同济大学

Linux 操作系统实现原理

内核程序集

赵炯

2017/9/18

Linux 0.12 版内核已注释程序集

目录

第5章 内核编译批处理程序

5.1 程序 5-1 linux/Makefile 文件

```
#
# if you want the ram-disk device, define this to be the
# size in blocks.
#
# 如果你要使用 RAM 盘(RAMDISK)设备的话就定义块的大小。这里默认 RAMDISK 没有定义（注释掉了），
# 否则 gcc 编译时会带有选项'-DRAMDISK=512'，参见第 13 行。
RAMDISK = #-DRAMDISK=512

AS86    =as86 -0 -a         # 8086 汇编编译器和连接器，见列表后的介绍。后带的参数含义分别
LD86    =ld86 -0            # 是：-0 生成 8086 目标程序；-a 生成与 gas 和 gld 部分兼容的代码。

AS      =gas                # GNU 汇编编译器和连接器，见列表后的介绍。
LD      =gld
# 下面是 GNU 链接器 gld 运行时用到的选项。含义是：-s 输出文件中省略所有的符号信息；-x 删除
# 所有局部符号；-M 表示需要在标准输出设备(显示器)上打印连接映像(link map)，是指由连接程序
# 产生的一种内存地址映像，其中列出了程序段装入到内存中的位置信息。具体来讲有如下信息：
# • 目标文件及符号信息映射到内存中的位置；
# • 公共符号如何放置；
# • 连接中包含的所有文件成员及其引用的符号。
LDFLAGS =-s -x -M

# gcc 是 GNU C 程序编译器。对于 UNIX 类的脚本(script)程序而言，在引用定义的标识符时，需在前
# 面加上$符号并用括号括住标识符。
CC      =gcc $(RAMDISK)

# 下面指定 gcc 使用的选项。前一行最后的'\'符号表示下一行是续行。选项含义为：-Wall 打印所有
# 警告信息；-O 对代码进行优化。'-f 标志'指定与机器无关的编译标志。其中-fstrength-reduce 用
# 于优化循环语句；-fcombine-regs 用于指明编译器在组合编译阶段把复制一个寄存器到另一个寄存
# 器的指令组合在一起。-fomit-frame-pointer 指明对于无需帧指针（Frame pointer）的函数不要
# 把帧指针保留在寄存器中。这样在函数中可以避免对帧指针的操作和维护。-mstring-insns 是
# Linus 在学习 gcc 编译器时为 gcc 增加的选项，用于 gcc-1.40 在复制结构等操作时使用 386 CPU 的
# 字符串指令，可以去掉。
CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer \
-fcombine-regs -mstring-insns

# 下面 cpp 是 gcc 的前(预)处理器程序。前处理器用于进行程序中的宏替换处理、条件编译处理以及
# 包含进指定文件的内容，即把使用'#include'指定的文件包含进来。源程序文件中所有以符号'#'
# 开始的行均需要由前处理器进行处理。程序中所有'#define'定义的宏都会使用其定义部分替换掉。
# 程序中所有'#if'、'#ifdef'、'#ifndef'和'#endif'等条件判别行用于确定是否包含其指定范围中
# 的语句。
# '-nostdinc -Iinclude'含义是不要搜索标准头文件目录中的文件，即不用系统/usr/include/目录
```

```
# 下的头文件，而是使用'-I'选项指定的目录或者是在当前目录里搜索头文件。
CPP     =cpp -nostdinc -Iinclude

#
# ROOT_DEV specifies the default root-device when making the image.
# This can be either FLOPPY, /dev/xxxx or empty, in which case the
# default of /dev/hd6 is used by 'build'.
#
# ROOT_DEV 指定在创建内核映像(image)文件时所使用的默认根文件系统所
# 在的设备，这可以是软盘(FLOPPY)、/dev/xxxx 或者干脆空着，空着时
# build 程序（在 tools/目录中）就使用默认值/dev/hd6。
#
# 这里/dev/hd6 对应第 2 个硬盘的第 1 个分区。这是 Linus 开发 Linux 内核时自己的机器上根
# 文件系统所在的分区位置。/dev/hd2 表示把第 1 个硬盘的第 2 个分区用作交换分区。
ROOT_DEV=/dev/hd6
SWAP_DEV=/dev/hd2

# 下面是 kernel 目录、mm 目录和 fs 目录所产生的目标代码文件。为了方便引用在这里将它们用
# ARCHIVES（归档文件）标识符表示。
ARCHIVES=kernel/kernel.o mm/mm.o fs/fs.o

# 块和字符设备库文件。'.a'表示该文件是个归档文件，也即包含有许多可执行二进制代码子程序
# 集合的库文件，通常是用 GNU 的 ar 程序生成。ar 是 GNU 的二进制文件处理程序，用于创建、修改
# 以及从归档文件中抽取文件。
DRIVERS =kernel/blk_drv/blk_drv.a kernel/chr_drv/chr_drv.a
MATH    =kernel/math/math.a        # 数学运算库文件。
LIBS    =lib/lib.a                 # 由 lib/目录中的文件所编译生成的通用库文件。

# 下面是 make 老式的隐式后缀规则。该行指示 make 利用下面的命令将所有的'.c'文件编译生成'.s'
# 汇编程序。':'表示下面是该规则的命令。整句表示让 gcc 采用前面 CFLAGS 所指定的选项以及仅使
# 用 include/目录中的头文件，在适当地编译后不进行汇编就停止（-S），从而产生与输入的各个 C
# 文件对应的汇编语言形式的代码文件。默认情况下所产生的汇编程序文件是原 C 文件名去掉'.c'后
# 再加上'.s'后缀。'-o'表示其后是输出文件的形式。其中'$*.s'（或'$@'）是自动目标变量，'$<'
# 代表第一个先决条件，这里即是符合条件'*.c'的文件。
# 下面这 3 个不同规则分别用于不同的操作要求。若目标是.s 文件，而源文件是.c 文件则会使
# 用第一个规则；若目录是.o，而原文件是.s，则使用第 2 个规则；若目标是.o 文件而原文件
# 是 c 文件，则可直接使用第 3 个规则。
.c.s:
        $(CC) $(CFLAGS) \
        -nostdinc -Iinclude -S -o $*.s $<

# 表示将所有.s 汇编程序文件编译成.o 目标文件。 整句表示使用 gas 编译器将汇编程序编译成.o
# 目标文件。-c 表示只编译或汇编，但不进行连接操作。
.s.o:
        $(AS) -c -o $*.o $<
# 类似上面，*.c 文件-→*.o 目标文件。整句表示使用 gcc 将 C 语言文件编译成目标文件但不连接。
.c.o:
        $(CC) $(CFLAGS) \
        -nostdinc -Iinclude -c -o $*.o $<


# 下面'all'表示创建 Makefile 所知的最顶层的目标。这里即是 Image 文件。这里生成的 Image 文件
# 即是引导启动盘映像文件 bootimage。若将其写入软盘就可以使用该软盘引导 Linux 系统了。在
```

```
# Linux 下将 Image 写入软盘的命令参见 46 行。DOS 系统下可以使用软件 rawrite.exe。
all:    Image

# 说明目标（Image 文件）是由冒号后面的 4 个元素产生，分别是 boot/目录中的 bootsect 和 setup
# 文件、tools/目录中的 system 和 build 文件。42--43 行这是执行的命令。42 行表示使用 tools 目
# 录下的 build 工具程序（下面会说明如何生成）将 bootsect、setup 和 system 文件以$(ROOT_DEV)
# 为根文件系统设备组装成内核映像文件 Image。第 43 行的 sync 同步命令是迫使缓冲块数据立即写盘
# 并更新超级块。
Image: boot/bootsect boot/setup tools/system tools/build
        tools/build boot/bootsect boot/setup tools/system $(ROOT_DEV) \
                $(SWAP_DEV) > Image
        sync

# 表示 disk 这个目标要由 Image 产生。dd 为 UNIX 标准命令：复制一个文件，根据选项进行转换和格
# 式化。bs=表示一次读/写的字节数。if=表示输入的文件，of=表示输出到的文件。这里/dev/PS0 是
# 指第一个软盘驱动器(设备文件)。在 Linux 系统下使用/dev/fd0。
disk: Image
        dd bs=8192 if=Image of=/dev/PS0

tools/build: tools/build.c              # 由 tools 目录下的 build.c 程序生成执行程序 build。
        $(CC) $(CFLAGS) \
        -o tools/build tools/build.c    # 编译生成执行程序 build 的命令。

boot/head.o: boot/head.s                # 利用上面给出的.s.o 规则生成 head.o 目标文件。

# 表示 tools 目录中的 system 文件要由冒号右边所列的元素生成。56--61 行是生成 system 的命令。
# 最后的 > System.map 表示 gld 需要将连接映像重定向存放在 System.map 文件中。
# 关于 System.map 文件的用途参见注释后的说明。
tools/system:   boot/head.o init/main.o \
                $(ARCHIVES) $(DRIVERS) $(MATH) $(LIBS)
        $(LD) $(LDFLAGS) boot/head.o init/main.o \
        $(ARCHIVES) \
        $(DRIVERS) \
        $(MATH) \
        $(LIBS) \
        -o tools/system > System.map

# 数学协处理函数文件 math.a 由 64 行上的命令实现：进入 kernel/math/目录；运行 make 工具程序。
kernel/math/math.a:
        (cd kernel/math; make)

kernel/blk_drv/blk_drv.a:           # 生成块设备库文件 blk_drv.a，其中含有可重定位目标文件。
        (cd kernel/blk_drv; make)

kernel/chr_drv/chr_drv.a:           # 生成字符设备函数文件 chr_drv.a。
        (cd kernel/chr_drv; make)

kernel/kernel.o:                    # 内核目标模块 kernel.o
        (cd kernel; make)

mm/mm.o:                            # 内存管理模块 mm.o
        (cd mm; make)

```

```
80 fs/fs.o:                             # 文件系统目标模块 fs.o
81         (cd fs; make)
82
83 lib/lib.a:                           # 库函数 lib.a
84         (cd lib; make)
85
86 boot/setup: boot/setup.s                          # 这里开始的三行是使用 8086 汇编和连接器
87         $(AS86) -o boot/setup.o boot/setup.s     # 对 setup.s 文件进行编译生成 setup 文件。
88         $(LD86) -s -o boot/setup boot/setup.o    # -s 选项表示要去除目标文件中的符号信息。
89
90 boot/setup.s:   boot/setup.S include/linux/config.h       # 执行 C 语言预处理，替换*.S 文
91         $(CPP) -traditional boot/setup.S -o boot/setup.s  # 件中的宏生成对应的*.s 文件。
92
93 boot/bootsect.s:        boot/bootsect.S include/linux/config.h
94         $(CPP) -traditional boot/bootsect.S -o boot/bootsect.s
95
96 boot/bootsect:  boot/bootsect.s                  # 同上。生成 bootsect.o 磁盘引导块。
97         $(AS86) -o boot/bootsect.o boot/bootsect.s
98         $(LD86) -s -o boot/bootsect boot/bootsect.o
99
   # 当执行'make clean'时，就会执行 98--103 行上的命令，去除所有编译连接生成的文件。
   # 'rm'是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
100 clean:
101         rm -f Image System.map tmp_make core boot/bootsect boot/setup \
102                 boot/bootsect.s boot/setup.s
103         rm -f init/*.o tools/system tools/build boot/*.o
104         (cd mm;make clean)       # 进入 mm/目录；执行该目录 Makefile 文件中的 clean 规则。
105         (cd fs;make clean)
106         (cd kernel;make clean)
107         (cd lib;make clean)
108
   # 该规则将首先执行上面的 clean 规则，然后对 linux/目录进行压缩，生成'backup.Z'压缩文件。
   # 'cd .. '表示退到 linux/的上一级（父）目录；'tar cf - linux'表示对 linux/目录执行 tar 归档
   # 程序。'-cf'表示需要创建新的归档文件 '| compress -'表示将 tar 程序的执行通过管道操作('|')
   # 传递给压缩程序 compress，并将压缩程序的输出存成 backup.Z 文件。
109 backup: clean
110         (cd .. ; tar cf - linux | compress - > backup.Z)
111         sync                                     # 迫使缓冲块数据立即写盘并更新磁盘超级块。
112
113 dep:
   # 该目标或规则用于产生各文件之间的依赖关系。创建这些依赖关系是为了让 make 命令用它们来确定
   # 是否需要重建一个目标对象。比如当某个头文件被改动过后，make 就能通过生成的依赖关系，重新
   # 编译与该头文件有关的所有*.c 文件。具体方法如下：
   # 使用字符串编辑程序 sed 对 Makefile 文件（这里即是本文件）进行处理，输出为删除了 Makefile
   # 文件中'### Dependencies'行后面的所有行，即删除了下面从 122 开始到文件末的所有行，并生成
   # 一个临时文件 tmp_make（也即 114 行的作用）。然后对指定目录下（init/）的每一个 C 文件（其实
   # 只有一个文件 main.c）执行 gcc 预处理操作。标志'-M'告诉预处理程序 cpp 输出描述每个目标文件
   # 相关性的规则，并且这些规则符合 make 语法。对于每一个源文件，预处理程序会输出一个规则，其
   # 结果形式就是相应源程序文件的目标文件名加上其依赖关系，即该源文件中包含的所有头文件列表。
   # 然后把预处理结果都添加到临时文件 tmp_make 中，最后将该临时文件复制成新的 Makefile 文件。
   # 115 行上的'$$i'实际上是'$($i) '。这里'$i'是这句前面的 shell 变量'i'的值。
114         sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
115         (for i in init/*.c;do echo -n "init/";$(CPP) -M $$i;done) >> tmp_make
```

```
116         cp tmp_make Makefile
117         (cd fs; make dep)                   # 对 fs/目录下的 Makefile 文件也作同样的处理。
118         (cd kernel; make dep)
119         (cd mm; make dep)
120
121 ### Dependencies:
122 init/main.o : init/main.c include/unistd.h include/sys/stat.h \
123   include/sys/types.h include/sys/time.h include/time.h include/sys/times.h \
124   include/sys/utsname.h include/sys/param.h include/sys/resource.h \
125   include/utime.h include/linux/tty.h include/termios.h include/linux/sched.h \
126   include/linux/head.h include/linux/fs.h include/linux/mm.h \
127   include/linux/kernel.h include/signal.h include/asm/system.h \
128   include/asm/io.h include/stddef.h include/stdarg.h include/fcntl.h \
129   include/string.h
```

第6章 引导启动程序

6.1 程序 6 -1 linux/boot/bootsect.S

```
1 !
2 ! SYS_SIZE is the number of clicks (16 bytes) to be loaded.
3 ! 0x3000 is 0x30000 bytes = 196kB, more than enough for current
4 ! versions of linux
  ! SYS_SIZE 是要加载的系统模块长度，单位是节，每节 16 字节。0x3000 共为 0x30000 字节=196KB。
  ！若以 1024 字节为 1KB 计，则应该是 192KB。对于当前内核版本这个空间长度已足够了。当该值为
  ! 0x8000 时，表示内核最大为 512KB。因为内存 0x90000 处开始存放移动后的 bootsect 和 setup
  ！的代码，因此该值最大不得超过 0x9000（表示 584KB）。
  ！这里感叹号'!'或分号';'表示程序注释语句开始。
5 !
  ！头文件 linux/config.h 中定义了内核用到的一些常数符号和 Linus 自己使用的默认硬盘参数块。
  ！例如其中定义了以下一些常数：
  ! DEF_SYSSIZE   = 0x3000 -  默认系统模块长度。单位是节，每节为 16 字节；
  ! DEF_INITSEG   = 0x9000 -  默认本程序代码移动目的段位置；
  ! DEF_SETUPSEG = 0x9020 -  默认 setup 程序代码段位置；
  ! DEF_SYSSEG    = 0x1000 -  默认从磁盘加载系统模块到内存的段位置。
6 #include <linux/config.h>
7 SYSSIZE = DEF_SYSSIZE                  ！定义一个标号或符号。指明编译连接后 system 模块的大小。
8 !
9 !           bootsect.s                  (C) 1991 Linus Torvalds
10 !          modified by Drew Eckhardt
11 !
12 ! bootsect.s is loaded at 0x7c00 by the bios-startup routines, and moves
13 ! iself out of the way to address 0x90000, and jumps there.
14 !
15 ! It then loads 'setup' directly after itself (0x90200), and the system
16 ! at 0x10000, using BIOS interrupts.
17 !
18 ! NOTE! currently system is at most 8*65536 bytes long. This should be no
19 ! problem, even in the future. I want to keep it simple. This 512 kB
20 ! kernel size should be enough, especially as this doesn't contain the
21 ! buffer cache as in minix
22 !
23 ! The loader has been made as simple as possible, and continuos
24 ! read errors will result in a unbreakable loop. Reboot by hand. It
```

```
25 ! loads pretty fast by getting whole sectors at a time whenever possible.
   !
   ! 以下是前面文字的译文：
   !       bootsect.s                      (C) 1991 Linus Torvalds
   !       Drew Eckhardt 修改
   !
   ! bootsect.s 被 ROM BIOS 启动子程序加载至 0x7c00 (31KB)处，并将自己移到了地址 0x90000
   ! (576KB)处，并跳转至那里。
   !
   ! 它然后使用 BIOS 中断将'setup'直接加载到自己的后面(0x90200)(576.5KB)，并将 system 加
   ! 载到地址 0x10000 处。
   !
   ! 注意! 目前的内核系统最大长度限制为(8*65536)(512KB)字节，即使是在将来这也应该没有问
   ! 题的。我想让它保持简单明了。这样 512KB 的最大内核长度应该足够了，尤其是这里没有象
   ! MINIX 中一样包含缓冲区高速缓冲。
   !
   ! 加载程序已经做得够简单了，所以持续地读操作出错将导致死循环。只能手工重启。只要可能，
   ! 通过一次读取所有的扇区，加载过程可以做得很快。
26
   ! 伪指令（伪操作符）.globl 或.global 用于定义随后的标识符是外部的或全局的，并且即使不
   ! 使用也强制引入。 .text、.data 和.bss 用于分别定义当前代码段、数据段和未初始化数据段。
   ! 在链接多个目标模块时，链接程序（ld86）会根据它们的类别把各个目标模块中的相应段分别
   ! 组合（合并）在一起。这里把三个段都定义在同一重叠地址范围中，因此本程序实际上不分段。
   ! 另外，后面带冒号的字符串是标号，例如下面的'begtext:'。
   ! 一条汇编语句通常由标号（可选）、指令助记符（指令名）和操作数三个字段组成。标号位于
   ! 一条指令的第一个字段。它代表其所在位置的地址，通常指明一个跳转指令的目标位置。
27 .globl begtext, begdata, begbss, endtext, enddata, endbss
28 .text                                   ! 文本段（代码段）。
29 begtext:
30 .data                                   ! 数据段。
31 begdata:
32 .bss                                    ! 未初始化数据段。
33 begbss:
34 .text                                   ! 文本段（代码段）。
35
   ! 下面等号'='或符号'EQU'用于定义标识符或标号所代表的值。
36 SETUPLEN = 4                             ! nr of setup-sectors
                                           ! setup 程序代码占用磁盘扇区数(setup-sectors)值；
37 BOOTSEG  = 0x07c0                       ! original address of boot-sector
                                           ! bootsect 代码所在内存原始段地址；
38 INITSEG  = DEF_INITSEG                  ! we move boot here - out of the way
                                           ! 将 bootsect 移到位置 0x90000 - 避开系统模块占用处；
39 SETUPSEG = DEF_SETUPSEG                 ! setup starts here
                                           ! setup 程序从内存 0x90200 处开始；
```

```
40 SYSSEG     = DEF_SYSSEG                  ! system loaded at 0x10000 (65536).
                                           ! system 模块加载到 0x10000（64 KB）处；
41 ENDSEG     = SYSSEG + SYSSIZE            ! where to stop loading
                                           ！ 停止加载的段地址；
42
43 ! ROOT_DEV & SWAP_DEV are now written by "build".
   ！ 根文件系统设备号 ROOT_DEV 和交换设备号 SWAP_DEV 现在由 tools 目录下的 build 程序写入。
   ！ 设备号 0x306 指定根文件系统设备是第 2 个硬盘的第 1 个分区。当年 Linus 是在第 2 个硬盘上
   ！ 安装了 Linux 0.11 系统，所以这里 ROOT_DEV 被设置为 0x306。在编译这个内核时你可以根据
   ！ 自己根文件系统所在设备位置修改这个设备号。这个设备号是 Linux 系统老式的硬盘设备号命
   ！ 名方式，硬盘设备号具体值的含义如下：
   ！ 设备号=主设备号*256 + 次设备号（也即 dev_no = (major<<8) + minor ）
   ！ （主设备号：1-内存,2-磁盘,3-硬盘,4-ttyx,5-tty,6-并行口,7-非命名管道）
   ! 0x300 - /dev/hd0 - 代表整个第 1 个硬盘；
   ! 0x301 - /dev/hd1 - 第 1 个盘的第 1 个分区；
   ！ …
   ! 0x304 - /dev/hd4 - 第 1 个盘的第 4 个分区；
   ! 0x305 - /dev/hd5 - 代表整个第 2 个硬盘；
   ! 0x306 - /dev/hd6 - 第 2 个盘的第 1 个分区；
   ！ …
   ! 0x309 - /dev/hd9 - 第 2 个盘的第 4 个分区；
   ！ 从 Linux 内核 0.95 版后就已经使用与现在内核相同的命名方法了。
44 ROOT_DEV = 0                          ！ 根文件系统设备使用与系统引导时同样的设备；
45 SWAP_DEV = 0                          ！ 交换设备使用与系统引导时同样的设备；
46
   ！ 伪指令 entry 迫使链接程序在生成的执行程序（a.out）中包含指定的标识符或标号。这里是
   ！ 程序执行开始点。49 -- 58 行作用是将自身(bootsect)从目前段位置 0x07c0(31KB) 移动到
   ! 0x9000(576KB) 处，共 256 字（512 字节），然后跳转到移动后代码的 go 标号处，也即本程
   ！ 序的下一语句处。
47 entry start                           ！ 告知链接程序，程序从 start 标号开始执行。
48 start:
49          mov     ax,#BOOTSEG          ！ 将 ds 段寄存器置为 0x7C0；
50          mov     ds,ax
51          mov     ax,#INITSEG          ！ 将 es 段寄存器置为 0x9000；
52          mov     es,ax
53          mov     cx,#256              ！ 设置移动计数值=256 字（512 字节）；
54          sub     si,si                ！ 源地址    ds:si = 0x07C0:0x0000
55          sub     di,di                ！ 目的地址  es:di = 0x9000:0x0000
56          rep                          ！ 重复执行并递减 cx 的值，直到 cx = 0 为止。
57          movw                         ！ 即 movs 指令。从内存[si]处移动 cx 个字到[di]处。
58          jmpi    go,INITSEG           ！ 段间跳转（Jump Intersegment）。这里 INITSEG
                                         ！ 指出跳转到的段地址，标号 go 是段内偏移地址。
59
   ！ 从下面开始，CPU 在已移动到 0x90000 位置处的代码中执行。
```

```
! 这段代码设置几个段寄存器，包括栈寄存器 ss 和 sp。栈指针 sp 只要指向远大于 512 字节偏移
! （即地址 0x90200）处都可以。因为从 0x90200 地址开始处还要放置 setup 程序，而此时 setup
! 程序大约为 4 个扇区，因此 sp 要指向大于（0x200 + 0x200 * 4 +堆栈大小）位置处。这里 sp
! 设置为 0x9ff00 - 12（参数表长度），即 sp = 0xfef4。在此之上位置会存放一个自建的驱动
! 器参数表，见下面说明。实际上 BIOS 把引导扇区加载到 0x7c00 处并把执行权交给引导程序时，
! ss = 0x00，sp = 0xfffe。
! 另外，第 65 行上 push 指令的期望作用是想暂时把段值保留在栈中，然后等下面执行完判断磁道
! 扇区数后再弹出栈，并给段寄存器 fs 和 gs 赋值（第 109 行）。但是由于第 67、68 两语句修改
! 了栈段的位置，因此除非在执行栈弹出操作之前把栈段恢复到原位置，否则这样设计就是错误的。
! 因此这里存在一个 bug。改正的方法之一是去掉第 65 行，并把第 109 行修改成“mov ax,cs”。
60 go:     mov     ax,cs                   ! 将 ds、es 和 ss 都置成移动后代码所在的段处(0x9000)。
61         mov     dx,#0xfef4              ! arbitrary value >>512 - disk parm size
62
63         mov     ds,ax
64         mov     es,ax
65         push    ax                      ! 临时保存段值（0x9000），供 109 行使用。（滑头!）
66
67         mov     ss,ax                   ! put stack at 0x9ff00 - 12.
68         mov     sp,dx
69 /*
70  *      Many BIOS's default disk parameter tables will not
71  *      recognize multi-sector reads beyond the maximum sector number
72  *      specified in the default diskette parameter tables - this may
73  *      mean 7 sectors in some cases.
74  *
75  *      Since single sector reads are slow and out of the question,
76  *      we must take care of this by creating new parameter tables
77  *      (for the first disk) in RAM.    We will set the maximum sector
78  *      count to 18 - the most we will encounter on an HD 1.44.
79  *
80  *      High doesn't hurt.    Low does.
81  *
82  *      Segments are as follows: ds=es=ss=cs - INITSEG,
83  *              fs = 0, gs = parameter table segment
84  */
   /*
    *      对于多扇区读操作所读的扇区数超过默认磁盘参数表中指定的最大扇区数时，
    *      很多 BIOS 将不能进行正确识别。在某些情况下是 7 个扇区。
    *
    *      由于单扇区读操作太慢，不予以考虑，因此我们必须通过在内存中重创建新的
    *      参数表（为第 1 个驱动器）来解决这个问题。我们将把其中最大扇区数设置为
    *      18 -- 即在 1.44MB 磁盘上会碰到的最大数值。
    *
    *      这个数值大了不会出问题，但是太小就不行了。
```

```
 *
 *      段寄存器将被设置成：ds=es=ss=cs - 都为 INITSEG（0x9000），
 *      fs = 0，gs = 参数表所在段值。
 */
! BIOS 设置的中断 0x1E 的中断向量值是软驱参数表地址。该向量值位于内存 0x1E * 4 = 0x78
! 处。这段代码首先从内存 0x0000:0x0078 处复制原软驱参数表到 0x9000:0xfef4 处，然后修改
! 表中的每磁道最大扇区数为 18。

        push    #0                      ! 置段寄存器 fs = 0。
        pop     fs                      ! fs:bx 指向存有软驱参数表地址处（指针的指针）。
        mov     bx,#0x78                ! fs:bx is parameter table address
! 下面指令表示下一条语句的操作数在 fs 段寄存器所指的段中。它只影响其下一条语句。这里
! 把 fs:bx 所指内存位置处的表地址放到寄存器对 gs:si 中作为原地址。寄存器对 es:di =
! 0x9000:0xfef4 为目的地址。
        seg fs
        lgs     si,(bx)                 ! gs:si is source

        mov     di,dx                   ! es:di is destination ! dx=0xfef4，在 61 行被设置。
        mov     cx,#6                   ! copy 12 bytes
        cld                             ! 清方向标志。复制时指针递增。

        rep                             ! 复制 12 字节的软驱参数表到 0x9000:0xfef4 处。
        seg gs
        movw

        mov     di,dx                   ! es:di 指向新表，修改表中偏移 4 处的最大扇区数为 18。
        movb    4(di),*18               ! patch sector count

        seg fs                          ! 让中断向量 0x1E 的值指向新表。
        mov     (bx),di
        seg fs
        mov     2(bx),es

        pop     ax                      ! 此时 ax 中是上面第 65 行保留下来的段值（0x9000）。
        mov     fs,ax                   ! 设置 fs = gs = 0x9000。
        mov     gs,ax

        xor     ah,ah                   ! reset FDC ! 复位软盘控制器，让其采用新参数。
        xor     dl,dl                   ! dl = 0，第 1 个软驱。
        int     0x13

! load the setup-sectors directly after the bootblock.
! Note that 'es' is already set up.
! 在 bootsect 程序块后紧根着加载 setup 模块的代码数据。
```

```
    ! 注意 es 已经设置好了。(在移动代码时 es 已经指向目的段地址处 0x9000)。
119
    ! 121--137 行的用途是利用 ROM BIOS 中断 INT 0x13 将 setup 模块从磁盘第 2 个扇区开始读到
    ! 0x90200 开始处，共读 4 个扇区。在读操作过程中如果读出错，则显示磁盘上出错扇区位置，
    ! 然后复位驱动器并重试，没有退路。
    ! INT 0x13 读扇区使用调用参数设置如下：
    ! ah = 0x02 - 读磁盘扇区到内存；al = 需要读出的扇区数量；
    ! ch = 磁道(柱面)号的低 8 位；    cl = 开始扇区(位 0-5)，磁道号高 2 位(位 6-7)；
    ! dh = 磁头号；                   dl = 驱动器号（如果是硬盘则位 7 要置位)；
    ! es:bx →指向数据缓冲区；   如果出错则 CF 标志置位，ah 中是出错码。
120 load_setup:
121         xor     dx, dx                  ! drive 0, head 0
122         mov     cx,#0x0002              ! sector 2, track 0
123         mov     bx,#0x0200              ! address = 512, in INITSEG
124         mov     ax,#0x0200+SETUPLEN     ! service 2, nr of sectors
125         int     0x13                    ! read it
126         jnc     ok_load_setup           ! ok - continue
127
128         push    ax                      ! dump error code ! 显示出错信息。出错码入栈。
129         call    print_nl                ! 屏幕光标回车。
130         mov     bp, sp                  ! ss:bp 指向欲显示的字（word)。
131         call    print_hex               ! 显示十六进制值。
132         pop     ax
133
134         xor     dl, dl                  ! reset FDC ! 复位磁盘控制器，重试。
135         xor     ah, ah
136         int     0x13
137         j       load_setup              ! j 即 jmp 指令。
138
139 ok_load_setup:
140
141 ! Get disk drive parameters, specifically nr of sectors/track
    ! 这段代码取磁盘驱动器的参数，实际上是取每磁道扇区数，并保存在位置 sectors 处。
    ! 取磁盘驱动器参数 INT 0x13 调用格式和返回信息如下：
    ! ah = 0x08       dl = 驱动器号（如果是硬盘则要置位 7 为 1)。
    ! 返回信息：
    ! 如果出错则 CF 置位，并且 ah = 状态码。
    ! ah = 0，  al = 0，              bl = 驱动器类型（AT/PS2）
    ! ch = 最大磁道号的低 8 位，   cl = 每磁道最大扇区数(位 0-5)，最大磁道号高 2 位(位 6-7)
    ! dh = 最大磁头数，             dl = 驱动器数量，
    ! es:di -→ 软驱磁盘参数表。
142
143         xor     dl,dl
144         mov     ah,#0x08            ! AH=8 is get drive parameters
```

```
145             int     0x13
146             xor     ch,ch
    ! 下面指令表示下一条语句的操作数在 cs 段寄存器所指的段中。它只影响其下一条语句。实际
    ! 上，由于本程序代码和数据都被设置处于同一个段中，即段寄存器 cs 和 ds、es 的值相同，因
    ! 此本程序中此处可以不使用该指令。
147             seg cs
    ! 下句保存每磁道扇区数。对于软盘来说（dl=0），其最大磁道号不会超过 256，ch 已经足够表
    ! 示它，因此 cl 的位 6-7 肯定为 0。又 146 行已置 ch=0，因此此时 cx 中是每磁道扇区数。
148             mov     sectors,cx
149             mov     ax,#INITSEG
150             mov     es,ax           ! 因为上面取磁盘参数中断改了 es 值，这里重新改回。
151
152 ! Print some inane message
    ! 显示信息：“'Loading'+回车+换行”，共显示包括回车和换行控制字符在内的 9 个字符。
    ! BIOS 中断 0x10 功能号 ah = 0x03，读光标位置。
    ! 输入：bh = 页号
    ! 返回：ch = 扫描开始线；cl = 扫描结束线；dh = 行号(0x00 顶端)；dl = 列号(0x00 最左边)。
    !
    ! BIOS 中断 0x10 功能号 ah = 0x13，显示字符串。
    ! 输入：al = 放置光标的方式及规定属性。0x01-表示使用 bl 中的属性值，光标停在字符串结尾处。
    ! es:bp 此寄存器对指向要显示的字符串起始位置处。cx = 显示的字符串字符数。bh = 显示页面号；
    ! bl = 字符属性。dh = 行号；dl = 列号。
153
154             mov     ah,#0x03        ! read cursor pos
155             xor     bh,bh           ! 首先读光标位置。返回光标位置值在 dx 中。
156             int     0x10            ! dh - 行（0--24）；dl - 列(0--79)。
157
158             mov     cx,#9           ! 共显示 9 个字符。
159             mov     bx,#0x0007      ! page 0, attribute 7 (normal)
160             mov     bp,#msg1        ! es:bp 指向要显示的字符串。
161             mov     ax,#0x1301      ! write string, move cursor
162             int     0x10            ! 写字符串并移动光标到串结尾处。
163
164 ! ok, we've written the message, now
165 ! we want to load the system (at 0x10000)
    ! 现在开始将 system 模块加载到 0x10000（64KB）开始处。
166
167             mov     ax,#SYSSEG
168             mov     es,ax           ! segment of 0x010000 ! es = 存放 system 的段地址。
169             call    read_it         ! 读磁盘上 system 模块，es 为输入参数。
170             call    kill_motor      ! 关闭驱动器马达，这样就可以知道驱动器的状态了。
171             call    print_nl        ! 光标回车换行。
172
173 ! After that we check which root-device to use. If the device is
```

```
174 ! defined (!= 0), nothing is done and the given device is used.
175 ! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
176 ! on the number of sectors that the BIOS reports currently.
    ！此后，我们检查要使用哪个根文件系统设备（简称根设备）。如果已经指定了设备(!=0)，
    ！就直接使用给定的设备。否则就需要根据 BIOS 报告的每磁道扇区数来确定到底使用/dev/PS0
    ! (2,28)，还是 /dev/at0 (2,8)。
    !! 上面一行中两个设备文件的含义：
    !! 在 Linux 中软驱的主设备号是 2(参见第 43 行的注释)，次设备号 = type*4 + nr，其中
    !! nr 为 0-3 分别对应软驱 A、B、C 或 D；type 是软驱的类型（2→1.2MB 或 7→1.44MB 等）。
    !! 因为 7*4 + 0 = 28，所以 /dev/PS0 (2,28)指的是 1.44MB A 驱动器,其设备号是 0x021c
    !! 同理 /dev/at0 (2,8)指的是 1.2MB A 驱动器，其设备号是 0x0208。

    ！下面 root_dev 定义在引导扇区 508，509 字节处，指根文件系统所在设备号。0x0306 指第 2
    ！个硬盘第 1 个分区。这里默认为 0x0306 是因为当时 Linus 开发 Linux 系统时是在第 2 个硬
    ！盘第 1 个分区中存放根文件系统。这个值需要根据你自己根文件系统所在硬盘和分区进行修
    ！改。例如，如果你的根文件系统在第 1 个硬盘的第 1 个分区上，那么该值应该为 0x0301，即
    ！（0x01, 0x03）。如果根文件系统是在第 2 个 Bochs 软盘上，那么该值应该为 0x021D，即
    ！（0x1D,0x02）。当编译内核时，你可以在 Makefile 文件中另行指定你自己的值，内核映像
    ！文件 Image 的创建程序 tools/build 会使用你指定的值来设置你的根文件系统所在设备号。
177
178             seg cs
179             mov     ax,root_dev             ！取 508,509 字节处的根设备号并判断是否已被定义。
180             or      ax,ax
181             jne     root_defined
    ！取上面第 148 行保存的每磁道扇区数。如果 sectors=15 则说明是 1.2MB 的驱动器；如果
    ! sectors=18，则说明是 1.44MB 软驱。因为是可引导的驱动器，所以肯定是 A 驱。
182             seg cs
183             mov     bx,sectors
184             mov     ax,#0x0208              ! /dev/ps0 - 1.2Mb
185             cmp     bx,#15                  ！判断每磁道扇区数是否=15
186             je      root_defined            ！如果等于，则 ax 中就是引导驱动器的设备号。
187             mov     ax,#0x021c              ! /dev/PS0 - 1.44Mb
188             cmp     bx,#18
189             je      root_defined
190 undef_root:                                ！如果都不一样，则死循环（死机）。
191             jmp undef_root
192 root_defined:
193             seg cs
194             mov     root_dev,ax             ！将检查过的设备号保存到 root_dev 中。
195
196 ! after that (everyting loaded), we jump to
197 ! the setup-routine loaded directly after
198 ! the bootblock:
    ！到此，所有程序都加载完毕，我们就跳转到被加载在 bootsect 后面的 setup 程序去。
```

```
    ! 下面段间跳转指令（Jump Intersegment）。跳转到 0x9020:0000(setup.s 程序开始处)去执行。
199
200             jmpi    0,SETUPSEG              !!!! 到此本程序就结束了。!!!!

    ! 下面是几个子程序。read_it 用于读取磁盘上的 system 模块。kill_moter 用于关闭软驱马达。
    ! 还有一些屏幕显示子程序。
201
202 ! This routine loads the system at address 0x10000, making sure
203 ! no 64kB boundaries are crossed. We try to load it as fast as
204 ! possible, loading whole tracks whenever we can.
205 !
206 ! in:     es - starting address segment (normally 0x1000)
207 !
    ! 该子程序将系统模块加载到内存地址 0x10000 处，并确定没有跨越 64KB 的内存边界。
    ! 我们试图尽快地进行加载，只要可能，就每次加载整条磁道的数据。
    ! 输入：es – 开始内存地址段值（通常是 0x1000）
    !
    ! 下面伪操作符.word 定义一个 2 字节目标。相当于 C 语言程序中定义的变量和所占内存空间大小。
    ! '1+SETUPLEN'表示开始时已经读进 1 个引导扇区和 setup 程序所占的扇区数 SETUPLEN。
208 sread:  .word 1+SETUPLEN        ! sectors read of current track !当前磁道中已读扇区数。
209 head:   .word 0                 ! current head      !当前磁头号。
210 track:  .word 0                 ! current track     !当前磁道号。
211
212 read_it:
    ! 首先测试输入的段值。从盘上读入的数据必须存放在位于内存地址 64KB 的边界开始处，否则
    ! 进入死循环。清 bx 寄存器，用于表示当前段内存放数据的开始位置。
    ! 153 行上的指令 test 以比特位逻辑与两个操作数。若两个操作数对应的比特位都为 1，则结果
    ! 值的对应比特位为 1，否则为 0。该操作结果只影响标志（零标志 ZF 等）。例如若 AX=0x1000，
    ! 那么 test 指令的执行结果是(0x1000 & 0x0fff) = 0x0000，于是 ZF 标志置位。此时即下一条
    ! 指令 jne 条件不成立。
213             mov ax,es
214             test ax,#0x0fff
215 die:    jne die                 ! es must be at 64kB boundary ! es 值必须位于 64KB 边界!
216             xor bx,bx           ! bx is starting address within segment! bx 为段内偏移。
217 rp_read:
    ! 接着判断是否已经读入全部数据。比较当前所读段是否就是系统数据末端所处的段(#ENDSEG)，
    ! 如果不是就跳转至下面 ok1_read 标号处继续读数据。否则退出子程序返回。
218             mov ax,es
219             cmp ax,#ENDSEG          ! have we loaded all yet?   ! 是否已经加载了全部数据？
220             jb ok1_read
221             ret
222 ok1_read:
    ! 然后计算和验证当前磁道需要读取的扇区数，放在 ax 寄存器中。
    ! 根据当前磁道还未读取的扇区数以及段内数据字节开始偏移位置，计算如果全部读取这些未读
```

```
    ! 扇区，所读总字节数是否会超过 64KB 段长度的限制。若会超过，则根据此次最多能读入的字节
    ! 数 (64KB –段内偏移位置)，反算出此次需要读取的扇区数。
223             seg cs
224             mov ax,sectors          ! 取每磁道扇区数。
225             sub ax,sread            ! 减去当前磁道已读扇区数。
226             mov cx,ax               ! cx = ax = 当前磁道未读扇区数。
227             shl cx,#9               ! cx = cx * 512 字节 + 段内当前偏移值(bx)。
228             add cx,bx               ! = 此次读操作后，段内共读入的字节数。
229             jnc ok2_read            ! 若没有超过 64KB 字节，则跳转至 ok2_read 处执行。
230             je ok2_read
    ! 若加上此次将读磁道上所有未读扇区时会超过 64KB，则计算此时最多能读入的字节数：
    ! (64KB–段内读偏移位置)，再转换成需读取的扇区数。其中 0 减某数就是取该数 64KB 的补值。
231             xor ax,ax
232             sub ax,bx
233             shr ax,#9
234 ok2_read:
    ! 读当前磁道上指定开始扇区（cl）和需读扇区数（al）的数据到 es:bx 开始处。然后统计当前
    ! 磁道上已经读取的扇区数并与磁道最大扇区数 sectors 作比较。如果小于 sectors 说明当前磁
    ! 道上的还有扇区未读。于是跳转到 ok3_read 处继续操作。
235             call read_track         ! 读当前磁道上指定开始扇区和需读扇区数的数据。
236             mov cx,ax               ! cx = 该次操作已读取的扇区数。
237             add ax,sread            ! 加上当前磁道上已经读取的扇区数。
238             seg cs
239             cmp ax,sectors          ! 若当前磁道上的还有扇区未读，则跳转到 ok3_read 处。
240             jne ok3_read
    ! 若该磁道的当前磁头面所有扇区已经读取，则读该磁道的下一磁头面（1 号磁头）上的数据。
    ! 如果已经完成，则去读下一磁道。
241             mov ax,#1
242             sub ax,head             ! 判断当前磁头号。
243             jne ok4_read            ! 如果是 0 磁头，则再去读 1 磁头面上的扇区数据。
244             inc track               ! 否则去读下一磁道。
245 ok4_read:
246             mov head,ax             ! 保存当前磁头号。
247             xor ax,ax               ! 清当前磁道已读扇区数。
248 ok3_read:
    ! 如果当前磁道上的还有未读扇区，则首先保存当前磁道已读扇区数，然后调整存放数据处的开
    ! 始位置。若小于 64KB 边界值，则跳转到 rp_read(217 行)处，继续读数据。
249             mov sread,ax            ! 保存当前磁道已读扇区数。
250             shl cx,#9               ! 上次已读扇区数*512 字节。
251             add bx,cx               ! 调整当前段内数据开始位置。
252             jnc rp_read
    ! 否则说明已经读取 64KB 数据。此时调整当前段，为读下一段数据作准备。
253             mov ax,es
254             add ah,#0x10            ! 将段基址调整为指向下一个 64KB 内存开始处。
```

```
255             mov es,ax
256             xor bx,bx                       ！清段内数据开始偏移值。
257             jmp rp_read                     ！跳转至 rp_read(217 行)处，继续读数据。
258
    ! read_track 子程序。读当前磁道上指定开始扇区和需读扇区数的数据到 es:bx 开始处。参见
    ！第 67 行下对 BIOS 磁盘读中断 int 0x13，ah=2 的说明。
    ! al – 需读扇区数；es:bx – 缓冲区开始位置。
259 read_track:
    ！首先调用 BIOS 中断 0x10，功能 0x0e（以电传方式写字符），光标前移一位置。显示一个'.'。
260             pusha                           ！压入所有寄存器（push all）。
261             pusha                           ！为调用显示中断压入所有寄存器值。
262             mov ax, #0xe2e                  ! loading... message 2e = .
263             mov bx, #7                      ！字符前景色属性。
264             int 0x10
265             popa
266
    ！然后正式进行磁道扇区读操作。
267             mov dx,track                    ！取当前磁道号。
268             mov cx,sread                    ！取当前磁道上已读扇区数。
269             inc cx                          ! cl = 开始读扇区。
270             mov ch,dl                       ! ch = 当前磁道号。
271             mov dx,head                     ！取当前磁头号。
272             mov dh,dl                       ! dh = 磁头号，dl = 驱动器号(为 0 表示当前 A 驱动器)。
273             and dx,#0x0100                  ！磁头号不大于 1。
274             mov ah,#2                       ! ah = 2，读磁盘扇区功能号。
275
276             push dx                         ! save for error dump
277             push cx                         ！为出错情况保存一些信息。
278             push bx
279             push ax
280
281             int 0x13
282             jc bad_rt                       ！若出错，则跳转至 bad_rt。
283             add sp,#8                       ！没有出错。因此丢弃为出错情况保存的信息。
284             popa
285             ret
286
    ！读磁盘操作出错。则先显示出错信息，然后执行驱动器复位操作（磁盘中断功能号 0），再跳转
    ！到 read_track 处重试。
287 bad_rt: push ax                            ! save error code
288             call print_all                  ! ah = error, al = read
289
290
291             xor ah,ah
```

```
292             xor dl,dl
293             int 0x13
294
295
296             add sp, #10                     ！丢弃为出错情况保存的信息。
297             popa
298             jmp read_track
299
300 /*
301  *          print_all is for debugging purposes.
302  *          It will print out all of the registers.    The assumption is that this is
303  *          called from a routine, with a stack frame like
304  *          dx
305  *          cx
306  *          bx
307  *          ax
308  *          error
309  *          ret <- sp
310  *
311 */
    /*
     *          子程序 print_all 用于调试目的。它会显示所有寄存器的内容。前提条件是需要从
     *          一个子程序中调用，并且栈帧结构为如下所示：（见上面）
     */
    ！若标志寄存器的 CF=0，则不显示寄存器名称。
312
313 print_all:
314             mov cx, #5                      ! error code + 4 registers   ！显示值个数。
315             mov bp, sp                      ！保存当前栈指针 sp。
316
317 print_loop:
318             push cx                         ! save count left    ！保存需要显示的剩余个数。
319             call print_nl                   ! nl for readability   ！为可读性先让光标回车换行。
320             jae no_reg                      ! see if register name is needed
321                                             ！若 FLAGS 的标志 CF=0 则不显示寄存器名，于是跳转。
    ！对应入栈寄存器顺序分别显示它们的名称“AX：”等。
322             mov ax, #0xe05 + 0x41 - 1   ! ah =功能号（0x0e）；al =字符（0x05 + 0x41 -1）。
323             sub al, cl
324             int 0x10
325
326             mov al, #0x58                   ! X       ！显示字符'X'。
327             int 0x10
328
329             mov al, #0x3a                   ! :       ！显示字符':'。
```

```
330             int 0x10
331
    ！显示寄存器 bp 所指栈中内容。开始时 bp 指向返回地址。
332 no_reg:
333             add bp, #2                  ! next register     ！栈中下一个位置。
334             call print_hex              ! print it          ！以十六进制显示。
335             pop cx
336             loop print_loop
337             ret
338
    ！调用 BIOS 中断 0x10，以电传方式显示回车换行。
339 print_nl:
340             mov ax, #0xe0d              ! CR
341             int 0x10
342             mov al, #0xa                ! LF
343             int 0x10
344             ret
345
346 /*
347   *         print_hex is for debugging purposes, and prints the word
348   *         pointed to by ss:bp in hexadecmial.
349 */
    /*
     *         子程序 print_hex 用于调试目的。它使用十六进制在屏幕上显示出
     *         ss:bp 指向的字。
     */
350
    ！调用 BIOS 中断 0x10，以电传方式和 4 个十六进制数显示 ss:bp 指向的字。
351 print_hex:
352             mov     cx, #4              ! 4 hex digits          ！要显示 4 个十六进制数字。
353             mov     dx, (bp)            ! load word into dx     ！显示值放入 dx 中。
354 print_digit:
    ！先显示高字节，因此需要把 dx 中值左旋 4 比特，此时高 4 比特在 dx 的低 4 位中。
355             rol     dx, #4              ! rotate so that lowest 4 bits are used
356             mov     ah, #0xe            ！中断功能号。
357             mov     al, dl              ! mask off so we have only next nibble
358             and     al, #0xf            ！放入 al 中并只取低 4 比特（1 个值）。
    ！加上'0' 的 ASCII 码值 0x30，把显示值转换成基于数字'0' 的字符。若此时 al 值超过 0x39，
    ！表示欲显示值超过数字 9，因此需要使用'A'--'F'来表示。
359             add     al, #0x30           ! convert to 0 based digit, '0'
360             cmp     al, #0x39           ! check for overflow
361             jbe     good_digit
362             add     al, #0x41 - 0x30 - 0xa      ! 'A' - '0' - 0xa
363
```

```
364 good_digit:
365         int     0x10
366         loop    print_digit         ! cx--。若 cx>0 则去显示下一个值。
367         ret
368
369
370 /*
371  * This procedure turns off the floppy drive motor, so
372  * that we enter the kernel in a known state, and
373  * don't have to worry about it later.
374  */
    /* 这个子程序用于关闭软驱的马达，这样我们进入内核后就能
     * 知道它所处的状态，以后也就无须担心它了。
     */
    ! 下面第 377 行上的值 0x3f2 是软盘控制器的一个端口，被称为数字输出寄存器（DOR）端口。它是
    ! 一个 8 位的寄存器，其位 7--位 4 分别用于控制 4 个软驱（D--A）的启动和关闭。位 3--位 2 用于
    ! 允许/禁止 DMA 和中断请求以及启动/复位软盘控制器 FDC。 位 1--位 0 用于选择选择操作的软驱。
    ! 第 378 行上在 al 中设置并输出的 0 值，就是用于选择 A 驱动器，关闭 FDC，禁止 DMA 和中断请求，
    ! 关闭马达。有关软驱控制卡编程的详细信息请参见 kernel/blk_drv/floppy.c 程序后面的说明。
375 kill_motor:
376         push dx
377         mov dx,#0x3f2           ! 软驱控制卡的数字输出寄存器端口，只写。
378         xor al, al              ! A 驱动器，关闭 FDC，禁止 DMA 和中断请求，关闭马达。
379         outb                    ! 将 al 中的内容输出到 dx 指定的端口去。
380         pop dx
381         ret
382
383 sectors:
384         .word 0                 ! 存放当前启动软盘每磁道的扇区数。
385
386 msg1:                           ! 开机调用 BIOS 中断显示的信息。共 9 个字符。
387         .byte 13,10             ! 回车、换行的 ASCII 码。
388         .ascii "Loading"
389
    ! 表示下面语句从地址 508(0x1FC)开始，所以 root_dev 在启动扇区的第 508 开始的 2 个字节中。
390 .org 506
391 swap_dev:
392         .word SWAP_DEV          ! 这里存放交换系统所在设备号(init/main.c 中会用)。
393 root_dev:
394         .word ROOT_DEV          ! 这里存放根文件系统所在设备号(init/main.c 中会用)。

    ! 下面是启动盘具有有效引导扇区的标志。仅供 BIOS 中的程序加载引导扇区时识别使用。它必须
    ! 位于引导扇区的最后两个字节中。
395 boot_flag:
```

```
396                 .word 0xAA55
397
398 .text
399 endtext:
400 .data
401 enddata:
402 .bss
403 endbss:
404
```

