0(int,idle)
48 static inline syscall0(int,fork)
49 static inline syscall0(int,pause)
50 static inline syscall1(int,setup,void *,BIOS)
51 static inline syscall0(int,sync)
52 static inline syscall0(pid_t,setsid)
53 static inline syscall3(int,write,int,fd,const char *,buf,off_t,count)

```

```

54 static inline __syscall1(int,dup,int,fd)
55 static inline __syscall3(int,execve,const char *,file,char **,argv,char **,envp)
56 static inline __syscall3(int,open,const char *,file,int,flag,int,mode)
57 static inline __syscall1(int,close,int,fd)
58 static inline __syscall1(int, _exit,int,exitcode)
59 static inline __syscall3(pid_t,waitpid,pid_t,pid,int *,wait_stat,int,options)
60
61 static inline pid_t wait(int * wait_stat)
62 {
63     return waitpid(-1,wait_stat,0);
64 }
65
66 static char printbuf[1024];
67
68 extern int console_loglevel;
69
70 extern char empty_zero_page[PAGE_SIZE];
71 extern int vsprintf(char *,const char *,va_list);
72 extern void init(void);
73 extern void init_IRQ(void);
74 extern long kmalloc_init (long,long);
75 extern long blk_dev_init(long,long);
76 extern long chr_dev_init(long,long);
77 extern void floppy_init(void);
78 extern void sock_init(void);
79 extern long rd_init(long mem_start, int length);
80 unsigned long net_dev_init(unsigned long, unsigned long);
81 extern unsigned long simple strtoul(const char *,char **,unsigned int);
82
83 extern void hd_setup(char *str, int *ints);
84 extern void bmouse_setup(char *str, int *ints);
85 extern void eth_setup(char *str, int *ints);
86 extern void xd_setup(char *str, int *ints);
87 extern void mcd_setup(char *str, int *ints);
88 extern void st0x_setup(char *str, int *ints);
89 extern void tmc8xx_setup(char *str, int *ints);
90 extern void t128_setup(char *str, int *ints);
91 extern void generic_NCR5380_setup(char *str, int *intr);
92 extern void aha152x_setup(char *str, int *ints);
93 extern void sound_setup(char *str, int *ints);
94 #ifdef CONFIG_SBP_CD
95 extern void sbpcd_setup(char *str, int *ints);
96 #endif CONFIG_SBP_CD
97

```

```

98 #ifdef CONFIG_SYSVIPC
99 extern void ipc_init(void);
100#endif
101#ifndef CONFIG_SCSI
102 extern unsigned long scsi_dev_init(unsigned long, unsigned long);
103#endif
104
105/*
106 * This is set up by the setup-routine at boot-time
107 */
    ! 下面的值由引导程序在引导系统时设置
108#define PARAM empty_zero_page
109#define EXT_MEM_K (*(unsigned short *) (PARAM+2))
110#define DRIVE_INFO (*(struct drive_info_struct *) (PARAM+0x80))
    ! 第一块硬盘的参数表，大小 32Byte，在Boot/setup.s取得
111#define SCREEN_INFO (*(struct screen_info *) (PARAM+0))
    ! 当前屏幕的信息
112#define MOUNT_ROOT_RDONLY (*(unsigned short *) (PARAM+0x1F2))
113#define RAMDISK_SIZE (*(unsigned short *) (PARAM+0x1F8))
114#define ORIG_ROOT_DEV (*(unsigned short *) (PARAM+0x1FC))
    ! 根设备存储的位置，请看boot/bootsect.s中的代码行 447—448
115#define AUX_DEVICE_INFO (*(unsigned char *) (PARAM+0x1FF))
    ! 针设备的信息，以上的信息，存储在 empty_zero_page 中，
    ! empty_zero_page 共 4k，前 2k 放的是在 setup.s 中用 BIOS 获取
    ! 的系统配置值，具体的这些配置值，在 setup.s 中已经列出了，
    ! 后 2k 放的命令行启动参数，请参看head.s 的注释。
116
117 /*
118 * Boot command-line arguments
    ! 启动命令行参数
119 */
120#define MAX_INIT_ARGS 8
121#define MAX_INIT_ENVS 8
122#define COMMAND_LINE ((char *) (PARAM+2048))
    ! COMMAND_LINE 指向 empty_zero_page+2048 处
    ! empty_zero_page 定义于 Boot/head.s 的第 321 行
123
124 extern void time_init(void);
125
126 static unsigned long memory_start = 0; /* After mem_init, stores the */
127                         /* amount of free user memory */
128 static unsigned long memory_end = 0;
129 static unsigned long low_memory_start = 0;
130

```

```

131 static char term[21];
132 int rows, cols;
133
134 static char * argv_init[MAX_INIT_ARGS+2] = { "init", NULL, };
135 static char * envp_init[MAX_INIT_ENVS+2] = { "HOME=/", term, NULL, };
136
137 static char * argv_rc[] = { "/bin/sh", NULL };
138 static char * envp_rc[] = { "HOME=/", term, NULL };
139
140 static char * argv[] = { "-/bin/sh", NULL };
141 static char * envp[] = { "HOME=/usr/root", term, NULL };
142
143 struct drive_info_struct { char dummy[32]; } drive_info;
144 struct screen_info screen_info;
145
146 unsigned char aux_device_present;
147 int ramdisk_size;
148 int root_mountflags = 0;
149
150 static char fpu_error = 0;
151
152 static char command_line[80] = { 0, };
153
154 char *get_options(char *str, int *ints)
155 {
156     char *cur = str;
157     int i=1;
158
159     while (cur && isdigit(*cur) && i <= 10) {
160         ints[i++] = simple_strtoul(cur,NULL,0);
161         if ((cur = strchr(cur, ',')) != NULL)
162             cur++;
163     }
164     ints[0] = i-1;
165     return(cur);
166 }
167
168 struct {
169     char *str;
170     void (*setup_func)(char *, int *);
171 } bootsetups[] = {
172     { "reserve=", reserve_setup },
173 #ifdef CONFIG_INET
174     { "ether=", eth_setup },

```

```

175 #endif
176 #ifdef CONFIG_BLK_DEV_HD
177     { "hd=", hd\_setup },
178 #endif
179 #ifdef CONFIG_BUSMOUSE
180     { "bmouse=", bmouse\_setup },
181 #endif
182 #ifdef CONFIG_SCSI_SEAGATE
183     { "st0x=", st0x\_setup },
184     { "tmc8xx=", tmc8xx\_setup },
185 #endif
186 #ifdef CONFIG_SCSI_T128
187     { "t128=", t128\_setup },
188 #endif
189 #ifdef CONFIG_SCSI_GENERIC_NCR5380
190     { "ncr5380=", generic\_NCR5380\_setup },
191 #endif
192 #ifdef CONFIG_SCSI_AHA152X
193     { "aha152x=", aha152x\_setup },
194 #endif
195 #ifdef CONFIG_BLK_DEV_XD
196     { "xd=", xd\_setup },
197 #endif
198 #ifdef CONFIG_MCD
199     { "mcd=", mcd\_setup },
200 #endif
201 #ifdef CONFIG_SOUND
202     { "sound=", sound\_setup },
203 #endif
204 #ifdef CONFIG_SPCD
205     { "sbpcd=", sbpcd\_setup },
206 #endif CONFIG_SPCD
207     { 0, 0 }
208 };
209

```

! 根据命令行参数中给定的内容，调用对应的设置函数

```

210 int checksetup(char *line)
211 {
212     int i = 0;
213     int ints[11];
214
215     while (bootsetups[i].str) {
216         ! 遍历bootsetups
217         int n = strlen(bootsetups[i].str);

```

```

217         if (!strcmp(line,bootsetups[i].str,n)) {
        ! 判断在命令行参数中
        ! 是否定义了和bootsetups有相同的str,有则调用对应的函数
218             bootsetups[i].setup_func(get_options(line+n,ints), ints);
219             return(0);
220         }
221         i++;
222     }
223     return(1);
224 }
225
226 unsigned long loops_per_sec = 1;
227
        ! 延迟校准 (获得时钟jiffies与CPU主频ticks的延迟)
228 static void calibrate_delay(void)
229 {
230     int ticks;
231
232     printk("Calibrating delay loop.. ");
        ! 打印延迟校准循环
233     while (loops_per_sec <= 1) {
        ! loops_per_sec左移动 1 位
234         ticks = jiffies;
        ! 把时钟滴答数赋给ticks
235         delay(loops_per_sec);
        ! 延时loops_per_sec
236         ticks = jiffies - ticks;
        ! 计算执行上面的延时后, 经过了多少ticks
237         if (ticks >= HZ) {
            ! 如果大于等于定义的系统时钟滴答频率
238             __asm__("mull %1 ; divl %2"
239                 :"=a" (loops_per_sec)
240                 :"d" (HZ),
241                 "r" (ticks),
242                 "" (loops_per_sec)
243                 :"dx");
244             printk("ok - %lu.%02lu BogoMips\n",
            ! 打印上面计算出的值
245                 loops_per_sec/500000,
246                 (loops_per_sec/5000) % 100);
247             return;
        ! 直接返回

```

```

248         }
249     }
250     printk("failed\n");
    ! 否则打印校准失败
251 }
252
253
254 /*
255 * This is a simple kernel command line parsing function: it parses
256 * the command line, and fills in the arguments/environment to init
257 * as appropriate. Any cmd-line option is taken to be an environment
258 * variable if it contains the character '='.
259 *
260 */
261 * This routine also checks for options meant for the kernel - currently
262 * only the "root=XXXX" option is recognized. These options are not given
263 * to init - they are for internal kernel use only.
264 */
    ! 这是个简单的内核命令行解析函数：它解析命令行参数，并且
    ! 正确的填充进 init 的参数环境。任何命令行选项可以用
    ! “=” “附给环境变量值

265 static void parse_options(char *line)
266 {
267     char *next;
268     char *devnames[] = { "hda", "hdb", "sda", "sdb", "sdc", "sdd",
269                         "sde", "fd", "xda", "xdb", NULL };
    ! 各种设备名
270     int devnums[] = { 0x300, 0x340, 0x800, 0x810, 0x820,
271                     0x830, 0x840, 0x200, 0xC00, 0xC40, 0 };
    ! 对应于上面设备的编号
272     int args, envs;
273
274     if (!*line)
275         return;
    ! 如果在copy_options中并没有任何值被拷贝到line中，则直接退出
276     args = 0;
277     envs = 1; /* TERM is set to 'console' by default */
278     next = line;
279     while ((line = next) != NULL) {
        if ((next = strchr(line, ' ')) != NULL)
            *next++ = 0;
    ! shrchr 函数和标准 C 库函数相当，这里不解释。
    ! 在命令行参数中查找空格字符，找到后把该位置置
    ! 0，同时跳过置 0 的字符

```

```

280      /*
281 * check for kernel options first..
282 */  

283     if (!strcmp(line,"root=",5)) {  

284         int n;  

285         line += 5;  

286         if (strcmp(line,"/dev/",5)) {  

287             ROOT_DEV = simple strtoul(line,NULL,16);  

288             continue;  

289         }  

290         line += 5;  

291         ! 跳过/dev/  

292         for (n = 0 ; devnames[n] ; n++) {  

293             ! 循环查找devnames,  

294             int len = strlen(devnames[n]);  

295             ! 看是以那个设备做为根设备的  

296             if (!strcmp(line,devnames[n],len)) {  

297                 ROOT_DEV =  

298                 devnums[n]+simple strtoul(line+len,NULL,16);  

299                 break;  

300                 ! 找到后, 转换后作为根系统设备  

301             }  

302             }  

303             ! else if (!strcmp(line,"ro"))  

304                 root_mountflags |= MS_RDONLY;  

305             ! 设置为只读  

306             else if (!strcmp(line,"rw"))  

307                 root_mountflags &= ~MS_RDONLY;  

308             ! 可读写  

309             else if (!strcmp(line,"debug"))  

310                 console_loglevel = 10;  

311             ! 打印log的级别为 10, 表示写在屏幕上  

312             else if (!strcmp(line,"no387")) {  

313                 hard_math = 0;  

314                 __asm__ ("movl %%cr0,%%eax\n\t"  

315                         "orl $0xE,%%eax\n\t"

```

```

! 判断是否定义了协处理器，如果参数为“no387”表示没有
! 协处理器，这时需要复位协处理器存在标志位1(MP)，所以用了
! "orl $0xE,%%eax\n\t"，这条指令来复位它。
308                                     "movl %%eax,%%cr0\n\t"  :  : :
"ax" );
! 复位后，再送入寄存器CR0 中
309         } else
310             checksetup(line);
! 当并不是上面代码中定义的值时，便调用它，
! 它会调用它找到的值的对应的函数。
! 定义于本文件
311     /*
312 * Then check if it's an environment variable or
313 * an option.
! 接下来检查是否有其他的环境值或者选项
314 */
315     if (strchr(line,'=') {
316         if (envs >= MAX_INIT_ENVS)
317             break;
! 如果envs大于8直接退出循环
318         envp_init[envs] = line;
319     } else {
320         if (args >= MAX_INIT_ARGS)
321             break;
322         argv_init[args] = line;
323     }
324 }
325 argv_init[args+1] = NULL;
! 置空init的参数值的结尾
326 envp_init[envs+1] = NULL;
! 置空init的环境值的结尾
327 }
328

```

! 如果在命令行参数中定义了内存的大小的(
! 形如: mem=0xAB), 则转换成整数值并且
! 作为内存值附给 memory\_end,
! 接着在把剩余的命令行参数拷贝到 to 中(注意跳过已经
! 跳过了型如 mem=0xAB)
× to = 拷贝的目的地址
× from = 源地址
329 static void copy\_options(char \* to, char \* from)
330 {

```
331     char c = ' ';
332
333     do {
334         if (c == '' && !memcmp("mem=", from, 4))
335             //如果c=”或从from开始的 4 个字节是”mem=”的话
336             memory_end = simple strtoul(from+4, &from, 0);
337             // 定义于 kernel/vsprintf.c, 从命令行得到
338             // 内存的大小附给memory_end
339
340     static void copro_timeout(void)
341     {
342         fpu_error = 1;
343         timer_table[COPRO_TIMER].expires = jiffies+100;
344         timer_active |= 1<<COPRO_TIMER;
345         printk("387 failed: trying to reset\n");
346         send_sig(SIGFPE, last_task_used_math, 1);
347         outb_p(0,0xf1);
348         outb_p(0,0xf0);
349     }
350 }
```

```
! start_kernel(void)在 head.s 中调用，这也整个核心的引爆器
! asmlinkage 定义为如果是 c++ 编译器来编译时，就为"extern c"
! 否则为空

351 asmlinkage void start_kernel(void)
352 {
353 /*
354 * Interrupts are still disabled. Do necessary setups, then
355 * enable them
356 */
! 中断仍然被关闭着，在做完必要的设置后，才会打开他们。

357     set_call_gate(&default_ldt,lcall7);
! set_call_gate 定义于 Include/asm/system.h, lcall7 定义于 Kernel/sys_call.s
! 将 lcall7 设置到 default_ldt 中
! default_ldt 定义于本文件 (Init/main.c) 中
! 这里调用 set_call_gate 是为了让 Linux 支持 Intel 二进制兼容规范标准
! 的版本 2 (iBCS2)，这样的话 Linux 便可以支持类 Unix 及 Solaris 中的
! 可执行文件了，不过在的 1.0 核心中还未真正支持 iBCS2。
358     ROOT_DEV = ORIG_ROOT_DEV;
```

```

    ! 把保存的根设备号，置于 ROOT_DEV 中,
    ! ROOT_DEV 定义在 Fs/super.c 中
    ! ORIG_ROOT_DEV 定义于本文件的第 114 行
359     drive_info = DRIVE_INFO;
    ! 把硬盘参数表的信息保存进 driver_info 中，DRIVER_INFO 定义
    ! 于本文件的第 110 行
360     screen_info = SCREEN_INFO;
    ! 驱动器信息及显卡配置信息（可查考boot/setup.s中的图boot/setup.s-1）
361     aux_device_present = AUX_DEVICE_INFO;
    ! aux设备信息（可查考boot/setup.s中的图boot/setup.s-1）
362     memory_end = (1<<20) + (EXT_MEM_K<<10);
    ! 计算计算机所拥有内存大小（内存大小=1M+扩展内存（kb）X1024）
363     memory_end &= PAGE_MASK;
    ! 让计算所得的物理内存按页对齐，这里的做法会最多忽略掉 4K 的物理内存
    ! （因为操作系统对内存的分配是按每 4k 为一个单位来分配的）
364     ramdisk_size = RAMDISK_SIZE;
    ! 虚拟内存的大小，定义于Boot/bootsect.s中，被拷贝到empty_zero_page
365     copy_options(command_line,COMMAND_LINE);
    ! copy_options 定义于本文件
    ! COMMAND_LINE 定义于本文件
    ! 从 COMMND_LINE 处开始拷贝到 command_line 中,其中如果定义了内存的
    ! 大小则重新设定 memory_end
366 #ifdef CONFIG_MAX_16M
367     if (memory_end > 16*1024*1024)
368         memory_end = 16*1024*1024;
    ! 如果在配置核心时定义了最大的内存为 16M，纵然实际内存或命令行参数中设置
    ! 的内存值大于 16M 时，仍然设置内存为 16M
369 #endif
370     if (MOUNT_ROOT_RDONLY)
    ! 如果根文件系统为只读，对应于 Boot/bootsect.s 的第 481 到 482 定义的值
    ! 形如:
    ! root_flags:
    ! .word CONFIG_ROOT_RDONLY
371         root_mountflags |= MS_RDONLY;
    ! MS_RDONLY 定义于include/linux/fs.h
372         if ((unsigned long)&end >= (1024*1024)) {
    ! 如果编译出的内核大于或等于 1M (end 对应于核心的大小)
373             memory_start = (unsigned long) &end;
    ! 则让可用内存的的起始位置为内核的大小
374             low_memory_start = PAGE_SIZE;
    ! low_memory_start = 4K
375         } else {
    ! 否则就让可用内存的的起始位置为 1M
376             memory_start = 1024*1024;

```

```

377         low_memory_start = (unsigned long) &end;
        ! low_memory_start = 内核大小
378     }
379     low_memory_start = PAGE_ALIGN(low_memory_start);
        ! 让low_memory_start按页对齐
380     memory_start = paging_init(memory_start,memory_end);
        ! 把系统可用内存分页 (可用内存=系统所拥有的最大内存值 - memory_start)
        ! 定义于mm/memory.c
381     if (strcmp((char*)0xFFFFD9, "EISA", 4) == 0)
382         EISA_bus = 1;
        ! strcmp 一般 C 语言书籍都有说明, 这里不再注释。
        ! 比较 0xFFFFD9 处的 4 个字节是否是'EISA', 是则让 EISA_bus = 1
        ! 这里的 0xFFFFD9,即为真正的物理地址值, 因为
        ! 加上基址 0xC0000000 后, 即是定义在 head.s
        ! 页目录的第 768 项, 0xFF=256,即映射内存的第一
        ! 256*4k 到 256*4k + 4k, 再加上偏移地址值 0xFD9
        ! 即 256*4k + 0xFD9,即比 1M 小 39 个字节地址
        ! 处开始的 4 个字节如果是 EISA,则说明是 EISA 总线
383     trap_init();
        ! 硬件中断向量初始化, 定义于kernel/traps.c
384     init_IRQ();
        ! IRQ中断初始化, 定义于kernel/irq.c
385     sched_init();
        ! 初始化任务 0, 即调度程序。定义于kernel/sched.c
386     parse_options(command_line);
        ! 解析在上面 365 行代码处拷贝的命令行参数。定义于本文件中
387 #ifdef CONFIG_PROFILE
        ! 如果定义了 PROFILE, 则在主内存中为 profile 划分出一些内存
        ! 同时可用主内存减小
388     prof_buffer = (unsigned long *) memory_start;
389     prof_len = (unsigned long) &end;
390     prof_len >= 2;
391     memory_start += prof_len * sizeof(unsigned long);
392 #endif
393     memory_start = kmalloc_init(memory_start,memory_end);
        ! 检查系统定义的分页信息是否正确, 定义于Mm/kmalloc.c
394     memory_start = chr_dev_init(memory_start,memory_end);
        ! 字符设备初始化, 定义于Drivers/char/mem.c
395     memory_start = blk_dev_init(memory_start,memory_end);
        ! 块设备初始化, 定义于Drivers/block/ll_rw_blk.c
396     sti();
        ! 开中断
397     calibrate_delay();
        ! 延迟校准, 定义于本文件中

```

```

398 #ifdef CONFIG_INET
399     memory_start = net_dev_init(memory_start,memory_end);
    ! 初始化网络设备，定义于Drivers/net/net_init.c
400 #endif
    ! 忽略SCSI设备
401 #ifdef CONFIG_SCSI
402     memory_start = scsi_dev_init(memory_start,memory_end);
403 #endif
404     memory_start = inode_init(memory_start,memory_end);
    ! I节点初始化，定义于Fs/inode.c
405     memory_start = file_table_init(memory_start,memory_end);
    ! 初始化文件描述符表，定义于Fs/file_table.c
406     mem_init(low_memory_start,memory_start,memory_end);
    ! 物理内存的初始化，定义于Mm/memory.c
407     buffer_init();
    ! 高速缓冲初始化，定义于Fs/buffer.c
408     time_init();
    ! 设置开机启动时间，定义于Kernel/time.c
409     floppy_init();
    ! 软驱初始化，定义于Drivers/block/floppy.c
410     sock_init();
411 #ifdef CONFIG_SYSVIPC
    ! 忽略IPC（进程间通讯）
412     ipc_init();
413 #endif
414     sti();           ! 开中断
415
416     /*
417 * check if exception 16 works correctly.. This is truly evil
418 * code: it disables the high 8 interrupts to make sure that
419 * the irq13 doesn't happen. But as this will lead to a lockup
420 * if no exception16 arrives, it depends on the fact that the
421 * high 8 interrupts will be re-enabled by the next timer tick.
422 * So the irq13 will happen eventually, but the exception 16
423 * should get there first..
424 */
    ! 检查异常 16 是否工作正常。这块代码是有害的：它禁止高的 8 号
    ! 中断以禁止 irq13 号中断的产生。假如没有 16 位异常的发生话，这块
    ! 代码会导致机器死锁，8 号中断会在下一次的时间滴答时重新开启。以
    ! 使得 irq13 中断又可以发生，但是异常 16 必须先发生
425     if (hard_math) {
        ! 如果，系统的 CPU 支持协处理器，则做该块代码。
        ! hard_math在boot/head.s中被设置
426     unsigned short control_word;

```

```

427
428     printk("Checking 386/387 coupling... ");
429     timer_table[COPRO_TIMER].expires = jiffies+50;
430     timer_table[COPRO_TIMER].fn = copro_timeout;
431     timer_active |= 1<<COPRO_TIMER;
432     __asm__("clts ; fninit ; fnstcw %0 ; fwait":":=m" (*&control_word));
    ! 首先用 clts 指令清除 CR0 中的任务已交换标志 (TS)
    ! 用 fninit 指令初始化协处理器
    ! fnstcw 把协处理器的控制字写到 control_word 中
    ! fwait向协处理器发送WAIT指令
433     control_word &= 0xffc0;
    ! 清零协处理器控制字的低 6 位
434     __asm__("fldcw %0 ; fwait":":m" (*&control_word));
    ! fldcw 把 control_word 作为控制字装入协处理器
    ! fwait向协处理器发送WAIT指令
435     outb_p(inb_p(0x21) | (1 << 2), 0x21);
    ! 屏蔽主中断控制器 (8259A) 的第 3 脚的中断请求，该脚连着从中断控制器
    ! 这样也起到了屏蔽irq13 号请求的效果
436     __asm__("fldz ; fldl ; fdiv %st,%st(1) ; fwait");
    ! fldz 将 0.0 压入堆栈，fldl 将 1.0 压入堆栈
    ! fdiv 把 st/st1 后的结果放入 st 中
    ! fwait向协处理器发送WAIT指令
437     timer_active &= ~(1<<COPRO_TIMER);
438     if (!fpu_error)
439         printk("Ok, fpu using %s error reporting.\n",
440               ignore_irq13?"exception 16":"irq13");
441 }
442 #ifndef CONFIG_MATH_EMULATION
    ! 如果，系统的 CPU 不支持协处理器并且在配置核心时也没有打开
    ! 软件模拟协处理器的选项，则打印错误后，死机！
443 else {
444     printk("No coprocessor found and no math emulation present.\n");
445     printk("Giving up.\n");
446     for (;;) ;
447 }
448 #endif
449
450     system_utsname.machine[1] = '0' + x86;
    ! x86 在boot/head.s中设置
451     printk(linux_banner);
    ! 打印 linux_banner，	printk 定义于 Kernel/printk.c
    ! linux_banner定义于tools/version.h
452
453     move_to_user_mode();

```

```

! 切换到用户模式, 定义于include/asm/system.h
454     if (!fork())      /* we count on this going ok */
        ! 我们全靠它了
    ! 我们可以看到 fork(), 在整个核心代码中并没有对应的定义。
    ! (其实它对应于 sys_fork(), 定义于 Kernel/fork.c)
    ! 它是在那定义的呢?
    ! 请看本文件的第 48 行代码 (如下)
    ! 48 static inline __syscall0(int,fork)
    ! 请跳到Include/linux/unistd.h

455     init();
    ! init()定义于本文件 480 行
    ! 它其实是系统的init进程

456 /*
457 * task[0] is meant to be used as an "idle" task: it may not sleep, but
458 * it might do some general things like count free pages or it could be
459 * used to implement a reasonable LRU algorithm for the paging routines:
460 * anything that can be useful, but shouldn't take time from the real
461 * processes.
462 *
463 * Right now task[0] just does a infinite idle loop.
464 */
    ! 任务 0 意味着它是个 “idle” 任务: 它不能去睡眠, 但是它可以做些标准的
    ! 事情比如计算空闲页面或者为分页程序执行一个合理的 LRU 算法: 任何有
    ! 益处的事情, 但是它不能获取真正进程的时间。
    ! 当前的任务 0 只执行一个无限的 idle 循环。

465     for(;;)
466         idle();
    ! 根据 static inline __syscall0(int,idle)
    ! idle 对应于 sys_idle (), 该函数定义在 Mm/swap.c 中 (本书没有列出), 它
    ! 什么也不做! 只是设上要调用的标志后直接返回!

467 }
468
469 static int printf(const char *fmt, ...)
470 {
471     va_list args;
472     int i;
473
474     va_start(args, fmt);
475     write(1,printbuf,i=vsprintf(printbuf, fmt, args));
476     va_end(args);
477     return i;
478 }
479
    ! init进程

```

```

480 void init(void)
481 {
482     int pid,i;
483
484     setup((void *) &drive_info);
        ! 我们可以看到 setup(), 在整个核心代码中并没有对应的定义。
        ! 它是在那定义的呢?
        ! (其实它对应于 Drivers/block/genhd.c 的 sys_setup ())
        ! 请看本文件的第 50 行代码 (如下)
        ! static inline _syscall1(int,setup,void *,BIOS)
        ! 请跳到Include/linux/unistd.h
485     sprintf(term, "TERM=con%dx%d", ORIG_VIDEO_COLS, ORIG_VIDEO_LINES);
        ! 格式化控制台的配置信息。不作注释
486     (void) open("/dev/tty1",O_RDWR,0);
        ! 以只读方式打开/dev/tty1, 定义于lib/open.c
487     (void) dup(0);
        ! 复制文件句柄, 这里产生文件句柄 1
        ! 为什么呢?
        ! 因为在上面的 open 代码中, 占用了当前进程文件描述符数组的第 0 项
        ! 所以这里的 1 就是 1 了!
488     (void) dup(0);
        ! 复制文件句柄, 这里产生文件句柄 2
        ! 定义于Fs/fcntl.c
489
490     execve("/etc/init",argv_init,envp_init);
        ! 执行/etc/init, execve 对应于 sys_execve, 定义于Fs/exec.c
491     execve("/bin/init",argv_init,envp_init);
        ! 执行/bin/init, execve 对应于 sys_execve, 定义于Fs/exec.c
492     execve("/sbin/init",argv_init,envp_init);
        ! 执行/sbin/init, execve 对应于 sys_execve, 定义于 Fs/exec.c
        ! 从 490 行到 492 行, 只要有一个成功, 就不会做接下来得代码了
        ! 比如: 490 行成功了 (即执行/etc/init 成功, 则下面得代码就不会被执行了)
        ! 如果 491 行成功了 (即执行/bin/init 成功, 则下面得代码就不会被执行了)
        ! 如果一个都没有成功 (可能系统中并不存在这些可执行文件)
        ! 则会做下面得代码。
493     /* if this fails, fall through to original stuff */
494
495     if (!(pid=fork())) {
        ! 调用 fork 创建新得进程, 并关闭句柄 0, 并以只读方式打开"/etc/rc"文件, 如果
        ! 失败则调用 exit 退出, 否则接着执行"/bin/sh"程序, 请注意这时得标准输入是
        ! /etc/rc, 原因是 496 行得 close 函数, 已经关闭了当前进程得 0 号句柄
496         close(0);
        ! 关闭文件句柄, 对应于 sys_close, 定义于Fs/open.c
497         if (open("/etc/rc",O_RDONLY,0))

```

```

498         _exit(1);
    ! 以只读方式打开/etc/rc, 定义于 lib/open.c, 如果失败则退出
    ! _exit对应于sys_exit, 定义于Kernel/exit.c
499         execve("/bin/sh",argv,envp);
500         _exit(2);
    ! open 对应于 sys_open, 定义于 lib/open.c
    ! _exit 对应于 sys_exit, 定义于 Kernel/exit.c
    ! execve对应于sys_execve, 定义于Fs/exec.c
501     }
502     if (pid>0)
503         while (pid != wait(&i))
504             /* nothing */;
    ! 从 502 行到这是父进程执行得语句, 调用 wait 等待子进程结束
    ! wait对应于sys_waitpid, 定义于Kernel/exit.c
505     while (1) {
506         if ((pid = fork()) < 0) {
507             printf("Fork failed in init\n\r");
508             continue;
509         }
510         if (!pid) {
511             close(0);close(1);close(2);
512             setsid();
513             (void) open("/dev/tty1",O_RDWR,0);
514             (void) dup(0);
515             (void) dup(0);
516             _exit(execve("/bin/sh",argv,envp));
517         }
518         while (1)
519             if (pid == wait(&i))
520                 break;
521             printf("\n\rc child %d died with code %04x\n\r",pid,i);
522             sync();
    ! 同步文件系统, 定义于Fs/buffer.c
523     }
    ! 如果从 505 行到这得代码被执行了, 则说明上面创建得子进程得执行已经停止
    ! 了, 这块代码会在创建一个子进程, 如果还有错误发生, 则打印错误提示后,
    ! 继续创建新得子进程。
    ! 如果创建得子进程成功执行了, 则会在该子进程中打开/dev/tty1,执行/bin/sh 程
    ! 序, 而父进程则继续等待, 直到子进程结束, 如果子进程结束了, 则打印提示
    ! 信息后, 同步文件系统后, 继续再次执行!
524     _exit(0);
525 }
526

```

## Ipc/shm.c (部分代码)

```

477 /*
478 * remove the first attach descriptor from the list *shmdp.
479 * free memory for segment if it is marked destroyed.
480 * The descriptor is detached before the sleep in unmap_page_range.
481 */
    ! 从shmdp列表中去除首个附加描述符。
    ! 假如段对应的内存被标记为释放的话，将释放之。
    ! 这个描述符在 unmap_page_range 睡眠之前被分离。
(shm 即共享内存，在类 UNIX 平台上可以使用线程之前，要提高服务器性能的主要办法是在多个 CPU 上使用多进程，这样服务器可通过在不同的 CPU 上并行地执行代码来获取更快的速度，但是将服务器设计为多进程需要在进程之间有效的共享一定数量的信息，所以便有了共享内存，即 shm)
482 static void detach (struct shm_desc **shmdp)
483 {
484     struct shm_desc *shmd = *shmdp;
485     struct shmid_ds *shp;
486     int id;
487
488     id = (shmd->shm_sgn >> SHM_ID_SHIFT) & SHM_ID_MASK;
    ! 根据shm的签名，计算出id
489     shp = shm_segs[id];
    ! 用上述的id，从全局shm表中取得shp
490     *shmdp = shmd->task_next;      ! 修改shmdp指向下一个shm
491     for (shmdp = &shp->attaches; *shmdp; shmdp = &(*shmdp)->seg_next)
    ! 对找到的 shm，进行比较，是则跳到 found 处执行，同时修改
    ! shmdp指向下一个段
492     if (*shmdp == shmd) {
493         *shmdp = shmd->seg_next;
494         goto found;
495     }
    ! 没有找到则打印提示信息
496     printk("detach: shm segment (id=%d) attach list inconsistent\n",id);
497
498 found:
499     unmap_page_range (shmd->start, shp->shm_segsz); /* sleeps */
500     kfree_s (shmd, sizeof (*shmd));
501     shp->shm_lpid = current->pid;
    ! 当前进程的id送入shm中
502     shp->shm_dtime = CURRENT_TIME;

```

```

    ! 最后操作的时间送入shm中
503     if (--shp->shm_nattach <= 0 && shp->shm_perm.mode & SHM_DEST)
504         killseg (id); /* sleeps */
505     return;
    ! 返回
506 }
507 /*
508 /*
509 * detach and kill segment if marked destroyed.
510 * The work is done in detach.
511 */
    ! 分离并且杀死段标记假如被标识了的话
    ! 这个工作在 detach 中做
    × shmaddr=共享内存地址只
512 int sys_shmdt (char *shmaddr)
513 {
514     struct shm_desc *shmd, **shmdp;
515
516     for (shmdp = &current->shm; (shmd = *shmdp); shmdp=&shmd->task_next) {
        ! 循环遍历shm链表，如果找到了，则调用deattach释放之
517         if (shmd->start == (ulong) shmaddr) {
518             detach (shmdp);
519             return 0;
            ! 成功后，返回 0
520         }
521     }
522     return -EINVAL;
    ! 否则返回-EINVAL
523 }
524
525 /*
526 * detach all attached segments.
527 */
    ! 分离所有的附加段
    ! shm 是每个进程内部为了管理段而设置的
528 void shm_exit (void)
529 {
530     while (current->shm)
        ! 当前进程shm不为空，则detach之
531         detach(&current->shm);
        ! 定义于本文件中
532     return;
533 }

```

## ipc/sem.c (部分代码)

```

458 /*
459  * add semadj values to semaphores, free undo structures.
460  * undo structures are not freed when semaphore arrays are destroyed
461  * so some of them may be out of date.
462 */
    ! 添加调整过的信号值到信号量中，释放可撤消的结构。
    ! 当信号量数组被销毁时，可撤消结构没有被释放
    ! 因此它们当中的一些撤消结构可能是已经废弃的