6.2 程序 6-2 linux/boot/setup.S

```
 1 !
 2 !       setup.s         (C) 1991 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.
 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 !
   ! setup.s 负责从 BIOS 中获取系统数据，并将这些数据放到系统内存的适当
   ! 地方。此时 setup.s 和 system 已经由 bootsect 引导块加载到内存中。
   !
   ! 这段代码询问 bios 有关内存/磁盘/其他参数，并将这些参数放到一个
   ! “安全的”地方：0x90000-0x901FF，也即原来 bootsect 代码块曾经在
   ! 的地方，然后在被缓冲块覆盖掉之前由保护模式的 system 读取。
14
15 ! NOTE! These had better be the same as in bootsect.s!
   ! 以下这些参数最好和 bootsect.s 中的相同!
16 #include <linux/config.h>
   ! config.h 中定义了 DEF_INITSEG = 0x9000; DEF_SYSSEG = 0x1000; DEF_SETUPSEG = 0x9020。
17
18 INITSEG  = DEF_INITSEG  ! we move boot here - out of the way ! 原来 bootsect 所处的段。
19 SYSSEG   = DEF_SYSSEG   ! system loaded at 0x10000 (65536).  ! system 在 0x10000 处。
20 SETUPSEG = DEF_SETUPSEG ! this is the current segment        ! 本程序所在的段地址。
21
22 .globl begtext, begdata, begbss, endtext, enddata, endbss
23 .text
24 begtext:
25 .data
26 begdata:
27 .bss
28 begbss:
29 .text
30
31 entry start
32 start:
33
34 ! ok, the read went well so we get current cursor position and save it for
35 ! posterity.
   ! ok，整个读磁盘过程都正常，现在将光标位置保存以备今后使用（相关代码在 59--62 行）。
36
   ! 下句将 ds 置成 INITSEG(0x9000)。这已经在 bootsect 程序中设置过，但是现在是 setup 程序，
   ! Linus 觉得需要再重新设置一下。
37         mov     ax,#INITSEG
38         mov     ds,ax
```

```
39
40 ! Get memory size (extended mem, kB)
   ! 取扩展内存的大小值（KB）。
   ! 利用 BIOS 中断 0x15 功能号 ah = 0x88 取系统所含扩展内存大小并保存在内存 0x90002 处。
   ! 返回：ax = 从 0x100000（1M）处开始的扩展内存大小(KB)。若出错则 CF 置位，ax = 出错码。
41
42          mov     ah,#0x88
43          int     0x15
44          mov     [2],ax              ! 将扩展内存数值存在 0x90002 处（1 个字）。
45
46 ! check for EGA/VGA and some config parameters
   ! 检查显示方式（EGA/VGA）并取参数。
   ! 调用 BIOS 中断 0x10，附加功能选择方式信息。功能号：ah = 0x12，bl = 0x10
   ! 返回：bh =显示状态。0x00 -彩色模式，I/O 端口=0x3dX；0x01 -单色模式，I/O 端口=0x3bX。
   ! bl = 安装的显示内存。0x00 - 64k；0x01 - 128k；0x02 - 192k；0x03 = 256k。
   ! cx = 显示卡特性参数(参见程序后对 BIOS 视频中断 0x10 的说明)。
47
48          mov     ah,#0x12
49          mov     bl,#0x10
50          int     0x10
51          mov     [8],ax              ! 0x90008 = ??
52          mov     [10],bx             ! 0x9000A =安装的显示内存；0x9000B=显示状态(彩/单色)
53          mov     [12],cx             ! 0x9000C =显示卡特性参数。
   ! 检测屏幕当前行列值。若显示卡是 VGA 卡时则请求用户选择显示行列值，并保存到 0x9000E 处。
54          mov     ax,#0x5019          ! 在 ax 中预置屏幕默认行列值（ah = 80 列；al=25 行）。
55          cmp     bl,#0x10            ! 若中断返回 bl 值为 0x10，则表示不是 VGA 显示卡，跳转。
56          je      novga
57          call    chsvga              ! 检测显示卡厂家和类型，修改显示行列值（第 215 行）。
58 novga:   mov     [14],ax             ! 保存屏幕当前行列值（0x9000E，0x9000F）。

   ! 这段代码使用 BIOS 中断取屏幕当前光标位置（列、行），并保存在内存 0x90000 处（2 字节）。
   ! 控制台初始化程序会到此处读取该值。
   ! BIOS 中断 0x10 功能号 ah = 0x03，读光标位置。
   ! 输入：bh = 页号
   ! 返回：ch = 扫描开始线；cl = 扫描结束线；dh = 行号(0x00 顶端)；dl = 列号(0x00 最左边)。
59          mov     ah,#0x03            ! read cursor pos
60          xor     bh,bh
61          int     0x10                ! save it in known place, con_init fetches
62          mov     [0],dx              ! it from 0x90000.
63
64 ! Get video-card data:
   ! 下面这段用于取显示卡当前显示模式。
   ! 调用 BIOS 中断 0x10，功能号 ah = 0x0f
   ! 返回：ah = 字符列数；al = 显示模式；bh = 当前显示页。
   ! 0x90004(1 字)存放当前页；0x90006 存放显示模式；0x90007 存放字符列数。
65
66          mov     ah,#0x0f
67          int     0x10
68          mov     [4],bx              ! bh = display page
69          mov     [6],ax              ! al = video mode, ah = window width
70
71 ! Get hd0 data
   ! 取第一个硬盘的信息（复制硬盘参数表）。
```

```
! 第 1 个硬盘参数表的首地址竟然是中断向量 0x41 的向量值！而第 2 个硬盘参数表紧接在第 1 个
! 表的后面，中断向量 0x46 的向量值也指向第 2 个硬盘的参数表首址。表的长度是 16 个字节。
! 下面两段程序分别复制 ROM BIOS 中有关两个硬盘的参数表，0x90080 处存放第 1 个硬盘的表，
! 0x90090 处存放第 2 个硬盘的表。
72
! 第 75 行语句从内存指定位置处读取一个长指针值并放入 ds 和 si 寄存器中。ds 中放段地址，
! si 是段内偏移地址。这里是把内存地址 4 * 0x41（= 0x104）处保存的 4 个字节读出。这 4 字
! 节即是硬盘参数表所处位置的段和偏移值。
73          mov     ax,#0x0000
74          mov     ds,ax
75          lds     si,[4*0x41]         ! 取中断向量 0x41 的值，即 hd0 参数表的地址➔ds:si
76          mov     ax,#INITSEG
77          mov     es,ax
78          mov     di,#0x0080          ! 传输的目的地址: 0x9000:0x0080 ➔ es:di
79          mov     cx,#0x10            ! 共传输 16 字节。
80          rep
81          movsb
82
83  ! Get hd1 data
84
85          mov     ax,#0x0000
86          mov     ds,ax
87          lds     si,[4*0x46]         ! 取中断向量 0x46 的值，即 hd1 参数表的地址➔ds:si
88          mov     ax,#INITSEG
89          mov     es,ax
90          mov     di,#0x0090          ! 传输的目的地址: 0x9000:0x0090 ➔ es:di
91          mov     cx,#0x10
92          rep
93          movsb
94
95  ! Check that there IS a hd1 :-)
    ! 检查系统是否有第 2 个硬盘。如果没有则把第 2 个表清零。
    ! 利用 BIOS 中断调用 0x13 的取盘类型功能，功能号 ah = 0x15；
    ! 输入：dl = 驱动器号（0x8X 是硬盘：0x80 指第 1 个硬盘，0x81 第 2 个硬盘）
    ! 输出：ah = 类型码；00 - 没有这个盘，CF 置位；01 - 是软驱，没有 change-line 支持；
    !                   02 - 是软驱(或其他可移动设备)，有 change-line 支持； 03 - 是硬盘。
96
97          mov     ax,#0x01500
98          mov     dl,#0x81
99          int     0x13
100         jc      no_disk1
101         cmp     ah,#3               ! 是硬盘吗？(类型 = 3 ？)。
102         je      is_disk1
103 no_disk1:
104         mov     ax,#INITSEG         ! 第 2 个硬盘不存在，则对第 2 个硬盘表清零。
105         mov     es,ax
106         mov     di,#0x0090
107         mov     cx,#0x10
108         mov     ax,#0x00
109         rep
110         stosb
111 is_disk1:
112
```

```
113 ! now we want to move to protected mode ...
    ! 现在我们要进入保护模式中了...
114
115         cli                     ! no interrupts allowed !      ! 从此开始不允许中断。
116
117 ! first we move the system to it's rightful place
    ! 首先我们将 system 模块移到正确的位置。
    ! bootsect 引导程序会把 system 模块读入到内存 0x10000（64KB）开始的位置。由于当时假设
    ! system 模块最大长度不会超过 0x80000（512KB），即其末端不会超过内存地址 0x90000，所以
    ! bootsect 会把自己移动到 0x90000 开始的地方，并把 setup 加载到它的后面。下面这段程序的
    ! 用途是再把整个 system 模块移动到 0x00000 位置，即把从 0x10000 到 0x8ffff 的内存数据块
    ! （512KB）整块地向内存低端移动了 0x10000（64KB）的位置。
118
119         mov     ax,#0x0000
120         cld                     ! 'direction'=0, movs moves forward
121 do_move:
122         mov     es,ax           ! destination segment ! es:di 是目的地址(初始为 0x0:0x0)
123         add     ax,#0x1000
124         cmp     ax,#0x9000      ! 已经把最后一段（从 0x8000 段开始的 64KB）代码移动完?
125         jz      end_move        ! 是，则跳转。
126         mov     ds,ax           ! source segment  ! ds:si 是源地址(初始为 0x1000:0x0)
127         sub     di,di
128         sub     si,si
129         mov     cx,#0x8000      ! 移动 0x8000 字（64KB 字节）。
130         rep
131         movsw
132         jmp     do_move
133
134 ! then we load the segment descriptors
    ! 此后，我们加载段描述符。
    ! 从这里开始会遇到 32 位保护模式的操作，因此需要 Intel 32 位保护模式编程方面的知识了，
    ! 有关这方面的信息请查阅列表后的简单介绍或附录中的详细说明。这里仅作概要说明。在进入
    ! 保护模式中运行之前，我们需要首先设置好需要使用的段描述符表。这里需要设置全局描述符
    ! 表和中断描述符表。
    !
    ! 下面指令 lidt 用于加载中断描述符表（IDT）寄存器。它的操作数（idt_48）有 6 字节。前 2
    ! 字节（字节 0-1）是描述符表的字节长度值；后 4 字节（字节 2-5）是描述符表的 32 位线性基
    ! 地址，其形式参见下面 218--220 行和 222--224 行说明。中断描述符表中的每一个 8 字节表项
    ! 指出发生中断时需要调用的代码信息。与中断向量有些相似，但要包含更多的信息。
    !
    ! lgdt 指令用于加载全局描述符表（GDT）寄存器，其操作数格式与 lidt 指令的相同。全局描述
    ! 符表中的每个描述符项（8 字节）描述了保护模式下数据段和代码段（块）的信息。 其中包括
    ! 段的最大长度限制（16 位）、段的线性地址基址（32 位）、段的特权级、段是否在内存、读写
    ! 许可权以及其他一些保护模式运行的标志。参见后面 205--216 行。
135
136 end_move:
137         mov     ax,#SETUPSEG    ! right, forgot this at first. didn't work :-)
138         mov     ds,ax           ! ds 指向本程序(setup)段。
139         lidt    idt_48          ! load idt with 0,0                  ! 加载 IDT 寄存器。
140         lgdt    gdt_48          ! load gdt with whatever appropriate ! 加载 GDT 寄存器。
141
142 ! that was painless, now we enable A20
    ! 以上的操作很简单，现在我们开启 A20 地址线。
```

```
! 为了能够访问和使用 1MB 以上的物理内存，我们需要首先开启 A20 地址线。参见本程序列表后
! 有关 A20 信号线的说明。关于所涉及的一些端口和命令，可参考 kernel/chr_drv/keyboard.S
! 程序后对键盘接口的说明。至于机器是否真正开启了 A20 地址线，我们还需要在进入保护模式
! 之后（能访问 1MB 以上内存之后）在测试一下。这个工作放在了 head.S 程序中（32--36 行）。
143
144         call    empty_8042              ! 测试 8042 状态寄存器，等待输入缓冲器空。
                                            ! 只有当输入缓冲器为空时才可以对其执行写命令。
145         mov     al,#0xD1                ! command write ! 0xD1 命令码-表示要写数据到
146         out     #0x64,al                ! 8042 的 P2 端口。P2 端口位 1 用于 A20 线的选通。
147         call    empty_8042              ! 等待输入缓冲器空，看命令是否被接受。
148         mov     al,#0xDF                ! A20 on        ! 选通 A20 地址线的参数。
149         out     #0x60,al                ! 数据要写到 0x60 口。
150         call    empty_8042              ! 若此时输入缓冲器为空，则表示 A20 线已经选通。
151
152 ! well, that went ok, I hope. Now we have to reprogram the interrupts :-(
153 ! we put them right after the intel-reserved hardware interrupts, at
154 ! int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
155 ! messed this up with the original PC, and they haven't been able to
156 ! rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
157 ! which is used for the internal hardware interrupts as well. We just
158 ! have to reprogram the 8259's, and it isn't fun.
    !
    ! 希望以上一切正常。现在我们必须重新对中断进行编程 :-( 我们将它们放在正好
    ! 处于 Intel 保留的硬件中断后面，即 int 0x20--0x2F。在那里它们不会引起冲突。
    ! 不幸的是 IBM 在原 PC 机中搞糟了，以后也没有纠正过来。所以 PC 机 BIOS 把中断
    ! 放在了 0x08--0x0f，这些中断也被用于内部硬件中断。所以我们就必须重新对 8259
    ! 中断控制器进行编程，这一点都没意思。
    !
    ! PC 机使用 2 个 8259A 芯片，关于对可编程控制器 8259A 芯片的编程方法请参见本程序后的介绍。
    ! 第 162 行上定义的两个字（0x00eb）是直接使用机器码表示的两条相对跳转指令，起延时作用。
    ! 0xeb 是直接近跳转指令的操作码，带 1 个字节的相对位移值。因此跳转范围是-127 到 127。CPU
    ! 通过把这个相对位移值加到 EIP 寄存器中就形成一个新的有效地址。此时 EIP 指向下一条被执行
    ! 的指令。执行时所花费的 CPU 时钟周期数是 7 至 10 个。0x00eb 表示跳转值是 0 的一条指令，因
    ! 此还是直接执行下一条指令。这两条指令共可提供 14--20 个 CPU 时钟周期的延迟时间。在 as86
    ! 中没有表示相应指令的助记符，因此 Linus 在 setup.s 等一些汇编程序中就直接使用机器码来表
    ! 示这种指令。另外，每个空操作指令 NOP 的时钟周期数是 3 个，因此若要达到相同的延迟效果就
    ! 需要 6 至 7 个 NOP 指令。
159
    ! 8259 芯片主片端口是 0x20-0x21，从片端口是 0xA0-0xA1。输出值 0x11 表示初始化命令开始，
    ! 它是 ICW1 命令字，表示边沿触发、多片 8259 级连、最后要发送 ICW4 命令字。
160         mov     al,#0x11                ! initialization sequence
161         out     #0x20,al                ! send it to 8259A-1  ! 发送到 8259A 主芯片。
162         .word   0x00eb,0x00eb           ! jmp $+2, jmp $+2    ! '$'表示当前指令的地址，
163         out     #0xA0,al                ! and to 8259A-2      ! 再发送到 8259A 从芯片。
164         .word   0x00eb,0x00eb
    ! Linux 系统硬件中断号被设置成从 0x20 开始。参见表 3-2：硬件中断请求信号与中断号对应表。
165         mov     al,#0x20                ! start of hardware int's (0x20)
166         out     #0x21,al                ! 送主芯片 ICW2 命令字，设置起始中断号，要送奇端口。
167         .word   0x00eb,0x00eb
168         mov     al,#0x28                ! start of hardware int's 2 (0x28)
169         out     #0xA1,al                ! 送从芯片 ICW2 命令字，从芯片的起始中断号。
170         .word   0x00eb,0x00eb
171         mov     al,#0x04                ! 8259-1 is master
```

```
172         out     #0x21,al                ! 送主芯片 ICW3 命令字，主芯片的 IR2 连从芯片 INT。
                                            ! 参见代码列表后的说明。
173         .word   0x00eb,0x00eb
174         mov     al,#0x02                ! 8259-2 is slave
175         out     #0xA1,al                ! 送从芯片 ICW3 命令字，表示从芯片的 INT 连到主芯
                                            ! 片的 IR2 引脚上。
176         .word   0x00eb,0x00eb
177         mov     al,#0x01                ! 8086 mode for both
178         out     #0x21,al                ! 送主芯片 ICW4 命令字。8086 模式；普通 EOI、非缓冲
                                            ! 方式，需发送指令来复位。初始化结束，芯片就绪。
179         .word   0x00eb,0x00eb
180         out     #0xA1,al                ! 送从芯片 ICW4 命令字，内容同上。
181         .word   0x00eb,0x00eb
182         mov     al,#0xFF                ! mask off all interrupts for now
183         out     #0x21,al                ! 屏蔽主芯片所有中断请求。
184         .word   0x00eb,0x00eb
185         out     #0xA1,al                ! 屏蔽从芯片所有中断请求。
186
187 ! well, that certainly wasn't fun :-(. Hopefully it works, and we don't
188 ! need no steenking BIOS anyway (except for the initial loading :-).
189 ! The BIOS-routine wants lots of unnecessary data, and it's less
190 ! "interesting" anyway. This is how REAL programmers do it.
191 !
192 ! Well, now's the time to actually move into protected mode. To make
193 ! things as simple as possible, we do no register set-up or anything,
194 ! we let the gnu-compiled 32-bit programs do that. We just jump to
195 ! absolute address 0x00000, in 32-bit protected mode.
    !
    ! 哼，上面这段编程当然没劲:-(，但希望这样能工作，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。
    !
    ! 好了，现在是真正开始进入保护模式的时候了。为了把事情做得尽量简单，我们并不对
    ! 寄存器内容进行任何设置。我们让 gnu 编译的 32 位程序去处理这些事。在进入 32 位保
    ! 护模式时我们仅是简单地跳转到绝对地址 0x00000 处。
196
    ! 下面设置并进入 32 位保护模式运行。首先加载机器状态字(lmsw-Load Machine Status Word)，
    ! 也称控制寄存器 CR0，其比特位 0 置 1 将导致 CPU 切换到保护模式，并且运行在特权级 0 中，即
    ! 当前特权级 CPL=0。此时段寄存器仍然指向与实地址模式中相同的线性地址处（在实地址模式下
    ! 线性地址与物理内存地址相同）。在设置该比特位后，随后一条指令必须是一条段间跳转指令以
    ! 用于刷新 CPU 当前指令队列。因为 CPU 是在执行一条指令之前就已从内存读取该指令并对其进行
    ! 解码。然而在进入保护模式以后那些属于实模式的预先取得的指令信息就变得不再有效。而一条
    ! 段间跳转指令就会刷新 CPU 的当前指令队列，即丢弃这些无效信息。另外，在 Intel 公司的手册
    ! 上建议 80386 或以上 CPU 应该使用指令“mov cr0,ax”切换到保护模式。lmsw 指令仅用于兼容以
    ! 前的 286 CPU。

197         mov     ax,#0x0001      ! protected mode (PE) bit       ! 保护模式比特位(PE)。
198         lmsw    ax              ! This is it!                   ! 就这样加载机器状态字!
199         jmpi    0,8             ! jmp offset 0 of segment 8 (cs) ! 跳转至 cs 段偏移 0 处。
    ! 我们已经将 system 模块移动到 0x00000 开始的地方，所以上句中的偏移地址是 0。而段值 8 已经
    ! 是保护模式下的段选择符了，用于选择描述符表和描述符表项以及所要求的特权级。段选择符长
    ! 度为 16 位（2 字节）；位 0-1 表示请求的特权级 0--3，但 Linux 操作系统只用到两级：0 级（内
    ! 核级）和 3 级（用户级）；位 2 用于选择全局描述符表（0）还是局部描述符表(1)；位 3-15 是描
```

```
    ! 述符表项的索引，指出选择第几项描述符。所以段选择符 8（0b0000,0000,0000,1000）表示请求
    ! 特权级 0、使用全局描述符表 GDT 中第 2 个段描述符项，该项指出代码的基地址是 0（参见 571 行），
    ! 因此这里的跳转指令就会去执行 system 中的代码。另外，
200
201 ! This routine checks that the keyboard command queue is empty
202 ! No timeout is used - if this hangs there is something wrong with
203 ! 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（键盘控制器状态寄存器位 1 = 0）才可以对其执行写命令。
204 empty_8042:
205         .word   0x00eb,0x00eb
206         in      al,#0x64        ! 8042 status port          ! 读 AT 键盘控制器状态寄存器。
207         test    al,#2           ! is input buffer full?     ! 测试位 1，输入缓冲器满？
208         jnz     empty_8042      ! yes - loop
209         ret
210
211 ! Routine trying to recognize type of SVGA-board present (if any)
212 ! and if it recognize one gives the choices of resolution it offers.
213 ! If one is found the resolution chosen is given by al,ah (rows,cols).
    ! 下面是用于识别 SVGA 显示卡（若有的话）的子程序。若识别出一块就向用户
    ! 提供选择分辨率的机会，并把分辨率放入寄存器 al、ah（行、列）中返回。
    !
    ! 注意下面 215--566 行代码牵涉到众多显示卡端口信息，因此比较复杂。但由于这段代码与内核
    ! 运行关系不大，因此可以跳过不看。
    ! 下面首先显示 588 行上的 msg1 字符串（"按<回车键>查看存在的 SVGA 模式，或按任意键继续"），
    ! 然后循环读取键盘控制器输出缓冲器，等待用户按键。如果用户按下回车键就去检查系统具有
    ! 的 SVGA 模式，并在 AL 和 AH 中返回最大行列值，否则设置默认值 AL=25 行、AH=80 列并返回。
214
215 chsvga: cld
216         push    ds              ! 保存 ds 值。将在 231 行（或 490 或 492 行）弹出。
217         push    cs              ! 把默认数据段设置成和代码段同一个段。
218         pop     ds
219         mov     ax,#0xc000
220         mov     es,ax           ! es 指向 0xc000 段。此处是 VGA 卡上的 ROM BIOS 区。
221         lea     si,msg1         ! ds:si 指向 msg1 字符串。
222         call    prtstr          ! 显示以 NULL 结尾的 msg1 字符串。
223 nokey:  in      al,#0x60        ! 读取键盘控制器输出缓冲器（来自键盘的扫描码或命令）。
224         cmp     al,#0x82        ! 如果收到比 0x82 小的扫描码则是接通扫描码，因为 0x82 是
225         jb      nokey           ! 最小断开扫描码值。小于 0x82 表示还没有按键松开。
226         cmp     al,#0xe0        ! 如果扫描码大于 0xe0，表示收到的是扩展扫描码前缀。
227         ja      nokey
228         cmp     al,#0x9c        ! 如果断开扫描码是 0x9c，表示用户按下/松开了回车键，
229         je      svga            ! 于是程序跳转去检查系统是否具有 SVGA 模式。
230         mov     ax,#0x5019      ! 否则把 AX 中返回行列值默认设置为 AL=25 行、AH=80 列。
231         pop     ds
232         ret
    ! 下面根据 VGA 显示卡上的 ROM BIOS 指定位置处的特征数据串或者支持的特别功能来判断机器上
    ! 安装的是什么牌子的显示卡。本程序共支持 10 种显示卡的扩展功能。注意，此时程序已经在第
    ! 220 行把 es 指向 VGA 卡上 ROM BIOS 所在的段 0xc000（参见第 2 章）。
    ! 首先判断是不是 ATI 显示卡。我们把 ds:si 指向 595 行上 ATI 显示卡特征数据串，并把 es:si 指
    ! 向 VGA BIOS 中指定位置（偏移 0x31）处。因为该特征串共有 9 个字符（"761295520"），因此我
```

```
! 们循环比较这个特征串。如果相同则表示机器中的 VGA 卡是 ATI 牌子的，于是让 ds:si 指向该显
! 示卡可以设置的行列模式值 dscati（第 615 行），让 di 指向 ATI 卡可设置的行列个数和模式，
! 并跳转到标号 selmod（438 行）处进一步进行设置。
233 svga:   lea     si,idati        ! Check ATI 'clues'  ! 检查判断 ATI 显示卡的数据。
234         mov     di,#0x31        ! 特征串从 0xc000:0x0031 开始。
235         mov     cx,#0x09        ! 特征串有 9 个字节。
236         repe
237         cmpsb
238         jne     noati           ! 若特征串不同则表示不是 ATI 显示卡。跳转继续检测卡。
239         lea     si,dscati       ! 如果 9 个字节都相同，表示系统中有一块 ATI 牌显示卡。
240         lea     di,moati        ! 于是 si 指向 ATI 卡具有的可选行列值，di 指向可选个数
241         lea     cx,selmod       ! 和模式列表，然后跳转到 selmod（438 行）处继续处理。
242         jmp     cx

! 现在来判断是不是 Ahead 牌子的显示卡。首先向 EGA/VGA 图形索引寄存器 0x3ce 写入想访问的
! 主允许寄存器索引号 0x0f，同时向 0x3cf 端口（此时对应主允许寄存器）写入开启扩展寄存器
! 标志值 0x20。然后通过 0x3cf 端口读取主允许寄存器值，以检查是否可以设置开启扩展寄存器
! 标志。如果可以则说明是 Ahead 牌子的显示卡。注意 word 输出时 al➔端口 n，ah➔端口 n+1。
243 noati:  mov     ax,#0x200f      ! Check Ahead 'clues'
244         mov     dx,#0x3ce       ! 数据端口指向主允许寄存器（0x0f➔0x3ce 端口），
245         out     dx,ax           ! 并设置开启扩展寄存器标志（0x20➔0x3cf 端口）。
246         inc     dx              ! 然后再读取该寄存器，检查该标志是否被设置上。
247         in      al,dx
248         cmp     al,#0x20        ! 如果读取值是 0x20，则表示是 Ahead A 显示卡。
249         je      isahed          ! 如果读取值是 0x21，则表示是 Ahead B 显示卡。
250         cmp     al,#0x21        ! 否则说明不是 Ahead 显示卡，于是跳转继续检测其余卡。
251         jne     noahed
252 isahed: lea     si,dscahead     ! si 指向 Ahead 显示卡可选行列值表，di 指向扩展模式个
253         lea     di,moahead      ! 数和扩展模式号列表。然后跳转到 selmod（438 行）处继
254         lea     cx,selmod       ! 续处理。
255         jmp     cx

! 现在来检查是不是 Chips & Tech 生产的显示卡。通过端口 0x3c3（0x94 或 0x46e8）设置 VGA 允许
! 寄存器的进入设置模式标志（位 4），然后从端口 0x104 读取显示卡芯片集标识值。如果该标识值
! 是 0xA5，则说明是 Chips & Tech 生产的显示卡。
256 noahed: mov     dx,#0x3c3       ! Check Chips & Tech. 'clues'
257         in      al,dx           ! 从 0x3c3 端口读取 VGA 允许寄存器值，添加上进入设置模式
258         or      al,#0x10        ! 标志（位 4）后再写回。
259         out     dx,al
260         mov     dx,#0x104       ! 在设置模式时从全局标识端口 0x104 读取显示卡芯片标识值，
261         in      al,dx           ! 并暂时存放在 bl 寄存器中。
262         mov     bl,al
263         mov     dx,#0x3c3       ! 然后把 0x3c3 端口中的进入设置模式标志复位。
264         in      al,dx
265         and     al,#0xef
266         out     dx,al
267         cmp     bl,[idcandt]    ! 再把 bl 中标识值与位于 idcandt 处（第 596 行）的 Chips &
268         jne     nocant          ! Tech 的标识值 0xA5 作比较。如果不同则跳转比较下一种显卡。
269         lea     si,dsccandt     ! 让 si 指向这种显示卡的可选行列值表，di 指向扩展模式个数
270         lea     di,mocandt      ! 和扩展模式号列表。然后跳转到 selmod（438 行）进行设置
271         lea     cx,selmod       ! 显示模式的操作。
272         jmp     cx
```

```
! 现在检查是不是 Cirrus 显示卡。方法是使用 CRT 控制器索引号 0x1f 寄存器的内容来尝试禁止扩展
! 功能。该寄存器被称为鹰标（Eagle ID）寄存器，将其值高低半字节交换一下后写入端口 0x3c4 索
! 引的 6 号（定序/扩展）寄存器应该会禁止 Cirrus 显示卡的扩展功能。如果不会则说明不是 Cirrus
! 显示卡。因为从端口 0x3d4 索引的 0x1f 鹰标寄存器中读取的内容是鹰标值与 0x0c 索引号对应的显
! 存起始地址高字节寄存器内容异或操作之后的值，因此在读 0x1f 中内容之前我们需要先把显存起始
! 高字节寄存器内容保存后清零，并在检查后恢复之。另外，将没有交换过的 Eagle ID 值写到 0x3c4
! 端口索引的 6 号定序/扩展寄存器会重新开启扩展功能。
273 nocant: mov     dx,#0x3d4       ! Check Cirrus 'clues'
274         mov     al,#0x0c        ! 首先向 CRT 控制寄存器的索引寄存器端口 0x3d4 写入要访问
275         out     dx,al           ! 的寄存器索引号 0x0c（对应显存起始地址高字节寄存器），
276         inc     dx              ! 然后从 0x3d5 端口读入显存起始地址高字节并暂存在 bl 中，
277         in      al,dx           ! 再把显存起始地址高字节寄存器清零。
278         mov     bl,al
279         xor     al,al
280         out     dx,al
281         dec     dx              ! 接着向 0x3d4 端口输出索引 0x1f，指出我们要在 0x3d5 端口
282         mov     al,#0x1f        ! 访问读取“Eagle ID”寄存器内容。
283         out     dx,al
284         inc     dx
285         in      al,dx           ! 从 0x3d5 端口读取“Eagle ID”寄存器值，并暂存在 bh 中。
286         mov     bh,al           ! 然后把该值高低 4 比特互换位置存放到 cl 中。再左移 8 位
287         xor     ah,ah           ! 后放入 ch 中，而 cl 中放入数值 6。
288         shl     al,#4
289         mov     cx,ax
290         mov     al,bh
291         shr     al,#4
292         add     cx,ax
293         shl     cx,#8
294         add     cx,#6           ! 最后把 cx 值存放入 ax 中。此时 ah 中是换位后的“Eagle
295         mov     ax,cx           ! ID”值，al 中是索引号 6，对应定序/扩展寄存器。把 ah
296         mov     dx,#0x3c4       ! 写到 0x3c4 端口索引的定序/扩展寄存器应该会导致 Cirrus
297         out     dx,ax           ! 显示卡禁止扩展功能。
298         inc     dx
299         in      al,dx           ! 如果扩展功能真的被禁止，那么此时读入的值应该为 0。
300         and     al,al           ! 如果不为 0 则表示不是 Cirrus 显示卡，跳转继续检查其他卡。
301         jnz     nocirr
302         mov     al,bh           ! 是 Cirrus 显示卡，则利用第 286 行保存在 bh 中的“Eagle
303         out     dx,al           ! ID”原值再重新开启 Cirrus 卡扩展功能。此时读取的返回
304         in      al,dx           ! 值应该为 1。若不是，则仍然说明不是 Cirrus 显示卡。
305         cmp     al,#0x01
306         jne     nocirr
307         call    rst3d4          ! 恢复 CRT 控制器的显示起始地址高字节寄存器内容。
308         lea     si,dsccirrus    ! si 指向 Cirrus 显示卡的可选行列值，di 指向扩展模式个数
309         lea     di,mocirrus     ! 和对应模式号。然后跳转到 selmod 处去选择显示模式。
310         lea     cx,selmod
311         jmp     cx
    ! 该子程序利用保存在 bl 中的值（第 278 行）恢复 CRT 控制器的显示起始地址高字节寄存器内容。
312 rst3d4: mov     dx,#0x3d4
313         mov     al,bl
314         xor     ah,ah
315         shl     ax,#8
316         add     ax,#0x0c
317         out     dx,ax           ! 注意，这是 word 输出！！ al ➔0x3d4，ah ➔0x3d5。
```

```
318             ret

    ! 现在检查系统中是不是 Everex 显示卡。方法是利用中断 int 0x10 功能 0x70（ax =0x7000，
    ! bx=0x0000）调用 Everex 的扩展视频 BIOS 功能。对于 Everes 类型显示卡，该中断调用应该
    ! 会返回模拟状态，即有以下返回信息：
    ! al = 0x70，若是基于 Trident 的 Everex 显示卡；
    ! cl = 显示器类型：00-单色；01-CGA；02-EGA；03-数字多频；04-PS/2；05-IBM 8514；06-SVGA。
    ! ch = 属性：位 7-6：00-256K，01-512K，10-1MB，11-2MB；位 4-开启 VGA 保护；位 0-6845 模拟。
    ! dx = 板卡型号：位 15-4：板类型标识号；位 3-0：板修正标识号。
    !      0x2360-Ultragraphics II；0x6200-Vision VGA；0x6730-EVGA；0x6780-Viewpoint。
    ! di = 用 BCD 码表示的视频 BIOS 版本号。
319 nocirr: call    rst3d4          ! Check Everex 'clues'
320         mov     ax,#0x7000      ! 设置 ax = 0x7000, bx=0x0000，调用 int 0x10。
321         xor     bx,bx
322         int     0x10
323         cmp     al,#0x70        ! 对于 Everes 显示卡，al 中应该返回值 0x70。
324         jne     noevrx
325         shr     dx,#4           ! 忽律板修正号（位 3-0）。
326         cmp     dx,#0x678       ! 板类型号是 0x678 表示是一块 Trident 显示卡，则跳转。
327         je      istrid
328         cmp     dx,#0x236       ! 板类型号是 0x236 表示是一块 Trident 显示卡，则跳转。
329         je      istrid
330         lea     si,dsceverex    ! 让 si 指向 Everex 显示卡的可选行列值表，让 di 指向扩展
331         lea     di,moeverex     ! 模式个数和模式号列表。然后跳转到 selmod 去执行选择
332         lea     cx,selmod       ! 显示模式的操作。
333         jmp     cx
334 istrid: lea     cx,ev2tri       ! 是 Trident 类型的 Everex 显示卡，则跳转到 ev2tri 处理。
335         jmp     cx

    ! 现在检查是不是 Genoa 显示卡。方式是检查其视频 BIOS 中的特征数字串（0x77、0x00、0x66、
    ! 0x99）。注意，此时 es 已经在第 220 行被设置成指向 VGA 卡上 ROM BIOS 所在的段 0xc000。
336 noevrx: lea     si,idgenoa      ! Check Genoa 'clues'
337         xor     ax,ax           ! 让 ds:si 指向第 597 行上的特征数字串。
338         seg es
339         mov     al,[0x37]       ! 取 VGA 卡上 BIOS 中 0x37 处的指针（它指向特征串）。
340         mov     di,ax           ! 因此此时 es:di 指向特征数字串开始处。
341         mov     cx,#0x04
342         dec     si
343         dec     di
344 l1:     inc     si              ! 然后循环比较这 4 个字节的特征数字串。
345         inc     di
346         mov     al,(si)
347         seg es
348         and     al,(di)
349         cmp     al,(si)
350         loope   l1
351         cmp     cx,#0x00        ! 如果特征数字串完全相同，则表示是 Genoa 显示卡，
352         jne     nogen           ! 否则跳转去检查其他类型的显示卡。
353         lea     si,dscgenoa     ! 让 si 指向 Genoa 显示卡的可选行列值表，让 di 指向扩展
354         lea     di,mogenoa      ! 模式个数和模式号列表。然后跳转到 selmod 去执行选择
355         lea     cx,selmod       ! 显示模式的操作。
356         jmp     cx
```

```
                ! 现在检查是不是 Paradise 显示卡。同样是采用比较显示卡上 BIOS 中特征串（“VGA=”）的方式。
357 nogen:  lea     si,idparadise   ! Check Paradise 'clues'
358         mov     di,#0x7d        ! es:di 指向 VGA ROM BIOS 的 0xc000:0x007d 处，该处应该有
359         mov     cx,#0x04        ! 4 个字符“VGA=”。
360         repe
361         cmpsb
362         jne     nopara          ! 若有不同的字符，表示不是 Paradise 显示卡，于是跳转。
363         lea     si,dscparadise  ! 否则让 si 指向 Paradise 显示卡的可选行列值表，让 di 指
364         lea     di,moparadise   ! 向扩展模式个数和模式号列表。然后跳转到 selmod 处去选
365         lea     cx,selmod       ! 择想要使用的显示模式。
366         jmp     cx

                ! 现在检查是不是 Trident（TVGA）显示卡。TVGA 显示卡扩充的模式控制寄存器 1（0x3c4 端口索引
                ! 的 0x0e）的位 3--0 是 64K 内存页面个数值。这个字段值有一个特性：当写入时，我们需要首先把
                ! 值与 0x02 进行异或操作后再写入；当读取该值时则不需要执行异或操作，即异或前的值应该与写
                ! 入后再读取的值相同。下面代码就利用这个特性来检查是不是 Trident 显示卡。
367 nopara: mov     dx,#0x3c4       ! Check Trident 'clues'
368         mov     al,#0x0e        ! 首先在端口 0x3c4 输出索引号 0x0e，索引模式控制寄存器 1。
369         out     dx,al           ! 然后从 0x3c5 数据端口读入该寄存器原值，并暂存在 ah 中。
370         inc     dx
371         in      al,dx
372         xchg    ah,al
373         mov     al,#0x00        ! 然后我们向该寄存器写入 0x00，再读取其值➔al。
374         out     dx,al           ! 写入 0x00 就相当于“原值”0x02 异或 0x02 后的写入值，
375         in      al,dx           ! 因此若是 Trident 显示卡，则此后读入的值应该是 0x02。
376         xchg    al,ah           ! 交换后，al=原模式控制寄存器 1 的值，ah=最后读取的值。
                ! 下面语句右则英文注释是“真奇怪...书中并没有要求这样操作，但是这对我的 Trident 显示卡
                ! 起作用。如果不这样做，屏幕就会变模糊...”。这几行附带有英文注释的语句执行如下操作：
                ! 如果 bl 中原模式控制寄存器 1 的位 1 在置位状态的话就将其复位，否则就将位 1 置位。
                ! 实际上这几条语句就是对原模式控制寄存器 1 的值执行异或 0x02 的操作，然后用结果值去设置
                ! （恢复）原寄存器值。
377         mov     bl,al           ! Strange thing ... in the book this wasn't
378         and     bl,#0x02        ! necessary but it worked on my card which
379         jz      setb2           ! is a trident. Without it the screen goes
380         and     al,#0xfd        ! blurred ...
381         jmp     clrb2           !
382 setb2:  or      al,#0x02        !
383 clrb2:  out     dx,al
384         and     ah,#0x0f        ! 取 375 行最后读入值的页面个数字段（位 3--0），如果
385         cmp     ah,#0x02        ! 该字段值等于 0x02，则表示是 Trident 显示卡。
386         jne     notrid
387 ev2tri: lea     si,dsctrident   ! 是 Trident 显示卡，于是让 si 指向该显示卡的可选行列
388         lea     di,motrident    ! 值列表，让 di 指向对应扩展模式个数和模式号列表，然
389         lea     cx,selmod       ! 后跳转到 selmod 去执行模式选择操作。
390         jmp     cx

                ! 现在检查是不是 Tseng 显示卡（ET4000AX 或 ET4000/W32 类）。方法是对 0x3cd 端口对应的段
                ! 选择（Segment Select）寄存器执行读写操作。该寄存器高 4 位（位 7--4）是要进行读操作的
                ! 64KB 段号（Bank number），低 4 位（位 3--0）是指定要写的段号。如果指定段选择寄存器的
                ! 的值是 0x55（表示读、写第 6 个 64KB 段），那么对于 Tseng 显示卡来说，把该值写入寄存器
                ! 后再读出应该还是 0x55。
391 notrid: mov     dx,#0x3cd       ! Check Tseng 'clues'
392         in      al,dx           ! Could things be this simple ! :-)
```

```
393             mov     bl,al           ! 先从 0x3cd 端口读取段选择寄存器原值，并保存在 bl 中。
394             mov     al,#0x55        ! 然后我们向该寄存器中写入 0x55。再读入并放在 ah 中。
395             out     dx,al
396             in      al,dx
397             mov     ah,al
398             mov     al,bl           ! 接着恢复该寄存器的原值。
399             out     dx,al
400             cmp     ah,#0x55        ! 如果读取的就是我们写入的值，则表明是 Tseng 显示卡。
401             jne     notsen
402             lea     si,dsctseng     ! 于是让 si 指向 Tseng 显示卡的可选行列值的列表，让 di
403             lea     di,motseng      ! 指向对应扩展模式个数和模式号列表，然后跳转到 selmod
404             lea     cx,selmod       ! 去执行模式选择操作。
405             jmp     cx

        ! 下面检查是不是 Video7 显示卡。端口 0x3c2 是混合输出寄存器写端口，而 0x3cc 是混合输出寄存
        ! 器读端口。该寄存器的位 0 是单色/彩色标志。如果为 0 则表示是单色，否则是彩色。判断是不是
        ! Video7 显示卡的方式是利用这种显示卡的 CRT 控制扩展标识寄存器（索引号是 0x1f）。该寄存器
        ! 的值实际上就是显存起始地址高字节寄存器（索引号 0x0c）的内容和 0xea 进行异或操作后的值。
        ! 因此我们只要向显存起始地址高字节寄存器中写入一个特定值，然后从标识寄存器中读取标识值
        ! 进行判断即可。
        ! 通过对以上显示卡和这里 Video7 显示卡的检查分析，我们可知检查过程通常分为三个基本步骤。
        ! 首先读取并保存测试需要用到的寄存器原值，然后使用特定测试值进行写入和读出操作，最后恢
        ! 复原寄存器值并对检查结果作出判断。
406 notsen: mov     dx,#0x3cc       ! Check Video7 'clues'
407             in      al,dx
408             mov     dx,#0x3b4       ! 先设置 dx 为单色显示 CRT 控制索引寄存器端口号 0x3b4。
409             and     al,#0x01        ! 如果混合输出寄存器的位 0 等于 0（单色）则直接跳转，
410             jz      even7           ! 否则 dx 设置为彩色显示 CRT 控制索引寄存器端口号 0x3d4。
411             mov     dx,#0x3d4
412 even7:     mov     al,#0x0c        ! 设置寄存器索引号为 0x0c，对应显存起始地址高字节寄存器。
413             out     dx,al
414             inc     dx
415             in      al,dx           ! 读取显示内存起始地址高字节寄存器内容，并保存在 bl 中。
416             mov     bl,al
417             mov     al,#0x55        ! 然后在显存起始地址高字节寄存器中写入值 0x55，再读取出来。
418             out     dx,al
419             in      al,dx
420             dec     dx              ! 然后通过 CRTC 索引寄存器端口 0x3b4 或 0x3d4 选择索引号是
421             mov     al,#0x1f        ! 0x1f 的 Video7 显示卡标识寄存器。该寄存器内容实际上就是
422             out     dx,al           ! 显存起始地址高字节和 0xea 进行异或操作后的结果值。
423             inc     dx
424             in      al,dx           ! 读取 Video7 显示卡标识寄存器值，并保存在 bh 中。
425             mov     bh,al
426             dec     dx              ! 然后再选择显存起始地址高字节寄存器，恢复其原值。
427             mov     al,#0x0c
428             out     dx,al
429             inc     dx
430             mov     al,bl
431             out     dx,al
432             mov     al,#0x55        ! 随后我们来验证“Video7 显示卡标识寄存器值就是显存起始
433             xor     al,#0xea        ! 地址高字节和 0xea 进行异或操作后的结果值”。因此 0x55
434             cmp     al,bh           ! 和 0xea 进行异或操作的结果就应该等于标识寄存器的测试值。
435             jne     novid7          ! 若不是 Video7 显示卡，则设置默认显示行列值（492 行）。
```

```
436         lea     si,dscvideo7        ! 是 Video7 显示卡，于是让 si 指向该显示卡行列值表，让 di
437         lea     di,movideo7         ! 指向扩展模式个数和模式号列表。

    ! 下面根据上述代码判断出的显示卡类型以及取得的相关扩展模式信息（si 指向的行列值列表；di
    ! 指向扩展模式个数和模式号列表），提示用户选择可用的显示模式，并设置成相应显示模式。最后
    ! 子程序返回系统当前设置的屏幕行列值（ah = 列数；al=行数）。例如，如果系统中是 ATI 显示卡，
    ! 那么屏幕上会显示以下信息：
    ! Mode:  COLSxROWS:
    ! 0.      132 x 25
    ! 1.      132 x 44
    ! Choose mode by pressing the corresponding number.
    !
    ! 这段程序首先在屏幕上显示 NULL 结尾的字符串信息“Mode:  COLSxROWS:”。
438 selmod: push    si
439         lea     si,msg2
440         call    prtstr
441         xor     cx,cx
442         mov     cl,(di)             ! 此时 cl 中是检查出的显示卡的扩展模式个数。
443         pop     si
444         push    si
445         push    cx
    ! 然后并在每一行上显示出当前显示卡可选择的扩展模式行列值，供用户选用。
446 tbl:    pop     bx                  ! bx = 显示卡的扩展模式总个数。
447         push    bx
448         mov     al,bl
449         sub     al,cl
450         call    dprnt               ! 以十进制格式显示 al 中的值。
451         call    spcing              ! 显示一个点再空 4 个空格。
452         lodsw                       ! 在 ax 中加载 si 指向的行列值，随后 si 指向下一个 word 值。
453         xchg    al,ah               ! 交换位置后 al = 列数。
454         call    dprnt               ! 显示列数；
455         xchg    ah,al               ! 此时 al 中是行数值。
456         push    ax
457         mov     al,#0x78            ! 显示一个小“x”，即乘号。
458         call    prnt1
459         pop     ax                  ! 此时 al 中是行数值。
460         call    dprnt               ! 显示行数。
461         call    docr                ! 回车换行。
462         loop    tbl                 ! 再显示下一个行列值。cx 中扩展模式计数值递减 1。
    ! 在扩展模式行列值都显示之后，显示“Choose mode by pressing the corresponding number.”，
    ! 然后从键盘口读取用户按键的扫描码，根据该扫描码确定用户选择的行列值模式号，并利用 ROM
    ! BIOS 的显示中断 int 0x10 功能 0x00 来设置相应的显示模式。
    ! 第 468 行的“模式个数值+0x80”是所按数字键-1 的松开扫描码。对于 0--9 数字键，它们的松开
    ! 扫描码分别是：0 - 0x8B；1 - 0x82；2 - 0x83；3 - 0x84；4 - 0x85；
    !              5 - 0x86；6 - 0x87；7 - 0x88；8 - 0x89；9 - 0x8A。
    ! 因此，如果读取的键盘松开扫描码小于 0x82 就表示不是数字键；如果扫描码等于 0x8B 则表示用户
    ! 按下数字 0 键。
463         pop     cx                  ! cl 中是显示卡扩展模式总个数值。
464         call    docr
465         lea     si,msg3             ! 显示“请按相应数字键来选择模式。”
466         call    prtstr
467         pop     si                  ! 弹出原行列值指针（指向显示卡行列值表开始处）。
468         add     cl,#0x80            ! cl + 0x80 = 对应“数字键-1”的松开扫描码。
```

```
nonum:  in      al,#0x60        ! Quick and dirty...
        cmp     al,#0x82        ! 若键盘松开扫描码小于 0x82 则表示不是数字键，忽律该键。
        jb      nonum
        cmp     al,#0x8b        ! 若键盘松开扫描码等于 0x8b，表示按下了数字键 0。
        je      zero
        cmp     al,cl           ! 若扫描码大于扩展模式个数值对应的最大扫描码值，表示
        ja      nonum           ! 键入的值超过范围或不是数字键的松开扫描码。否则表示
        jmp     nozero          ! 用户按下并松开了一个非 0 数字按键。
! 下面把松开扫描码转换成对应的数字按键值，然后利用该值从模式个数和模式号列表中选择对应的
! 的模式号。接着调用机器 ROM BIOS 中断 int 0x10 功能 0 把屏幕设置成模式号指定的模式。最后再
! 利用模式号从显示卡行列值表中选择并在 ax 中返回对应的行列值。
zero:   sub     al,#0x0a        ! al = 0x8b - 0x0a = 0x81。
nozero: sub     al,#0x80        ! 再减去 0x80 就可以得到用户选择了第几个模式。
        dec     al              ! 从 0 起计数。
        xor     ah,ah           ! int 0x10 显示功能号=0（设置显示模式）。
        add     di,ax
        inc     di              ! di 指向对应的模式号（跳过第 1 个模式个数字节值）。
        push    ax
        mov     al,(di)         ! 取模式号➔al 中，并调用系统 BIOS 显示中断功能 0。
        int     0x10
        pop     ax
        shl     ax,#1           ! 模式号乘 2，转换成为行列值表中对应值的指针。
        add     si,ax
        lodsw                   ! 取对应行列值到 ax 中（ah = 列数，al = 行数）。
        pop     ds              ! 恢复第 216 行保存的 ds 原值。在 ax 中返回当前显示行列值。
        ret

! 若都不是上面检测的显示卡，那么我们只好采用默认的 80 x 25 的标准行列值。
novid7: pop     ds              ! Here could be code to support standard 80x50,80x30
        mov     ax,#0x5019
        ret

! Routine that 'tabs' to next col.
! 光标移动到下一制表位的子程序。

! 显示一个点字符'.'和 4 个空格。
spcing: mov     al,#0x2e        ! 显示一个点字符'.'。
        call    prnt1
        mov     al,#0x20
        call    prnt1
        mov     al,#0x20
        call    prnt1
        mov     al,#0x20
        call    prnt1
        mov     al,#0x20
        call    prnt1
        ret

! Routine to print asciiz-string at DS:SI
! 显示位于 DS:SI 处以 NULL（0x00）结尾的字符串。

prtstr: lodsb
        and     al,al
```

```
514         jz      fin
515         call    prnt1           ! 显示 al 中的一个字符。
516         jmp     prtstr
517 fin:    ret
518
519 ! Routine to print a decimal value on screen, the value to be
520 ! printed is put in al (i.e 0-255).
    ! 显示十进制数字的子程序。显示值放在寄存器 al 中（0--255）。
521
522 dprnt:  push    ax
523         push    cx
524         mov     ah,#0x00
525         mov     cl,#0x0a
526         idiv    cl
527         cmp     al,#0x09
528         jbe     lt100
529         call    dprnt
530         jmp     skip10
531 lt100:  add     al,#0x30
532         call    prnt1
533 skip10: mov     al,ah
534         add     al,#0x30
535         call    prnt1
536         pop     cx
537         pop     ax
538         ret
539
540 ! Part of above routine, this one just prints ascii al
    ! 上面子程序的一部分。显示 al 中的一个字符。
    ! 该子程序使用中断 0x10 的 0x0E 功能，以电传方式在屏幕上写一个字符。光标会自动移到下一个
    ! 位置处。如果写完一行光标就会移动到下一行开始处。如果已经写完一屏最后一行，则整个屏幕
    ! 会向上滚动一行。字符 0x07（BEL）、0x08（BS）、0x0A(LF)和 0x0D（CR）被作为命令不会显示。
    ! 输入：AL -- 欲写字符；BH -- 显示页号；BL -- 前景显示色（图形方式时）。
541
542 prnt1:  push    ax
543         push    cx
544         mov     bh,#0x00        ! 显示页面。
545         mov     cx,#0x01
546         mov     ah,#0x0e
547         int     0x10
548         pop     cx
549         pop     ax
550         ret
551
552 ! Prints <CR> + <LF>     ! 显示回车+换行。
553
554 docr:   push    ax
555         push    cx
556         mov     bh,#0x00
557         mov     ah,#0x0e
558         mov     al,#0x0a
559         mov     cx,#0x01
560         int     0x10
```

```
561         mov     al,#0x0d
562         int     0x10
563         pop     cx
564         pop     ax
565         ret
566
    ! 全局描述符表开始处。描述符表由多个 8 字节长的描述符项组成。这里给出了 3 个描述符项。
    ! 第 1 项无用（568 行），但须存在。第 2 项是系统代码段描述符（570-573 行），第 3 项是系
    ! 统数据段描述符(575-578 行)。
567 gdt:
568         .word   0,0,0,0         ! dummy    ! 第 1 个描述符，不用。
569
    ! 在 GDT 表中这里的偏移量是 0x08。它是内核代码段选择符的值。
570         .word   0x07FF          ! 8Mb - limit=2047  (0--2047，因此是 2048*4096=8Mb)
571         .word   0x0000          ! base address=0
572         .word   0x9A00          ! code read/exec          ! 代码段为只读、可执行。
573         .word   0x00C0          ! granularity=4096, 386   ! 颗粒度为 4096，32 位模式。
574
    ! 在 GDT 表中这里的偏移量是 0x10。它是内核数据段选择符的值。
575         .word   0x07FF          ! 8Mb - limit=2047 (2048*4096=8Mb)
576         .word   0x0000          ! base address=0
577         .word   0x9200          ! data read/write         ! 数据段为可读可写。
578         .word   0x00C0          ! granularity=4096, 386   ! 颗粒度为 4096，32 位模式。
579
    ! 下面是加载中断描述符表寄存器 idtr 的指令 lidt 要求的 6 字节操作数。前 2 字节是 IDT 表的
    ! 限长，后 4 字节是 idt 表在线性地址空间中的 32 位基地址。CPU 要求在进入保护模式之前需设
    ! 置 IDT 表，因此这里先设置一个长度为 0 的空表。
580 idt_48:
581         .word   0               ! idt limit=0
582         .word   0,0             ! idt base=0L
583
    ! 这是加载全局描述符表寄存器 gdtr 的指令 lgdt 要求的 6 字节操作数。前 2 字节是 gdt 表的限
    ! 长，后 4 字节是 gdt 表的线性基地址。这里全局表长度设置为 2KB（0x7ff 即可），因为每 8
    ! 字节组成一个段描述符项，所以表中共可有 256 项。4 字节的线性基地址为 0x0009<<16 +
    ! 0x0200 + gdt，即 0x90200 + gdt。(符号 gdt 是全局表在本程序段中的偏移地址，见 205 行)
584 gdt_48:
585         .word   0x800           ! gdt limit=2048, 256 GDT entries
586         .word   512+gdt,0x9     ! gdt base = 0X9xxxx
587
588 msg1:   .ascii  "Press <RETURN> to see SVGA-modes available or any other key to continue."
589                 db      0x0d, 0x0a, 0x0a, 0x00
590 msg2:           .ascii  "Mode:  COLSxROWS:"
591                 db      0x0d, 0x0a, 0x0a, 0x00
592 msg3:           .ascii  "Choose mode by pressing the corresponding number."
593                 db      0x0d, 0x0a, 0x00
594
    ! 下面是 4 个显示卡的特征数据串。
595 idati:          .ascii  "761295520"
596 idcandt:        .byte   0xa5                    ! 标号 idcandt 意思是 ID of Chip AND Tech.
597 idgenoa:        .byte   0x77, 0x00, 0x66, 0x99
598 idparadise:     .ascii  "VGA="
599
    ! 下面是各种显示卡可使用的扩展模式个数和对应的模式号列表。其中每一行第 1 个字节是模式个
```

```
    ! 数值，随后的一些值是中断 0x10 功能 0（AH=0）可使用的模式号。例如从 602 行可知，对于 ATI
    ! 牌子的显示卡，除了标准模式以外还可使用两种扩展模式：0x23 和 0x33。
600 ! Manufacturer:   Numofmodes:   Mode:
    ! 厂家：          模式数量：     模式列表：
601
602 moati:          .byte   0x02,   0x23, 0x33
603 moahead:        .byte   0x05,   0x22, 0x23, 0x24, 0x2f, 0x34
604 mocandt:        .byte   0x02,   0x60, 0x61
605 mocirrus:       .byte   0x04,   0x1f, 0x20, 0x22, 0x31
606 moeverex:       .byte   0x0a,   0x03, 0x04, 0x07, 0x08, 0x0a, 0x0b, 0x16, 0x18, 0x21, 0x40
607 mogenoa:        .byte   0x0a,   0x58, 0x5a, 0x60, 0x61, 0x62, 0x63, 0x64, 0x72, 0x74, 0x78
608 moparadise:     .byte   0x02,   0x55, 0x54
609 motrident:      .byte   0x07,   0x50, 0x51, 0x52, 0x57, 0x58, 0x59, 0x5a
610 motseng:        .byte   0x05,   0x26, 0x2a, 0x23, 0x24, 0x22
611 movideo7:       .byte   0x06,   0x40, 0x43, 0x44, 0x41, 0x42, 0x45
612
    ! 下面是各种牌子 VGA 显示卡可使用的模式对应的列、行值列表。例如第 615 行表示 ATI 显示卡两
    ! 种扩展模式的列、行值分别是 132 x 25、 132 x 44。
613 !                       msb = Cols   lsb = Rows:
    !                       高字节=列数  低字节=行数：
614
615 dscati:          .word   0x8419, 0x842c                          ! ATI 卡可设置列、行值。
616 dscahead:        .word   0x842c, 0x8419, 0x841c, 0xa032, 0x5042  ! Ahead 卡可设置值。
617 dsccandt:        .word   0x8419, 0x8432
618 dsccirrus:       .word   0x8419, 0x842c, 0x841e, 0x6425
619 dsceverex:       .word   0x5022, 0x503c, 0x642b, 0x644b, 0x8419, 0x842c, 0x501e, 0x641b, 0xa040,
0x841e
620 dscgenoa:        .word   0x5020, 0x642a, 0x8419, 0x841d, 0x8420, 0x842c, 0x843c, 0x503c, 0x5042,
0x644b
621 dscparadise:     .word   0x8419, 0x842b
622 dsctrident:      .word   0x501e, 0x502b, 0x503c, 0x8419, 0x841e, 0x842b, 0x843c
623 dsctseng:        .word   0x503c, 0x6428, 0x8419, 0x841c, 0x842c
624 dscvideo7:      .word   0x502b, 0x503c, 0x643c, 0x8419, 0x842c, 0x841c
625
626 .text
627 endtext:
628 .data
629 enddata:
630 .bss
631 endbss:
```