463 void sem_exit(void)
464 {
465     struct sem_undo *u, *un = NULL, **up, **unp;
466     struct semid_ds *sma;
467     struct sem *sem = NULL;
468
469     for (up = &current->semun; (u = *up); *up = u->proc_next, kfree(u)) {
        ! 循环查找当前进程的可撤消信号量
470         sma = semary[u->semid % SEMMNI];
        ! 根据当前进程的信号量id从全局信号量数组中查找semid_ds
471         if (sma == IPC_UNUSED || sma == IPC_NOID)
472             continue;
        ! 如果上面找到的 sma=-1 或者-2，则下面的代码不做！
        ! -1 表示全局信号量数组中的第u->semid % SEMMNI项没有被使用，
        ! 也就是没有该信号
        ! -2 表示全局信号量数组中的第u->semid % SEMMNI项也没有被使用，
        ! 对于当前进程
        ! 也没有该信号量，不过是被其他进程使用了，并且在等待着释放或者分配内存
473         if (sma->sem_perm.seq != u->semid / SEMMNI)
474             continue;
        ! 如果查找到的sma的权限许可序列号与当前进程的 (semid / SEMMNI) 不等！
        ! 则下面代码不做（也就是sma中权限许可序列号是由semid / SEMMNI计算
        ! 而得的）
475         for (unp = &sma->undo; (un = *unp); unp = &un->id_next) {
476             if (u == un)
477                 goto found;
478         }
479         printk ("sem_exit undo list error id=%d\n", u->semid);
480         break;
        ! 从在全局信号量数组查找到的sma中，循环查找是否真的有我们要处理的进程
        ! (即：当前进程)，有则跳转到 found 处处理，没有则打印错误提示后，


```

! 跳出循环

481 found:

! 处理当前进程中的可撤消信号

482       \*unp = un->id\_next;

! 指向下一个撤消请求

483       if (!un->semadj)

484           continue;

! 如果调整过的信号值为 0，则

! 不做下面代码

485       while (1) {

486           if (sma->sem\_perm.seq != un->semid / SEMMNI)

487               break;

! 见 474 下注释

488           sem = &sma->sem\_base[un->sem\_num];

! 根据信号量索引定位信号量数组，俯给sem

489           if (sem->semval + un->semadj >= 0) {

490               sem->semval += un->semadj;

! sem 中当前已有值加上当前进程中

491               sem->sempid = current->pid;

! 把当前进程pid送入sempid

492               sma->sem\_otime = CURRENT\_TIME;

! 当前时间送入sem\_otime

493               if (un->semadj > 0 && sma->eventn)

494                   wake up (&sma->eventn);

! 定义于 Kernel/sched.c

! 如果当前进程中的信号值大于 0，并且存在等待队列，则唤醒等待队列

495               if (!sem->semval && sma->eventz)

496                   wake up (&sma->eventz);

! 定义于Kernel/sched.c

497               break;

! 如果sem中当前值是 0，并且存在等待队列，则唤醒等待队列

498       }

499       if (current->signal & ~current->blocked)

500           break;

! 如果当前进程的信号位没有被屏蔽，则退出

501       sem->semncnt++;

! semncnt加一，意味着等待处理该信号的进程又

! 多了一个

502       interruptible sleep on (&sma->eventn);

! 定义于Kernel/sched.c

! 进入可中断睡眠状态，让其他

! 进程处理该信号

503       sem->semncnt--;

```

    ! 返回到这说明其他进程已经处
    ! 理完，所以要减一
504     }
505     }
506     current->semun = NULL;
        ! 置当前进程的可撤消信号量
        ! 指针为空！
507     return;
508 }
509

```

## lbcos/emuulate.c

```

1 /*
2 * linux/abi/emuulate.c
3 *
4 * Copyright (C) 1993 Linus Torvalds
5 */
6
7 /*
8 * Emulate.c contains the entry point for the 'lcall 7,xxx' handler.
9 */
10
11 #include <linux/errno.h>
12 #include <linux/sched.h>
13 #include <linux/kernel.h>
14 #include <linux/mm.h>
15 #include <linux/stddef.h>
16 #include <linux/unistd.h>
17 #include <linux/segment.h>
18 #include <linux/ptrace.h>
19
20 #include <asm/segment.h>
21 #include <asm/system.h>
22
        ! 目前的 1.0 核心还不支持 iBCS2 的二进制代码
        ! 所以打印出提示消息后，发送SIGSEGV信号后结束
23 asmlinkage void iABI_emulate(struct pt_regs * regs)
24 {
25     printk("iBCS2 binaries not supported yet\n");
        ! 定义于kernel/printk.c
26     do_exit(SIGSEGV);

```

! 定义于kernel/exit.c  
27 }

## Include/linux/unistd.h

```

1 #ifndef __LINUX_UNISTD_H
2 #define __LINUX_UNISTD_H
3
4 /*
5 * This file contains the system call numbers and the syscalls
6 * macros
7 */
8
9 #define __NR_setup          0      /* used only by init, to get
system going */
10 #define __NR_exit           1
11 #define __NR_fork           2
12 #define __NR_read            3
13 #define __NR_write           4
14 #define __NR_open            5
15 #define __NR_close           6
16 #define __NR_waitpid         7
17 #define __NR_creat           8
18 #define __NR_link            9
19 #define __NR_unlink          10
20 #define __NR_execve          11
21 #define __NR_chdir           12
22 #define __NR_time            13
23 #define __NR_mknod           14
24 #define __NR_chmod           15
25 #define __NR_chown           16
26 #define __NR_break           17
27 #define __NR_oldstat          18
28 #define __NR_lseek            19
29 #define __NR_getpid          20
30 #define __NR_mount           21
31 #define __NR_umount          22
32 #define __NR_setuid          23
33 #define __NR_getuid          24
34 #define __NR_stime            25
35 #define __NR_ptrace           26

```

36 #define <u>NR_alarm</u>	27
37 #define <u>NR_oldfstat</u>	28
38 #define <u>NR_pause</u>	29
39 #define <u>NR_utime</u>	30
40 #define <u>NR_stty</u>	31
41 #define <u>NR_gtty</u>	32
42 #define <u>NR_access</u>	33
43 #define <u>NR_nice</u>	34
44 #define <u>NR_fftime</u>	35
45 #define <u>NR_sync</u>	36
46 #define <u>NR_kill</u>	37
47 #define <u>NR_rename</u>	38
48 #define <u>NR_mkdir</u>	39
49 #define <u>NR_rmdir</u>	40
50 #define <u>NR_dup</u>	41
51 #define <u>NR_pipe</u>	42
52 #define <u>NR_times</u>	43
53 #define <u>NR_prof</u>	44
54 #define <u>NR_brk</u>	45
55 #define <u>NR_setgid</u>	46
56 #define <u>NR_getgid</u>	47
57 #define <u>NR_signal</u>	48
58 #define <u>NR_geteuid</u>	49
59 #define <u>NR_getegid</u>	50
60 #define <u>NR_acct</u>	51
61 #define <u>NR_phys</u>	52
62 #define <u>NR_lock</u>	53
63 #define <u>NR_ioctl</u>	54
64 #define <u>NR_fcntl</u>	55
65 #define <u>NR_mpx</u>	56
66 #define <u>NR_setpgid</u>	57
67 #define <u>NR_ulimit</u>	58
68 #define <u>NR_oldolduname</u>	59
69 #define <u>NR_umask</u>	60
70 #define <u>NR_chroot</u>	61
71 #define <u>NR_ustat</u>	62
72 #define <u>NR_dup2</u>	63
73 #define <u>NR_getppid</u>	64
74 #define <u>NR_getpgrp</u>	65
75 #define <u>NR_setsid</u>	66
76 #define <u>NR_sigaction</u>	67
77 #define <u>NR_sgetmask</u>	68
78 #define <u>NR_ssetmask</u>	69
79 #define <u>NR_setreuid</u>	70

80 #define <u>NR_setregid</u>	71
81 #define <u>NR_sigsuspend</u>	72
82 #define <u>NR_sigpending</u>	73
83 #define <u>NR_sethostname</u>	74
84 #define <u>NR_setrlimit</u>	75
85 #define <u>NR_getrlimit</u>	76
86 #define <u>NR_getrusage</u>	77
87 #define <u>NR_gettimeofday</u>	78
88 #define <u>NR_settimeofday</u>	79
89 #define <u>NR_getgroups</u>	80
90 #define <u>NR_setgroups</u>	81
91 #define <u>NR_select</u>	82
92 #define <u>NR_symlink</u>	83
93 #define <u>NR_oldlstat</u>	84
94 #define <u>NR_readlink</u>	85
95 #define <u>NR_uselib</u>	86
96 #define <u>NR_swapon</u>	87
97 #define <u>NR_reboot</u>	88
98 #define <u>NR_readdir</u>	89
99 #define <u>NR_mmap</u>	90
100 #define <u>NR_munmap</u>	91
101 #define <u>NR_truncate</u>	92
102 #define <u>NR_ftruncate</u>	93
103 #define <u>NR_fchmod</u>	94
104 #define <u>NR_fchown</u>	95
105 #define <u>NR_getpriority</u>	96
106 #define <u>NR_setpriority</u>	97
107 #define <u>NR_profil</u>	98
108 #define <u>NR_statfs</u>	99
109 #define <u>NR_fstatfs</u>	100
110 #define <u>NR_ioperm</u>	101
111 #define <u>NR_socketcall</u>	102
112 #define <u>NR_syslog</u>	103
113 #define <u>NR_setitimer</u>	104
114 #define <u>NR_getitimer</u>	105
115 #define <u>NR_stat</u>	106
116 #define <u>NR_lstat</u>	107
117 #define <u>NR_fstat</u>	108
118 #define <u>NR_olduname</u>	109
119 #define <u>NR_iopl</u>	110
120 #define <u>NR_vhangup</u>	111
121 #define <u>NR_idle</u>	112
122 #define <u>NR_vm86</u>	113
123 #define <u>NR_wait4</u>	114

```

124 #define __NR_swapoff          115
125 #define __NR_sysinfo          116
126 #define __NR_ipc               117
127 #define __NR_fsync              118
128 #define __NR_sigreturn          119
129 #define __NR_clone              120
130 #define __NR_setdomainname      121
131 #define __NR_uname              122
132 #define __NR_modify_ldt         123
133 #define __NR_adjtimex           124
134 #define __NR_mprotect            125
135 #define __NR_sigprocmask         126
136 #define __NR_create_module       127
137 #define __NR_init_module          128
138 #define __NR_delete_module        129
139 #define __NR_get_kernel_syms     130
140 #define __NR_quotactl            131
141 #define __NR_getpgid             132
142 #define __NR_fchdir              133
143 #define __NR_bdfflush             134
144
145 extern int errno;
146
147 /* XXX - _foo needs to be __foo, while __NR_bar could be _NR_bar.
 */
148 #define __syscall0(type,name) \
149 type name(void) \
150 { \
151     long __res; \
152     __asm__ volatile ("int $0x80" \
153                     : "=a" (__res) \
154                     : "" (__NR_##name)); \
155     if (__res >= 0) \
156         return (type) __res; \
157     errno = -__res; \
158     return -1; \
159 }
! 针对 Init/main.c 中的
! static inline __syscall0(int,fork) 和这里的宏，我们可以得到如下的代码
    int fork(void) \
    { \
        long __res; \
        __asm__ volatile ("int $0x80" \
                         : "=a" (__res) \

```

```

        : "" (__NR_fork)); \
if (__res >= 0) \
    return (int) __res; \
errno = -__res; \
return -1; \
}

```

! 根据本文件的第 11 行可以知道 “`__NR_fork=2`”，所以上面的代码可以变为

```

int fork(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : ""); \
if (__res >= 0) \
    return (int) __res; \
errno = -__res; \
return -1; \
}

```

! 通过变换后的代码我们可以看到 `fork()` 是个宏，这样在 `Init/main.c` 调用它

! 时，就不会占用到堆栈，从而保证了任务 0 的堆栈的干净。

! 我们就从这开始分析 `fork()` 的代码！

! 该代码中 `eax` 既是输出寄出器又是输入寄存器，调用的指令是 `int 0x80`，该

! `int 0x80` 中断是整个系统调用的入口点（在 `Kernel/sched.c` 的 `sched_init`

! 中设置代码如：`set_system_gate(0x80,&system_call);` 所以当我们执行这

! 条指令时，便会调用 `system_call` 的代码，接着我们跳到 `Kernel/sys_call.s`

! 中分析 `system_call`

```

160
161 #define _syscall(type,name,atype,a) \
162 type name(atype a) \
163 { \
164     long __res; \
165     __asm__ volatile ("int $0x80" \
166         : "=a" (__res) \
167         : "" (__NR_##name), "b" ((long)(a))); \
168     if (__res >= 0) \
169         return (type) __res; \
170     errno = -__res; \
171     return -1; \

```

172 }

! 针对 Init/main.c 中的

! static inline \_syscall1(int,setup,void \*,BIOS) 和这里的宏,

! 我们可以得到如下的代码

```
int setup(void *BIOS) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_setup), "b" ((long)(a))); \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
}
```

! 根据本文件的第 11 行可以知道 “\_\_NR\_setup=0”，所以上面的代码可以变为

```
int setup(void *BIOS) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (0), "b" ((long)(a))); \
    if (__res >= 0) \
        return (type) __res; \
    errno = -__res; \
    return -1; \
}
```

! 通过变换后的代码我们可以看到 setup() 是个宏。

! 我们就从这开始分析 setup() 的代码！

! 该代码中 eax 既是输出寄出器又是输入寄存器，调用的指令是 int 0x80，该

! int 0x80 中断是整个系统调用的入口点(在 Kernel/sched.c 的 sched\_init

! 中设置代码如： set\_system\_gate(0x80,&amp;system\_call); )所以当我们执行这

! 条指令时，便会调用 system\_call 的代码，接着我们跳到 Kernel/sys\_call.s

! 中分析 system\_call

173

```
#define _syscall2(type,name,atype,a,btype,b) \
type name(atype a,btype b) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "" (__NR_##name), "b" ((long)(a)), "c" ((long)(b))); \
}
```

```

181 if (__res >= 0) \
182     return (type) __res; \
183 errno = -__res; \
184 return -1; \
185 }
186
187 #define _syscall3(type,name,atype,a,btype,b,ctype,c) \
188 type name(atype a,btype b,ctype c) \
189 { \
190 long __res; \
191 __asm__ volatile ("int $0x80" \
192         : "=a" (__res) \
193         : "" (__NR_##name), "b" ((long)(a)), "c" ((long)(b)), "d" \
194         ((long)(c))); \
195 if (__res>=0) \
196     return (type) __res; \
197 errno=-__res; \
198 return -1; \
199 }
200
201 #define _syscall4(type,name,atype,a,btype,b,ctype,c,dtype,d) \
202 type name (atype a, btype b, ctype c, dtype d) \
203 { \
204 long __res; \
205 __asm__ volatile ("int $0x80" \
206         : "=a" (__res) \
207         : "" (__NR_##name), "b" ((long)(a)), "c" ((long)(b)), \
208         "d" ((long)(c)), "S" ((long)(d))); \
209 if (__res>=0) \
210     return (type) __res; \
211 errno=-__res; \
212 }
213
214 #define
215 _syscall5(type,name,atype,a,btype,b,ctype,c,dtype,d,etype,e) \
216 type name (atype a,btype b,ctype c,dtype d,etype e) \
217 { \
218 long __res; \
219 __asm__ volatile ("int $0x80" \
220         : "=a" (__res) \
221         : "" (__NR_##name), "b" ((long)(a)), "c" ((long)(b)), \
222         "d" ((long)(c)), "S" ((long)(d)), "D" ((long)(e))); \
223 if (__res>=0) \

```

```

223         return (type) __res; \
224     errno=-__res; \
225     return -1; \
226 }
227
228 #endif /* _LINUX_UNISTD_H */
229

```

## Include/linux/sched.h

```

401 /*
402 * The wait-queues are circular lists, and you have to be *very* sure
403 * to keep them correct. Use only these two functions to add/remove
404 * entries in the queues.
405 */
    ! 这个等待队列是环行列表，你们不得不非常小心的保持它们的正确行。仅仅
    ! 使用这两个函数添加、删除到请求队列头部
406 extern inline void add_wait_queue(struct wait_queue ** p, struct wait_queue * wait)
407 {
408     unsigned long flags;
409
410 #ifdef DEBUG
    ! 在调试状态下
411     if (wait->next) {
412         unsigned long pc;
413         __asm__ __volatile__("call lf\n"
414             "l:\tpopl %0":"=r"(pc));
415         printk("add_wait_queue (%08x): wait->next = %08x\n",pc,(unsigned long)
416         wait->next);
416     }
    ! 打印该队列的下以队列地址
417 #endif
418     save_flags(flags);
    ! 保存flag寄存器的值，定义于Include/asm/system.h
419     cli();
    ! 关中断
420     if (!*p) {
    ! 如果，I节点中不存在等待队列
421         wait->next = wait;

```

```

422         *p = wait;
        ! 让*p指向当前进程
423     } else {
424         wait->next = (*p)->next;
425         (*p)->next = wait;
        ! 让*p->next指向当前进程
426     }
427     restore_flags(flags);
        ! 恢复flag
428 }

430 extern inline void remove_wait_queue(struct wait_queue ** p, struct wait_queue * wait)
431 {
432     unsigned long flags;
433     struct wait_queue * tmp;
434 #ifdef DEBUG
435     unsigned long ok = 0;
436 #endif
437
438     save_flags(flags);
        ! 保存flag寄存器
439     cli();
        ! 关中断
440     if ((*p == wait) &&
441 #ifdef DEBUG
442         (ok = 1) &&
443 #endif
444         ((*p = wait->next) == wait)) {
445         *p = NULL;
446     } else {
447         tmp = wait;
448         while (tmp->next != wait) {
449             tmp = tmp->next;
450 #ifdef DEBUG
451             if (tmp == *p)
452                 ok = 1;
453 #endif
454         }
455         tmp->next = wait->next;
456     }
457     wait->next = NULL;
        ! 置下一等待队列为空
458     restore_flags(flags);
        ! 恢复flag寄存器

```

```

459 #ifdef DEBUG
460     if (!ok) {
461         printk("removed wait_queue not on list.\n");
462         printk("list = %08x, queue = %08x\n", (unsigned long) p, (unsigned long) wait);
463         __asm__ ("call 1f\n1:tpopl %0":"=r" (ok));
464         printk("eip = %08x\n", ok);
465     }
466 #endif
467 }

537 #define SET_LINKS(p) do { unsigned long flags; \
538     save_flags(flags); cli(); \
    ! 保存寄出器及关中断
539     (p)->next_task = &init_task; \
540     (p)->prev_task = init_task.prev_task; \
541     init_task.prev_task->next_task = (p); \
542     init_task.prev_task = (p); \
543     restore_flags(flags); \
544     (p)->p_ysptr = NULL; \
545     if (((p)->p_osptr = (p)->p_pptr->p_cptr) != NULL) \
546         (p)->p_osptr->p_ysptr = p; \
547     (p)->p_pptr->p_cptr = p; \
548 } while (0)

```

Include/asm/system.h (部分代码)

```

! 从核心态切换进用户模式
6 #define move_to_user_mode() \
7 __asm__ __volatile__ ("movl %%esp,%%eax\n\t" \
    ! 保存堆栈指针esp到eax中
8 "pushl %0\n\t" \
    ! 将用户态数据段压入堆栈中
9 "pushl %%eax\n\t" \
    ! 将第 7 行保存的堆栈指针入栈
10 "pushfl\n\t" \
    ! 标志寄存器入栈
11 "pushl %1\n\t" \
    ! 将用户态代码段寄出器入栈
12 "pushl $1\n\t" \
    ! 将第 14 行代码处地址入栈,

```

```

13      "iret\n" \
! 中断返回，从而跳到第 14 行执行，因为在第 12 行已经将其地址入栈
14      "1:\tmovl %0,%eax\n\t" \
15      "mov %%ax,%%ds\n\t" \
16      "mov %%ax,%%es\n\t" \
17      "mov %%ax,%%fs\n\t" \
18      "mov %%ax,%%gs" \
19      :/* no outputs */:"i" (USER\_DS), "i" (USER\_CS):"ax")
20
21 #define sti() __asm__ __volatile__ ("sti": :"memory")
22 #define cli() __asm__ __volatile__ ("cli": :"memory")
23 #define nop() __asm__ __volatile__ ("nop")
24
25 /*
26 * Clear and set 'TS' bit respectively
27 */
28 #define clts() __asm__ __volatile__ ("clts")
29 #define stts() \
30 __asm__ __volatile__ ( \
31     "movl %%cr0,%eax\n\t" \
32     "orl $8,%eax\n\t" \
33     "movl %%eax,%cr0" \
34     :/* no outputs */ \
35     :/* no inputs */ \
36     :"ax")
37
38
39 extern inline int tas(char * m)
40 {
41     char res;
42
43     __asm__("xchgb %0,%1": "=q" (res),"=m" (*m):"" (0x1));
44     return res;
45 }
46
47 #define save_flags(x) \
48 __asm__ __volatile__("pushfl ; popl %0": "=r" (x):/* no input */ :"memory")
! 取eflags的值到x中
49
50 #define restore_flags(x) \
51 __asm__ __volatile__("pushl %0 ; popfl":/* no output */ :"r" (x):"memory")
! 把x做为eflag的值
52
53 #define iret() __asm__ __volatile__ ("iret": :"memory")

```

```

54
55 #define set_gate(gate_addr,type,dpl,addr) \
56 __asm__ __volatile__ ("movw %%dx,%%ax\n\t" \
    ! 将偏移地址的低 2 个字节与选择符组合成描述符的低 4 个字节放在eax中
57     "movw %2,%%dx\n\t" \
    ! 将类型标志字与偏移高字组合成高 4 个字节，放在edx中
58     "movl %%eax,%60\n\t" \
59     "movl %%edx,%1" \
    ! 分别设置门描述符的低 4 个字节和高 4 个字节
60     :"=m" (*(long *) (gate_addr))), \
61     "=m" (*(1+(long *) (gate_addr))) \
62     :"i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
63     "d" ((char *) (addr)), "a" (KERNEL_CS << 16) \
64     :"ax","dx")
65
66 #define set_intr_gate(n,addr) \
67     set_gate(&idt[n],14,0,addr)
    ! 设置中断描述符表(IDT)，类型是 14，特权级是 0
    ! 特权级 0,表示用户态不可以访问
68
69 #define set_trap_gate(n,addr) \
70     set_gate(&idt[n],15,0,addr)
    ! 设置中断描述符表(IDT)，类型是 15，特权级是 0
    ! 特权级 0,表示用户态不可以访问
71
72 #define set_system_gate(n,addr) \
73     set_gate(&idt[n],15,3,addr)
    ! 设置中断描述符表(IDT)，类型是 15，特权级是 3
    ! 特权级 0,表示用户态可以访问
74
75 #define set_call_gate(a,addr) \
76     set_gate(a,12,3,addr)
    ! 类型是 12，特权级是 3
77
78 #define set_seg_desc(gate_addr,type,dpl,base,limit) { \
79     *((gate_addr)+1) = ((base) & 0xff000000) | \
80         (((base) & 0x0ff0000)>>16) | \
81         (((limit) & 0xf0000) | \
82         ((dpl)<<13)) | \
83         (0x00408000) | \
84         (((type)<<8); \
85     *(gate_addr) = (((base) & 0x0000ffff)<<16) | \
86         (((limit) & 0x0ffff); \
87

```

```

88 #define set_tssldt_desc(n,addr,limit,type) \
89 __asm__ __volatile__ ("movw $" #limit ",%I\n\t" \
90     "movw %%ax,%2\n\t" \
91     "rorl $16,%%eax\n\t" \
92     "movb %%al,%3\n\t" \
93         ! 基地址高字中的低字节放入第 4 个字节中
94     "movb $" type ",%4\n\t" \
95     "movb $0x00,%5\n\t" \
96         ! 第 6 个字节设置为 0
97     "movb %%ah,%6\n\t" \
98         ! 基地址高字中的高字节放入第 7
99         ! 个字节中
100    "rorl $16,%%eax" \
101   :/* no output */ \
102   :"a" (addr+0xc0000000), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)), \
103   "m" (*(n+5)), "m" (*(n+6)), "m" (*(n+7)) \
104 )
105 ! 在全局表中设置任务状态段描述符和局部描述表描述符
106 #define set_tss_desc(n,addr) set_tssldt_desc((char *) (n),((int)(addr)),235,"0x89")
107 #define set_ldt_desc(n,addr,size) \
108   set_tssldt_desc((char *) (n),((int)(addr)),((size << 3) - 1),"0x82")

```

K

## Kernel/panic.c

```
1 /*  
2 *  linux/kernel/panic.c  
3 *  
4 *  Copyright (C) 1991, 1992  Linus Torvalds  
5 */  
6  
7 /*  
8 * This function is used through-out the kernel (include in mm and fs)  
9 * to indicate a major problem.  
10 */  
11 #include <stdarg.h>  
12  
13 #include <linux/kernel.h>
```

```

14 #include <linux/sched.h>
15
16 asmlinkage void sys_sync(void); /* it's really int */
17
18 extern int vsprintf(char * buf, const char * fmt, va_list args);
19
20 NORET_TYPE void panic(const char * fmt, ...)
21 {
22     static char buf[1024];
23     va_list args;
24
25     va_start(args, fmt);
26     vsprintf(buf, fmt, args);
    ! 格式化字符串, 不作解释, 大家可以查看各种 C 语言书籍
27     va_end(args);
28     printk(KERN_EMERG "Kernel panic: %s\n",buf);
    ! 打印错误消息
29     if (current == task[0])
30         printk(KERN_EMERG "In swapper task - not syncing\n");
    ! 如果当前进程是任务 0, 则直接打印错误提示
31     else
32         sys_sync();
    ! 同步文件系统, 定义于 Fs/buffer.c
33     for(;;)
    ! 死机
34 }
35

```

## Kernel/traps.c

! 中断向量初始化, 该函数比较简单, 把对应的中断向量处理函数设置到对应的中断  
! 向量号上去。  
! 无参数

```

197 void trap_init(void)
198 {
199     int i;
200
201     set_trap_gate(0,&divide_error);

```

```

! set_trap_gate 定义于 include/asm/system.h
202     set_trap_gate(1,&debug);
203     set_trap_gate(2,&nmi);
204     set_system_gate(3,&int3);      /* int3-5 can be called
from all */
205     set_system_gate(4,&overflow);

! set_system_gate 定义于 include/asm/system.h
206     set_system_gate(5,&bounds);
207     set_trap_gate(6,&invalid_op);
208     set_trap_gate(7,&device_not_available);
209     set_trap_gate(8,&double_fault);
210     set_trap_gate(9,&coprocessor_segment_overrun);
211     set_trap_gate(10,&invalid_TSS);
212     set_trap_gate(11,&segment_not_present);
213     set_trap_gate(12,&stack_segment);
214     set_trap_gate(13,&general_protection);
215     set_trap_gate(14,&page_fault);
216     set_trap_gate(15,&reserved);
217     set_trap_gate(16,&coprocessor_error);
218     set_trap_gate(17,&alignment_check);
219     for (_i=18; _i<48; _i++)
220         set_trap_gate(_i,&reserved);

! 将 18 到 48 的中断向量设为 reserved, 后面会根据需要修改它们。
221 }
222

```

## Kernel/irq.c (部分代码)

```

168 static void (*bad_interrupt[16])(void) = {
169     bad IRQ0_interrupt, bad IRQ1_interrupt,
170     bad IRQ2_interrupt, bad IRQ3_interrupt,
171     bad IRQ4_interrupt, bad IRQ5_interrupt,
172     bad IRQ6_interrupt, bad IRQ7_interrupt,
173     bad IRQ8_interrupt, bad IRQ9_interrupt,
174     bad IRQ10_interrupt, bad IRQ11_interrupt,
175     bad IRQ12_interrupt, bad IRQ13_interrupt,
176     bad IRQ14_interrupt, bad IRQ15_interrupt
177 };
178 /*

```

```

180 * Initial irq handlers.
181 */
182 static struct sigaction irq_sigaction[16] = {
183     { NULL, 0, 0, NULL }, { NULL, 0, 0, NULL },
184     { NULL, 0, 0, NULL }, { NULL, 0, 0, NULL },
185     { NULL, 0, 0, NULL }, { NULL, 0, 0, NULL },
186     { NULL, 0, 0, NULL }, { NULL, 0, 0, NULL },
187     { NULL, 0, 0, NULL }, { NULL, 0, 0, NULL },
188     { NULL, 0, 0, NULL }, { NULL, 0, 0, NULL },
189     { NULL, 0, 0, NULL }, { NULL, 0, 0, NULL },
190     { NULL, 0, 0, NULL }, { NULL, 0, 0, NULL }
191 };
    ! 开启对应于irq的中断，并且打开对应的中断允许标志位,
    ! 让其可处理该irq号的中断。
221 int irqaction(unsigned int irq, struct sigaction * new_sa)
222 {
223     struct sigaction * sa;
224     unsigned long flags;
225
226     if (irq > 15)
227         return -EINVAL;
    ! 如果传入的值大于15，则返回-EINVAL
228     sa = irq + irq_sigaction;
    ! 让sa指向从irq_sigaction中找到的irq信号处理结构
229     if (sa->sa_mask)
230         return -EBUSY;
    !如果有信号的屏蔽码，即有信号被阻塞，则返回-EBUSY
231     if (!new_sa->sa_handler)
232         return -EINVAL;
    ! 如果传入的新信号处理句柄为NULL，则返回-EINVAL
233     save_flags(flags);
    ! 保存eflags寄存器，定义于include/asm/system.h
234     cli();
    ! 关中断，定义于include/asm/system.h
235     *sa = *new_sa;
    ! 复制传入的信号结构到sa中
236     sa->sa_mask = 1;
    ! 置屏蔽SIGHUP位
237     if (sa->sa_flags & SA_INTERRUPT)
    ! 如果信号集中有SA_INTERRUPTDE的话
    ! (SA_INTERRUPT意味着该中断必须在关中断情况下才能执行)
238     set_intr_gate(0x20+irq.fast_interrupt[irq]);
    ! 则设置对应的fast_interrupt[irq]（意味着关中断才可以执行，所以
    ! 这里传入的处理函数必须快速处理该中断）

```

```

239      else
        ! 否则，设置成通常的中断处理函数即可！因为我们可以在开中断的
        ! 情况下处理（这也被称为bottom half）
240      set_intr_gate(0x20+irq.interrupt[irq]);
241      if (irq < 8) {
        ! 如果irq小于8
242          cache_21 &= ~(1<<irq);
243          outb(cache_21,0x21);
        ! 则允许对应的irq中断请求
244      } else {
          cache_21 &= ~(1<<2);
246          cache_A1 &= ~(1<<(irq-8));
247          outb(cache_21,0x21);
          outb(cache_A1,0xA1);
        ! 则允许对应的irq中断请求（注意该irq号中断，在从8259A芯片上）
249      }
250      restore_flags(flags);
        ! 恢复eflags寄存器，定义于include/asm/system.h
251      return 0;
        ! 返回0
252 }

```

! 设置对应与 irq 的 IRQ

```

254 int request_irq(unsigned int irq, void (*handler)(int))
255 {
256     struct sigaction sa;
257
258     sa.sa_handler = handler;
        ! 设置信号处理句柄为传入的句柄值
259     sa.sa_flags = 0;
260     sa.sa_mask = 0;
261     sa.sa_restorer = NULL;
262     return irqaction(irq,&sa);
        ! 定义于本文件中
263 }

```

```

316 static struct sigaction ignore_IRQ = {
317     no_action,
318     0,
319     SA_INTERRUPT,
320     NULL
321 };

```

! IRQ 中断初始化，也即是两片 8259A 中断控制芯片可处理中断的初始化

```

323 void init_IRQ(void)
324 {
325     int i;
326
327     for (i = 0; i < 16 ; i++)
328         set_intr_gate(0x20+i,bad_interrupt[i]);
    ! 定义于 include/asm/system.h
    ! bad_interrupt[i] 定义于本文件，保存中断处理的句柄（这里只是个占位符）
    ! 顺序设置 8259A 中断控制芯片（共 16 个）从 idt[32] 到 idt[48]
329     if (irqaction(2,&ignore_IRQ))
        ! 开启对应的中断号，ignore_IRQ 为信号处理的结构，已经在 316 行代码处进行
        ! 了填充。定义于本文件。
330     printk("Unable to get IRQ2 for cascade\n");
331     if (request_irq(13,math_error_irq))      ! 定义于 kernel/irq.c
332         printk("Unable to get IRQ13 for math-error handler\n");
333
334     /* initialize the bottom half routines. */
    ! 初始化 bottom half 中断处理程序
335     for (i = 0; i < 32; i++) {
        ! 所谓 bottom half 指的是可以在开中断情况下执行的中断处理程序
336         bh_base[i].routine = NULL;
337         bh_base[i].data = NULL;
338     }
339     bh_active = 0;
340     intr_count = 0;
        ! bh_active=intr_count=0
341 }
342

```

## Kernel/time.c

! 设置开机启动时间  
 ! 关于实时时钟的读取，请参考基础部分

```

38 void time_init(void)
39 {

```

```

40     struct mktme time;
41     int i;
42
43     /* checking for Update-In-Progress could be done more elegantly
44 * (using the "update finished"-interrupt for example), but that
45 * would require excessive testing. promise I'll do that when I find
46 * the time. - Torsten
47 */
    ! 在程序中能够更好的检查更新（使用“update finished”中断）但是那需要太多
    ! 的测试，当我发现这个中断时请相信我会处理的。
48     /* read RTC exactly on falling edge of update flag */
49     for (i = 0 ; i < 1000000 ; i++) /* may take up to 1 second... */
50         if (CMOS_READ(RTC_FREQ_SELECT) & RTC_UIP)
51             break;
    ! 快速的读RTC（实时钟）的更新标志，使得误差控制在1s之内
52     for (i = 0 ; i < 1000000 ; i++) /* must try at least 2.228 ms*/
53         if (!(CMOS_READ(RTC_FREQ_SELECT) & RTC_UIP))
54             break;
55     do { /* Isn't this overkill ? UIP above should guarantee consistency */
56         time.sec = CMOS_READ(RTC_SECONDS);
57         time.min = CMOS_READ(RTC_MINUTES);
58         time.hour = CMOS_READ(RTC_HOURS);
59         time.day = CMOS_READ(RTC_DAY_OF_MONTH);
60         time.mon = CMOS_READ(RTC_MONTH);
61         time.year = CMOS_READ(RTC_YEAR);
62     } while (time.sec != CMOS_READ(RTC_SECONDS));
    ! 控制误差在一秒之内
63     if (!(CMOS_READ(RTC_CONTROL) & RTC_DM_BINARY) ||
RTC_ALWAYS_BCD)
64     {
65         BCD_TO_BIN(time.sec);
66         BCD_TO_BIN(time.min);
67         BCD_TO_BIN(time.hour);
68         BCD_TO_BIN(time.day);
69         BCD_TO_BIN(time.mon);
70         BCD_TO_BIN(time.year);
71     }
72     time.mon--;
73     xtime.tv_sec = kernel_mktme(&time);
    ! kernel_mktme返回从1970年1月1号0时起到开机后,
    ! 经过的秒数，这里不在对其注释。
74 }

```

## Kernel/sched.c

```

1 /*
2  * linux/kernel/sched.c
3  *
4  * Copyright (C) 1991, 1992 Linus Torvalds
5  */
6
7 /*
8  * 'sched.c' is the main kernel file. It contains scheduling primitives
9  * (sleep_on, wakeup, schedule etc) as well as a number of simple system
10 * call functions (type getpid(), which just extracts a field from
11 * current-task
12 */
13
14 #include <linux/config.h>
15 #include <linux/signal.h>
16 #include <linux/sched.h>
17 #include <linux/timer.h>
18 #include <linux/kernel.h>
19 #include <linux/kernel_stat.h>
20 #include <linux/sys.h>
21 #include <linux/fdreg.h>
22 #include <linux/errno.h>
23 #include <linux/time.h>
24 #include <linux/ptrace.h>
25 #include <linux/segment.h>
26 #include <linux/delay.h>
27 #include <linux/interrupt.h>
28
29 #include <asm/system.h>
30 #include <asm/io.h>
31 #include <asm/segment.h>
32
33 #define TIMER_IRQ 0
34
35 #include <linux/timex.h>
36
37 /*
38  * kernel variables

```

```

39 */
40 long tick = 1000000 / HZ;           /* timer interrupt period */
41 volatile struct timeval xtime;      /* The current time */
42 int tickadj = 500/HZ;              /* microsecs */
43
44 /*
45 * phase-lock loop variables
46 */
47 int time_status = TIME_BAD;        /* clock synchronization status */
48 long time_offset = 0;              /* time adjustment (us) */
49 long time_constant = 0;            /* pll time constant */
50 long time_tolerance = MAXFREQ;    /* frequency tolerance (ppm) */
51 long time_precision = 1;           /* clock precision (us) */
52 long time_maxerror = 0x70000000;   /* maximum error */
53 long time_esterror = 0x70000000;   /* estimated error */
54 long time_phase = 0;               /* phase offset (scaled us) */
55 long time_freq = 0;                /* frequency offset (scaled ppm) */
56 long time_adj = 0;                 /* tick adjust (scaled 1 / HZ) */
57 long time_reftime = 0;             /* time at last adjustment (s) */
58
59 long time_adjust = 0;
60 long time_adjust_step = 0;
61
62 int need_resched = 0;
63
64 /*
65 * Tell us the machine setup..
66 */
67 int hard_math = 0;                 /* set by boot/head.S */
68 int x86 = 0;                      /* set by boot/head.S to 3 or 4 */
69 int ignore_irq13 = 0;              /* set if exception 16 works */
70 int wp_works_ok = 0;               /* set if paging hardware honours WP */
71
72 /*
73 * Bus types ..
74 */
75 int EISA_bus = 0;
76
77 extern int setitimer(int, struct itimerval *, struct itimerval *);
78 unsigned long * prof_buffer = NULL;
79 unsigned long prof_len = 0;
80
81 #define S(nr) (1<<((nr)-1))
82

```