6.3 程序 6-3 linux/boot/head.s

```
1 /*
2  *  linux/boot/head.s
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7 /*
8  *  head.s contains the 32-bit startup code.
9  *
10 * NOTE!!! Startup happens at absolute address 0x00000000, which is also where
11 * the page directory will exist. The startup code will be overwritten by
12 * the page directory.
13 */
   /*
    *  head.s 含有 32 位启动代码。
    * 注意!!! 32 位启动代码是从绝对地址 0x00000000 开始的，这里也同样是页目录将存在的地方，
    * 因此这里的启动代码将被页目录覆盖掉。
    */
14 .text
15 .globl _idt,_gdt,_pg_dir,_tmp_floppy_area
16 _pg_dir:                       # 页目录将会存放在这里。

   # 再次注意!!! 这里已经处于 32 位运行模式，因此这里的$0x10 并不是把地址 0x10 装入各个
   # 段寄存器，它现在其实是全局段描述符表中的偏移值，或者更准确地说是一个描述符表项
   # 的选择符。有关选择符的说明请参见 setup.s 中 193 行下的说明。这里$0x10 的含义是请求
   # 特权级 0(位 0-1=0)、选择全局描述符表(位 2=0)、选择表中第 2 项(位 3-15=2)。它正好指
   # 向表中的数据段描述符项。（描述符的具体数值参见前面 setup.s 中 212，213 行）
   # 下面代码的含义是：设置 ds,es,fs,gs 为 setup.s 中构造的数据段（全局段描述符表第 2 项）
   # 的选择符=0x10，并将堆栈放置在 stack_start 指向的 user_stack 数组区，然后使用本程序
   # 后面定义的新中断描述符表和全局段描述表。新全局段描述表中初始内容与 setup.s 中的基本
   # 一样，仅段限长从 8MB 修改成了 16MB。stack_start 定义在 kernel/sched.c，69 行。它是指向
   # user_stack 数组末端的一个长指针。第 23 行设置这里使用的栈，姑且称为系统栈。但在移动到
   # 任务 0 执行（init/main.c 中 137 行）以后该栈就被用作任务 0 和任务 1 共同使用的用户栈了。

17 startup_32:                  # 18-22 行设置各个数据段寄存器。
18         movl $0x10,%eax      # 对于 GNU 汇编，每个直接操作数要以'$'开始，否则表示地址。
                                # 每个寄存器名都要以'%'开头，eax 表示是 32 位的 ax 寄存器。
19         mov %ax,%ds
20         mov %ax,%es
21         mov %ax,%fs
22         mov %ax,%gs
```

```
23          lss _stack_start,%esp       # 表示_stack_start➔ss:esp，设置系统堆栈。
                                        # stack_start 定义在 kernel/sched.c，69 行。
24          call setup_idt              # 调用设置中断描述符表子程序。
25          call setup_gdt              # 调用设置全局描述符表子程序。
26          movl $0x10,%eax             # reload all the segment registers
27          mov %ax,%ds                 # after changing gdt. CS was already
28          mov %ax,%es                 # reloaded in 'setup_gdt'
29          mov %ax,%fs                 # 因为修改了 gdt，所以需要重新装载所有的段寄存器。
30          mov %ax,%gs                 # CS 代码段寄存器已经在 setup_gdt 中重新加载过了。

    # 由于段描述符中的段限长从 setup.s 中的 8MB 改成了本程序设置的 16MB（见 setup.s 行 208-216
    # 和本程序后面的 235-236 行），因此这里再次对所有段寄存器执行加载操作是必须的。另外，通过
    # 使用 bochs 跟踪观察，如果不对 CS 再次执行加载，那么在执行到 26 行时 CS 代码段不可见部分中
    # 的限长还是 8MB。这样看来应该重新加载 CS。但是由于 setup.s 中的内核代码段描述符与本程序中
    # 重新设置的代码段描述符除了段限长以外其余部分完全一样，8MB 的限长在内核初始化阶段不会有
    # 问题，而且在以后内核执行过程中段间跳转时会重新加载 CS。因此这里没有加载它并没有让程序
    # 出错。
    # 针对该问题，目前内核中就在第 25 行之后添加了一条长跳转指令：'ljmp $(__KERNEL_CS),$1f'，
    # 跳转到第 26 行来确保 CS 确实又被重新加载。

31          lss _stack_start,%esp

    # 32-36 行用于测试 A20 地址线是否已经开启。采用的方法是向内存地址 0x000000 处写入任意
    # 一个数值，然后看内存地址 0x100000(1M)处是否也是这个数值。如果一直相同的话，就一直
    # 比较下去，也即死循环、死机。表示地址 A20 线没有选通，结果内核就不能使用 1MB 以上内存。
    #
    # 33 行上的'1:'是一个局部符号构成的标号。标号由符号后跟一个冒号组成。此时该符号表示活动
    # 位置计数（Active location counter）的当前值，并可以作为指令的操作数。局部符号用于帮助
    # 编译器和编程人员临时使用一些名称。共有 10 个局部符号名，可在整个程序中重复使用。这些符号
    # 名使用名称'0'、'1'、...、'9'来引用。为了定义一个局部符号，需把标号写成'N:'形式（其中 N
    # 表示一个数字）。为了引用先前最近定义的这个符号，需要写成'Nb'，其中 N 是定义标号时使用的
    # 数字。为了引用一个局部标号的下一个定义，需要写成'Nf'，这里 N 是 10 个前向引用之一。上面
    # 'b'表示“向后（backwards）”，'f'表示“向前（forwards）”。在汇编程序的某一处，我们最大
    # 可以向后/向前引用 10 个标号（最远第 10 个）。

32          xorl %eax,%eax
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                       # '1b'表示向后(backward)跳转到标号 1 去（33 行）。
                                        # 若是'5f'则表示向前(forward)跳转到标号 5 去。
37  /*
38   * NOTE! 486 should set bit 16, to check for write-protect in supervisor
39   * mode. Then it would be unnecessary with the "verify_area()"-calls.
```

```
40   * 486 users probably want to set the NE (#5) bit also, so as to use
41   * int 16 for math errors.
42   */
     /*
      * 注意！在下面这段程序中，486 应该将位 16 置位，以检查在超级用户模式下的写保护，
      * 此后 "verify_area()" 调用就不需要了。486 的用户通常也会想将 NE(#5)置位，以便
      * 对数学协处理器的出错使用 int 16。
      */
     # 上面原注释中提到的 486 CPU 中 CR0 控制寄存器的位 16 是写保护标志 WP（Write-Protect），
     # 用于禁止超级用户级的程序向一般用户只读页面中进行写操作。该标志主要用于操作系统在创建
     # 新进程时实现写时复制（copy-on-write）方法。
     # 下面这段程序（43-65）用于检查数学协处理器芯片是否存在。方法是修改控制寄存器 CR0，在
     # 假设存在协处理器的情况下执行一个协处理器指令，如果出错的话则说明协处理器芯片不存在，
     # 需要设置 CR0 中的协处理器仿真位 EM（位 2），并复位协处理器存在标志 MP（位 1）。

43           movl %cr0,%eax          # check math chip
44           andl $0x80000011,%eax   # Save PG,PE,ET
45   /* "orl $0x10020,%eax" here for 486 might be good */
46           orl $2,%eax             # set MP
47           movl %eax,%cr0
48           call check_x87
49           jmp after_page_tables   # 跳转到 135 行。
50
51   /*
52    * We depend on ET to be correct. This checks for 287/387.
53    */
     /*
      * 我们依赖于 ET 标志的正确性来检测 287/387 存在与否。
      */
     # 下面 fninit 和 fstsw 是数学协处理器（80287/80387）的指令。
     # finit 向协处理器发出初始化命令，它会把协处理器置于一个未受以前操作影响的已知状态，设置
     # 其控制字为默认值、清除状态字和所有浮点栈式寄存器。非等待形式的这条指令（fninit）还会让
     # 协处理器终止执行当前正在执行的任何先前的算术操作。fstsw 指令取协处理器的状态字。如果系
     # 统中存在协处理器的话，那么在执行了 fninit 指令后其状态字低字节肯定为 0。

54   check_x87:
55           fninit                  # 向协处理器发出初始化命令。
56           fstsw %ax               # 取协处理器状态字到 ax 寄存器中。
57           cmpb $0,%al             # 初始化后状态字应该为 0，否则说明协处理器不存在。
58           je 1f                   /* no coprocessor: have to set bits */
59           movl %cr0,%eax          # 如果存在则向前跳转到标号 1 处，否则改写 cr0。
60           xorl $6,%eax            /* reset MP, set EM */
61           movl %eax,%cr0
62           ret
```

```
# 下面是一汇编语言指示符。其含义是指存储边界对齐调整。"2"表示把随后的代码或数据的偏移位置
# 调整到地址值最后 2 比特位为零的位置（2^2），即按 4 字节方式对齐内存地址。不过现在 GNU as
# 直接时写出对齐的值而非 2 的次方值了。使用该指示符的目的是为了提高 32 位 CPU 访问内存中代码
# 或数据的速度和效率。参见程序后的详细说明。
# 下面的两个字节值是 80287 协处理器指令 fsetpm 的机器码。其作用是把 80287 设置为保护模式。
# 80387 无需该指令，并且将会把该指令看作是空操作。

.align 2
1:      .byte 0xDB,0xE4         /* fsetpm for 287, ignored by 387 */    # 287 协处理器码。
        ret

/*
 *  setup_idt
 *
 *  sets up a idt with 256 entries pointing to
 *  ignore_int, interrupt gates. It then loads
 *  idt. Everything that wants to install itself
 *  in the idt-table may do so themselves. Interrupts
 *  are enabled elsewhere, when we can be relatively
 *  sure everything is ok. This routine will be over-
 *  written by the page tables.
 */
/*
 * 下面这段是设置中断描述符表子程序 setup_idt
 *
 * 将中断描述符表 idt 设置成具有 256 个项，并都指向 ignore_int 中断门。然后加载中断
 * 描述符表寄存器(用 lidt 指令)。真正实用的中断门以后再安装。当我们在其他地方认为一切
 * 都正常时再开启中断。该子程序将会被页表覆盖掉。
 */
# 中断描述符表中的项虽然也是 8 字节组成，但其格式与全局表中的不同，被称为门描述符
# (Gate Descriptor)。它的 0-1,6-7 字节是偏移量，2-3 字节是选择符，4-5 字节是一些标志。
# 这段代码首先在 edx、eax 中组合设置出 8 字节默认的中断描述符值，然后在 idt 表每一项中
# 都放置该描述符，共 256 项。eax 含有描述符低 4 字节，edx 含有高 4 字节。内核在随后的初始
# 化过程中会替换安装那些真正实用的中断描述符项。

setup_idt:
        lea ignore_int,%edx         # 将 ignore_int 的有效地址（偏移值）值➔edx 寄存器
        movl $0x00080000,%eax       # 将选择符 0x0008 置入 eax 的高 16 位中。
        movw %dx,%ax                  /* selector = 0x0008 = cs */
                                    # 偏移值的低 16 位置入 eax 的低 16 位中。此时 eax 含有
                                    # 门描述符低 4 字节的值。
        movw $0x8E00,%dx              /* interrupt gate - dpl=0, present */
                                    # 此时 edx 含有门描述符高 4 字节的值。
```

```
84              lea _idt,%edi                   # _idt 是中断描述符表的地址。
85              mov $256,%ecx
86 rp_sidt:
87              movl %eax,(%edi)                 # 将哑中断门描述符存入表中。
88              movl %edx,4(%edi)                # eax 内容放到 edi+4 所指内存位置处。
89              addl $8,%edi                    # edi 指向表中下一项。
90              dec %ecx
91              jne rp_sidt
92              lidt idt_descr                  # 加载中断描述符表寄存器值。
93              ret
94
95 /*
96   *   setup_gdt
97   *
98   *   This routines sets up a new gdt and loads it.
99   *   Only two entries are currently built, the same
100  *   ones that were built in init.s. The routine
101  *   is VERY complicated at two whole lines, so this
102  *   rather long comment is certainly needed :-).
103  *   This routine will beoverwritten by the page tables.
104  */
     /*
      * 设置全局描述符表项 setup_gdt
      * 这个子程序设置一个新的全局描述符表 gdt，并加载。此时仅创建了两个表项，与前
      * 面的一样。该子程序只有两行，“非常的”复杂，所以当然需要这么长的注释了☺。
      * 该子程序将被页表覆盖掉。
      */
105 setup_gdt:
106             lgdt gdt_descr                  # 加载全局描述符表寄存器(内容已设置好，见 234-238 行)。
107             ret
108
109 /*
110  * I put the kernel page tables right after the page directory,
111  * using 4 of them to span 16 Mb of physical memory. People with
112  * more than 16MB will have to expand this.
113  */
     /* Linus 将内核的内存页表直接放在页目录之后，使用了 4 个表来寻址 16 MB 的物理内存。
      * 如果你有多于 16 Mb 的内存，就需要在这里进行扩充修改。
      */
      # 每个页表长为 4 Kb 字节（1 页内存页面），而每个页表项需要 4 个字节，因此一个页表共可以存放
      # 1024 个表项。如果一个页表项寻址 4 KB 的地址空间，则一个页表就可以寻址 4 MB 的物理内存。
      # 页表项的格式为：项的前 0-11 位存放一些标志，例如是否在内存中(P 位 0)、读写许可(R/W 位 1)、
      # 普通用户还是超级用户使用(U/S 位 2)、是否修改过(是否脏了)(D 位 6)等；表项的位 12-31 是
      # 页框地址，用于指出一页内存的物理起始地址。
```

```
114 .org 0x1000          # 从偏移 0x1000 处开始是第 1 个页表（偏移 0 开始处将存放页表目录）。
115 pg0:
116
117 .org 0x2000
118 pg1:
119
120 .org 0x3000
121 pg2:
122
123 .org 0x4000
124 pg3:
125
126 .org 0x5000             # 定义下面的内存数据块从偏移 0x5000 处开始。
127 /*
128  * tmp_floppy_area is used by the floppy-driver when DMA cannot
129  * reach to a buffer-block. It needs to be aligned, so that it isn't
130  * on a 64kB border.
131  */
    /* 当 DMA（直接存储器访问）不能访问缓冲块时，下面的 tmp_floppy_area 内存块
     * 就可供软盘驱动程序使用。其地址需要对齐调整，这样就不会跨越 64KB 边界。
     */
132 _tmp_floppy_area:
133             .fill 1024,1,0                  # 共保留 1024 项，每项 1 字节，填充数值 0。
134
    # 下面这几个入栈操作用于为跳转到 init/main.c 中的 main()函数作准备工作。第 139 行上
    # 的指令在栈中压入了返回地址，而第 140 行则压入了 main()函数代码的地址。当 head.s
    # 最后在第 218 行执行 ret 指令时就会弹出 main()的地址，并把控制权转移到 init/main.c
    # 程序中。参见第 3 章中有关 C 函数调用机制的说明。
    # 前面 3 个入栈 0 值应该分别表示 envp、argv 指针和 argc 的值，但 main()没有用到。
    # 139 行的入栈操作是模拟调用 main.c 程序时首先将返回地址入栈的操作，所以如果
    # main.c 程序真的退出时，就会返回到这里的标号 L6 处继续执行下去，也即死循环。
    # 140 行将 main.c 的地址压入堆栈，这样，在设置分页处理（setup_paging）结束后
    # 执行'ret'返回指令时就会将 main.c 程序的地址弹出堆栈，并去执行 main.c 程序了。
    # 有关 C 函数调用机制请参见程序后的说明。
135 after_page_tables:
136             pushl $0                        # These are the parameters to main :-)
137             pushl $0                        # 这些是调用 main 程序的参数（指 init/main.c）。
138             pushl $0                        # 其中的'$'符号表示这是一个立即操作数。
139             pushl $L6                       # return address for main, if it decides to.
140             pushl $_main                    # '_main'是编译程序对 main 的内部表示方法。
141             jmp setup_paging                # 跳转至第 198 行。
142 L6:
143             jmp L6                          # main should never return here, but
```

```
144                                     # just in case, we know what happens.
                                        # main 程序绝对不应该返回到这里。不过为了以防万一，
                                        # 所以添加了该语句。这样我们就知道发生什么问题了。
145
146 /* This is the default interrupt "handler" :-) */
    /* 下面是默认的中断“向量句柄”☺ */
147 int_msg:
148         .asciz "Unknown interrupt\n\r"      # 定义字符串“未知中断(回车换行)”。
149 .align 2                       # 按 4 字节方式对齐内存地址。
150 ignore_int:
151         pushl %eax
152         pushl %ecx
153         pushl %edx
154         push %ds                # 这里请注意!! ds,es,fs,gs 等虽然是 16 位的寄存器，但入栈后
155         push %es                # 仍然会以 32 位的形式入栈，也即需要占用 4 个字节的堆栈空间。
156         push %fs
157         movl $0x10,%eax         # 置段选择符（使 ds,es,fs 指向 gdt 表中的数据段）。
158         mov %ax,%ds
159         mov %ax,%es
160         mov %ax,%fs
161         pushl $int_msg          # 把调用 printk 函数的参数指针（地址）入栈。注意！若 int_msg
162         call _printk            # 前不加'$'，则表示把 int_msg 符号处的长字（'Unkn'）入栈☺。
163         popl %eax               # 该函数在/kernel/printk.c 中。'_printk'是 printk 编译后模块中
164         pop %fs                 # 的内部表示法。
165         pop %es
166         pop %ds
167         popl %edx
168         popl %ecx
169         popl %eax
170         iret                    # 中断返回（把中断调用时压入栈的 CPU 标志寄存器（32 位）值也弹出）。
171
172
173 /*
174  * Setup_paging
175  *
176  * This routine sets up paging by setting the page bit
177  * in cr0. The page tables are set up, identity-mapping
178  * the first 16MB. The pager assumes that no illegal
179  * addresses are produced (ie >4Mb on a 4Mb machine).
180  *
181  * NOTE! Although all physical memory should be identity
182  * mapped by this routine, only the kernel page functions
183  * use the >1Mb addresses directly. All "normal" functions
184  * use just the lower 1Mb, or the local data space, which
```

```
185   * will be mapped to some other place - mm keeps track of
186   * that.
187   *
188   * For those with more memory than 16 Mb - tough luck. I've
189   * not got it, why should you :-) The source is here. Change
190   * it. (Seriously - it shouldn't be too difficult. Mostly
191   * change some constants etc. I left it at 16Mb, as my machine
192   * even cannot be extended past that (ok, but it was cheap :-)
193   * I've tried to show which constants to change by having
194   * some kind of marker at them (search for "16Mb"), but I
195   * won't guarantee that's all :-( )
196   */
      /*
       * 这个子程序通过设置控制寄存器 cr0 的标志（PG 位 31）来启动对内存的分页处理功能，
       * 并设置各个页表项的内容，以恒等映射前 16 MB 的物理内存。分页器假定不会产生非法的
       * 地址映射（也即在只有 4Mb 的机器上设置出大于 4Mb 的内存地址）。
       *
       * 注意！尽管所有的物理地址都应该由这个子程序进行恒等映射，但只有内核页面管理函数能
       * 直接使用>1Mb 的地址。所有“普通”函数仅使用低于 1Mb 的地址空间，或者是使用局部数据
       * 空间，该地址空间将被映射到其他一些地方去 -- mm（内存管理程序）会管理这些事的。
       *
       * 对于那些有多于 16Mb 内存的家伙 – 真是太幸运了，我还没有，为什么你会有☺。代码就在
       * 这里，对它进行修改吧。（实际上，这并不太困难的。通常只需修改一些常数等。我把它设置
       * 为 16Mb，因为我的机器再怎么扩充甚至不能超过这个界限（当然，我的机器是很便宜的☺）。
       * 我已经通过设置某类标志来给出需要改动的地方（搜索“16Mb”），但我不能保证作这些
       * 改动就行了☹）。
       */
      # 上面英文注释第 2 段的含义是指在机器物理内存中大于 1MB 的内存空间主要被用于主内存区。
      # 主内存区空间由 mm 模块管理。它涉及到页面映射操作。内核中所有其他函数就是这里指的一般
      #（普通）函数。若要使用主内存区的页面，就需要使用 get_free_page()等函数获取。因为主内
      # 存区中内存页面是共享资源，必须有程序进行统一管理以避免资源争用和竞争。
      #
      # 在内存物理地址 0x0 处开始存放 1 页页目录表和 4 页页表。页目录表是系统所有进程公用的，而
      # 这里的 4 页页表则属于内核专用，它们一一映射线性地址起始 16MB 空间范围到物理内存上。对于
      # 新的进程，系统会在主内存区为其申请页面存放页表。另外，1 页内存长度是 4096 字节。

197 .align 2                            # 按 4 字节方式对齐内存地址边界。
198 setup_paging:                       # 首先对 5 页内存（1 页目录 + 4 页页表）清零。
199         movl $1024*5,%ecx             /* 5 pages - pg_dir+4 page tables */
200         xorl %eax,%eax
201         xorl %edi,%edi                /* pg_dir is at 0x000 */
                                          # 页目录从 0x000 地址开始。
202         cld;rep;stosl                 # eax 内容存到 es:edi 所指内存位置处，且 edi 增 4。
```

```
    # 下面 4 句设置页目录表中的项，因为我们（内核）共有 4 个页表所以只需设置 4 项。
    # 页目录项的结构与页表中项的结构一样，4 个字节为 1 项。参见上面 113 行下的说明。
    # 例如"$pg0+7"表示：0x00001007，是页目录表中的第 1 项。
    # 则第 1 个页表所在的地址 = 0x00001007 & 0xfffff000 = 0x1000；
    # 第 1 个页表的属性标志 = 0x00001007 & 0x00000fff = 0x07，表示该页存在、用户可读写。
203             movl $pg0+7,_pg_dir             /* set present bit/user r/w */
204             movl $pg1+7,_pg_dir+4           /*    --------- " " --------- */
205             movl $pg2+7,_pg_dir+8           /*    --------- " " --------- */
206             movl $pg3+7,_pg_dir+12          /*    --------- " " --------- */

    # 下面 6 行填写 4 个页表中所有项的内容，共有：4(页表)*1024(项/页表)=4096 项(0 - 0xfff)，
    # 也即能映射物理内存 4096*4Kb = 16Mb。
    # 每项的内容是：当前项所映射的物理内存地址 + 该页的标志（这里均为 7）。
    # 使用的方法是从最后一个页表的最后一项开始按倒退顺序填写。一个页表的最后一项在页表中的
    # 位置是 1023*4 = 4092。因此最后一页的最后一项的位置就是$pg3+4092。

207             movl $pg3+4092,%edi             # edi➔最后一页的最后一项。
208             movl $0xfff007,%eax             /*    16Mb - 4096 + 7 (r/w user,p) */
                                                 # 最后 1 项对应物理内存页面的地址是 0xfff000，
                                                 # 加上属性标志 7，即为 0xfff007。
209             std                              # 方向位置位，edi 值递减(4 字节)。
210 1:          stosl                           /* fill pages backwards - more efficient :-) */
211             subl $0x1000,%eax                # 每填写好一项，物理地址值减 0x1000。
212             jge 1b                           # 如果小于 0 则说明全添写好了。
    # 设置页目录表基址寄存器 cr3 的值，指向页目录表。cr3 中保存的是页目录表的物理地址。
213             xorl %eax,%eax                   /* pg_dir is at 0x0000 */     # 页目录表在 0x0000 处。
214             movl %eax,%cr3                   /* cr3 - page directory start */
    # 设置启动使用分页处理（cr0 的 PG 标志，位 31）
215             movl %cr0,%eax
216             orl $0x80000000,%eax            # 添上 PG 标志。
217             movl %eax,%cr0                   /* set paging (PG) bit */
218             ret                              /* this also flushes prefetch-queue */

    # 在改变分页处理标志后要求使用转移指令刷新预取指令队列，这里用的是返回指令 ret。
    # 该返回指令的另一个作用是将 140 行压入堆栈中的 main 程序的地址弹出，并跳转到/init/main.c
    # 程序去运行。本程序到此就真正结束了。

219
220 .align 2                                   # 按 4 字节方式对齐内存地址边界。
221 .word 0                                    # 这里先空出 2 字节，这样 224 行上的长字是 4 字节对齐的。

    ! 下面是加载中断描述符表寄存器 idtr 的指令 lidt 要求的 6 字节操作数。前 2 字节是 idt 表的限长，
    ! 后 4 字节是 idt 表在线性地址空间中的 32 位基地址。
222 idt_descr:
```

```
223             .word 256*8-1                   # idt contains 256 entries    #  共 256 项，限长=长度 - 1。
224             .long _idt
225 .align 2
226 .word 0
```

```
    ！下面加载全局描述符表寄存器 gdtr 的指令 lgdt 要求的 6 字节操作数。前 2 字节是 gdt 表的限长，
    ！后 4 字节是 gdt 表的线性基地址。这里全局表长度设置为 2KB 字节（0x7ff 即可），因为每 8 字节
    ！组成一个描述符项，所以表中共可有 256 项。符号_gdt 是全局表在本程序中的偏移位置，见 234 行。
227 gdt_descr:
228             .word 256*8-1                    # so does gdt (not that that's any      #  注：not → note
229             .long _gdt                      # magic number, but it works for me :^)
230
231             .align 3                        #  按 8（2^3）字节方式对齐内存地址边界。
232 _idt:     .fill 256,8,0                   # idt is uninitialized      # 256 项，每项 8 字节，填 0。
233
    #  全局表。前 4 项分别是空项（不用）、代码段描述符、数据段描述符、系统调用段描述符，其中
    #  系统调用段描述符并没有派用处，Linus 当时可能曾想把系统调用代码专门放在这个独立的段中。
    #  后面还预留了 252 项的空间，用于放置所创建任务的局部描述符(LDT)和对应的任务状态段 TSS
    #  的描述符。
    # (0-nul, 1-cs, 2-ds, 3-syscall, 4-TSS0, 5-LDT0, 6-TSS1, 7-LDT1, 8-TSS2 etc...)
```

```
234 _gdt:     .quad 0x0000000000000000          /* NULL descriptor */
235             .quad 0x00c09a0000000fff          /* 16Mb */          # 0x08，内核代码段最大长度 16MB。
236             .quad 0x00c0920000000fff          /* 16Mb */          # 0x10，内核数据段最大长度 16MB。
237             .quad 0x0000000000000000          /* TEMPORARY - don't use */
238             .fill 252,8,0                   /* space for LDT's and TSS's etc */   #  预留空间。
```

第7章 内核初始化程序

7.1 程序 7-1 linux/init/main.c

```
1  /*
2   *  linux/init/main.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
   // 定义宏“__LIBRARY__”是为了包括定义在 unistd.h 中的内嵌汇编代码等信息。
7  #define __LIBRARY__
   // *.h 头文件所在的默认目录是 include/，则在代码中就不用明确指明其位置。如果不是 UNIX 的
   // 标准头文件，则需要指明所在的目录，并用双引号括住。unistd.h 是标准符号常数与类型文件。
   // 其中定义了各种符号常数和类型，并声明了各种函数。如果还定义了符号__LIBRARY__，则还会
   // 包含系统调用号和内嵌汇编代码 syscall0()等。
8  #include <unistd.h>
9  #include <time.h>    // 时间类型头文件。其中最主要定义了 tm 结构和一些有关时间的函数原形。
10
11 /*
12  * we need this inline - forking from kernel space will result
13  * in NO COPY ON WRITE (!!!), until an execve is executed. This
14  * is no problem, but for the stack. This is handled by not letting
15  * main() use the stack at all after fork(). Thus, no function
16  * calls - which means inline code for fork too, as otherwise we
17  * would use the stack upon exit from 'fork()'.
18  *
19  * Actually only pause and fork are needed inline, so that there
20  * won't be any messing with the stack from main(), but we define
21  * some others too.
22  */
   /*
    * 我们需要下面这些内嵌语句 - 从内核空间创建进程将导致没有写时复制(COPY ON WRITE)!!!
    * 直到执行一个 execve 调用。这对堆栈可能带来问题。处理方法是在 fork()调用后不让 main()
    * 使用任何堆栈。因此就不能有函数调用 - 这意味着 fork 也要使用内嵌的代码，否则我们在从
    * fork()退出时就要使用堆栈了。
    *
    * 实际上只有 pause 和 fork 需要使用内嵌方式，以保证从 main()中不会弄乱堆栈，但是我们同
    * 时还定义了其他一些函数。
    */
   // Linux 在内核空间创建进程时不使用写时复制技术（Copy on write）。main()在移动到用户
   // 模式（到任务 0）后执行内嵌方式的 fork()和 pause()，因此可保证不使用任务 0 的用户栈。
   // 在执行 moveto_user_mode()之后，本程序 main()就以任务 0 的身份在运行了。而任务 0 是所
   // 有将创建子进程的父进程。当它创建一个子进程时（init 进程），由于任务 1 代码属于内核
   // 空间，因此没有使用写时复制功能。此时任务 0 的用户栈就是任务 1 的用户栈，即它们共同
   // 使用一个栈空间。因此希望在 main.c 运行在任务 0 的环境下时不要有对堆栈的任何操作，以
   // 免弄乱堆栈。而在再次执行 fork()并执行过 execve()函数后，被加载程序已不属于内核空间，
```

```
   // 因此可以使用写时复制技术了。请参见 5.3 节“Linux 内核使用内存的方法”内容。

   // 下面_syscall0()是 unistd.h 中的内嵌宏代码。以嵌入汇编的形式调用 Linux 的系统调用中断
   // 0x80。该中断是所有系统调用的入口。该条语句实际上是 int fork()创建进程系统调用。可展
   // 开看之就会立刻明白。syscall0 名称中最后的 0 表示无参数，1 表示 1 个参数。
   // 参见 include/unistd.h，133 行。
23 static inline _syscall0(int,fork)
   // int pause()系统调用：暂停进程的执行，直到收到一个信号。
24 static inline _syscall0(int,pause)
   // int setup(void * BIOS)系统调用，仅用于 linux 初始化（仅在这个程序中被调用）。
25 static inline _syscall1(int,setup,void *,BIOS)
   // int sync()系统调用：更新文件系统。
26 static inline _syscall0(int,sync)
27
28 #include <linux/tty.h>     // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
29 #include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、第 1 个初始任务
                              // 的数据。还有一些以宏的形式定义的有关描述符参数设置和获取的
                              // 嵌入式汇编函数程序。
30 #include <linux/head.h>    // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
31 #include <asm/system.h>    // 系统头文件。以宏形式定义了许多有关设置或修改描述符/中断门
                              // 等的嵌入式汇编子程序。
32 #include <asm/io.h>        // io 头文件。以宏的嵌入汇编程序形式定义对 io 端口操作的函数。
33
34 #include <stddef.h>        // 标准定义头文件。定义了 NULL, offsetof(TYPE, MEMBER)。
35 #include <stdarg.h>        // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了-个
                              // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，vsprintf、
                              // vprintf、vfprintf。
36 #include <unistd.h>
37 #include <fcntl.h>         // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
38 #include <sys/types.h>     // 类型头文件。定义了基本的系统数据类型。
39
40 #include <linux/fs.h>      // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
41                            // 其中有定义：extern int ROOT_DEV。
42 #include <string.h>        // 字符串头文件。主要定义了一些有关内存或字符串操作的嵌入函数。
43
44 static char printbuf[1024];           // 静态字符串数组，用作内核显示信息的缓存。
45
46 extern char *strcpy();
47 extern int vsprintf();                // 送格式化输出到一字符串中（vsprintf.c，92 行）。
48 extern void init(void);               // 函数原形，初始化（本程序 168 行）。
49 extern void blk_dev_init(void);       // 块设备初始化子程序（blk_drv/ll_rw_blk.c,157 行）
50 extern void chr_dev_init(void);       // 字符设备初始化（chr_drv/tty_io.c，347 行）
51 extern void hd_init(void);            // 硬盘初始化程序（blk_drv/hd.c，343 行）
52 extern void floppy_init(void);        // 软驱初始化程序（blk_drv/floppy.c，457 行）
53 extern void mem_init(long start, long end);      // 内存管理初始化（mm/memory.c，399 行）
54 extern long rd_init(long mem_start, int length); // 虚拟盘初始化(blk_drv/ramdisk.c,52)
55 extern long kernel_mktime(struct tm * tm);       // 计算系统开机启动时间（秒）。
56
   // 内核专用 sprintf()函数。该函数用于产生格式化信息并输出到指定缓冲区 str 中。参数'*fmt'
   // 指定输出将采用的格式，参见标准 C 语言书籍。该子程序正好是 vsprintf 如何使用的一个简单
   // 例子。函数使用 vsprintf()将格式化字符串放入 str 缓冲区，参见第 179 行上的 printf()函数。
57 static int sprintf(char * str, const char *fmt, ...)
58 {
```

```
59          va_list args;
60          int i;
61
62          va_start(args, fmt);
63          i = vsprintf(str, fmt, args);
64          va_end(args);
65          return i;
66  }
67
68  /*
69   * This is set up by the setup-routine at boot-time
70   */
    /*
     * 以下这些数据是在内核引导期间由 setup.s 程序设置的。
     */
    // 下面三行分别将指定的线性地址强行转换为给定数据类型的指针，并获取指针所指内容。由于
    // 内核代码段被映射到从物理地址零开始的地方，因此这些线性地址正好也是对应的物理地址。
    // 这些指定地址处内存值的含义请参见第 6 章的表 6-2（setup 程序读取并保存的参数）。
    // drive_info 结构请参见下面第 125 行。
71  #define EXT_MEM_K (*(unsigned short *)0x90002)      // 1MB 以后的扩展内存大小（KB）。
72  #define CON_ROWS ((*(unsigned short *)0x9000e) & 0xff)  // 选定的控制台屏幕行、列数。
73  #define CON_COLS (((*(unsigned short *)0x9000e) & 0xff00) >> 8)
74  #define DRIVE_INFO (*(struct drive_info *)0x90080)  // 硬盘参数表 32 字节内容。
75  #define ORIG_ROOT_DEV (*(unsigned short *)0x901FC)  // 根文件系统所在设备号。
76  #define ORIG_SWAP_DEV (*(unsigned short *)0x901FA)  // 交换文件所在设备号。
77
78  /*
79   * Yeah, yeah, it's ugly, but I cannot find how to do this correctly
80   * and this seems to work. I anybody has more info on the real-time
81   * clock I'd be interested. Most of this was trial and error, and some
82   * bios-listing reading. Urghh.
83   */
    /*
     * 是啊，是啊，下面这段程序很差劲，但我不知道如何正确地实现，而且好象
     * 它还能运行。如果有关于实时时钟更多的资料，那我很感兴趣。这些都是试
     * 探出来的，另外还看了一些 bios 程序，呵！
     */
84
    // 这段宏读取 CMOS 实时时钟信息。outb_p 和 inb_p 是 include/asm/io.h 中定义的端口输入输出宏。
85  #define CMOS_READ(addr) ({ \
86  outb_p(0x80|addr,0x70); \       // 0x70 是写地址端口号，0x80|addr 是要读取的 CMOS 内存地址。
87  inb_p(0x71); \                  // 0x71 是读数据端口号。
88  })
89
    // 定义宏。将 BCD 码转换成二进制数值。BCD 码利用半个字节（4 比特）表示一个 10 进制数，因此
    // 一个字节表示 2 个 10 进制数。(val)&15 取 BCD 表示的 10 进制个位数，而 (val)>>4 取 BCD 表示
    // 的 10 进制十位数，再乘以 10。因此最后两者相加就是一个字节 BCD 码的实际二进制数值。
90  #define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10)
91
    // 该函数取 CMOS 实时钟信息作为开机时间，并保存到全局变量 startup_time(秒)中。参见后面
    // CMOS 内存列表说明。其中调用的函数 kernel_mktime()用于计算从 1970 年 1 月 1 日 0 时起到
    // 开机当日经过的秒数，作为开机时间（kernel/mktime.c 41 行）。
92  static void time_init(void)
```

```
93 {
94         struct tm time;                          // 时间结构 tm 定义在 include/time.h 中。
95
   // CMOS 的访问速度很慢。为了减小时间误差，在读取了下面循环中所有数值后，若此时 CMOS 中
   // 秒值发生了变化，那么就重新读取所有值。这样内核就能把与 CMOS 时间误差控制在 1 秒之内。
96         do {
97                 time.tm_sec = CMOS_READ(0);      // 当前时间秒值（均是 BCD 码值）。
98                 time.tm_min = CMOS_READ(2);      // 当前分钟值。
99                 time.tm_hour = CMOS_READ(4);     // 当前小时值。
100                time.tm_mday = CMOS_READ(7);     // 一月中的当天日期。
101                time.tm_mon = CMOS_READ(8);      // 当前月份（1—12）。
102                time.tm_year = CMOS_READ(9);     // 当前年份。
103        } while (time.tm_sec != CMOS_READ(0));
104        BCD_TO_BIN(time.tm_sec);                 // 转换成二进制数值。
105        BCD_TO_BIN(time.tm_min);
106        BCD_TO_BIN(time.tm_hour);
107        BCD_TO_BIN(time.tm_mday);
108        BCD_TO_BIN(time.tm_mon);
109        BCD_TO_BIN(time.tm_year);
110        time.tm_mon--;                           // tm_mon 中月份范围是 0—11。
111        startup_time = kernel_mktime(&time);     // 计算开机时间。kernel/mktime.c 41 行。
112 }
113
   // 下面定义一些局部变量。
114 static long memory_end = 0;                     // 机器具有的物理内存容量（字节数）。
115 static long buffer_memory_end = 0;              // 高速缓冲区末端地址。
116 static long main_memory_start = 0;              // 主内存（将用于分页）开始的位置。
117 static char term[32];                           // 终端设置字符串（环境参数）。
118
   // 读取并执行/etc/rc 文件时所使用的命令行参数和环境参数。
119 static char * argv_rc[] = { "/bin/sh", NULL };       // 调用执行程序时参数的字符串数组。
120 static char * envp_rc[] = { "HOME=/", NULL ,NULL };  // 调用执行程序时的环境字符串数组。
121
   // 运行登录 shell 时所使用的命令行参数和环境参数。
   // 第 122 行中 argv[0]中的字符“-”是传递给 shell 程序 sh 的一个标志。通过识别该标志，
   // sh 程序会作为登录 shell 执行。其执行过程与在 shell 提示符下执行 sh 不一样。
122 static char * argv[] = { "-/bin/sh",NULL };        // 同上。
123 static char * envp[] = { "HOME=/usr/root", NULL, NULL };
124
125 struct drive_info { char dummy[32]; } drive_info;   // 用于存放硬盘参数表信息。
126
   // 内核初始化主程序。初始化结束后将以任务 0（idle 任务即空闲任务）的身份运行。
   // 英文注释含义是“这里确实是 void，没错。在 startup 程序(head.s)中就是这样假设的”。参见
   // head.s 程序第 136 行开始的几行代码。
127 void main(void)          /* This really IS void, no error here. */
128 {                        /* The startup routine assumes (well, ...) this */
129 /*
130  * Interrupts are still disabled. Do necessary setups, then
131  * enable them
132  */
    /*
     * 此时中断仍被禁止着，做完必要的设置后就将其开启。
     */
```

```
    // 首先保存根文件系统设备号和交换文件设备号，并根据 setup.s 程序中获取的信息设置控制台
    // 终端屏幕行、列数环境变量 TERM，并用其设置初始 init 进程中执行 etc/rc 文件和 shell 程序
    // 使用的环境变量，以及复制内存 0x90080 处的硬盘参数表。
    // 其中 ROOT_DEV 已在前面包含进的 include/linux/fs.h 文件第 206 行上被声明为 extern int，
    // 而 SWAP_DEV 在 include/linux/mm.h 文件内也作了相同声明。这里 mm.h 文件并没有显式地列在
    // 本程序前部，因为前面包含进的 include/linux/sched.h 文件中已经含有它。
133         ROOT_DEV = ORIG_ROOT_DEV;               // ROOT_DEV 定义在 fs/super.c，29 行。
134         SWAP_DEV = ORIG_SWAP_DEV;               // SWAP_DEV 定义在 mm/swap.c，36 行。
135         sprintf(term, "TERM=con%dx%d", CON_COLS, CON_ROWS);
136         envp[1] = term;
137         envp_rc[1] = term;
138         drive_info = DRIVE_INFO;                // 复制内存 0x90080 处的硬盘参数表。

    // 接着根据机器物理内存容量设置高速缓冲区和主内存区的位置和范围。
    // 高速缓存末端地址➔buffer_memory_end；机器内存容量➔memory_end；
    // 主内存开始地址  ➔main_memory_start；
139         memory_end = (1<<20) + (EXT_MEM_K<<10); // 内存大小=1Mb + 扩展内存(k)*1024 字节。
140         memory_end &= 0xfffff000;               // 忽略不到 4Kb（1 页）的内存数。
141         if (memory_end > 16*1024*1024)          // 如果内存量超过 16Mb，则按 16Mb 计。
142                 memory_end = 16*1024*1024;
143         if (memory_end > 12*1024*1024)          // 如果内存>12Mb，则设置缓冲区末端=4Mb
144                 buffer_memory_end = 4*1024*1024;
145         else if (memory_end > 6*1024*1024)      // 否则若内存>6Mb，则设置缓冲区末端=2Mb
146                 buffer_memory_end = 2*1024*1024;
147         else
148                 buffer_memory_end = 1*1024*1024; // 否则则设置缓冲区末端=1Mb
149         main_memory_start = buffer_memory_end;  // 主内存起始位置 = 缓冲区末端。

    // 如果在 Makefile 文件中定义了内存虚拟盘符号 RAMDISK，则初始化虚拟盘。此时主内存将减少。
    //参见 kernel/blk_drv/ramdisk.c。
150 #ifdef RAMDISK
151         main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
152 #endif
    // 以下是内核进行所有方面的初始化工作。阅读时最好跟着调用的程序深入进去看，若实在看
    // 不下去了，就先放一放，继续看下一个初始化调用 -- 这是经验之谈☺。
153         mem_init(main_memory_start,memory_end);  // 主内存区初始化。（mm/memory.c，399）
154         trap_init();            // 陷阱门（硬件中断向量）初始化。（kernel/traps.c，181）
155         blk_dev_init();         // 块设备初始化。      （blk_drv/ll_rw_blk.c，157）
156         chr_dev_init();         // 字符设备初始化。    （chr_drv/tty_io.c，347）
157         tty_init();             // tty 初始化。        （chr_drv/tty_io.c，406）
158         time_init();            // 设置开机启动时间。（见第 92 行）
159         sched_init();           // 调度程序初始化（加载任务 0 的 tr,ldtr）（kernel/sched.c，385）
160         buffer_init(buffer_memory_end); // 缓冲管理初始化，建内存链表等。（fs/buffer.c，348）
161         hd_init();              // 硬盘初始化。        （blk_drv/hd.c，343）
162         floppy_init();          // 软驱初始化。        （blk_drv/floppy.c，457）
163         sti();                  // 所有初始化工作都做完了，于是开启中断。

    // 下面过程通过在堆栈中设置的参数，利用中断返回指令启动任务 0 执行。
164         move_to_user_mode();    // 移到用户模式下执行。（include/asm/system.h，第 1 行）
165         if (!fork()) {          /* we count on this going ok */
166                 init();         // 在新建的子进程（任务 1 即 init 进程）中执行。
167         }
```

```
    // 下面代码开始以任务 0 的身份运行。
168 /*
169  *  NOTE!!  For any other task 'pause()' would mean we have to get a
170  * signal to awaken, but task0 is the sole exception (see 'schedule()')
171  * as task 0 gets activated at every idle moment (when no other tasks
172  * can run). For task0 'pause()' just means we go check if some other
173  * task can run, and if not we return here.
174  */
    /* 注意!! 对于任何其他的任务，'pause()'将意味着我们必须等待收到一个信号
     * 才会返回就绪态，但任务 0（task0）是唯一例外情况（参见'schedule()'），
     * 因为任务 0 在任何空闲时间里都会被激活（当没有其他任务在运行时），因此
     * 对于任务 0'pause()'仅意味着我们返回来查看是否有其他任务可以运行，如果
     * 没有的话我们就回到这里，一直循环执行'pause()'。
     */
    // pause()系统调用（kernel/sched.c,144）会把任务 0 转换成可中断等待状态，再执行调度函数。
    // 但是调度函数只要发现系统中没有其他任务可以运行时就会切换到任务 0，而不依赖于任务 0 的
    // 状态。
175         for(;;)
176                 __asm__("int $0x80"::"a" (__NR_pause):"ax");  // 即执行系统调用 pause()。
177 }
178
    // 下面函数产生格式化信息并输出到标准输出设备 stdout(1)，这里是指屏幕上显示。参数'*fmt'
    // 指定输出将采用的格式，参见标准 C 语言书籍。该子程序正好是 vsprintf 如何使用的一个简单
    // 例子。该程序使用 vsprintf()将格式化的字符串放入 printbuf 缓冲区，然后用 write()将缓冲
    // 区的内容输出到标准设备（1--stdout）。vsprintf()函数的实现见 kernel/vsprintf.c。
179 static int printf(const char *fmt, ...)
180 {
181         va_list args;
182         int i;
183
184         va_start(args, fmt);
185         write(1,printbuf,i=vsprintf(printbuf, fmt, args));
186         va_end(args);
187         return i;
188 }
189


    // 在 main()中已经进行了系统初始化，包括内存管理、各种硬件设备和驱动程序。init()函数
    // 运行在任务 0 第 1 次创建的子进程（任务 1）中。它首先对第一个将要执行的程序（shell）
    // 的环境进行初始化，然后以登录 shell 方式加载该程序并执行之。
190 void init(void)
191 {
192         int pid,i;
193
    // setup() 是一个系统调用。用于读取硬盘参数包括分区表信息并加载虚拟盘（若存在的话）和
    // 安装根文件系统设备。该函数用 25 行上的宏定义，对应函数是 sys_setup()，在块设备子目录
    // kernel/blk_drv/hd.c，74 行。
194         setup((void *) &drive_info);

    // 下面以读写访问方式打开设备“/dev/tty0”，它对应终端控制台。由于这是第一次打开文件
    // 操作，因此产生的文件句柄号（文件描述符）肯定是 0。该句柄是 UNIX 类操作系统默认的控
    // 制台标准输入句柄 stdin。这里再把它以读和写的方式分别打开是为了复制产生标准输出（写）
    // 句柄 stdout 和标准出错输出句柄 stderr。函数前面的“(void)”前缀用于表示强制函数无需
```

```
    // 返回值。
195         (void) open("/dev/tty1",O_RDWR,0);
196         (void) dup(0);              // 复制句柄，产生句柄 1 号--stdout 标准输出设备。
197         (void) dup(0);              // 复制句柄，产生句柄 2 号--stderr 标准出错输出设备。

    // 下面打印缓冲区块数和总字节数，每块 1024 字节，以及主内存区空闲内存字节数。
198         printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS,
199                 NR_BUFFERS*BLOCK_SIZE);
200         printf("Free mem: %d bytes\n\r",memory_end-main_memory_start);

    // 下面 fork()用于创建一个子进程（任务 2）。对于被创建的子进程，fork()将返回 0 值，对于
    // 原进程（父进程）则返回子进程的进程号 pid。所以第 202--206 行是子进程执行的内容。该子
    // 进程关闭了句柄 0（stdin）、以只读方式打开/etc/rc 文件，并使用 execve()函数将进程自身
    // 替换成 /bin/sh 程序（即 shell 程序），然后执行 /bin/sh 程序。所携带的参数和环境变量分
    // 别由 argv_rc 和 envp_rc 数组给出。关闭句柄 0 并立刻打开 /etc/rc 文件的作用是把标准输入
    // stdin 重定向到 /etc/rc 文件。这样 shell 程序/bin/sh 就可以运行 rc 文件中设置的命令。由
    // 于这里 sh 的运行方式是非交互式的，因此在执行完 rc 文件中的命令后就会立刻退出，进程 2
    // 也随之结束。关于 execve()函数说明请参见 fs/exec.c 程序，207 行。
    // 函数_exit()退出时的出错码 1 - 操作未许可；2 -- 文件或目录不存在。
201         if (!(pid=fork())) {
202                 close(0);
203                 if (open("/etc/rc",O_RDONLY,0))
204                         _exit(1);                   // 若打开文件失败，则退出(lib/_exit.c,10)。
205                 execve("/bin/sh",argv_rc,envp_rc);          // 替换成/bin/sh 程序并执行。
206                 _exit(2);                                   // 若 execve()执行失败则退出。
207         }

    // 下面还是父进程（1）执行的语句。wait()等待子进程停止或终止，返回值应是子进程的进程号
    // (pid)。这三句的作用是父进程等待子进程的结束。&i 是存放返回状态信息的位置。如果 wait()
    // 返回值不等于子进程号，则继续等待。
208         if (pid>0)
209                 while (pid != wait(&i))
210                         /* nothing */;         /* 空循环 */

    // 如果执行到这里，说明刚创建的子进程的执行已停止或终止了。下面循环中首先再创建一个子
    // 进程，如果出错，则显示“初始化程序创建子进程失败”信息并继续执行。对于所创建的子进
    // 程将关闭所有以前还遗留的句柄(stdin, stdout, stderr)，新创建一个会话并设置进程组号，
    // 然后重新打开 /dev/tty0 作为 stdin，并复制成 stdout 和 stderr。再次执行系统解释程序
    // /bin/sh。但这次执行所选用的参数和环境数组另选了一套（见上面 122--123 行）。然后父进
    // 程再次运行 wait()等待。如果子进程又停止了执行，则在标准输出上显示出错信息“子进程
    // pid 停止了运行，返回码是 i”，然后继续重试下去…，形成“大”死循环。
211         while (1) {
212                 if ((pid=fork())<0) {
213                         printf("Fork failed in init\r\n");
214                         continue;
215                 }
216                 if (!pid) {                             // 新的子进程。
217                         close(0);close(1);close(2);
218                         setsid();                       // 创建一新的会话期，见后面说明。
219                         (void) open("/dev/tty1",O_RDWR,0);
220                         (void) dup(0);
221                         (void) dup(0);
222                         _exit(execve("/bin/sh",argv,envp));
```

```
223                 }
224                 while (1)
225                         if (pid == wait(&i))
226                                 break;
227                 printf("\n\rchild %d died with code %04x\n\r",pid,i);
228                 sync();                                 // 同步操作，刷新缓冲区。
229         }
230         _exit(0);       /* NOTE! _exit, not exit() */ /*注意！是_exit()，非 exit()*/
    // _exit()和 exit()都用于正常终止一个函数。但_exit()直接是一个 sys_exit 系统调用，而
    // exit()则通常是普通函数库中的一个函数。它会先执行一些清除操作，例如调用执行各终止
    // 处理程序、关闭所有标准 IO 等，然后调用 sys_exit。
231 }
232
```