```

127 sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
128 sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
129 sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
130 sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
131 sys_olduname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
132 sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
133 sys_setreuid, sys_setregid, sys_sigsuspend, sys_sigpending,
134 sys_sethostname, sys_setrlimit, sys_getrlimit, sys_getrusage,
135 sys_gettimeofday, sys_settimeofday, sys_getgroups, sys_setgroups,
136 sys_select, sys_symlink, sys_lstat, sys_readlink, sys_uselib,
137 sys_swapon, sys_reboot, sys_readdir, sys_mmap, sys_munmap, sys_truncate,
138 sys_ftruncate, sys_fchmod, sys_fchown, sys_getpriority, sys_setpriority,
139 sys_profil, sys_statfs, sys_fstatfs, sys_ioperm, sys_socketcall,
140 sys_syslog, sys_setitimer, sys_getitimer, sys_newstat, sys_newlstat,
141 sys_newfstat, sys_uname, sys_iopl, sys_vhangup, sys_idle, sys_vm86,
142 sys_wait4, sys_swapoff, sys_sysinfo, sys_ipc, sys_fsync, sys_sigreturn,
143 sys_clone, sys_setdomainname, sys_newuname, sys_modify_ldt,
144 sys_adjtimex, sys_mprotect, sys_sigprocmask, sys_create_module,
145 sys_init_module, sys_delete_module, sys_get_kernel_syms, sys_quotactl,
146 sys_getpgid, sys_fchdir, sys_bflush };

147

148 /* So we don't have to do any more manual updating.... */

149 int NR_syscalls = sizeof(sys_call_table)/sizeof(fn_ptr);

150

151 #ifdef __cplusplus
152 }
153 #endif
154
155 /*
156 * 'math_state_restore()' saves the current math information in the
157 * old math state array, and gets the new ones from the current task
158 *
159 * Careful.. There are problems with IBM-designed IRQ13 behaviour.
160 * Don't touch unless you *really* know how it works.
161 */
162 asmlinkage void math_state_restore(void)
163 {
164     __asm__ __volatile__ ("clts");
165     if (last_task_used_math == current)
166         return;
167     timer_table[COPRO_TIMER].expires = jiffies+50;
168     timer_active |= 1<<COPRO_TIMER;
169     if (last_task_used_math)
170         __asm__ ("fnsave %0":"=m" (last_task_used_math->tss.i387));

```

```

171     else
172         __asm__("fncllex");
173     last_task_used_math = current;
174     if (current->used_math) {
175         __asm__("frstor %0": :"m" (current->tss.i387));
176     } else {
177         __asm__("fninit");
178         current->used_math=1;
179     }
180     timer_active &= ~(1<<COPRO_TIMER);
181 }
182
183 #ifndef CONFIG_MATH_EMULATION
184
185 asmlinkage void math_emulate(long arg)
186 {
187     printk("math-emulation not enabled and no coprocessor found.\n");
188     printk("killing %s.\n",current->comm);
189     send_sig(SIGFPE,current,1);
190     schedule();
191 }
192
193 #endif /* CONFIG_MATH_EMULATION */
194
195 unsigned long itimer_ticks = 0;
196 unsigned long itimer_next = ~0;
197 static unsigned long lost_ticks = 0;
198
199 /*
200  * 'schedule()' is the scheduler function. It's a very simple and nice
201  * scheduler: it's not perfect, but certainly works for most things.
202  * The one thing you might take a look at is the signal-handler code here.
203  *
204  * NOTE!! Task 0 is the 'idle' task, which gets called when no other
205  * tasks can run. It can not be killed, and it cannot sleep. The 'state'
206  * information in task[0] is never used.
207  *
208  * The "confuse_gcc" goto is used only to get better assembly code..
209  * Djikstra probably hates me.
210 */
211 asmlinkage void schedule(void)
212 {
213     int c;
214     struct task_struct * p;

```

```

215     struct task_struct * next;
216     unsigned long ticks;
217
218 /* check alarm, wake up any interruptible tasks that have got a signal */
219
220     cli();
221     ticks = itimer_ticks;
222     itimer_ticks = 0;
223     itimer_next = ~0;
224     sti();
225     need_resched = 0;
226     p = &init_task;
227     for (;;) {
228         if ((p = p->next_task) == &init_task)
229             goto confuse_gcc1;
230         if (ticks && p->it_real_value) {
231             if (p->it_real_value <= ticks) {
232                 send_sig(SIGALRM, p, 1);
233                 if (!p->it_real_incr) {
234                     p->it_real_value = 0;
235                     goto end_itimer;
236                 }
237                 do {
238                     p->it_real_value += p->it_real_incr;
239                 } while (p->it_real_value <= ticks);
240             }
241             p->it_real_value -= ticks;
242             if (p->it_real_value < itimer_next)
243                 itimer_next = p->it_real_value;
244         }
245     end_itimer:
246         if (p->state != TASK_INTERRUPTIBLE)
247             continue;
248         if (p->signal & ~p->blocked) {
249             p->state = TASK_RUNNING;
250             continue;
251         }
252         if (p->timeout && p->timeout <= jiffies) {
253             p->timeout = 0;
254             p->state = TASK_RUNNING;
255         }
256     }
257     confuse_gcc1:
258

```

```

259 /* this is the scheduler proper: */
260 #if 0
261     /* give processes that go to sleep a bit higher priority.. */
262     /* This depends on the values for TASK_XXX */
263     /* This gives smoother scheduling for some things, but */
264     /* can be very unfair under some circumstances, so.. */
265     if (TASK_UNINTERRUPTIBLE) >= (unsigned) current->state &&
266         current->counter < current->priority*2) {
267         ++current->counter;
268     }
269 #endif
270     c = -1;
271     next = p = &init_task;
272     for (;;) {
273         if ((p = p->next_task) == &init_task)
274             goto confuse_gcc2;
275         if (p->state == TASK_RUNNING && p->counter > c)
276             c = p->counter, next = p;
277     }
278 confuse_gcc2:
279     if (!c) {
280         for_each_task(p)
281             p->counter = (p->counter >> 1) + p->priority;
282     }
283     if(current != next)
284         kstat.context_swtch++;
285     switch_to(next);
286     /* Now maybe reload the debug registers */
287     if(current->debugreg[7]){
288         loaddebug(0);
289         loaddebug(1);
290         loaddebug(2);
291         loaddebug(3);
292         loaddebug(6);
293     };
294 }
295
296 asmlinkage int sys_pause(void)
297 {
298     current->state = TASK_INTERRUPTIBLE;
299     schedule();
300     return -ERESTARTNOHAND;
301 }
302

```

```

303 /*
304 * wake_up doesn't wake up stopped processes - they have to be awakened
305 * with signals or similar.
306 *
307 * Note that this doesn't need cli-sti pairs: interrupts may not change
308 * the wait-queue structures directly, but only call wake_up() to wake
309 * a process. The process itself must remove the queue once it has woken.
310 */
!
```

! wake\_up不是用于唤醒已经停止的进程，它用于唤醒被信号或和其类似的条件  
! 切换出 CPU 的进程。  
! 注意它不需要关中断及开中断，因为中断在等待队列中中断  
! 可能已经发生了变化，  
! 但是仅仅通过调用 wake\_up () 唤醒一个进程，  
! 这个被唤醒的进程必须自己把它从等待队列中移出来。

```

311 void wake_up(struct wait_queue **q)
312 {
313     struct wait_queue *tmp;
314     struct task_struct * p;
315
316     if (!q || !(tmp = *q))          ! 等待队列及等待队列项指针为空时退出
317         return;
318     do {
319         if ((p = tmp->task) != NULL) {    ! 从等待队列取得任务指针
320             if ((p->state == TASK_UNINTERRUPTIBLE) ||
321                 (p->state == TASK_INTERRUPTIBLE)) {
322                 ! 当取得的任务状态为TASK_UNINTERRUPTIBLE或者TASK_INTERRUPTIBLE
323                 ! 时，置其为可运行状态
324                 p->state = TASK_RUNNING;
325                 if (p->counter > current->counter)
326                     need_resched = 1;
327                 ! 当取得的任务的可运行时间大于当前任务的可运行时间时，设置need_resched=1
328             }
329             if (!tmp->next) {           ! 如果下一请求队列指针空，则出错
330                 printk("wait_queue is bad (eip = %08lx)\n",((unsigned long *) q)[-1]);
331                 printk("      q = %p\n",q);
332                 printk("      *q = %p\n",*q);
333                 printk("      tmp = %p\n",tmp);
334                 break;
335             }
336             tmp = tmp->next;
337         } while (tmp != *q);          ! 当请求队列中所有项都访问完后，循环结束
338     }
339 }
```

```

    ! 唤醒可中断任务

338 void wake_up_interruptible(struct wait_queue **q)
339 {
340     struct wait_queue *tmp;
341     struct task_struct * p;
342
343     if (!q || !(tmp = *q))          ! 如果q=null或者*q=null则退出（即
                                    ! 等待队列及等待队列项指针为空时退出）
344         return;
345     do {
346         if ((p = tmp->task) != NULL) {      ! 从等待队列取得任务指针
                                            ! 如果等待队列不为空
347             if (p->state == TASK_INTERRUPTIBLE) {

348                 p->state = TASK_RUNNING;        ! 则设置状态为可调度状态
349                 if (p->counter > current->counter)   ! 如果可运行时间,
                                                ! 大于当前的可运行时间
350                     need_resched = 1;           ! 设置need_resched =1
351             }
352         }
353         if (!tmp->next) {            ! 如果下一请求队列指针空, 则出错
354             printk("wait_queue is bad (eip = %08lx)\n",((unsigned long *) q)[-1]);
355             printk("      q = %p\n",q);
356             printk("      *q = %p\n",*q);
357             printk("      tmp = %p\n",tmp);
358             break;
359         }
360         tmp = tmp->next;
361     } while (tmp != *q);          ! 循环完等待队列列表
362 }
363

364 void down(struct semaphore * sem)
365 {
366     struct wait_queue wait = { current, NULL };
367     add_wait_queue(&sem->wait, &wait);
368     current->state = TASK_UNINTERRUPTIBLE;
369     while (sem->count <= 0) {
370         schedule();
371         current->state = TASK_UNINTERRUPTIBLE;
372     }
373     current->state = TASK_RUNNING;
374     remove_wait_queue(&sem->wait, &wait);
375 }
```

```

376
377 static inline void __sleep_on(struct wait_queue **p, int state)
378 {
379     unsigned long flags;
380     struct wait_queue wait = { current, NULL };
381
382     if (!p)
383         return;
384     if (current == task[0])
385         panic("task[0] trying to sleep");
386     current->state = state;
387     add_wait_queue(p, &wait);
388     save_flags(flags);
389     sti();
390     schedule();
391     remove_wait_queue(p, &wait);
392     restore_flags(flags);
393 }
394
395 void interruptible_sleep_on(struct wait_queue **p)
396 {
397     __sleep_on(p, TASK_INTERRUPTIBLE);
398 }
399
400 void sleep_on(struct wait_queue **p)
401 {
402     __sleep_on(p, TASK_UNINTERRUPTIBLE);
403 }
404
405 static struct timer_list * next_timer = NULL;
406
407 void add_timer(struct timer_list * timer)
408 {
409     unsigned long flags;
410     struct timer_list ** p;
411
412     if (!timer)
413         return;
414     timer->next = NULL;
415     p = &next_timer;
416     save_flags(flags);
417     cli();
418     while (*p) {
419         if ((*p)->expires > timer->expires) {

```

```

420             (*p)->expires -= timer->expires;
421             timer->next = *p;
422             break;
423         }
424         timer->expires -= (*p)->expires;
425         p = &(*p)->next;
426     }
427     *p = timer;
428     restore_flags(flags);
429 }
430
431 int del_timer(struct timer_list * timer)
432 {
433     unsigned long flags;
434     unsigned long expires = 0;
435     struct timer_list **p;
436
437     p = &next_timer;
438     save_flags(flags);
439     cli();
440     while (*p) {
441         if (*p == timer) {
442             if ((*p = timer->next) != NULL)
443                 (*p)->expires += timer->expires;
444             timer->expires += expires;
445             restore_flags(flags);
446             return 1;
447         }
448         expires += (*p)->expires;
449         p = &(*p)->next;
450     }
451     restore_flags(flags);
452     return 0;
453 }
454
455 unsigned long timer_active = 0;
456 struct timer_struct timer_table[32];
457
458 /*
459  * Hmm.. Changed this, as the GNU make sources (load.c) seems to
460  * imply that avenrun[] is the standard name for this kind of thing.
461  * Nothing else seems to be standardized: the fractional size etc
462  * all seem to differ on different machines.
463 */

```

```

464 unsigned long avenrun[3] = { 0,0,0 };
465
466 /*
467 * Nr of active tasks - counted in fixed-point numbers
468 */
469 static unsigned long count_active_tasks(void)
470 {
471     struct task_struct **p;
472     unsigned long nr = 0;
473
474     for(p = &LAST_TASK; p > &FIRST_TASK; --p)
475         if (*p && ((*p)->state == TASK_RUNNING ||
476                     (*p)->state == TASK_UNINTERRUPTIBLE ||
477                     (*p)->state == TASK_SWAPPING))
478             nr += FIXED_1;
479
480     return nr;
481 }
482 static inline void calc_load(void)
483 {
484     unsigned long active_tasks; /*fixed-point */
485     static int count = LOAD_FREQ;
486
487     if (count-- > 0)
488         return;
489     count = LOAD_FREQ;
490     active_tasks = count_active_tasks();
491     CALC_LOAD(avenrun[0], EXP_1, active_tasks);
492     CALC_LOAD(avenrun[1], EXP_5, active_tasks);
493     CALC_LOAD(avenrun[2], EXP_15, active_tasks);
494 }
495
496 /*
497 * this routine handles the overflow of the microsecond field
498 *
499 * The tricky bits of code to handle the accurate clock support
500 * were provided by Dave Mills (Mills@UDEL.EDU) of NTP fame.
501 * They were originally developed for SUN and DEC kernels.
502 * All the kudos should go to Dave for this stuff.
503 *
504 * These were ported to Linux by Philip Gladstone.
505 */
506 static void second_overflow(void)
507 {

```

```

508     long ltemp;
509     /* last time the cmos clock got updated */
510     static long last_rtc_update=0;
511     extern int set_rtc_mmss(unsigned long);
512
513     /* Bump the maxerror field */
514     time_maxerror = (0x70000000-time_maxerror < time_tolerance) ?
515         0x70000000 : (time_maxerror + time_tolerance);
516
517     /* Run the PLL */
518     if (time_offset < 0) {
519         ltemp = -(time_offset+1) >> (SHIFT_KG + time_constant) + 1;
520         time_adj = ltemp << (SHIFT_SCALE - SHIFT_HZ - SHIFT_UPDATE);
521         time_offset += (time_adj * HZ) >> (SHIFT_SCALE - SHIFT_UPDATE);
522         time_adj = - time_adj;
523     } else if (time_offset > 0) {
524         ltemp = ((time_offset-1) >> (SHIFT_KG + time_constant)) + 1;
525         time_adj = ltemp << (SHIFT_SCALE - SHIFT_HZ - SHIFT_UPDATE);
526         time_offset -= (time_adj * HZ) >> (SHIFT_SCALE - SHIFT_UPDATE);
527     } else {
528         time_adj = 0;
529     }
530
531     time_adj += (time_freq >> (SHIFT_KF + SHIFT_HZ - SHIFT_SCALE))
532     + FINETUNE;
533
534     /* Handle the leap second stuff */
535     switch (time_status) {
536         case TIME_INS:
537             /* ugly divide should be replaced */
538             if (xtime.tv_sec % 86400 == 0) {
539                 xtime.tv_sec--; /* !! */
540                 time_status = TIME_OOP;
541                 printk("Clock: inserting leap second 23:59:60 GMT\n");
542             }
543             break;
544
545         case TIME_DEL:
546             /* ugly divide should be replaced */
547             if (xtime.tv_sec % 86400 == 86399) {
548                 xtime.tv_sec++;
549                 time_status = TIME_OK;
550                 printk("Clock: deleting leap second 23:59:59 GMT\n");
551             }

```

```

552         break;
553
554     case TIME_OOP:
555         time_status = TIME_OK;
556         break;
557     }
558     if (xtime.tv_sec > last_rtc_update + 660)
559         if (set_rtc_mmss(xtime.tv_sec) == 0)
560             last_rtc_update = xtime.tv_sec;
561 }
562
563 /*
564 * disregard lost ticks for now.. We don't care enough.
565 */
566 static void timer_bh(void * unused)
567 {
568     unsigned long mask;
569     struct timer_struct *tp;
570
571     cli();
572     while (next_timer && next_timer->expires == 0) {
573         void (*fn)(unsigned long) = next_timer->function;
574         unsigned long data = next_timer->data;
575         next_timer = next_timer->next;
576         sti();
577         fn(data);
578         cli();
579     }
580     sti();
581
582     for (mask = 1, tp = timer_table+0 ; mask ; tp++,mask += mask) {
583         if (mask > timer_active)
584             break;
585         if (!(mask & timer_active))
586             continue;
587         if (tp->expires > jiffies)
588             continue;
589         timer_active &= ~mask;
590         tp->fn();
591         sti();
592     }
593 }
594
595 /*

```

```

596 * The int argument is really a (struct pt_regs *), in case the
597 * interrupt wants to know from where it was called. The timer
598 * irq uses this to decide if it should update the user or system
599 * times.
600 */
601 static void do_timer(struct pt_regs * regs)
602 {
603     unsigned long mask;
604     struct timer_struct *tp;
605
606     long ltemp;
607
608     /* Advance the phase, once it gets to one microsecond, then
609      * advance the tick more.
610      */
611     time_phase += time_adj;
612     if (time_phase < -FINEUSEC) {
613         ltemp = -time_phase >> SHIFT_SCALE;
614         time_phase += ltemp << SHIFT_SCALE;
615         xtime.tv_usec += tick + time_adjust_step - ltemp;
616     }
617     else if (time_phase > FINEUSEC) {
618         ltemp = time_phase >> SHIFT_SCALE;
619         time_phase -= ltemp << SHIFT_SCALE;
620         xtime.tv_usec += tick + time_adjust_step + ltemp;
621     } else
622         xtime.tv_usec += tick + time_adjust_step;
623
624     if (time_adjust)
625     {
626         /* We are doing an adjtime thing.
627         *
628         * Modify the value of the tick for next time.
629         * Note that a positive delta means we want the clock
630         * to run fast. This means that the tick should be bigger
631         *
632         * Limit the amount of the step for *next* tick to be
633         * in the range -tickadj .. +tickadj
634         */
635     if (time_adjust > tickadj)
636         time_adjust_step = tickadj;
637     else if (time_adjust < -tickadj)
638         time_adjust_step = -tickadj;
639     else

```

```

640         time_adjust_step = time_adjust;
641
642         /* Reduce by this step the amount of time left */
643         time_adjust -= time_adjust_step;
644     }
645     else
646         time_adjust_step = 0;
647
648     if (xtime.tv_usec >= 1000000) {
649         xtime.tv_usec -= 1000000;
650         xtime.tv_sec++;
651         second_overflow();
652     }
653
654     jiffies++;
655     calc_load();
656     if ((VM_MASK & regs->eflags) || (3 & regs->cs)) {
657         current->utime++;
658         if (current != task[0]) {
659             if (current->priority < 15)
660                 kstat.cpu_nice++;
661             else
662                 kstat.cpu_user++;
663         }
664         /* Update ITIMER_VIRT for current task if not in a system call */
665         if (current->it_virt_value && !(-current->it_virt_value)) {
666             current->it_virt_value = current->it_virt_incr;
667             send_sig(SIGVTALRM,current,1);
668         }
669     } else {
670         current->stime++;
671         if(current != task[0])
672             kstat.cpu_system++;
673 #ifdef CONFIG_PROFILE
674         if (prof_buffer && current != task[0]) {
675             unsigned long eip = regs->eip;
676             eip >>= 2;
677             if (eip < prof_len)
678                 prof_buffer[eip]++;
679         }
680 #endif
681     }
682     if (current == task[0] || (--current->counter)<=0) {
683         current->counter=0;

```

```

684         need_resched = 1;
685     }
686     /* Update ITIMER_PROF for the current task */
687     if (current->it_prof_value && !(--current->it_prof_value)) {
688         current->it_prof_value = current->it_prof_incr;
689         send_sig(SIGPROF,current,1);
690     }
691     for (mask = 1, tp = timer_table+0 ; mask ; tp++,mask += mask) {
692         if (mask > timer_active)
693             break;
694         if (!(mask & timer_active))
695             continue;
696         if (tp->expires > jiffies)
697             continue;
698         mark_bh(TIMER_BH);
699     }
700     cli();
701     itimer_ticks++;
702     if (itimer_ticks > itimer_next)
703         need_resched = 1;
704     if (next_timer) {
705         if (next_timer->expires) {
706             next_timer->expires--;
707             if (!next_timer->expires)
708                 mark_bh(TIMER_BH);
709         } else {
710             lost_ticksmark_bh(TIMER_BH);
712         }
713     }
714     sti();
715 }
716
717 asmlinkage int sys_alarm(long seconds)
718 {
719     struct itimerval it_new, it_old;
720
721     it_new.it_interval.tv_sec = it_new.it_interval.tv_usec = 0;
722     it_new.it_value.tv_sec = seconds;
723     it_new.it_value.tv_usec = 0;
724     setitimer(ITIMER_REAL, &it_new, &it_old);
725     return(it_old.it_value.tv_sec + (it_old.it_value.tv_usec / 1000000));
726 }
727

```

```
728 asmlinkage int sys_getpid(void)
729 {
730     return current->pid;
731 }
732
733 asmlinkage int sys_getppid(void)
734 {
735     return current->p_opptr->pid;
736 }
737
738 asmlinkage int sys_getuid(void)
739 {
740     return current->uid;
741 }
742
743 asmlinkage int sys_geteuid(void)
744 {
745     return current->euid;
746 }
747
748 asmlinkage int sys_getgid(void)
749 {
750     return current->gid;
751 }
752
753 asmlinkage int sys_getegid(void)
754 {
755     return current->egid;
756 }
757
758 asmlinkage int sys_nice(long increment)
759 {
760     int newprio;
761
762     if (increment < 0 && !suser())
763         return -EPERM;
764     newprio = current->priority - increment;
765     if (newprio < 1)
766         newprio = 1;
767     if (newprio > 35)
768         newprio = 35;
769     current->priority = newprio;
770     return 0;
771 }
```

```

772
773 static void show_task(int nr, struct task_struct * p)
774 {
775     static char * stat_nam[] = { "R", "S", "D", "Z", "T", "W" };
776
777     printf("%-8s %3d ", p->comm, (p == current) ? -nr : nr);
778     if (((unsigned) p->state) < sizeof(stat_nam)/sizeof(char *))
779         printf(stat_nam[p->state]);
780     else
781         printf(" ");
782     if (p == current)
783         printf(" current ");
784     else
785         printf(" %08lX ", ((unsigned long *)p->tss.esp)[3]);
786     printf("%5lu %5d %6d ",
787             p->tss.esp - p->kernel_stack_page, p->pid, p->p_pptr->pid);
788     if (p->p_cptr)
789         printf("%5d ", p->p_cptr->pid);
790     else
791         printf("      ");
792     if (p->p_ysptr)
793         printf("%7d", p->p_ysptr->pid);
794     else
795         printf("      ");
796     if (p->p_osptr)
797         printf(" %5d\n", p->p_osptr->pid);
798     else
799         printf("\\n");
800 }
801
802 void show_state(void)
803 {
804     int i;
805
806     printf("%-15s %-15s %-15s\n", "free", "stack", "sibling\\n");
807     printf("%-15s %-15s %-15s %-15s %-15s %-15s %-15s %-15s\n", "task", "PC", "stack", "pid", "father", "child", "younger", "older\\n");
808     for (i=0 ; i<NR_TASKS ; i++)
809         if (task[i])
810             show_task(i, task[i]);
811 }
812
!
```

任务 0 的初始化以及对系统中任务数组初始化

```

813 void sched_init(void)
814 {

```

```

815     int i;
816     struct desc_struct * p;
817
818     bh_base[TIMER_BH].routine = timer_bh;
        ! 数值对应的 bottom half 处理列程为 timer_bh。
        ! 为时钟中断的bottom half处理程序
819     if (sizeof(struct sigaction) != 16)
820         panic("Struct sigaction MUST be 16 bytes");
        ! 如果struct sigaction大小不是 16，则死机，定于Kernel/panic.c
821     set_tss_desc(gdt+FIRST_TSS_ENTRY,&init_task.tss);
        ! 设置任务 0 的任务状态段
822     set_ldt_desc(gdt+FIRST_LDT_ENTRY,&default_ldt,1);
        ! 设置任务 0 的局部描述符地址为 default_ldt 地址！
        ! 定义于include/asm/system.h
823     set_system_gate(0x80,&system_call);
        ! 设置系统调用门
824     p = gdt+2+FIRST_TSS_ENTRY;
825     for(i=1 ; i<NR_TASKS ; i++) {
826         task[i] = NULL;
827         p->a=p->b=0;
828         p++;
829         p->a=p->b=0;
830         p++;
831     }
        ! 清任务数组和全局描述符项
832 /* Clear NT, so that we won't have troubles with that later on */
        ! 清除标志寄存器中的NT位，这样以后就不会有问题了
833     __asm__ ("pushfl ; andl $0xffffbfff,%esp ; popfl");
        ! 复位NT位
834     load_TR(0);
        ! 将任务 0 的TSS加载到任务寄存器TR中
835     load_ldt(0);
        ! 将局部描述符表加载到局部描述符寄存器
836     outb_p(0x34,0x43);      /* binary, mode 2, LSB/MSB, ch 0 */
837     outb_p(LATCH & 0xff, 0x40); /* LSB */! 定时值低字节
838     outb(LATCH >> 8, 0x40); /* MSB */! 定时值高字节
839     if (request_irq(TIMER_IRQ,(void (*)(int)) do_timer)!=0)
        ! 定义于 kerel/irq.c
        ! 设置时钟中断处理程序！
840     panic("Could not allocate timer IRQ!");
        ! 定义于Kernel/panic.c
841 }
842

```

## Kernel/exit.c(部分代码)

```

! 对信号的处理
× sig=信号值
× p=任务指针

25 static int generate(unsigned long sig, struct task_struct * p)
26 {
27     unsigned long mask = 1 << (sig-1);
28     struct sigaction * sa = sig + p->sigaction - 1;
    ! 根据sig取得对应的信号处理结构

29
30     /* always generate signals for traced processes ??? */
31     if (p->flags & PF_PTRACED) {
32         p->signal |= mask;
33         return 1;
34     }
35     /* don't bother with ignored signals (but SIGCHLD is special) */
36     if (sa->sa_handler == SIG_IGN && sig != SIGCHLD)
37         return 0;
    ! 如果信号处理句柄为SIG_IGN或者SIGCHLD则直接退出
38     /* some signals are ignored by default.. (but SIGCONT already did its deed) */
39     if ((sa->sa_handler == SIG_DFL) &&
40         (sig == SIGCONT || sig == SIGCHLD || sig == SIGCHLD))
41         return 0;
    ! 如果信号处理句柄为SIG_DEF并且sig=SIGCONT或者SIGCHLD
    或者SIGCHLD则直接退出
42     p->signal |= mask;
    ! 屏蔽对应的信号位图
43     return 1;
44 }

```

! 给任务发送信号
 × sig=信号值
 × p=任务指针
 × priv=优先级

46 int send\_sig(unsigned long sig,struct task\_struct \* p,int priv)

47 {

```

48     if (!p || sig > 32)
49         return -EINVAL;
    ! 任务不存在或信号大于 32, 返回-EINVAL
50     if (!priv && ((sig != SIGCONT) || (current->session != p->session)) &&
51         (current->euid != p->euid) && (current->uid != p->uid) && !suser())
52         return -EPERM;
    ! 判断权限的值, 如果不符号则返回-EPERM, 这里条件很多这里不做注释
53     if (!sig)
54         return 0;
    ! 如果信号为 0, 则直接返回
55     if ((sig == SIGKILL) || (sig == SIGCONT)) {
    ! 如果信号是SIGKILL或SIGCONT
56         if (p->state == TASK_STOPPED)
57             p->state = TASK_RUNNING;
    ! 如果当前进程的状态是TASK_STOPPED的话, 则设置其为TASK_RUNNING
58         p->exit_code = 0;
    ! 退出代码为
59         p->signal &= ~( (1<<(SIGSTOP-1)) | (1<<(SIGTSTP-1)) |
60                         (1<<(SIGTTIN-1)) | (1<<(SIGTTOU-1)) );
    ! 屏蔽一些位 (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU)
61     }
62     /* Depends on order SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU */
63     if ((sig >= SIGSTOP) && (sig <= SIGTTOU))
64         p->signal &= ~(1<<(SIGCONT-1));
65     /* Actually generate the signal */
    ! 从 50 行到这, 为根据信号值设置进程的状态及对应的信号位图
66     generate(sig,p);
    ! 定义于本文件
67     return 0;
68 }
69

```

```

214 /*
215 * kill_pg() sends a signal to a process group: this is what the tty
216 * control characters do (^C, ^Z etc)
217 */
    ! kill_pg发送信号给一个进程组
218 int kill_pg(int pgrp, int sig, int priv)
219 {
220     struct task_struct *p;
221     int err,retval = -ESRCH;
222     int found = 0;

```

```

223
224     if (sig<0 || sig>32 || pgrp<=0)
225         return -EINVAL;
    ! 如果信号小于 0, 大于 32, 或者进程组id小于 0, 则返回-EINVAL
226     for_each_task(p) {
227         if (p->pgrp == pgrp) {
228             if ((err = send_sig(sig,p,priv)) != 0)
229                 retval = err;
230             else
231                 found++;
    ! 循环扫描进程组, 当发现相等的进程组时, 则发送信号
    ! send_sig定义于本文件
232     }
233     }
234     return(found ? 0 : retval);
235 }

338 static void forget_original_parent(struct task_struct * father)
339 {
340     struct task_struct * p;
341
342     for_each_task(p) {
343         if (p->p_opptr == father)
344             if (task[1])
345                 p->p_opptr = task[1];
346             else
347                 p->p_opptr = task[0];
    ! 如果, 存在任务 1, 则修改father的子进程的父进程为任务 1, 反之为任务 0
348     }
349 }

```

! 程序退出处理程序, code 是错误码

```

351 NORET_TYPE void do_exit(long code)
352 {
353     struct task_struct *p;
354     int i;
355
356 fake_volatile:
357     if (current->semun)
        ! 如果当前进程有需要撤消的信号量, 则转到
358         sem_exit();
    ! sem_exit()结束它, 定义于Ipc/sem.c

```

```

359     if (current->shm)
        ! 如果当前进程有共享内存，则释放之
360         shm_exit();
        ! 定义于Ipc/shm.c
361         free_page_tables(current);
        ! 释放当前进程的页表
362         for (i=0 ; i<NR_OPEN ; i+)
363             if (current->filp[i])
364                 sys_close(i);
        ! 关闭当前进程打开的所有文件，定义于Fs/open.c
365         forget_original_parent(current);
        ! 定义于本文件，修改当前进程的子进程的父进程
366         iput(current->pwd);
        ! 释放当前进程工作目录所占用的I节点，定义于Fs/inode.c
367         current->pwd = NULL;
        ! 置空当前进程工作目录
368         iput(current->root);
        ! 释放当前进程根目录所占用的I节点，定义于Fs/inode.c
369         current->root = NULL;
        ! 置空当前进程根目录
370         iput(current->executable);
        ! 释放当前进程可执行I节点所占用的I节点，定义于Fs/inode.c
371         current->executable = NULL;
        ! 置空当前进程可执行I节点
372     /* Release all of the old mmap stuff. */
373
        ! 释放所有的老的mmap缓冲
374     {
375         struct vm_area_struct * mpnt, *mpnt1;
376         mpnt = current->mmap;
377         current->mmap = NULL;
378         while (mpnt) {
379             mpnt1 = mpnt->vm_next;
380             if (mpnt->vm_ops && mpnt->vm_ops->close)
381                 mpnt->vm_ops->close(mpnt);
382             kfree(mpnt);
383             mpnt = mpnt1;
384         }
385     }
386
387     if (current->ldt) {
        ! 如果当前进程的局部描述符存在
388         vfree(current->ldt);
        ! 释放当前进程的局部描述符，定义于Mm/vmalloc.c
389         current->ldt = NULL;

```

```

! 置空当前进程的局部描述符
390     for (i=1 ; i<NR_TASKS ; i++) {
    ! 从第一个任务开始扫描任务数组
391     if (task[i] == current) {
392         set_ldt_desc(gdt+(i<<1)+FIRST_LDT_ENTRY, &default_ldt, 1);
    ! 如果找到了，则在全局描述符表设置局部描述符表
    ! 定义于Include/asm/system.h
393         load_ldt(i);
    ! 加载第i个任务局部描述符表到ldtr中
394     }
395     }
396   }
397
398   current->state = TASK_ZOMBIE;
    ! 置当前进程为僵死
399   current->exit_code = code;
    ! 置错误码
400   current->rss = 0;
    ! 置rss为0
401   /*
402   * Check to see if any process groups have become orphaned
403   * as a result of our exiting, and if they have any stopped
404   * jobs, send them a SIGUP and then a SIGCONT. (POSIX 3.2.2.2)
405   *
406   * Case i: Our father is in a different pgrp than we are
407   * and we were the only connection outside, so our pgrp
408   * is about to become orphaned.
409   */
    ! 检查是否有因为我们的离开（停止进程）而导致有任何的孤儿进程组产生,
    ! 假如它们有任何已停止任务，给它们发送 SIGUP 及 SIGCONT 信号（POSIX3.2.2.2）
    ! 例如：我们的父进程有一个不同进程组，从而我们只能从外部连接它们,
    ! 因此我们的进程组将会变成孤儿
410   if ((current->p_pptr->pgrp != current->pgrp) &&
411       (current->p_pptr->session == current->session) &&
412       is_orphaned_pgrp(current->pgrp) &&
413       has_stopped_jobs(current->pgrp)) {
414       kill_pg(current->pgrp,SIGHUP,1);
415       kill_pg(current->pgrp,SIGCONT,1);
    ! 定义于本文件
416   }
417   /* Let father know we died */
    ! 通知父进程，子进程将退出
418   notify_parent(current);
419

```

```

420      /*
421       * This loop does two things:
422       *
423       * A. Make init inherit all the child processes
424       * B. Check to see if any process groups have become orphaned
425       *      as a result of our exiting, and if they have any stopped
426       *      jobs, send them a SIGHUP and then a SIGCONT. (POSIX 3.2.2.2)
427      */

```

！下面的循环做两件事情：

！A：使得 init 进程接管所有的子进程（失去父亲的进程）

！B：检查是否有因为我们的离开（停止进程）而导致有任何的孤儿进程组产生，

！假如它们有任何已停止任务，给它们发送 SIGUP 及 SIGCONT 信号（POSIX3.2.2.2）

```

428     while ((p = current->p_cptr) != NULL) {
429         current->p_cptr = p->p_osptr;
430         p->p_ysptr = NULL;
431         p->flags &= ~(PF_PTRACED|PF_TRACESYS);
432         if (task[1] && task[1] != current)
433             p->p_pptr = task[1];
434         else
435             p->p_pptr = task[0];
436         p->p_osptr = p->p_pptr->p_cptr;
437         p->p_osptr->p_ysptr = p;
438         p->p_pptr->p_cptr = p;
439         if (p->state == TASK_ZOMBIE)
440             notify_parent(p);
441         /*
442          * process group orphan check
443          * Case ii: Our child is in a different pgrp
444          * than we are, and it was the only connection
445          * outside, so the child pgrp is now orphaned.
446         */

```

！进程组孤儿检查

！例如：我们的父进程有一个不同进程组，从而我们只能从外部连接它们，

！因此我们的进程组将会变成孤儿