第8章 内核核心程序

8.1 程序 8-1 linux/kernel/asm.s

```
 1 /*
 2  *  linux/kernel/asm.s
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * asm.s contains the low-level code for most hardware faults.
 9  * page_exception is handled by the mm, so that isn't here. This
10  * file also handles (hopefully) fpu-exceptions due to TS-bit, as
11  * the fpu must be properly saved/resored. This hasn't been tested.
12  */
   /*
    * asm.s 程序中包括大部分的硬件故障（或出错）处理的底层次代码。页异常由内存管理程序
    * mm 处理，所以不在这里。此程序还处理（希望是这样）由于 TS-位而造成的 fpu 异常，因为
    * fpu 必须正确地进行保存/恢复处理，这些还没有测试过。
    */
13
   # 本代码文件主要涉及对 Intel 保留中断 int0--int16 的处理（int17-int31 留作今后使用）。
   # 以下是一些全局函数名的声明，其原形在 traps.c 中说明。
14 .globl _divide_error,_debug,_nmi,_int3,_overflow,_bounds,_invalid_op
15 .globl _double_fault,_coprocessor_segment_overrun
16 .globl _invalid_TSS,_segment_not_present,_stack_segment
17 .globl _general_protection,_coprocessor_error,_irq13,_reserved
18 .globl _alignment_check
19
   # 下面这段程序处理无出错号的情况。
   # int0 -- 处理被零除出错的情况。  类型：错误；出错号：无。
   # 在执行 DIV 或 IDIV 指令时，若除数是 0，CPU 就会产生这个异常。当 EAX（或 AX、AL）容纳
   # 不了一个合法除操作的结果时，也会产生这个异常。21 行标号'_do_divide_error'实际上是
   # C 语言函数 do_divide_error()编译后所生成模块中对应的名称。函数'do_divide_error'在
   # traps.c 中实现（第 101 行开始）。
20 _divide_error:
21         pushl $_do_divide_error     # 首先把将要调用的函数地址入栈。
22 no_error_code:                      # 这里是无出错号处理的入口处，见下面第 56 行等。
23         xchgl %eax,(%esp)           # _do_divide_error 的地址 ➔ eax，eax 被交换入栈。
24         pushl %ebx
25         pushl %ecx
26         pushl %edx
27         pushl %edi
28         pushl %esi
29         pushl %ebp
30         push %ds                    # ！！16 位的段寄存器入栈后也要占用 4 个字节。
31         push %es
32         push %fs
33         pushl $0                    # "error code"   # 将数值 0 作为出错码入栈。
```

```
34          lea 44(%esp),%edx           # 取有效地址，即栈中原调用返回地址处的栈指针位置，
35          pushl %edx                  # 并压入堆栈。
36          movl $0x10,%edx             # 初始化段寄存器 ds、es 和 fs，加载内核数据段选择符。
37          mov %dx,%ds
38          mov %dx,%es
39          mov %dx,%fs
    # 下行上的'*'号表示调用操作数指定地址处的函数，称为间接调用。这句的含义是调用引起本次
    # 异常的 C 处理函数，例如 do_divide_error()等。第 41 行是将堆栈指针加 8 相当于执行两次 pop
    # 操作，弹出（丢弃）最后入堆栈的两个 C 函数参数（33 行和 35 行入栈的值），让堆栈指针重新
    # 指向寄存器 fs 入栈处。
40          call *%eax
41          addl $8,%esp
42          pop %fs
43          pop %es
44          pop %ds
45          popl %ebp
46          popl %esi
47          popl %edi
48          popl %edx
49          popl %ecx
50          popl %ebx
51          popl %eax                   # 弹出原来 eax 中的内容。
52          iret
53
    # int1 -- debug 调试中断入口点。处理过程同上。类型：错误/陷阱（Fault/Trap）；无错误号。
    # 当 eflags 中 TF 标志置位时而引发的中断。当发现硬件断点（数据：陷阱，代码：错误）；或者
    # 开启了指令跟踪陷阱或任务交换陷阱，或者调试寄存器访问无效（错误），CPU 就会产生该异常。
54  _debug:
55          pushl $_do_int3         # _do_debug  # C 函数指针入栈。以下同。
56          jmp no_error_code
57
    # int2 -- 非屏蔽中断调用入口点。   类型：陷阱；无错误号。
    # 这是仅有的被赋予固定中断向量的硬件中断。每当接收到一个 NMI 信号，CPU 内部就会产生中断
    # 向量 2，并执行标准中断应答周期，因此很节省时间。NMI 通常保留为极为重要的硬件事件使用。
    # 当 CPU 收到一个 NMI 信号并且开始执行其中断处理过程时，随后所有的硬件中断都将被忽略。
58  _nmi:
59          pushl $_do_nmi
60          jmp no_error_code
61
    # int3 -- 断点指令引起中断的入口点。   类型：陷阱；无错误号。
    # 由 int 3 指令引发的中断，与硬件中断无关。该指令通常由调式器插入被调式程序的代码中。
    # 处理过程同_debug。
62  _int3:
63          pushl $_do_int3
64          jmp no_error_code
65
    # int4 -- 溢出出错处理中断入口点。   类型：陷阱；无错误号。
    # EFLAGS 中 OF 标志置位时 CPU 执行 INTO 指令就会引发该中断。通常用于编译器跟踪算术计算溢出。
66  _overflow:
67          pushl $_do_overflow
68          jmp no_error_code
69
    # int5 -- 边界检查出错中断入口点。   类型：错误；无错误号。
```

```
    # 当操作数在有效范围以外时引发的中断。当 BOUND 指令测试失败就会产生该中断。BOUND 指令有
    # 3 个操作数，如果第 1 个不在另外两个之间，就产生异常 5。
70  _bounds:
71          pushl $_do_bounds
72          jmp no_error_code
73
    # int6 -- 无效操作指令出错中断入口点。   类型：错误；无错误号。
    # CPU 执行机构检测到一个无效的操作码而引起的中断。
74  _invalid_op:
75          pushl $_do_invalid_op
76          jmp no_error_code
77
    # int9 -- 协处理器段超出出错中断入口点。   类型：放弃；无错误号。
    # 该异常基本上等同于协处理器出错保护。因为在浮点指令操作数太大时，我们就有这个机会来
    # 加载或保存超出数据段的浮点值。
78  _coprocessor_segment_overrun:
79          pushl $_do_coprocessor_segment_overrun
80          jmp no_error_code
81
    # int15 - 其他 Intel 保留中断的入口点。
82  _reserved:
83          pushl $_do_reserved
84          jmp no_error_code
85
    # int45 -- (0x20 + 13) Linux 设置的数学协处理器硬件中断。
    # 当协处理器执行完一个操作时就会发出 IRQ13 中断信号，以通知 CPU 操作完成。80387 在执行
    # 计算时，CPU 会等待其操作完成。下面 89 行上 0xF0 是协处理端口，用于清忙锁存器。通过写
    # 该端口，本中断将消除 CPU 的 BUSY 延续信号，并重新激活 80387 的处理器扩展请求引脚 PEREQ。
    # 该操作主要是为了确保在继续执行 80387 的任何指令之前，CPU 响应本中断。
86  _irq13:
87          pushl %eax
88          xorb %al,%al
89          outb %al,$0xF0
90          movb $0x20,%al
91          outb %al,$0x20              # 向 8259 主中断控制芯片发送 EOI（中断结束）信号。
92          jmp 1f                      # 这两个跳转指令起延时作用。
93  1:      jmp 1f
94  1:      outb %al,$0xA0              # 再向 8259 从中断控制芯片发送 EOI（中断结束）信号。
95          popl %eax
96          jmp _coprocessor_error      # 该函数原在本程序中，现已放到 system_call.s 中。
97
    # 以下中断在调用时 CPU 会在中断返回地址之后将出错号压入堆栈，因此返回时也需要将出错号
    # 弹出（参见图 5.3(b)）。

    # int8 -- 双出错故障。   类型：放弃；有错误码。
    # 通常当 CPU 在调用前一个异常的处理程序而又检测到一个新的异常时，这两个异常会被串行地进行
    # 处理，但也会碰到很少的情况，CPU 不能进行这样的串行处理操作，此时就会引发该中断。
98  _double_fault:
99          pushl $_do_double_fault     # C 函数地址入栈。
100 error_code:
101         xchgl %eax,4(%esp)          # error code <-> %eax，eax 原来的值被保存在堆栈上。
102         xchgl %ebx,(%esp)           # &function <-> %ebx，ebx 原来的值被保存在堆栈上。
103         pushl %ecx
```

```
104         pushl %edx
105         pushl %edi
106         pushl %esi
107         pushl %ebp
108         push %ds
109         push %es
110         push %fs
111         pushl %eax                  # error code    # 出错号入栈。
112         lea 44(%esp),%eax           # offset        # 程序返回地址处堆栈指针位置值入栈。
113         pushl %eax
114         movl $0x10,%eax             # 置内核数据段选择符。
115         mov %ax,%ds
116         mov %ax,%es
117         mov %ax,%fs
118         call *%ebx                  # 间接调用，调用相应的 C 函数，其参数已入栈。
119         addl $8,%esp                # 丢弃入栈的 2 个用作 C 函数的参数。
120         pop %fs
121         pop %es
122         pop %ds
123         popl %ebp
124         popl %esi
125         popl %edi
126         popl %edx
127         popl %ecx
128         popl %ebx
129         popl %eax
130         iret
131
    # int10 -- 无效的任务状态段(TSS)。   类型：错误；有出错码。
    # CPU 企图切换到一个进程，而该进程的 TSS 无效。根据 TSS 中哪一部分引起了异常，当由于 TSS
    # 长度超过 104 字节时，这个异常在当前任务中产生，因而切换被终止。其他问题则会导致在切换
    # 后的新任务中产生本异常。
132 _invalid_TSS:
133         pushl $_do_invalid_TSS
134         jmp error_code
135
    # int11 -- 段不存在。   类型：错误；有出错码。
    # 被引用的段不在内存中。段描述符中标志指明段不在内存中。
136 _segment_not_present:
137         pushl $_do_segment_not_present
138         jmp error_code
139
    # int12 -- 堆栈段错误。   类型：错误；有出错码。
    # 指令操作试图超出堆栈段范围，或者堆栈段不在内存中。这是异常 11 和 13 的特例。有些操作
    # 系统可以利用这个异常来确定什么时候应该为程序分配更多的栈空间。
140 _stack_segment:
141         pushl $_do_stack_segment
142         jmp error_code
143
    # int13 -- 一般保护性出错。   类型：错误；有出错码。
    # 表明是不属于任何其他类的错误。若一个异常产生时没有对应的处理向量（0--16），通常就
    # 会归到此类。
144 _general_protection:
```

```
145         pushl $_do_general_protection
146         jmp error_code
147
    # int17 -- 边界对齐检查出错。
    # 在启用了内存边界检查时，若特权级 3（用户级）数据非边界对齐时会产生该异常。
148 _alignment_check:
149         pushl $_do_alignment_check
150         jmp error_code
151
    # int7 -- 设备不存在（_device_not_available）在 kernel/sys_call.s，158 行。
    # int14 -- 页错误（_page_fault）在 mm/page.s，14 行。
    # int16 -- 协处理器错误（_coprocessor_error）在 kernel/sys_call.s，140 行。
    # 时钟中断 int 0x20（_timer_interrupt）在 kernel/sys_call.s，189 行。
    # 系统调用 int 0x80（_system_call）在 kernel/sys_call.s，84 行。
```

8.2 程序 8-2 linux/kernel/traps.c

```
 1 /*
 2  *  linux/kernel/traps.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * 'Traps.c' handles hardware traps and faults after we have saved some
 9  * state in 'asm.s'. Currently mostly a debugging-aid, will be extended
10  * to mainly kill the offending process (probably by giving it a signal,
11  * but possibly by killing it outright if necessary).
12  */
    /*
     * 在程序 asm.s 中保存了一些状态后，本程序用来处理硬件陷阱和故障。目前主要用于调试目的，
     * 以后将扩展用来杀死遭损坏的进程（主要是通过发送一个信号，但如果必要也会直接杀死）。
     */
13 #include <string.h>        // 字符串头文件。主要定义了一些有关内存或字符串操作的嵌入函数。
14
15 #include <linux/head.h>    // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
16 #include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                              // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
17 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
18 #include <asm/system.h>    // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
19 #include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
20 #include <asm/io.h>        // 输入/输出头文件。定义硬件端口输入/输出宏汇编语句。
21
    // 以下语句定义了三个嵌入式汇编宏语句函数。有关嵌入式汇编的基本语法见本程序列表后的说明。
    // 用圆括号括住的组合语句（花括号中的语句）可以作为表达式使用，其中最后的__res 是其输出值。
    // 第 23 行定义了一个寄存器变量__res。该变量将被保存在一个寄存器中，以便于快速访问和操作。
    // 如果想指定寄存器（例如 eax），那么我们可以把该句写成“register char __res asm("ax");”。
    // 取段 seg 中地址 addr 处的一个字节。
    // 参数：seg - 段选择符；addr - 段内指定地址。
    // 输出：%0 - eax (__res)；输入：%1 - eax (seg)；%2 - 内存地址 (*(addr))。
22 #define get_seg_byte(seg,addr) ({ \
23 register char __res; \
24 __asm__("push %%fs;mov %%ax,%%fs;movb %%fs:%2,%%al;pop %%fs" \
25         :"=a" (__res):"0" (seg),"m" (*(addr))); \
26 __res;})
27
    // 取段 seg 中地址 addr 处的一个长字（4 字节）。
    // 参数：seg - 段选择符；addr - 段内指定地址。
    // 输出：%0 - eax (__res)；输入：%1 - eax (seg)；%2 - 内存地址 (*(addr))。
28 #define get_seg_long(seg,addr) ({ \
29 register unsigned long __res; \
30 __asm__("push %%fs;mov %%ax,%%fs;movl %%fs:%2,%%eax;pop %%fs" \
31         :"=a" (__res):"0" (seg),"m" (*(addr))); \
32 __res;})
33
    // 取 fs 段寄存器的值（选择符）。
```

```
    // 输出：%0 - eax (__res)。
34  #define _fs() ({ \
35  register unsigned short __res; \
36  __asm__("mov %%fs,%%ax":"=a" (__res):); \
37  __res;})
38
    // 以下定义了一些函数原型。
39  void page_exception(void);                  // 页异常。实际是 page_fault（mm/page.s，14）。
40
41  void divide_error(void);                    // int0（kernel/asm.s，20）。
42  void debug(void);                           // int1（kernel/asm.s，54）。
43  void nmi(void);                             // int2（kernel/asm.s，58）。
44  void int3(void);                            // int3（kernel/asm.s，62）。
45  void overflow(void);                        // int4（kernel/asm.s，66）。
46  void bounds(void);                          // int5（kernel/asm.s，70）。
47  void invalid_op(void);                      // int6（kernel/asm.s，74）。
48  void device_not_available(void);            // int7（kernel/sys_call.s，158）。
49  void double_fault(void);                    // int8（kernel/asm.s，98）。
50  void coprocessor_segment_overrun(void);     // int9（kernel/asm.s，78）。
51  void invalid_TSS(void);                     // int10（kernel/asm.s，132）。
52  void segment_not_present(void);             // int11（kernel/asm.s，136）。
53  void stack_segment(void);                   // int12（kernel/asm.s，140）。
54  void general_protection(void);              // int13（kernel/asm.s，144）。
55  void page_fault(void);                      // int14（mm/page.s，14）。
56  void coprocessor_error(void);               // int16（kernel/sys_call.s，140）。
57  void reserved(void);                        // int15（kernel/asm.s，82）。
58  void parallel_interrupt(void);              // int39（kernel/sys_call.s，295）。
59  void irq13(void);                           // int45 协处理器中断处理（kernel/asm.s，86）。
60  void alignment_check(void);                 // int46（kernel/asm.s，148）。
61
    // 该子程序用来打印出错中断的名称、出错号、调用程序的 EIP、EFLAGS、ESP、fs 段寄存器值、
    // 段的基址、段的长度、进程号 pid、任务号、10 字节指令码。如果堆栈在用户数据段，则还
    // 打印 16 字节的堆栈内容。这些信息可用于程序调试。
62  static void die(char * str,long esp_ptr,long nr)
63  {
64          long * esp = (long *) esp_ptr;
65          int i;
66
67          printk("%s: %04x\n\r",str,nr&0xffff);
    // 下行打印语句显示当前调用进程的 CS:EIP、EFLAGS 和 SS:ESP 的值。参照错误!未找到引用源。可知，这
里 esp[0]
    // 即为图中的 esp0 位置。因此我们把这句拆分开来看为：
    // (1) EIP:\t%04x:%p\n  -- esp[1]是段选择符（cs），esp[0]是 eip
    // (2) EFLAGS:\t%p      -- esp[2]是 eflags
    // (3) ESP:\t%04x:%p\n  -- esp[4]是原 ss，esp[3]是原 esp
68          printk("EIP:\t%04x:%p\nEFLAGS:\t%p\nESP:\t%04x:%p\n",
69                  esp[1],esp[0],esp[2],esp[4],esp[3]);
70          printk("fs: %04x\n",_fs());
71          printk("base: %p, limit: %p\n",get_base(current->ldt[1]),get_limit(0x17));
72          if (esp[4] == 0x17) {               // 若原 ss 值为 0x17（用户栈），则还打印出
73                  printk("Stack: ");          // 用户栈中的 4 个长字值（16 字节）。
74                  for (i=0;i<4;i++)
75                          printk("%p ",get_seg_long(0x17,i+(long *)esp[3]));
```

```
76                  printk("\n");
77          }
78          str(i);                 // 取当前运行任务的任务号（include/linux/sched.h，210 行）。
79          printk("Pid: %d, process nr: %d\n\r",current->pid,0xffff & i); // 进程号，任务号。
80          for(i=0;i<10;i++)
81                  printk("%02x ",0xff & get_seg_byte(esp[1],(i+(char *)esp[0])));
82          printk("\n\r");
83          do_exit(11);            /* play segment exception */
84  }
85
    // 以下这些以 do_开头的函数是 asm.s 中对应中断处理程序调用的 C 函数。
86  void do_double_fault(long esp, long error_code)
87  {
88          die("double fault",esp,error_code);
89  }
90
91  void do_general_protection(long esp, long error_code)
92  {
93          die("general protection",esp,error_code);
94  }
95
96  void do_alignment_check(long esp, long error_code)
97  {
98      die("alignment check",esp,error_code);
99  }
100
101 void do_divide_error(long esp, long error_code)
102 {
103         die("divide error",esp,error_code);
104 }
105
    // 参数是进入中断后被顺序压入堆栈的寄存器值。参见 asm.s 程序第 24--35 行。
106 void do_int3(long * esp, long error_code,
107                 long fs,long es,long ds,
108                 long ebp,long esi,long edi,
109                 long edx,long ecx,long ebx,long eax)
110 {
111         int tr;
112
113         __asm__("str %%ax":"=a" (tr):"" (0));                   // 取任务寄存器值➔tr。
114         printk("eax\t\tebx\t\tecx\t\tedx\n\r%8x\t%8x\t%8x\t%8x\n\r",
115                 eax,ebx,ecx,edx);
116         printk("esi\t\tedi\t\tebp\t\tesp\n\r%8x\t%8x\t%8x\t%8x\n\r",
117                 esi,edi,ebp,(long) esp);
118         printk("\n\rds\tes\tfs\ttr\n\r%4x\t%4x\t%4x\t%4x\n\r",
119                 ds,es,fs,tr);
120         printk("EIP: %8x   CS: %4x  EFLAGS: %8x\n\r",esp[0],esp[1],esp[2]);
121 }
122
123 void do_nmi(long esp, long error_code)
124 {
125         die("nmi",esp,error_code);
126 }
```

```
127
128 void do_debug(long esp, long error_code)
129 {
130         die("debug",esp,error_code);
131 }
132
133 void do_overflow(long esp, long error_code)
134 {
135         die("overflow",esp,error_code);
136 }
137
138 void do_bounds(long esp, long error_code)
139 {
140         die("bounds",esp,error_code);
141 }
142
143 void do_invalid_op(long esp, long error_code)
144 {
145         die("invalid operand",esp,error_code);
146 }
147
148 void do_device_not_available(long esp, long error_code)
149 {
150         die("device not available",esp,error_code);
151 }
152
153 void do_coprocessor_segment_overrun(long esp, long error_code)
154 {
155         die("coprocessor segment overrun",esp,error_code);
156 }
157
158 void do_invalid_TSS(long esp,long error_code)
159 {
160         die("invalid TSS",esp,error_code);
161 }
162
163 void do_segment_not_present(long esp,long error_code)
164 {
165         die("segment not present",esp,error_code);
166 }
167
168 void do_stack_segment(long esp,long error_code)
169 {
170         die("stack segment",esp,error_code);
171 }
172
173 void do_coprocessor_error(long esp, long error_code)
174 {
175         if (last_task_used_math != current)
176                 return;
177         die("coprocessor error",esp,error_code);
178 }
179
```

```
180 void do_reserved(long esp, long error_code)
181 {
182         die("reserved (15,17-47) error",esp,error_code);
183 }
184
    // 下面是异常（陷阱）中断程序初始化子程序。设置它们的中断调用门（中断向量）。
    // set_trap_gate()与 set_system_gate()都使用了中断描述符表 IDT 中的陷阱门（Trap Gate），
    // 它们之间的主要区别在于前者设置的特权级为 0，后者是 3。因此断点陷阱中断 int3、溢出中断
    // overflow 和边界出错中断 bounds 可以由任何程序调用。 这两个函数均是嵌入式汇编宏程序，
    // 参见 include/asm/system.h，第 36 行、39 行。
185 void trap_init(void)
186 {
187         int i;
188
189         set_trap_gate(0,&divide_error);      // 设置除操作出错的中断向量值。以下雷同。
190         set_trap_gate(1,&debug);
191         set_trap_gate(2,&nmi);
192         set_system_gate(3,&int3);            /* int3-5 can be called from all */
193         set_system_gate(4,&overflow);        /* int3-5 可以被所有程序执行 */
194         set_system_gate(5,&bounds);
195         set_trap_gate(6,&invalid_op);
196         set_trap_gate(7,&device_not_available);
197         set_trap_gate(8,&double_fault);
198         set_trap_gate(9,&coprocessor_segment_overrun);
199         set_trap_gate(10,&invalid_TSS);
200         set_trap_gate(11,&segment_not_present);
201         set_trap_gate(12,&stack_segment);
202         set_trap_gate(13,&general_protection);
203         set_trap_gate(14,&page_fault);
204         set_trap_gate(15,&reserved);
205         set_trap_gate(16,&coprocessor_error);
206         set_trap_gate(17,&alignment_check);

    // 下面把 int17-47 的陷阱门先均设置为 reserved，以后各硬件初始化时会重新设置自己的陷阱门。
207         for (i=18;i<48;i++)
208                 set_trap_gate(i,&reserved);

    // 设置协处理器中断 0x2d（45）陷阱门描述符，并允许其产生中断请求。设置并行口中断描述符。
209         set_trap_gate(45,&irq13);
210         outb_p(inb_p(0x21)&0xfb,0x21);          // 允许 8259A 主芯片的 IRQ2 中断请求。
211         outb(inb_p(0xA1)&0xdf,0xA1);            // 允许 8259A 从芯片的 IRQ13 中断请求。
212         set_trap_gate(39,&parallel_interrupt);  // 设置并行口 1 的中断 0x27 陷阱门描述符。
213 }
214
```

8.3 程序 8-3 linux/kernel/sys_call.s

```
 1 /*
 2  *  linux/kernel/system_call.s
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  *  system_call.s  contains the system-call low-level handling routines.
 9  * This also contains the timer-interrupt handler, as some of the code is
10  * the same. The hd- and flopppy-interrupts are also here.
11  *
12  * NOTE: This code handles signal-recognition, which happens every time
13  * after a timer-interrupt and after each system call. Ordinary interrupts
14  * don't handle signal-recognition, as that would clutter them up totally
15  * unnecessarily.
16  *
17  * Stack layout in 'ret_from_system_call':
18  *
19  *       0(%esp) - %eax
20  *       4(%esp) - %ebx
21  *       8(%esp) - %ecx
22  *       C(%esp) - %edx
23  *      10(%esp) - original %eax        (-1 if not system call)
24  *      14(%esp) - %fs
25  *      18(%esp) - %es
26  *      1C(%esp) - %ds
27  *      20(%esp) - %eip
28  *      24(%esp) - %cs
29  *      28(%esp) - %eflags
30  *      2C(%esp) - %oldesp
31  *      30(%esp) - %oldss
32  */
   /*
    * system_call.s 文件包含系统调用（system-call）底层处理子程序。由于有些代码比较类似，
    * 所以同时也包括时钟中断处理（timer-interrupt）句柄。硬盘和软盘的中断处理程序也在这里。
    *
    * 注意：这段代码处理信号（signal）识别，在每次时钟中断和系统调用之后都会进行识别。一般
    * 中断过程并不处理信号识别，因为会给系统造成混乱。
    *
    * 从系统调用返回（'ret_from_system_call'）时堆栈的内容见上面 19-30 行。
    */
   # 上面 Linus 原注释中一般中断过程是指除了系统调用中断（int 0x80）和时钟中断（int 0x20）
   # 以外的其他中断。这些中断会在内核态或用户态随机发生，若在这些中断过程中也处理信号识别
   # 的话，就有可能与系统调用中断和时钟中断过程中对信号的识别处理过程相冲突，，违反了内核
   # 代码非抢占原则。因此系统既无必要在这些“其他”中断中处理信号，也不允许这样做。
33
34 SIG_CHLD        = 17                # 定义 SIG_CHLD 信号（子进程停止或结束）。
35
36 EAX             = 0x00              # 堆栈中各个寄存器的偏移位置。
```

```
37 EBX             = 0x04
38 ECX             = 0x08
39 EDX             = 0x0C
40 ORIG_EAX        = 0x10          # 如果不是系统调用（是其它中断）时，该值为-1。
41 FS              = 0x14
42 ES              = 0x18
43 DS              = 0x1C
44 EIP             = 0x20          # 44 -- 48 行 由 CPU 自动入栈。
45 CS              = 0x24
46 EFLAGS          = 0x28
47 OLDESP          = 0x2C          # 当特权级变化时，原堆栈指针也会入栈。
48 OLDSS           = 0x30
49
   # 以下这些是任务结构（task_struct）中变量的偏移值，参见 include/linux/sched.h，105 行开始。
50 state   = 0            # these are offsets into the task-struct.  # 进程状态码。
51 counter = 4            # 任务运行时间计数(递减)（滴答数），运行时间片。
52 priority = 8          # 运行优先数。任务开始运行时 counter=priority，越大则运行时间越长。
53 signal  = 12           # 是信号位图，每个比特位代表一种信号，信号值=位偏移值+1。
54 sigaction = 16        # MUST be 16 (=len of sigaction) # sigaction 结构长度必须是 16 字节。
55 blocked = (33*16)      # 受阻塞信号位图的偏移量。
56
   # 以下定义在 sigaction 结构中的偏移量，参见 include/signal.h，第 55 行开始。
57 # offsets within sigaction
58 sa_handler = 0                  # 信号处理过程的句柄（描述符）。
59 sa_mask = 4                     # 信号屏蔽码。
60 sa_flags = 8                    # 信号集。
61 sa_restorer = 12                # 恢复函数指针，参见 kernel/signal.c 程序说明。
62
63 nr_system_calls = 82            # Linux 0.12 版内核中的系统调用总数。
64
65 ENOSYS = 38                     # 系统调用号出错码。
66
67 /*
68  * Ok, I get parallel printer interrupts while using the floppy for some
69  * strange reason. Urgel. Now I just ignore them.
70  */
   /* 好了，在使用软驱时我收到了并行打印机中断，很奇怪。呵，现在不管它。
    */
71 .globl _system_call,_sys_fork,_timer_interrupt,_sys_execve
72 .globl _hd_interrupt,_floppy_interrupt,_parallel_interrupt
73 .globl _device_not_available, _coprocessor_error
74
   # 系统调用号错误时将返回出错码-ENOSYS。
75 .align 2                        # 内存 4 字节对齐。
76 bad_sys_call:
77         pushl $-ENOSYS          # eax 中置-ENOSYS。
78         jmp ret_from_sys_call

   # 重新执行调度程序入口。调度程序 schedule()在（kernel/sched.c，119 行处开始。
   # 当调度程序 schedule()返回时就从 ret_from_sys_call 处（107 行）继续执行。
79 .align 2
80 reschedule:
81         pushl $ret_from_sys_call   # 将 ret_from_sys_call 的地址入栈（107 行）。
```

```
82          jmp _schedule

    #### int 0x80 --linux 系统调用入口点（调用中断 int 0x80，eax 中是调用号）。
83  .align 2
84  _system_call:
85          push %ds                    # 保存原段寄存器值。
86          push %es
87          push %fs
88          pushl %eax                  # save the orig_eax    # 保存 eax 原值。

    # 一个系统调用最多可带有 3 个参数，也可以不带参数。下面入栈的 ebx、ecx 和 edx 中放着系统
    # 调用相应 C 语言函数（见第 99 行）的调用参数。这几个寄存器入栈的顺序是由 GNU gcc 规定的，
    # ebx 中可存放第 1 个参数，ecx 中存放第 2 个参数，edx 中存放第 3 个参数。
    # 系统调用语句可参见头文件 include/unistd.h 中第 150 到 200 行的系统调用宏。
89          pushl %edx
90          pushl %ecx                  # push %ebx,%ecx,%edx as parameters
91          pushl %ebx                  # to the system call

    # 在保存过段寄存器之后，让 ds,es 指向内核数据段，而 fs 指向当前局部数据段，即指向执行本
    # 次系统调用的用户程序的数据段。注意，在 Linux 0.12 中内核给任务分配的代码和数据内存段
    # 是重叠的，它们的段基址和段限长相同。参见 fork.c 程序中 copy_mem()函数。
92          movl $0x10,%edx             # set up ds,es to kernel space
93          mov %dx,%ds
94          mov %dx,%es
95          movl $0x17,%edx             # fs points to local data space
96          mov %dx,%fs

97          cmpl _NR_syscalls,%eax      # 调用号如果超出范围的话就跳转。
98          jae bad_sys_call

    # 下面这句操作数的含义是：调用地址=[_sys_call_table + %eax * 4]。参见程序后的说明。
    # sys_call_table[]是一个指针数组，定义在 include/linux/sys.h 中，该数组中设置了内核
    # 所有 82 个系统调用 C 处理函数的地址。
99          call _sys_call_table(,%eax,4)    # 间接调用指定功能 C 函数。
100         pushl %eax                       # 把系统调用返回值入栈。

    # 下面 101-106 行查看当前任务的运行状态。如果不在就绪状态（state 不等于 0）就去执行调度
    # 程序。如果该任务在就绪状态，但是其时间片已经用完（counter=0），则也去执行调度程序。
    # 例如当后台进程组中的进程执行控制终端读写操作时，那么默认条件下该后台进程组所有进程
    # 会收到 SIGTTIN 或 SIGTTOU 信号，导致进程组中所有进程处于停止状态。而当前进程则会立刻
    # 返回。
101 2:
102         movl _current,%eax          # 取当前任务（进程）数据结构指针➔eax。
103         cmpl $0,state(%eax)         # state
104         jne reschedule
105         cmpl $0,counter(%eax)       # counter
106         je reschedule

    # 以下这段代码执行从系统调用 C 函数返回后，对信号进行识别处理。其他中断服务程序退出时也
    # 将跳转到这里进行处理后才退出中断过程，例如后面 131 行上的处理器出错中断 int 16。
    # 首先判别当前任务是否是初始任务 task0，如果是则不必对其进行信号量方面的处理，直接返回。
    # 109 行上的_task 对应 C 程序中的 task[]数组，直接引用 task 相当于引用 task[0]。
107 ret_from_sys_call:
```

```
108         movl _current,%eax
109         cmpl _task,%eax             # task[0] cannot have signals
110         je 3f                       # 向前(forward)跳转到标号 3 处退出中断处理。

    # 通过对原调用程序代码选择符的检查来判断调用程序是否是用户任务。如果不是则直接退出中断。
    # 这是因为任务在内核态执行时不可抢占。否则对任务进行信号量的识别处理。这里比较选择符是
    # 否为用户代码段的选择符 0x000f（RPL=3，局部表，代码段）来判断是否为用户任务。如果不是
    # 则说明是某个中断服务程序（例如中断 16）跳转到第 107 行执行到此，于是跳转退出中断程序。
    # 另外，如果原堆栈段选择符不为 0x17（即原堆栈不在用户段中），也说明本次系统调用的调用者
    # 不是用户任务，则也退出。
111         cmpw $0x0f,CS(%esp)         # was old code segment supervisor ?
112         jne 3f
113         cmpw $0x17,OLDSS(%esp)      # was stack segment = 0x17 ?
114         jne 3f

    # 下面这段代码（115-128）用于处理当前任务中的信号。首先取当前任务结构中的信号位图（32 位，
    # 每位代表 1 种信号），然后用任务结构中的信号阻塞（屏蔽）码，阻塞不允许的信号位，取得数值
    # 最小的信号值，再把原信号位图中该信号对应的位复位（置 0），最后将该信号值作为参数之一调
    # 用 do_signal()。do_signal()在（kernel/signal.c,128）中，其参数包括 13 个入栈的信息。
    # 在 do_signal()或信号处理函数返回之后，若返回值不为 0 则再看看是否需要切换进程或继续处理
    # 其它信号。
115         movl signal(%eax),%ebx      # 取信号位图➔ebx，每 1 位代表 1 种信号，共 32 个信号。
116         movl blocked(%eax),%ecx     # 取阻塞（屏蔽）信号位图➔ecx。
117         notl %ecx                   # 每位取反。
118         andl %ebx,%ecx              # 获得许可的信号位图。
119         bsfl %ecx,%ecx              # 从低位（位 0）开始扫描位图，看是否有 1 的位，
                                        # 若有，则 ecx 保留该位的偏移值（即第几位 0--31）。
120         je 3f                       # 如果没有信号则向前跳转退出。
121         btrl %ecx,%ebx              # 复位该信号（ebx 含有原 signal 位图）。
122         movl %ebx,signal(%eax)      # 重新保存 signal 位图信息➔current->signal。
123         incl %ecx                   # 将信号调整为从 1 开始的数（1--32）。
124         pushl %ecx                  # 信号值入栈作为调用 do_signal 的参数之一。
125         call _do_signal             # 调用 C 函数信号处理程序（kernel/signal.c，128）。
126         popl %ecx                   # 弹出入栈的信号值。
127         testl %eax, %eax            # 测试返回值，若不为 0 则跳转到前面标号 2（101 行）处。
128         jne 2b                      # see if we need to switch tasks, or do more signals

129 3:      popl %eax                   # eax 中含有第 100 行入栈的系统调用返回值。
130         popl %ebx
131         popl %ecx
132         popl %edx
133         addl $4, %esp               # skip orig_eax     # 跳过（丢弃）原 eax 值。
134         pop %fs
135         pop %es
136         pop %ds
137         iret
138
    #### int16 -- 处理器错误中断。  类型：错误；无错误码。
    # 这是一个外部的基于硬件的异常。当协处理器检测到自己发生错误时，就会通过 ERROR 引脚
    # 通知 CPU。下面代码用于处理协处理器发出的出错信号。并跳转去执行 C 函数 math_error()
    # （kernel/math/error.c 11）。返回后将跳转到标号 ret_from_sys_call 处继续执行。
139 .align 2
140 _coprocessor_error:
```

```
141         push %ds
142         push %es
143         push %fs
144         pushl $-1                   # fill in -1 for orig_eax  # 填-1，表明不是系统调用。
145         pushl %edx
146         pushl %ecx
147         pushl %ebx
148         pushl %eax
149         movl $0x10,%eax             # ds,es 置为指向内核数据段。
150         mov %ax,%ds
151         mov %ax,%es
152         movl $0x17,%eax             # fs 置为指向局部数据段（出错程序的数据段）。
153         mov %ax,%fs
154         pushl $ret_from_sys_call    # 把下面调用返回的地址入栈。
155         jmp _math_error             # 执行 math_error()（kernel/math/error.c，11）。
156
    #### int7 -- 设备不存在或协处理器不存在。  类型：错误；无错误码。
    # 如果控制寄存器 CR0 中 EM（模拟）标志置位，则当 CPU 执行一个协处理器指令时就会引发该
    # 中断，这样 CPU 就可以有机会让这个中断处理程序模拟协处理器指令（181 行）。
    # CR0 的交换标志 TS 是在 CPU 执行任务转换时设置的。TS 可以用来确定什么时候协处理器中的
    # 内容与 CPU 正在执行的任务不匹配了。当 CPU 在运行一个协处理器转义指令时发现 TS 置位时，
    # 就会引发该中断。此时就可以保存前一个任务的协处理器内容，并恢复新任务的协处理器执行
    # 状态（176 行）。参见 kernel/sched.c，92 行。该中断最后将转移到标号 ret_from_sys_call
    # 处执行下去（检测并处理信号）。
157 .align 2
158 _device_not_available:
159         push %ds
160         push %es
161         push %fs
162         pushl $-1                   # fill in -1 for orig_eax  # 填-1，表明不是系统调用。
163         pushl %edx
164         pushl %ecx
165         pushl %ebx
166         pushl %eax
167         movl $0x10,%eax             # ds,es 置为指向内核数据段。
168         mov %ax,%ds
169         mov %ax,%es
170         movl $0x17,%eax             # fs 置为指向局部数据段（出错程序的数据段）。
171         mov %ax,%fs
    # 清 CR0 中任务已交换标志 TS，并取 CR0 值。若其中协处理器仿真标志 EM 没有置位，说明不是
    # EM 引起的中断，则恢复任务协处理器状态，执行 C 函数 math_state_restore()，并在返回时
    # 去执行 ret_from_sys_call 处的代码。
172         pushl $ret_from_sys_call    # 把下面跳转或调用的返回地址入栈。
173         clts                        # clear TS so that we can use math
174         movl %cr0,%eax
175         testl $0x4,%eax             # EM (math emulation bit)
176         je _math_state_restore      # 执行 math_state_restore()（kernel/sched.c，92 行）。

    # 若 EM 标志置位，则去执行数学仿真程序 math_emulate()。
177         pushl %ebp
178         pushl %esi
179         pushl %edi
180         pushl $0                    # temporary storage for ORIG_EIP
```

```
181         call _math_emulate          # 调用 C 函数（math/math_emulate.c，476 行）。
182         addl $4,%esp                # 丢弃临时存储。
183         popl %edi
184         popl %esi
185         popl %ebp
186         ret                         # 这里的 ret 将跳转到 ret_from_sys_call(107 行)。
187
    #### int32 -- (int 0x20) 时钟中断处理程序。中断频率设置为 100Hz(include/linux/sched.h,4)，
    # 定时芯片 8253/8254 是在(kernel/sched.c,438)处初始化的。因此这里 jiffies 每 10 毫秒加 1。
    # 这段代码将 jiffies 增 1，发送结束中断指令给 8259 控制器，然后用当前特权级作为参数调用
    # C 函数 do_timer(long CPL)。当调用返回时转去检测并处理信号。
188 .align 2
189 _timer_interrupt:
190         push %ds                    # save ds,es and put kernel data space
191         push %es                    # into them. %fs is used by _system_call
192         push %fs                    # 保存 ds、es 并让其指向内核数据段。fs 将用于 system_call。
193         pushl $-1                   # fill in -1 for orig_eax  # 填-1，表明不是系统调用。

    # 下面我们保存寄存器 eax、ecx 和 edx。这是因为 gcc 编译器在调用函数时不会保存它们。这里也
    # 保存了 ebx 寄存器，因为在后面 ret_from_sys_call 中会用到它。
194         pushl %edx                  # we save %eax,%ecx,%edx as gcc doesn't
195         pushl %ecx                  # save those across function calls. %ebx
196         pushl %ebx                  # is saved as we use that in ret_sys_call
197         pushl %eax
198         movl $0x10,%eax             # ds,es 置为指向内核数据段。
199         mov %ax,%ds
200         mov %ax,%es
201         movl $0x17,%eax             # fs 置为指向局部数据段（程序的数据段）。
202         mov %ax,%fs
203         incl _jiffies
    # 由于初始化中断控制芯片时没有采用自动 EOI，所以这里需要发指令结束该硬件中断。
204         movb $0x20,%al              # EOI to interrupt controller #1
205         outb %al,$0x20

    # 下面从堆栈中取出执行系统调用代码的选择符（CS 段寄存器值）中的当前特权级别(0 或 3)并压入
    # 堆栈，作为 do_timer 的参数。do_timer()函数执行任务切换、计时等工作，在 kernel/sched.c，
    # 324 行实现。
206         movl CS(%esp),%eax
207         andl $3,%eax                # %eax is CPL (0 or 3, 0=supervisor)
208         pushl %eax
209         call _do_timer              # 'do_timer(long CPL)' does everything from
210         addl $4,%esp                # task switching to accounting ...
211         jmp ret_from_sys_call
212
    #### 这是 sys_execve()系统调用。取中断调用程序的代码指针作为参数调用 C 函数 do_execve()。
    # do_execve()在 fs/exec.c，207 行。
213 .align 2
214 _sys_execve:
215         lea EIP(%esp),%eax          # eax 指向堆栈中保存用户程序 eip 指针处。
216         pushl %eax
217         call _do_execve
218         addl $4,%esp                # 丢弃调用时压入栈的 EIP 值。
219         ret
```

```
220
    #### sys_fork()调用，用于创建子进程，是 system_call 功能 2。原形在 include/linux/sys.h 中。
    # 首先调用 C 函数 find_empty_process()，取得一个进程号 last_pid。若返回负数则说明目前任务
    # 数组已满。然后调用 copy_process()复制进程。
221 .align 2
222 _sys_fork:
223         call _find_empty_process    # 为新进程取得进程号 last_pid。（kernel/fork.c，143）。
224         testl %eax,%eax             # 在 eax 中返回进程号。若返回负数则退出。
225         js 1f
226         push %gs
227         pushl %esi
228         pushl %edi
229         pushl %ebp
230         pushl %eax
231         call _copy_process          # 调用 C 函数 copy_process()（kernel/fork.c，68）。
232         addl $20,%esp               # 丢弃这里所有压栈内容。
233 1:      ret
234
    #### int 46 -- (int 0x2E) 硬盘中断处理程序，响应硬件中断请求 IRQ14。
    # 当请求的硬盘操作完成或出错就会发出此中断信号。(参见 kernel/blk_drv/hd.c)。
    # 首先向 8259A 中断控制从芯片发送结束硬件中断指令(EOI)，然后取变量 do_hd 中的函数指针放入 edx
    # 寄存器中，并置 do_hd 为 NULL，接着判断 edx 函数指针是否为空。如果为空，则给 edx 赋值指向
    # unexpected_hd_interrupt()，用于显示出错信息。随后向 8259A 主芯片送 EOI 指令，并调用 edx 中
    # 指针指向的函数: read_intr()、write_intr()或 unexpected_hd_interrupt()。
235 _hd_interrupt:
236         pushl %eax
237         pushl %ecx
238         pushl %edx
239         push %ds
240         push %es
241         push %fs
242         movl $0x10,%eax             # ds,es 置为内核数据段。
243         mov %ax,%ds
244         mov %ax,%es
245         movl $0x17,%eax             # fs 置为调用程序的局部数据段。
246         mov %ax,%fs
    # 由于初始化中断控制芯片时没有采用自动 EOI，所以这里需要发指令结束该硬件中断。
247         movb $0x20,%al
248         outb %al,$0xA0              # EOI to interrupt controller #1 # 送从 8259A。
249         jmp 1f                      # give port chance to breathe    # 这里 jmp 起延时作用。
250 1:      jmp 1f
    # do_hd 定义为一个函数指针，将被赋值 read_intr()或 write_intr()函数地址。放到 edx 寄存器后
    # 就将 do_hd 指针变量置为 NULL。然后测试得到的函数指针，若该指针为空，则赋予该指针指向 C
    # 函数 unexpected_hd_interrupt()，以处理未知硬盘中断。
251 1:      xorl %edx,%edx
252         movl %edx,_hd_timeout       # hd_timeout 置为 0。表示控制器已在规定时间内产生了中断。
253         xchgl _do_hd,%edx
254         testl %edx,%edx
255         jne 1f                      # 若空，则让指针指向 C 函数 unexpected_hd_interrupt()。
256         movl $_unexpected_hd_interrupt,%edx
257 1:      outb %al,$0x20              # 送 8259A 主芯片 EOI 指令（结束硬件中断）。
258         call *%edx                  # "interesting" way of handling intr.
259         pop %fs                     # 上句调用 do_hd 指向的 C 函数。
```

```
260         pop %es
261         pop %ds
262         popl %edx
263         popl %ecx
264         popl %eax
265         iret
266
    #### int38 -- (int 0x26) 软盘驱动器中断处理程序，响应硬件中断请求 IRQ6。
    # 其处理过程与上面对硬盘的处理基本一样。（kernel/blk_drv/floppy.c）。
    # 首先向 8259A 中断控制器主芯片发送 EOI 指令，然后取变量 do_floppy 中的函数指针放入 eax
    # 寄存器中，并置 do_floppy 为 NULL，接着判断 eax 函数指针是否为空。如为空，则给 eax 赋值指向
    # unexpected_floppy_interrupt ()，用于显示出错信息。随后调用 eax 指向的函数：rw_interrupt,
    # seek_interrupt,recal_interrupt,reset_interrupt 或 unexpected_floppy_interrupt。
267 _floppy_interrupt:
268         pushl %eax
269         pushl %ecx
270         pushl %edx
271         push %ds
272         push %es
273         push %fs
274         movl $0x10,%eax             # ds,es 置为内核数据段。
275         mov %ax,%ds
276         mov %ax,%es
277         movl $0x17,%eax             # fs 置为调用程序的局部数据段。
278         mov %ax,%fs
279         movb $0x20,%al              # 送主 8259A 中断控制器 EOI 指令（结束硬件中断）。
280         outb %al,$0x20              # EOI to interrupt controller #1
    # do_floppy 为一函数指针，将被赋值实际处理 C 函数指针。该指针在被交换放到 eax 寄存器后就将
    # do_floppy 变量置空。然后测试 eax 中原指针是否为空，若是则使指针指向 C 函数
    # unexpected_floppy_interrupt()。
281         xorl %eax,%eax
282         xchgl _do_floppy,%eax
283         testl %eax,%eax             # 测试函数指针是否=NULL?
284         jne 1f                  # 若空，则使指针指向 C 函数 unexpected_floppy_interrupt()。
285         movl $_unexpected_floppy_interrupt,%eax
286 1:      call *%eax                  # "interesting" way of handling intr.   # 间接调用。
287         pop %fs                     # 上句调用 do_floppy 指向的函数。
288         pop %es
289         pop %ds
290         popl %edx
291         popl %ecx
292         popl %eax
293         iret
294
    #### int 39 -- (int 0x27) 并行口中断处理程序，对应硬件中断请求信号 IRQ7。
    # 本版本内核还未实现。这里只是发送 EOI 指令。
295 _parallel_interrupt:
296         pushl %eax
297         movb $0x20,%al
298         outb %al,$0x20
299         popl %eax
300         iret
```