```

447         if ((p->pgrp != current->pgrp) &&
448             (p->session == current->session) &&
449             is_orphaned_pgrp(p->pgrp) &&
450             has_stopped_jobs(p->pgrp)) {
451             kill_pg(p->pgrp,SIGHUP,1);
452             kill_pg(p->pgrp,SIGCONT,1);
453         }
454     }
455     if (current->leader)
456         disassociate_ctty(1);

```

```

    ! 如果当前进程是领头进程，则释放之
457     if (last_task_used_math == current)
458         last_task_used_math = NULL;
    ! 如果当前进程上次使用过协处理器，则置空之
459 #ifdef DEBUG_PROC_TREE
460     audit_ptree();
461 #endif
462     schedule();
    ! 重新调度
463 /*
464 * In order to get rid of the "volatile function does return" message
465 * I did this little loop that confuses gcc to think do_exit really
466 * is volatile. In fact it's schedule() that is volatile in some
467 * circumstances: when current->state = ZOMBIE, schedule() never
468 * returns.
469 *
470 * In fact the natural way to do all this is to have the label and the
471 * goto right after each other, but I put the fake_volatile label at
472 * the start of the function just in case something /really/ bad
473 * happens, and the schedule returns. This way we can try again. I'm
474 * not paranoid: it's just that everybody is out to get me.
475 */
    ! 为了摆脱不稳定的函数返回消息，我做了个小循环使得gcc认为do_exit是可变的
    ! 实际上 schedule 在一些环境下是可变的：当 current->state=ZOMBIE, schedule
    ! 从来不返回。
    ! 实际上自然的方法是用 label，但是我放了个 fake_volatile 在函数的开始，将会导致
    ! 真正的坏的事情发生，并且当 schedule 返回时，它可以在重试。我不是个狂妄的人，
    ! 它使得任何人不在找我
476     goto fake_volatile;
477 }
    ! 退出进程
    × error_code=退出时的错误码
479 asmlinkage int sys_exit(int error_code)
480 {
481     do_exit((error_code&0xff)<<8);
    ! 定义于本文件
482 }

```

## Kernel/signal.c

```

303 /*
304 * Note that 'init' is a special process: it doesn't get signals it doesn't
305 * want to handle. Thus you cannot kill init even with a SIGKILL even by
306 * mistake.
307 *
308 * Note that we go through the signals twice: once to check the signals that
309 * the kernel can handle, and then we build all the user-level signal handling
310 * stack-frames in one go after that.
311 */
    ! 注意init是一个特殊的进程：它不能得到信号也不想处理。因此你不能通过发送
    ! SIGKILL 来结束它。
    ! 注意我们检测两次信号：首先检查该信号是否能被核心处理，然后我们用户态的
    ! 信号处理函数的栈

312 asmlinkage int do_signal(unsigned long oldmask, struct pt_regs * regs)
313 {
314     unsigned long mask = ~current->blocked;
315     unsigned long handler_signal = 0;
316     unsigned long *frame = NULL;
317     unsigned long eip = 0;
318     unsigned long signr;
319     struct sigaction * sa;
320
321     while ((signr = current->signal & mask)) {
        ! 获取没有被屏蔽的信号位图
322         __asm__("bsf %2,%1\n\t"
            ! 从位 0 扫描signr是 1 的第一位，找到后放在signr中
323             "btrl %1,%0"
        ! 复位对应的信号位图
324             :"=m" (current->signal), "=r" (signr)
325             :"1" (signr));
326         sa = current->sigaction + signr;
        ! 获取对应的信号处理结构（该结构中含有信号处理句柄）
327         signr++;
        ! 信号值自增一，应为 322 行的汇编代码是从第 0 位开始扫描的
328         if ((current->flags & PF_PTRACED) && signr != SIGKILL) {
            ! 如果当前进程调用了ptrace并且信号不为SIGKILL，则做下面代码
329             current->exit_code = signr;
330             current->state = TASK_STOPPED;
331             notify_parent(current);
            ! 置退出代码，任务作态以及通知父进程
332             schedule();
            ! 重新调度定义于Kernel/sched.c
333             if (!(signr = current->exit_code))
334                 continue;

```

```

    ! 直到当前进程被置上退出码
335         current->exit_code = 0;
    ! 置退出码为 0
336         if (signr == SIGSTOP)
337             continue;
338         if (S(signr) & current->blocked) {
339             current->signal |= S(signr);
340             continue;
    ! 如果对应的信号位图被阻塞，则继续循环
341         }
342         sa = current->sigaction + signr - 1;
    ! 获取对应的信号处理结构（该结构中含有信号处理句柄），请注意
    ! 第 327 行，增加了一，所以这里要减一
343     }
344     if (sa->sa_handler == SIG_IGN) {
345         if (signr != SIGCHLD)
346             continue;
    ! 信号不是SIGCHLD，即不是子进程接送，则继续循环
347     /* check for SIGCHLD: it's special */
    ! 检查SIGCHLD信号，它是特殊的
348     while (sys_waitpid(-1,NULL,WNOHANG) > 0)
349         /* nothing */;
    ! 等待子进程结束，sys_waitpid来源于lib.a，不作注释
350     continue;
351     }
352     if (sa->sa_handler == SIG_DFL) {
    ! 如果信号处理函数为系统缺省的
353         if (current->pid == 1)
354             continue;
355         switch (signr) {
356             case SIGCONT: case SIGCHLD: case SIGWINCH:
357                 continue;
358
359             case SIGSTOP: case SIGTSTP: case SIGTTIN: case SIGTTOU:
360                 if (current->flags & PF_PTRACED)
361                     continue;
362                 current->state = TASK_STOPPED;
    ! 只要是上面信号中任何一个，则循环之，否则置当前任务进入僵死作态
363                 current->exit_code = signr;
    ! 置退出码
364                 if (!(current->p_pptr->sigaction[SIGCHLD-1].sa_flags &
365                         SA_NOCLDSTOP))
366                     notify_parent(current);
367                     schedule();

```

! 重新调度, 定义于Kernel/sched.c

```

368         continue;
369
370         case SIGQUIT: case SIGILL: case SIGTRAP:
371         case SIGIOT: case SIGFPE: case SIGSEGV:
372             if (core_dump(signr,regs))
373                 signr |= 0x80;
374             /* fall through */
375         default:
376             current->signal |= S(signr & 0x7f);
377             do_exit(signr);

```

! 调用do\_exit, 定义于Kernel/exit.c

```

378     }
379     }
380     /*
381     * OK, we're invoking a handler
382     */
383     if (regs->orig_eax >= 0) {
384         if (regs->eax == ERESTARTNOHAND ||
385             (regs->eax == ERESTARTSYS && !(sa->sa_flags & SA_RESTART)))
386             regs->eax = EINTR;
387     }
388     handler_signal |= 1 << (signr-1);
389     mask &= ~sa->sa_mask;
390 }
391 if (regs->orig_eax >= 0 &&
392     (regs->eax == ERESTARTNOHAND ||
393      regs->eax == ERESTARTSYS ||
394      regs->eax == ERESTARTNOINTR)) {
395     regs->eax = regs->orig_eax;
396     regs->eip -= 2;
397 }
398 if (!handler_signal)      /* no handler will be called - return 0 */
399     return 0;
400 eip = regs->eip;
401 frame = (unsigned long *) regs->esp;
402 signr = 1;
403 sa = current->sigaction;
404 for (mask = 1 ; mask ; sa++,signr++,mask += mask) {
405     if (mask > handler_signal)
406         break;
407     if (!(mask & handler_signal))
408         continue;
409     setup_frame(sa,&frame,eip,regs,signr,oldmask);

```

```

410         eip = (unsigned long) sa->sa_handler;
411         if (sa->sa_flags & SA_ONESHOT)
412             sa->sa_handler = NULL;
413 /*force a supervisor-mode page-in of the signal handler to reduce races */
414         __asm__("testb $0,%%fs:%0": :"m" (*(char *) eip));
415         regs->cs = USER_CS; regs->ss = USER_DS;
416         regs->ds = USER_DS; regs->es = USER_DS;
417         regs->gs = USER_DS; regs->fs = USER_DS;
418         current->blocked |= sa->sa_mask;
419         oldmask |= sa->sa_mask;
420     }
421     regs->esp = (unsigned long) frame;
422     regs->eip = eip;           /* "return" to the first handler */
423     current->tss.trap_no = current->tss.error_code = 0;
424     return 1;
425 }
426

```

## Kernel/printk.c (部分代码)

! 核心专用的消息打印函数，类似于 printf ()

× 参数和 printf 相同，不在注释

```

142 asmlinkage int printk(const char *fmt, ...)
143 {
144     va_list args;
145     ! typedef char* va_list
146     int i;
147     char *msg, *p, *buf_end;
148     static char msg_level = -1;
149     ! 要打印消息的优先级
150     long flags;
151     ! save_flags(flags);
152     ! 保存寄存器值，定义于Include/asm/system.h
153     cli();
154     ! 关中断
155     va_start(args, fmt);
156     ! 在 1.0 的核心中我们看不到 va_start 的定义

```

! 它被定义在 gcc 中，用于变参函数中，关于

! 变参在关于C语言书有描述

153      i = vsprintf(buf + 3, fmt, args); /\* hopefully i < sizeof(buf)-4 \*/

! 使用格式串fmt将参数列表args输出到buf+3 开始处（注意buf的大小为 1024,

! 并且输出开始于 buf+3 处，所以超过 1000 个字符的话将会被丢弃掉）

! vsprintf () 不在注释，可以参看各种介绍 C 语言的书籍

! 为什么会在 buf+3 处开始呢？

! 因为前面的 3 个字符将会被写上优先级（形如：<2>）

! 优先级是用来让核心决定要输出的字符串是直接写到屏幕上，还是写到 log 中。

! 1.0 核心中对消息的输出处理共有 8 种优先级 (0-7)

! 其中 0 级最高，7 最低。

154      buf\_end = buf + 3 + i;

! buf\_end指向字符串的结尾

155      va\_end(args);

156      for (p = buf + 3; p < buf\_end; p++) {

157      msg = p;

158      if (msg\_level < 0) {

159      if (

160      p[0] != '<' ||

161      p[1] < " ||

162      p[1] > '7' ||

163      p[2] != '>'

164      ) {

! 检查输出消息优先级是否在规定的范围内。在则向前跳 3 个字符，即跳到了

! “buf” 头部，然后把优先级写入前 3 个字符处

165      p -= 3;

166      p[0] = '<;

167      p[1] = DEFAULT\_MESSAGE\_LOGLEVEL - 1 + '0';

168      p[2] = '>;

169      } else

170      msg += 3;

171      msg\_level = p[1] - '0';

! 否则，跳过 3 个字符，即优先级的字符串，取的第一个字符，付给 msg\_level，作为

! 消息优先级。

172      }

173      for (; p < buf\_end; p++) {

174      log\_buf[(log\_start+log\_size) & (LOG\_BUF\_LEN-1)] = \*p;

! 把要打印的字符一个个放入log\_buf，当不能在屏幕上打印时，可以写入log中

175      if (log\_size < LOG\_BUF\_LEN)

176      log\_size++;

177      else

178      log\_start++;

179      logged\_chars++;

180      if (\*p == '\n')

```

181         break;
182     }
183     if (msg_level < console_loglevel && console_print_proc) {
    ! 消息优先级小于console_loglevel时并且console_print_proc不为空
    ! 打印字符串到屏幕上
    ! console_print_proc对应于Drivers/char/console.c的console_print
184     char tmp = p[1];
185     p[1] = '\0';
186     (*console_print_proc)(msg);
187     p[1] = tmp;
188   }
189   if (*p == '\n')
190     msg_level = -1;
191 }
192 restore_flags(flags);
    ! 恢复标志寄存器
193 wake_up_interruptible(&log_wait);
    ! 唤醒可中断等待队列，定义于Kernel/sched.c
194 return i;
    ! 返回输出字符串的大小
195 }

```

```

197 /*
198 * The console driver calls this routine during kernel initialization
199 * to register the console printing procedure with printk() and to
200 * print any messages that were printed by the kernel before the
201 * console driver was initialized.
202 */
    ! 控制台驱动在内核初始化期间调用这个程序来注册控制台打印函数
    ! 并且在控制台驱动初始化完成后，可用来打印任何信息。
203 void register_console(void (*proc)(const char *))
204 {
205     int i,j;
206     int p = log_start;
207     char buf[16];
208     char msg_level = -1;
209     char *q;
210
211     console_print_proc = proc;
    ! 设置控制台打印函数
212
213     for (i=0,j=0; i < log_size; i++) {

```

```
214     buf[j++] = log_buf[p];
215     p++; p &= LOG_BUF_LEN-1;
216     if (buf[j-1] != '\n' && i < log_size - 1 && j < sizeof(buf)-1)
217         continue;
218     buf[j] = 0;
219     q = buf;
220     if (msg_level < 0) {
221         msg_level = buf[1] - ' ';
222         q = buf + 3;
223     }
224     if (msg_level < console_loglevel)
225         (*proc)(q);
226     if (buf[j-1] == '\n')
227         msg_level = -1;
228     j = 0;
229 }
230 }
```

## Kernel/vsprintf.c (部分代码)

! 把字符串转换成整数值后返回，并且跳过已转换的字符，让  
! \*endp 指向跳过后的字符串开始处

- × cp = 指向要转换的字符串指针
  - × endp = 指向被转换字符串的后面的内容
  - × base = 数进制

```
17 unsigned long simple strtoul(const char *cp,char **endp,unsigned int base)
18 {
19     unsigned long result = 0,value;
20
21     if (!base) {
22         ! 如果base = 0
23         base = 10;
24         ! 则让base = 10
25         if (*cp == '0') {
26             ! 如果传过来的第一个字符是'0'的话
27             base = 8;
28             ! 则让base = 8
29             cp++;
30         }
31     }
32 }
```

```

    ! 指针向前加一
26         if ((*cp == 'x') && isxdigit(cp[1])) {
    ! 如果第二个字符是'x'并且第 3 个字符是数字的话
27             cp++;
28             base = 16;
    ! 则跳过第二个字符，并让base=16
29         }
30     }
31 }
32 while (isxdigit(*cp) && (value = isdigit(*cp) ? *cp - '0' : islower(*cp)
33     ? toupper(*cp) : *cp) - 'A' + 10) < base) {
34     result = result * base + value;
35     cp++;
    ! 字符转换成整数，例如：AB转换后为 171
36 }
37     if (endp)
38         *endp = (char *)cp;
    ! 指向开始处
39     return result;
    ! 返回转换后的值
40 }

```

## Kernel/fork.c (部分代码)

```

    ! 为新进程取得不重复的进程号，并返回在任务数组中的任务编号
38 static int find_empty_process(void)
39 {
40     int free_task;
41     int i, tasks_free;
42     int this_user_tasks;
43
44 repeat:
45     if ((++last_pid) & 0xffff8000)
46         last_pid = 1;
    ! 通过第 45 行我们可以看到系统的最大进程号是 0x8000-1,
    ! 如果大于它时会让进程号从 1 开始（跳过任务 0）
47     this_user_tasks = 0;
48     tasks_free = 0;

```

```

49     free_task = -EAGAIN;
50     i = NR_TASKS;
    ! i等于系统的最大任务数
51     while (--i > 0) {
52         if (!task[i]) {
53             free_task = i;
54             tasks_free++;
    ! 如果对应 i 的任务号没有被占用, 让 free_task 等于找到的任务号, 同时让可用的任务
    ! 数自增一
55         continue;
56     }
57     if (task[i]->uid == current->uid)
58         this_user_tasks++;
59     if (task[i]->pid == last_pid || task[i]->pgrp == last_pid ||
60         task[i]->session == last_pid)
61         goto repeat;
62     }
63     if (tasks_free <= MIN_TASKS_LEFT_FOR_ROOT ||
64         this_user_tasks > MAX_TASKS_PER_USER)
65         if (current->uid)
66             return -EAGAIN;
67     return free_task;
    ! 返回找到的任务号
68 }
69
    ! 拷贝老的文件描述符到新的
    × old_file=老文件描述符
70 static struct file * copy_fd(struct file * old_file)
71 {
72     struct file * new_file = get_empty_filp();
    ! 从全局文件描述符中取个新的描述符, 定义于Fs/file_table.c
73     int error;
74
75     if (new_file) {
76         memcpy(new_file,old_file,sizeof(struct file));
    ! 使用老的文件描述符, 填充新的描述符
77         new_file->f_count = 1;
78         if (new_file->f_inode)
79             new_file->f_inode->i_count++;
    ! 新描述符占用了I节点, 则增加其引用计数。
80         if (new_file->f_op && new_file->f_op->open) {
81             error = new_file->f_op->open(new_file->f_inode,new_file);
    ! 调用对应的open函数, 打开之
82             if (error) {

```

```

    ! 如果失败，则释放其I节点，iput定义于Fs/inode.c
83         iput(new_file->f_inode);
84         new_file->f_count = 0;
85         new_file = NULL;
86     }
87 }
88 }
89 return new_file;
    ! 返回新的文件描述符
90 }
91

119 /*
120 * Ok, this is the main fork-routine. It copies the system process
121 * information (task[nr]) and sets up the necessary registers. It
122 * also copies the data segment in its entirety.
123 */
    ! 好，这是fork()的主程序。它拷贝系统进程的信息（task[nr]）并且设置必要的
    ! 寄存器。它也完全拷贝数据段。
    ! 创建新的进程
    ! regs=从用户态进入核心时的堆栈排布结构

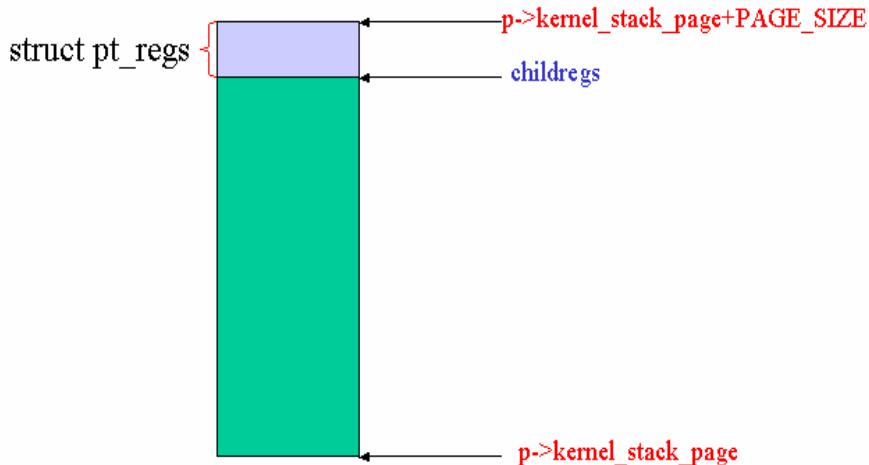
124 asmlinkage int sys_fork(struct pt_regs regs)
125 {
126     struct pt_regs * childregs;
127     struct task_struct *p;
128     int i,nr;
129     struct file *f;
130     unsigned long clone_flags = COPYVM | SIGCHLD;
131
132     if(!(p = (struct task_struct*) get_free_page(GFP_KERNEL)))
133         goto bad_fork;
    ! 获取一页内存，如果失败跳转到bad_fork，定义于Mm/swap.c
134     nr = find_empty_process();
    ! 查找空闲的任务号并且设置全局进程号（last_pid），定义于本文件中
135     if (nr < 0)
136         goto bad_fork_free;
    ! 找到的任务号小于0,跳转到bad_fork_free
137     task[nr] = p;
138     *p = *current;
    ! 复制父进程的信息到获取的内存中
139     p->did_exec = 0;
140     p->kernel_stack_page = 0;
141     p->state = TASK_UNINTERRUPTIBLE;
    ! 置子进程为不可中断状态

```

```

142     p->flags &= ~(PF_PTRACED|PF_TRACESYS);
    ! 去除子进程的PF_PTRACED和PF_TRACESYS位
143     p->pid = last_pid;
    ! 设置子进程的进程号
144     p->swappable = 1;
145     p->p_pptr = p->p_opptr = current;
146     p->p_cptr = NULL;
147     SET_LINKS(p);
    ! 把该任务加入任务双向列表中，定义于Include/linux/sched.h
148     p->signal = 0;
149     p->it_real_value = p->it_virt_value = p->it_prof_value = 0;
150     p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0;
151     p->leader = 0;      /* process leadership doesn't inherit */
152     p->utime = p->stime = 0;
153     p->cutime = p->cstime = 0;
154     p->min_flt = p->maj_flt = 0;
155     p->cmin_flt = p->cmaj_flt = 0;
156     p->start_time = jiffies;
157 /*
158 * set up new TSS and kernel stack
159 */
    ! 设置新的TSS和核心态堆栈
160     if (!(p->kernel_stack_page = get_free_page(GFP_KERNEL)))
161         goto bad_fork_cleanup;
    ! 为新进程在核心态下执行时取的一页内存作为其核心堆栈
162     p->tss.es = KERNEL_DS;
163     p->tss.cs = KERNEL_CS;
164     p->tss.ss = KERNEL_DS;
165     p->tss.ds = KERNEL_DS;
166     p->tss.fs = USER_DS;
167     p->tss.gs = KERNEL_DS;
168     p->tss.ss0 = KERNEL_DS;
169     p->tss.esp0 = p->kernel_stack_page + PAGE_SIZE;
170     p->tss.tr = TSS(nr);
    ! 设置TSS（任务状态段）
171     childregs = ((struct pt_regs *) (p->kernel_stack_page + PAGE_SIZE)) - 1;

```



为什么这句代码

`childregs = ((struct pt_regs *) (p->kernel_stack_page + PAGE_SIZE)) - 1;`

执行后，`childregs`会指向如图中所标处呢，原因是该代码先做了个强制转化工作。

转换成`(struct pt_regs *)`类型后在减去1，此时减去1并不是简单的减4个字节，而是`struct pt_regs`结构的大小，所以会象图中所标！

图 Kernel/fork.c-1

! `childregs`指向核心堆栈顶部，原因请看图Kernel/fork.c-1

```

172     p->tss.esp = (unsigned long) childregs;
173     p->tss.eip = (unsigned long) ret_from_sys_call;
174     *childregs = regs;
    ! 把传入的堆栈寄存器内容复制到图Kernel/fork.c-1 中 struct pt_regs 中
175     childregs->eax = 0;
    ! 新进程为返回 0 原因，便是这句代码的原因
176     p->tss.back_link = 0;
177     p->tss.eflags = regs.eflags & 0xffffcfff;      /* iopl is always 0 for a new process */
    ! 对于新进程标志寄存器eflag的iopl位总是 0
178     if (IS_CLONE) {
        ! 如果当前进程是调用sys_clone系统调用的话，则做该块代码
179         if (regs.ebx)
180             childregs->esp = regs.ebx;
181             clone_flags = regs.ecx;
182             if (childregs->esp == regs.esp)
183                 clone_flags |= COPYVMM;
184         }
185         p->exit_signal = clone_flags & CSIGNAL;
186         p->tss.ldt = LDT(nr);
    ! 把新任务nr的局部描述符表选择符送入任务状态段的ldt
187         if (p->ldt) {
            ! 如果当前任务的ldt不为空

```

```

188         p->ldt = (struct desc_struct*) vmalloc(LDT_ENTRIES*LDT_ENTRY_SIZE);
        ! 则让其指向新分配的内存
189         if (p->ldt != NULL)
190             memcpy(p->ldt, current->ldt, LDT_ENTRIES*LDT_ENTRY_SIZE);
        ! 分配成功的话，则拷贝当前进程的ldt
191     }
192     p->tss.bitmap = offsetof(struct tss_struct, io_bitmap);
        ! 计算io_bitmap在struct tss_struct中的偏移值
193     for (i = 0; i < IO_BITMAP_SIZE+1 ; i++) /* IO bitmap is actually SIZE+1 */
194         p->tss.io_bitmap[i] = ~0;
        ! 置位tss中的I/O 比特位图
195     if (last_task_used_math == current)
196         __asm__("clts ; fnsave %0 ; frstor %0": "=m" (p->tss.i387));
        ! 如果当前任务使用了协处理器则保存其上下文
197     p->semun = NULL; p->shm = NULL;
198     if (copy_vm(p) || shm_fork(current, p))
199         goto bad_fork_cleanup;
200     if (clone_flags & COPYFD) {
        ! 如果是克隆的
201         for (i=0; i<NR_OPEN;i++)
202             if ((f = p->filp[i]) != NULL)
203                 p->filp[i] = copy_fd(f);
        ! 如果当前有文件是打开的，则拷贝其文件描述符到新创建的任务中
        ! copy_fd定义于本文件
204     } else {
205         for (i=0; i<NR_OPEN;i++)
206             if ((f = p->filp[i]) != NULL)
207                 f->f_count++;
        ! 否则，则将对应文件的打开次数加一
208     }
209     if (current->pwd)
210         current->pwd->i_count++;
211     if (current->root)
212         current->root->i_count++;
213     if (current->executable)
214         current->executable->i_count++;
        ! 当前进程的pwd, root, executable引用计数加 1
215     dup_mmap(p);
216     set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
        ! 在gdt中设置新任务的tss
217     if (p->ldt)
218         set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,p->ldt, 512);
        ! 如果设置了当前任务的ldt，则在gdt设置该ldt
219     else

```

```

220      set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&default_ldt, 1);
    ! 否则，设置缺省的ldt
221
222      p->counter = current->counter >> 1;
    ! 设置任务运行时间滴答数
223      p->state = TASK_RUNNING; /* do this last, just in case */
    ! 设当前任务的状态为可运行状态（即可以被调度器调度执行）
224      return p->pid;
    ! 返回新的进程号
225 bad_fork_cleanup:
226      task[nr] = NULL;
227      REMOVE_LINKS(p);
228      free_page(p->kernel_stack_page);
229 bad_fork_free:
230      free_page((long) p);
231 bad_fork:
232      return -EAGAIN;
    ! 从 225 到这，都是在做fork时出错情况的处理。
233 }
234

```

## Kernel/sys\_call.s

```

1 /*
2  * linux/kernel/sys_call.S
3  *
4  * Copyright (C) 1991, 1992 Linus Torvalds
5  */
6
7 /*
8  * sys_call.S contains the system-call and fault low-level handling routines.
9  * This also contains the timer-interrupt handler, as well as all interrupts
10 * and faults that can result in a task-switch.
11 *
12 * NOTE: This code handles signal-recognition, which happens every time
13 * after a timer-interrupt and after each system call.
14 *
15 * I changed all the .align's to 4 (16 byte alignment), as that's faster

```

```

16 * on a 486.
17 *
18 * Stack layout in 'ret_from_system_call':
19 *      ptrace needs to have all regs on the stack.
20 *      if the order here is changed, it needs to be
21 *      updated in fork.c:copy_process, signal.c:do_signal,
22 *      ptrace.c and ptrace.h
23 *
24 *      0(%esp) - %ebx
25 *      4(%esp) - %ecx
26 *      8(%esp) - %edx
27 *      C(%esp) - %esi
28 *      10(%esp) - %edi
29 *      14(%esp) - %ebp
30 *      18(%esp) - %eax
31 *      1C(%esp) - %ds
32 *      20(%esp) - %es
33 *      24(%esp) - %fs
34 *      28(%esp) - %gs
35 *      2C(%esp) - orig_eax
36 *      30(%esp) - %eip
37 *      34(%esp) - %cs
38 *      38(%esp) - %eflags
39 *      3C(%esp) - %oldesp
40 *      40(%esp) - %oldss
41 */
42
43 #include <linux/segment.h>
44
45 EBX      = 0x00
46 ECX      = 0x04
47 EDX      = 0x08
48 ESI      = 0x0C
49 EDI      = 0x10
50 EBP      = 0x14
51 EAX      = 0x18
52 DS       = 0x1C
53 ES       = 0x20
54 FS       = 0x24
55 GS       = 0x28
56 ORIG_EAX = 0x2C
57 EIP      = 0x30
58 CS       = 0x34
59 EFLAGS   = 0x38

```

```

60 OLDESP      = 0x3C
61 OLDSS       = 0x40
62
63 CF_MASK     = 0x00000001
64 IF_MASK     = 0x00000200
65 NT_MASK     = 0x00004000
66 VM_MASK     = 0x00020000
67
68 /*
69 * these are offsets into the task-struct.
70 */
71 state        = 0
72 counter      = 4
73 priority     = 8
74 signal       = 12
75 blocked      = 16
76 flags        = 20
77 errno        = 24
78 dbgreg6     = 52
79 dbgreg7     = 56
80
81 ENOSYS = 38
82
83 .globl _system_call,_lcall7
84 .globl _device_not_available, _coprocessor_error
85 .globl _divide_error,_debug,_nmi,_int3,_overflow,_bounds,_invalid_op
86 .globl _double_fault,_coprocessor_segment_overrun
87 .globl _invalid_TSS,_segment_not_present,_stack_segment
88 .globl _general_protection,_reserved
89 .globl _alignment_check,_page_fault
90 .globl ret_from_sys_call
91
92 #define SAVE_ALL \
93     cld; \
94     ! 清方向位, df=0
95     push %gs; \
96     ! 保存用户态的gs
97     push %fs; \
98     ! 保存用户态的fs
99     push %es; \
100    ! 保存用户态的es
101    push %ds; \
102    ! 保存用户态的ds
103    pushl %eax; \

```

```

99      pushl %ebp; \
100     pushl %edi; \
101     pushl %esi; \
102     pushl %edx; \
103     pushl %ecx; \
104     pushl %ebx; \
! 从 98 行到此
! 把系统调用相应的C函数的参数入栈
105    movl $(KERNEL_DS),%edx; \
106    mov %dx,%ds; \
! ds指向内核数据段
107    mov %dx,%es; \
! es指向内核数据段
108    movl $(USER_DS),%edx; \
109    mov %dx,%fs;
! fs指向局部数据段
110
111 #define RESTORE_ALL \
! RESTORE_ALL对应于上面的SAVE_ALL，不过作用相反
112    cmpw $(KERNEL_CS),CS(%esp); \
! 察看是否在核心态下
113    je 1f; \
! 是则跳到 1 处，否则先作 114 到 116 的代码
114    movl _current,%eax; \
115    movl dbgreg7(%eax),%ebx; \
! 取的当前任务的硬编码用于调试的寄存器
116    movl %ebx,%db7; \
117 1:   popl %ebx; \
118    popl %ecx; \
119    popl %edx; \
120    popl %esi; \
121    popl %edi; \
122    popl %ebp; \
123    popl %eax; \
124    pop %ds; \
125    pop %es; \
126    pop %fs; \
127    pop %gs; \
128    addl $4,%esp; \
129    iret
! 从 116 到这，弹出所有入栈得寄存器，最后中断返回！
130
! lcall7 用来兼容不同的类 Unix 系统的系统调用。
! 1.0 的核心还没有提供对 iBCS2 的支持，这里只是个接口

```

! Linux 支持 Intel 二进制兼容规范标准的版本 2 (iBCS2) ,iBCS2 的规范中规定了所有基于 x86 的 Unix 系统的应用程序的标准内核接口，这些系统不仅包括 Linux，而且还包括其它自由的 x86 Unix (例如 FreeBSD)，也还包括 Solaris/x86，SCO Unix 等等。这些标准接口使得为其它 Unix 系统开发的二进制商业软件在 Linux 系统中能够直接运行，反之亦然。

cs	esp+0x38	EFLAGS
eip	esp+0x34	CS
eflags	esp+0x30	EIP
eax	esp+0x2C	ORIG_EAX
gs	esp+0x28	GS
fs	esp+0x24	FS
es	esp+0x20	ES
ds	esp+0x1C	DS
eax	esp+0x18	EAX
ebp	esp+0x14	EBP
edi	esp+0x10	EDI
esi	esp+0x0C	ESI
edx	esp+0x08	EDX
ecx	esp+0x04	ECX
ebx	esp+0x00	EBX

图 Kernel/sys\_call.s - 1

```

131 .align 4
132 _lcall7:
133     pushfl
    ! 标志寄存器入栈
    # We get a different stack layout with call gates,
    ! 通过调用门我们可以得到个不同的栈布局

134     pushl %eax
    ! 系统调用号入栈
    # which has to be cleaned up later..
    ! 我们会在后面清除它

135     SAVE_ALL
    ! 定义于本文件中，见 92 行

136     movl EIP(%esp),%eax
    ! 把 eflags 赋给 eax
    # due to call gates, this is eflags, not eip..
    ! 由于是调用门，所以这是 eflags，不是 eip
    ! 请看图Kernel/sys_call.s - 1 注释

137     movl CS(%esp),%edx
    ! eip 赋给 edx
    # this is eip..
    ! 这是eip

138     movl EFLAGS(%esp),%ecx # and this is cs..

```

```

        ! 这是cs
139      movl %eax,EFLAGS(%esp)
        ! 把eflags送入原来cs的放置处
140      movl %edx,EIP(%esp)    # Now we move them to their "normal" places
                                ! 现在，我们把他们移到正常的位置
                                ! eip送入原来eflags的位置处
141      movl %ecx,CS(%esp)
        ! cs送入原来eip的位置处
142      movl %esp,%eax
        ! 目前的 esp 值送入 eax，也就是
        ! 图Kernel/sys_call.s-1 中ebx处
143      pushl %eax
        ! 堆栈指针入栈
144      call _iABI_emulate
        ! 定义于ibcs/emulate.c
145      popl %eax
        ! 弹出堆栈指针
146      jmp ret_from_sys_call
        ! 跳转到ret_from_sys_call， 定义于本文件中
147
148 .align 4
149 handle_bottom_half:
150     pushfl
151     incl _intr_count
152     sti
153     call _do_bottom_half
154     popfl
155     decl _intr_count
156     jmp 9f
157 .align 4
158 reschedule:
159     pushl $ret_from_sys_call
160     jmp _schedule
161 .align 4
        ! 系统调用的入口处
162 _system_call:
163     pushl %eax          # save orig_eax
        ! 保存eax， 其实就是系统调用号
164     SAVE_ALL
        ! 定义于本文件
165     movl $-ENOSYS,EAX(%esp)
        ! 把-ENOSYS送入图Kernel/sys_call.s-1 的esp+0x18 处
166     cmpl _NR_syscalls,%eax
        ! 比较传入的系统调用号是否超出了系统最大的调用号

```

```

167      jae ret_from_sys_call
    ! 系统调用号超出范围，则跳到ret_from_sys_call处执行，定义于本文件
168      movl _current,%ebx
    ! 将当前任务结构地址送入ebx
169      andl $~CF_MASK,EFLAGS(%esp)    # clear carry - assume no errors
    ! 清 CF 标志位，假设没有错误

```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	IF	SF	SF		AF		PF		CF

图 Kernel/sys\_call.s-1

```

    ! 图Kernel/sys_call.s-1 描述了标志寄存器中的内容，所以其实还清除了PF标志
170      movl $0,errno(%ebx)
    ! 把 0 送入当前任务的errno字段
171      movl %db6,%edx
172      movl %edx,dbggreg6(%ebx) # save current hardware debugging status
    ! 把当前的硬件debug状态写入当前任务的第 52 个字节处
173      testb $0x20,flags(%ebx)    # PF_TRACESYS
    ! 测试当前任务的flags是否是 0x20
174      jne 1f
    ! 如果不是，则跳到处执行
175      call _sys_call_table(%eax,4)
    ! 调用对应于 eax(系统调用号)的系统调用，调用地址=_sys_call_table+%eax*4
    ! 以 fork () 来讲，它的_NR_fork=2,所以对应于 sys_fork()函数！
    ! sys_fork(), 定义于Kerenl/fork.c
176      movl %eax,EAX(%esp)        # save the return value
    ! eax中保存的是系统调用后的返回值，并把该值送esp+0x18 处
177      movl errno(%ebx),%edx
    ! 取当前任务的errno处值送入edx
178      negl %edx
    ! edx = -edx
179      je ret_from_sys_call
    ! ZF=1，则跳转到ret_from_sys_call执行，即第 178 行中edx=0
180      movl %edx,EAX(%esp)
    ! 否则，把edx送入esp+0x18 处
181      orl $(CF_MASK),EFLAGS(%esp)    # set carry to indicate error
    ! 设置CF=1，指示有错误发生
182      jmp ret_from_sys_call
    ! 跳转到ret_from_sys_call执行
183 .align 4
184 1:   call _syscall_trace
185      movl ORIG_EAX(%esp),%eax
186      call _sys_call_table(%eax,4)
187      movl %eax,EAX(%esp)        # save the return value
188      movl _current,%eax
189      movl errno(%eax),%edx

```

```

190      negl %edx
191      je 1f
192      movl %edx,EAX(%esp)
193      orl $(CF_MASK),EFLAGS(%esp)    # set carry to indicate error
194 1:   call _syscall_trace
195
196      .align 4,0x90
197 ret_from_sys_call:
198      cmpl $0,_intr_count
        ! 检查_intr_count是否为 0
199      jne 2f
        ! 不为 0 跳到 2 处
200      movl _bh_mask,%eax
        ! 取得bottom half 的bh_mask
201      andl _bh_active,%eax
        ! 如果bh没有被激活，则跳转到handle_bottom_half处理
202      jne handle_bottom_half
203 9:   movl EFLAGS(%esp),%eax      # check VM86 flag: CS/SS are
        ! 检查VM86flag
204      testl $(VM_MASK),%eax      # different then
205      jne 1f
        ! 没有跳转到 1 处
206      cmpw $(KERNEL_CS),CS(%esp)    # was old code segment supervisor ?
207      je 2f
        ! 如果是超级代码段，则跳转到 2 处
208 1:   sti
        ! 开中断
209      orl $(IF_MASK),%eax      # these just try to make sure
210      andl $~NT_MASK,%eax      # the program doesn't do anything
211      movl %eax,EFLAGS(%esp)    # stupid
212      cmpl $0,_need_resched
213      jne reschedule
        ! 确认该任务是否要重新调度
214      movl _current,%eax
215      cmpl _task,%eax      # task[0] cannot have signals
216      je 2f
        ! 如果当前任务是任务 0 话，则跳到 2 处执行
217      cmpl $0,state(%eax)      # state
218      jne reschedule
        ! 如果当前任务的状态是不是 0,则重新调度
219      cmpl $0,counter(%eax)      # counter
220      je reschedule
        ! 如果时间片用完也重新调度
221      movl blocked(%eax),%ecx

```

```

! 取屏蔽信号位图
222    movl %ecx,%ebx          # save blocked in %ebx for signal handling
223    notl %ecx
! 取反
224    andl signal(%eax),%ecx
! 取信号位图
225    jne signal_return
! 有信号要处理，调用信号处理函数，定义于本文件
226 2:   RESTORE_ALL
! 恢复寄存器并中断返回，定义于本文件
227 .align 4
228 signal_return:
229    movl %esp,%ecx
230    pushl %ecx
231    testl $(VM_MASK),EFLAGS(%ecx)
232    jne v86_signal_return
! 如果是vm86，则跳到v86 处理函数
233    pushl %ebx
! 信号值入栈
234    call _do_signal
! 调用do_signal， 定义于Kernel/signal.c
235    popl %ebx
236    popl %ebx
237    RESTORE_ALL
! 定义于本文件
238 .align 4
239 v86_signal_return:
240    call _save_v86_state
241    movl %eax,%esp
242    pushl %eax
243    pushl %ebx
244    call _do_signal
245    popl %ebx
246    popl %ebx
247    RESTORE_ALL
248
249 .align 4
250 _divide_error:
251    pushl $0          # no error code
252    pushl $_do_divide_error
253 .align 4,0x90
254 error_code:
255    push %fs
256    push %es

```

```

257    push %ds
258    pushl %eax
259    pushl %ebp
260    pushl %edi
261    pushl %esi
262    pushl %edx
263    pushl %ecx
264    pushl %ebx
265    movl $0,%eax
266    movl %eax,%db7          # disable hardware debugging...
267    cld
268    movl $-1, %eax
269    xchgl %eax, ORIG_EAX(%esp)  # orig_eax (get the error code. )
270    xorl %ebx,%ebx          # zero ebx
271    mov %gs,%bx             # get the lower order bits of gs
272    xchgl %ebx, GS(%esp)     # get the address and save gs.
273    pushl %eax              # push the error code
274    lea 4(%esp),%edx
275    pushl %edx
276    movl $(KERNEL_DS),%edx
277    mov %dx,%ds
278    mov %dx,%es
279    movl $(USER_DS),%edx
280    mov %dx,%fs
281    pushl %eax
282    movl _current,%eax
283    movl %db6,%edx
284    movl %edx,dbggreg6(%eax) # save current hardware debugging status
285    popl %eax
286    call *%ebx
287    addl $8,%esp
288    jmp ret_from_sys_call
289
290 .align 4
291 _coprocessor_error:
292    pushl $0
293    pushl $_do_coprocessor_error
294    jmp error_code
295
296 .align 4
297 _device_not_available:
298    pushl $-1          # mark this as an int
299    SAVE_ALL
300    pushl $ret_from_sys_call

```

```
301     movl %cr0,%eax
302     testl $0x4,%eax          # EM (math emulation bit)
303     je _math_state_restore
304     pushl $0                # temporary storage for ORIG_EIP
305     call _math_emulate
306     addl $4,%esp
307     ret
308
309 .align 4
310 _debug:
311     pushl $0
312     pushl $_do_debug
313     jmp error_code
314
315 .align 4
316 _nmi:
317     pushl $0
318     pushl $_do_nmi
319     jmp error_code
320
321 .align 4
322 _int3:
323     pushl $0
324     pushl $_do_int3
325     jmp error_code
326
327 .align 4
328 _overflow:
329     pushl $0
330     pushl $_do_overflow
331     jmp error_code
332
333 .align 4
334 _bounds:
335     pushl $0
336     pushl $_do_bounds
337     jmp error_code
338
339 .align 4
340 _invalid_op:
341     pushl $0
342     pushl $_do_invalid_op
343     jmp error_code
344
```

```
345 .align 4
346 _coprocessor_segment_overrun:
347     pushl $0
348     pushl $_do_coprocessor_segment_overrun
349     jmp error_code
350
351 .align 4
352 _reserved:
353     pushl $0
354     pushl $_do_reserved
355     jmp error_code
356
357 .align 4
358 _double_fault:
359     pushl $_do_double_fault
360     jmp error_code
361
362 .align 4
363 _invalid_TSS:
364     pushl $_do_invalid_TSS
365     jmp error_code
366
367 .align 4
368 _segment_not_present:
369     pushl $_do_segment_not_present
370     jmp error_code
371
372 .align 4
373 _stack_segment:
374     pushl $_do_stack_segment
375     jmp error_code
376
377 .align 4
378 _general_protection:
379     pushl $_do_general_protection
380     jmp error_code
381
382 .align 4
383 _alignment_check:
384     pushl $_do_alignment_check
385     jmp error_code
386
387 .align 4
388 _page_fault:
```

---

```

389      pushl $ _do_page_fault
390      jmp error_code

```

## L

### Lib/\_exit.c

```

1 /*
2  * linux/lib/_exit.c
3  *
4  * Copyright (C) 1991, 1992 Linus Torvalds
5  */
6
7 #define _LIBRARY_
8 #include <linux/unistd.h>
9

! 在 Init/main.c 中被调用，用于程序的退出操作
×exit_code=退出码

10 volatile void _exit(int exit_code)
11 {
12 fake_volatile:
13     __asm__("movl %1,%%ebx\n\t"
14             "int $0x80"
15             :/* no outputs */
16             :"a"(_NR_exit),"g"(exit_code));
17     goto fake_volatile;
! 该函数采用嵌入汇编执行sys_exit系统调用
18 }
19

```

### Lib/open.c

```

1 /*

```

```

2  *  linux/lib/open.c
3  *
4  *  Copyright (C) 1991, 1992  Linus Torvalds
5  */
6
7 #define _LIBRARY_
8 #include <linux/unistd.h>
9 #include <stdarg.h>
10
    ! 用户态方式下的 open 函数，其实该函数只是简单的对系统调用的
    ! sys_open 的包裹，请跳转到 sys_open 函数阅读。
    ! sys_open 定义于 Fs/open.c
    ! 关于这里的汇编如何和系统调用联系起来请看基础部分的说明
11 int open(const char * filename, int flag, ...)
12 {
13     register int res;
14     va_list arg;
15
16     va_start(arg,flag);
17     __asm__ ("movl %2,%%ebx\n\t"
18             "int $0x80"
19             :"=a" (res)
20             :"" (_NR_open), "g" ((long)(filename)), "c" (flag),
21             "d" (va_arg(arg,int)));
22     if (res>=0)
23         return res;
24     errno = -res;
25     return -1;
26 }
27

```

## M

### Mm/vmalloc.c (部分代码)

```

48 static int free_area_pages(unsigned long dindex, unsigned long index, unsigned long nr)
49 {
50     unsigned long page, *pte;

```

```

51
52     if (!(PAGE_PRESENT & (page = swapper_pg_dir[dindex])))
53         return 0;
    ! 如果对应dindex页目录没有被使用，直接返回 0
54     page &= PAGE_MASK;
    ! 让page按页对齐
55     pte = index + (unsigned long *) page;
    ! 得到第index个页表地址
56     do {
57         unsigned long pg = *pte;
58         *pte = 0;
59         if (pg & PAGE_PRESENT)
60             free_page(pg);
    ! 如果该页被使用，则释放之，定义于Mm/swap.c
61         pte++;
    ! 页表自增一项
62     } while (--nr);
63     pte = (unsigned long *) page;
64     for (nr = 0 ; nr < 1024 ; nr++, pte++)
65         if (*pte)
66             return 0;
    ! 检查交换页目录，如果被占用直接返回
67     set_pgdir(dindex,0);
    ! 置页目录为空
68     mem_map[MAP_NR(page)] = 1;
69     free_page(page);
    ! 定义于Mm/swap.c
70     invalidate();
    ! 刷新高速缓冲
71     return 0;
72 }

```

```

106 static int do_area(void * addr, unsigned long size,
107         int (*area_fn)(unsigned long,unsigned long,unsigned long))
108 {
109     unsigned long nr, dindex, index;
110
111     nr = size >> PAGE_SHIFT;
    ! 计算size大小的内存可以分多少页
112     dindex = (TASK_SIZE + (unsigned long) addr) >> 22;
    ! 计算addr在内核页目录中的索引
113     index = (((unsigned long) addr) >> PAGE_SHIFT) & (PTRS_PER_PAGE-1);
    ! 计算addr在页表中的索引

```

```

114     while (nr > 0) {
115         unsigned long i = PTRS_PER_PAGE - index;
116
117         if (i > nr)
118             i = nr;
119         nr -= i;
120         if (area_fn(dindex, index, i))
121             return -1;
122     index = 0;
123     dindex++;
124 }
125     return 0;
126 }
127
128 void vfree(void * addr)
129 {
130     struct vm_struct **p, *tmp;
131
132     if (!addr)
133         return;
134     if ((PAGE_SIZE-1) & (unsigned long) addr) {
135         printk("Trying to vfree() bad address (%p)\n", addr);
136         return;
137     }
138     for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
139         if (tmp->addr == addr) {
140             *p = tmp->next;
141             do_area(tmp->addr, tmp->size, free_area_pages);
142             kfree(tmp);
143             return;
144         }
145     }
146     printk("Trying to vfree() nonexistent vm area (%p)\n", addr);
147 }

```

148

## Mm/kmalloc.c (部分代码)

```
92 struct size_descriptor sizes[] = {  
93     { NULL, 32,127, 0,0,0,0 },  
94     { NULL, 64, 63, 0,0,0,0 },  
95     { NULL, 128, 31, 0,0,0,0 },  
96     { NULL, 252, 16, 0,0,0,0 },  
97     { NULL, 508, 8, 0,0,0,0 },  
98     { NULL, 1020, 4, 0,0,0,0 },  
99     { NULL, 2040, 2, 0,0,0,0 },  
100    { NULL, 4080, 1, 0,0,0,0 },  
101    { NULL, 0, 0, 0,0,0,0 }  
102};  
103  
104  
105 #define NBLOCKS(order)      (sizes[order].nbblocks)  
106 #define BLOCKSIZE(order)     (sizes[order].size)  
107  
108  
109
```

！ 检查系统定义的分页信息是否正确。

× start\_mem=可用内存起始值

× end\_mem=可用内存结束值

110 long **kmalloc\_init** (long start\_mem,long end\_mem)

111 {

112 int order;

113

114 /\*

115\* Check the static info array. Things will blow up terribly if it's

116 \*incorrect. This is a late "compile time" check.....

117 \*/

! 检查静态信息数组，假如它们不正确时危害非常大。

! 这是个后编译时检查

[118](#) for (order = 0;BLOCKSIZE(order);order++)

{

120 if ((NBLOCKS(order)\*BLOCKSIZE(order) + sizeof (struct page descriptor)) >

```

121      PAGE_SIZE)
    ! 如果每块的大小和块数相乘并加上page_descriptor的结构大小大于
    ! PAGE_SIZE的话，则打印出错内容后，死机！
122      {
123      printk ("Cannot use %d bytes out of %d in order = %d block m allocate\n",
124          NBLOCKS (order) * BLOCKSIZE (order) +
125              sizeof (struct page_descriptor),
126          (int) PAGE_SIZE,
127          BLOCKSIZE (order));
128      panic ("This only happens if someone messes with kmalloc");
129      }
130  }
131 return start_mem;
    ! 返回传入的值，在该函数中没有对其做任何操作
132 }
```

## Mm/swap.c (部分代码)

```

157 void swap_free(unsigned long entry)
158 {
159     struct swap_info_struct * p;
160     unsigned long offset, type;
161
162     if (!entry)
    ! 如果，entry为空，则直接返回
163         return;
164     type = SWP_TYPE(entry);
165     if (type == SHM_SWP_TYPE)
166         return;
    ! 如果取得的类型为SHM_SWP_TYPE则直接返回
167     if (type >= nr_swapfiles) {
168         printk ("Trying to free nonexistent swap-page\n");
169         return;
170     }
171     p = & swap_info[type];
```

```

    ! 让p指向通过类型索引到的swap信息
172     offset = SWP_OFFSET(entry);
    ! 计算交换页的页表指针
173     if (offset >= p->max) {
174         printk("swap_free: weirdness\n");
175         return;
176     }
177     if (!(p->flags & SWP_USED)) {
178         printk("Trying to free swap from unused swap-device\n");
179         return;
180     }
181     while (set_bit(offset,p->swap_lockmap))
182         sleep_on(&lock_queue);
183     if (offset < p->lowest_bit)
184         p->lowest_bit = offset;
185     if (offset > p->highest_bit)
186         p->highest_bit = offset;
187     if (!p->swap_map[offset])
188         printk("swap_free: swap-space map bad (entry %08lx)\n",entry);
189     else
190         if (!--p->swap_map[offset])
191             nr_swap_pagesclear_bit(offset,p->swap_lockmap))
193         printk("swap_free: lock already cleared\n");
194     wake_up(&lock_queue);
    ! 以上代码测试各种条件后，唤醒等待队列
195 }

```

```

489 /*
490 * Note that this must be atomic, or bad things will happen when
491 * pages are requested in interrupts (as malloc can do). Thus the
492 * cli/sti's.
493 */
    ! 注意它必须是原子互斥的，当页在中断时被请求时将会发生错误的事情
    (比如malloc)，所以要关中断
494 static inline void add_mem_queue(unsigned long addr, unsigned long * queue)
495 {
496     addr &= PAGE_MASK;
    ! 把页调整为按页对齐
497     *(unsigned long *)addr = *queue;
498     *queue = addr;
    ! 加入等待列表中
499 }
500

```

```

501 /*
502 * Free_page() adds the page to the free lists. This is optimized for
503 * fast normal cases (no error jumps taken normally).
504 *
505 * The way to optimize jumps for gcc-2.2.2 is to:
506 * - select the "normal" case and put it inside the if () { XXX }
507 * - no else-statements if you can avoid them
508 *
509 * With the above two rules, you get a straight-line execution path
510 * for the normal case, giving better asm-code.
511 */
    ! Free_page()把要释放的页加入释放列表。它为 fast 正常情况做了优化（正常
    ! 跳转没有错误发生）
    ! 该优化是对 gcc-2.2.2 的跳转做了优化：
    ! 选择“normal”情况用它来替代 if(){XXX}
    ! 没有 else 段意味着你可以避免使用它
    ! 通过上面的两个规格，在正常的情况下你可以得到一个流水的执行路径，
    ! 以获取最好的汇编代码

512 void free_page(unsigned long addr)
513 {
514     if (addr < high_memory) {
        ! 如果要释放的地址大于系统所拥有的最大
        ! 内存值，则跳出该代码块，同时在 537 行
        ! 打印出错误提示后，返回
515         unsigned short * map = mem_map + MAP_NR(addr);
        ! 让map等于要释放地址在内存映射位图中的值
516
517         if (*map) {
            ! 如果该页面存在
518             if (!(map & MAP_PAGE_RESERVED)) {
                ! 该页被确实进程占用
519                 unsigned long flag;
520
521                 save_flags(flag);
            ! 保存寄存器值，定义于Include/asm/system.h
522                 cli();          ! 关中断
523                 if (!--*map) {
                    ! 把取得的地址减一（因为，内存地址是从 0 开始编址的）
524                     if (nr_secondary_pages < MAX_SECONDARY_PAGES) {
                        ! 如果当前交换页记数小于系统预留的最大交换页面值
                        ! 则把该地址值加入交换页面的列表中，定义于本文件中
525                         add_mem_queue(addr,&secondary_page_list);
526                         nr_secondary_pages++;
                    ! 交换页记数自增一

```

```

527         restore_flags(flag);
528         return;
      ! 直接返回
529     }
530     add_mem_queue(addr,&free_page_list);
      ! 否则，把该地址值加入free_page_list中
531     nr_free_pages++;
      ! nr_free_pages自增一
532   }
533   restore_flags(flag);
534 }
535 return;
536 }

      ! 执行到这里的话，说明内存影射出了问题，打印错误提示后，退出！
537     printk("Trying to free free memory (%08lx): memory probably
corrupted\n",addr);
538     printk("PC = %08lx\n",*((unsigned long *)&addr)-1));
539     return;
540   }
541 }

543 /*
544 * This is one ugly macro, but it simplifies checking, and makes
545 * this speed-critical place reasonably fast, especially as we have
546 * to do things with the interrupt flag etc.
547 *
548 * Note that this #define is heavily optimized to give fast code
549 * for the normal case - the if-statements are ordered so that gcc-2.2.2
550 * will make *no* jumps for the normal code. Don't touch unless you
551 * know what you are doing.
552 */
      ! 这是个丑陋的宏，但是它简化检查，并且使得有临界的地方的速度可以适当的加快
      ! 特别是我们不根据中断标志做事情等等。
      ! 注意这个#define 对正常的情况做了很好的优化以得到更快的代码—这个 if 模块是根
      ! 据 gcc-2.2.2 规整的以使得它对于正常的代码时没有 jump 指令。不要修改它除非你
      ! 知道你在做什么。
553 #define REMOVE_FROM_MEM_QUEUE(queue,nr) \
554     cli(); \
      ! 关中断
555     if ((result = queue) != 0) { \
556         if (!(result & ~PAGE_MASK) && result < high_memory) { \
557             queue = *(unsigned long *) result; \
558             if (!mem_map[MAP_NR(result)]) { \
559                 mem_map[MAP_NR(result)] = 1; \

```

```

560             nr--; \
561 last_free_pages[index = (index + 1) & (NR_LAST_FREE_PAGES - 1)] = result; \
562             restore_flags(flag); \
563             return result; \
564         } \
565         printf("Free page %08lx has mem_map = %d\n",
566             result,mem_map[MAP_NR(result)]); \
567     } else \
568         printf("Result = 0x%08lx - memory map destroyed\n", result); \
569         queue = 0; \
570         nr = 0; \
571     } else if (nr) { \
572         printf(#nr " is %d, but "#queue " is empty\n",nr); \
573         nr = 0; \
574     } \
575     restore_flags(flag)

577 /*
578 * Get physical address of first (actually last :-) free page, and mark it
579 * used. If no free pages left, return 0.
580 *
581 * Note that this is one of the most heavily called functions in the kernel,
582 * so it's a bit timing-critical (especially as we have to disable interrupts
583 * in it). See the above macro which does most of the work, and which is
584 * optimized for a fast normal path of execution.
585 */

```

! 取的第一个（实际是最后一个）空闲页表，并标记它为可用，假如没有空闲  
 ! 页面了便返回 0  
 ! 注意该函数是核心中调用最多的函数之一，因此它有一个时间临界（尤其是  
 ! 在该函数中我们不得不禁止中断）。看上面的宏它将做大部分的工作，并且  
 ! 已经被优化了以得到最好的执行路径

```

586 unsigned long get_free_page(int priority)
587 {
588     extern unsigned long intr_count;
589     unsigned long result, flag;
590     static unsigned long index = 0;
591
592     /* this routine can be called at interrupt time via
593 malloc. We want to make sure that the critical
594 sections of code have interrupts disabled. -RAB
595 Is this code reentrant? */
596
597     if (intr_count && priority != GFP_ATOMIC) {

```

! 该程序在中断时间可以通过 malloc 调用。我们确信中断被禁时它是临界的代码—RAB

```

598     printk("gfp called nonatomically from interrupt %08lx\n",
599             ((unsigned long *)&priority)[-1]);
600     priority = GFP_ATOMIC;
601 }
602     save_flags(flag);
! 保存寄存器值, 定义于Include/asm/system.h
603 repeat:
604     REMOVE_FROM_MEM_QUEUE(free_page_list,nr_free_pages);
605     if (priority == GFP_BUFFER)
606         return 0;
607     if (priority != GFP_ATOMIC)
608         if (try_to_free_page())
609             goto repeat;
610     REMOVE_FROM_MEM_QUEUE(secondary_page_list,nr_secondary_pages);
611     return 0;
612 }

```

## Mm/memory.c (部分代码)

```

82 static void free_one_table(unsigned long * page_dir)
83 {
84     int j;
85     unsigned long pg_table = *page_dir;
86     unsigned long * page_table;
87
88     if (!pg_table)
89         return;
90     *page_dir = 0;
91     if (pg_table >= high_memory || !(pg_table & PAGE_PRESENT)) {
! 如果要释放的页表大于系统所拥有的最大内存或者该页表并没有被占用
! 则字节返回
92         printk("Bad page table: [%p]=%08lx\n",page_dir,pg_table);
93         return;
94     }
95     if (mem_map[MAP_NR(pg_table)] & MAP_PAGE_RESERVED)
96         return;
! 如果该页表是保留的也退出
97     page_table = (unsigned long *) (pg_table & PAGE_MASK);

```

```

! 对齐页表
98     for (j = 0 ; j < PTRS_PER_PAGE ; j++,page_table++) {
99         unsigned long pg = *page_table;
100
101         if (!pg)
102             continue;
103         *page_table = 0;
104         if (pg & PAGE_PRESENT)
    ! 如果当前页表指的内存存在，则调用free_page， 定义于Mm/swap.c
105             free_page(PAGE_MASK & pg);
106         else
107             swap_free(pg);          ! 定义于Mm/swap.c
108     }
109     free_page(PAGE_MASK & pg_table); Mm/swap.c
110 }

```

```

112 /*
113 * This function clears all user-level page tables of a process - this
114 * is needed by execve(), so that old pages aren't in the way. Note that
115 * unlike 'free_page_tables()', this function still leaves a valid
116 * page-table-tree in memory: it just removes the user pages. The two
117 * functions are similar, but there is a fundamental difference.
118 */

```

! 这个函数清空一个进程的所有的用户态的页表—execve（）函数  
 ! 需要它，所以对于老的页表会有问题。注意：它不象 free\_page\_tables，  
 ! 这个函数仍然一个内存中有效的页表树：它们正好是用户态的页表。  
 ! 这两个函数是相似的。但是它们还是有差别的。

```

119 void clear_page_tables(struct task_struct * tsk)
120 {
121     int i;
122     unsigned long pg_dir;
123     unsigned long * page_dir;
124
125     if (!tsk)
126         return;
127     if (tsk == task[0])
128         panic("task[0] (swapper) doesn't support exec()\n");
    ! 如果传入的任务指针是第 0 任务，则死机，因为任务 0 是不可以被释放的
129     pg_dir = tsk->tss.cr3;
    ! 取得当前任务的页目录
130     page_dir = (unsigned long *) pg_dir;
131     if (!page_dir || page_dir == swapper_pg_dir) {
132         printk("Trying to clear kernel page-directory: not good\n");
133         return;
  
```

```

    ! 如果为空，或者与swapper_pg_dir相等，则打印错误提示后，字节返回
134      }
135      if (mem_map[MAP_NR(pg_dir)] > 1) {
    ! 如果内存映射位图大于 1（说明该页内存被占用了
136          unsigned long * new_pg;
137
138          if (!(new_pg = (unsigned long*) get_free_page(GFP_KERNEL))) {
    ! 重新分配一页内存
139              oom(tsk);
    ! 如果失败，则打印错误提示
140              return;
    ! 返回
141      }
142      for (i = 768 ; i < 1024 ; i++)
143          new_pg[i] = page_dir[i];
    ! 从第 768 项开始拷贝，因为 1.0 核心核心段是从 3G开始的
144          free_page(pg_dir);
    ! 释放页目录，定义于Mm/swap.c
145          tsk->tss.cr3 = (unsigned long) new_pg;
    ! 指向新的页目录
146          return;
    ! 返回
147      }
    ! 否则，即对于的内存位图位并没有被占用
148      for (i = 0 ; i < 768 ; i++,page_dir++)
149          free_one_table(page_dir);
    ! 则释放之，该函数不做注释
150          invalidate();
    ! 刷新页高速缓冲
151          return;
    ! 返回
152  }
153
154 /*
155 * This function frees up all page tables of a process when it exits.
156 */
    ! 该函数当一个进程离开时释放所有的页表
157 void free_page_tables(struct task_struct * tsk)
158 {
159     int i;
160     unsigned long pg_dir;
161     unsigned long * page_dir;
162
163     if (!tsk)

```

```

164         return;
    ! 如果, 要释放的任务不存在则直接退出
165     if (tsk == task[0]) {
        ! 如果要释放的任务等于任务 0, 则打印错误提示后, 死机
166         printk("task[0] (swapper) killed: unable to recover\n");
167         panic("Trying to free up swapper memory space");
168     }
169     pg_dir = tsk->tss.cr3;  ! 获取当前任务的页目录
170     if (!pg_dir || pg_dir == (unsigned long) swapper_pg_dir) {
        ! 如果页目录不存在或者不等于交换页目录, 则打印错误提示, 返回!
171         printk("Trying to free kernel page-directory: not good\n");
172         return;
173     }
174     tsk->tss.cr3 = (unsigned long) swapper_pg_dir;
175     if (tsk == current)
176         __asm__ __volatile__ ("movl %0,%%cr3": :"a" (tsk->tss.cr3));
        ! 如果, 要释放页表的进程等于当前进程, 则把该进程的页目录送入cr3 寄存器中
177     if (mem_map[MAP_NR(pg_dir)] > 1) {
        ! 如果, 内存映射位图大于 1 (即对应的位图被置位了)
178         free_page(pg_dir);
        ! 释放内存, 定义于Mm/swap.c
179         return;
180     }
        ! 不然的话, 一页一页释放页表后, 最后, 释放页目录
181     page_dir = (unsigned long *) pg_dir;
182     for (i = 0 ; i < PTRS_PER_PAGE ; i++,page_dir++)
183         free_one_table(page_dir);      ! 定义于本文件中
184         free_page(pg_dir);          定义于Mm/swap.c
185         invalidate();            ! 刷新高速缓冲
186 }

268 /*
269 * a more complete version of free_page_tables which performs with page
270 * granularity.
271 */
    ! 按页执行的free_page_tables的一个完美版本
272 int unmap_page_range(unsigned long from, unsigned long size)
273 {
274     unsigned long page, page_dir;
275     unsigned long *page_table, *dir;
276     unsigned long poff, pcnt, pc;
277
278     if (from & ~PAGE_MASK) {      ! 校验地址 (from) 是否是按页对齐的
                                    ! 不是则打印错误提示, 返回-EINVAL

```

```

279         printk("unmap_page_range called with wrong alignment\n");
280         return -EINVAL;
281     }
282     size = (size + ~PAGE_MASK) >> PAGE_SHIFT;
    ! 根据size的大小，计算所占页表数
283     dir = PAGE_DIR_OFFSET(current->tss.cr3,from);
    ! 从当前进程的cr3 寄存器中取得页表目录进入点
284     poff = (from >> PAGE_SHIFT) & (PTRS_PER_PAGE-1);
    ! 计算from在页表目录中的偏移
285     if ((pcnt = PTRS_PER_PAGE - poff) > size)
286         pcnt = size;
    ! 如果映射内存的页表指针数（1024 个）减去 from 在页表中的偏移大于 size (size=
    ! 要释放页表计数，每一页页表对应于 4K内存），则让pcnt=size
287
288     for ( ; size > 0; ++dir, size -= pcnt,
289         pcnt = (size > PTRS_PER_PAGE ? PTRS_PER_PAGE : size)) {
290         if (!(*page_dir = *dir)) {
    ! 如果取得的页目录没有被占用的话，设置偏移为 0,跳过下面的代码
291             poff = 0;
292             continue;
293         }
294         if (!(page_dir & PAGE_PRESENT)) {
    ! 如果，取得的页目录没有设置上已经被使用的标志位，则打印错误提示，同时跳过下
    ! 面代码
295             printk("unmap_page_range: bad page directory.");
296             continue;
297         }
298         page_table = (unsigned long *)(PAGE_MASK & page_dir);
    ! 通过，页目录取得页表地址
299         if (poff) {
300             page_table += poff;
    ! 加上偏移
301             poff = 0;
302         }
303         for (pc = pcnt; pc--; page_table++) {
304             if ((*page = *page_table) != 0) {
    ! 如果，页表值不为 0，即被用于映射了内存
305                 *page_table = 0;
    ! 清零该页表中的地址值
306                 if (1 & page) {
    ! 如果，该页确实被占用（即第 0 位被置位）
307                     if (!(mem_map[MAP_NR(page)] & MAP_PAGE_RESERVED))
    ! 如果，通过页表映射的内存地址值确实存在，则做下面的代码
308                     if (current->rss > 0)

```

```

309             --<current>->rss;
    ! 如果, 当前进程的常驻页记数大于 0, 则减一
310             free_page(PAGE_MASK & page);
    ! 释放该页面, 定义于Mm/swap.c
311         } else
312             swap_free(page);
    ! 执行到这里, 说明该页并没有被占用 (即该页没有被任何进程占用,
    ! 但还在占用着内存中) 定义于Mm/swap.c
313     }
314 }
315 if (pcnt == PTRS_PER_PAGE) {
    ! 当pcnt=1024 (即该页目录所指的内存全部已经被释放了) 释放该页目录
316     *dir = 0;
317     free_page(PAGE_MASK & page_dir);
    定义于Mm/swap.c
318 }
319 }
320 invalidate();
    ! 刷新页交换高速缓冲
321 return 0;
322 }

```

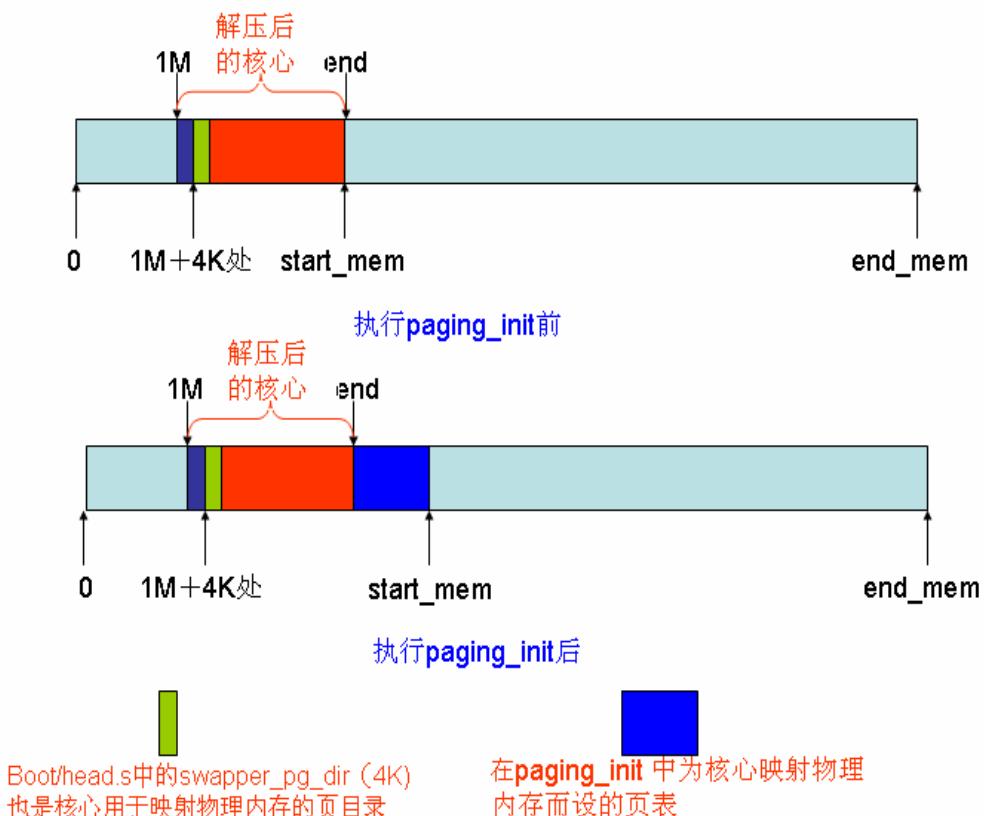


图 Mm/memory.c -1

```

1001 /*
1002 * paging_init() sets up the page tables - note that the first 4MB are
1003 * already mapped by head.S.
1004 *
1005 * This routines also unmaps the page at virtual kernel address 0, so
1006 * that we can trap those pesky NULL-reference errors in the kernel.
1007 */

```

! paging\_init()函数设置页表-注意起先的 4M 已经在 head.s 中被映射了  
 ! 该子程序没有映射核心虚拟地址 0，所以我们在内核中捕捉  
 ! 令人讨厌的 NULL 指针引用错误。  
 ! 该程序把机器所拥有的物理内存分页，分页情况填入页目录 swapper\_pg\_dir  
 ! 页目录 swapper\_pg\_dir 的第 0 项和第 768 项分别是向下对应相同的（即第 0 项  
 ! 和第 768 项是相同的，第 1 项和第 769 项的内容是相同的……）  
 ! 第 0 个页表是在 head.s 中\_pg0，第 1 个页表是从 PAGE\_ALIGN(start\_mem)  
 ! 开始的 4k，第 2 个页表是从 PAGE\_ALIGN(start\_mem)+4k 开始的 4k……  
 ! 直到所有的物理内存被映射完。  
 ! (每个页表可映射 4M)

× start\_mem = 可用内存的起始值  
 × end\_mem = 机器所拥有的物理内存值

```
1008 unsigned long paging_init(unsigned long start_mem, unsigned long end_mem)
```

```

1009 {
1010     unsigned long * pg_dir;
1011     unsigned long * pg_table;
1012     unsigned long tmp;
1013     unsigned long address;
1014
1015 /*

```

```

1016 * Physical page 0 is special; it's not touched by Linux since BIOS
1017 * and SMM (for laptops with [34]86/SL chips) may need it. It is read
1018 * and write protected to detect null pointer references in the
1019 * kernel.
1020 */

```

! 第 0 个物理页是特殊的;他没有被 linux 使用,因为 BIOS 和 SMM 可能会需要它  
 ! 在内核它被用来侦察NULL 指针

```

1021 #if 0
        ! 调试状态
1022     memset((void *) 0, 0, PAGE_SIZE);
        ! 把从物理地址 0 开始的 4k 清 0,在非调试时这 4k 是要保留的,因为这里放的是
        ! BIOS的中断向量表，并且笔记本机器的管理软件可能需要它。
1023 #endif

```