8.4 程序 8-4 linux/kernel/mktime.c 程序

```
 1 /*
 2  *  linux/kernel/mktime.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <time.h>            // 时间头文件，定义了标准时间数据结构 tm 和一些处理时间函数原型。
 8
 9 /*
10  * This isn't the library routine, it is only used in the kernel.
11  * as such, we don't care about years<1970 etc, but assume everything
12  * is ok. Similarly, TZ etc is happily ignored. We just do everything
13  * as easily as possible. Let's find something public for the library
14  * routines (although I think minix times is public).
15  */
16 /*
17  * PS. I hate whoever though up the year 1970 - couldn't they have gotten
18  * a leap-year instead? I also hate Gregorius, pope or no. I'm grumpy.
19  */
   /*
    * 这不是库函数，它仅供内核使用。因此我们不关心小于 1970 年的年份等，但假定一切均很正常。
    * 同样，时间区域 TZ 问题也先忽略。我们只是尽可能简单地处理问题。最好能找到一些公开的库函数
    * （尽管我认为 minix 的时间函数是公开的）。
    * 另外，我恨那个设置 1970 年开始的人 - 难道他们就不能选择从一个闰年开始？我恨格里高利历、
    * 罗马教皇、主教，我什么都不在乎。我是个脾气暴躁的人。
    */
20 #define MINUTE 60                    // 1 分钟的秒数。
21 #define HOUR (60*MINUTE)             // 1 小时的秒数。
22 #define DAY (24*HOUR)                // 1 天的秒数。
23 #define YEAR (365*DAY)               // 1 年的秒数。
24
25 /* interestingly, we assume leap-years */
   /* 有趣的是我们考虑进了闰年 */
   // 下面以年为界限，定义了每个月开始时的秒数时间。
26 static int month[12] = {
27         0,
28         DAY*(31),
29         DAY*(31+29),
30         DAY*(31+29+31),
31         DAY*(31+29+31+30),
32         DAY*(31+29+31+30+31),
33         DAY*(31+29+31+30+31+30),
34         DAY*(31+29+31+30+31+30+31),
35         DAY*(31+29+31+30+31+30+31+31),
36         DAY*(31+29+31+30+31+30+31+31+30),
37         DAY*(31+29+31+30+31+30+31+31+30+31),
38         DAY*(31+29+31+30+31+30+31+31+30+31+30)
39 };
40
```

```
   // 该函数计算从 1970 年 1 月 1 日 0 时起到开机当日经过的秒数，作为开机时间。
   // 参数 tm 中各字段已经在 init/main.c 中被赋值，信息取自 CMOS。
41 long kernel_mktime(struct tm * tm)
42 {
43         long res;
44         int year;
45
   // 首先计算 70 年到现在经过的年数。因为是 2 位表示方式，所以会有 2000 年问题。我们可以
   // 简单地在最前面添加一条语句来解决这个问题：if (tm->tm_year<70) tm->tm_year += 100;
   // 由于 UNIX 计年份 y 是从 1970 年算起。到 1972 年就是一个闰年，因此过 3 年（71，72，73）
   // 就是第 1 个闰年，这样从 1970 年开始的闰年数计算方法就应该是为 1 + (y - 3)/4，即为
   // (y + 1)/4。res = 这些年经过的秒数时间 + 每个闰年时多 1 天的秒数时间 + 当年到当月时
   // 的秒数。另外，month[]数组中已经在 2 月份的天数中包含进了闰年时的天数，即 2 月份天数
   // 多算了 1 天。因此，若当年不是闰年并且当前月份大于 2 月份的话，我们就要减去这天。因
   // 为从 70 年开始算起，所以当年是闰年的判断方法是 (y + 2) 能被 4 除尽。若不能除尽（有余
   // 数）就不是闰年。
46         year = tm->tm_year - 70;
47 /* magic offsets (y+1) needed to get leapyears right.*/
   /* 为了获得正确的闰年数，这里需要这样一个魔幻值(y+1) */
48         res = YEAR*year + DAY*((year+1)/4);
49         res += month[tm->tm_mon];
50 /* and (y+2) here. If it wasn't a leap-year, we have to adjust */
   /* 以及(y+2)。如果(y+2)不是闰年，那么我们就必须进行调整(减去一天的秒数时间)。*/
51         if (tm->tm_mon>1 && ((year+2)%4))
52                 res -= DAY;
53         res += DAY*(tm->tm_mday-1);          // 再加上本月过去的天数的秒数时间。
54         res += HOUR*tm->tm_hour;             // 再加上当天过去的小时数的秒数时间。
55         res += MINUTE*tm->tm_min;            // 再加上 1 小时内过去的分钟数的秒数时间。
56         res += tm->tm_sec;                   // 再加上 1 分钟内已过的秒数。
57         return res;                          // 即等于从 1970 年以来经过的秒数时间。
58 }
59
```

8.5 程序 8-5 linux/kernel/sched.c

```
 1 /*
 2  *  linux/kernel/sched.c
 3  *
 4  *  (C) 1991  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  */
   /*
    * 'sched.c'是主要的内核文件。其中包括有关调度的基本函数(sleep_on、wakeup、schedule 等)
    * 以及一些简单的系统调用函数（比如 getpid()，仅从当前任务中获取一个字段）。
    */

   // 下面是调度程序头文件。定义了任务结构 task_struct、第 1 个初始任务的数据。还有一些以宏
   // 的形式定义的有关描述符参数设置和获取的嵌入式汇编函数程序。
13 #include <linux/sched.h>
14 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
15 #include <linux/sys.h>    // 系统调用头文件。含有 82 个系统调用 C 函数程序,以'sys_'开头。
16 #include <linux/fdreg.h>  // 软驱头文件。含有软盘控制器参数的一些定义。
17 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
18 #include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
19 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
20
21 #include <signal.h>       // 信号头文件。定义信号符号常量，sigaction 结构，操作函数原型。
22
   // 该宏取信号 nr 在信号位图中对应位的二进制数值。信号编号 1-32。比如信号 5 的位图数值等于
   // 1<<(5-1) = 16 = 00010000b。
23 #define _S(nr) (1<<((nr)-1))
   // 除了 SIGKILL 和 SIGSTOP 信号以外其他信号都是可阻塞的(…1011,1111,1110,1111,1111b)。
24 #define _BLOCKABLE (~(_S(SIGKILL) | _S(SIGSTOP)))
25
   // 内核调试函数。显示任务号 nr 的进程号、进程状态和内核堆栈空闲字节数（大约）。
26 void show_task(int nr,struct task_struct * p)
27 {
28         int i,j = 4096-sizeof(struct task_struct);
29
30         printk("%d: pid=%d, state=%d, father=%d, child=%d, ",nr,p->pid,
31                 p->state, p->p_pptr->pid, p->p_cptr ? p->p_cptr->pid : -1);
32         i=0;
33         while (i<j && !((char *)(p+1))[i])  // 检测指定任务数据结构以后等于 0 的字节数。
34                 i++;
35         printk("%d/%d chars free in kstack\n\r",i,j);
36         printk("   PC=%08X.", *(1019 + (unsigned long *) p));
37         if (p->p_ysptr || p->p_osptr)
38                 printk("   Younger sib=%d, older sib=%d\n\r",
```

```
39                          p->p_ysptr ? p->p_ysptr->pid : -1,
40                          p->p_osptr ? p->p_osptr->pid : -1);
41          else
42                  printk("\n\r");
43  }
44
    // 显示所有任务的任务号、进程号、进程状态和内核堆栈空闲字节数（大约）。
    // NR_TASKS 是系统能容纳的最大进程(任务)数量(64 个)，定义在 include/kernel/sched.h 第 6 行。
45  void show_state(void)
46  {
47          int i;
48
49          printk("\rTask-info:\n\r");
50          for (i=0;i<NR_TASKS;i++)
51                  if (task[i])
52                          show_task(i,task[i]);
53  }
54
    // PC 机 8253 定时芯片的输入时钟频率约为 1.193180MHz。Linux 内核希望定时器发出中断的频率是
    // 100Hz，也即每 10ms 发出一次时钟中断。因此这里 LATCH 是设置 8253 芯片的初值，参见 438 行。
55  #define LATCH (1193180/HZ)
56
57  extern void mem_use(void);          // [??]没有任何地方定义和引用该函数。
58
59  extern int timer_interrupt(void);  // 时钟中断处理程序（kernel/system_call.s，176）。
60  extern int system_call(void);      // 系统调用中断处理程序（kernel/system_call.s，80）。
61
    // 每个任务（进程）在内核态运行时都有自己的内核态堆栈。这里定义了任务的内核态堆栈结构。
    // 这里定义任务联合（任务结构成员和 stack 字符数组成员）。因为一个任务的数据结构与其内核
    // 态堆栈放在同一内存页中，所以从堆栈段寄存器 ss 可以获得其数据段选择符。
62  union task_union {
63          struct task_struct task;
64          char stack[PAGE_SIZE];
65  };
66
    // 设置初始任务的数据。初始数据在 include/kernel/sched.h 中，第 156 行开始。
67  static union task_union init_task = {INIT_TASK,};
68
    // 从开机开始算起的滴答数时间值全局变量（10ms/滴答）。系统时钟中断每发生一次即一个滴答。
    // 前面的限定符 volatile，英文解释是易改变的、不稳定的意思。这个限定词的含义是向编译器
    // 指明变量的内容可能会由于被其他程序修改而变化。通常在程序中申明一个变量时， 编译器会
    // 尽量把它存放在通用寄存器中，例如 ebx，以提高访问效率。当 CPU 把其值放到 ebx 中后一般
    // 就不会再关心该变量对应内存位置中的内容。若此时其他程序（例如内核程序或一个中断过程）
    // 修改了内存中该变量的值，ebx 中的值并不会随之更新。为了解决这种情况就创建了 volatile
    // 限定符，让代码在引用该变量时一定要从指定内存位置中取得其值。这里即是要求 gcc 不要对
    // jiffies 进行优化处理，也不要挪动位置，并且需要从内存中取其值。因为时钟中断处理过程
    // 等程序会修改它的值。
69  unsigned long volatile jiffies=0;
70  unsigned long startup_time=0;                   // 开机时间。从 1970:0:0:0 开始计时的秒数。
    // 这个变量用于累计需要调整地时间嘀嗒数。
71  int jiffies_offset = 0;         /* # clock ticks to add to get "true
72                                     time".  Should always be less than
73                                     1 second's worth.  For time fanatics
```

```
74                                      who like to syncronize their machines
75                                      to WWV :-) */
   /* 为调整时钟而需要增加的时钟嘀嗒数，以获得“精确时间”。这些调整用嘀嗒数
    * 的总和不应该超过 1 秒。这样做是为了那些对时间精确度要求苛刻的人，他们喜
    * 欢自己的机器时间与 WWV 同步 :-)
    */
76
77 struct task_struct *current = &(init_task.task);  // 当前任务指针（初始化指向任务 0）。
78 struct task_struct *last_task_used_math = NULL;   // 使用过协处理器任务的指针。
79
   // 定义任务指针数组。第 1 项被初始化指向初始任务（任务 0）的任务数据结构。
80 struct task_struct * task[NR_TASKS] = {&(init_task.task), };
81
   // 定义用户堆栈，共 1K 项，容量 4K 字节。在内核初始化操作过程中被用作内核栈，初始化完成
   // 以后将被用作任务 0 的用户态堆栈。在运行任务 0 之前它是内核栈，以后用作任务 0 和 1 的用
   // 户态栈。下面结构用于设置堆栈 ss:esp（数据段选择符，指针），见 head.s，第 23 行。
   // ss 被设置为内核数据段选择符（0x10），指针 esp 指在 user_stack 数组最后一项后面。这是
   // 因为 Intel CPU 执行堆栈操作时是先递减堆栈指针 sp 值，然后在 sp 指针处保存入栈内容。
82 long user_stack [ PAGE_SIZE>>2 ] ;
83
84 struct {
85         long * a;
86         short b;
87         } stack_start = { & user_stack [PAGE_SIZE>>2] , 0x10 };
88 /*
89  *  'math_state_restore()' saves the current math information in the
90  * old math state array, and gets the new ones from the current task
91  */
   /*
    * 将当前协处理器内容保存到老协处理器状态数组中，并将当前任务的协处理器
    * 内容加载进协处理器。
    */
   // 当任务被调度交换过以后，该函数用以保存原任务的协处理器状态（上下文）并恢复新调度进
   // 来的当前任务的协处理器执行状态。
92 void math_state_restore()
93 {
   // 如果任务没变则返回(上一个任务就是当前任务)。这里"上一个任务"是指刚被交换出去的任务。
94         if (last_task_used_math == current)
95                 return;
   // 在发送协处理器命令之前要先发 WAIT 指令。如果上个任务使用了协处理器，则保存其状态。
96         __asm__("fwait");
97         if (last_task_used_math) {
98                 __asm__("fnsave %0"::"m" (last_task_used_math->tss.i387));
99         }
   // 现在，last_task_used_math 指向当前任务，以备当前任务被交换出去时使用。此时如果当前
   // 任务用过协处理器，则恢复其状态。否则的话说明是第一次使用，于是就向协处理器发初始化
   // 命令，并设置使用了协处理器标志。
100        last_task_used_math=current;
101        if (current->used_math) {
102                __asm__("frstor %0"::"m" (current->tss.i387));
103        } else {
104                __asm__("fninit"::);            // 向协处理器发初始化命令。
105                current->used_math=1;           // 设置使用已协处理器标志。
```

```
106         }
107 }
108
109 /*
110  *  'schedule()' is the scheduler function.  This is GOOD CODE! There
111  * probably won't be any reason to change this, as it should work well
112  * in all circumstances (ie gives IO-bound processes good response etc).
113  * The one thing you might take a look at is the signal-handler code here.
114  *
115  *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
116  * tasks can run.  It can not be killed, and it cannot sleep. The 'state'
117  * information in task[0] is never used.
118  */
    /*
     * 'schedule()'是调度函数。这是个很好的代码！没有任何理由对它进行修改，因为
     * 它可以在所有的环境下工作（比如能够对 IO-边界处理很好的响应等）。只有一件
     * 事值得留意，那就是这里的信号处理代码。
     *
     *   注意！！任务 0 是个闲置('idle')任务，只有当没有其他任务可以运行时才调用
     * 它。它不能被杀死，也不能睡眠。任务 0 中的状态信息'state'是从来不用的。
     */
119 void schedule(void)
120 {
121         int i,next,c;
122         struct task_struct ** p;                // 任务结构指针的指针。
123
124 /* check alarm, wake up any interruptible tasks that have got a signal */
    /* 检测 alarm（进程的报警定时值），唤醒任何已得到信号的可中断任务 */
125
    // 从任务数组中最后一个任务开始循环检测 alarm。在循环时跳过空指针项。
126         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
127                 if (*p) {
    // 如果设置过任务超时定时 timeout，并且已经超时，则复位超时定时值，并且如果任务处于可
    // 中断睡眠状态 TASK_INTERRUPTIBLE 下，将其置为就绪状态（TASK_RUNNING）。
128                         if ((*p)->timeout && (*p)->timeout < jiffies) {
129                                 (*p)->timeout = 0;
130                                 if ((*p)->state == TASK_INTERRUPTIBLE)
131                                         (*p)->state = TASK_RUNNING;
132                         }
    // 如果设置过任务的定时值 alarm，并且已经过期(alarm<jiffies),则在信号位图中置 SIGALRM
    // 信号，即向任务发送 SIGALARM 信号。然后清 alarm。该信号的默认操作是终止进程。jiffies
    // 是系统从开机开始算起的滴答数（10ms/滴答）。定义在 sched.h 第 139 行。
133                         if ((*p)->alarm && (*p)->alarm < jiffies) {
134                                 (*p)->signal |= (1<<(SIGALRM-1));
135                                 (*p)->alarm = 0;
136                         }
    // 如果信号位图中除被阻塞的信号外还有其他信号，并且任务处于可中断状态，则置任务为就绪
    // 状态。其中'~(_BLOCKABLE & (*p)->blocked)'用于忽略被阻塞的信号，但 SIGKILL 和 SIGSTOP
    // 不能被阻塞。
137                         if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
138                         (*p)->state==TASK_INTERRUPTIBLE)
139                                 (*p)->state=TASK_RUNNING;   //置为就绪（可执行）状态。
140                 }
```

```
141
142 /* this is the scheduler proper: */
    /* 这里是调度程序的主要部分 */
143
144         while (1) {
145                 c = -1;
146                 next = 0;
147                 i = NR_TASKS;
148                 p = &task[NR_TASKS];
    // 这段代码也是从任务数组的最后一个任务开始循环处理，并跳过不含任务的数组槽。比较每个
    // 就绪状态任务的 counter（任务运行时间的递减滴答计数）值，哪一个值大，运行时间还不长，
    // next 就指向哪个的任务号。
149                 while (--i) {
150                         if (!*--p)
151                                 continue;
152                         if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
153                                 c = (*p)->counter, next = i;
154                 }
    // 如果比较得出有 counter 值不等于 0 的结果，或者系统中没有一个可运行的任务存在（此时 c
    // 仍然为-1，next=0），则退出 144 行开始的循环，执行 161 行上的任务切换操作。否则就根据
    // 每个任务的优先权值，更新每一个任务的 counter 值，然后回到 125 行重新比较。counter 值
    // 的计算方式为 counter = counter /2 + priority。注意，这里计算过程不考虑进程的状态。
155                 if (c) break;
156                 for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
157                         if (*p)
158                                 (*p)->counter = ((*p)->counter >> 1) +
159                                                 (*p)->priority;
160         }
    // 用下面宏（定义在 sched.h 中）把当前任务指针 current 指向任务号为 next 的任务，并切换
    // 到该任务中运行。在 146 行上 next 被初始化为 0。因此若系统中没有任何其他任务可运行时，
    // 则 next 始终为 0。因此调度函数会在系统空闲时去执行任务 0。 此时任务 0 仅执行 pause()
    // 系统调用，并又会调用本函数。
161         switch_to(next);                        // 切换到任务号为 next 的任务，并运行之。
162 }
163
    //// pause()系统调用。转换当前任务的状态为可中断的等待状态，并重新调度。
    // 该系统调用将导致进程进入睡眠状态，直到收到一个信号。该信号用于终止进程或者使进程
    // 调用一个信号捕获函数。只有当捕获了一个信号，并且信号捕获处理函数返回，pause()才
    // 会返回。此时 pause()返回值应该是 -1，并且 errno 被置为 EINTR。这里还没有完全实现
    // （直到 0.95 版）。
164 int sys_pause(void)
165 {
166         current->state = TASK_INTERRUPTIBLE;
167         schedule();
168         return 0;
169 }
170
    // 把当前任务置为指定的睡眠状态（可中断的或不可中断的），并让睡眠队列头指针指向当前任务。
    // 函数参数 p 是等待任务队列头指针。指针是含有一个变量地址的变量。这里参数 p 使用了指针的
    // 指针形式 '**p'，这是因为 C 函数参数只能传值，没有直接的方式让被调用函数改变调用该函数
    // 程序中变量的值。但是指针'*p'指向的目标（这里是任务结构）会改变，因此为了能修改调用该
    // 函数程序中原来就是指针变量的值，就需要传递指针'*p'的指针，即'**p'。参见程序前示例图中
    // p 指针的使用情况。
```

```
    // 参数 state 是任务睡眠使用的状态：TASK_UNINTERRUPTIBLE 或 TASK_INTERRUPTIBLE。处于不可
    // 中断睡眠状态（TASK_UNINTERRUPTIBLE）的任务需要内核程序利用 wake_up()函数明确唤醒之。
    // 处于可中断睡眠状态（TASK_INTERRUPTIBLE）可以通过信号、任务超时等手段唤醒 （置为就绪
    // 状态 TASK_RUNNING）。
    // *** 注意，由于本内核代码不是很成熟，因此下列与睡眠相关的代码存在一些问题，不宜深究。
171 static inline void __sleep_on(struct task_struct **p, int state)
172 {
173         struct task_struct *tmp;
174
    // 若指针无效，则退出。（指针所指的对象可以是 NULL，但指针本身不会为 0）。
    // 如果当前任务是任务 0，则死机(impossible!)。
175         if (!p)
176                 return;
177         if (current == &(init_task.task))
178                 panic("task[0] trying to sleep");
    // 让 tmp 指向已经在等待队列上的任务(如果有的话)，例如 inode->i_wait。并且将睡眠队列头
    // 的等待指针指向当前任务。这样就把当前任务插入到了 *p 的等待队列中。然后将当前任务置
    // 为指定的等待状态，并执行重新调度。
179         tmp = *p;
180         *p = current;
181         current->state = state;
182 repeat: schedule();
    // 只有当这个等待任务被唤醒时，程序才又会返回到这里，表示进程已被明确地唤醒并执行。
    // 如果等待队列中还有等待任务，并且队列头指针 *p 所指向的任务不是当前任务时，说明
    // 在本任务插入等待队列后还有任务进入等待队列。于是我们应该也要唤醒这个任务，而我
    // 们自己应按顺序让这些后面进入队列的任务唤醒，因此这里将等待队列头所指任务先置为
    // 就绪状态，而自己则置为不可中断等待状态，即自己要等待这些后续进队列的任务被唤醒
    // 而执行时来唤醒本任务。然后重新执行调度程序。
183         if (*p && *p != current) {
184                 (**p).state = 0;
185                 current->state = TASK_UNINTERRUPTIBLE;
186                 goto repeat;
187         }
    // 执行到这里，说明本任务真正被唤醒执行。此时等待队列头指针应该指向本任务，若它为
    // 空，则表明调度有问题，于是显示警告信息。最后我们让头指针指向在我们前面进入队列
    // 的任务（*p = tmp）。 若确实存在这样一个任务，即队列中还有任务（tmp 不为空），就
    // 唤醒之。最先进入队列的任务在唤醒后运行时最终会把等待队列头指针置成 NULL。
188         if (!*p)
189                 printk("Warning: *P = NULL\n\r");
190         if (*p = tmp)
191                 tmp->state=0;
192 }
193
    // 将当前任务置为可中断的等待状态（TASK_INTERRUPTIBLE），并放入头指针*p 指定的等待
    // 队列中。
194 void interruptible_sleep_on(struct task_struct **p)
195 {
196         __sleep_on(p,TASK_INTERRUPTIBLE);
197 }
198
    // 把当前任务置为不可中断的等待状态（TASK_UNINTERRUPTIBLE），并让睡眠队列头指针指向
    // 当前任务。只有明确地唤醒时才会返回。该函数提供了进程与中断处理程序之间的同步机制。
199 void sleep_on(struct task_struct **p)
```

```
200 {
201         sleep_on(p,TASK_UNINTERRUPTIBLE);
202 }
203
    // 唤醒 *p 指向的任务。*p 是任务等待队列头指针。由于新等待任务是插入在等待队列头指针
    // 处的，因此唤醒的是最后进入等待队列的任务。若该任务已经处于停止或僵死状态，则显示
    // 警告信息。
204 void wake_up(struct task_struct **p)
205 {
206         if (p && *p) {
207                 if ((**p).state == TASK_STOPPED)          // 处于停止状态。
208                         printk("wake_up: TASK_STOPPED");
209                 if ((**p).state == TASK_ZOMBIE)           // 处于僵死状态。
210                         printk("wake_up: TASK_ZOMBIE");
211                 (**p).state=0;                            // 置为就绪状态 TASK_RUNNING。
212         }
213 }
214
215 /*
216  * OK, here are some floppy things that shouldn't be in the kernel
217  * proper. They are here because the floppy needs a timer, and this
218  * was the easiest way of doing it.
219  */
    /*
     * 好了，从这里开始是一些有关软盘的子程序，本不应该放在内核的主要部分
     * 中的。将它们放在这里是因为软驱需要定时处理，而放在这里是最方便的。
     */
    // 下面 220 -- 281 行代码用于处理软驱定时。在阅读这段代码之前请先看一下块设备一章中
    // 有关软盘驱动程序（floppy.c）后面的说明， 或者到阅读软盘块设备驱动程序时在来看这
    // 段代码。其中时间单位：1 个滴答 = 1/100 秒。
    // 下面数组 wait_motor[]用于存放等待软驱马达启动到正常转速的进程指针。数组索引 0-3
    // 分别对应软驱 A--D。数组 mon_timer[]存放各软驱马达启动所需要的滴答数。程序中默认
    // 启动时间为 50 个滴答（0.5 秒）。数组 moff_timer[] 存放各软驱在马达停转之前需维持
    // 的时间。程序中设定为 10000 个滴答（100 秒）。
220 static struct task_struct * wait_motor[4] = {NULL,NULL,NULL,NULL};
221 static int  mon_timer[4]={0,0,0,0};
222 static int moff_timer[4]={0,0,0,0};

    // 下面变量对应软驱控制器中当前数字输出寄存器。该寄存器每位的定义如下：
    // 位 7-4：分别控制驱动器 D-A 马达的启动。1 - 启动；0 - 关闭。
    // 位 3  ：1 - 允许 DMA 和中断请求；0 - 禁止 DMA 和中断请求。
    // 位 2  ：1 - 启动软盘控制器；     0 - 复位软盘控制器。
    // 位 1-0：00 - 11，用于选择控制的软驱 A-D。
    // 这里设置初值为：允许 DMA 和中断请求、启动 FDC。
223 unsigned char current_DOR = 0x0C;
224
    // 指定软驱启动到正常运转状态所需等待时间。
    // 参数 nr -- 软驱号(0--3)，返回值为滴答数。
    // 局部变量 selected 是选中软驱标志（blk_drv/floppy.c，123 行）。mask 是所选软驱对应的
    // 数字输出寄存器中启动马达比特位。mask 高 4 位是各软驱启动马达标志。
225 int ticks_to_floppy_on(unsigned int nr)
226 {
227         extern unsigned char selected;
```

```
228         unsigned char mask = 0x10 << nr;
229
    // 系统最多有 4 个软驱。首先预先设置好指定软驱 nr 停转之前需要经过的时间（100 秒）。然后
    // 取当前 DOR 寄存器值到临时变量 mask 中，并把指定软驱的马达启动标志置位。
230         if (nr>3)
231                 panic("floppy_on: nr>3");
232         moff_timer[nr]=10000;           /* 100 s = very big :-) */   // 停转维持时间。
233         cli();                          /* use floppy_off to turn it off */ // 关中断。
234         mask |= current_DOR;
    // 如果当前没有选择软驱，则首先复位其他软驱的选择位，然后置指定软驱选择位。
235         if (!selected) {
236                 mask &= 0xFC;
237                 mask |= nr;
238         }
    // 如果数字输出寄存器的当前值与要求的值不同，则向 FDC 数字输出端口输出新值(mask)，并且
    // 如果要求启动的马达还没有启动，则置相应软驱的马达启动定时器值（HZ/2 = 0.5 秒或 50 个
    // 滴答）。若已经启动，则再设置启动定时为 2 个滴答，能满足下面 do_floppy_timer()中先递
    // 减后判断的要求。执行本次定时代码的要求即可。此后更新当前数字输出寄存器 current_DOR。
239         if (mask != current_DOR) {
240                 outb(mask,FD_DOR);
241                 if ((mask ^ current_DOR) & 0xf0)
242                         mon_timer[nr] = HZ/2;
243                 else if (mon_timer[nr] < 2)
244                         mon_timer[nr] = 2;
245                 current_DOR = mask;
246         }
247         sti();                          // 开中断。
248         return mon_timer[nr];           // 最后返回启动马达所需的时间值。
249 }
250
    // 等待指定软驱马达启动所需的一段时间，然后返回。
    // 设置指定软驱的马达启动到正常转速所需的延时，然后睡眠等待。在定时中断过程中会一直
    // 递减判断这里设定的延时值。当延时到期，就会唤醒这里的等待进程。
251 void floppy_on(unsigned int nr)
252 {
    // 关中断。如果马达启动定时还没到，就一直把当前进程置为不可中断睡眠状态并放入等待马达
    // 运行的队列中。然后开中断。
253         cli();
254         while (ticks_to_floppy_on(nr))
255                 sleep_on(nr+wait_motor);
256         sti();
257 }
258
    // 置关闭相应软驱马达停转定时器（3 秒）。
    // 若不使用该函数明确关闭指定的软驱马达，则在马达开启 100 秒之后也会被关闭。
259 void floppy_off(unsigned int nr)
260 {
261         moff_timer[nr]=3*HZ;
262 }
263
    // 软盘定时处理子程序。更新马达启动定时值和马达关闭停转计时值。该子程序会在时钟定时
    // 中断过程中被调用，因此系统每经过一个滴答(10ms)就会被调用一次，随时更新马达开启或
    // 停转定时器的值。如果某一个马达停转定时到，则将数字输出寄存器马达启动位复位。
```

```
264 void do_floppy_timer(void)
265 {
266         int i;
267         unsigned char mask = 0x10;
268
269         for (i=0 ; i<4 ; i++,mask <<= 1) {
270                 if (!(mask & current_DOR))              // 如果不是 DOR 指定的马达则跳过。
271                         continue;
272                 if (mon_timer[i]) {                     // 如果马达启动定时到则唤醒进程。
273                         if (!--mon_timer[i])
274                                 wake_up(i+wait_motor);
275                 } else if (!moff_timer[i]) {            // 如果马达停转定时到则
276                         current_DOR &= ~mask;           // 复位相应马达启动位，并且
277                         outb(current_DOR,FD_DOR);       // 更新数字输出寄存器。
278                 } else
279                         moff_timer[i]--;                // 否则马达停转计时递减。
280         }
281 }
282
    // 下面是关于定时器的代码。最多可有 64 个定时器。
283 #define TIME_REQUESTS 64
284
    // 定时器链表结构和定时器数组。该定时器链表专用于供软驱关闭马达和启动马达定时操作。
    // 这种类型定时器类似现代 Linux 系统中的动态定时器（Dynamic Timer），仅供内核使用。
285 static struct timer_list {
286         long jiffies;                   // 定时滴答数。
287         void (*fn)();                   // 定时处理程序。
288         struct timer_list * next;       // 链接指向下一个定时器。
289 } timer_list[TIME_REQUESTS], * next_timer = NULL;  // next_timer 是定时器队列头指针。
290
    // 添加定时器。输入参数为指定的定时值(滴答数)和相应的处理程序指针。
    // 软盘驱动程序（floppy.c）利用该函数执行启动或关闭马达的延时操作。
    // 参数 jiffies - 以 10 毫秒计的滴答数；*fn()- 定时时间到时执行的函数。
291 void add_timer(long jiffies, void (*fn)(void))
292 {
293         struct timer_list * p;
294
    // 如果定时处理程序指针为空，则退出。否则关中断。
295         if (!fn)
296                 return;
297         cli();
    // 如果定时值<=0，则立刻调用其处理程序。并且该定时器不加入链表中。
298         if (jiffies <= 0)
299                 (fn)();
300         else {
    // 否则从定时器数组中，找一个空闲项。
301                 for (p = timer_list ; p < timer_list + TIME_REQUESTS ; p++)
302                         if (!p->fn)
303                                 break;
    // 如果已经用完了定时器数组，则系统崩溃☺。否则向定时器数据结构填入相应信息，并链入
    // 链表头。
304                 if (p >= timer_list + TIME_REQUESTS)
305                         panic("No more time requests free");
```

```
306                 p->fn = fn;
307                 p->jiffies = jiffies;
308                 p->next = next_timer;
309                 next_timer = p;
    // 链表项按定时值从小到大排序。在排序时减去排在前面需要的滴答数，这样在处理定时器时
    // 只要查看链表头的第一项的定时是否到期即可。[[?? 这段程序好象没有考虑周全。如果新
    // 插入的定时器值小于原来头一个定时器值时则根本不会进入循环中，但此时还是应该将紧随
    // 其后面的一个定时器值减去新的第 1 个的定时值。即如果第 1 个定时值<=第 2 个，则第 2 个
    // 定时值扣除第 1 个的值即可，否则进入下面循环中进行处理。]]
310                 while (p->next && p->next->jiffies < p->jiffies) {
311                         p->jiffies -= p->next->jiffies;
312                         fn = p->fn;
313                         p->fn = p->next->fn;
314                         p->next->fn = fn;
315                         jiffies = p->jiffies;
316                         p->jiffies = p->next->jiffies;
317                         p->next->jiffies = jiffies;
318                         p = p->next;
319                 }
320         }
321         sti();
322 }
323
    //// 时钟中断 C 函数处理程序，在 sys_call.s 中的_timer_interrupt（189 行）被调用。
    // 参数 cpl 是当前特权级 0 或 3，是时钟中断发生时正被执行的代码选择符中的特权级。
    // cpl=0 时表示中断发生时正在执行内核代码；cpl=3 时表示中断发生时正在执行用户代码。
    // 对于一个进程由于执行时间片用完时，则进行任务切换。并执行一个计时更新工作。
324 void do_timer(long cpl)
325 {
326         static int blanked = 0;
327
    // 首先判断是否经过了一定时间而让屏幕黑屏（blankout）。如果 blankcount 计数不为零，
    // 或者黑屏延时间隔时间 blankinterval 为 0 的话，那么若已经处于黑屏状态（黑屏标志
    // blanked = 1），则让屏幕恢复显示。若 blankcount 计数不为零，则递减之，并且复位
    // 黑屏标志。
328         if (blankcount || !blankinterval) {
329                 if (blanked)
330                         unblank_screen();
331                 if (blankcount)
332                         blankcount--;
333                 blanked = 0;
    // 否则的话若黑屏标志未置位，则让屏幕黑屏，并且设置黑屏标志。
334         } else if (!blanked) {
335                 blank_screen();
336                 blanked = 1;
337         }
    // 接着处理硬盘操作超时问题。如果硬盘超时计数递减之后为 0，则进行硬盘访问超时处理。
338         if (hd_timeout)
339                 if (!--hd_timeout)
340                         hd_times_out();  // 硬盘访问超时处理（blk_drv/hdc，318 行）。
341
    // 如果发声计数次数到，则关闭发声。(向 0x61 口发送命令，复位位 0 和 1。位 0 控制 8253
    // 计数器 2 的工作，位 1 控制扬声器)。
```

```
342         if (beepcount)          // 扬声器发声时间滴答数（chr_drv/console.c,950 行）。
343                 if (!--beepcount)
344                         sysbeepstop();
345
    // 如果当前特权级(cpl)为 0（最高，表示是内核程序在工作），则将内核代码运行时间 stime
    // 递增；[ Linus 把内核程序统称为超级用户(supervisor)的程序，见 sys_call.s，207 行
    // 上的英文注释。这种称呼来自于 Intel CPU 手册。] 如果 cpl > 0，则表示是一般用户程序
    // 在工作，增加 utime。
346         if (cpl)
347                 current->utime++;
348         else
349                 current->stime++;
350
    // 如果有定时器存在，则将链表第 1 个定时器的值减 1。如果已等于 0，则调用相应的处理程序，
    // 并将该处理程序指针置为空。然后去掉该项定时器。next_timer 是定时器链表的头指针。
351         if (next_timer) {
352                 next_timer->jiffies--;
353                 while (next_timer && next_timer->jiffies <= 0) {
354                         void (*fn)(void);       // 这里插入了一个函数指针定义！！☹
355
356                         fn = next_timer->fn;
357                         next_timer->fn = NULL;
358                         next_timer = next_timer->next;
359                         (fn)();                 // 调用定时处理函数。
360                 }
361         }
    // 如果当前软盘控制器 FDC 的数字输出寄存器中马达启动位有置位的，则执行软盘定时程序。
362         if (current_DOR & 0xf0)
363                 do_floppy_timer();
  // 如果进程运行时间还没完，则退出。否则置当前任务运行计数值为 0。并且若发生时钟中断时
  // 正在内核代码中运行则返回，否则调用执行调度函数。
364         if ((--current->counter)>0) return;
365         current->counter=0;
366         if (!cpl) return;               // 对于内核态程序，不依赖 counter 值进行调度。
367         schedule();
368 }
369
    // 系统调用功能 - 设置报警定时时间值(秒)。
    // 若参数 seconds 大于 0，则设置新定时值，并返回原定时时刻还剩余的间隔时间。否则返回 0。
    // 进程数据结构中报警定时值 alarm 的单位是系统滴答（1 滴答为 10 毫秒），它是系统开机起到
    // 设置定时操作时系统滴答值 jiffies 和转换成滴答单位的定时值之和，即'jiffies + HZ*定时
    // 秒值'。而参数给出的是以秒为单位的定时值，因此本函数的主要操作是进行两种单位的转换。
    // 其中常数 HZ = 100，是内核系统运行频率。定义在 include/sched.h 第 4 行上。
    // 参数 seconds 是新的定时时间值，单位是秒。
370 int sys_alarm(long seconds)
371 {
372         int old = current->alarm;
373
374         if (old)
375                 old = (old - jiffies) / HZ;
376         current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
377         return (old);
378 }
```

```
379
    // 取当前进程号 pid。
380 int sys_getpid(void)
381 {
382         return current->pid;
383 }
384
    // 取父进程号 ppid。
385 int sys_getppid(void)
386 {
387         return current->p_pptr->pid;
388 }
389
    // 取用户号 uid。
390 int sys_getuid(void)
391 {
392         return current->uid;
393 }
394
    // 取有效的用户号 euid。
395 int sys_geteuid(void)
396 {
397         return current->euid;
398 }
399
    // 取组号 gid。
400 int sys_getgid(void)
401 {
402         return current->gid;
403 }
404
    // 取有效的组号 egid。
405 int sys_getegid(void)
406 {
407         return current->egid;
408 }
409
    // 系统调用功能 -- 降低对 CPU 的使用优先权（有人会用吗？☺）。
    // 应该限制 increment 为大于 0 的值，否则可使优先权增大！！
410 int sys_nice(long increment)
411 {
412         if (current->priority-increment>0)
413                 current->priority -= increment;
414         return 0;
415 }
416
    // 内核调度程序的初始化子程序。
417 void sched_init(void)
418 {
419         int i;
420         struct desc_struct * p;         // 描述符表结构指针。
421
    // Linux 系统开发之初，内核不成熟。内核代码会被经常修改。Linus 怕自己无意中修改了这些
```

```
    // 关键性的数据结构，造成与 POSIX 标准的不兼容。这里加入下面这个判断语句并无必要，纯粹
    // 是为了提醒自己以及其他修改内核代码的人。
422         if (sizeof(struct sigaction) != 16)     // sigaction 是存放有关信号状态的结构。
423                 panic("Struct sigaction MUST be 16 bytes");
    // 在全局描述符表中设置初始任务（任务 0）的任务状态段描述符和局部数据表描述符。
    // FIRST_TSS_ENTRY 和 FIRST_LDT_ENTRY 的值分别是 4 和 5，定义在 include/linux/sched.h
    // 中；gdt 是一个描述符表数组（include/linux/head.h ），实际上对应程序 head.s 中
    // 第 234 行上的全局描述符表基址（_gdt）。因此 gdt + FIRST_TSS_ENTRY 即为
    // gdt[FIRST_TSS_ENTRY]（即是 gdt[4]），也即 gdt 数组第 4 项的地址。 参见
    // include/asm/system.h,第 65 行开始。
424         set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
425         set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
    // 清任务数组和描述符表项（注意 i=1 开始，所以初始任务的描述符还在）。描述符项结构
    // 定义在文件 include/linux/head.h 中。
426         p = gdt+2+FIRST_TSS_ENTRY;
427         for(i=1;i<NR_TASKS;i++) {
428                 task[i] = NULL;
429                 p->a=p->b=0;
430                 p++;
431                 p->a=p->b=0;
432                 p++;
433         }
434 /* Clear NT, so that we won't have troubles with that later on */
    /* 清除标志寄存器中的位 NT，这样以后就不会有麻烦 */
    // EFLAGS 中的 NT 标志位用于控制任务的嵌套调用。当 NT 位置位时，那么当前中断任务执行
    // IRET 指令时就会引起任务切换。NT 指出 TSS 中的 back_link 字段是否有效。NT=0 时无效。
435         __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");    // 复位 NT 标志。
    // 将任务 0 的 TSS 段选择符加载到任务寄存器 tr。将局部描述符表段选择符加载到局部描述
    // 符表寄存器 ldtr 中。注意！！是将 GDT 中相应 LDT 描述符的选择符加载到 ldtr。只明确加
    // 这一次，以后新任务 LDT 的加载，是 CPU 根据 TSS 中的 LDT 项自动加载。
436         ltr(0);                     // 定义在 include/linux/sched.h 第 157-158 行。
437         lldt(0);                    // 其中参数（0）是任务号。
    // 下面代码用于初始化 8253 定时器。通道 0，选择工作方式 3，二进制计数方式。通道 0 的
    // 输出引脚接在中断控制主芯片的 IRQ0 上，它每 10 毫秒发出一个 IRQ0 请求。LATCH 是初始
    // 定时计数值。
438         outb_p(0x36,0x43);              /* binary, mode 3, LSB/MSB, ch 0 */
439         outb_p(LATCH & 0xff , 0x40);    /* LSB */   // 定时值低字节。
440         outb(LATCH >> 8 , 0x40);        /* MSB */   // 定时值高字节。
    // 设置时钟中断处理程序句柄（设置时钟中断门）。修改中断控制器屏蔽码，允许时钟中断。
    // 然后设置系统调用中断门。这两个设置中断描述符表 IDT 中描述符的宏定义在文件
    // include/asm/system.h 中第 33、39 行处。两者的区别参见 system.h 文件开始处的说明。
441         set_intr_gate(0x20,&timer_interrupt);
442         outb(inb_p(0x21)&~0x01,0x21);
443         set_system_gate(0x80,&system_call);
444 }
445
```

8.6 程序 8-6 linux/kernel/signal.c

```
 1 /*
 2  *  linux/kernel/signal.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、初始任务 0 的数据，
                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
 8 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
 9 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
10
11 #include <signal.h>       // 信号头文件。定义信号符号常量，信号结构及信号操作函数原型。
12 #include <errno.h>        // 出错号头文件。定义出错号符号常量。
13
   // 获取当前任务信号屏蔽位图（屏蔽码或阻塞码）。sgetmask 可分解为 signal-get-mask。以下类似。
14 int sys_sgetmask()
15 {
16         return current->blocked;
17 }
18
   // 设置新的信号屏蔽位图。信号 SIGKILL 和 SIGSTOP 不能被屏蔽。返回值是原信号屏蔽位图。
19 int sys_ssetmask(int newmask)
20 {
21         int old=current->blocked;
22
23         current->blocked = newmask & ~(1<<(SIGKILL-1)) & ~(1<<(SIGSTOP-1));
24         return old;
25 }
26
   // 检测并取得进程收到的但被屏蔽（阻塞）的信号。还未处理信号的位图将被放入 set 中。
27 int sys_sigpending(sigset_t *set)
28 {
29     /* fill in "set" with signals pending but blocked. */
       /* 用还未处理并且被阻塞信号的位图填入 set 指针所指位置处 */
   // 首先验证进程提供的用户存储空间应有 4 个字节。然后把还未处理并且被阻塞信号的位图填入
   // set 指针所指位置处。
30     verify_area(set,4);
31     put_fs_long(current->blocked & current->signal, (unsigned long *)set);
32     return 0;
33 }
34
35 /* atomically swap in the new signal mask, and wait for a signal.
36  *
37  * we need to play some games with syscall restarting.  We get help
38  * from the syscall library interface.  Note that we need to coordinate
39  * the calling convention with the libc routine.
40  *
41  * "set" is just the sigmask as described in 1003.1-1988, 3.3.7.
42  *      It is assumed that sigset_t can be passed as a 32 bit quantity.
```

```
43  *
44  * "restart" holds a restart indication.  If it's non-zero, then we
45  *      install the old mask, and return normally.  If it's zero, we store
46  *      the current mask in old_mask and block until a signal comes in.
47  */
    /* 自动地更换成新的信号屏蔽码，并等待信号的到来。
     *
     * 我们需要对系统调用（syscall）做一些处理。我们会从系统调用库接口取得某些信息。
     * 注意，我们需要把调用规则与 libc 库中的子程序统一考虑。
     *
     * "set" 正是 POSIX 标准 1003.1-1988 的 3.3.7 节中所描述的信号屏蔽码 sigmask。
     *       其中认为类型 sigset_t 能够作为一个 32 位量传递。
     *
     * "restart"中保持有重启指示标志。如果为非 0 值，那么我们就设置原来的屏蔽码，
     *       并且正常返回。如果它为 0，那么我们就把当前的屏蔽码保存在 oldmask 中
     *       并且阻塞进程，直到收到任何一个信号为止。
     */
    // 该系统调用临时把进程信号屏蔽码替换成参数中给定的 set，然后挂起进程，直到收到一个
    // 信号为止。
    // restart 是一个被中断的系统调用重新启动标志。当第 1 次调用该系统调用时，它是 0。并且
    // 在该函数中会把进程原来的阻塞码 blocked 保存起来（old_mask），并设置 restart 为非 0
    // 值。因此当进程第 2 次调用该系统调用时，它就会恢复进程原来保存在 old_mask 中的阻塞码。
48  int sys_sigsuspend(int restart, unsigned long old_mask, unsigned long set)
49  {
    // pause()系统调用将导致调用它的进程进入睡眠状态，直到收到一个信号。该信号或者会终止
    // 进程的执行，或者导致进程去执行相应的信号捕获函数。
50      extern int sys_pause(void);
51
    // 如果 restart 标志不为 0，表示是重新让程序运行起来。于是恢复前面保存在 old_mask 中的
    // 原进程阻塞码。并返回码-EINTR（系统调用被信号中断）。
52      if (restart) {
53          /* we're restarting */          /* 我们正在重新启动系统调用 */
54          current->blocked = old_mask;
55          return -EINTR;
56      }
    // 否则表示 restart 标志的值是 0。表示第 1 次调用。于是首先设置 restart 标志（置为 1），
    // 保存进程当前阻塞码 blocked 到 old_mask 中，并把进程的阻塞码替换成 set。然后调用
    // pause()让进程睡眠，等待信号的到来。当进程收到一个信号时，pause() 就会返回，并且
    // 进程会去执行信号处理函数，然后本调用返回 -ERESTARTNOINTR 码退出。这个返回码说明
    // 在处理完信号后要求返回到本系统调用中继续运行，即本系统调用不会被中断。
57      /* we're not restarting.  do the work */
        /* 我们不是重新重新运行，那么就干活吧 */
58      *(&restart) = 1;
59      *(&old_mask) = current->blocked;
60      current->blocked = set;
61      (void) sys_pause();                 /* return after a signal arrives */
62      return -ERESTARTNOINTR;             /* handle the signal, and come back */
63  }
64
    // 复制 sigaction 数据到 fs 数据段 to 处。即从内核空间复制到用户（任务）数据段中。
65  static inline void save_old(char * from,char * to)
66  {
67          int i;
```

```
68
   // 首先验证 to 处的内存空间是否足够大。然后把一个 sigaction 结构信息复制到 fs 段（用户）
   // 空间中。宏函数 put_fs_byte()在 include/asm/segment.h 中实现。
69          verify_area(to, sizeof(struct sigaction));
70          for (i=0 ; i< sizeof(struct sigaction) ; i++) {
71                  put_fs_byte(*from,to);
72                  from++;
73                  to++;
74          }
75  }
76
   // 把 sigaction 数据从 fs 数据段 from 位置复制到 to 处。即从用户数据空间取到内核数据段中。
77  static inline void get_new(char * from,char * to)
78  {
79          int i;
80
81          for (i=0 ; i< sizeof(struct sigaction) ; i++)
82                  *(to++) = get_fs_byte(from++);
83  }
84
   // signal()系统调用。类似于 sigaction()。为指定的信号安装新的信号句柄(信号处理程序)。
   // 信号句柄可以是用户指定的函数，也可以是 SIG_DFL（默认句柄）或 SIG_IGN（忽略）。
   // 参数 signum --指定的信号；handler -- 指定的句柄；restorer -恢复函数指针，该函数由
   // Libc 库提供。用于在信号处理程序结束后恢复系统调用返回时几个寄存器的原有值以及系统
   // 调用的返回值，就好象系统调用没有执行过信号处理程序而直接返回到用户程序一样。 函数
   // 返回原信号句柄。
85  int sys_signal(int signum, long handler, long restorer)
86  {
87          struct sigaction tmp;
88
   // 首先验证信号值在有效范围（1--32）内，并且不得是信号 SIGKILL（和 SIGSTOP）。因为这
   // 两个信号不能被进程捕获。
89          if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
90                  return -EINVAL;
   // 然后根据提供的参数组建 sigaction 结构内容。sa_handler 是指定的信号处理句柄（函数）。
   // sa_mask 是执行信号处理句柄时的信号屏蔽码。sa_flags 是执行时的一些标志组合。这里设定
   // 该信号处理句柄只使用 1 次后就恢复到默认值，并允许信号在自己的处理句柄中收到。
91          tmp.sa_handler = (void (*)(int)) handler;
92          tmp.sa_mask = 0;
93          tmp.sa_flags = SA_ONESHOT | SA_NOMASK;
94          tmp.sa_restorer = (void (*)(void)) restorer;        // 保存恢复处理函数指针。
   // 接着取该信号原来的处理句柄，并设置该信号的 sigaction 结构。最后返回原信号句柄。
95          handler = (long) current->sigaction[signum-1].sa_handler;
96          current->sigaction[signum-1] = tmp;
97          return handler;
98  }
99
   // sigaction()系统调用。改变进程在收到一个信号时的操作。signum 是除了 SIGKILL 以外的
   // 任何信号。[如果新操作（action）不为空 ]则新操作被安装。如果 oldaction 指针不为空，
   // 则原操作被保留到 oldaction。成功则返回 0，否则为-EINVAL。
100 int sys_sigaction(int signum, const struct sigaction * action,
101         struct sigaction * oldaction)
102 {
```

```
103         struct sigaction tmp;
104
    // 首先验证信号值在有效范围（1--32）内，并且不得是信号 SIGKILL（和 SIGSTOP）。因为这
    // 两个信号不能被进程捕获。
105         if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
106                 return -EINVAL;
    // 在信号的 sigaction 结构中设置新的操作（动作）。如果 oldaction 指针不为空的话，则将
    // 原操作指针保存到 oldaction 所指的位置。
107         tmp = current->sigaction[signum-1];
108         get_new((char *) action,
109                 (char *) (signum-1+current->sigaction));
110         if (oldaction)
111                 save_old((char *) &tmp,(char *) oldaction);
    // 如果允许信号在自己的信号句柄中收到，则令屏蔽码为 0，否则设置屏蔽本信号。
112         if (current->sigaction[signum-1].sa_flags & SA_NOMASK)
113                 current->sigaction[signum-1].sa_mask = 0;
114         else
115                 current->sigaction[signum-1].sa_mask |= (1<<(signum-1));
116         return 0;
117 }
118
119 /*
120  * Routine writes a core dump image in the current directory.
121  * Currently not implemented.
122  */
    /*
     * 在当前目录中产生 core dump 映像文件的子程序。目前还没有实现。
     */
123 int core_dump(long signr)
124 {
125         return(0);      /* We didn't do a dump */
126 }
127
    // 系统调用的中断处理程序中真正的信号预处理程序（在 kernel/sys_call.s,119 行）。这段
    // 代码的主要作用是将信号处理句柄插入到用户程序堆栈中，并在本系统调用结束返回后立刻
    // 执行信号句柄程序，然后继续执行用户的程序。
    // 函数的参数是进入系统调用处理程序 sys_call.s 开始，直到调用本函数（sys_call.s
    // 第 125 行）前逐步压入堆栈的值。这些值包括（在 sys_call.s 中的代码行）：
    // ① CPU 执行中断指令压入的用户栈地址 ss 和 esp、标志寄存器 eflags 和返回地址 cs 和 eip；
    // ② 第 85--91 行在刚进入 system_call 时压入栈的段寄存器 ds、es、fs 以及寄存器 eax
    //  （orig_eax）、edx、ecx 和 ebx 的值；
    // ③ 第 100 行调用 sys_call_table 后压入栈中的相应系统调用处理函数的返回值（eax）。
    // ④ 第 124 行压入栈中的当前处理的信号值（signr）。
128 int do_signal(long signr,long eax,long ebx, long ecx, long edx, long orig_eax,
129         long fs, long es, long ds,
130         long eip, long cs, long eflags,
131         unsigned long * esp, long ss)
132 {
133         unsigned long sa_handler;
134         long old_eip=eip;
135         struct sigaction * sa = current->sigaction + signr - 1;
136         int longs;                                      // 即 current->sigaction[signr-1]。
137
```

```
138         unsigned long * tmp_esp;
139
    // 以下是调试语句。当定义了 notdef 时会打印相关信息。
140 #ifdef notdef
141         printk("pid: %d, signr: %x, eax=%d, oeax = %d, int=%d\n",
142                 current->pid, signr, eax, orig_eax,
143                 sa->sa_flags & SA_INTERRUPT);
144 #endif
    // 如果不是系统调用而是其它中断执行过程中调用到本函数时， roig_eax 值为 -1。 参见
    // sys_call.s 第 144 行 等语句。因此当 orig_eax 不等于 -1 时，说明是在某个系统调用的
    // 最后调用了本函数。在 kernel/exit.c 的 waitpid() 函数中，如果收到了 SIGCHLD 信号，
    // 或者在读管道函数 fs/pipe.c 中管道当前读数据但没有读到任何数据等情况下，进程收到
    // 了任何一个非阻塞的信号，则都会以 -ERESTARTSYS 返回值返回。它表示进程可以被中断，
    // 但是在继续执行后会重新启动系统调用。返回码-ERESTARTNOINTR 说明在处理完信号后要求
    // 返回到原系统调用中继续运行，即系统调用不会被中断。参见前面第 62 行。
    // 因此下面语句说明如果是在系统调用中调用的本函数，并且相应系统调用的返回码 eax 等于
    // -ERESTARTSYS 或 -ERESTARTNOINTR 时进行下面的处理（实际上还没有真正回到用户程序中）。
145         if ((orig_eax != -1) &&
146             ((eax == -ERESTARTSYS) || (eax == -ERESTARTNOINTR))) {
    // 如果系统调用返回码是 -ERESTARTSYS（重新启动系统调用），并且 sigaction 中含有标志
    // SA_INTERRUPT（系统调用被信号中断后不重新启动系统调用）或者信号值小于 SIGCONT 或者
    // 信号值大于 SIGTTOU（即信号不是 SIGCONT、SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU），则
    // 修改系统调用的返回值为 eax = -EINTR，即被信号中断的系统调用。
147                 if ((eax == -ERESTARTSYS) && ((sa->sa_flags & SA_INTERRUPT) ||
148                     signr < SIGCONT || signr > SIGTTOU))
149                         *(&eax) = -EINTR;
150                 else {
    // 否则就恢复进程寄存器 eax 在调用系统调用之前的值，并且把原程序指令指针回调 2 字节。即
    // 当返回用户程序时，让程序重新启动执行被信号中断的系统调用。
151                         *(&eax) = orig_eax;
152                         *(&eip) = old_eip -= 2;
153                 }
154         }
    // 如果信号句柄为 SIG_IGN（1，默认忽略句柄）则不对信号进行处理而直接返回。
155         sa_handler = (unsigned long) sa->sa_handler;
156         if (sa_handler==1)
157                 return(1);   /* Ignore, see if there are more signals... */
    // 如果句柄为 SIG_DFL（0，默认处理），则根据具体的信号进行分别处理。
158         if (!sa_handler) {
159                 switch (signr) {
    // 如果信号是以下两个则也忽略之，并返回。
160                 case SIGCONT:
161                 case SIGCHLD:
162                         return(1);  /* Ignore, ... */
163
    // 如果信号是以下 4 种信号之一，则把当前进程状态置为停止状态 TASK_STOPPED。若当前进程
    // 父进程对 SIGCHLD 信号的 sigaction 处理标志 SA_NOCLDSTOP （即当子进程停止执行或又继
    // 续执行时不要产生 SIGCHLD 信号）没有置位，那么就给父进程发送 SIGCHLD 信号。
164                 case SIGSTOP:
165                 case SIGTSTP:
166                 case SIGTTIN:
167                 case SIGTTOU:
168                         current->state = TASK_STOPPED;
```

```
169                         current->exit_code = signr;
170                         if (!(current->p_pptr->sigaction[SIGCHLD-1].sa_flags &
171                                         SA_NOCLDSTOP))
172                                 current->p_pptr->signal |= (1<<(SIGCHLD-1));
173                         return(1);  /* Reschedule another event */
174
    // 如果信号是以下 6 种信号之一，那么若信号产生了 core dump，则以退出码为 signr|0x80
    // 调用 do_exit()退出。否则退出码就是信号值。do_exit()的参数是返回码和程序提供的退出
    // 状态信息。可作为 wait()或 waitpid()函数的状态信息。参见 sys/wait.h 文件第 13-18 行。
    // wait()或 waitpid()利用这些宏就可以取得子进程的退出状态码或子进程终止的原因（信号）。
175                 case SIGQUIT:
176                 case SIGILL:
177                 case SIGTRAP:
178                 case SIGIOT:
179                 case SIGFPE:
180                 case SIGSEGV:
181                         if (core_dump(signr))
182                                 do_exit(signr|0x80);
183                         /* fall through */
184                 default:
185                         do_exit(signr);
186                 }
187         }
188         /*
189          * OK, we're invoking a handler
190          */
            /*
             * OK，现在我们准备对信号句柄调用的设置
             */
    // 如果该信号句柄只需被调用一次，则将该句柄置空。注意，该信号句柄在前面已经保存在
    // sa_handler 指针中。
    // 在系统调用进入内核时，用户程序返回地址（eip、cs）被保存在内核态栈中。下面这段代
    // 码修改内核态堆栈上用户调用系统调用时的代码指针 eip 为指向信号处理句柄，同时也将
    // sa_restorer、signr、进程屏蔽码(如果 SA_NOMASK 没置位)、eax、ecx、edx 作为参数以及
    // 原调用系统调用的程序返回指针及标志寄存器值压入用户堆栈。 因此在本次系统调用中断
    // 返回用户程序时会首先执行用户的信号句柄程序，然后再继续执行用户程序。
191         if (sa->sa_flags & SA_ONESHOT)
192                 sa->sa_handler = NULL;
    // 将内核态栈上用户调用系统调用下一条代码指令指针 eip 指向该信号处理句柄。由于 C 函数
    // 是传值函数，因此给 eip 赋值时需要使用 "*(&eip)" 的形式。另外，如果允许信号自己的
    // 处理句柄收到信号自己，则也需要将进程的阻塞码压入堆栈。
    // 这里请注意，使用如下方式（第 193 行）对普通 C 函数参数进行修改是不起作用的。因为当
    // 函数返回时堆栈上的参数将会被调用者丢弃。这里之所以可以使用这种方式，是因为该函数
    // 是从汇编程序中被调用的，并且在函数返回后汇编程序并没有把调用 do_signal()时的所有
    // 参数都丢弃。eip 等仍然在堆栈中。
    // sigaction 结构的 sa_mask 字段给出了在当前信号句柄（信号描述符）程序执行期间应该被
    // 屏蔽的信号集。同时，引起本信号句柄执行的信号也会被屏蔽。 不过若 sa_flags 中使用了
    // SA_NOMASK 标志，那么引起本信号句柄执行的信号将不会被屏蔽掉。如果允许信号自己的处
    // 理句柄程序收到信号自己，则也需要将进程的信号阻塞码压入堆栈。
193         *(&eip) = sa_handler;
194         longs = (sa->sa_flags & SA_NOMASK)?7:8;
    // 将原调用程序的用户堆栈指针向下扩展 7（或 8）个长字（用来存放调用信号句柄的参数等），
    // 并检查内存使用情况（例如如果内存超界则分配新页等）。
```

```
195         *(&esp) -= longs;
196         verify_area(esp,longs*4);
    // 在用户堆栈中从下到上存放 sa_restorer、信号 signr、屏蔽码 blocked（如果 SA_NOMASK
    // 置位）、eax、ecx、edx、eflags 和用户程序原代码指针。
197         tmp_esp=esp;
198         put_fs_long((long) sa->sa_restorer,tmp_esp++);
199         put_fs_long(signr,tmp_esp++);
200         if (!(sa->sa_flags & SA_NOMASK))
201                 put_fs_long(current->blocked,tmp_esp++);
202         put_fs_long(eax,tmp_esp++);
203         put_fs_long(ecx,tmp_esp++);
204         put_fs_long(edx,tmp_esp++);
205         put_fs_long(eflags,tmp_esp++);
206         put_fs_long(old_eip,tmp_esp++);
207         current->blocked |= sa->sa_mask;   // 进程阻塞码(屏蔽码)添上 sa_mask 中的码位。
208         return(0);              /* Continue, execute handler */
209 }
210
```