```

1024     start_mem = PAGE_ALIGN(start_mem);
        ! 调整内存起始值，按页对齐
1025     address = 0;

```

```

1026     pg_dir = swapper_pg_dir;
! pg_dir 指向 swapper_pg_dir[1024] 的首地址，真正的定义在 head.s 中。
1027     while (address < end_mem) {
1028         tmp = *(pg_dir + 768);      /* at virtual addr 0xC0000000 */
! 在虚拟地址 3G 处
! tmp 指向页目录的 768 项，刚好是虚拟地址 0xC0000000 开始处。
! 怎么计算的呢？
! 计算方法：
! 每一页目录可以映射 4M 的内存，那么第 768 项可以映射的内存开始
! 值便是：4M × 768 = 0xC0000000，所以刚好是虚拟地址 3G 的开始处。
1029     if (!tmp) {
1030         tmp = start_mem | PAGE_TABLE;
1031         *(pg_dir + 768) = tmp;
! 如果页表不存在，则加上标志后，写到 tmp 中，再写入页目录的第 768 项
! PAGE_TABLE=PAGE_PRESENT|PAGE_RW|PAGE_USER|PAGE_ACSESSED
! PAGE_TABLE=存在|可读写|可使用|可存储
1032         start_mem += PAGE_SIZE;
! 可用内存的起始值调整为再加上 4k，即可用内存再减少 4k。
1033     }
1034     *pg_dir = tmp;           /* also map it in at 0x0000000 for init */
! 修改 swapper_pg_dir 第 0 项，为了 init 而映射
1035     pg_dir++;
! 跳过第 0 项，即指向 swapper_pg_dir 第 1 项
1036     pg_table = (unsigned long *) (tmp & PAGE_MASK);
! 取得页表，为什么可以得到的页表呢？
! 计算方法：
! PAGE_MASK=0xFFFFF000，tmp 的高 20 位是页表值，
! 所以相与后便是页表值
1037     for (tmp = 0; tmp < PTRS_PER_PAGE; tmp++, pg_table++) {
1038         if (address < end_mem)
1039             *pg_table = address | PAGE_SHARED;
1040         else
1041             *pg_table = 0;
1042         address += PAGE_SIZE;
! 我们可以计算出 PTRS_PER_PAGE=1024，所以这里的
! 循环中可以映射 4M 物理内存，1024×4K=4M。
1043     }
1044 }
! 到这里时，所有的物理内存便全部被映射完成了。
1045     invalidate();
! 刷新页变换高速缓冲
1046     return start_mem;
! 返回可用内存的起始值。已经被页表映射时修改了，
! 即可用内存划出一些内存用于页表的存放。其执行完后的

```

! 内存情况, 请看图 Mm/memory.c -1

[1047 }](#)

! 物理内存初始化

× start\_low\_mem=可用内存低起始值

× start\_mem=可用内存开始值

× end\_mem=可用内存结束值

```

1049 void mem_init(unsigned long start_low_mem,
1050           unsigned long start_mem, unsigned long end_mem)
1051 {
1052     int codepages = 0;
1053     int reservedpages = 0;
1054     int datapages = 0;
1055     unsigned long tmp;
1056     unsigned short * p;
1057     extern int etext;
1058
1059     cli();
    ! 关中断
1060     end_mem &= PAGE_MASK;
    ! 调整内存按页对齐
1061     high_memory = end_mem;
    ! high_memory=系统所拥有的最大内存值
1062     start_mem += 0x0000000f;
1063     start_mem &= ~0x0000000f;
    ! 调整起始内存值
1064     tmp = MAP_NR(end_mem);
    ! tmp中存储了系统内存按页分, 共可以分多少页
1065     mem_map = (unsigned short *) start_mem;
1066     p = mem_map + tmp;
1067     start_mem = (unsigned long) p;
    ! 在可用内存中留下 tmp×4 个字节空间用来放 mem_map, 同时还调整
    ! 新的可用内存大小。
1068     while (p > mem_map)
1069         *--p = MAP_PAGE_RESERVED;
    ! 为从 mem_map 开始, 大小为 tmp×4
    ! 个内存中的内容赋予MAP_PAGE_RESERVED
1070     start_low_mem = PAGE_ALIGN(start_low_mem);
    ! 调整低起始内存开始值(按页对齐)
1071     start_mem = PAGE_ALIGN(start_mem);
    ! 重新调整起始内存按页对齐
1072     while (start_low_mem < 0xA0000) {
1073         mem_map[MAP_NR(start_low_mem)] = 0;

```

```

1074         start_low_mem += PAGE_SIZE;
    ! 如果, 低起始内存小于 0xA0000, 则设置对应 mem_map 的项为 0
    ! 然后以 4k 为单位增加。
    ! 为什么小于 0xA0000 可以映射呢? 这个请看图引爆点图 2
1075     }
1076     while (start_mem < end_mem) {
1077         mem_map[MAP_NR(start_mem)] = 0;
1078         start_mem += PAGE_SIZE;
    ! 这里为从可用内存起始到可用内存结束, 设置 mem_map=0, 同样
    ! 也是以 4k 为单位。
1079     }
1080 #ifdef CONFIG_SOUND
1081     sound_mem_init();
    ! 如果配置了声卡, 则调用之。这里忽略
1082 #endif
1083     free_page_list = 0;
1084     nr_free_pages = 0;
1085     for (tmp = 0 ; tmp < end_mem ; tmp += PAGE_SIZE) {
1086         if (mem_map[MAP_NR(tmp)]) {
    ! 如果对应的内存映射项不为 0 (即被占用了, 加 1)
1087             if (tmp >= 0xA0000 && tmp < 0x100000)
                    reservedpages++;
    ! 如果内存值 tmp 大于等于 640K 却小于 1M 时, 则 reservedpages 自增一
    ! 为什么 640K 到 1M 时, 要保留呢? 做个可以看图引爆点图 2
    ! 因为这个区域是保留了 BIOS 的例程。
1089         else if (tmp < (unsigned long) &etext)
                    codepages++;
1090         ! etext 是连接程序时, 由 ld 产生, 表示所编译出核心的代码段大小
        ! 请注意核心连接起始地址是 1M
1091         else
                    datapages++;
1092         ! 否则数据段页数自增一
                    continue;
1093     }
1094     *(unsigned long *) tmp = free_page_list;
1095     free_page_list = tmp;
1096     nr_free_pagesnr_free_pages << PAGE_SHIFT;
1100     printk("Memory: %luk/%luk available (%dk kernel code, %dk reserved, %dk data)\n",
1101             tmp >> 10,
1102             end_mem >> 10,

```

```

1103         codepages << (PAGE_SHIFT-10),
1104         reservedpages << (PAGE_SHIFT-10),
1105         datapages << (PAGE_SHIFT-10));
    ! 打印内存信息。
1106 /* test if the WP bit is honoured in supervisor mode */
    ! 测试WP位的管理方式
1107     wp_works_ok = -1;
1108     pg0[0] = PAGE_READONLY;
    ! 设置pg0 的第 0 项
1109     invalidate();
    ! 刷新高速缓冲
1110     __asm__ __volatile__("movb 0,%%al ; movb %%al,0": :"ax", "memory");
    ! 把 0 放入al中，接下来把al送入地址 0 中（即 0xC0000000 处）
1111     pg0[0] = 0;
1112     invalidate();
    ! 刷新高速缓冲
1113     if (wp_works_ok < 0)
1114         wp_works_ok = 0;
    ! 设置wp_works_ok=0
1115     return;
1116 }

```

```

154 /*
155 * This function frees up all page tables of a process when it exits.
156 */
    ! 该函数当一个进程退出时释放所有页表
157 void free_page_tables(struct task_struct * tsk)
158 {
159     int i;
160     unsigned long pg_dir;
161     unsigned long * page_dir;
162
163     if (!tsk)
164         return;
    ! 当任务结构为NULL时，直接退出
165     if (tsk == task[0]) {
        ! 如果tsk等于第 0 个任务，则打印错误信息后死机
166         printk("task[0] (swapper) killed: unable to recover\n");
167         panic("Trying to free up swapper memory space");
168     }
169     pg_dir = tsk->tss.cr3;
    ! 取得tsk的页目录
170     if (!pg_dir || pg_dir == (unsigned long) swapper_pg_dir) {

```

```

171         printk("Trying to free kernel page-directory: not good\n");
172         return;
    ! 如果 pg_dir=null 或者 pg_dir=交换页目录
    ! 则打印出错信息，因为交换页目录用于内核代码，而
    ! 内核代码是不可以被释放的
173     }
174     tsk->tss.cr3 = (unsigned long) swapper_pg_dir;
175     if (tsk == current)
176         __asm__ __volatile__("movl %0,%%cr3": :"a" (tsk->tss.cr3));
    ! 如果 tsk 等于当前任务，则把交换页目录
    ! 放入cr3 中
177     if (mem_map[MAP_NR(pg_dir)] > 1) {
        ! 如果页目录对应的内存映射大于 1
178         free_page(pg_dir);
        ! 则释放之！定义于mm/swap.c
179         return;
180     }
181     page_dir = (unsigned long *) pg_dir;
182     for (i = 0 ; i < PTRS_PER_PAGE ; i++,page_dir++)
183         free_one_table(page_dir);
184     free_page(pg_dir);
185     invalidate();
186 }
```

## N

### Net/unix/sock.c (部分代码)

```

    ! 根据 level，打印提示信息
    ! 于 printk 类似
98 static void
99 dprintf(int level, char *fmt, ...)
100 {
101     va_list args;
102     char *buff;
103     extern int vsprintf(char * buf, const char * fmt, va_list args);
```

```

104
105     if (level != unix_debug) return;
106
107     buff = (char *) kmalloc(256, GFP_KERNEL);
108     if (buff != NULL) {
109         va_start(args, fmt);
110         vsprintf(buff, fmt, args);
111         va_end(args);
112         printk(buff);
113         kfree(buff);
114     }
115 }
116
928 void
    ! 初始化 unix 的通讯协议
    × pro=网络协议结构指针
929 unix_proto_init(struct ddi_proto *pro)
930 {
931     struct unix_proto_data *upd;
932
933     dprintf(1, "%s: init: initializing...\n", pro->name);
        ! 打印提示信息，dprintf 于 printk 功能差不多，前者会自己申请空间
        ! 来存放要打印的字符串，而不会占用系统为 printk 定义的缓冲区。
        ! 这里不在对其解释。有兴趣的朋友大家可用参考本文件的第 98 行到 116 行
934     if (register_chrdev(AF_UNIX_MAJOR, "af_unix", &unix_fops) < 0) {
935         printk("'%s: cannot register major device %d!\n",
936                         pro->name, AF_UNIX_MAJOR);
937         return;
        ! 注册字符设备，定义于Fs/devices.c
938     }
939
940     /* Tell SOCKET that we are alive... */
        ! 告诉SOCKET我们还在运行
941     (void) sock_register(unix_proto_ops.family, &unix_proto_ops);
        ! 定义于文件Net/socket.c
942
943     for(upd = unix_datas; upd <= last_unix_data; ++upd) {
944         upd->refcnt = 0;
945     }
946 }
947

```

## Net/space.c (部分代码)

```

38
39 struct ddi_proto protocols[] = {
40 #ifdef CONFIG_UNIX
41   { "UNIX",   unix_proto_init },
42 #endif
43 #ifdef CONFIG_IPX
44   { "IPX",    ipx_proto_init },
45 #endif
46 #ifdef CONFIG_AX25
47   { "AX.25",  ax25_proto_init },
48 #endif
49 #ifdef CONFIG_INET
50   { "INET",   inet_proto_init },
51 #endif
52   { NULL,    NULL      }
53 };
54

```

## Net/ddi.c (部分代码)

```

62
63 /*
64 * This is the function that is called by a kernel routine during
65 * system startup. Its purpose is to walk through the "devices"
66 * table (defined above), and to call all modules defined in it.
67 */
    ! 该函数在核心启动时被调用，它的目的是初始化设备表（定义在上面）
    ! 并且调用定义在里面的所有模块
68 void
69 ddi_init(void)
70 {
71   struct ddi_proto *pro;
72   struct ddi_device *dev;
73
74   PRINTK(("DDI: Starting up!\n"));

```

```

    ! 打印提示消息
75
76 /* First off, kick all configured protocols. */
    ! 马上，配置所有的协议
77 pro = protocols;
    ! protocols 定义于 Net/space.c
78 while (pro->name != NULL) {
79     (*pro->init)(pro);
    ! 1.0 核心支持的协议有如下种
    ! unix_proto_init
    ! ipx_proto_init
    ! ax25_proto_init
    ! inet_proto_init
    ! 这里我 unix 的通讯协议，定义于 Net/unix/sock.c
80     pro++;
81 }
82
83 /* Done. Now kick all configured device drivers. */
    ! 好了。现在配置所有的协议
84 dev = devices;
85 while (dev->title != NULL) {
86     (*dev->init)(dev);
87     dev++;
88 }
89
90 /* We're all done... */
    ! 我们已经做完所有的了。
91 }
92

```

## Net/socket.c (部分代码)

```

1044 /*
1045 * This function is called by a protocol handler that wants to
1046 * advertise its address family, and have it linked into the
1047 * SOCKET module.
1048 */
    ! 该函数通过一个通讯协议的处理函数调用以知道它的地址协议族,
    ! 并且它连接进SOCKET模块
1049 int

```

```

    ! 把协议注册到 ops 中
    ! family=协议号
    ! ops=协议指针
1050 sock_register(int family, struct proto_ops *ops)
1051 {
1052     int i;
1053
1054     cli();
    ! 关中断
1055     for(i = 0; i < NPROTO; i++) {
1056         if (pops[i] != NULL) continue;
1057         pops[i] = ops;
1058         pops[i]->family = family;
    ! 把协议写入
1059         sti();
    ! 开中断
1060         DPRINTF((net_debug, "NET: Installed protocol %d in slot %d (0x%X)\n",
1061                         family, i, (long)ops));
    ! 打印提示信息（DPRINTF为宏定义，打印提示信息）
1062         return(i);
1063     }
1064     sti();
1065     return(-ENOMEM);
1066 }
1069 void
    ! socket的初始化
1070 sock_init(void)
1071 {
1072     struct socket *sock;
1073     int i;
1074
1075     /* Set up our SOCKET VFS major device. */
1076     if (register_chrdev(SOCKET_MAJOR, "socket", &net_fops) < 0) {
1077         printk("NET: cannot register major device %d!\n", SOCKET_MAJOR);
1078         return;
    ! 注册网络设备，不成功打印提示消息后返回
    ! 定义于Fs/devices.c
1079     }
1080
1081     /* Release all sockets. */
    ! 释放所有scoket
1082     for (sock = sockets; sock <= last_socket; ++sock) sock->state = SS_FREE;
1083
1084     /* Initialize all address (protocol) families. */

```

```

! 初始化所有地址的协议族
! 1.0 核心支持 16 种协议
1085  for (i = 0; i < NPROTO; ++i) pops[i] = NULL;
1086
1087  /* Initialize the DDI module. */
    ! 初始化设备驱动接口模块, 定义于Net/ddi.c
1088  ddi_init();
1089
1090  /* Initialize the ARP module. */
1091 #if 0
1092  arp_init();
1093 #endif
1094 }
1095

```

## Z

### **zBoot/head.S**

#### 概述

关中断, 把从数据段尾后面的数据全部清零。接着把压缩的核心, 从 4k 处开始解压缩。并且从 1M 处开始放置, 成功后, 便跳转到 1M 处执行!

#### 代码分析

```

1 /*
2  *  linux/boot/head.S
3  *
4  *  Copyright (C) 1991, 1992, 1993  Linus Torvalds
5  */
6
7 /*
8  *  head.S contains the 32-bit startup code.
9  *
! head.S 包含了 32 位启动代码
10 * NOTE!!! Startup happens at absolute address 0x00001000, which is also where

```

```

11 * the page directory will exist. The startup code will be overwritten by
12 * the page directory.
13 *
! 请注意：引导发生在绝对地址 0x00001000 处，这也会将是页目录的放置处，
! 所以这里的代码会被页目录覆盖掉。
14 * Page 0 is deliberately kept safe, since System Management Mode code in
15 * laptops may need to access the BIOS data stored there. This is also
16 * useful for future device drivers that either access the BIOS via VM86
17 * mode.
18 */
! 页表 0 必须保证安全，因为笔记本的系统模式管理代码可能需要访问 BIOS 中的
! 数据段，这也用于未来的设备驱动，VM86 模式也需要访问它！

19 .text
20
21 #include <linux/segment.h>
22
23 startup_32:
24     cld
! 清方向位
25     cli
! 关中断
26     movl $(KERNEL_DS),%eax
! eax = 0x18
27     mov %ax,%ds
28     mov %ax,%es
29     mov %ax,%fs
30     mov %ax,%gs
! ds=es=fs=gs=0x18
31     lss _stack_start,%esp
! 设置堆栈,
32     xorl %eax,%eax
! eax = 0
33 1:   incl %eax      # check that A20 really IS enabled
34     movl %eax,0x000000    # loop forever if it isn't
35     cmpl %eax,0x100000
36     je 1b
! 检查是否真的进入了保护模式，采用的
! 办法是，在物理地址 0 处写入一个数，
! 再从物理地址 1M 处读，如果读出的数
! 和写入的数相同则表示并没有进入保护
! 模式，直到不相等！
37 /*
38 * Initialize eflags. Some BIOS's leave bits like NT set. This would
39 * confuse the debugger if this code is traced.

```

```

40    * XXX - best to initialize before switching to protected mode.
41    */
    ! 初始化标志寄存器 eflags。一些 BIOS 的离开位需要它，象 NT 位！假如调试这些代码的话它会搞乱调试器。
    ! XXX-切换保护模式前的最好的初始化
42    pushl $0
    ! 0 入栈
43    popfl
    ! 弹出后送入eflags，则eflags=0
44 /*
45    * Clear BSS
46    */
47    xorl %eax,%eax
48    movl $__edata,%edi
49    movl $__end,%ecx
50    subl %edi,%ecx
51    cld
52    rep
53    stosb
54 /*
    ! 上面 47 到 53 行的注释，请看boot/head.S中注释！
55    * Do the decompression, and jump to the new kernel..
56    */
57    call _decompress_kernel
    ! 调用 decompress_kernel 来解压缩代码，并从物理地址 1M 处开始放置，这里不
    ! 注释 decompress_kerne，它在 zBoot 的 misc.c 文件中，有兴趣的读者可自己
    ! 看看！（因为与核心没有关系）
58    ljmp $(KERNEL_CS), $0x100000
    ! 跳转到物理地址 1M 处执行，即 boot/head.s 中去执行
    这时的内存布局请看图 Z-1，在执行完上面的第 58 行代码后，便跳转到物理地址 1M 处
    执行，也就是跳到了 boot/head.s 中去了。我们现在就去 boot/head.s 看看！Let's go!

```

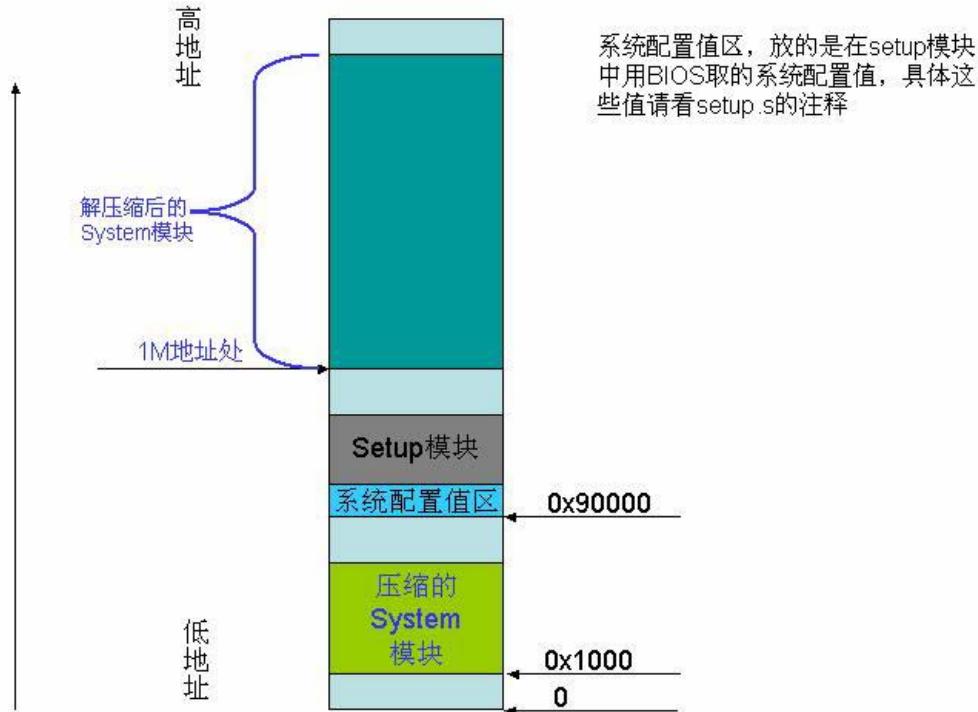


图 Z-1

## 核心游记总结（1.0 核心）

该节从 PC 点火启动开始，直到核心正常运作起来做一个简单的总结。该总结完全是按照核心启动的过程来说的。本来应该放在 Init/main.c 后比较合适（个人感觉）。

### 1. 系统自检

当我们按下计算机的 Power 按钮后，便会送一个电信号给主板，主板在收到这个信号后，接下来会将此电信号传给供电系统，于是供电系统开始工作，为整个系统供电，并送出一个电信号给 BIOS，通知 BIOS 供电系统已经准备完毕。随后 BIOS 启动一个程序，进行主机自检，主机自检的主要工作是确保系统的每一个部分都得到了电源支持，内存储器、主板上的其它芯片、键盘、鼠标、磁盘控制器及一些 I/O 端口正常可用，此后，自检程序将控制权还给 BIOS。接下来 BIOS 读取 BIOS 中的相关设置，得到引导驱动器的顺序，然后依次检查，直到找到可以用来引导的驱动器（或说可以用来引导的磁盘，包括软盘、硬盘、光盘等），然后调用这个驱动器上磁盘的引导扇区中的引导块程序进行引导。

对于 Intel 的 X86CPU 来说，会把代码段寄存器的值为全 1，而指令计数器的值为全部置为 0，即 CS=FFFF、IP=0000。因为这个时候还在实模式方式下，所以这两个寄存器组合而成的地址便是 FFFF0H。由于 FFFF0H 已经到了基本内存的高地址顶端，所以，

FFFFOH 处的指令一般总是一个 JMP 指令，以便 CPU 能够跳到比较低的地址去执行那里的代码，这个地址通常是 ROM BIOS 的入口地址。接着，ROM BIOS 进行开机自检，如检查内存，键盘等。在自检过程中，ROM BIOS 会在上位内存(UMB, UPPERMEMORY BLOCK) 中进行扫描，看看是否存在合法的设备控制卡 ROM BIOS (如:SCSI 卡上的 ROM)，如果有，就执行其中的一些初始化代码。最后，ROM BIOS 读取磁盘上的第一个扇区并将这个扇区的内存装入内存。

## 2. 系统引导

Boot/bootsect.s 便是在上面第一步中被装入内存的引导扇区，该部分代码是用 8086 汇编语言编写完成的。接下来便会执行该块代码（这里称这块代码名为 bootsect.o），bootsect.o 把自己从绝对地址 0x7C00 处开始的 512 个字节大小的内容移动到绝对地址 0x90000 的开始处。bootsect.o 移动完后便接着读入 Boot/setup.s 代码所编译成的模块（这里称这块代码名为 setup.o），共 4 个扇区，也即 2k 的大小，放置地址从绝对地址 0x90200 处开始。setup.o 加载完成后，便会加载 System 模块（该模块是真正的核心，同时也请注意该模块已经是压缩过的了，后面会做解压），放置地址在 0x10000 处。紧接着在把 System 模块移动到 0x1000 地址处（该移动动作是由 setup.o 去完成的）。共 508k 字节的大小，移动完后加载 idt 表和 gdt 表。接下来通过选通 A20 地址线来进入保护模式。真正进入保护模式后，初始化 8259A 中断芯片。

在下来会跳转到 0x1000 处执行，其实这里的代码就是 zBoot/head.s 编译所完成的代码，它完成内核的解压缩工作，它初始化寄存器并调用 decompress\_kernel()，它们依次是由 zBoot/inflate.c、zBoot/unzip.c 和 zBoot/misc.c 组成（这几个\*.C 文件没有作注释，因为它和核心关联不大，我认为）。被解压的数据存放到了地址 0x10000 处(1 兆)，这可能也是为什么 Linux 不能运行于少于 2 兆内存的主要原因吧。

## 3. 核心初始化

在第 2 步中完成核心解压缩和放置后，便会跳转到 start\_kernel () 中去执行了。下面我们按顺序来看看做了那些工作。

- 设置 lcall7，让 Linux 支持 Intel 二进制兼容规范标准的版本 2 (iBCS2)，这样的话 Linux 便可以支持类 Unix 及 Solaris 中的可执行文件了，不过在的 1.0 核心中还未真正支持 iBCS2。
- 读取根设备，驱动器，屏幕，aux 设备信息，机器拥有物理内存置（请注意这些信息是在 Boot/setup.s 中通过 BIOS 取得的）设置虚拟盘的大小，并把 COMMND\_LINE 拷贝到 command\_line 中
- 根据可用内存重新设置页表
- 初始化硬件中断向量初始化
- 初始化 IRQ 通道
- 初始化任务 0，即调度程序
- 解析命令行参数
- 为 profiling 分配一些数据缓冲区
- 检查系统定义的分页信息是否正确
- 初始化字符设备
- 初始化块设备
- 开中断
- 延迟校准

- 初始化网络设备
- 初始化 SCSI 设备
- i 节点 hash 表的初始化
- 文件描述符表数组的初始化
- 物理内存的初始化
- 高速缓冲初始化
- 设置开机启动时间
- 软驱初始化
- 套接字初始化
- IPC（进程间通讯）初始化
- 开中断
- 检查异常 16 是否工作正常。这块代码是有害的：它禁止高的 8 号中断以禁止 irq13 号中断的产生。假如没有 16 位异常的发生话，这块代码会导致机器死锁，8 号中断会在下一次的时间滴答时重新开启。以使得 irq13 中断又可以发生，但是异常 16 必须先发生如果，系统的 CPU 不支持协处理器并且在配置核心时也没有打开软件模拟协处理器的选项，则打印错误后，死机！
- 打印 linux\_banner
- 切换到用户模式
- 任务 0，是所有进程的父进程。它是个“idle”任务：它不能去睡眠，但是它可以做些标准的事情比如计算空闲页面或者为分页程序执行一个合理的 LRU 算法：任何有益处的事情，但是它不能获取真正进程的时间。当前的任务 0 只执行一个无限的 idle 循环
- 任务 0 调用 fork，创建 init 进程（下面看看在 init 中都做了什么）
  - i. 以读写方式打开/dev/tty1
  - ii. 创建标准输出，标准错误
  - iii. 执行/etc/init，成功后则不会在做下面的操作了
  - iv. 执行/bin/init，成功后则不会在做下面的操作了
  - v. 执行/sbin/init，成功后则不会在做下面的操作了
  - vi. 再次调用 fork，创建新的进程，关闭标准输入，并以只读方式打开/etc/rc，执行/bin/sh，把/etc/rc 中的内容作为其输入。而父进程则等待这个新创建的进程结束。
  - vii. 如果到这一步，则创建一个死循环，重复上述的过程（创建新进程→等待其结束）。

#### 4. 进程的运行模式

对于 Intel X86 CPU 共提供 4 种运行程序的级别，从 Ring0 到 Ring3。但是在 Linux 核心中只用到了 2 级（即 Ring0 和 Ring3）。分别被称为核心态、用户态。在正常情况下进程只能在用户态下运行，要想进入核心态运行，必须调用 INT 中断。也就是调用 0x80 号中断便可以进入核心态运行。进入核心态后，根据寄存器 EAX 值判断来得到不同系统调用函数，从而调用之。完成后，通过 IRET 指令返回用户态。

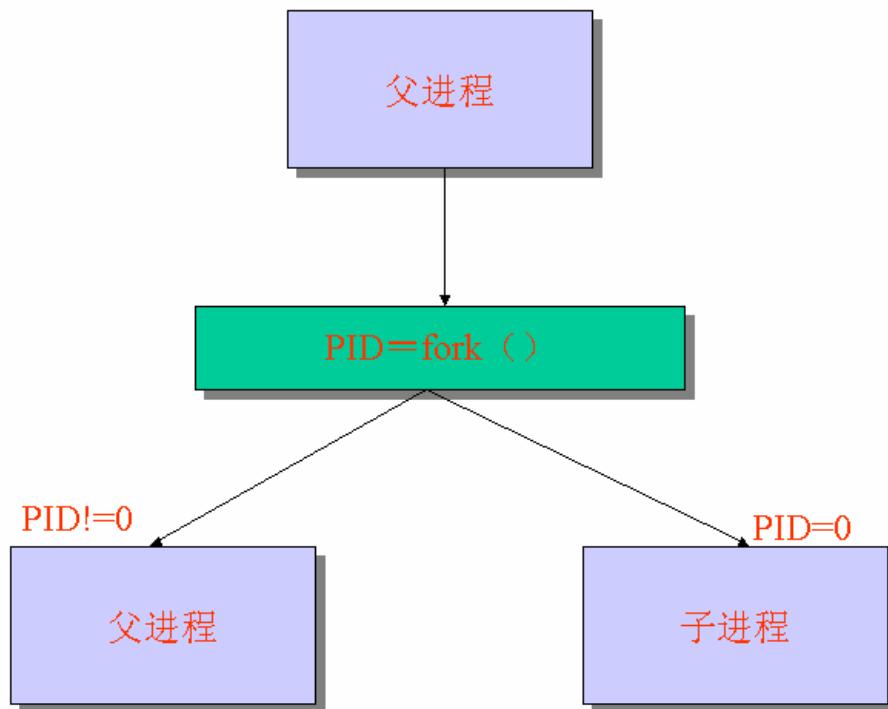
进程在核心态和用户态运行时，各自所用的堆栈是分开的。

#### 5. 进程的创建

Linux 中创建一个新的进程是通过 fork() 函数。执行过程描述如下。

- 取得一个空闲内存页面来保存 task\_struct
- 找到一个空闲的进程槽 (find\_empty\_process())
- 为内存堆栈页 kernel\_stack\_page 取得另一个空闲的内存页面
- 将父辈的 LDT 拷贝到子进程
- 复制父进程的 mmap 信息

通过上面的步骤后，我们便可以得到同一个进程的两个拷贝了（在内存中）。分别被称为父进程及子进程，既然它们是一样的。我们如何区分它们呢？答案是通过它们的返回值 (fork() 的返回值)。对于父进程返回值是子进程的进程标识号（也称 PID），而对于子进程返回值则是 0。我们用下图描述之（这样会更加清晰）



## 6. 执行新进程

在第 5 步中，我们用 fork() 创建了一个新的进程后，所执行的还是自己。要执行新的程序的话，就必须用另外的函数。来替换掉子进程的二进制镜像。能够完成这个功能的函数就是 execve 系列函数。1.0 核心支持 3 种可执行文件格式（分别是 a.out, elf, Coff）。同时还支持对这些格式的文件实现按需加载，即不需要把整个二进制文件都一股脑加载进内存。

## 7. 核心对进程的调度

上面完成了一个新进程的执行后，我们的系统便运作起来了。下面的工作是核心采用分时的办法对各个不同的进程进行调度。核心中设定每 10ms 发出一次时钟中断。在该中断中检查当前系统中所有进程的状态以确认下一步要做的工作。

## 8. 结束

## 第三部分 其他话题（Advanced part）

该部分包含了大家都感兴趣的话题，我的测试环境为 Vmware 5.0 + RedHat 9.0，还请注意的是核心我并没有升级到最新的 2.6，还是用系统自带的 2.4 核心。

## A1.模块的编写

这里我们只是简单的介绍模块的编程，目的是为了在下面我们能够编写我们所需要的模块。我们的测试环境是 WMware 5.0 + Redhat 9.0。另外还需要的是，读者必须具有超级用户的权限。（注：如果读者想知道更多的关于模块或者驱动的编写，大家可以搜索互联网或者看看 LDD2 “现在已经有 LDD3 了”）

### A1-1 模块代码及分析

#### A1.1-1/ helloworld.c

---

```
#ifndef __KERNEL__
#define __KERNEL__
#endif

#ifndef MODULE
#define MODULE
#endif

#define NULL 0

#include <linux/kernel.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");           //为了在插入模块时，避免有警告提示

int init_module(void)           //模块注册函数
{
printk("Hello everybody.\nI am a module.\n");
return 0;
}

void cleanup_module(void)        //模块注销函数
{
printk("Goodbye,everybody.\n");
}
```

---

#### A1.1-1/Makefile

```
GCC=gcc
KERNELDIR=/usr/src/linux
OBJS=helloworld.o
```

.c.o:

```
$(GCC) -D__KERNEL__ -I$(KERNELDIR) -c $^ -o $@
```

\$(OBJS):

```
insert:$(OBJS)          #这里加 insert, 是为了你可以直接运行 make insert
    /sbin/insmod $(OBJS)  #这样的话, 你也就不要在执行 insmod 命令了
                           #不过, 你完全可以执行 insmod 命令, 我们后面会
                           #介绍
```

remove:

```
/sbin/rmmod helloworld      #与 insert 类似
```

clean:

```
rm -f *.o core
```

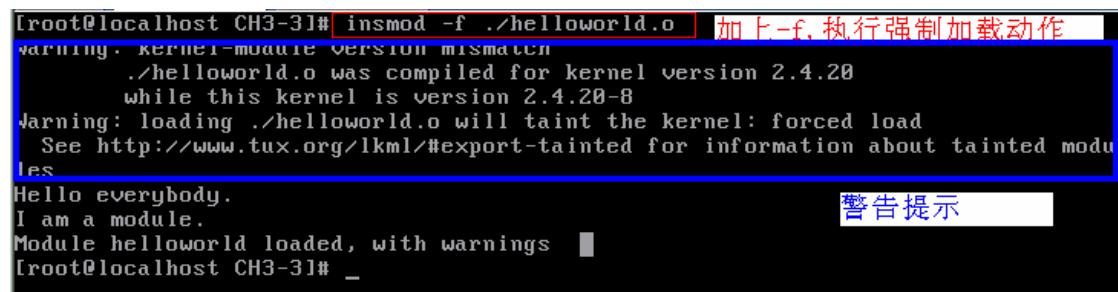
---

以上的代码都有了，我们只要执行一下 make，便可以编译出一个模块来，不过这里有个问题各位读者一定要注意，否则编译出的模块加载不会成功。就象下图（图 A1-1-1）描述的一样。

```
[root@localhost CH3-3]# insmod ./helloworld.o
./helloworld.o: kernel-module version mismatch
    ./helloworld.o was compiled for kernel version 2.4.20
        while this kernel is version 2.4.20-8.
[root@localhost CH3-3]# _
```

图 A1-1-1

通过图 A1-1-1，我们可以看出我们编译出的核心并没有加载成功，根据提示信息我们可以知道是模块的版本号与核心的版本不同。也许有的读者可能会这样做，强制模块加载，但总不会都成功。还有即使你强制加载成功，可能会对核心造成了一些不稳定的因素。不过暂时看不出来而已。（注：强制加载可能不会在所有读者的机器都成功，所以这里给出在我的系统上强制加载成功的截图，请看下图（图 A1-1-2））



```
[root@localhost CH3-3]# insmod -f ./helloworld.o 加上-f, 执行强制加载动作
warning: kernel-module version mismatch
    ./helloworld.o was compiled for kernel version 2.4.20
        while this kernel is version 2.4.20-8
warning: loading ./helloworld.o will taint the kernel: forced load
See http://www.tux.org/lkml/#export-tainted for information about tainted modules
Hello everybody.
I am a module.
Module helloworld loaded, with warnings
[root@localhost CH3-3]# _
```

图 A1-1-2

针对上面的问题怎么解决呢？下面我说说我常用的两种。

第一种，也是比较简单的一种，直接拷贝你的系统中核心源代码中的 /usr/src/linux/include/linux/version.h (假设你的核心在 /usr/src/linux 下) 到 /usr/include/linux 下，覆盖掉原来的 version.h 文件即可。原因是你的系统的核心的版本是用的该文件中的版本号，这样做后，我们编译出的模块版本号就与核心一样了。也就意味着可以加载了。

第二种，是比较麻烦的办法，不过对于初学者我认为用这种办法可以学到更多的知识和经验。那就是重新编译一个新的核心，用该核心来加载我们的模块。(注：关于如何编译核心请搜索互联网) 这里我们打开 /usr/src/linux/Makefile 文件后，去掉 EXTRAVERSION=-8custom 这行(见下图红线框中所标)，这样我们编译出的核心版本便是 2.4.20 了，也就是与 /usr/include/linux/version.h 相同了。

```
VERSION = 2
PATCHLEVEL = 4
SUBLEVEL = 20
EXTRAVERSION = -8custom

KERNELRELEASE=$(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$EXTRAVERSION

ARCH := $(shell uname -m | sed -e s/i.86/i386/ -e s/sun4u/sparc64/ -e s/arm.*/arm \
m/ -e s/sa110/arm/)

KERNELPATH=kernel-$(shell echo $(KERNELRELEASE) | sed -e "s/-//g")

CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \
else if [ -x /bin/bash ]; then echo /bin/bash; \
else echo sh; fi ; fi)
TOPDIR := $(shell /bin/pwd)

HPATH = $(TOPDIR)/include
FINDHPATH = $(HPATH)/asm $(HPATH)/linux $(HPATH)/scsi $(HPATH)/net $(HPATH) \
)/math-emu

HOSTCC = gcc
HOSTCFLAGS = -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer

CROSS_COMPILE =
"/usr/src/linux/Makefile" 617L, 20471C
```

1.1

Top

图 A1-1-3

## A1-2 模块的加载、注销及查看

到这里，我们就会得到了一个可以被加载的模块了。

加载模块我们 Linux 提供的命令是“insmod”，注销模块用的命令是“rmmod”。我们先执行 make 命令，以得到 helloworld.o 文件，请看图 A1-2-1

```
[root@localhost CH3-3]# ls -l
total 8
-rw-rw-rw-    1 ftp      ftp          324 Oct  1 22:19 helloworld.c
-rw-rw-rw-    1 ftp      ftp          228 Oct  1 22:26 Makefile
[root@localhost CH3-3]# make
gcc -D__KERNEL__ -I/usr/src/linux -c helloworld.c -o helloworld.o
[root@localhost CH3-3]# ls -l
total 12
-rw-rw-rw-    1 ftp      ftp          324 Oct  1 22:19 helloworld.c
-rw-r--r--    1 root     root        1112 Oct  1 23:12 helloworld.o
-rw-rw-rw-    1 ftp      ftp          228 Oct  1 22:26 Makefile
[root@localhost CH3-3]# _
```

编译得到的helloworld.o

图 A1-2-1

接下来我们就可以加载该模块了，请看图 A1-2-2。

```
[root@localhost CH3-3]# ls -l
total 12
-rw-rw-rw-    1 ftp      ftp          324 Oct  1 22:19 helloworld.c
-rw-r--r--    1 root     root        1112 Oct  1 23:12 helloworld.o
-rw-rw-rw-    1 ftp      ftp          228 Oct  1 22:26 Makefile
[root@localhost CH3-3]# insmod ./helloworld.o
Hello everybody.
I am a module.
[root@localhost CH3-3]# _
```

图 A1-2-2

我们可以看到模块中的 printk 的内容输出确实已经打印在了屏幕上，也许你仍然会抱有疑惑。我们的模块真的被加载了吗？没关系会让你确信的。请在命令行提示下输入如下命令“cat /proc/modules |more”或者“cat /proc/modules|grep helloworld”，请看图 A1-2-3

```
[root@localhost CH3-3]# cat /proc/modules |more
helloworld          796  0 (unused)           请看，这就是我们的模块
ide-cd              35708 0 (autoclean)
cdrom               33728 0 (autoclean) [ide-cd]
parport_pc          19076 1 (autoclean)
lp                  8996  0 (autoclean)
parport              37056 1 (autoclean) [parport_pc lp]
nls_iso8859-1       3516  0 (autoclean)
nls_cp437            5116  0 (autoclean)
vfat                 13004 0 (autoclean)
fat                  38808 0 (autoclean) [vfat]
usb-storage          69332 0
autofs              13268 0 (autoclean) (unused)
pcnet32              18240 1
mii                  3976  0 [pcnet32]
keybdev              2944  0 (unused)
mousedev             5492  0
hid                  22148 0 (unused)
input                 5856  0 [keybdev mousedev hid]
usb-uhci              26348 0 (unused)
usbcore              78784 1 [usb-storage hid usb-uhci]
ext3                  70784 2 [ext3]
jbd                  51892 2 [ext3]
BusLogic             100796 3
--More--
```

图 A1-2-3

最后，我们注销刚才的模块，请在命令行提示符下输入“`rmmod helloworld`”（注：是 `helloworld` 而非 `helloworld.o`），请看图 A1-2-4

```
[root@localhost CH3-3]# rmmod helloworld
Goodbye,everybody.
[root@localhost CH3-3]# _
```

图 A1-2-4

根据屏幕的提示信息，我们可以知道模块确实已经注销了，如何确认呢？与上面类似在命令行提示符下输入“`cat /proc/modules | grep helloworld`”，屏幕输出见图 A1-2-5。

```
[root@localhost CH3-3]# rmmod helloworld
Goodbye,everybody.
[root@localhost CH3-3]# cat /proc/modules |grep helloworld
[root@localhost CH3-3]# _
```

图 A1-2-5

看了提示，可以非常大胆的说：“是的，模块已被注销了”。

## A2.系统调用的添加

### A2-1 静态添加系统调用

所谓的静态静态添加系统调用，是指我们直接通过修改核心的源代码而达到的。只要我们知道 Linux 下系统调用实现的框架，添加（当然也可以修改）系统调用将会是件非常简单的事情。

该方法的缺点还是有的：

1. 修改好源代码后需要重新编译核心，这是个非常长和容易发生错误的过程。
2. 对于你修改及编译好后所得到的核心，你所做的添加（修改）是静态的，无法在运行时动态改变（所以也就有了下面的动态方法）

#### A2-1-1 讨论 Linux 系统调用的体系

在 Linux 的核心中，`0x80` 号中断是所有系统调用的入口（当然你也可以修改，因为我们有源代码吗 :），不过你改了之后只能自己玩玩，要不然人家 `0x80` 号中断的程序怎么执行呢？）。但是还是有办法（可能还有其他办法）。办法是在你看了下面的“动态添加系统调用”后就知道，这个就留给读者考虑了。

用 `0x80` 中断号功能作为所有的系统调用入口点，是系统编写者定义的（也可以说是 Linus 定义的）。下面我们看一下其设置的代码（取之 2.4 核心，我们只看 x86）

定义于 Arch/i386/kernel/traps.c (很简单, 就一个函数调用)

set\_system\_gate(SYSCALL\_VECTOR,&system\_call);!

设置 0x80 号中断

SYSCALL\_VECTOR 默认是 0x80 (你可以修改)

system\_call 定义在 Arch\i386\kernel\entry.S

set\_system\_gate 定义在 Arch/i386/kernel/traps.c, 具体的代码分析这里就不做介绍了。大致的功能是把 system\_call 的地址 (当然还有其他内容, 比如类型值及特权级) 设置到 IDT (中断描述符表) 的第 0x80 项中 (请注意每项是 8 个字节, 在基础有所介绍)。

当用了 set\_system\_gate 设置好中断号, 并且已经开中断。接下来我们就可以用编程的方式来调用该中断号。调用中断的汇编指令是 “int”。

### A2-1-1/ hello.c

! 该程序中使用 sys\_write 这个系统调用来输出要打印的字符。

! 同时请注意在该程序中我们也用了 strlen 函数, 它是 C 库中定义的标准函数

! 不过, 这里我们只需关注代码中的汇编代码即可。

```
#include <stdio.h>
#include <unistd.h>
int
main()
{
    int value = -1;
    char *lpBuffer = "Hello everybody.\n";
    unsigned long sys_num = 4;
    int iLen = strlen(lpBuffer);
    __asm__("int $0x80"
            :"=a"(value)           //输出值 (即 printf 执行后的返回值)
            :"0"((long)(sys_num)), //eax=sys_num=4,sys_write 的系统调用号
            "b"(1),               //参数一: 文件描述符(stdout)
            "c"(lpBuffer),         //参数二: 要显示的字符串
            "d"(iLen));          //参数三: 字符串长度
    return value;
}
```

### A2-1-1/Makefile

GCC=gcc

OBJS=hello.o

.c.o:

\$(GCC) -c \$<

```
all:${OBJS}
$(GCC) ${OBJS} -o hello
clean:
rm -f *.o core
clobber:clean
rm -f hello
```

这里的代码编译后，我们便可以执行了。其输出结果就如我们调用标准 C 库中的 printf 函数一样。请看下图

```
[root@localhost CH3-5]# ls -l
total 8
-rw-rw-rw- 1 ftp      ftp          292 Nov  6 12:37 hello.c
-rw-rw-rw- 1 ftp      ftp          142 Nov  6 12:41 Makefile
[root@localhost CH3-5]# make
gcc -c hello.c
gcc hello.o -o hello
[root@localhost CH3-5]# ls -l
total 24
-rwxr-xr-x  1 root    root        11598 Nov  6 12:48 hello
-rw-rw-rw-  1 ftp      ftp          292 Nov  6 12:37 hello.c
-rw-r--r--  1 root    root        856 Nov  6 12:48 hello.o
-rw-rw-rw-  1 ftp      ftp          142 Nov  6 12:41 Makefile
[root@localhost CH3-5]# ./hello
Hello everybody.
[root@localhost CH3-5]# _
```

我们更关心的是系统调用的实现机制。下面请跟我来看吧。

1. `__asm__("int $0x80"`
2. `:"=a"(value)` //输出值（即 sys\_write 执行后的返回值）
3. `:"0"((long)(sys_num)),` //eax=sys\_num=4,sys\_write 的系统调用号
4. `"b"(1),` //参数一：文件描述符(stdout)
5. `"c"(lpBuffer),` //参数二：要显示的字符串
6. `"d"(iLen);` //参数三：字符串长度

第 1 句代码用于执行 0x80 号中断。当程序执行到这句时，CPU 会从用户态切换进核心态（也就是我们通常说的 ring0 级），并且同时会把 ss, esp, eflags, cs, eip 按顺序入栈。

从第 2 句到第 6 句代码，用于把系统调用号及参数一到到参数三设置到对应的寄存器中。

`eax=sys_num` (系统调用号)

`ebx=1` (参数一，标准输出)

`ecx=lpBuffer` 的值 (参数二，要显示的字符串)

`edx=iLen` (参数三，字符串长度)

接下来执行 system\_call 函数（所以我们也说该函数是所有系统调用的入口点函数）。于是我们接着看 system\_call 函数。

```
ENTRY(system_call)
1. pushl %eax           # save orig_eax
2. SAVE_ALL
3. .....
4. cmpl $(NR_syscalls),%eax
```

```

5. jae badsys
6. call *SYMBOL_NAME(sys_call_table)(,%eax,4)

7. .....
8. restore_all:
9. RESTORE_ALL

```

第 1 句代码首先把 eax 值入栈，我们可以知道该 eax 中保存的就是我们从用户态传入的系统调用号（从代码的注释也可以看出：）。

第 2 句代码是个宏定义，定义如下：

```
#define SAVE_ALL \
    cld; \
    pushl %es; \
    pushl %ds; \
    pushl %eax; \
    pushl %ebp; \
    pushl %edi; \
    pushl %esi; \
    pushl %edx; \
    pushl %ecx; \
    pushl %ebx; \
    movl $(__KERNEL_DS),%edx; \
    movl %edx,%ds; \
    movl %edx,%es;
```

从宏的代码中，可以得知它把 CPU 中的一些寄存器值入栈（为了从核心态返回时还能回到进入前（用户态）的状态），同时还把核心态的数据段值写入 ds，es（这样的话，我们就可能以访问核心态的数据段了，代码段在执行 int 指令时已经由 CPU 自动设置了）。

第 4, 5 句代码是在测试我们传入的系统调用号是否超过了当前系统所支持的最大系统调用数（对于 2.4 核心，支持的最大系统调用数是 260 个，当然你可以修改）。

第 6 句代码用传入的系统调用号，查表获得对应的系统调用函数地址并 call 之。该表定义如下（定义在 Arch/i386/kernel/entry.S）：

```
.data
ENTRY(sys_call_table)
.long SYMBOL_NAME(sys_ni_syscall)    /* 0 - old "setup()" system call*/
.long SYMBOL_NAME(sys_exit)
.long SYMBOL_NAME(sys_fork)
.long SYMBOL_NAME(sys_read)
.long SYMBOL_NAME(sys_write)
.long SYMBOL_NAME(sys_open)          /* 5 */

.....
```

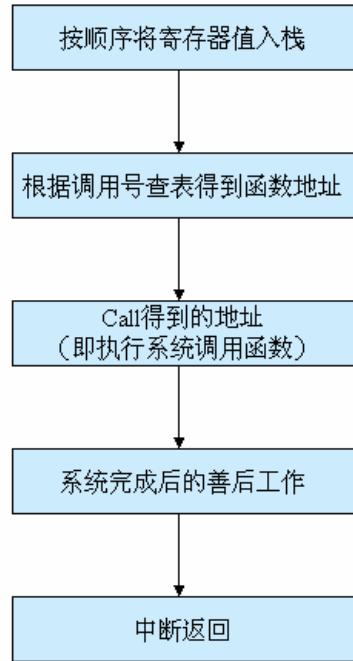
我们给出的例子程序调用号是 4，所以根据上表我们可以知道对应的系统调用函数是 sys\_write。这样我们就进入了真正的系统调用处理函数了。（关于 sys\_write 实现这里不做介绍）

从第 7 句向后的代码，便是系统调用执行完成后的善后处理工作。这里我们给出上面的

`SAVE_ALL` 宏的相反操作。即 `RESTORE_ALL`, 代码如下:

```
#define RESTORE_ALL      \
    popl %ebx;    \
    popl %ecx;    \
    popl %edx;    \
    popl %esi;    \
    popl %edi;    \
    popl %ebp;    \
    popl %eax;    \
1:   popl %ds;    \
2:   popl %es; \
    addl $4,%esp; \
3:   iret;     \
.section .fixup,"ax"; \
4:   movl $0,(%esp); \
    jmp 1b;     \
5:   movl $0,(%esp); \
    jmp 2b;     \
6:   pushl %ss;   \
    popl %ds;   \
    pushl %ss;   \
    popl %es; \
    pushl $11;   \
    call do_exit; \
.previous;     \
.section __ex_table,"a"; \
.align 4; \
.long 1b,4b; \
.long 2b,5b; \
.long 3b,6b; \
.previous
```

从宏的代码中, 可以得知它把 CPU 中的一些寄存器值出栈。并且执行 `iret` 指令返回到用户态。到这为止, 系统调用便执行完成了, 即实现了我们所需要的功能。最后我们用一张简单的图来描述之。请看下图 (该图为了描述方便, 没有把情况都描述清楚, 比如系统调用号超过系统定义的最大系统调用数等等):



## A2-1-2 修改代码来添加系统调用

通过上面的介绍，我们可以知道修改系统调用并不是件难事。那么我们就开始修改吧。  
(假定你的核心在/usr/src/linux 下)

第 1 步：

我们打开 include/linux/sys.h 文件，修改

```
#define NR_syscalls 260
```

为

```
#define NR_syscalls 261
```

(假设我们只要添加一个系统调用)

第 2 步：

个人认为这一步，也可以不做，因为作为系统调用的添加者，你当然是知道你加的系统调用号的。不过我们还是不忽略它。请打开 include/asm-i386/unistd.h 添加如下代码

```
#define __NR_helloworld 259
```

这个名字，你可以自己决定用什么，只要不和系统冲突。

第 3 步：

打开 arch/i386/kernel/entry.S，在 .long SYMBOL\_NAME(sys\_set\_tid\_address) 的后面加入你要添加的系统调用函数名。假设我们要添加的函数名是 sys\_helloworld，于是我们写成这样：

```
.long SYMBOL_NAME(sys_helloworld)
```

第 4 步：

在第 3 步我们只是添加了系统调用的声明，还要添加系统调用的实现体才行（关于系统

调用实现体的添加，有两个方法：第一个方法是写在系统核心的某个文件中，第二个方法是在核心中新添加一个文件，不过用该方法你需要修改对应的 Makefile 文件。这里我们采用第一个方法。）。

对于本例，我们把实现体写在 fs/read\_write.c 中。添加代码如下：

```
asmlinkage void sys_helloworld (void)
{
    printk("Hello world.\n");
}
```

第 4 步：

编译核心。

第 5 步：

核心编译成功后，我们便可以编写代码测试了。这里我们就修改上面的 CH3-5/hello.c 代码就可以的。Makefile 不变。

修改的代码如下：

#### A2-1-2/hello.c

---

```
#include <stdio.h>
#include <unistd.h>
int
main()
{
    unsigned long sys_num = 259;
    __asm__("int $0x80"
            ::"a"((long)(sys_num)));           //系统调用号
    return 0;
}
```

---

修改好后，我们就可以编译并执行了。执行情况如下：

```
[root@localhost CH3-5]# ls -l
total 8
-rw-rw-rw-    1 ftp      ftp          288 Nov  6 12:51 hello.c
-rw-rw-rw-    1 ftp      ftp          142 Nov  6 12:41 Makefile
[root@localhost CH3-5]# make
gcc -c hello.c
gcc hello.o -o hello
[root@localhost CH3-5]# ./hello
Hello world.
[root@localhost CH3-5]# _
```

到这为止，静态添加系统调用便完成了。

## A2-2 动态添加系统调用

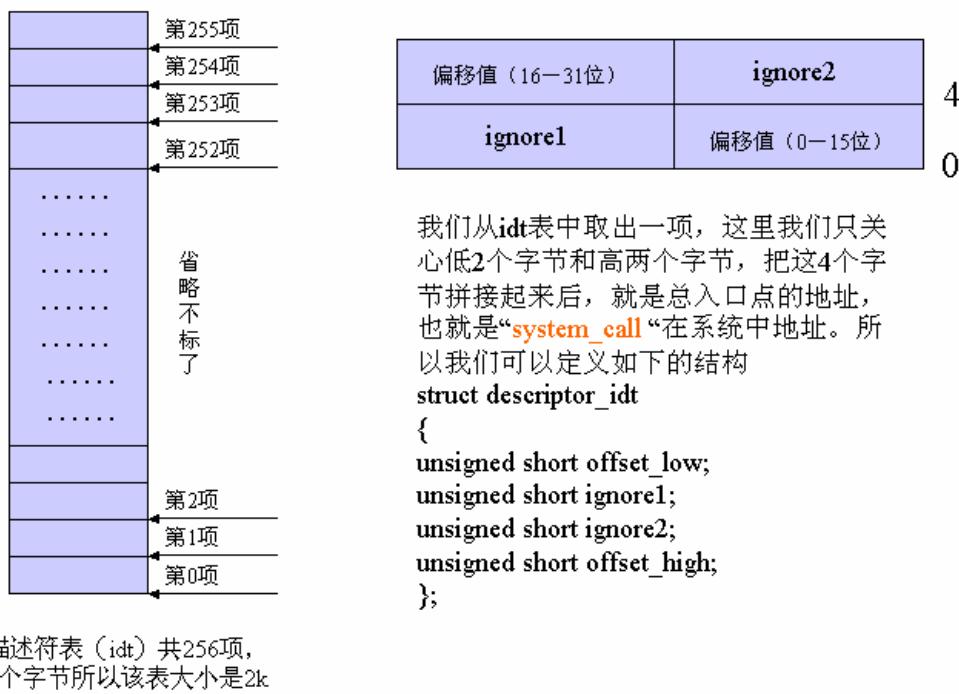
所谓动态添加系统调用，就是在 Linux 运行的时候把新的系统调用加入。从而避免了编

译核心的问题。

## A2-2-1 动态添加系统调用的原理

动态添加系统可能会有很多种方法，这里我们只讨论一种方法。个人认为本书讨论的这种方法是比较好的。同样你也要具有超级用户的权限。

通过上面的对 1.0 的代码的分析，我们可以知道 0x80 中断号是整个系统调用的入口点。进入该入口点后，通过比较传入各种系统调用号来查表得到对应的系统调用处理函数。所以这里我们要是能够取得中断描述符表的基地址，然后以此为起点计算出 0x80 项的地址值。在从该地址处中分解出系统调用的入口点，然后用我们自己定义的入口点替代它。在我们定义的入口点函数中处理我们要处理的系统调用号（随便你干什么）。处理完后在扔给被我们替代掉的入口点函数，即可！下面还是用图来表示，我怕写的不明白！



通过上图我们可以知道，idt 表是个什么样子，并且其中每项的内容。那么谁来描述 idt 表呢？即谁来定位 idt 表呢？在 X86 CPU 上有两条指令来和其相关，它们是“sidt”和“lidt”，分别用来复制、加载 idt。请注意这两条指令的操作数均为 48 位。我们可以定义如下结构来描述之：

```
struct idt_48 {
    unsigned short limit;      ! 描述 idt 表的大小
    unsigned long base;        ! idt 表的基地址，看到了吧这就是基地址了
};
```

所以，我们可以用 sidt 指令来获取系统 idt 表的 48 位值，然后从中获取基地址，再根据取得的基地址计算得到第 0x80 项中的值，取得第 0x80 项的值后，我们便可以根据上图中定

义的结构分解来得到系统调用的入口点函数地址。

啊！等等！如果我把这个地址修改了，不就实现了动态添加、修改系统调用了吗？

恭喜你！你说对了，下面我们可以看代码了。

## A2-2-2 实现动态添加、修改系统调用

### A2-2/capturemod.c

```
#ifndef MODULE
#define MODULE
#endif

#ifndef __KERNEL__
#define __KERNEL__
#endif

#ifndef NULL
#define NULL 0L
#endif

#include <linux/module.h>
#include <asm/unistd.h>
#include <linux/kernel.h>
#include <linux/slab.h>
MODULE_LICENSE("GPL");
//禁止警告提示,加上这句表明你的模块符合 GPL
//请查看模块的编写那节有描述

void new80_handle();
//新的 0x80 处理句柄

static unsigned long old80_handle;
//用于保存老的 0x80 处理句柄

extern char * getname(const char * filename);
//取的用户空间的程序名
extern kmem_cache_t *names_cachep;
//用于释放内存

static unsigned long eax, ebx, ecx;
//保存寄存器内容

struct descriptor_idt
```

```

{
    unsigned short offset_low;
    unsigned short ignore1;
    unsigned short ignore2;
    unsigned short offset_high;
};

//用于描述 idt 表中的一项,共 8 个字节

static struct {
    unsigned short limit;
    unsigned long base;
} __attribute__((packed)) idt48;
//用于定位 idt 表

static void puppet_handle(void)
{
    //首先一点要明确的是,我们不是替代系统中所有系统
    //调用,而只是在原来的基础上修改或者新增系统调用.
    //对原来的没有被你修改的系统调用不能有任何的影响.
    //所以要保证堆栈的正确.

    //看到这个函数名(puppet_handle), 你可能会感到这是个伪函数。
    //没错, 确实是的,该函数没有什么大的作用,只是用来包裹下面的
    //一段嵌入汇编.代码.
    //因为在*.c 文件中,你不能直接写汇编代码,
    //所以我们只有将其写成嵌入汇编的形式了.

    //现在我们讨论一下为什么要写用嵌入汇编.

    //我们先回忆一下发生系统调用时的堆栈的情况
    //系统调用发生时,CPU 会按顺序将 SS,ESP,EFLAGS,CS,EIP
    //这几个寄存器压入堆栈.,然后会调用” system_call”函数(也就
    //是系统调用的总入口点函数),在该函数中会按顺序压入如下
    //寄存器值 ORIG_EAX,ES,DS,EAX,EBP,EDI,ESI,EDX,ECX,EBX(这
    //里压入的寄存器值其实就是系统调用函数将会用的各个参数).通过这里
    //的描述,我们可以知道如果” new80_handle”,用 C 语言来写的是不可能
    //完成的.因为你要在调用” new80_handle”前把原来在 system_call 中压入的
    //寄存器内容先压入堆栈,并且还要修改从 new80_handle 返回地址为原系统的
    //总的入口点函数.可是系统的核心原来已经是编译好的了,总不能让你去修改
    //编译好的二进制文件吧.

    //所以,我们用汇编代码绕过去,并且也不需要知道核心的入栈顺序了.从而保证
    //了在进入老的系统调用总的入口点前堆栈是正确的.

```

```

//而在该汇编代码中调用另一个辅助的函数,
//在该辅助的函数中完成要做的所有工作

//这里还有特别的一点说明是,为了不污染正在运行核心的函数命名空间
//(因为模块被安装后,其全局变量或者函数会到处),所以,所有的函数和变量
//被我们用了 static 来做修饰.也许读者对 void new80_handle();这个函数有疑问
//因为我们没有用 static 来做修饰啊!难道它不会污染命名空间吗,会的.
//不过,我们在写嵌入汇编代码时,用了".type new80_handle,@function\n"
//做了修饰,这样的话 new80_handle 就如 static 类型的函数一样了

//不要担心,在下面我们会讨论这个用 C 语言编写的模块的
//反汇编代码的.你会更加明白的.

```

```

__asm__ (
    ".type new80_handle,@function\n"           //用于修饰函数,以让其和用 static 修饰的一样
    ".align 4\n"                                //内存对齐方式
    "new80_handle: \n"                           //new80_handle 入口点
    "pusha \n"                                  //所有通用寄存器入栈
    "pushl %%es\n"                             //段寄存器 fs 入栈
    "pushl %%ds\n"                             //段寄存器 ds 入栈
    "movl %%eax,%0\n"
    "movl %%ebx,%1\n"
    "movl %%ecx,%2\n"                          //取出我们要用的寄存器值
    "call real_handler \n"                      //调用 real_handler,正如其名.在该函数中
                                                //可以完成我们想做的任何事情
    "popl %%ds\n"                            //段寄存器 ds 出栈
    "popl %%es\n"                            //段寄存器 fs 出栈
    "popa \n"                                 //所有通用寄存器出栈
    "jmp *old80_handle"                       //我们的工作完成后,调用系统起先的
                                                //系统调用入口点,一来可以让它处理
                                                //我们没有做的事情(比如从核心态返回到
                                                //用户态).二来如果不是我们添加的系统调用
                                                //它还能继续处理
    ::"m"(eax),"m"(ebx),"m"(ecx)
);
}

static void real_handler()
{
    char *pName = NULL;          //指向拷贝到核心中程序名
    if(eax == __NR_execve)
        {//捕获 sys_execve 调用
            pName = getname((char*)ebx); //getname 用于把用户态的数据拷贝到
                                            //核心态,ebx 中保存了 sys_execve 系统调用
        }
}

```

```

//的第一个参数,也就是所要执行文件的文件名
//的地址

if(pName)
{
    printk(KERN_INFO"The program is %s.\n",pName);
    //打印提示信息
    kmem_cache_free(names_cachep, (void *)(pName));
    //释放放在 getname 中所分配的内存
}

}

else if(eax == 0x200)
{
    //截获 0x200 号系统调用, 该系统调用不存在, 用我给的程序测试!
    //假设这里只打印 eax,ebx,ecx,当然你也可以做其他事情!
    printk(KERN_INFO"eax=0x%x, ebx=0x%x, ecx=0x%x\n",eax,ebx,ecx);
}
}

int init_module(void)
{
    __asm__ volatile ("sidt %0": "=m" (idt48));
    //取得 idt 表的 48 位值
    struct descriptor_idt *pIdt80 = (struct descriptor_idt *) (idt48.base + 8*0x80);
    //并让 pIdt80 指向 idt 表中第 0x80 项
    old80_handle = (pIdt80->offset_high<<16 | pIdt80->offset_low);
    //保存老的总入口点
    unsigned long new80_addr = (unsigned long)new80_handle;
    //把新的 0x80 入口点函数地址转换成 unsigned long,为了便于下面可以分解
    pIdt80->offset_low = (unsigned short) (new80_addr & 0x0000FFFF);
    //把新入口点的低 16 位设置到 0x80 项中对应处
    pIdt80->offset_high = (unsigned short) (new80_addr >> 16);
    //把新入口点的高 16 位设置到 0x80 项中对应处

    //另外要注意的是,我们没有改变 ignore1 和 ignore2 的内容.如果你很想修改这两个字
    //段的内容,请确认你知道在做什么,这里不讨论.

    printk(KERN_INFO"Ok,we capture syscall successful.\n");
    //打印提示信息
    return 0;
}

void cleanup_module()
{
    __asm__ volatile ("sidt %0": "=m" (idt48));
    //取得 idt 表的 48 位值
}

```

---

```

struct descriptor_idt *pIdt80 = (struct descriptor_idt *) (idt48.base + 8*0x80);
//让 pIdt80 指向 idt 表中第 0x80 项
pIdt80->offset_low = (unsigned short) (old80_handle & 0x0000FFFF);
//恢复老的入口点的低 16 位
pIdt80->offset_high = (unsigned short) (old80_handle >> 16);
//恢复老的入口点的高 16 位
printk(KERN_INFO "Ok, we leave capture.\n");
//打印提示信息
}

```

---

## A2-2-2/Makefile

---

```

GCC=gcc
KERNELDIR=/usr/src/linux/include
OBJS=capturemod.o
TESTMOD=testmod/testmod

.c.o:
$(GCC) -D__KERNEL__ -I$(KERNELDIR) -c $^ -o $@

all: $(OBJS) $(TESTMOD)

$(TESTMOD):
    make -C testmod

insert: $(OBJS)
    /sbin/insmod $(OBJS)
remove:
    /sbin/rmmod helloworld

clean:
    rm -f *.o core
    rm -f testmod/*.o
clobber: clean
    rm -f $(TESTMOD)

```

---

## A2-2-2/testmod/ testmod.c

---

```

#include <stdio.h>
#include <unistd.h>
int
main()

```

```
{
int sys_num = 0x200;//512 号系统调用（不存在，用来测试上面的模块而已）
long value = 0;
__asm__("int $0x80"
        :"=a"(value)
        :"0"((int)sys_num));
printf("The value is %d.\n",value);
return value;
}
```

---

## A2-2-2/testmod/Makefile

```
GCC=gcc
OBJS=testmod.o

.c.o:
$(GCC) -c $<
all:$(OBJS)
$(GCC) $(OBJS) -o testmod

clean:
rm -f *.o core
clobber:clean
rm -f testmod
```

---

下面请看编译和执行情况,这里直给出截图,正所谓一图千言吗!

下图是编译的情况

```
[root@localhost CH3-6]# ls -l
total 16
-rw-rw-rw-    1 ftp      ftp          5635 Nov  9 10:52 capturemod.c
-rw-rw-rw-    1 ftp      ftp          364 Nov  9 11:18 Makefile
drw-rw-rw-   2 ftp      ftp         4096 Nov  9 11:22 testmod
[root@localhost CH3-6]# make
gcc -D__KERNEL__ -I/usr/src/linux/include -c capturemod.c -o capturemod.o
make -C testmod
make[1]: Entering directory '/var/ftp/pub/CH3-6/testmod'
gcc -c testmod.c
gcc testmod.o -o testmod
make[1]: Leaving directory '/var/ftp/pub/CH3-6/testmod'
[root@localhost CH3-6]# _
```

下图是我们插入模块时的情况

```
[root@localhost CH3-6]# insmod ./capturemod.o
Ok,we capture syscall successful.
[root@localhost CH3-6]# _
```

下图是测试程序执行的情况,请读者仔细看屏幕的提示,我不做解释了

```
[root@localhost CH3-6]# insmod ./capturemod.o
Ok,we capture syscall successful.
[root@localhost CH3-6]# cd testmod/
[root@localhost testmod]# ls
The program is /bin/ls.
Makefile testmod.c testmod.o
[root@localhost testmod]# ./testmod
The program is ./testmod.
eax=0x200, ebx=0x42130a14, ecx=0x42015554
The value is -38.
[root@localhost testmod]# _
```

下图是模块注销的情况

```
[root@localhost CH3-6]# rmmod capturemod
The program is /sbin/rmmod.
Ok,we leave capture.
[root@localhost CH3-6]# _
```

### A2-2-3 反汇编 capturemod.o 并分析之

本节来做 capturemod.o 的反汇编代码的分析,这样我们能够更好的理解上面的 C 语言所完成的代码.通过 objdump 命令可以反汇编上面的 capturemod.o 模块.

请在命令行下输入如下命令”objdump -D capturemod.o > mod.s”打开后便是下面的内容.请跟着我看吧

```
capturemod.o:      file format elf32-i386
//识别出该模块格式是 elf
Disassembly of section .text:
//代码段的反汇编
00000000 <puppet_handle>:
 0: 55                      push   %ebp
 1: 89 e5                   mov    %esp,%ebp
 3: 90                      nop
```

```
//上面的3句代码,是 puppet_handle 反汇编代码,什么也不做,并且没有任何地方调用它
00000004 <new80_handle>:           //new80_handle 函数的反汇编代码
    4: 60                         pusha      //所有通常寄存器值内容入栈
    5: 06                         push       %es //段寄存器 es 入栈
    6: 1e                         push       %ds //段寄存器 ds 入栈
    7: a3 04 00 00 00             mov        %eax,0x4
    c: 89 1d 08 00 00 00             mov        %ebx,0x8
   12: 89 0d 0c 00 00 00             mov        %ecx,0xc //把 eax,ebx,ecx 放入 0x4,0x8,0xc 处(看到
                                         //这个样子大家可能会有点奇怪,为什么
                                         //地址这么小呢?原因在于在该模块被
                                         //插入核心时,才会被重定位)
   18: e8 0b 00 00 00             call      28 <real_handler> //调用我们真正的处理程序
   1d: 1f                         pop       %ds
   1e: 07                         pop       %es
   1f: 61                         popa      //把上面入栈的内容退回到对应的寄
                                         //存器中,这样我们就保证了堆栈的
                                         //正确行,让真正的系统调用处理
                                         //函数感觉不到我们已经做过了
                                         //我们需要做的事
   20: ff 25 00 00 00 00             jmp      *0x0 //直接跳转到老的系统调用入口处
   26: c9                         leave
   27: c3                         ret      //因为跳到老的系统调用入口点时用的
                                         //是 jmp 指令

00000028 <real_handler>:           //真正完成工作的函数,这里就不做
                                         //分析了,因为它会随着你的不同实现
                                         //而不同
    28: 55                         push      %ebp
    29: 89 e5                      mov       %esp,%ebp
    2b: 83 ec 08                    sub       $0x8,%esp
    2e: c7 45 fc 00 00 00 00         movl     $0x0,0xfffffff(%ebp)
    35: 83 3d 04 00 00 00 0b         cmpl     $0xb,0x4
    3c: 75 43                      jne      81 <real_handler+0x59>
    3e: 83 ec 0c                    sub       $0xc,%esp
    41: ff 35 08 00 00 00             pushl    0x8
    47: e8 fc ff ff ff             call      48 <real_handler+0x20>
    4c: 83 c4 10                    add       $0x10,%esp
    4f: 89 45 fc                  mov       %eax,0xfffffff(%ebp)
    52: 83 7d fc 00                cmpl     $0x0,0xfffffff(%ebp)
    56: 74 54                      je       ac <real_handler+0x84>
    58: 83 ec 08                    sub       $0x8,%esp
    5b: ff 75 fc                  pushl    0xfffffff(%ebp)
    5e: 68 00 00 00 00             push     $0x0
   63: e8 fc ff ff ff             call      64 <real_handler+0x3c>
```