8.7 程序 8-7 linux/kernel/exit.c

```
 1 /*
 2  *  linux/kernel/exit.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #define DEBUG_PROC_TREE   // 定义符号“调试进程树”。
 8
 9 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)
10 #include <signal.h>       // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。
11 #include <sys/wait.h>     // 等待调用头文件。定义系统调用 wait()和 waitpid()及相关常数符号。
12
13 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
14 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
15 #include <linux/tty.h>    // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
16 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
17
18 int sys_pause(void);      // 把进程置为睡眠状态，直到收到信号（kernel/sched.c，164 行）。
19 int sys_close(int fd);    // 关闭指定文件的系统调用（fs/open.c，219 行）。
20
//// 释放指定进程占用的任务槽及其任务数据结构占用的内存页面。
// 参数 p 是任务数据结构指针。该函数在后面的 sys_kill() 和 sys_waitpid() 函数中被调用。
// 扫描任务指针数组表 task[] 以寻找指定的任务。如果找到，则首先清空该任务槽，然后释放
// 该任务数据结构所占用的内存页面，最后执行调度函数并在返回时立即退出。如果在任务数组
// 表中没有找到指定任务对应的项，则内核 panic☺。
21 void release(struct task_struct * p)
22 {
23         int i;
24
// 如果给定的任务结构指针为 NULL 则退出。如果该指针指向当前进程则显示警告信息退出。
25         if (!p)
26                 return;
27         if (p == current) {
28                 printk("task releasing itself\n\r");
29                 return;
30         }
// 扫描任务结构指针数组，寻找指定的任务 p。如果找到，则置空任务指针数组中对应项，并且
// 更新任务结构之间的关联指针，释放任务 p 数据结构占用的内存页面。最后在执行调度程序
// 返回后退出。如果没有找到指定的任务 p，则说明内核代码出错了，则显示出错信息并死机。
// 更新链接部分的代码会把指定任务 p 从双向链表中删除。
31         for (i=1 ; i<NR_TASKS ; i++)
32                 if (task[i]==p) {
33                         task[i]=NULL;
34                         /* Update links */    /* 更新链接 */
// 如果 p 不是最后（最老）的子进程，则让比其老的比邻进程指向比它新的比邻进程。如果 p
// 不是最新的子进程，则让比其新的比邻子进程指向比邻的老进程。 如果任务 p 就是最新的
// 子进程，则还需要更新其父进程的最新子进程指针 cptr 为指向 p 的比邻子进程。
// 指针 osptr（old sibling pointer）指向比 p 先创建的兄弟进程。
// 指针 ysptr（younger sibling pointer）指向比 p 后创建的兄弟进程。
```

```
   // 指针 pptr（parent pointer）指向 p 的父进程。
   // 指针 cptr（child pointer）是父进程指向最新（最后）创建的子进程。
35                          if (p->p_osptr)
36                                  p->p_osptr->p_ysptr = p->p_ysptr;
37                          if (p->p_ysptr)
38                                  p->p_ysptr->p_osptr = p->p_osptr;
39                          else
40                                  p->p_pptr->p_cptr = p->p_osptr;
41                          free_page((long)p);
42                          schedule();
43                          return;
44                  }
45          panic("trying to release non-existent task");
46  }
47
48  #ifdef DEBUG_PROC_TREE
   // 如果定义了符号 DEBUG_PROC_TREE，则编译时包括以下代码。
49  /*
50   * Check to see if a task_struct pointer is present in the task[] array
51   * Return 0 if found, and 1 if not found.
52   */
   /*
    * 检查 task[]数组中是否存在一个指定的 task_struct 结构指针 p。
    * 如果存在则返回 0，否则返回 1。
    */
   // 检测任务结构指针 p。
53  int bad_task_ptr(struct task_struct *p)
54  {
55          int     i;
56
57          if (!p)
58                  return 0;
59          for (i=0 ; i<NR_TASKS ; i++)
60                  if (task[i] == p)
61                          return 0;
62          return 1;
63  }
64
65  /*
66   * This routine scans the pid tree and make sure the rep invarient still
67   * holds.  Used for debugging only, since it's very slow....
68   *
69   * It looks a lot scarier than it really is.... we're doing nothing more
70   * than verifying the doubly-linked list found in p_ysptr and p_osptr,
71   * and checking it corresponds with the process tree defined by p_cptr and
72   * p_pptr;
73   */
   /*
    * 下面的函数用于扫描进程树，以确定更改过的链接仍然正确。仅用于调式，
    * 因为该函数比较慢....
    *
    * 该函数看上去要比实际的恐怖.... 其实我们仅仅验证了指针 p_ysptr 和
    * p_osptr 构成的双向链表，并检查了链表与指针 p_cptr 和 p_pptr 构成的
```

```
 * 进程树之间的关系。
 */
// 检查进程树。
74 void audit_ptree()
75 {
76         int     i;
77
// 扫描系统中的除任务 0 以外的所有任务，检查它们中 4 个指针（pptr、cptr、ysptr 和 osptr）
// 的正确性。若任务数组槽（项）为空则跳过。
78         for (i=1 ; i<NR_TASKS ; i++) {
79                 if (!task[i])
80                         continue;
// 如果任务的父进程指针 p_pptr 没有指向任何进程（即在任务数组中不存在），则显示警告信息
// “警告，pid 号 N 的父进程链接有问题”。以下语句对 cptr、ysptr 和 osptr 进行类似操作。
81                 if (bad_task_ptr(task[i]->p_pptr))
82                         printk("Warning, pid %d's parent link is bad\n",
83                                 task[i]->pid);
84                 if (bad_task_ptr(task[i]->p_cptr))
85                         printk("Warning, pid %d's child link is bad\n",
86                                 task[i]->pid);
87                 if (bad_task_ptr(task[i]->p_ysptr))
88                         printk("Warning, pid %d's ys link is bad\n",
89                                 task[i]->pid);
90                 if (bad_task_ptr(task[i]->p_osptr))
91                         printk("Warning, pid %d's os link is bad\n",
92                                 task[i]->pid);
// 如果任务的父进程指针 p_pptr 指向了自己，则显示警告信息“警告，pid 号 N 的父进程链接
// 指针指向了自己”。以下语句对 cptr、ysptr 和 osptr 进行类似操作。
93                 if (task[i]->p_pptr == task[i])
94                         printk("Warning, pid %d parent link points to self\n");
95                 if (task[i]->p_cptr == task[i])
96                         printk("Warning, pid %d child link points to self\n");
97                 if (task[i]->p_ysptr == task[i])
98                         printk("Warning, pid %d ys link points to self\n");
99                 if (task[i]->p_osptr == task[i])
100                        printk("Warning, pid %d os link points to self\n");
// 如果任务有比自己先创建的比邻兄弟进程，那么就检查它们是否有共同的父进程，并检查这个
// 老兄进程的 ysptr 指针是否正确地指向本进程。否则显示警告信息。
101                if (task[i]->p_osptr) {
102                        if (task[i]->p_pptr != task[i]->p_osptr->p_pptr)
103                                printk(
104                        "Warning, pid %d older sibling %d parent is %d\n",
105                                task[i]->pid, task[i]->p_osptr->pid,
106                                task[i]->p_osptr->p_pptr->pid);
107                        if (task[i]->p_osptr->p_ysptr != task[i])
108                                printk(
109                "Warning, pid %d older sibling %d has mismatched ys link\n",
110                                task[i]->pid, task[i]->p_osptr->pid);
111                }
// 如果任务有比自己后创建的比邻兄弟进程，那么就检查它们是否有共同的父进程，并检查这个
// 小弟进程的 osptr 指针是否正确地指向本进程。否则显示警告信息。
112                if (task[i]->p_ysptr) {
113                        if (task[i]->p_pptr != task[i]->p_ysptr->p_pptr)
```

```
114                             printk(
115                         "Warning, pid %d younger sibling %d parent is %d\n",
116                             task[i]->pid, task[i]->p_osptr->pid,
117                             task[i]->p_osptr->p_pptr->pid);
118                         if (task[i]->p_ysptr->p_osptr != task[i])
119                             printk(
120                 "Warning, pid %d younger sibling %d has mismatched os link\n",
121                             task[i]->pid, task[i]->p_ysptr->pid);
122                 }
    // 如果任务的最新子进程指针 cptr 不空，那么检查该子进程的父进程是否是本进程，并检查该
    // 子进程的小弟进程指针 yspter 是否为空。若不是则显示警告信息。
123                 if (task[i]->p_cptr) {
124                         if (task[i]->p_cptr->p_pptr != task[i])
125                             printk(
126                         "Warning, pid %d youngest child %d has mismatched parent link\n",
127                             task[i]->pid, task[i]->p_cptr->pid);
128                         if (task[i]->p_cptr->p_ysptr)
129                             printk(
130                         "Warning, pid %d youngest child %d has non-NULL ys link\n",
131                             task[i]->pid, task[i]->p_cptr->pid);
132                 }
133         }
134 }
135 #endif /* DEBUG_PROC_TREE */
136
    //// 向指定任务 p 发送信号 sig，权限为 priv。
    // 参数：sig - 信号值；p - 指定任务的指针；priv - 强制发送信号的标志。即不需要考虑进程
    // 用户属性或级别而能发送信号的权利。该函数首先判断参数的正确性，然后判断条件是否满足。
    // 如果满足就向指定进程发送信号 sig 并退出，否则返回未许可错误号。
137 static inline int send_sig(long sig,struct task_struct * p,int priv)
138 {
    // 如果没有权限，并且当前进程的有效用户 ID 与进程 p 的不同，并且也不是超级用户，则说明
    // 没有向 p 发送信号的权利。suser()定义为(current->euid==0)，用于判断是否是超级用户。
139         if (!p)
140                 return -EINVAL;
141         if (!priv && (current->euid!=p->euid) && !suser())
142                 return -EPERM;
    // 若需要发送的信号是 SIGKILL 或 SIGCONT，那么如果此时接收信号的进程 p 正处于停止状态
    // 就置其为就绪（运行）状态。然后修改进程 p 的信号位图 signal，去掉（复位）会导致进程
    // 停止的信号 SIGSTOP、SIGTSTP、SIGTTIN 和 SIGTTOU。
143         if ((sig == SIGKILL) || (sig == SIGCONT)) {
144                 if (p->state == TASK_STOPPED)
145                         p->state = TASK_RUNNING;
146                 p->exit_code = 0;
147                 p->signal &= ~( (1<<(SIGSTOP-1)) | (1<<(SIGTSTP-1)) |
148                                 (1<<(SIGTTIN-1)) | (1<<(SIGTTOU-1)) );
149         }
150         /* If the signal will be ignored, don't even post it */
        /* 如果要发送的信号 sig 将被进程 p 忽略掉，那么就根本不用发送 */
151         if ((int) p->sigaction[sig-1].sa_handler == 1)
152                 return 0;
153         /* Depends on order SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU */
        /* 以下判断依赖于 SIGSTOP、SIGTSTP、SIGTTIN 和 SIGTTOU 的次序 */
```

```
    // 如果信号是 SIGSTOP、SIGTSTP、SIGTTIN 和 SIGTTOU 之一，那么说明要让接收信号的进程 p
    // 停止运行。因此（若 p 的信号位图中有 SIGCONT 置位）就需要复位位图中继续运行的信号
    // SIGCONT 比特位。
154         if ((sig >= SIGSTOP) && (sig <= SIGTTOU))
155                 p->signal &= ~(1<<(SIGCONT-1));
156         /* Actually deliver the signal */
            /* 最后，我们向进程 p 发送信号 p */
157         p->signal |= (1<<(sig-1));
158         return 0;
159 }
160
    // 根据进程组号 pgrp 取得进程组所属的会话号。
    // 扫描任务数组，寻找进程组号为 pgrp 的进程，并返回其会话号。如果没有找到指定进程组号
    // 为 pgrp 的任何进程，则返回-1。
161 int session_of_pgrp(int pgrp)
162 {
163         struct task_struct **p;
164
165         for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
166                 if ((*p)->pgrp == pgrp)
167                         return((*p)->session);
168         return -1;
169 }
170
    // 终止进程组（向进程组发送信号）。
    // 参数：pgrp - 指定的进程组号；sig - 指定的信号；priv - 权限。
    // 即向指定进程组 pgrp 中的每个进程发送指定信号 sig。只要向一个进程发送成功最后就会
    // 返回 0，否则如果没有找到指定进程组号 pgrp 的任何一个进程，则返回出错号-ESRCH，若
    // 找到进程组号是 pgrp 的进程，但是发送信号失败，则返回发送失败的错误码。
171 int kill_pg(int pgrp, int sig, int priv)
172 {
173         struct task_struct **p;
174         int err,retval = -ESRCH;         // -ESRCH 表示指定的进程不存在。
175         int found = 0;
176
    // 首先判断给定的信号和进程组号是否有效。然后扫描系统中所有任务。若扫描到进程组号为
    // pgrp 的进程，就向其发送信号 sig。只要有一次信号发送成功，函数最后就会返回 0。
177         if (sig<1 || sig>32 || pgrp<=0)
178                 return -EINVAL;
179         for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
180                 if ((*p)->pgrp == pgrp) {
181                         if (sig && (err = send_sig(sig,*p,priv)))
182                                 retval = err;
183                         else
184                                 found++;
185                 }
186         return(found ? 0 : retval);
187 }
188
    // 终止进程（向进程发送信号）。
    // 参数：pid - 进程号；sig - 指定信号；priv - 权限。
    // 即向进程号为 pid 的进程发送指定信号 sig。若找到指定 pid 的进程，那么若信号发送成功，
    // 则返回 0，否则返回信号发送出错号。如果没有找到指定进程号 pid 的进程，则返回出错号
```

```
    // -ESRCH（指定进程不存在）。
189 int kill_proc(int pid, int sig, int priv)
190 {
191         struct task_struct **p;
192
193         if (sig<1 || sig>32)
194                 return -EINVAL;
195         for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
196                 if ((*p)->pid == pid)
197                         return(sig ? send_sig(sig,*p,priv) : 0);
198         return(-ESRCH);
199 }
200
201 /*
202  * POSIX specifies that kill(-1,sig) is unspecified, but what we have
203  * is probably wrong.  Should make it like BSD or SYSV.
204  */
    /*
     * POSIX 标准指明 kill(-1,sig)未定义。但是我所知道的可能错了。应该让它
     * 象 BSD 或 SYSV 系统一样。
     */
    //// 系统调用 kill()可用于向任何进程或进程组发送任何信号，而并非只是杀死进程☺。
    // 参数 pid 是进程号；sig 是需要发送的信号。
    // 如果 pid 值>0，则信号被发送给进程号是 pid 的进程。
    // 如果 pid=0，那么信号就会被发送给当前进程的进程组中所有的进程。
    // 如果 pid=-1，则信号 sig 就会发送给除第一个进程（初始进程）外的所有进程。
    // 如果 pid < -1，则信号 sig 将发送给进程组-pid 的所有进程。
    // 如果信号 sig 为 0，则不发送信号，但仍会进行错误检查。如果成功则返回 0。
    // 该函数扫描任务数组表，并根据 pid 对满足条件的进程发送指定信号 sig。若 pid 等于 0，
    // 表明当前进程是进程组组长，因此需要向所有组内的进程强制发送信号 sig。
205 int sys_kill(int pid,int sig)
206 {
207         struct task_struct **p = NR_TASKS + task;    // p 指向任务数组最后一项。
208         int err, retval = 0;
209
210         if (!pid)
211                 return(kill_pg(current->pid,sig,0));
212         if (pid == -1) {
213                 while (--p > &FIRST_TASK)
214                         if (err = send_sig(sig,*p,0))
215                                 retval = err;
216                 return(retval);
217         }
218         if (pid < 0)
219                 return(kill_pg(-pid,sig,0));
220         /* Normal kill */
221         return(kill_proc(pid,sig,0));
222 }
223
224 /*
225  * Determine if a process group is "orphaned", according to the POSIX
226  * definition in 2.2.2.52.  Orphaned process groups are not to be affected
227  * by terminal-generated stop signals.  Newly orphaned process groups are
```

```
228  * to receive a SIGHUP and a SIGCONT.
229  *
230  * "I ask you, have you ever known what it is to be an orphan?"
231  */
     /*
      * 根据 POSIX 标准 2.2.2.52 节中的定义，确定一个进程组是否是“孤儿”。孤儿进程
      * 组不会受到终端产生的停止信号的影响。新近产生的孤儿进程组将会收到一个 SIGHUP
      * 信号和一个 SIGCONT 信号。
      *
      * “我问你，你是否真正知道作为一个孤儿意味着什么？”
      */
     // 以上提到的 POSIX P1003.1 2.2.2.52 节是关于孤儿进程组的描述。在两种情况下当一个进程
     // 终止时可能导致进程组变成“孤儿”。  一个进程组到其组外的父进程之间的联系依赖于该父
     // 进程和其子进程两者。因此，若组外最后一个连接父进程的进程或最后一个父进程的直接后裔
     // 终止的话，那么这个进程组就会成为一个孤儿进程组。在任何一种情况下，如果进程的终止导
     // 致进程组变成孤儿进程组，那么进程组中的所有进程就会与它们的作业控制 shell 断开联系。
     // 作业控制 shell 将不再具有该进程组存在的任何信息。而该进程组中处于停止状态的进程将会
     // 永远消失。为了解决这个问题，含有停止状态进程的新近产生的孤儿进程组就需要接收到一个
     // SIGHUP 信号和一个 SIGCONT 信号，用于指示它们已经从它们的会话（ session）中断开联系。
     // SIGHUP 信号将导致进程组中成员被终止，除非它们捕获或忽略了 SIGHUP 信号。而 SIGCONT 信
     // 号将使那些没有被 SIGHUP 信号终止的进程继续运行。 但在大多数情况下，如果组中有一个进
     // 程处于停止状态，那么组中所有的进程可能都处于停止状态。
     //
     // 判断一个进程组是否是孤儿进程。如果不是则返回 0；如果是则返回 1。
     // 扫描任务数组。如果任务项空，或者进程的组号与指定的不同，或者进程已经处于僵死状态，
     // 或者进程的父进程是 init 进程，则说明扫描的进程不是指定进程组的成员，或者不满足要求，
     // 于是跳过。 否则说明该进程是指定组的成员并且其父进程不是 init 进程。此时如果该进程
     // 父进程的组号不等于指定的组号 pgrp，但父进程的会话号等于进程的会话号，则说明它们同
     // 属于一个会话。因此指定的 pgrp 进程组肯定不是孤儿进程组。否则...。
232  int is_orphaned_pgrp(int pgrp)
233  {
234          struct task_struct **p;
235
236          for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
237                  if (!(*p) ||
238                      ((*p)->pgrp != pgrp) ||
239                      ((*p)->state == TASK_ZOMBIE) ||
240                      ((*p)->p_pptr->pid == 1))
241                          continue;
242                  if (((*p)->p_pptr->pgrp != pgrp) &&
243                      ((*p)->p_pptr->session == (*p)->session))
244                          return 0;
245          }
246          return(1);      /* (sighing) "Often!" */    /* （唉）是孤儿进程组！ */
247  }
248
     // 判断进程组中是否含有处于停止状态的作业（进程组）。有则返回 1；无则返回 0。
     // 查找方法是扫描整个任务数组。检查属于指定组 pgrp 的任何进程是否处于停止状态。
249  static int has_stopped_jobs(int pgrp)
250  {
251          struct task_struct ** p;
252
253          for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
```

```
254                 if ((*p)->pgrp != pgrp)
255                         continue;
256                 if ((*p)->state == TASK_STOPPED)
257                         return(1);
258         }
259         return(0);
260 }
261
    // 程序退出处理函数。在下面 365 行处被系统调用处理函数 sys_exit()调用。
    // 该函数将根据当前进程自身的特性对其进行处理，并把当前进程状态设置成僵死状态
    // TASK_ZOMBIE，最后调用调度函数 schedule()去执行其它进程，不再返回。
262 volatile void do_exit(long code)
263 {
264         struct task_struct *p;
265         int i;
266
    // 首先释放当前进程代码段和数据段所占的内存页。 函数 free_page_tables() 的第 1 个参数
    // （get_base()返回值）指明在 CPU 线性地址空间中起始基地址，第 2 个（get_limit()返回值）
    // 说明欲释放的字节长度值。get_base()宏中的 current->ldt[1]给出进程代码段描述符的位置
    // （current->ldt[2]给出进程代码段描述符的位置）；get_limit()中的 0x0f 是进程代码段的
    // 选择符（0x17 是进程数据段的选择符）。即在取段基地址时使用该段的描述符所处地址作为
    // 参数，取段长度时使用该段的选择符作为参数。 free_page_tables()函数位于 mm/memory.c
    // 文件的第 69 行开始处；get_base()和 get_limit()宏位于 include/linux/sched.h 头文件的第
    // 264 行开始处。
267         free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
268         free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
    // 然后关闭当前进程打开着的所有文件。再对当前进程的工作目录 pwd、根目录 root、执行程序
    // 文件的 i 节点以及库文件进行同步操作，放回各个 i 节点并分别置空（释放）。 接着把当前
    // 进程的状态设置为僵死状态（TASK_ZOMBIE），并设置进程退出码。
269         for (i=0 ; i<NR_OPEN ; i++)
270                 if (current->filp[i])
271                         sys_close(i);
272         iput(current->pwd);
273         current->pwd = NULL;
274         iput(current->root);
275         current->root = NULL;
276         iput(current->executable);
277         current->executable = NULL;
278         iput(current->library);
279         current->library = NULL;
280         current->state = TASK_ZOMBIE;
281         current->exit_code = code;
282         /*
283          * Check to see if any process groups have become orphaned
284          * as a result of our exiting, and if they have any stopped
285          * jobs, send them a SIGUP and then a SIGCONT.  (POSIX 3.2.2.2)
286          *
287          * Case i: Our father is in a different pgrp than we are
288          * and we were the only connection outside, so our pgrp
289          * is about to become orphaned.
290          */
             /*
              * 检查当前进程的退出是否会造成任何进程组变成孤儿进程组。如果
```

```
            * 有，并且有处于停止状态（TASK_STOPPED）的组员，则向它们发送
            * 一个 SIGHUP 信号和一个 SIGCONT 信号。（POSIX 3.2.2.2 节要求）
            *
            * 情况 1：我们的父进程在另外一个与我们不同的进程组中，而本进程
            * 是我们与外界的唯一联系。所以我们的进程组将变成一个孤儿进程组。
            */
    // POSIX 3.2.2.2（1991 版）是关于 exit()函数的说明。如果父进程所在的进程组与当前进程的
    // 不同，但都处于同一个会话（session）中，并且当前进程所在进程组将要变成孤儿进程了并且
    // 当前进程的进程组中含有处于停止状态的作业（进程），那么就要向这个当前进程的进程组发
    // 送两个信号：SIGHUP 和 SIGCONT。发送这两个信号的原因见 232 行前的说明。
291         if ((current->p_pptr->pgrp != current->pgrp) &&
292             (current->p_pptr->session == current->session) &&
293             is_orphaned_pgrp(current->pgrp) &&
294             has_stopped_jobs(current->pgrp)) {
295                 kill_pg(current->pgrp,SIGHUP,1);
296                 kill_pg(current->pgrp,SIGCONT,1);
297         }
298         /* Let father know we died */         /* 通知父进程当前进程将终止 */
299         current->p_pptr->signal |= (1<<(SIGCHLD-1));
300
301         /*
302          * This loop does two things:
303          *
304          * A.  Make init inherit all the child processes
305          * B.  Check to see if any process groups have become orphaned
306          *      as a result of our exiting, and if they have any stopped
307          *      jons, send them a SIGUP and then a SIGCONT.  (POSIX 3.2.2.2)
308          */
            /*
             * 下面的循环做了两件事情：
             *
             * A.   让 init 进程继承当前进程所有子进程。
             * B.   检查当前进程的退出是否会造成任何进程组变成孤儿进程组。如果
             *      有，并且有处于停止状态（TASK_STOPPED）的组员，则向它们发送
             *      一个 SIGHUP 信号和一个 SIGCONT 信号。（POSIX 3.2.2.2 节要求）
             */
    // 如果当前进程有子进程（其 p_cptr 指针指向最近创建的子进程），则让进程 1（init 进程）
    // 成为其所有子进程的父进程。如果子进程已经处于僵死状态，则向 init 进程（父进程）发送
    // 子进程已终止信号 SIGCHLD。
309         if (p = current->p_cptr) {
310                 while (1) {
311                         p->p_pptr = task[1];
312                         if (p->state == TASK_ZOMBIE)
313                                 task[1]->signal |= (1<<(SIGCHLD-1));
314                         /*
315                          * process group orphan check
316                          * Case ii: Our child is in a different pgrp
317                          * than we are, and it was the only connection
318                          * outside, so the child pgrp is now orphaned.
319                          */
                            /* 孤儿进程组检测。
                             * 情况 2：我们的子进程在不同的进程组中，而本进程
                             * 是它们唯一与外界的连接。因此现在子进程所在进程
```

```
                         * 组将变成孤儿进程组了。
                         */
    // 如果子进程与当前进程不在同一个进程组中但属于同一个 session 中，并且当前进程所在进程
    // 组将要变成孤儿进程了，并且当前进程的进程组中含有处于停止状态的作业（进程），那么就
    // 要向这个当前进程的进程组发送两个信号：SIGHUP 和 SIGCONT。  如果该子进程有兄弟进程，
    // 则继续循环处理这些兄弟进程。
320                     if ((p->pgrp != current->pgrp) &&
321                         (p->session == current->session) &&
322                         is_orphaned_pgrp(p->pgrp) &&
323                         has_stopped_jobs(p->pgrp)) {
324                             kill_pg(p->pgrp,SIGHUP,1);
325                             kill_pg(p->pgrp,SIGCONT,1);
326                     }
327                     if (p->p_osptr) {
328                             p = p->p_osptr;
329                             continue;
330                     }
331                     /*
332                      * This is it; link everything into init's children
333                      * and leave
334                      */
                     /*
                      * 就这样：将所有子进程链接成为 init 的子进程并退出循环。
                      */
    // 通过上面处理，当前进程子进程的所有兄弟子进程都已经处理过。此时 p 指向最老的兄弟子
    // 进程。于是把这些兄弟子进程全部加入 init 进程的子进程双向链表头部中。加入后，init
    // 进程的 p_cptr 指向当前进程原子进程中最年轻的（the youngest）子进程，而原子进程中
    // 最老的（the oldest）兄弟子进程 p_osptr 指向原 init 进程的最年轻进程，而原 init 进
    // 程中最年轻进程的 p_ysptr 指向原子进程中最老的兄弟子进程。最后把当前进程的 p_cptr
    // 指针置空，并退出循环。
335                     p->p_osptr = task[1]->p_cptr;
336                     task[1]->p_cptr->p_ysptr = p;
337                     task[1]->p_cptr = current->p_cptr;
338                     current->p_cptr = 0;
339                     break;
340             }
341     }
    // 如果当前进程是会话头领(leader)进程，那么若它有控制终端，则首先向使用该控制终端的
    // 进程组发送挂断信号 SIGHUP，然后释放该终端。接着扫描任务数组，把属于当前进程会话中
    // 进程的终端置空（取消）。
342     if (current->leader) {
343             struct task_struct **p;
344             struct tty_struct *tty;
345
346             if (current->tty >= 0) {
347                     tty = TTY_TABLE(current->tty);
348                     if (tty->pgrp>0)
349                             kill_pg(tty->pgrp, SIGHUP, 1);
350                     tty->pgrp = 0;
351                     tty->session = 0;
352             }
353             for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
354                     if ((*p)->session == current->session)
```

```
355                                 (*p)->tty = -1;
356         }
    // 如果当前进程上次使用过协处理器，则把记录此信息的指针置空。若定义了调试进程树符号，
    // 则调用进程树检测显示函数。最后调用调度函数，重新调度进程运行，以让父进程能够处理
    // 僵死进程的其它善后事宜。
357         if (last_task_used_math == current)
358                 last_task_used_math = NULL;
359 #ifdef DEBUG_PROC_TREE
360         audit_ptree();
361 #endif
362         schedule();
363 }
364
    // 系统调用 exit()。终止进程。
    // 参数 error_code 是用户程序提供的退出状态信息，只有低字节有效。把 error_code 左移 8
    // 比特是 wait() 或 waitpid()函数的要求。低字节中将用来保存 wait()的状态信息。例如，
    // 如果进程处于暂停状态（TASK_STOPPED），那么其低字节就等于 0x7f。参见 sys/wait.h
    // 文件第 13--19 行。 wait() 或 waitpid() 利用这些宏就可以取得子进程的退出状态码或子
    // 进程终止的原因（信号）。
365 int sys_exit(int error_code)
366 {
367         do_exit((error_code&0xff)<<8);
368 }
369
    // 系统调用 waitpid()。挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止
    // 该进程的信号，或者是需要调用一个信号句柄（信号处理程序）。如果 pid 所指的子进程早已
    // 退出（已成所谓的僵死进程），则本调用将立刻返回。子进程使用的所有资源将释放。
    // 如果 pid > 0，表示等待进程号等于 pid 的子进程。
    // 如果 pid = 0，表示等待进程组号等于当前进程组号的任何子进程。
    // 如果 pid < -1，表示等待进程组号等于 pid 绝对值的任何子进程。
    // 如果 pid = -1，表示等待任何子进程。
    // 若 options = WUNTRACED，表示如果子进程是停止的，也马上返回（无须跟踪）。
    // 若 options = WNOHANG，表示如果没有子进程退出或终止就马上返回。
    // 如果返回状态指针 stat_addr 不为空，则就将状态信息保存到那里。
    // 参数 pid 是进程号；*stat_addr 是保存状态信息位置的指针；options 是 waitpid 选项。
370 int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
371 {
372         int flag;               // 该标志用于后面表示所选出的子进程处于就绪或睡眠态。
373         struct task_struct *p;
374         unsigned long oldblocked;
375
    // 首先验证将要存放状态信息的位置处内存空间足够。然后复位标志 flag。接着从当前进程的最
    // 年轻子进程开始扫描子进程兄弟链表。
376         verify_area(stat_addr,4);
377 repeat:
378         flag=0;
379         for (p = current->p_cptr ; p ; p = p->p_osptr) {
    // 如果等待的子进程号 pid>0，但与被扫描子进程 p 的 pid 不相等，说明它是当前进程另外的子
    // 进程，于是跳过该进程，接着扫描下一个进程。
380                 if (pid>0) {
381                         if (p->pid != pid)
382                                 continue;
    // 否则，如果指定等待进程的 pid=0，表示正在等待进程组号等于当前进程组号的任何子进程。
```

```
    // 如果此时被扫描进程 p 的进程组号与当前进程的组号不等，则跳过。
383                 } else if (!pid) {
384                         if (p->pgrp != current->pgrp)
385                                 continue;
    // 否则，如果指定的 pid < -1，表示正在等待进程组号等于 pid 绝对值的任何子进程。如果此时
    // 被扫描进程 p 的组号与 pid 的绝对值不等，则跳过。
386                 } else if (pid != -1) {
387                         if (p->pgrp != -pid)
388                                 continue;
389                 }
    // 如果前 3 个对 pid 的判断都不符合，则表示当前进程正在等待其任何子进程，也即 pid = -1
    // 的情况。此时所选择到的进程 p 或者是其进程号等于指定 pid，或者是当前进程组中的任何
    // 子进程，或者是进程号等于指定 pid 绝对值的子进程，或者是任何子进程（此时指定的 pid
    // 等于 -1）。接下来根据这个子进程 p 所处的状态来处理。
    // 当子进程 p 处于停止状态时，如果此时参数选项 options 中 WUNTRACED 标志没有置位，表示
    // 程序无须立刻返回，或者子进程此时的退出码等于 0，于是继续扫描处理其他子进程。 如果
    // WUNTRACED 置位且子进程退出码不为 0，则把退出码移入高字节，或上状态信息 0x7f 后放入
    // *stat_addr，在复位子进程退出码后就立刻返回子进程号 pid。这里 0x7f 表示的返回状态使
    // WIFSTOPPED()宏为真。参见 include/sys/wait.h，14 行。
390                 switch (p->state) {
391                         case TASK_STOPPED:
392                                 if (!(options & WUNTRACED) ||
393                                     !p->exit_code)
394                                         continue;
395                                 put_fs_long((p->exit_code << 8) | 0x7f,
396                                         stat_addr);
397                                 p->exit_code = 0;
398                                 return p->pid;
    // 如果子进程 p 处于僵死状态，则首先把它在用户态和内核态运行的时间分别累计到当前进程
    // （父进程）中，然后取出子进程的 pid 和退出码，把退出码放入返回状态位置 stat_addr 处
    // 并释放该子进程。最后返回子进程的退出码和 pid。 若定义了调试进程树符号，则调用进程
    // 树检测显示函数。
399                         case TASK_ZOMBIE:
400                                 current->cutime += p->utime;
401                                 current->cstime += p->stime;
402                                 flag = p->pid;
403                                 put_fs_long(p->exit_code, stat_addr);
404                                 release(p);
405 #ifdef DEBUG_PROC_TREE
406                                 audit_ptree();
407 #endif
408                                 return flag;
    // 如果这个子进程 p 的状态既不是停止也不是僵死，那么就置 flag = 1。表示找到过一个符合
    // 要求的子进程，但是它处于运行态或睡眠态。
409                         default:
410                                 flag=1;
411                                 continue;
412                 }
413         }
    // 在上面对任务数组扫描结束后，如果 flag 被置位，说明有符合等待要求的子进程并没有处
    // 于退出或僵死状态。此时如果已设置 WNOHANG 选项（表示若没有子进程处于退出或终止态就
    // 立刻返回），就立刻返回 0，退出。否则把当前进程置为可中断等待状态并，保留并修改
    // 当前进程信号阻塞位图，允许其接收到 SIGCHLD 信号。然后执行调度程序。当系统又开始
```

```
    // 执行本进程时，如果本进程收到除 SIGCHLD 以外的其他未屏蔽信号，则以退出码“重新启
    // 动系统调用”返回。否则跳转到函数开始处 repeat 标号处重复处理。
414         if (flag) {
415                 if (options & WNOHANG)
416                         return 0;
417                 current->state=TASK_INTERRUPTIBLE;
418                 oldblocked = current->blocked;
419                 current->blocked &= ~(1<<(SIGCHLD-1));
420                 schedule();
421                 current->blocked = oldblocked;
422                 if (current->signal & ~(current->blocked | (1<<(SIGCHLD-1))))
423                         return -ERESTARTSYS;
424                 else
425                         goto repeat;
426         }
    // 若 flag = 0，表示没有找到符合要求的子进程，则返回出错码（子进程不存在）。
427         return -ECHILD;
428 }
429
```

8.8 程序 8-8 linux/kernel/fork.c

```
 1 /*
 2  *  linux/kernel/fork.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  *  'fork.c' contains the help-routines for the 'fork' system call
 9  * (see also system_call.s), and some misc functions ('verify_area').
10  * Fork is rather simple, once you get the hang of it, but the memory
11  * management can be a bitch. See 'mm/mm.c': 'copy_page_tables()'
12  */
   /*
    * 'fork.c'中含有系统调用'fork'的辅助子程序（参见 system_call.s），以及一些
    * 其他函数（'verify_area'）。一旦你了解了 fork，就会发现它是非常简单的，但
    * 内存管理却有些难度。参见'mm/memory.c'中的'copy_page_tables()'函数。
    */
13 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
14
15 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据。
16 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
17 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
18 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
19
   // 写页面验证。若页面不可写，则复制页面。定义在 mm/memory.c 第 261 行开始。
20 extern void write_verify(unsigned long address);
21
22 long last_pid=0;          // 最新进程号，其值会由 get_empty_process()生成。
23
   //// 进程空间区域写前验证函数。
   // 对于 80386 CPU，在执行特权级 0 代码时不会理会用户空间中的页面是否是页保护的，因此
   // 在执行内核代码时用户空间中数据页面保护标志起不了作用，写时复制机制也就失去了作用。
   // verify_area()函数就用于此目的。但对于 80486 或后来的 CPU，其控制寄存器 CR0 中有一个
   // 写保护标志 WP（位 16），内核可以通过设置该标志来禁止特权级 0 的代码向用户空间只读
   // 页面执行写数据，否则将导致发生写保护异常。从而 486 以上 CPU 可以通过设置该标志来达
   // 到使用本函数同样的目的。
   // 该函数对当前进程逻辑地址从 addr 到 addr + size 这一段范围以页为单位执行写操作前
   // 的检测操作。由于检测判断是以页面为单位进行操作，因此程序首先需要找出 addr 所在页
   // 面开始地址 start，然后 start 加上进程数据段基址，使这个 start 变换成 CPU 4G 线性空
   // 间中的地址。最后循环调用 write_verify() 对指定大小的内存空间进行写前验证。若页面
   // 是只读的，则执行共享检验和复制页面操作（写时复制）。
24 void verify_area(void * addr,int size)
25 {
26         unsigned long start;
27
   // 首先将起始地址 start 调整为其所在页的左边界开始位置，同时相应地调整验证区域大小。
   // 下句中的 start & 0xfff 用来获得指定起始位置 addr（也即 start）在所在页面中的偏移
   // 值，原验证范围 size 加上这个偏移值即扩展成以 addr 所在页面起始位置开始的范围值。
   // 因此在 30 行上 也需要把验证开始位置 start 调整成页面边界值。参见前面的图“内存验
```

```
    // 证范围的调整”。
28          start = (unsigned long) addr;
29          size += start & 0xfff;
30          start &= 0xfffff000;                // 此时 start 是当前进程空间中的逻辑地址。
    // 下面把 start 加上进程数据段在线性地址空间中的起始基址，变成系统整个线性空间中的地
    // 址位置。对于 Linux 0.1x 内核，其数据段和代码段在线性地址空间中的基址和限长均相同。
    // 然后循环进行写页面验证。若页面不可写，则复制页面。（mm/memory.c，274 行）
31          start += get_base(current->ldt[2]);     // include/linux/sched.h，277 行。
32          while (size>0) {
33                  size -= 4096;
34                  write_verify(start);
35                  start += 4096;
36          }
37  }
38
    // 复制内存页表。
    // 参数 nr 是新任务号；p 是新任务数据结构指针。该函数为新任务在线性地址空间中设置代码
    // 段和数据段基址、限长，并复制页表。  由于 Linux 系统采用了写时复制（copy on write）
    // 技术， 因此这里仅为新进程设置自己的页目录表项和页表项，而没有实际为新进程分配物理
    // 内存页面。此时新进程与其父进程共享所有内存页面。操作成功返回 0，否则返回出错号。
39  int copy_mem(int nr,struct task_struct * p)
40  {
41          unsigned long old_data_base,new_data_base,data_limit;
42          unsigned long old_code_base,new_code_base,code_limit;
43
    // 首先取当前进程局部描述符表中代码段描述符和数据段描述符项中的段限长（字节数）。
    // 0x0f 是代码段选择符；0x17 是数据段选择符。然后取当前进程代码段和数据段在线性地址
    // 空间中的基地址。由于 Linux 0.12 内核还不支持代码和数据段分立的情况，因此这里需要
    // 检查代码段和数据段基址是否都相同，并且要求数据段的长度至少不小于代码段的长度
    // （参见图 5-12），否则内核显示出错信息，并停止运行。
    // get_limit()和 get_base()定义在 include/linux/sched.h 第 277 行和 279 行处。
44          code_limit=get_limit(0x0f);
45          data_limit=get_limit(0x17);
46          old_code_base = get_base(current->ldt[1]);
47          old_data_base = get_base(current->ldt[2]);
48          if (old_data_base != old_code_base)
49                  panic("We don't support separate I&D");
50          if (data_limit < code_limit)
51                  panic("Bad data_limit");
    // 然后设置创建中的新进程在线性地址空间中的基地址等于（64MB * 其任务号），并用该值
    // 设置新进程局部描述符表中段描述符中的基地址。接着设置新进程的页目录表项和页表项，
    // 即复制当前进程（父进程）的页目录表项和页表项。 此时子进程共享父进程的内存页面。
    // 正常情况下 copy_page_tables()返回 0，否则表示出错，则释放刚申请的页表项。
52          new_data_base = new_code_base = nr * TASK_SIZE;
53          p->start_code = new_code_base;
54          set_base(p->ldt[1],new_code_base);
55          set_base(p->ldt[2],new_data_base);
56          if (copy_page_tables(old_data_base,new_data_base,data_limit)) {
57                  free_page_tables(new_data_base,data_limit);
58                  return -ENOMEM;
59          }
60          return 0;
61  }
```

```
62
63 /*
64  *  Ok, this is the main fork-routine. It copies the system process
65  * information (task[nr]) and sets up the necessary registers. It
66  * also copies the data segment in it's entirety.
67  */
   /*
    * OK，下面是主要的 fork 子程序。它复制系统进程信息(task[n])
    * 并且设置必要的寄存器。它还整个地复制数据段（也是代码段）。
    */
   // 复制进程。
   // 该函数的参数是进入系统调用中断处理过程（sys_call.s）开始，直到调用本系统调用处理
   // 过程（sys_call.s 第 208 行）和调用本函数前（sys_call.s 第 217 行）逐步压入进程内核
   // 态栈的各寄存器的值。这些在 sys_call.s 程序中逐步压入内核态栈的值（参数）包括:
   // ① CPU 执行中断指令压入的用户栈地址 ss 和 esp、标志 eflags 和返回地址 cs 和 eip;
   // ② 第 83--88 行在刚进入 system_call 时入栈的段寄存器 ds、es、fs 和 edx、ecx、edx;
   // ③ 第 94 行调用 sys_call_table 中 sys_fork 函数时入栈的返回地址（参数 none 表示）;
   // ④ 第 212--216 行在调用 copy_process()之前入栈的 gs、esi、edi、ebp 和 eax（nr）。
   // 其中参数 nr 是调用 find_empty_process()分配的任务数组项号。
68 int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
69                 long ebx,long ecx,long edx, long orig_eax,
70                 long fs,long es,long ds,
71                 long eip,long cs,long eflags,long esp,long ss)
72 {
73         struct task_struct *p;
74         int i;
75         struct file *f;
76
   // 首先为新任务数据结构分配内存。如果内存分配出错，则返回出错码并退出。然后将新任务
   // 结构指针放入任务数组的 nr 项中。其中 nr 为任务号，由前面 find_empty_process()返回。
   // 接着把当前进程任务结构内容复制到刚申请到的内存页面 p 开始处。
77         p = (struct task_struct *) get_free_page();
78         if (!p)
79                 return -EAGAIN;
80         task[nr] = p;
81         *p = *current;  /* NOTE! this doesn't copy the supervisor stack */
                           /* 注意！这样做不会复制超级用户堆栈（只复制进程结构）*/
   // 随后对复制来的进程结构内容进行一些修改，作为新进程的任务结构。先将新进程的状态
   // 置为不可中断等待状态，以防止内核调度其执行。然后设置新进程的进程号 pid，并初始
   // 化进程运行时间片值等于其 priority 值（ 一般为 15 个嘀嗒）。接着复位新进程的信号
   // 位图、报警定时值、会话（session）领导标志 leader、 进程及其子进程在内核和用户
   // 态运行时间统计值，还设置进程开始运行的系统时间 start_time。
82         p->state = TASK_UNINTERRUPTIBLE;
83         p->pid = last_pid;              // 新进程号。也由 find_empty_process()得到。
84         p->counter = p->priority;       // 运行时间片值（嘀嗒数）。
85         p->signal = 0;                  // 信号位图。
86         p->alarm = 0;                   // 报警定时值（嘀嗒数）。
87         p->leader = 0;                  /* process leadership doesn't inherit */
                                           /* 进程的领导权是不能继承的 */
88         p->utime = p->stime = 0;        // 用户态时间和核心态运行时间。
89         p->cutime = p->cstime = 0;      // 子进程用户态和核心态运行时间。
90         p->start_time = jiffies;        // 进程开始运行时间（当前时间滴答数）。
   // 再修改任务状态段 TSS 数据（参见列表后说明）。由于系统给任务结构 p 分配了 1 页新
```

```
    // 内存，所以 (PAGE_SIZE + (long) p) 让 esp0 正好指向该页顶端。 ss0:esp0 用作程序
    // 在内核态执行时的栈。另外，在第 3 章中我们已经知道，每个任务在 GDT 表中都有两个
    // 段描述符，一个是任务的 TSS 段描述符，另一个是任务的 LDT 表段描述符。下面 111 行
    // 语句就是把 GDT 中本任务 LDT 段描述符的选择符保存在本任务的 TSS 段中。当 CPU 执行
    // 切换任务时，会自动从 TSS 中把 LDT 段描述符的选择符加载到 ldtr 寄存器中。
 91         p->tss.back_link = 0;
 92         p->tss.esp0 = PAGE_SIZE + (long) p;  // 任务内核态栈指针。
 93         p->tss.ss0 = 0x10;           // 内核态栈的段选择符（与内核数据段相同）。
 94         p->tss.eip = eip;            // 指令代码指针。
 95         p->tss.eflags = eflags;      // 标志寄存器。
 96         p->tss.eax = 0;              // 这是当 fork()返回时新进程会返回 0 的原因所在。
 97         p->tss.ecx = ecx;
 98         p->tss.edx = edx;
 99         p->tss.ebx = ebx;
100         p->tss.esp = esp;
101         p->tss.ebp = ebp;
102         p->tss.esi = esi;
103         p->tss.edi = edi;
104         p->tss.es = es & 0xffff;     // 段寄存器仅 16 位有效。
105         p->tss.cs = cs & 0xffff;
106         p->tss.ss = ss & 0xffff;
107         p->tss.ds = ds & 0xffff;
108         p->tss.fs = fs & 0xffff;
109         p->tss.gs = gs & 0xffff;
110         p->tss.ldt = _LDT(nr);       // 任务局部表描述符的选择符（LDT 描述符在 GDT 中）。
111         p->tss.trace_bitmap = 0x80000000;       //（高 16 位有效）。
    // 如果当前任务使用了协处理器，就保存其上下文。汇编指令 clts 用于清除控制寄存器 CR0
    // 中的任务已交换（TS）标志。每当发生任务切换，CPU 都会设置该标志。该标志用于管理
    // 数学协处理器：如果该标志置位，那么每个 ESC 指令都会被捕获（异常 7）。如果协处理
    // 器存在标志 MP 也同时置位的话，那么 WAIT 指令也会捕获。因此，如果任务切换发生在一
    // 个 ESC 指令开始执行之后，则协处理器中的内容就可能需要在执行新的 ESC 指令之前保存
    // 起来。捕获处理句柄会保存协处理器的内容并复位 TS 标志。指令 fnsave 用于把协处理器
    // 的所有状态保存到目的操作数指定的内存区域中（tss.i387）。
112         if (last_task_used_math == current)
113                 __asm__("clts ; fnsave %0 ; frstor %0"::"m" (p->tss.i387));

    // 接下来复制进程页表。即在线性地址空间中设置新任务代码段和数据段描述符中的基址
    // 和限长，并复制页表。如果出错（返回值不是 0），则复位任务数组中相应项并释放为
    // 该新任务分配的用于任务结构的内存页。
114         if (copy_mem(nr,p)) {               // 返回不为 0 表示出错。
115                 task[nr] = NULL;
116                 free_page((long) p);
117                 return -EAGAIN;
118         }
    // 如果父进程中有文件是打开的，则将对应文件的打开次数增 1。因为这里创建的子进程
    // 会与父进程共享这些打开的文件。将当前进程（父进程）的 pwd, root 和 executable
    // 引用次数均增 1。与上面同样的道理，子进程也引用了这些 i 节点。
119         for (i=0; i<NR_OPEN;i++)
120                 if (f=p->filp[i])
121                         f->f_count++;
122         if (current->pwd)
123                 current->pwd->i_count++;
124         if (current->root)
```

```
125                 current->root->i_count++;
126         if (current->executable)
127                 current->executable->i_count++;
128         if (current->library)
129                 current->library->i_count++;
    // 随后在 GDT 表中设置新任务 TSS 段和 LDT 段描述符项。这两个段的限长均被设置成 104
    // 字节。参见 include/asm/system.h，52—66 行代码。 然后设置进程之间的关系链表
    // 指针，即把新进程插入到当前进程的子进程链表中。把新进程的父进程设置为当前进程，
    // 把新进程的最新子进程指针 p_cptr 和年轻兄弟进程指针 p_ysptr 置空。接着让新进程
    // 的老兄进程指针 p_osptr 设置等于父进程的最新子进程指针。若当前进程却是还有其他
    // 子进程，则让比邻老兄进程的最年轻进程指针 p_yspter 指向新进程。最后把当前进程
    // 的最新子进程指针指向这个新进程。然后把新进程设置成就绪态。最后返回新进程号。
    // 另外， set_tss_desc() 和 set_ldt_desc() 定义在 include/asm/system.h 文件中。
    // “gdt+(nr<<1)+FIRST_TSS_ENTRY”是任务 nr 的 TSS 描述符项在全局表中的地址。因为
    // 每个任务占用 GDT 表中 2 项，因此上式中要包括'(nr<<1)'。
    // 请注意，在任务切换时，任务寄存器 tr 会由 CPU 自动加载。
130         set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
131         set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
132         p->p_pptr = current;                // 设置新进程的父进程指针。
133         p->p_cptr = 0;                      // 复位新进程的最新子进程指针。
134         p->p_ysptr = 0;                     // 复位新进程的比邻年轻兄弟进程指针。
135         p->p_osptr = current->p_cptr;       // 设置新进程的比邻老兄兄弟进程指针。
136         if (p->p_osptr)                     // 若新进程有老兄兄弟进程，则让其
137                 p->p_osptr->p_ysptr = p;    // 年轻进程兄弟指针指向新进程。
138         current->p_cptr = p;                // 让当前进程最新子进程指针指向新进程。
139         p->state = TASK_RUNNING;        /* do this last, just in case */
140         return last_pid;
141 }
142
    // 为新进程取得不重复的进程号 last_pid。函数返回在任务数组中的任务号(数组项)。
143 int find_empty_process(void)
144 {
145         int i;
146
    // 首先获取新的进程号。如果 last_pid 增 1 后超出进程号的正数表示范围，则重新从 1 开始
    // 使用 pid 号。 然后在任务数组中搜索刚设置的 pid 号是否已经被任何任务使用。如果是则
    // 跳转到函数开始处重新获得一个 pid 号。 接着在任务数组中为新任务寻找一个空闲项，并
    // 返回项号。 last_pid 是一个全局变量，不用返回。如果此时任务数组中 64 个项已经被全
    // 部占用，则返回出错码。
147         repeat:
148                 if ((++last_pid)<0) last_pid=1;
149                 for(i=0 ; i<NR_TASKS ; i++)
150                         if (task[i] && ((task[i]->pid == last_pid) ||
151                                         (task[i]->pgrp == last_pid)))
152                                 goto repeat;
153         for(i=1 ; i<NR_TASKS ; i++)             // 任务 0 项被排除在外。
154                 if (!task[i])
155                         return i;
156         return -EAGAIN;
157 }
158
```