```

68: 83 c4 10          add    $0x10,%esp
6b: 83 ec 08          sub    $0x8,%esp
6e: ff 75 fc          pushl  0xfffffc(%ebp)
71: ff 35 00 00 00 00 pushl  0x0
77: e8 fc ff ff ff   call   78 <real_handler+0x50>
7c: 83 c4 10          add    $0x10,%esp
7f: eb 2b              jmp    ac <real_handler+0x84>
81: 81 3d 04 00 00 00 cmpl   $0x200,0x4
88: 02 00 00
8b: 75 1f              jne    ac <real_handler+0x84>
8d: ff 35 0c 00 00 00 pushl  0xc
93: ff 35 08 00 00 00 pushl  0x8
99: ff 35 04 00 00 00 pushl  0x4
9f: 68 20 00 00 00    push   $0x20
a4: e8 fc ff ff ff   call   a5 <real_handler+0x7d>
a9: 83 c4 10          add    $0x10,%esp
ac: c9                leave
ad: c3                ret

```

000000ae <init\_module>: //模块插入时执行的函数

```

ae: 55                 push   %ebp
af: 89 e5              mov    %esp,%ebp
b1: 83 ec 08          sub    $0x8,%esp
                        //在栈上留下空间,该空间是为 pIdt80 和 new80_addr
                        //留的
b4: 0f 01 0d 10 00 00 00 sidtl 0x10 //取得 idt 的 48 位指针
bb: a1 12 00 00 00      mov    0x12,%eax //把后 32 位值送入 eax,也就是 idt 表
                        //的基址
c0: 05 00 04 00 00      add    $0x400,%eax //取得第 0x80 项后,放入 eax 中
c5: 89 45 fc          mov    %eax,0xfffffc(%ebp)
c8: 8b 45 fc          mov    0xfffffc(%ebp),%eax
cb: 0f b7 40 06      movzwl 0x6(%eax),%eax //取出第 0x80 项的第 6 个字节开始
                        //的两个字节(高两个字节)
cf: 89 c2              mov    %eax,%edx//把取出的字节放入 edx
d1: c1 e2 10          shl    $0x10,%edx //左移 16 位
d4: 8b 45 fc          mov    0xfffffc(%ebp),%eax
d7: 0f b7 00          movzwl (%eax),%eax //低 16 位
da: 09 d0              or     %edx,%eax//相或后便是老的系统入口点函数
dc: a3 00 00 00 00      mov    %eax,0x0 //保存起来
e1: c7 45 f8 04 00 00 00 movl   $0x4,0xfffff8(%ebp)
e8: 8b 55 fc          mov    0xfffffc(%ebp),%edx
eb: 8b 45 f8          mov    0xfffff8(%ebp),%eax
ee: 66 89 02          mov    %ax,(%edx)//取的低 16 位,刚好放在
                        //(edx)的低 16 位,也就是第

```

```

//0-1 两个字节中
f1: 8b 55 fc          mov    0xfffffff(%ebp),%edx
f4: 8b 45 f8          mov    0xfffffff8(%ebp),%eax
f7: c1 e8 10          shr    $0x10,%eax //高 16 位
fa: 66 89 42 06       mov    %ax,0x6(%edx)//从高 6 个字节处开始放
                           //到此时,也就意味着
                           //修改了老的入口点了
fe: 83 ec 0c          sub    $0xc,%esp
101: 68 60 00 00 00   push   $0x60
106: e8 fc ff ff ff  call   107 <init_module+0x59>//调用 printk 函数
10b: 83 c4 10          add    $0x10,%esp
10e: b8 00 00 00 00   mov    $0x0,%eax
113: c9                  leave
114: c3                  ret    //返回

```

00000115 <cleanup\_module>: //不作分析了和 init\_module  
//差不多的

```

115: 55                  push   %ebp
116: 89 e5                mov    %esp,%ebp
118: 83 ec 08              sub    $0x8,%esp
11b: 0f 01 0d 10 00 00 00  sidt   0x10
122: a1 12 00 00 00        mov    0x12,%eax
127: 05 00 04 00 00        add    $0x400,%eax
12c: 89 45 fc              mov    %eax,0xfffffff(%ebp)
12f: 8b 55 fc              mov    0xfffffff(%ebp),%edx
132: 66 a1 00 00 00 00    mov    0x0,%ax
138: 66 89 02              mov    %ax,(%edx)
13b: 8b 55 fc              mov    0xfffffff(%ebp),%edx
13e: a1 00 00 00 00        mov    0x0,%eax
143: c1 e8 10              shr    $0x10,%eax
146: 66 89 42 06          mov    %ax,0x6(%edx)
14a: 83 ec 0c              sub    $0xc,%esp
14d: 68 86 00 00 00        push   $0x86
152: e8 fc ff ff ff      call   153 <cleanup_module+0x3e>
157: 83 c4 10              add    $0x10,%esp
15a: c9                  leave
15b: c3                  ret

```

Disassembly of section .data:

Disassembly of section .modinfo:

00000000 <\_\_module\_kernel\_version>: //版本信息

```

0: 6b 65 72 6e          imul   $0x6e,0x72(%ebp),%esp
4: 65                  gs
5: 6c                  insb   (%dx),%es:(%edi)

```

```

6: 5f          pop    %edi
7: 76 65       jbe    6e <real_handler+0x46>
9: 72 73       jb     7e <real_handler+0x56>
b: 69 6f 6e 3d 32 2e 34 imul   $0x342e323d,0x6e(%edi),%ebp
12: 2e 32 30  xor    %cs:(%eax),%dh
15: 2d 38 00 6c 69 sub    $0x696c0038,%eax

```

00000018 <\_\_module\_license>: //License 信息

```

18: 6c          insb   (%dx),%es:(%edi)
19: 69 63 65 6e 73 65 3d imul   $0x3d65736e,0x65(%ebx),%esp
20: 47          inc    %edi
21: 50          push   %eax
22: 4c          dec    %esp

```

...

Disassembly of section .rodata:

00000000 <.rodata>: //只读数据区不做分析

```

0: 3c 32         cmp    $0x32,%al
2: 3e          ds
3: 54          push   %esp
4: 68 65 20 70 72 push   $0x72702065
9: 6f          outsl  %ds:(%esi),(%dx)
a: 67 72 61     addr16 jb 6e <.rodata+0x6e>
d: 6d          insl   (%dx),%es:(%edi)
e: 20 69 73     and    %ch,0x73(%ecx)
11: 20 25 73 2e 0a 00 and    %ah,0xa2e73
...
1f: 00 3c 32     add    %bh,(%edx,%esi,1)
22: 3e          ds
23: 65          gs
24: 61          popa
25: 78 3d         js    64 <.rodata+0x64>
27: 30 78 25     xor    %bh,0x25(%eax)
2a: 78 2c         js    58 <.rodata+0x58>
2c: 20 65 62     and    %ah,0x62(%ebp)
2f: 78 3d         js    6e <.rodata+0x6e>
31: 30 78 25     xor    %bh,0x25(%eax)
34: 78 2c         js    62 <.rodata+0x62>
36: 20 65 63     and    %ah,0x63(%ebp)
39: 78 3d         js    78 <.rodata+0x78>
3b: 30 78 25     xor    %bh,0x25(%eax)
3e: 78 0a         js    4a <.rodata+0x4a>
...
60: 3c 32         cmp    $0x32,%al

```

```

62: 3e          ds
63: 4f          dec    %edi
64: 6b 2c 77 65 imul   $0x65,(%edi,%esi,2),%ebp
68: 20 63 61   and    %ah,0x61(%ebx)
6b: 70 74       jo     e1 <init_module+0x33>
6d: 75 72       jne    e1 <init_module+0x33>
6f: 65 20 73 79 and    %dh,%gs:0x79(%ebx)
73: 73 63       jae    d8 <init_module+0x2a>
75: 61          popa
76: 6c          insb   (%dx),%es:(%edi)
77: 6c          insb   (%dx),%es:(%edi)
78: 20 73 75   and    %dh,0x75(%ebx)
7b: 63 63 65   arpl   %sp,0x65(%ebx)
7e: 73 73       jae    f3 <init_module+0x45>
80: 66          data16
81: 75 6c       jne    ef <init_module+0x41>
83: 2e 0a 00   or     %cs:(%eax),%al
86: 3c 32       cmp    $0x32,%al
88: 3e          ds
89: 4f          dec    %edi
8a: 6b 2c 77 65 imul   $0x65,(%edi,%esi,2),%ebp
8e: 20 6c 65 61 and    %ch,0x61(%ebp,2)
92: 76 65       jbe    f9 <init_module+0x4b>
94: 20 63 61   and    %ah,0x61(%ebx)
97: 70 74       jo     10d <init_module+0x5f>
99: 75 72       jne    10d <init_module+0x5f>
9b: 65 2e 0a 00 or     %cs:%gs:(%eax),%al

```

## A3.函数库的编写

在 Linux 系统下的函数库有两种不同的情况，分为静态以及动态函数库。这两种函数库的各有各的优点和缺点。请看下面的描述。

### A3-1 静态函数库的编写

所谓的静态函数库就是一些目标模块的组合，这些模块被集中放在一个文件中，然后我们在连接程序时便可以连接这个文件了。

我们这里把 3 个排序算法写成一个静态库，请看代码（注：这里的排序算法有“希尔”，“起泡”，“选择”，用他们对整型数组进行排序，并且排序时并不是从我们通常的第 0 项开始的，而是从第 1 项开始的，这个大家请注意！），这里我们不对算法进行分析，互连网上可以找到各种关于这些算法的讨论。

### A3-1-1 包含算法的各个文件及 Makefile

sortlib/ shellsort.c

---

```
#include "sortlib.h"
//希尔排序算法
//r=要排序的数组（请注意该数组的第0项没有参加排序，如果有兴趣读者可以自己修改）
//n=数组的大小（等于你所定义数组的大小减一）
void shellsort(int r[], int n)
{
    int i,j,gap,x;
    gap = n/2;
    while(gap > 0)
    {
        for(i=gap+1; i<=n; ++i)
        {
            j = i - gap;
            while(j > 0)
            if(r[j] > r[j+gap])
            {
                x = r[j];
                r[j] = r[j+gap];
                r[j+gap] = x;
                j = j - gap;
            }
            else j = 0;
        }
        gap = gap/2;
    }
}
```

---

sortlib/ bubblesort.c

---

```
#include "sortlib.h"
//起泡排序算法
//r=要排序的数组（请注意该数组的第0项没有参加排序，如果有兴趣读者可以自己修改）
//n=数组的大小（等于你所定义数组的大小减一）
void bubblesort(int r[],int n)
{
    int i,j,w;
    for(i=1; i<=n-1; ++i)
        for(j=n; j>=i+1; --j)
            if(r[j] < r[j-1])
```

---

```

{
w = r[j];
r[j] = r[j-1];
r[j-1] = w;
}
}
```

---

## sortlib/selectsort.c

---

```

#include "sortlib.h"
//选择排序算法
//r=要排序的数组（请注意该数组的第0项没有参加排序，如果有兴趣读者可以自己修改）
//n=数组的大小（等于你所定义数组的大小减一）
void selectsort(int r[], int n)
{
    int i,j,k,temp;
    for(i=1; i<=n-1; ++i)
    {
        k = i;
        for(j=i+1; j<=n; ++j)
            if(r[j] < r[k]) k = j;
        temp = r[i];
        r[i] = r[k];
        r[k] = temp;
    }
}
```

---

## sortlib/ sortlib.h

---

```

//包含了上面3中算法的函数定义原形
#ifndef __SORT_LIB_H__
#define __SORT_LIB_H__
void shellsort(int r[],int n);
void bubblesort(int r[], int n);
void selectsort(int r[],int n);
#endif
```

---

## sortlib/Makefile

---

```
#用于编译生成 libsort.a 的静态库
```

```
GCC=gcc
OBJS=shellsort.o bubblesort.o selectsort.o
LIB=libsort.a
```

```
.c.o:
$(GCC) -c -Wall $<
SLIB:$ (OBJS)
ar r $(LIB) $(OBJS)
```

```
clean:
rm -f *.o core
clobber:clean
rm -f *.a
```

---

## A3-1-2 测试静态函数库的程序及 Makefile

### A3-1-1/test.c

---

```
#include "stdio.h"
#include "sortlib/sortlib.h"      //我们这里包含 sortlib/sortlib.h 为的是可以顺利通过编译
                                //就象上面我们包含了 stdio.h 一样，这样我们就可以使用
                                //printf (...) 了。
int
main()
{
    int i=0;
    int n[10] = {-1,-1,32,-33,34,35,36,-7,8,9};
    printf("Before sort:\n");
    for(i=1; i<10; ++i)
        printf("%d    ",n[i]);
    printf("\n");
    //selectsort(n,9);
    //bubblesort(n,9);
    shellsort(n,9);           //请注意这里我只调用了 shellsort 的函数（希尔排序），其他的被注释了，读者可以自行关闭注释来测试它们
    printf("After sort:\n");
    for(i=1; i<10; ++i)
        printf("%d    ",n[i]);
    printf("\n");
    return 0;
```

---

```
}
```

---

---

### A3-1-1/Makefile

---

```
GCC=gcc
OBJS=test.o
SLIB=sortlib/libsort.a

.c.o:
    $(GCC) -c -Wall $<

all:$(OBJS) $(SLIB)
    $(GCC) $(OBJS) $(SLIB) -o test

$(SLIB):
    make -C sortlib
clean:
    rm -f *.o core
    rm -f sortlib/*.o
clobber:clean
    rm -f sortlib/*.a
    rm -f test
```

---

### A3-1-3 静态库编译情况

我们首先进入 sortlib 文件夹编译生成 libsort.a(注：也可以直接在 A3-1-1 编译生成该库)  
请看下图

```
[root@localhost CH3-1]# cd sortlib/
[root@localhost sortlib]# ls -l
total 20
-rw-rw-rw- 1 ftp      ftp          191 Oct  1 14:07 bubblesort.c
-rw-rw-rw- 1 ftp      ftp          190 Oct  1 15:22 Makefile
-rw-rw-rw- 1 ftp      ftp          214 Oct  1 14:20 selectsort.c
-rw-rw-rw- 1 ftp      ftp          305 Oct  1 15:04 shellsort.c
-rw-rw-rw- 1 ftp      ftp          153 Oct  1 14:20 sortlib.h
[root@localhost sortlib]# make
gcc -c -Wall shellsort.c
gcc -c -Wall bubblesort.c
gcc -c -Wall selectsort.c
ar r libsort.a shellsort.o bubblesort.o selectsort.o
[root@localhost sortlib]# ls -l
total 36
-rw-rw-rw- 1 ftp      ftp          191 Oct  1 14:07 bubblesort.c
-rw-r--r-- 1 root    root          781 Oct  1 17:51 bubblesort.o
-rw-r--r-- 1 root    root          2756 Oct  1 17:51 libsort.a
-rw-rw-rw- 1 ftp      ftp          190 Oct  1 15:22 Makefile
-rw-rw-rw- 1 ftp      ftp          214 Oct  1 14:20 selectsort.c
-rw-r--r-- 1 root    root          801 Oct  1 17:51 selectsort.o
-rw-rw-rw- 1 ftp      ftp          305 Oct  1 15:04 shellsort.c
-rw-r--r-- 1 root    root          875 Oct  1 17:51 shellsort.o
-rw-rw-rw- 1 ftp      ftp          153 Oct  1 14:20 sortlib.h
[root@localhost sortlib]#
```

看上图用红色线框标出的文件（libsort.a）便是生成的静态库文件，该库文件可以被各种程序连接，用于调用其中定义的排序函数。

那么该静态库是如何生成的呢？请看上图中用蓝色线框标出的内容，这里的内容其实对应于 Makefile 中的“`ar r $(LIB) $(OBJS)`”。所以通过这里我们也可以看出所谓的静态库只不过是一些目标文件的合并后的结果而已。让我们验证一下这中说法。请看下图

```
[root@localhost sortlib]# ls
bubblesort.c libsort.a selectsort.c shellsort.c sortlib.h
bubblesort.o Makefile selectsort.o shellsort.o
[root@localhost sortlib]# ar tv libsort.a
rw-r--r-- 0/0    875 Oct  1 17:51 2005 shellsort.o
rw-r--r-- 0/0    781 Oct  1 17:51 2005 bubblesort.o
rw-r--r-- 0/0    801 Oct  1 17:51 2005 selectsort.o
[root@localhost sortlib]#
```

到这里读者也许会有一个问题，如果我修改了静态库中的一个目标文件，是不是一定要重新 make 一次呢？当然也不是未尝不可。但是如果对于大型的函数库来说恐怕就有点不妥了。因为我们要重新编译整个函数库，纵然不编译也要浪费时间的。正是因为这样所以我们单独更新改变了的目编文件，请看下图（这里我们假设 shellsort.o 已经被更新了）

```
[root@localhost sortlib]# ar tv libsort.a
rw-r--r-- 0/0    875 Oct  1 18:05 2005 shellsort.o
rw-r--r-- 0/0    781 Oct  1 17:51 2005 bubblesort.o
rw-r--r-- 0/0    801 Oct  1 17:51 2005 selectsort.o
[root@localhost sortlib]# ar r libsort.a shellsort.o
[root@localhost sortlib]# ar tv libsort.a
rw-r--r-- 0/0    875 Oct  1 18:07 2005 shellsort.o
rw-r--r-- 0/0    781 Oct  1 17:51 2005 bubblesort.o
rw-r--r-- 0/0    801 Oct  1 17:51 2005 selectsort.o
[root@localhost sortlib]#
```

更新静态库

请看在执行了 ar r libsort.a shellsort.o 后，libsort.a 中的 shellsort.o 真的被更新了，图中红色线框中所标出的可以看出这种变化。

### A3-1-3 主程序与静态库连接

进入 A3-1-1，执行 make 后，我们会得到一各名为 test 的可执行文件，编译情况如下图

```
[root@localhost CH3-1]# ls -l
total 12
-rw-rw-rw-    1 ftp      ftp          255 Oct  1 15:25 Makefile
drw-rw-rw-    2 ftp      ftp          4096 Oct  1 18:37 sortlib
-rw-rw-rw-    1 ftp      ftp          360 Oct  1 15:26 test.c
[root@localhost CH3-1]# make
gcc -c -Wall test.c
gcc test.o sortlib/libsort.a -o test
[root@localhost CH3-1]# ls -l
total 28
-rw-rw-rw-    1 ftp      ftp          255 Oct  1 15:25 Makefile
drw-rw-rw-    2 ftp      ftp          4096 Oct  1 18:37 sortlib
-rwxr-xr-x    1 root     root         12163 Oct  1 18:37 test
-rw-rw-rw-    1 ftp      ftp          360 Oct  1 15:26 test.c
-rw-r--r--    1 root     root         1176 Oct  1 18:37 test.o
[root@localhost CH3-1]#
```

我们可以看到执行 make 时，test.o 与 sortlib/libsort.a 连接后，生成可执行文件 test，图中黄色线框中已标出。下面我们看看其执行情况。

```
[root@localhost CH3-1]# ls -l
total 28
-rw-rw-rw-    1 ftp      ftp          255 Oct  1 15:25 Makefile
drw-rw-rw-    2 ftp      ftp          4096 Oct  1 18:37 sortlib
-rwxr-xr-x    1 root     root         12163 Oct  1 18:37 test
-rw-rw-rw-    1 ftp      ftp          360 Oct  1 15:26 test.c
-rw-r--r--    1 root     root         1176 Oct  1 18:37 test.o
[root@localhost CH3-1]# ./test
Before sort:
-1   32   -33   34   35   36   -7   8   9
After sort:
-33   -7   -1   8   9   32   34   35   36
[root@localhost CH3-1]#
```

图中红色线框标出的便是执行前后的的结果！

### A3-2 动态函数库的编写

在介绍动态库编写前，我们先说说有了静态库后为什么还要动态库。

假设有一个巨大的静态库，它里面包含了在 firefox 浏览器中所要使用的 90% 以上的函

数，在生成 firefox 时使用了这个函数库，最终我们得到了一个大小为 10M 的可执行文件，我们要说的是：“firefox 中包含了 9M 以上的该静态库的拷贝”。

现在有一个人或者某公司或者某组织，想要做个利用该静态库做出另外一个版本浏览器来（假设为 firefox-II）或者网络程序来，假设和该静态库连接后得到的新的可执行程序大小为 20M，同 firefox 一样该新生成的程序中也包含了 9M 以上的该静态库的拷贝（天啊：他们都是一样的啊）如果你认为这没有什么大不了的，那么假如另外还有 n 多个程序都用了这个库时，将是个多大的问题，你系统要配置多少内存才行呢（512M，1G，甚至更多）？

于是便有了动态函数库，以允许在系统中多个程序可以使用同一份代码。下面我们介绍它。

这里我们还以上面的代码为例，做些修改。以便生成动态库。

### A3-2-1 动态库编译情况

这里我们需要修改 sortlib/Makefile 文件，以让其生成动态库。

---

#### sortlib/Makefile (该 Makefile 用于生成动态库)

---

```
#用于编译生成 libsort.so 的动态库
GCC=gcc
OBJS=shellsort.o bubblesort.o selectsort.o
LIB=libsort.so

.c.o:
    $(GCC) -c -Wall $<

DLIB:$(OBJS)
    gcc $(OBJS) -shared -o $(LIB)

clean:
    rm -f *.o core
clobber:clean
    rm -f *.so
```

---

我们再修改 A3-1-1 下的 Makefile 文件，用于连接生成的动态库。

---

#### A3-1-1/Makefile (修改后的 Makefile)

---

```
GCC=gcc
OBJS=test.o
DLIB=./sortlib/sort

.c.o:
    $(GCC) -c -Wall $<
```

```

all:$(OBJS) $(DLIB)
    $(GCC) $(OBJS) -L./sortlib -lsort -o test

$(DLIB):
    make -C sortlib
clean:
    rm -f *.o core
    rm -f sortlib/*.o
clobber:clean
    rm -f sortlib/*.a
    rm -f test

```

---

修改好后，我们便可以重新编译以得到动态库，执行请看如下图

```

[root@localhost sortlib]# ls
bubblesort.c  Makefile  selectsort.c  shellsort.c  sortlib.h
[root@localhost sortlib]# make
gcc -c -Wall shellsort.c
gcc -c -Wall bubblesort.c
gcc -c -Wall selectsort.c
gcc shellsort.o bubblesort.o selectsort.o -shared -o libsort.so
[root@localhost sortlib]# ls -l
total 40
-rw-rw-rw-  1 ftp      ftp          191 Oct  1 14:07 bubblesort.c
-rw-r--r--  1 root     root         781 Oct  1 19:59 bubblesort.o
-rwxr-xr-x  1 root     root        7192 Oct  1 19:59 libsort.so
-rw-rw-rw-  1 ftp      ftp          202 Oct  1 19:22 Makefile
-rw-rw-rw-  1 ftp      ftp          214 Oct  1 14:20 selectsort.c
-rw-r--r--  1 root     root         801 Oct  1 19:59 selectsort.o
-rw-rw-rw-  1 ftp      ftp          302 Oct  1 18:07 shellsort.c
-rw-r--r--  1 root     root         875 Oct  1 19:59 shellsort.o
-rw-rw-rw-  1 ftp      ftp          153 Oct  1 14:20 sortlib.h
[root@localhost sortlib]# -

```

图中红色线框标出的便是生成的动态库。

上面生成了动态库，接下来我们编译 A3-1-1 下的 test.c，让它动态连接到该动态库上，执行情况请看下图

```

[root@localhost CH3-2]# make
gcc -c -Wall test.c
make -C sortlib
make[1]: Entering directory '/var/ftp/pub/CH3-2/sortlib'
gcc shellsort.o bubblesort.o selectsort.o -shared -o libsort.so
make[1]: Leaving directory '/var/ftp/pub/CH3-2/sortlib'
gcc test.o -L./sortlib -lsort -o test
[root@localhost CH3-2]# -

```

图中红色线框标出的便是动态连接生成可执行文件 test 的命令，对应于 Makefile 中的“\$(GCC) \$(OBJS) -L./sortlib -lsort -o test”，这里具体讲一下该命令。

命令中-L 用于指示将要被连接的动态库所在的目录。对于本例，动态库在 sortlib 文件夹下。

命令中-lsort 用于告诉连接器查找一个动态库 libsort.so 或者一个静态库 libsort.a。请注意 lib 是选项-l 中给出的名称的前缀，并且附加了后缀.a 和.so 来判断给出的名称的函数库是否存在。连接器查找所有标准库的目录和所有以-L 指定的目录。

在生成了 test 后，我们可以用 ldd 命令来检查 libsort.so 是否真的被动态连接了。执行情况请看下图

```
[gdu@localhost CH3-2]$ ldd test
 libsort.so => not found
 libc.so.6 => /lib/tls/libc.so.6 (0x42000000)
 /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[gdu@localhost CH3-2]$ -
```

从上图的红色线框所标的，我们缺省可以看到 libsort.so 确实被动态连接进了 test 程序，不过从提示上看来系统并不能找到它。如果这个时候我们执行 test 程序，就会有错误提示，请看下图

```
[gdu@localhost CH3-2]$ ldd test
 libsort.so => not found
 libc.so.6 => /lib/tls/libc.so.6 (0x42000000)
 /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[gdu@localhost CH3-2]$ ./test
./test: error while loading shared libraries: libsort.so: cannot open shared object file: No such file or directory
[gdu@localhost CH3-2]$ -
```

图中红色线框所标即是错误提示。为什么会有这种情况的发生，我们明明已经动态连接了啊！这是因为动态装载器不能找到该 libsort.so 库。

## A3-2-2 使用动态装载器

在上一节我们可以看到动态库虽然被连接进了程序，但是装载器是找不到它的。在 Linux 下对于要在运行时装载的共享函数库，动态装载器必须知道运行时在那里可以定位找到它。如我们在 shell 下执行命令时，shell 可以根据设置的路径找到要执行的命令一样。所以动态装载器也需要一个查找路径。

我们可以有多种方式定位共享函数库，这里我们介绍两种（这也是比较简单的）。

第一种办法是使用环境变量，我们可以执行如下的命令

“export LD\_LIBRARY\_PATH=“\$LD\_LIBRARY\_PATH:你自己的路径”

请看下图执行情况

```
[root@localhost CH3-2]# export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/var/ftp/pub/CH3-2/sortlib"
[root@localhost CH3-2]# ldd ./test
    libsort.so => /var/ftp/pub/CH3-2/sortlib/libsort.so (0x40017000)
    libc.so.6 => /lib/tls/libc.so.6 (0x42000000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[root@localhost CH3-2]# ./test
Before sort:
-1   32   -33   34   35   36   -7   8   9
After sort:
-33   -7   -1   8   9   32   34   35   36
[root@localhost CH3-2]#
```

第二种办法是把动态库路径加入/etc/ld.so.conf 文件中，这样做需要你具有超级用户的权限，我们可以先看看系统的/etc/ld.so.conf 文件，在我的系统下包含如下的内容。

```
/usr/kerberos/lib
/usr/X11R6/lib
/usr/lib/qt-3.1/lib
-
-
-
```

下面我们把动态库所在的路径加入到该文件中（注：该文件是个文本文件，用 vi, gedit 之类的编辑器均可以打开）在我的系统中加入这样一行便可以（读者可以根据自己的路径加入）

“/var/ftp/pub/CH3-2/sortlib”

请注意在把该行内容加入后，并不能立即生效，我们需要执行/sbin/ldconfig 命令来更新 /etc/ld.so.conf 文件的缓冲/etc/ld.so.conf.cache，因为/etc/ld.so.conf.cache 这个文件才是动态装载器使用的。

当执行完/sbin/ldconfig 命令后，我们便可以使用了，执行的情况和上面的一样，这里就不在描述。

## A3-3 动态/静态函数库优点

### A3-3-1 静态库优点

尽管静态库会增加系统的开销，但它还有优点的。

1. 静态函数使用简单。
2. 可执行文件不依赖相关的外部部件，因为可执行文件中包含了一切它需要的东西。
3. 静态函数库没有环境或者管理问题。
4. 静态函数库不需要是位置独立代码。

### A3-3-2 动态库优点

1. 共享库节省了系统的资源。
2. 可以通过修改环境变量，使得我们可以使用一个可替换的共享函数库。
3. 可以编写程序装载动态库，在连接的时候，不需要任何预先的安排。

## 第四部分 附录 (Appendix)

该部分资料是关于服务器配置的。不全，但是比较实用。

- 如何在 Apache 中设置基于 IP 的虚拟主机服务？这里我们假定两个 IP 地址值分别为 192.168.1.11,192.168.1.12，并且在每个虚拟主机中配置不同的 DocumentRoot,ServerAdmin,ServerName 等信息，然后设置每个虚拟主机的监听方式(在 80 还是 8080 端口监听)。

解决办法：

- 打开配置文件/etc/httpd/conf/httpd.conf  
加入  

```
<VirtualHost 192.168.1.x>
    DocumentRoot "/var/www/VirtualDir"
    ServerAdmin admin@localhost.com
    ServerName 192.168.1.x
</VirtualHost>
    Listen 192.168.1.11:80
    Listen 192.168.1.12:8080
```
- 创建上述/var/www/ VirtualDir 目录
- 加入相关测试文件\*.html3.通过浏览器进行测试访问 <http://192.168.1.x>

- Linux 下的防火墙设置，

要求

- 1) 设定一个用户链
- 2) 记录所有通过 23 端口的所有数据包，并用良好的可识别的格式记
- 3) 查看日志中记录的信息

解决办法：

- iptables -N TELNET
- iptables -A TELNET -j LOG --log-tcp-options --log-ip-options --log-prefix "[IPTABLES Telnet]:"
- iptables -A LOG\_DROP -j ACCEPT
- iptables -A INPUT -p tcp --dport 23 -j LOG\_DROP
- iptables -A OUTPUT -p tcp --sport 23 -j LOG\_DROP

- 配置 Linux 下的 DHCP，完成如下功能：

1. 指定默认租期时间为 1 天
2. 指定当前分配的 IP 地址属于 C 类网
3. 指定发布网关和 DNS 给客户机  
网关：192.168.1.1  
DNS：210.34.48.34
4. 指定子网为 192.168.1.0，IP 范围为  
192.168.1.1 到 192.168.1.100

- 5. 固定某台主机的 IP 地址
- 6. 指定 DHCP 的监听接口为 eth0
- 7. 使用 windows 和 Linux 主机作为客户机进行 DHCP 测试

解决办法：

- 打开并编辑 dhcpcd 的配置文件/etc/dhcpcd.conf
 

```
default-lease-time 86400;
option subnet-mask 255.255.255.0;
option routers 192.168.1.1;
option domain-name-servers 210.34.48.34;
subnet 192.168.1.0 netmask 255.255.255.0 {
    range 192.168.1.1 192.168.1.100;
}
host pc1{
    hardware ethernet xx:xx:xx:xx:xx:xx
    fixed-address 192.168.1.1;
}
```
- 创建 dhcpcd.leases
 

```
touch /var/dhcp/dhcpcd.leases
```
- 启用 DHCP 服务进程
 

```
/usr/sbin/dhcpcd eth0
```
- 进行测试

- 扫描 210.29.88.0 网络,看看哪些主机是上线的,并针对其中一台上线的主机进行具体的探测,以便确定该主机提供的服务和开放的端口

解决办法：

- nmap -sP 210.29.88.0/24
- nmap 210.29.88.1

- 监听并捕获 20 个来自主机 192.168.0.1 和 192.168.0.2 的所有基于 tcp 协议端口为 21, 20 的所有数据包, 并保存到 test.dump 文件中, 然后使用 tcpdump 打开并阅读

解决办法：

- tcpdump hots 192.168.0.1 and host 192.168.0.2 and tcp and port 21 -c 20 -w test.dump
- tcpdump -r test.dump

- 列出系统上可用的活跃的 TCP 和 UDP 网络连接; 在此基础上列出是哪些进程打开或者连接这些网络接口; 启动系统地 ftp 服务器进程, 列出与 ftp 相关的所有网络连接

解决办法：

- netstat -t -u
- netstat -t -u -p
- netstat -t -u -p | grep ftp

- 发送 5 个 ICMP 包给 218.193.118.79, 描述反馈信息的具体含义; 增加每个包的数据长度到 1024, 然后再发 5 个包到上述 IP; 增加包的数量到 100, 但是要求不需要返回所

有信息行，只需要头尾几行和统计信息

解决办法：

- ping -c 5 218.193.118.79
- ping -c 5 -s 1024 218.193.118.79
- ping -c 100 -q 218.193.118.79

- 限制系统用户 linux 远程登录，限制根用户从 tty2 登录，从远程任何地方登陆

解决办法：

- 打开编辑/etc/securetty
- 注释其中的 tty2 所在行，这样可以限制 root 用户在 tty2 的登录
- 编辑/etc/security/access.conf
- 添加-:linux:EXCEPT LOCAL
- 编辑/etc/pam.d/login
- 增加 account required /lib/security/pam\_access.so

- 配置 SSH 服务，以便 root 用户可以登录，允许本地网(192.168.1.0)主机可以连接该服务并使用 Linux 下的 scp,sftp 等工具进行文件传输；使用 Windows 下的 putty.exe,scp.exe,sftp.exe 等工具进行文件传输。

解决办法：

- 打开编辑/etc/ssh/sshd\_config
- Port 22, Protocol 2,1, PermitRootLogin yes
- 修改/etc/hosts.allow
- sshd:192.168.1.
- service sshd start
- ssh root@SSH\_IP

- 配置 Samba 服务器，让每台主机有不同的连接共享目录，针对主机名提供不同的连接共享目录，使其具有写的权限，匿名用户不能访问。也就是说，主机 ABC 会连接到 SMB 服务器指定的/usr/remotepc/ABC 目录上。使用 smbclient 进行连接并下载其中的一个文件。

解决办法：

- 创建/usr/remotepc 目录和相应的主机名子目录
- 修改 smb.conf 文件，增加相应的段
- [pchome]
 

```
comment = Remote PC Directories
path = /usr/remotepc/%m
public = no
writable = yes
```

- 限定一个内部网IP在访问任何一个地址的时候访问的都是某个指定的网站主页呢？所有主页访问转向到www.baidu.com <--> 202.108.22.5

例如：IP 192.168.1.2

输入：<http://www.google.com/>

访问：<http://www.baidu.com/>

解决办法：

- `iptables -t nat -A PREROUTING -s 192.168.1.2 -p tcp --dport 80 -j DNAT --to 202.108.22.5:80`
- `iptables -t nat -A POSTROUTING -s 192.168.1.2 -p tcp --dport 80 -j SNAT --to 218.193.118.x`

- 利用 iptables 实现内外互通，假定当前网络为 218.193.118.0，网关为 218.193.118.254，本机 IP 为 218.193.118.x，192.168.1.1，内部网为 192.168.1.x  
192.168.1.11 为内网 Web 站点，试做好定向访问。

解决办法：

- `ifconfig eth0 218.193.118.x`
- `route add default gw 218.193.118.254`
- `ifconfig eth0:0 192.168.1.1`
- `iptables -t nat -A PREROUTING -d 218.193.118.x -p tcp --dport http -j DNAT --to 192.168.1.11`
- `iptables -t nat -A PREROUTING -d 218.193.118.x -j DNAT --to 192.168.1.1`
- `iptables -t nat -A POSTROUTING -s 192.168.1.0/24 -j SNAT --to 218.193.118.x`

## 第五部分 参考资料 (Reference)

<http://www.kernel.org>

<http://www.oldlinux.org>

<http://www.linuxsir.org>

<http://www.linuxforum.net>

<http://www.joyfire.net>

<http://www-128.ibm.com/developerworks/cn/linux>

谭浩强 《C 程序设计》 清华大学出版社

沈美明 温冬婵 《IBM-PC 汇编语言程序设计 (第 2 版)》 清华大学出版社

杨季文 《80X86 汇编语言程序设计教程》 清华大学出版社

毛德操 胡希明 《Linux 内核源代码情景分析》 浙江大学出版社

赵炯 《Linux 内核完全注释》 机械工业出版社

Randal E.Bryant David R.O'Hallaron 《Computer Systems A Programmer's Perspective (Beta Draft)》

Maurice J.Bach UNIX 《操作系统设计与实现》 陈葆玉等译 机械工业出版社

Andrew S. Tanenbaum 《操作系统：设计于实现》 尤晋元等译 电子工业出版社

Alessandro Rubini,Jonathan 《Linux 设备驱动程序》 魏永明等译 中国电力出版社

Evi Nemeth, Garth Snyder, Trent R. Hein 《Linux 系统管理技术手册》 张辉译 人民邮电出版社

Warren W.Gay 《Linux 编程 24 小时教程》 机械工业出版社

INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986