8.9 程序 8-9 linux/kernel/sys.c 程序

```
1  /*
2   *  linux/kernel/sys.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
8
9  #include <linux/sched.h>  // 调度程序头文件。定义了任务结构 task_struct、任务 0 的数据，
                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
10 #include <linux/tty.h>    // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <linux/config.h> // 内核常数配置文件。这里主要使用其中的系统名称常数符号信息。
13 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14 #include <sys/times.h>    // 定义了进程中运行时间的结构 tms 以及 times()函数原型。
15 #include <sys/utsname.h>  // 系统名称结构头文件。
16 #include <sys/param.h>    // 系统参数头文件。含有系统一些全局常数符号。例如 HZ 等。
17 #include <sys/resource.h> // 系统资源头文件。含有有关进程资源使用情况的结构等信息。
18 #include <string.h>       // 字符串头文件。字符串或内存字节序列操作函数。
19
20 /*
21  * The timezone where the local system is located.  Used as a default by some
22  * programs who obtain this value by using gettimeofday.
23  */
   /*
    * 本系统所在的时区（timezone）。作为某些程序使用 gettimeofday 系统调用获取
    * 时区的默认值。
    */
   // 时区结构 timezone 第 1 个字段（tz_minuteswest）表示距格林尼治标准时间 GMT 以西的分钟
   // 数；第 2 个字段（tz_dsttime）是夏令时 DST（Daylight Savings Time）调整类型。 该结构
   // 定义在 include/sys/time.h 中。
24 struct timezone sys_tz = { 0, 0};
25
   // 根据进程组号 pgrp 取得进程组所属会话（session）号。该函数在 kernel/exit.c 中实现。
26 extern int session_of_pgrp(int pgrp);
27
   // 返回日期和时间（ftime - Fetch time）。
   // 以下返回值是-ENOSYS 的系统调用函数均表示在本版本内核中还未实现。
28 int sys_ftime()
29 {
30         return -ENOSYS;
31 }
32
33 int sys_break()
34 {
35         return -ENOSYS;
36 }
37
   // 用于当前进程对子进程进行调试(debugging)。
```

```
38 int sys_ptrace()
39 {
40         return -ENOSYS;
41 }
42
   // 改变并打印终端行设置。
43 int sys_stty()
44 {
45         return -ENOSYS;
46 }
47
   // 取终端行设置信息。
48 int sys_gtty()
49 {
50         return -ENOSYS;
51 }
52
   // 修改文件名。
53 int sys_rename()
54 {
55         return -ENOSYS;
56 }
57
58 int sys_prof()
59 {
60         return -ENOSYS;
61 }
62
63 /*
64  * This is done BSD-style, with no consideration of the saved gid, except
65  * that if you set the effective gid, it sets the saved gid too.  This
66  * makes it possible for a setgid program to completely drop its privileges,
67  * which is often a useful assertion to make when you are doing a security
68  * audit over a program.
69  *
70  * The general idea is that a program which uses just setregid() will be
71  * 100% compatible with BSD.  A program which uses just setgid() will be
72  * 100% compatible with POSIX w/ Saved ID's.
73  */
   /*
    * 以下是 BSD 形式的实现，没有考虑保存的 gid（saved gid 或 sgid），除了当你
    * 设置了有效的 gid（effective gid 或 egid）时，保存的 gid 也会被设置。这使
    * 得一个使用 setgid 的程序可以完全放弃其特权。当你在对一个程序进行安全审
    * 计时，这通常是一种很好的处理方法。
    *
    * 最基本的考虑是一个使用 setregid()的程序将会与 BSD 系统 100%的兼容。而一
    * 个使用 setgid()和保存的 gid 的程序将会与 POSIX 100%的兼容。
    */
   // 设置当前任务的实际以及/或者有效组 ID（gid）。如果任务没有超级用户特权，那么只能互
   // 换其实际组 ID 和有效组 ID。如果任务具有超级用户特权，就能任意设置有效的和实际的组
   // ID。保留的 gid（saved gid）被设置成与有效 gid。实际组 ID 是指进程当前的 gid。
74 int sys_setregid(int rgid, int egid)
75 {
```

```
76          if (rgid>0) {
77                  if ((current->gid == rgid) ||
78                      suser())
79                          current->gid = rgid;
80                  else
81                          return(-EPERM);
82          }
83          if (egid>0) {
84                  if ((current->gid == egid) ||
85                      (current->egid == egid) ||
86                      suser()) {
87                          current->egid = egid;
88                          current->sgid = egid;
89                  } else
90                          return(-EPERM);
91          }
92          return 0;
93  }
94
95  /*
96   * setgid() is implemeneted like SysV w/ SAVED_IDS
97   */
    /*
     * setgid()的实现与具有 SAVED_IDS 的 SYSV 的实现方法相似。
     */
    // 设置进程组号(gid)。如果任务没有超级用户特权，它可以使用 setgid() 将其有效 gid
    // （effective gid）设置为成其保留 gid(saved gid)或其实际 gid(real gid)。如果任务
    // 有超级用户特权，则实际 gid、有效 gid 和保留 gid 都被设置成参数指定的 gid。
98  int sys_setgid(int gid)
99  {
100         if (suser())
101                 current->gid = current->egid = current->sgid = gid;
102         else if ((gid == current->gid) || (gid == current->sgid))
103                 current->egid = gid;
104         else
105                 return -EPERM;
106         return 0;
107 }
108
    // 打开或关闭进程计帐功能。
109 int sys_acct()
110 {
111         return -ENOSYS;
112 }
113
    // 映射任意物理内存到进程的虚拟地址空间。
114 int sys_phys()
115 {
116         return -ENOSYS;
117 }
118
119 int sys_lock()
120 {
```

```
121         return -ENOSYS;
122 }
123
124 int sys_mpx()
125 {
126         return -ENOSYS;
127 }
128
129 int sys_ulimit()
130 {
131         return -ENOSYS;
132 }
133
    // 返回从 1970 年 1 月 1 日 00:00:00 GMT 开始计时的时间值（秒）。如果 tloc 不为 null，
    // 则时间值也存储在那里。
    // 由于参数是一个指针，而其所指位置在用户空间，因此需要使用函数 put_fs_long() 来
    // 访问该值。在进入内核中运行时，段寄存器 fs 被默认地指向当前用户数据空间。因此该
    // 函数就可利用 fs 来访问用户空间中的值。
134 int sys_time(long * tloc)
135 {
136         int i;
137
138         i = CURRENT_TIME;
139         if (tloc) {
140                 verify_area(tloc,4);        // 验证内存容量是否够（这里是 4 字节）。
141                 put_fs_long(i,(unsigned long *)tloc);   // 放入用户数据段 tloc 处。
142         }
143         return i;
144 }
145
146 /*
147  * Unprivileged users may change the real user id to the effective uid
148  * or vice versa.  (BSD-style)
149  *
150  * When you set the effective uid, it sets the saved uid too.  This
151  * makes it possible for a setuid program to completely drop its privileges,
152  * which is often a useful assertion to make when you are doing a security
153  * audit over a program.
154  *
155  * The general idea is that a program which uses just setreuid() will be
156  * 100% compatible with BSD.  A program which uses just setuid() will be
157  * 100% compatible with POSIX w/ Saved ID's.
158  */
    /*
     * 无特权的用户可以见实际的 uid（real uid）改成有效的 uid（effective uid），
     * 反之也然。（BSD 形式的实现）
     *
     * 当你设置有效的 uid 时，它同时也设置了保存的 uid。这使得一个使用 setuid
     * 的程序可以完全放弃其特权。当你在对一个程序进行安全审计时，这通常是一种
     * 很好的处理方法。
     * 最基本的考虑是一个使用 setreuid()的程序将会与 BSD 系统 100%的兼容。而一
     * 个使用 setuid()和保存的 gid 的程序将会与 POSIX 100%的兼容。
     */
```

```
    // 设置任务的实际以及/或者有效的用户 ID（uid）。如果任务没有超级用户特权，那么只能
    // 互换其实际的 uid 和有效的 uid。如果任务具有超级用户特权，就能任意设置有效的和实
    // 际的用户 ID。保存的 uid（saved uid）被设置成与有效 uid 同值。
159 int sys_setreuid(int ruid, int euid)
160 {
161         int old_ruid = current->uid;
162
163         if (ruid>0) {
164                 if ((current->euid==ruid) ||
165                     (old_ruid == ruid) ||
166                     suser())
167                         current->uid = ruid;
168                 else
169                         return(-EPERM);
170         }
171         if (euid>0) {
172                 if ((old_ruid == euid) ||
173                     (current->euid == euid) ||
174                     suser()) {
175                         current->euid = euid;
176                         current->suid = euid;
177                 } else {
178                         current->uid = old_ruid;
179                         return(-EPERM);
180                 }
181         }
182         return 0;
183 }
184
185 /*
186  * setuid() is implemeneted like SysV w/ SAVED_IDS
187  *
188  * Note that SAVED_ID's is deficient in that a setuid root program
189  * like sendmail, for example, cannot set its uid to be a normal
190  * user and then switch back, because if you're root, setuid() sets
191  * the saved uid too.  If you don't like this, blame the bright people
192  * in the POSIX commmittee and/or USG.  Note that the BSD-style setreuid()
193  * will allow a root program to temporarily drop privileges and be able to
194  * regain them by swapping the real and effective uid.
195  */
    /*
     * setuid()的实现与具有 SAVED_IDS 的 SYSV 的实现方法相似。
     *
     * 请注意使用 SAVED_ID 的 setuid()在某些方面是不完善的。例如，一个使用
     * setuid 的超级用户程序 sendmail 就做不到把其 uid 设置成一个普通用户的
     * uid，然后再交换回来。 因为如果你是一个超级用户，setuid() 也同时会
     * 设置保存的 uid。如果你不喜欢这样的做法的话，就责怪 POSIX 组委会以及
     * /或者 USG 中的聪明人吧。不过请注意 BSD 形式的 setreuid()实现能够允许
     * 一个超级用户程序临时放弃特权，并且能通过交换实际的和有效的 uid 而
     * 再次获得特权。
     */
    // 设置任务用户 ID（uid）。如果任务没有超级用户特权，它可以使用 setuid()将其有效的
    // uid（effective uid）设置成其保存的 uid（saved uid）或其实际的 uid（real uid）。
```

```
    // 如果任务有超级用户特权，则实际的 uid、有效的 uid 和保存的 uid 都会被设置成参数指
    // 定的 uid。
196 int sys_setuid(int uid)
197 {
198         if (suser())
199                 current->uid = current->euid = current->suid = uid;
200         else if ((uid == current->uid) || (uid == current->suid))
201                 current->euid = uid;
202         else
203                 return -EPERM;
204         return(0);
205 }
206
    // 设置系统开机时间。参数 tptr 是从 1970 年 1 月 1 日 00:00:00 GMT 开始计时的时间值（秒）。
    // 调用进程必须具有超级用户权限。其中 HZ=100，是内核系统运行频率。
    // 由于参数是一个指针，而其所指位置在用户空间，因此需要使用函数 get_fs_long()来访问该
    // 值。在进入内核中运行时，段寄存器 fs 被默认地指向当前用户数据空间。因此该函数就可利
    // 用 fs 来访问用户空间中的值。
    // 函数参数提供的当前时间值减去系统已经运行的时间秒值（jiffies/HZ）即是开机时间秒值。
207 int sys_stime(long * tptr)
208 {
209         if (!suser())                   // 如果不是超级用户则出错返回（许可）。
210                 return -EPERM;
211         startup_time = get_fs_long((unsigned long *)tptr) - jiffies/HZ;
212         jiffies_offset = 0;
213         return 0;
214 }
215
    // 获取当前任务运行时间统计值。
    // 在 tbuf 所指用户数据空间处返回 tms 结构的任务运行时间统计值。tms 结构中包括进程用户
    // 运行时间、内核（系统）时间、子进程用户运行时间、子进程系统运行时间。函数返回值是
    // 系统运行到当前的嘀嗒数。
216 int sys_times(struct tms * tbuf)
217 {
218         if (tbuf) {
219                 verify_area(tbuf,sizeof *tbuf);
220                 put_fs_long(current->utime,(unsigned long *)&tbuf->tms_utime);
221                 put_fs_long(current->stime,(unsigned long *)&tbuf->tms_stime);
222                 put_fs_long(current->cutime,(unsigned long *)&tbuf->tms_cutime);
223                 put_fs_long(current->cstime,(unsigned long *)&tbuf->tms_cstime);
224         }
225         return jiffies;
226 }
227
    // 当参数 end_data_seg 数值合理，并且系统确实有足够的内存，而且进程没有超越其最大数据
    // 段大小时，该函数设置数据段末尾为 end_data_seg 指定的值。该值必须大于代码结尾并且要
    // 小于堆栈结尾 16KB。返回值是数据段的新结尾值（如果返回值与要求值不同，则表明有错误
    // 发生）。该函数并不被用户直接调用，而由 libc 库函数进行包装，并且返回值也不一样。
228 int sys_brk(unsigned long end_data_seg)
229 {
    // 如果参数值大于代码结尾，并且小于（堆栈 - 16KB），则设置新数据段结尾值。
230         if (end_data_seg >= current->end_code &&
231             end_data_seg < current->start_stack - 16384)
```

```
232                 current->brk = end_data_seg;
233         return current->brk;                // 返回进程当前的数据段结尾值。
234 }
235
236 /*
237  * This needs some heave checking ...
238  * I just haven't get the stomach for it. I also don't fully
239  * understand sessions/pgrp etc. Let somebody who does explain it.
240  *
241  * OK, I think I have the protection semantics right.... this is really
242  * only important on a multi-user system anyway, to make sure one user
243  * can't send a signal to a process owned by another.  -TYT, 12/12/91
244  */
    /*
     * 下面代码需要某些严格的检查…
     * 我只是没有胃口来做这些。我也不完全明白 sessions/pgrp 等的含义。还是让
     * 了解它们的人来做吧。
     *
     * OK，我想我已经正确地实现了保护语义....。总之，这其实只对多用户系统是
     * 重要的，以确定一个用户不能向其他用户的进程发送信号。 -TYT 12/12/91
     */
    // 设置指定进程 pid 的进程组号为 pgid。
    // 参数 pid 是指定进程的进程号。如果它为 0，则让 pid 等于当前进程的进程号。参数 pgid
    // 是指定的进程组号。如果它为 0，则让它等于进程 pid 的进程组号。如果该函数用于将进程
    // 从一个进程组移到另一个进程组，则这两个进程组必须属于同一个会话(session)。在这种
    // 情况下，参数 pgid 指定了要加入的现有进程组 ID，此时该组的会话 ID 必须与将要加入进
    // 程的相同(263 行)。
245 int sys_setpgid(int pid, int pgid)
246 {
247         int i;
248
    // 如果参数 pid 为 0，则 pid 取值为当前进程的进程号 pid。如果参数 pgid 为 0，则 pgid 也
    // 取值为当前进程的 pid。[?? 这里与 POSIX 标准的描述有出入 ]。若 pgid 小于 0，则返回
    // 无效错误码。
249         if (!pid)
250                 pid = current->pid;
251         if (!pgid)
252                 pgid = current->pid;
253         if (pgid < 0)
254                 return -EINVAL;
    // 扫描任务数组，查找指定进程号 pid 的任务。如果找到了进程号是 pid 的进程，并且该进程
    // 的父进程就是当前进程或者该进程就是当前进程，那么若该任务已经是会话首领，则出错返回。
    // 若该任务的会话号（session）与当前进程的不同，或者指定的进程组号 pgid 与 pid 不同并且
    // pgid 进程组所属会话号与当前进程所属会话号不同，则也出错返回。 否则把查找到的进程的
    // pgrp 设置为 pgid，并返回 0。若没有找到指定 pid 的进程，则返回进程不存在出错码。
255         for (i=0 ; i<NR_TASKS ; i++)
256                 if (task[i] && (task[i]->pid == pid) &&
257                     ((task[i]->p_pptr == current) ||
258                      (task[i] == current))) {
259                         if (task[i]->leader)
260                                 return -EPERM;
261                         if ((task[i]->session != current->session) ||
262                             ((pgid != pid) &&
```

```
263                             (session_of_pgrp(pgid) != current->session)))
264                                 return -EPERM;
265                         task[i]->pgrp = pgid;
266                         return 0;
267                 }
268         return -ESRCH;
269 }
270
    // 返回当前进程的进程组号。与 getpgid(0)等同。
271 int sys_getpgrp(void)
272 {
273         return current->pgrp;
274 }
275
    // 创建一个会话(session)（即设置其 leader=1），并且设置其会话号=其组号=其进程号。
    // 如果当前进程已是会话首领并且不是超级用户，则出错返回。否则设置当前进程为新会话
    // 首领（leader = 1），并且设置当前进程会话号 session 和组号 pgrp 都等于进程号 pid，
    // 而且设置当前进程没有控制终端。最后系统调用返回会话号。
276 int sys_setsid(void)
277 {
278         if (current->leader && !suser())
279                 return -EPERM;
280         current->leader = 1;
281         current->session = current->pgrp = current->pid;
282         current->tty = -1;                      // 表示当前进程没有控制终端。
283         return current->pgrp;
284 }
285
286 /*
287  * Supplementary group ID's
288  */
    /*
     * 进程的其他用户组号。
     */
    // 取当前进程其他辅助用户组号。
    // 任务数据结构中 groups[]数组保存着进程同时所属的多个用户组号。该数组共 NGROUPS 个项，
    // 若某项的值是 NOGROUP（即为 -1），则表示从该项开始以后所有项都空闲。否则数组项中保
    // 存的是用户组号。
    // 参数 gidsetsize 是获取的用户组号个数；grouplist 是存储这些用户组号的用户空间缓存。
289 int sys_getgroups(int gidsetsize, gid_t *grouplist)
290 {
291         int     i;
292
    // 首先验证 grouplist 指针所指的用户缓存空间是否足够，然后从当前进程结构的 groups[]
    // 数组中逐个取得用户组号并复制到用户缓存中。在复制过程中，如果 groups[] 中的项数
    // 大于给定的参数 gidsetsize 所指定的个数，则表示用户给出的缓存太小，不能容下当前
    // 进程所有组号，因此此次取组号操作会出错返回。若复制过程正常，则函数最后会返回复
    // 制的用户组号个数。（gidsetsize - gid set size，用户组号集大小）。
293         if (gidsetsize)
294                 verify_area(grouplist, sizeof(gid_t) * gidsetsize);
295
296         for (i = 0; (i < NGROUPS) && (current->groups[i] != NOGROUP);
297              i++, grouplist++) {
```

```
298                 if (gidsetsize) {
299                         if (i >= gidsetsize)
300                                 return -EINVAL;
301                         put_fs_word(current->groups[i], (short *) grouplist);
302                 }
303         }
304         return(i);              // 返回实际含有的用户组号个数。
305 }
306
    // 设置当前进程同时所属的其他辅助用户组号。
    // 参数 gidsetsize 是将设置的用户组号个数；grouplist 是含有用户组号的用户空间缓存。
307 int sys_setgroups(int gidsetsize, gid_t *grouplist)
308 {
309         int     i;
310
    // 首先查权限和参数的有效性。只有超级用户可以修改或设置当前进程的辅助用户组号，而且
    // 设置的项数不能超过进程的 groups[NGROUPS]数组的容量。然后从用户缓冲中逐个复制用户
    // 组号，共 gidsetsize 个。如果复制的个数没有填满 groups[]，则在随后一项上填上值为-1
    // 的项（NOGROUP）。最后函数返回 0。
311         if (!suser())
312                 return -EPERM;
313         if (gidsetsize > NGROUPS)
314                 return -EINVAL;
315         for (i = 0; i < gidsetsize; i++, grouplist++) {
316                 current->groups[i] = get_fs_word((unsigned short *) grouplist);
317         }
318         if (i < NGROUPS)
319                 current->groups[i] = NOGROUP;
320         return 0;
321 }
322
    // 检查当前进程是否在指定的用户组 grp 中。是则返回 1，否则返回 0。
323 int in_group_p(gid_t grp)
324 {
325         int     i;
326
    // 如果当前进程的有效组号就是 grp，则表示进程属于 grp 进程组。函数返回 1。否则就在
    // 进程的辅助用户组数组中扫描是否有 grp 进程组号。若有则函数也返回 1。若扫描到值
    // 为 NOGROUP 的项，表示已扫描完全部有效项而没有发现匹配的组号，因此函数返回 0。
327         if (grp == current->egid)
328                 return 1;
329
330         for (i = 0; i < NGROUPS; i++) {
331                 if (current->groups[i] == NOGROUP)
332                         break;
333                 if (current->groups[i] == grp)
334                         return 1;
335         }
336         return 0;
337 }
338
    // utsname 结构含有一些字符串字段。用于保存系统的名称。其中包含 5 个字段，分别是：
    // 当前操作系统的名称、网络节点名称（主机名）、当前操作系统发行级别、操作系统版本
```

```
    // 号以及系统运行的硬件类型名称。该结构定义在 include/sys/utsname.h 文件中。 这里
    // 内核使用 include/linux/config.h 文件中的常数符号设置了它们的默认值。它们分别为
    // “Linux”，“(none)”，“0”，“0.12”，“i386”。
339 static struct utsname thisname = {
340         UTS_SYSNAME, UTS_NODENAME, UTS_RELEASE, UTS_VERSION, UTS_MACHINE
341 };
342
    // 获取系统名称等信息。
343 int sys_uname(struct utsname * name)
344 {
345         int i;
346
347         if (!name) return -ERROR;
348         verify_area(name,sizeof *name);
349         for(i=0;i<sizeof *name;i++)
350                 put_fs_byte(((char *) &thisname)[i],i+(char *) name);
351         return 0;
352 }
353
354 /*
355  * Only sethostname; gethostname can be implemented by calling uname()
356  */
    /*
     * 通过调用 uname()只能实现 sethostname 和 gethostname。
     */
    // 设置系统主机名（系统的网络节点名）。
    // 参数 name 指针指向用户数据区中含有主机名字符串的缓冲区；len 是主机名字符串长度。
357 int sys_sethostname(char *name, int len)
358 {
359         int     i;
360
    // 系统主机名只能由超级用户设置或修改，并且主机名长度不能超过最大长度 MAXHOSTNAMELEN。
361         if (!suser())
362                 return -EPERM;
363         if (len > MAXHOSTNAMELEN)
364                 return -EINVAL;
365         for (i=0; i < len; i++) {
366                 if ((thisname.nodename[i] = get_fs_byte(name+i)) == 0)
367                         break;
368         }
    // 在复制完毕后，如果用户提供的字符串中没有包含 NULL 字符，那么若复制的主机名长度还没有
    // 超过 MAXHOSTNAMELEN，则在主机名字符串后添加一个 NULL。若已经填满 MAXHOSTNAMELEN 个字
    // 符，则把最后一个字符改成 NULL 字符。即 thisname.nodename[min(i,MAXHOSTNAMELEN)] = 0。
369         if (thisname.nodename[i]) {
370                 thisname.nodename[i>MAXHOSTNAMELEN ? MAXHOSTNAMELEN : i] = 0;
371         }
372         return 0;
373 }
374
    // 取当前进程指定资源的界限值。
    // 进程的任务结构中定义有一个数组 rlim[RLIM_NLIMITS]，用于控制进程使用系统资源的界限。
    // 数组每个项是一个 rlimit 结构，其中包含两个字段。 一个说明进程对指定资源的当前限制
    // 界限（soft limit，即软限制），另一个说明系统对指定资源的最大限制界限（hard limit，
```

```
    // 即硬限制）。 rlim[] 数组的每一项对应系统对当前进程一种资源的界限信息。Linux 0.12
    // 系统共对 6 种资源规定了界限，即 RLIM_NLIMITS=6。请参考头文件 include/sys/resource.h
    // 中第 41 — 46 行的说明。
    // 参数 resource 指定我们咨询的资源名称，实际上它是任务结构中 rlim[]数组的索引项值。
    // 参数 rlim 是指向 rlimit 结构的用户缓冲区指针，用于存放取得的资源界限信息。
375 int sys_getrlimit(int resource, struct rlimit *rlim)
376 {
    // 所咨询的资源 resource 实际上是进程任务结构中 rlim[]数组的索引项值。该索引值当然不能
    // 大于数组的最大项数 RLIM_NLIMITS。在验证过 rlim 指针所指用户缓冲足够以后，这里就把
    // 参数指定的资源 resource 结构信息复制到用户缓冲中，并返回 0。
377         if (resource >= RLIM_NLIMITS)
378                 return -EINVAL;
379         verify_area(rlim,sizeof *rlim);
380         put_fs_long(current->rlim[resource].rlim_cur,       // 当前（软）限制值。
381                     (unsigned long *) rlim);
382         put_fs_long(current->rlim[resource].rlim_max,       // 系统（硬）限制值。
383                     ((unsigned long *) rlim)+1);
384         return 0;
385 }
386
    // 设置当前进程指定资源的界限值。
    // 参数 resource 指定我们设置界限的资源名称，实际上它是任务结构中 rlim[]数组的索引
    // 项值。参数 rlim 是指向 rlimit 结构的用户缓冲区指针，用于内核读取新的资源界限信息。
387 int sys_setrlimit(int resource, struct rlimit *rlim)
388 {
389         struct rlimit new, *old;
390
    // 首先判断参数 resource（任务结构 rlim[]项索引值）有效性。然后先让 rlimit 结构指针
    // old 指向指进程任务结构中指定资源的当前 rlimit 结构信息。接着把用户提供的资源界限
    // 信息复制到临时 rlimit 结构 new 中。此时如果判断出 new 结构中的软界限值或硬界限值
    // 大于进程该资源原硬界限值，并且当前不是超级用户的话，就返回许可错。否则表示 new
    // 中信息合理或者进程是超级用户进程，则修改原进程指定资源信息等于 new 结构中的信息，
    // 并成功返回 0。
391         if (resource >= RLIM_NLIMITS)
392                 return -EINVAL;
393         old = current->rlim + resource;         // 即 old = current->rlim[resource]。
394         new.rlim_cur = get_fs_long((unsigned long *) rlim);
395         new.rlim_max = get_fs_long(((unsigned long *) rlim)+1);
396         if (((new.rlim_cur > old->rlim_max) ||
397              (new.rlim_max > old->rlim_max)) &&
398             !suser())
399                 return -EPERM;
400         *old = new;
401         return 0;
402 }
403
404 /*
405  * It would make sense to put struct rusuage in the task_struct,
406  * except that would make the task_struct be *really big*.  After
407  * task_struct gets moved into malloc'ed memory, it would
408  * make sense to do this.  It will make moving the rest of the information
409  * a lot simpler!  (Which we're not doing right now because we're not
410  * measuring them yet).
```

```
411  */
     /*
      * 把 rusuage 结构放进任务结构 task struct 中是恰当的，除非它会使任务
      * 结构长度变得非常大。在把任务结构移入内核 malloc 分配的内存中之后，
      * 这样做即使任务结构很大也没问题了。这将使得其余信息的移动变得非常
      * 方便！（我们还没有这样做，因为我们还没有测试过它们的大小）。
      */
     // 获取指定进程的资源利用信息。
     // 本系统调用提供当前进程或其已终止的和等待着的子进程资源使用情况。如果参数 who 等于
     // RUSAGE_SELF，则返回当前进程的资源利用信息。如果指定进程 who 是 RUSAGE_CHILDREN，
     // 则返回当前进程的已终止和等待着的子进程资源利用信息。 符号常数 RUSAGE_SELF 和
     // RUSAGE_CHILDREN 以及 rusage 结构都定义在 include/sys/resource.h 头文件中。
412  int sys_getrusage(int who, struct rusage *ru)
413  {
414          struct rusage r;
415          unsigned long   *lp, *lpend, *dest;
416
     // 首先判断参数指定进程的有效性。如果 who 既不是 RUSAGE_SELF（指定当前进程），也不是
     // RUSAGE_CHILDREN （指定子进程），则以无效参数码返回。否则在验证了指针 ru 指定的用
     // 户缓冲区域后，把临时 rusage 结构区域 r 清零。
417          if (who != RUSAGE_SELF && who != RUSAGE_CHILDREN)
418                  return -EINVAL;
419          verify_area(ru, sizeof *ru);
420          memset((char *) &r, 0, sizeof(r));      // 在 include/strings.h 文件最后。
     // 若参数 who 是 RUSAGE_SELF，则复制当前进程资源利用信息到 r 结构中。若指定进程 who
     // 是 RUSAGE_CHILDREN， 则复制当前进程的已终止和等待着的子进程资源利用信息到临时
     // rusuage 结构 r 中。宏 CT_TO_SECS 和 CT_TO_USECS 用于把系统当前嘀嗒数转换成用秒值
     // 加微秒值表示。它们定义在 include/linux/sched.h 文件中。 jiffies_offset 是系统
     // 嘀嗒数误差调整数。
421          if (who == RUSAGE_SELF) {
422                  r.ru_utime.tv_sec = CT_TO_SECS(current->utime);
423                  r.ru_utime.tv_usec = CT_TO_USECS(current->utime);
424                  r.ru_stime.tv_sec = CT_TO_SECS(current->stime);
425                  r.ru_stime.tv_usec = CT_TO_USECS(current->stime);
426          } else {
427                  r.ru_utime.tv_sec = CT_TO_SECS(current->cutime);
428                  r.ru_utime.tv_usec = CT_TO_USECS(current->cutime);
429                  r.ru_stime.tv_sec = CT_TO_SECS(current->cstime);
430                  r.ru_stime.tv_usec = CT_TO_USECS(current->cstime);
431          }
     // 然后让 lp 指针指向 r 结构，lpend 指向 r 结构末尾处，而 dest 指针指向用户空间中的 ru
     // 结构。最后把 r 中信息复制到用户空间 ru 结构中，并返回 0。
432          lp = (unsigned long *) &r;
433          lpend = (unsigned long *) (&r+1);
434          dest = (unsigned long *) ru;
435          for (; lp < lpend; lp++, dest++)
436                  put_fs_long(*lp, dest);
437          return(0);
438  }
439
     // 取得系统当前时间，并用指定格式返回。
     // timeval 结构和 timezone 结构都定义在 include/sys/time.h 文件中。timeval 结构含有秒
     // 和微秒（tv_sec 和 tv_usec）两个字段。timezone 结构含有本地距格林尼治标准时间以西
```

```
    // 的分钟数（tz_minuteswest）和夏令时间调整类型（tz_dsttime）两个字段。
    //  （dst -- Daylight Savings Time）
440 int sys_gettimeofday(struct timeval *tv, struct timezone *tz)
441 {
    // 如果参数给定的 timeval 结构指针不空，则在该结构中返回当前时间（秒值和微秒值）；
    // 如果参数给定的用户数据空间中 timezone 结构的指针不空，则也返回该结构的信息。
    // 程序中 startup_time 是系统开机时间（秒值）。 宏 CT_TO_SECS 和 CT_TO_USECS 用于
    // 把系统当前嘀嗒数转换成用秒值加微秒值表示。它们定义在 include/linux/sched.h
    // 文件中。jiffies_offset 是系统嘀嗒数误差调整数。
442         if (tv) {
443                 verify_area(tv, sizeof *tv);
444                 put_fs_long(startup_time + CT_TO_SECS(jiffies+jiffies_offset),
445                             (unsigned long *) tv);
446                 put_fs_long(CT_TO_USECS(jiffies+jiffies_offset),
447                             ((unsigned long *) tv)+1);
448         }
449         if (tz) {
450                 verify_area(tz, sizeof *tz);
451                 put_fs_long(sys_tz.tz_minuteswest, (unsigned long *) tz);
452                 put_fs_long(sys_tz.tz_dsttime, ((unsigned long *) tz)+1);
453         }
454         return 0;
455 }
456
457 /*
458  * The first time we set the timezone, we will warp the clock so that
459  * it is ticking GMT time instead of local time.  Presumably,
460  * if someone is setting the timezone then we are running in an
461  * environment where the programs understand about timezones.
462  * This should be done at boot time in the /etc/rc script, as
463  * soon as possible, so that the clock can be set right.  Otherwise,
464  * various programs will get confused when the clock gets warped.
465  */
    /*
     * 在第 1 次设置时区（timezone）时，我们会改变时钟值以让系统使用格林
     * 尼治标准时间（GMT）运行，而非使用本地时间。 推测起来说，如果某人
     * 设置了时区时间，那么我们就运行在程序知晓时区时间的环境中。设置时
     * 区操作应该在系统启动阶段，尽快地在/etc/rc 脚本程序中进行。这样时
     * 钟就可以设置正确。 否则的话，若我们以后才设置时区而导致时钟时间
     * 改变，可能会让一些程序的运行出现问题。
     */
    // 设置系统当前时间。
    // 参数 tv 是指向用户数据区中 timeval 结构信息的指针。参数 tz 是用户数据区中 timezone
    // 结构的指针。该操作需要超级用户权限。如果两者皆为空，则什么也不做，函数返回 0。
466 int sys_settimeofday(struct timeval *tv, struct timezone *tz)
467 {
468         static int      firsttime = 1;
469         void            adjust_clock();
470
    // 设置系统当前时间需要超级用户权限。如果 tz 指针不空，则设置系统时区信息。即复制用户
    // timezone 结构信息到系统中的 sys_tz 结构中（见第 24 行）。如果是第 1 次调用本系统调用
    // 并且参数 tv 指针不空，则调整系统时钟值。
471         if (!suser())
```

```
472                 return -EPERM;
473         if (tz) {
474                 sys_tz.tz_minuteswest = get_fs_long((unsigned long *) tz);
475                 sys_tz.tz_dsttime = get_fs_long(((unsigned long *) tz)+1);
476                 if (firsttime) {
477                         firsttime = 0;
478                         if (!tv)
479                                 adjust_clock();
480                 }
481         }
    // 如果参数的 timeval 结构指针 tv 不空，则用该结构信息设置系统时钟。首先从 tv 所指处
    // 获取以秒值（sec）加微秒值（usec）表示的系统时间，然后用秒值修改系统开机时间全局
    // 变量 startup_time 值，并用微秒值设置系统嘀嗒误差值 jiffies_offset。
482         if (tv) {
483                 int sec, usec;
484
485                 sec = get_fs_long((unsigned long *)tv);
486                 usec = get_fs_long(((unsigned long *)tv)+1);
487
488                 startup_time = sec - jiffies/HZ;
489                 jiffies_offset = usec * HZ / 1000000 - jiffies%HZ;
490         }
491         return 0;
492 }
493
494 /*
495  * Adjust the time obtained from the CMOS to be GMT time instead of
496  * local time.
497  *
498  * This is ugly, but preferable to the alternatives.  Otherwise we
499  * would either need to write a program to do it in /etc/rc (and risk
500  * confusion if the program gets run more than once; it would also be
501  * hard to make the program warp the clock precisely n hours)  or
502  * compile in the timezone information into the kernel.  Bad, bad....
503  *
504  * XXX Currently does not adjust for daylight savings time.  May not
505  * need to do anything, depending on how smart (dumb?) the BIOS
506  * is.  Blast it all.... the best thing to do not depend on the CMOS
507  * clock at all, but get the time via NTP or timed if you're on a
508  * network....                          - TYT, 1/1/92
509  */
    /*
     * 把从 CMOS 中读取的时间值调整为 GMT 时间值保存，而非本地时间值。
     *
     * 这里的做法很蹩脚，但要比其他方法好。否则我们就需要写一个程序并让它
     * 在/etc/rc 中运行来做这件事（并且冒着该程序可能会被多次执行而带来的
     * 问题。 而且这样做也很难让程序把时钟精确地调整 n 小时） 或者把时区信
     * 息编译进内核中。当然这样做就非常、非常差劲了...
     *
     * 目前下面函数（XXX）的调整操作并没有考虑到夏令时问题。依据 BIOS 有多
     * 么智能（愚蠢？）也许根本就不用考虑这方面。当然，最好的做法是完全不
     * 依赖于 CMOS 时钟，而是让系统通过 NTP（网络时钟协议）或者 timed（时间
     * 服务器）获得时间，如果机器联上网的话...。          - TYT，1/1/92
```

```
 */
// 把系统启动时间调整为以 GMT 为标准的时间。
// startup_time 是秒值，因此这里需要把时区分钟值乘上 60。
void adjust_clock()
{
        startup_time += sys_tz.tz_minuteswest*60;
}

// 设置当前进程创建文件属性屏蔽码为 mask & 0777。并返回原屏蔽码。
int sys_umask(int mask)
{
        int old = current->umask;

        current->umask = mask & 0777;
        return (old);
}

```

8.10 程序 8-10 linux/kernel/vsprintf.c

```
 1 /*
 2  *  linux/kernel/vsprintf.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /* vsprintf.c -- Lars Wirzenius & Linus Torvalds. */
 8 /*
 9  * Wirzenius wrote this portably, Torvalds fucked it up :-)
10  */
   // Lars Wirzenius 是 Linus 的好友，在 Helsinki 大学时曾同处一间办公室。在 1991 年夏季开发 Linux
   // 时，Linus 当时对 C 语言还不是很熟悉，还不会使用可变参数列表函数功能。因此 Lars Wirzenius
   // 就为他编写了这段用于内核显示信息的代码。他后来(1998 年)承认在这段代码中有一个 bug，直到
   // 1994 年才有人发现，并予以纠正。这个 bug 是在使用*作为输出域宽度时，忘记递增指针跳过这个星
   // 号了。在本代码中这个 bug 还仍然存在（130 行）。 他的个人主页是 http://liw.iki.fi/liw/
11
12 #include <stdarg.h>       // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了-个
                             // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于
                             // vsprintf、vprintf、vfprintf 函数。
13 #include <string.h>       // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
14
15 /* we use this so that we can do without the ctype library */
   /* 我们使用下面的定义，这样我们就可以不使用 ctype 库了 */
16 #define is_digit(c)     ((c) >= '0' && (c) <= '9')   // 判断字符 c 是否为数字字符。
17
   // 该函数将字符数字串转换成整数。输入是数字串指针的指针，返回是结果数值。另外指针将前移。
18 static int skip_atoi(const char **s)
19 {
20         int i=0;
21
22         while (is_digit(**s))
23                 i = i*10 + *((*s)++) - '0';
24         return i;
25 }
26
   // 这里定义转换类型的各种符号常数。
27 #define ZEROPAD 1               /* pad with zero */         /* 填充零 */
28 #define SIGN    2               /* unsigned/signed long */  /* 无符号/符号长整数 */
29 #define PLUS    4               /* show plus */             /* 显示加 */
30 #define SPACE   8               /* space if plus */         /* 如是加，则置空格 */
31 #define LEFT    16              /* left justified */        /* 左调整 */
32 #define SPECIAL 32              /* 0x */                    /* 0x */
33 #define SMALL   64              /* use 'abcdef' instead of 'ABCDEF' */ /* 使用小写字母 */
34
   // 除操作。输入：n 为被除数，base 为除数；结果：n 为商，函数返回值为余数。
   // 参见 4.5.3 节有关嵌入汇编的信息。
35 #define do_div(n,base) ({ \
36 int __res; \
37 __asm__("divl %4":"=a" (n),"=d" (__res):"0" (n),"1" (0),"r" (base)); \
```

```
38 __res; })
39
   // 将整数转换为指定进制的字符串。
   // 输入：num-整数；base-进制；size-字符串长度；precision-数字长度(精度)；type-类型选项。
   // 输出：数字转换成字符串后指向该字符串末端后面的指针。
40 static char * number(char * str, int num, int base, int size, int precision
41         ,int type)
42 {
43         char c,sign,tmp[36];
44         const char *digits="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
45         int i;
46
   // 如果类型 type 指出用小写字母，则定义小写字母集。
   // 如果类型指出要左调整（靠左边界），则屏蔽类型中的填零标志。
   // 如果进制基数小于 2 或大于 36，则退出处理，也即本程序只能处理基数在 2-32 之间的数。
47         if (type&SMALL) digits="0123456789abcdefghijklmnopqrstuvwxyz";
48         if (type&LEFT) type &= ~ZEROPAD;
49         if (base<2 || base>36)
50                 return 0;
   // 如果类型指出要填零，则置字符变量 c='0'，否则 c 等于空格字符。
   // 如果类型指出是带符号数并且数值 num 小于 0，则置符号变量 sign=负号，并使 num 取绝对值。
   // 否则如果类型指出是加号，则置 sign=加号，否则若类型带空格标志则 sign=空格，否则置 0。
51         c = (type & ZEROPAD) ? '0' : ' ';
52         if (type&SIGN && num<0) {
53                 sign='-';
54                 num = -num;
55         } else
56                 sign=(type&PLUS) ? '+' : ((type&SPACE) ? ' ' : 0);
   // 若带符号，则宽度值减 1。若类型指出是特殊转换，则对于十六进制宽度再减少 2 位(用于 0x)，
   // 对于八进制宽度减 1（用于八进制转换结果前放一个零）。
57         if (sign) size--;
58         if (type&SPECIAL)
59                 if (base==16) size -= 2;
60                 else if (base==8) size--;
   // 如果数值 num 为 0，则临时字符串='0'；否则根据给定的基数将数值 num 转换成字符形式。
61         i=0;
62         if (num==0)
63                 tmp[i++]='0';
64         else while (num!=0)
65                 tmp[i++]=digits[do_div(num,base)];
   // 若数值字符个数大于精度值，则精度值扩展为数字个数值。
   // 宽度值 size 减去用于存放数值字符的个数。
66         if (i>precision) precision=i;
67         size -= precision;

   // 从这里真正开始形成所需要的转换结果，并暂时放在字符串 str 中。
   // 若类型中没有填零(ZEROPAD)和左靠齐（左调整）标志，则在 str 中首先
   // 填放剩余宽度值指出的空格数。若需带符号位，则存入符号。
68         if (!(type&(ZEROPAD+LEFT)))
69                 while(size-->0)
70                         *str++ = ' ';
71         if (sign)
72                 *str++ = sign;
```

```
    // 若类型指出是特殊转换，则对于八进制转换结果头一位放置一个'0'；而对于十六进制则存放'0x'。
73          if (type&SPECIAL)
74                  if (base==8)
75                          *str++ = '0';
76                  else if (base==16) {
77                          *str++ = '0';
78                          *str++ = digits[33];    // 'X'或'x'
79                  }
    // 若类型中没有左调整（左靠齐）标志，则在剩余宽度中存放 c 字符（'0'或空格），见 51 行。
80          if (!(type&LEFT))
81                  while(size-->0)
82                          *str++ = c;
    // 此时 i 存有数值 num 的数字个数。若数字个数小于精度值，则 str 中放入（精度值-i）个'0'。
83          while(i<precision--)
84                  *str++ = '0';
    // 将数值转换好的数字字符填入 str 中。共 i 个。
85          while(i-->0)
86                  *str++ = tmp[i];
    // 若宽度值仍大于零，则表示类型标志中有左靠齐标志。则在剩余宽度中放入空格。
87          while(size-->0)
88                  *str++ = ' ';
89          return str;     // 返回转换好的指向字符串末端后的指针。
90  }
91
    // 下面函数是送格式化输出到字符串中。
    // 为了能在内核中使用格式化的输出，Linus 在内核实现了该 C 标准函数。
    // 其中参数 fmt 是格式字符串；args 是个数变化的值；buf 是输出字符串缓冲区。
    // 请参见本代码列表后的有关格式转换字符的介绍。
92  int vsprintf(char *buf, const char *fmt, va_list args)
93  {
94          int len;
95          int i;
96          char * str;             // 用于存放转换过程中的字符串。
97          char *s;
98          int *ip;
99
100         int flags;              /* flags to number() */
101                                 /* number()函数使用的标志 */
102         int field_width;        /* width of output field */
                                    /* 输出字段宽度*/
103         int precision;          /* min. # of digits for integers; max
104                                    number of chars for from string */
                                    /* min. 整数数字个数；max. 字符串中字符个数 */
105         int qualifier;          /* 'h', 'l', or 'L' for integer fields */
106                                 /* 'h', 'l',或'L'用于整数字段 */
    // 首先将字符指针指向 buf，然后扫描格式字符串，对各个格式转换指示进行相应的处理。
107         for (str=buf ; *fmt ; ++fmt) {
    // 格式转换指示字符串均以'%'开始，这里从 fmt 格式字符串中扫描'%'，寻找格式转换字符串的开始。
    // 不是格式指示的一般字符均被依次存入 str。
108                 if (*fmt != '%') {
109                         *str++ = *fmt;
110                         continue;
111                 }
```

```
112
    // 下面取得格式指示字符串中的标志域，并将标志常量放入 flags 变量中。
113                 /* process flags */
114                 flags = 0;
115                 repeat:
116                         ++fmt;          /* this also skips first '%' */
117                         switch (*fmt) {
118                                 case '-': flags |= LEFT; goto repeat;    // 左靠齐调整。
119                                 case '+': flags |= PLUS; goto repeat;    // 放加号。
120                                 case ' ': flags |= SPACE; goto repeat;   // 放空格。
121                                 case '#': flags |= SPECIAL; goto repeat; // 是特殊转换。
122                                 case '0': flags |= ZEROPAD; goto repeat;  // 要填零(即'0')。
123                                 }
124
    // 取当前参数字段宽度域值，放入 field_width 变量中。如果宽度域中是数值则直接取其为宽度值。
    // 如果宽度域中是字符'*'，表示下一个参数指定宽度。因此调用 va_arg 取宽度值。若此时宽度值
    // 小于 0，则该负数表示其带有标志域'-'标志（左靠齐），因此还需在标志变量中添入该标志，并
    // 将字段宽度值取为其绝对值。
125                 /* get field width */
126                 field_width = -1;
127                 if (is_digit(*fmt))
128                         field_width = skip_atoi(&fmt);
129                 else if (*fmt == '*') {
130                         /* it's the next argument */     // 这里有个 bug，应插入++fmt;
131                         field_width = va_arg(args, int);
132                         if (field_width < 0) {
133                                 field_width = -field_width;
134                                 flags |= LEFT;
135                         }
136                 }
137
    // 下面这段代码，取格式转换串的精度域，并放入 precision 变量中。精度域开始的标志是'.'。
    // 其处理过程与上面宽度域的类似。如果精度域中是数值则直接取其为精度值。如果精度域中是
    // 字符'*'，表示下一个参数指定精度。因此调用 va_arg 取精度值。若此时宽度值小于 0，则将
    // 字段精度值取为 0。
138                 /* get the precision */
139                 precision = -1;
140                 if (*fmt == '.') {
141                         ++fmt;
142                         if (is_digit(*fmt))
143                                 precision = skip_atoi(&fmt);
144                         else if (*fmt == '*') {
145                                 /* it's the next argument */   // 同上这里也应插入++fmt;
146                                 precision = va_arg(args, int);
147                         }
148                         if (precision < 0)
149                                 precision = 0;
150                 }
151
    // 下面这段代码分析长度修饰符，并将其存入 qualifer 变量。（h,l,L 的含义参见列表后的说明）。
152                 /* get the conversion qualifier */
153                 qualifier = -1;
154                 if (*fmt == 'h' || *fmt == 'l' || *fmt == 'L') {
```

```
155                     qualifier = *fmt;
156                     ++fmt;
157             }
158
    // 下面分析转换指示符。
159             switch (*fmt) {
    // 如果转换指示符是'c'，则表示对应参数应是字符。此时如果标志域表明不是左靠齐，则该字段前面
    // 放入'宽度域值-1'个空格字符，然后再放入参数字符。如果宽度域还大于 0，则表示为左靠齐，则在
    // 参数字符后面添加'宽度值-1'个空格字符。
160             case 'c':
161                     if (!(flags & LEFT))
162                             while (--field_width > 0)
163                                     *str++ = ' ';
164                     *str++ = (unsigned char) va_arg(args, int);
165                     while (--field_width > 0)
166                             *str++ = ' ';
167                     break;
168
    // 如果转换指示符是's'，则表示对应参数是字符串。首先取参数字符串的长度，若其超过了精度域值，
    // 则扩展精度域=字符串长度。此时如果标志域表明不是左靠齐，则该字段前放入'宽度值-字符串长度'
    // 个空格字符。然后再放入参数字符串。如果宽度域还大于 0，则表示为左靠齐，则在参数字符串后面
    // 添加'宽度值-字符串长度'个空格字符。
169             case 's':
170                     s = va_arg(args, char *);
171                     len = strlen(s);
172                     if (precision < 0)
173                             precision = len;
174                     else if (len > precision)
175                             len = precision;
176
177                     if (!(flags & LEFT))
178                             while (len < field_width--)
179                                     *str++ = ' ';
180                     for (i = 0; i < len; ++i)
181                             *str++ = *s++;
182                     while (len < field_width--)
183                             *str++ = ' ';
184                     break;
185
    // 如果格式转换符是'o'，表示需将对应的参数转换成八进制数的字符串。调用 number()函数处理。
186             case 'o':
187                     str = number(str, va_arg(args, unsigned long), 8,
188                             field_width, precision, flags);
189                     break;
190
    // 如果格式转换符是'p'，表示对应参数是一个指针类型。此时若该参数没有设置宽度域，则默认宽度
    // 为 8，并且需要添零。然后调用 number()函数进行处理。
191             case 'p':
192                     if (field_width == -1) {
193                             field_width = 8;
194                             flags |= ZEROPAD;
195                     }
196                     str = number(str,
```

```
197                             (unsigned long) va_arg(args, void *), 16,
198                             field_width, precision, flags);
199                     break;
200
    // 若格式转换指示是'x'或'X'，则表示对应参数需要打印成十六进制数输出。'x'表示用小写字母表示。
201                 case 'x':
202                     flags |= SMALL;
203                 case 'X':
204                     str = number(str, va_arg(args, unsigned long), 16,
205                             field_width, precision, flags);
206                     break;
207
    // 如果格式转换字符是'd','i'或'u'，则表示对应参数是整数，'d', 'i'代表符号整数，因此需要加上
    // 带符号标志。'u'代表无符号整数。
208                 case 'd':
209                 case 'i':
210                     flags |= SIGN;
211                 case 'u':
212                     str = number(str, va_arg(args, unsigned long), 10,
213                             field_width, precision, flags);
214                     break;
215
    // 若格式转换指示符是'n'，则表示要把到目前为止转换输出字符数保存到对应参数指针指定的位置中。
    // 首先利用 va_arg()取得该参数指针，然后将已经转换好的字符数存入该指针所指的位置。
216                 case 'n':
217                     ip = va_arg(args, int *);
218                     *ip = (str - buf);
219                     break;
220
    // 若格式转换符不是'%'，则表示格式字符串有错，直接将一个'%'写入输出串中。
    // 如果格式转换符的位置处还有字符，则也直接将该字符写入输出串中，并返回到 107 行继续处理
    // 格式字符串。否则表示已经处理到格式字符串的结尾处，则退出循环。
221                 default:
222                     if (*fmt != '%')
223                         *str++ = '%';
224                     if (*fmt)
225                         *str++ = *fmt;
226                     else
227                         --fmt;
228                     break;
229                 }
230         }
231         *str = '\0';            // 最后在转换好的字符串结尾处添上 null。
232         return str-buf;         // 返回转换好的字符串长度值。
233 }
234
```

8.11 程序 8-11 linux/kernel/printk.c

```
 1 /*
 2  *  linux/kernel/printk.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * When in kernel-mode, we cannot use printf, as fs is liable to
 9  * point to 'interesting' things. Make a printf with fs-saving, and
10  * all is well.
11  */
   /*
    * 当处于内核模式时，我们不能使用 printf，因为寄存器 fs 指向其他不感兴趣
    * 的地方。自己编制一个 printf 并在使用前保存 fs，一切就解决了。
    */
   // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了-个类型(va_list)和三个宏
   // va_start、va_arg 和 va_end，用于 vsprintf、vprintf、vfprintf 函数。
12 #include <stdarg.h>
13 #include <stddef.h>       // 标准定义头文件。定义了 NULL, offsetof(TYPE, MEMBER)。
14
15 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
16
17 static char buf[1024];    // 显示用临时缓冲区。
18
   // 函数 vsprintf()定义在 linux/kernel/vsprintf.c 中 92 行开始处。
19 extern int vsprintf(char * buf, const char * fmt, va_list args);
20
   // 内核使用的显示函数。
21 int printk(const char *fmt, ...)
22 {
23         va_list args;                   // va_list 实际上是一个字符指针类型。
24         int i;
25
   // 运行参数处理开始函数。然后使用格式串 fmt 将参数列表 args 输出到 buf 中。返回值 i
   // 等于输出字符串的长度。再运行参数处理结束函数。最后调用控制台显示函数并返回显示
   // 字符数。
26         va_start(args, fmt);
27         i=vsprintf(buf,fmt,args);
28         va_end(args);
29         console_print(buf);             // chr_drv/console.c，第 995 行开始。
30         return i;
31 }
32
```

8.12 程序 8-12 linux/kernel/panic.c

```
 1 /*
 2  *  linux/kernel/panic.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * This function is used through-out the kernel (includeinh mm and fs)
 9  * to indicate a major problem.
10  */
   /*
    * 该函数在整个内核中使用（包括在 头文件*.h, 内存管理程序 mm 和文件系统 fs 中），
    * 用以指出主要的出错问题。
    */
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
13
14 void sys_sync(void);    /* it's really int */ /* 实际上是整型 int (fs/buffer.c,44) */
15
   // 该函数用来显示内核中出现的重大错误信息，并运行文件系统同步函数，然后进入死循环--死机。
   // 如果当前进程是任务 0 的话，还说明是交换任务出错，并且还没有运行文件系统同步函数。
   // 函数名前的关键字 volatile 用于告诉编译器 gcc 该函数不会返回。这样可让 gcc 产生更好一些的
   // 代码，更重要的是使用这个关键字可以避免产生某些（未初始化变量的）假警告信息。
   // 等同于现在 gcc 的函数属性说明：void panic(const char *s) __attribute__ ((noreturn));
16 volatile void panic(const char * s)
17 {
18         printk("Kernel panic: %s\n\r",s);
19         if (current == task[0])
20                 printk("In swapper task - not syncing\n\r");
21         else
22                 sys_sync();
23         for(;;);
24 }
25
```

第9章 内核块设备程序

9.1 程序 9-1 linux/kernel/blk_drv/blk.h

```
 1 #ifndef _BLK_H
 2 #define _BLK_H
 3
 4 #define NR_BLK_DEV      7           // 块设备类型数量。
 5 /*
 6  * NR_REQUEST is the number of entries in the request-queue.
 7  * NOTE that writes may use only the low 2/3 of these: reads
 8  * take precedence.
 9  *
10  * 32 seems to be a reasonable number: enough to get some benefit
11  * from the elevator-mechanism, but not so much as to lock a lot of
12  * buffers when they are in the queue. 64 seems to be too many (easily
13  * long pauses in reading when heavy writing/syncing is going on)
14  */
    /*
     * 下面定义的 NR_REQUEST 是请求队列中所包含的项数。
     * 注意，写操作仅使用这些项中低端的 2/3 项；读操作优先处理。
     *
     * 32 项好象是一个合理的数字：该数已经足够从电梯算法中获得好处，
     * 但当缓冲区在队列中而锁住时又不显得是很大的数。64 就看上去太
     * 大了（当大量的写/同步操作运行时很容易引起长时间的暂停）。
     */
15 #define NR_REQUEST      32
16
17 /*
18  * Ok, this is an expanded form so that we can use the same
19  * request for paging requests when that is implemented. In
20  * paging, 'bh' is NULL, and 'waiting' is used to wait for
21  * read/write completion.
22  */
    /*
     * OK，下面是 request 结构的一个扩展形式，因而当实现以后，我们
     * 就可以在分页请求中使用同样的 request 结构。 在分页处理中，
     * 'bh'是 NULL，而'waiting'则用于等待读/写的完成。
     */
    // 下面是请求队列中项的结构。其中如果字段 dev = -1，则表示队列中该项没有被使用。
    // 字段 cmd 可取常量 READ（0）或 WRITE（1）（定义在 include/linux/fs.h 中）。
    // 其中，内核并没有用到 waiting 指针，起而代之地内核使用了缓冲块的等待队列。因为
    // 等待一个缓冲块与等待请求项完成是对等的。
23 struct request {
24         int dev;                    /* -1 if no request */  // 发请求的设备号。
25         int cmd;                    /* READ or WRITE */     // READ 或 WRITE 命令。
26         int errors;                  //操作时产生的错误次数。
27         unsigned long sector;        // 起始扇区。(1 块=2 扇区)
28         unsigned long nr_sectors;    // 读/写扇区数。
29         char * buffer;               // 数据缓冲区。
```

```
30          struct task_struct * waiting;   // 任务等待请求完成操作的地方（队列）。
31          struct buffer_head * bh;        // 缓冲区头指针(include/linux/fs.h,68)。
32          struct request * next;          // 指向下一请求项。
33  };
34
35  /*
36   * This is used in the elevator algorithm: Note that
37   * reads always go before writes. This is natural: reads
38   * are much more time-critical than writes.
39   */
    /*
     * 下面的定义用于电梯算法：注意读操作总是在写操作之前进行。
     * 这是很自然的：读操作对时间的要求要比写操作严格得多。
     */
    // 下面宏中参数 s1 和 s2 的取值是上面定义的请求结构 request 的指针。该宏定义用于根据两个参数
    // 指定的请求项结构中的信息（命令 cmd（READ 或 WRITE）、设备号 dev 以及所操作的扇区号 sector）
    // 来判断出两个请求项结构的前后排列顺序。这个顺序将用作访问块设备时的请求项执行顺序。
    // 这个宏会在程序 blk_drv/ll_rw_blk.c 中函数 add_request()中被调用（第 96 行）。该宏部分
    // 地实现了 I/O 调度功能，即实现了对请求项的排序功能（另一个是请求项合并功能）。
40  #define IN_ORDER(s1,s2) \
41  ((s1)->cmd<(s2)->cmd || (s1)->cmd==(s2)->cmd && \
42  ((s1)->dev < (s2)->dev || ((s1)->dev == (s2)->dev && \
43  (s1)->sector < (s2)->sector)))
44
    // 块设备处理结构。
45  struct blk_dev_struct {
46          void (*request_fn)(void);                   // 请求处理函数指针。
47          struct request * current_request;           // 当前处理的请求结构。
48  };
49
    // 块设备表（数组）。每种块设备占用一项，共 7 项。
50  extern struct blk_dev_struct blk_dev[NR_BLK_DEV];
    // 请求队列数组，共 32 项。
51  extern struct request request[NR_REQUEST];
    // 等待空闲请求项的进程队列头指针。
52  extern struct task_struct * wait_for_request;
53
    // 设备数据块总数指针数组。每个指针项指向指定主设备号的总块数数组 hd_sizes[]。该总
    // 块数数组每一项对应子设备号确定的一个子设备上所拥有的数据块总数（1 块大小 = 1KB）。
54  extern int * blk_size[NR_BLK_DEV];
55
    // 在块设备驱动程序（如 hd.c）包含此头文件时，必须先定义驱动程序处理设备的主设备号。
    // 这样，在下面 63 行—90 行就能为包含本文件的驱动程序给出正确的宏定义。
56  #ifdef MAJOR_NR                     // 主设备号。
57
58  /*
59   * Add entries as needed. Currently the only block devices
60   * supported are hard-disks and floppies.
61   */
    /*
     * 需要时加入条目。目前块设备仅支持硬盘和软盘（还有虚拟盘）。
     */
62
```

```
    // 如果定义了 MAJOR_NR = 1（RAM 盘主设备号），就是用以下符号常数和宏。
63  #if (MAJOR_NR == 1)
64  /* ram disk */
65  #define DEVICE_NAME "ramdisk"                   // 设备名称（“内存虚拟盘”）。
66  #define DEVICE_REQUEST do_rd_request            // 设备请求项处理函数。
67  #define DEVICE_NR(device) ((device) & 7)        // 设备号（0 - 7）。
68  #define DEVICE_ON(device)                       // 开启设备（虚拟盘无须开启和关闭）。
69  #define DEVICE_OFF(device)                      // 关闭设备。
70
    // 否则，如果定义了 MAJOR_NR = 2（软驱主设备号），就是用以下符号常数和宏。
71  #elif (MAJOR_NR == 2)
72  /* floppy */
73  #define DEVICE_NAME "floppy"                    // 设备名称（“软盘驱动器”）。
74  #define DEVICE_INTR do_floppy                   // 设备中断处理函数。
75  #define DEVICE_REQUEST do_fd_request            // 设备请求项处理函数。
76  #define DEVICE_NR(device) ((device) & 3)        // 设备号（0 - 3）。
77  #define DEVICE_ON(device) floppy_on(DEVICE_NR(device))    // 开启设备宏。
78  #define DEVICE_OFF(device) floppy_off(DEVICE_NR(device))  // 关闭设备宏。
79
    // 否则，如果定义了 MAJOR_NR = 3（硬盘主设备号），就是用以下符号常数和宏。
80  #elif (MAJOR_NR == 3)
81  /* harddisk */
82  #define DEVICE_NAME "harddisk"                  // 设备名称（“硬盘”）。
83  #define DEVICE_INTR do_hd                       // 设备中断处理函数。
84  #define DEVICE_TIMEOUT hd_timeout               // 设备超时值。
85  #define DEVICE_REQUEST do_hd_request            // 设备请求项处理函数。
86  #define DEVICE_NR(device) (MINOR(device)/5)     // 设备号。
87  #define DEVICE_ON(device)                       // 开启设备。
88  #define DEVICE_OFF(device)                      // 关闭设备。
89
    // 否则在编译预处理阶段显示出错信息：“未知块设备”。
90  #elif
91  /* unknown blk device */
92  #error "unknown blk device"
93
94  #endif
95
    // 为了便于编程表示，这里定义了两个宏：CURRENT 是指定住设备号的当前请求结构项指针，
    // CURRENT_DEV 是当前请求项 CURRENT 中设备号。
96  #define CURRENT (blk_dev[MAJOR_NR].current_request)
97  #define CURRENT_DEV DEVICE_NR(CURRENT->dev)
98
    // 如果定义了设备中断处理符号常数，则把它声明为一个函数指针，并默认为 NULL。
99  #ifdef DEVICE_INTR
100 void (*DEVICE_INTR)(void) = NULL;
101 #endif
    // 如果定义了设备超时符号常数，则令其值等于 0，并定义 SET_INTR()宏。否则只定义宏。
102 #ifdef DEVICE_TIMEOUT
103 int DEVICE_TIMEOUT = 0;
104 #define SET_INTR(x) (DEVICE_INTR = (x),DEVICE_TIMEOUT = 200)
105 #else
106 #define SET_INTR(x) (DEVICE_INTR = (x))
107 #endif
```

```
    // 声明设备请求符号常数 DEVICE_REGUEST 是一个不带参数并无反回的静态函数指针。
108 static void (DEVICE_REQUEST)(void);
109
    // 解锁指定的缓冲块。
    // 如果指定缓冲块 bh 并没有被上锁，则显示警告信息。否则将该缓冲块解锁，并唤醒等待
    // 该缓冲块的进程。此为内嵌函数。参数是缓冲块头指针。
110 extern inline void unlock_buffer(struct buffer_head * bh)
111 {
112         if (!bh->b_lock)
113                 printk(DEVICE_NAME ": free buffer being unlocked\n");
114         bh->b_lock=0;
115         wake_up(&bh->b_wait);
116 }
117
    // 结束请求处理。
    // 参数 uptodate 是更新标志。
    // 首先关闭指定块设备，然后检查此次读写缓冲区是否有效。如果有效则根据参数值设置缓冲
    // 区数据更新标志，并解锁该缓冲区。 如果更新标志参数值是 0，表示此次请求项的操作已失
    // 败，因此显示相关块设备 IO 错误信息。 最后，唤醒等待该请求项的进程以及等待空闲请求
    // 项出现的进程，释放并从请求链表中删除本请求项，并把当前请求项指针指向下一请求项。
118 extern inline void end_request(int uptodate)
119 {
120         DEVICE_OFF(CURRENT->dev);                       // 关闭设备。
121         if (CURRENT->bh) {                              // CURRENT 为当前请求结构项指针。
122                 CURRENT->bh->b_uptodate = uptodate;     // 置更新标志。
123                 unlock_buffer(CURRENT->bh);             // 解锁缓冲区。
124         }
125         if (!uptodate) {                                // 若更新标志为 0 则显示出错信息。
126                 printk(DEVICE_NAME " I/O error\n\r");
127                 printk("dev %04x, block %d\n\r",CURRENT->dev,
128                         CURRENT->bh->b_blocknr);
129         }
130         wake_up(&CURRENT->waiting);                     // 唤醒等待该请求项的进程。
131         wake_up(&wait_for_request);                     // 唤醒等待空闲请求项的进程。
132         CURRENT->dev = -1;                              // 释放该请求项。
133         CURRENT = CURRENT->next;                        // 指向下一请求项。
134 }
135
    // 如果定义了设备超时符号常量 DEVICE_TIMEOUT，则定义 CLEAR_DEVICE_TIMEOUT 符号常量
    // 为“DEVICE_TIMEOUT = 0”。否则定义 CLEAR_DEVICE_TIMEOUT 为空。
136 #ifdef DEVICE_TIMEOUT
137 #define CLEAR_DEVICE_TIMEOUT DEVICE_TIMEOUT = 0;
138 #else
139 #define CLEAR_DEVICE_TIMEOUT
140 #endif
141
    // 如果定义了设备中断符号常量 DEVICE_INTR，则定义 CLEAR_DEVICE_INTR 符号常量为
    // “DEVICE_INTR = 0”，否则定义其为空。
142 #ifdef DEVICE_INTR
143 #define CLEAR_DEVICE_INTR DEVICE_INTR = 0;
144 #else
145 #define CLEAR_DEVICE_INTR
146 #endif
```

```
147
    // 定义初始化请求项宏。
    // 由于几个块设备驱动程序开始处对请求项的初始化操作相似，因此这里为它们定义了一个
    // 统一的初始化宏。该宏用于对当前请求项进行一些有效性判断。所做工作如下：
    // 如果设备当前请求项为空（NULL），表示本设备目前已无需要处理的请求项。于是略作扫尾
    // 工作就退出相应函数。否则，如果当前请求项中设备的主设备号不等于驱动程序定义的主设
    // 备号，说明请求项队列乱掉了，于是内核显示出错信息并停机。否则若请求项中用的缓冲块
    // 没有被锁定，也说明内核程序出了问题，于是显示出错信息并停机。
148 #define INIT_REQUEST \
149 repeat: \
150         if (!CURRENT) {\                            // 如果当前请求项指针为 NULL 则返回。
151                 CLEAR_DEVICE_INTR \
152                 CLEAR_DEVICE_TIMEOUT \
153                 return; \
154         } \
155         if (MAJOR(CURRENT->dev) != MAJOR_NR) \    // 如果当前设备主设备号不对则停机。
156                 panic(DEVICE_NAME ": request list destroyed"); \
157         if (CURRENT->bh) { \
158                 if (!CURRENT->bh->b_lock) \       // 如果请求项的缓冲区没锁定则停机。
159                         panic(DEVICE_NAME ": block not locked"); \
160         }
161
162 #endif
163
164 #endif
165
```

9.2 程序 9-2 linux/kernel/blk_drv/hd.c

```
 1 /*
 2  *  linux/kernel/hd.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * This is the low-level hd interrupt support. It traverses the
 9  * request-list, using interrupts to jump between functions. As
10  * all the functions are called within interrupts, we may not
11  * sleep. Special care is recommended.
12  *
13  *  modified by Drew Eckhardt to check nr of hd's from the CMOS.
14  */
   /*
    * 本程序是底层硬盘中断辅助程序。主要用于扫描请求项队列，使用中断
    * 在函数之间跳转。由于所有的函数都是在中断里调用的，所以这些函数
    * 不可以睡眠。请特别注意。
    *
    * 由 Drew Eckhardt 修改，利用 CMOS 信息检测硬盘数。
    */
15
16 #include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）选项。
17 #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 数据等。
18 #include <linux/fs.h>     // 文件系统头文件。定义文件表结构（file、m_inode）等。
19 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
20 #include <linux/hdreg.h>  // 硬盘参数头文件。定义硬盘寄存器端口、状态码、分区表等信息。
21 #include <asm/system.h>   // 系统头文件。定义设置或修改描述符/中断门等的汇编宏。
22 #include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
23 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
24
   // 定义硬盘主设备号符号常数。在驱动程序中，主设备号必须在包含 blk.h 文件之前被定义。
   // 因为 blk.h 文件中要用到这个符号常数值来确定一些列其他相关符号常数和宏。
25 #define MAJOR_NR 3        // 硬盘主设备号是 3。
26 #include "blk.h"          // 块设备头文件。定义请求数据结构、块设备数据结构和宏等信息。
27
   // 读 CMOS 参数宏函数。
   // 这段宏读取 CMOS 中硬盘信息。outb_p、inb_p 是 include/asm/io.h 中定义的端口输入输出宏。
   // 与 init/main.c 中读取 CMOS 时钟信息的宏完全一样。
28 #define CMOS_READ(addr) ({ \
29 outb_p(0x80|addr,0x70); \          // 0x70 是写端口号，0x80|addr 是要读的 CMOS 内存地址。
30 inb_p(0x71); \                     // 0x71 是读端口号。
31 })
32
33 /* Max read/write errors/sector */
   /* 每扇区读/写操作允许的最多出错次数 */
34 #define MAX_ERRORS      7         // 读/写一个扇区时允许的最多出错次数。
35 #define MAX_HD          2         // 系统支持的最多硬盘数。
36
```

```
   // 重新校正处理函数。
   // 复位操作时在硬盘中断处理程序中调用的重新校正函数(311 行)。
37 static void recal_intr(void);
   // 读写硬盘失败处理调用函数。
   // 结束本次请求项处理或者设置复位标志要求执行复位硬盘控制器操作后再重试（242 行）。
38 static void bad_rw_intr(void);
39
   // 重新校正标志。当设置了该标志，程序中会调用 recal_intr()以将磁头移动到 0 柱面。
40 static int recalibrate = 0;
  // 复位标志。当发生读写错误时会设置该标志并调用相关复位函数，以复位硬盘和控制器。
41 static int reset = 0;
42
43 /*
44  *  This struct defines the HD's and their types.
45  */
   /* 下面结构定义了硬盘参数及类型 */
   // 硬盘信息结构（Harddisk information struct）。
   // 各字段分别是磁头数、每磁道扇区数、柱面数、写前预补偿柱面号、磁头着陆区柱面号、
   // 控制字节。它们的含义请参见程序列表后的说明。
46 struct hd_i_struct {
47         int head,sect,cyl,wpcom,lzone,ctl;
48         };

   // 如果已经在 include/linux/config.h 配置文件中定义了符号常数 HD_TYPE，就取其中定义
   // 好的参数作为硬盘信息数组 hd_info[]中的数据。否则先默认都设为 0 值，在 setup()函数
   // 中会重新进行设置。
49 #ifdef HD_TYPE
50 struct hd_i_struct hd_info[] = { HD_TYPE };                    // 硬盘信息数组。
51 #define NR_HD ((sizeof (hd_info))/(sizeof (struct hd_i_struct)))  // 计算硬盘个数。
52 #else
53 struct hd_i_struct hd_info[] = { {0,0,0,0,0,0},{0,0,0,0,0,0} };
54 static int NR_HD = 0;
55 #endif
56
   // 定义硬盘分区结构。给出每个分区从硬盘 0 道开始算起的物理起始扇区号和分区扇区总数。
   // 其中 5 的倍数处的项（例如 hd[0]和 hd[5]等）代表整个硬盘的参数。
57 static struct hd_struct {
58         long start_sect;                // 分区在硬盘中的起始物理（绝对）扇区。
59         long nr_sects;                  // 分区中扇区总数。
60 } hd[5*MAX_HD]={{0,0},};
61
   // 硬盘每个分区数据块总数数组。
62 static int hd_sizes[5*MAX_HD] = {0, };
63
   // 读端口嵌入汇编宏。读端口 port，共读 nr 字，保存在 buf 中。
64 #define port_read(port,buf,nr) \
65 __asm__("cld;rep;insw"::"d" (port),"D" (buf),"c" (nr):"cx","di")
66
   // 写端口嵌入汇编宏。写端口 port，共写 nr 字，从 buf 中取数据。
67 #define port_write(port,buf,nr) \
68 __asm__("cld;rep;outsw"::"d" (port),"S" (buf),"c" (nr):"cx","si")
69
```

```
70 extern void hd_interrupt(void);          // 硬盘中断过程（sys_call.s，235 行）。
71 extern void rd_load(void);              // 虚拟盘创建加载函数（ramdisk.c，71 行）。
72
73 /* This may be used only once, enforced by 'static int callable' */
   /* 下面该函数只在初始化时被调用一次。用静态变量 callable 作为可调用标志。*/
   // 系统设置函数。
   // 函数参数 BIOS 是由初始化程序 init/main.c 中 init 子程序设置为指向硬盘参数表结构的指针。
   // 该硬盘参数表结构包含 2 个硬盘参数表的内容（共 32 字节），是从内存 0x90080 处复制而来。
   // 0x90080 处的硬盘参数表是由 setup.s 程序利用 ROM BIOS 功能取得。硬盘参数表信息参见程序
   // 列表后的说明。 本函数主要功能是读取 CMOS 和硬盘参数表信息，用于设置硬盘分区结构 hd，
   // 并尝试加载 RAM 虚拟盘和根文件系统。
74 int sys_setup(void * BIOS)
75 {
76         static int callable = 1;                // 限制本函数只能被调用 1 次的标志。
77         int i,drive;
78         unsigned char cmos_disks;
79         struct partition *p;
80         struct buffer_head * bh;
81
   // 首先设置 callable 标志，使得本函数只能被调用 1 次。然后设置硬盘信息数组 hd_info[]。
   // 如果在 include/linux/config.h 文件中已定义了符号常数 HD_TYPE，那么 hd_info[]数组
   // 已经在前面第 49 行上设置好了。否则就需要读取 boot/setup.s 程序存放在内存 0x90080 处
   // 开始的硬盘参数表。setup.s 程序在内存此处连续存放着一到两个硬盘参数表。
82         if (!callable)
83                 return -1;
84         callable = 0;
85 #ifndef HD_TYPE                                 // 如果没有定义 HD_TYPE，则读取。
86         for (drive=0 ; drive<2 ; drive++) {
87                 hd_info[drive].cyl = *(unsigned short *) BIOS;      // 柱面数。
88                 hd_info[drive].head = *(unsigned char *) (2+BIOS);  // 磁头数。
89                 hd_info[drive].wpcom = *(unsigned short *) (5+BIOS); // 写前预补偿柱面号。
90                 hd_info[drive].ctl = *(unsigned char *) (8+BIOS);    // 控制字节。
91                 hd_info[drive].lzone = *(unsigned short *) (12+BIOS);// 磁头着陆区柱面号。
92                 hd_info[drive].sect = *(unsigned char *) (14+BIOS);  // 每磁道扇区数。
93                 BIOS += 16;             // 每个硬盘参数表长 16 字节，这里 BIOS 指向下一表。
94         }
   // setup.s 程序在取 BIOS 硬盘参数表信息时，如果系统中只有 1 个硬盘，就会将对应第 2 个
   // 硬盘的 16 字节全部清零。因此这里只要判断第 2 个硬盘柱面数是否为 0 就可以知道是否有
   // 第 2 个硬盘了。
95         if (hd_info[1].cyl)
96                 NR_HD=2;                // 硬盘数置为 2。
97         else
98                 NR_HD=1;
99 #endif
   // 到这里，硬盘信息数组 hd_info[]已经设置好，并且确定了系统含有的硬盘数 NR_HD。现在
   // 开始设置硬盘分区结构数组 hd[]。该数组的项 0 和项 5 分别表示两个硬盘的整体参数，而
   // 项 1—4 和 6—9 分别表示两个硬盘的 4 个分区的参数。 因此这里仅设置表示硬盘整体信息
   // 的两项（项 0 和 5）。
100        for (i=0 ; i<NR_HD ; i++) {
101                hd[i*5].start_sect = 0;                         // 硬盘起始扇区号。
102                hd[i*5].nr_sects = hd_info[i].head*
103                                hd_info[i].sect*hd_info[i].cyl;  // 硬盘总扇区数。
104        }
```

```
105
106         /*
107                 We querry CMOS about hard disks : it could be that
108                 we have a SCSI/ESDI/etc controller that is BIOS
109                 compatable with ST-506, and thus showing up in our
110                 BIOS table, but not register compatable, and therefore
111                 not present in CMOS.
112
113                 Furthurmore, we will assume that our ST-506 drives
114                 <if any> are the primary drives in the system, and
115                 the ones reflected as drive 1 or 2.
116
117                 The first drive is stored in the high nibble of CMOS
118                 byte 0x12, the second in the low nibble.  This will be
119                 either a 4 bit drive type or 0xf indicating use byte 0x19
120                 for an 8 bit type, drive 1, 0x1a for drive 2 in CMOS.
121
122                 Needless to say, a non-zero value means we have
123                 an AT controller hard disk for that drive.
124
125
126         */
          /*
                  我们对 CMOS 有关硬盘的信息有些怀疑：可能会出现这样的情况，
                  我们有一块 SCSI/ESDI/等的控制器，它是以 ST-506 方式与 BIOS
                  相兼容的，因而会出现在我们的 BIOS 参数表中，但却又不是寄存
                  器兼容的，因此这些参数在 CMOS 中又不存在。

                  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，也即以驱动器 1 或 2 出现的驱动器。

                  第 1 个驱动器参数存放在 CMOS 字节 0x12 的高半字节中，第 2 个
                  存放在低半字节中。该 4 位字节信息可以是驱动器类型，也可能
                  仅是 0xf。0xf 表示使用 CMOS 中 0x19 字节作为驱动器 1 的 8 位
                  类型字节，使用 CMOS 中 0x1A 字节作为驱动器 2 的类型字节。

                  总之，一个非零值意味着硬盘是一个 AT 控制器兼容硬盘。
           */
127
    // 这里根据上述原理，下面代码用来检测硬盘到底是不是 AT 控制器兼容的。有关 CMOS 信息
    // 请参见第 4 章中 4.2.3.1 节。这里从 CMOS 偏移地址 0x12 处读出硬盘类型字节。如果低半
    // 字节值（存放着第 2 个硬盘类型值）不为 0，则表示系统有两硬盘，否则表示系统只有 1
    // 个硬盘。如果 0x12 处读出的值为 0，则表示系统中没有 AT 兼容硬盘。
128         if ((cmos_disks = CMOS_READ(0x12)) & 0xf0)
129                 if (cmos_disks & 0x0f)
130                         NR_HD = 2;
131                 else
132                         NR_HD = 1;
133         else
134                 NR_HD = 0;
    // 若 NR_HD = 0，则两个硬盘都不是 AT 控制器兼容的，两个硬盘数据结构全清零。
    // 若 NR_HD = 1，则将第 2 个硬盘的参数清零。
135         for (i = NR_HD ; i < 2 ; i++) {
```

```
136                 hd[i*5].start_sect = 0;
137                 hd[i*5].nr_sects = 0;
138         }
    // 好，到此为止我们已经真正确定了系统中所含的硬盘个数 NR_HD。现在我们来读取每个硬盘
    // 上第 1 个扇区中的分区表信息，用来设置分区结构数组 hd[] 中硬盘各分区的信息。首先利
    // 用读块函数 bread()读硬盘第 1 个数据块（fs/buffer.c，第 267 行），第 1 个参数（0x300、
    // 0x305 ）分别是两个硬盘的设备号，第 2 个参数（0）是所需读取的块号。若读操作成功，
    // 则数据会被存放在缓冲块 bh 的数据区中。若缓冲块头指针 bh 为 0，则说明读操作失败，则
    // 显示出错信息并停机。否则我们根据硬盘第 1 个扇区最后两个字节应该是 0xAA55 来判断扇
    // 区中数据的有效性，从而可以知道扇区中位于偏移 0x1BE 开始处的分区表是否有效。若有效
    // 则将硬盘分区表信息放入硬盘分区结构数组 hd[]中。最后释放 bh 缓冲区。
139         for (drive=0 ; drive<NR_HD ; drive++) {
140                 if (!(bh = bread(0x300 + drive*5,0))) {  // 0x300、0x305 是设备号。
141                         printk("Unable to read partition table of drive %d\n\r",
142                                 drive);
143                         panic("");
144                 }
145                 if (bh->b_data[510] != 0x55 || (unsigned char)
146                     bh->b_data[511] != 0xAA) {            // 判断硬盘标志 0xAA55。
147                         printk("Bad partition table on drive %d\n\r",drive);
148                         panic("");
149                 }
150                 p = 0x1BE + (void *)bh->b_data;     // 分区表位于第 1 扇区 0x1BE 处。
151                 for (i=1;i<5;i++,p++) {
152                         hd[i+5*drive].start_sect = p->start_sect;
153                         hd[i+5*drive].nr_sects = p->nr_sects;
154                 }
155                 brelse(bh);                 // 释放为存放硬盘数据块而申请的缓冲区。
156         }
    // 现在再对每个分区中的数据块总数进行统计，并保存在硬盘分区总数据块数组 hd_sizes[]中。
    // 然后让设备数据块总数指针数组的本设备项指向该数组。
157         for (i=0 ; i<5*MAX_HD ; i++)
158                 hd_sizes[i] = hd[i].nr_sects>>1 ;
159         blk_size[MAJOR_NR] = hd_sizes;
    // 现在总算完成设置硬盘分区结构数组 hd[]的任务。如果确实有硬盘存在并且已读入其分区
    // 表，则显示“分区表正常”信息。然后尝试在系统内存虚拟盘中加载启动盘中包含的根文
    // 件系统映像（blk_drv/ramdisk.c，第 71 行）。即在系统设置有虚拟盘的情况下判断启动盘
    // 上是否还含有根文件系统的映像数据。如果有（此时该启动盘称为集成盘）则尝试把该映像
    // 加载并存放到虚拟盘中，然后把此时的根文件系统设备号 ROOT_DEV 修改成虚拟盘的设备号。
    // 接着再对交换设备进行初始化。最后安装根文件系统。
160         if (NR_HD)
161                 printk("Partition table%s ok.\n\r",(NR_HD>1)?"s":"");
162         rd_load();                          // blk_drv/ramdisk.c，第 71 行。
163         init_swapping();                    // mm/swap.c，第 199 行。
164         mount_root();                       // fs/super.c，第 241 行。
165         return (0);
166 }
167
    //// 判断并循环等待硬盘控制器就绪。
    // 读硬盘控制器状态寄存器端口 HD_STATUS(0x1f7)，循环检测其中的驱动器就绪比特位（位 6）
    // 是否被置位并且控制器忙位（位 7）是否被复位。 如果返回值 retries 为 0，则表示等待控制
    // 器空闲的时间已经超时而发生错误，若返回值不为 0 则说明在等待（循环）时间期限内控制器
    // 回到空闲状态，OK!
```

```
    // 实际上，我们仅需检测状态寄存器忙位（位 7）是否为 1 来判断控制器是否处于忙状态，驱动
    // 器是否就绪（即位 6 是否为 1）与控制器的状态无关。因此我们可以把第 172 行语句改写成：
    // “while (--retries && (inb_p(HD_STATUS)&0x80));”另外，由于现在的 PC 机速度都很快，
    // 因此我们可以把等待的循环次数再加大一些，例如再增加 10 倍！
168 static int controller_ready(void)
169 {
170         int retries = 100000;
171
172         while (--retries && (inb_p(HD_STATUS)&0xc0)!=0x40);
173         return (retries);                       // 返回等待循环次数。
174 }
175
    //// 检测硬盘执行命令后的状态。（win 表示温切斯特硬盘的缩写）
    // 读取状态寄存器中的命令执行结果状态。 返回 0 表示正常；1 表示出错。如果执行命令错，
    // 则需要再读错误寄存器 HD_ERROR（0x1f1）。
176 static int win_result(void)
177 {
178         int i=inb_p(HD_STATUS);             // 取状态信息。
179
180         if ((i & (BUSY_STAT | READY_STAT | WRERR_STAT | SEEK_STAT | ERR_STAT))
181                 == (READY_STAT | SEEK_STAT))
182                 return(0); /* ok */
183         if (i&1) i=inb(HD_ERROR);           // 若 ERR_STAT 置位，则读取错误寄存器。
184         return (1);
185 }
186
    //// 向硬盘控制器发送命令块。
    // 参数：drive - 硬盘号(0-1)；nsect - 读写扇区数；sect  - 起始扇区；
    //       head  - 磁头号；      cyl   - 柱面号；     cmd   - 命令码（见控制器命令列表）；
    //       intr_addr() - 硬盘中断处理程序中将调用的 C 处理函数指针。
    // 该函数在硬盘控制器就绪之后，先设置全局指针变量 do_hd 为硬盘中断处理程序中将调用的
    // C 处理函数指针。然后再发送硬盘控制字节和 7 字节的参数命令块。
    // 该函数在硬盘控制器就绪之后，先设置全局函数指针变量 do_hd 指向硬盘中断处理程序中将会
    // 调用的 C 处理函数，然后再发送硬盘控制字节和 7 字节的参数命令块。硬盘中断处理程序的代
    // 码位于 kernel/sys_call.s 程序第 235 行处。
    // 第 191 行定义 1 个寄存器变量__res。该变量将被保存在 1 个寄存器中，以便于快速访问。
    // 如果想指定寄存器（如 eax），则我们可以把该句写成“register char __res asm("ax");”。
187 static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
188                 unsigned int head,unsigned int cyl,unsigned int cmd,
189                 void (*intr_addr)(void))
190 {
191         register int port asm("dx");    // 定义局部寄存器变量并放在指定寄存器 dx 中。
192
    // 首先对参数进行有效性检查。如果驱动器号大于 1（只能是 0、1）或者磁头号大于 15，则程
    // 序不支持，停机。否则就判断并循环等待驱动器就绪。如果等待一段时间后仍未就绪则表示
    // 硬盘控制器出错，也停机。
193         if (drive>1 || head>15)
194                 panic("Trying to write bad sector");
195         if (!controller_ready())
196                 panic("HD controller not ready");
    // 接着我们设置硬盘中断发生时将调用的 C 函数指针 do_hd（该函数指针定义在 blk.h 文件的
    // 第 56--109 行之间，请特别留意其中的第 83 行和 100 行）。然后在向硬盘控制器发送参数
    // 和命令之前，规定要先向控制器命令端口（0x3f6）发送一指定硬盘的控制字节，以建立相
```

```
    // 应的硬盘控制方式。该控制字节即是硬盘信息结构数组中的 ctl 字段。然后向控制器端口
    // 0x1f1 - 0x1f7 发送 7 字节的参数命令块。
197         SET_INTR(intr_addr);                    // do_hd = intr_addr 在中断中被调用。
198         outb_p(hd_info[drive].ctl,HD_CMD);      // 向控制寄存器输出控制字节。
199         port=HD_DATA;                           // 置 dx 为数据寄存器端口(0x1f0)。
200         outb_p(hd_info[drive].wpcom>>2,++port); // 参数：写预补偿柱面号(需除 4)。
201         outb_p(nsect,++port);                   // 参数：读/写扇区总数。
202         outb_p(sect,++port);                    // 参数：起始扇区。
203         outb_p(cyl,++port);                     // 参数：柱面号低 8 位。
204         outb_p(cyl>>8,++port);                  // 参数：柱面号高 8 位。
205         outb_p(0xA0|(drive<<4)|head,++port);    // 参数：驱动器号+磁头号。
206         outb(cmd,++port);                       // 命令：硬盘控制命令。
207 }
208
    //// 等待硬盘就绪。
    // 该函数循环等待主状态控制器忙标志位复位。若仅有就绪或寻道结束标志置位，则表示硬盘
    // 就绪，成功返回 0。若经过一段时间仍为忙，则返回 1。
209 static int drive_busy(void)
210 {
211         unsigned int i;
212         unsigned char c;
213
    // 循环读取控制器的主状态寄存器 HD_STATUS，等待就绪标志位置位并且忙位复位。然后检测
    // 其中忙位、就绪位和寻道结束位。若仅有就绪或寻道结束标志置位，则表示硬盘就绪，返回
    // 0。否则表示等待超时。于是警告显示信息。并返回 1。
214         for (i = 0; i < 50000; i++) {
215                 c = inb_p(HD_STATUS);                   // 取主控制器状态字节。
216                 c &= (BUSY_STAT | READY_STAT | SEEK_STAT);
217                 if (c == (READY_STAT | SEEK_STAT))
218                         return 0;
219         }
220         printk("HD controller times out\n\r");  // 等待超时，显示信息。并返回 1。
221         return(1);
222 }
223
    //// 诊断复位（重新校正）硬盘控制器。
    // 首先向控制寄存器端口（0x3f6）发送允许复位（4）控制字节。然后循环空操作等待一段时
    // 间让控制器执行复位操作。接着再向该端口发送正常的控制字节(不禁止重试、重读)，并等
    // 待硬盘就绪。若等待硬盘就绪超时，则显示警告信息。然后读取错误寄存器内容，若其不等
    // 于 1（表示无错误）则显示硬盘控制器复位失败信息。
224 static void reset_controller(void)
225 {
226         int     i;
227
228         outb(4,HD_CMD);                         // 向控制寄存器端口发送复位控制字节。
229         for(i = 0; i < 1000; i++) nop();        // 等待一段时间。
230         outb(hd_info[0].ctl & 0x0f ,HD_CMD);    // 发送正常控制字节(不禁止重试、重读)。
231         if (drive_busy())
232                 printk("HD-controller still busy\n\r");
233         if ((i = inb(HD_ERROR)) != 1)
234                 printk("HD-controller reset failed: %02x\n\r",i);
235 }
236
```

```
    //// 硬盘复位操作。
    // 首先复位（重新校正）硬盘控制器。然后发送硬盘控制器命令“建立驱动器参数”。在本
    // 命令引起的硬盘中断处理程序中又会调用本函数。此时该函数会根据执行该命令的结果判
    // 断是否要进行出错处理或是继续执行请求项处理操作。
237 static void reset_hd(void)
238 {
239         static int i;
240
    // 如果复位标志 reset 是置位的，则在把复位标志清零后，执行复位硬盘控制器操作。然后
    // 针对第 i 个硬盘向控制器发送“建立驱动器参数”命令。当控制器执行了该命令后，又会
    // 发出硬盘中断信号。此时本函数会被中断过程调用而再次执行。由于 reset 已经标志复位，
    // 因此会首先去执行 246 行开始的语句，判断命令执行是否正常。若还是发生错误就会调用
    // bad_rw_intr() 函数以统计出错次数并根据次确定是否在设置 reset 标志。如果又设置了
    // reset 标志则跳转到 repeat 重新执行本函数。若复位操作正常，则针对下一个硬盘发送
    // “建立驱动器参数”命令，并作上述同样处理。如果系统中 NR_HD 个硬盘都已经正常执行
    // 了发送的命令，则再次 do_hd_request()函数开始对请求项进行处理。
241 repeat:
242         if (reset) {
243                 reset = 0;
244                 i = -1;                         // 初始化当前硬盘号（静态变量）。
245                 reset_controller();
246         } else if (win_result()) {
247                 bad_rw_intr();
248                 if (reset)
249                         goto repeat;
250         }
251         i++;                                    // 处理下一个硬盘（第 1 个是 0）。
252         if (i < NR_HD) {
253                 hd_out(i,hd_info[i].sect,hd_info[i].sect,hd_info[i].head-1,
254                         hd_info[i].cyl,WIN_SPECIFY,&reset_hd);
255         } else
256                 do_hd_request();                // 执行请求项处理。
257 }
258
    //// 意外硬盘中断调用函数。
    // 发生意外硬盘中断时，硬盘中断处理程序中调用的默认 C 处理函数。在被调用函数指针为
    // NULL 时调用该函数。参见（kernel/sys_call.s，第 256 行）。该函数在显示警告信息后
    // 设置复位标志 reset，然后继续调用请求项函数 go_hd_request()并在其中执行复位处理
    // 操作。
259 void unexpected_hd_interrupt(void)
260 {
261         printk("Unexpected HD interrupt\n\r");
262         reset = 1;
263         do_hd_request();
264 }
265
    //// 读写硬盘失败处理调用函数。
    // 如果读扇区时的出错次数大于或等于 7 次时，则结束当前请求项并唤醒等待该请求的进程，
    // 而且对应缓冲区更新标志复位，表示数据没有更新。如果读写一扇区时的出错次数已经大于
    // 3 次，则要求执行复位硬盘控制器操作（设置复位标志）。
266 static void bad_rw_intr(void)
267 {
268         if (++CURRENT->errors >= MAX_ERRORS)
```

```
269                 end_request(0);
270         if (CURRENT->errors > MAX_ERRORS/2)
271                 reset = 1;
272 }
273
    //// 读操作中断调用函数。
    // 该函数将在硬盘读命令结束时引发的硬盘中断过程中被调用。
    // 在读命令执行后会产生硬盘中断信号，并执行硬盘中断处理程序，此时在硬盘中断处理程序
    // 中调用的 C 函数指针 do_hd 已经指向 read_intr()，因此会在一次读扇区操作完成（或出错）
    // 后就会执行该函数。
274 static void read_intr(void)
275 {
    // 该函数首先判断此次读命令操作是否出错。若命令结束后控制器还处于忙状态，或者命令
    // 执行错误，则处理硬盘操作失败问题，接着再次请求硬盘作复位处理并执行其他请求项。
    // 然后返回。每次读操作出错都会对当前请求项作出错次数累计，若出错次数不到最大允许
    // 出错次数的一半，则会先执行硬盘复位操作，然后再执行本次请求项处理。若出错次数已
    // 经大于等于最大允许出错次数 MAX_ERRORS（7 次），则结束本次请求项的处理而去处理队
    // 列中下一个请求项。
276         if (win_result()) {                 // 若控制器忙、读写错或命令执行错，
277                 bad_rw_intr();              // 则进行读写硬盘失败处理。
278                 do_hd_request();            // 再次请求硬盘作相应(复位)处理。
279                 return;
280         }
    // 如果读命令没有出错，则从数据寄存器端口把 1 个扇区的数据读到请求项的缓冲区中，并且
    // 递减请求项所需读取的扇区数值。若递减后不等于 0，表示本项请求还有数据没取完，于是
    // 再次置中断调用 C 函数指针 do_hd 为 read_intr()并直接返回，等待硬盘在读出另 1 个扇区
    // 数据后发出中断并再次调用本函数。注意：281 行语句中的 256 是指内存字，即 512 字节。
    // 注意 1：262 行再次置 do_hd 指针指向 read_intr()是因为硬盘中断处理程序每次调用 do_hd
    // 时都会将该函数指针置空。参见 sys_call.s 程序第 251—253 行。
281         port_read(HD_DATA,CURRENT->buffer,256);     // 读数据到请求结构缓冲区。
282         CURRENT->errors = 0;                // 清出错次数。
283         CURRENT->buffer += 512;             // 调整缓冲区指针，指向新的空区。
284         CURRENT->sector++;                  // 起始扇区号加 1，
285         if (--CURRENT->nr_sectors) {        // 如果所需读出的扇区数还没读完，则再
286                 SET_INTR(&read_intr);       // 置硬盘调用 C 函数指针为 read_intr()。
287                 return;
288         }
    // 执行到此，说明本次请求项的全部扇区数据已经读完，则调用 end_request()函数去处理请
    // 求项结束事宜。 最后再次调用 do_hd_request()，去处理其他硬盘请求项。执行其他硬盘
    // 请求操作。
289         end_request(1);                         // 数据已更新标志置位（1）。
290         do_hd_request();
291 }
292
    //// 写扇区中断调用函数。
    // 该函数将在硬盘写命令结束时引发的硬盘中断过程中被调用。函数功能与 read_intr()类似。
    // 在写命令执行后会产生硬盘中断信号，并执行硬盘中断处理程序，此时在硬盘中断处理程序
    // 中调用的 C 函数指针 do_hd 已经指向 write_intr()，因此会在一次写扇区操作完成（或出错）
    // 后就会执行该函数。
293 static void write_intr(void)
294 {
    // 该函数首先判断此次写命令操作是否出错。若命令结束后控制器还处于忙状态，或者命令
    // 执行错误，则处理硬盘操作失败问题，接着再次请求硬盘作复位处理并执行其他请求项。
```

```
    // 然后返回。在 bad_rw_intr() 函数中，每次操作出错都会对当前请求项作出错次数累计，
    // 若出错次数不到最大允许出错次数的一半，则会先执行硬盘复位操作，然后再执行本次请
    // 求项处理。若出错次数已经大于等于最大允许出错次数 MAX_ERRORS（7 次），则结束本次
    // 请求项的处理而去处理队列中下一个请求项。do_hd_request()中会根据当时具体的标志
    // 状态来判别是否需要先执行复位、重新校正等操作，然后再继续或处理下一个请求项。
295         if (win_result()) {                 // 如果硬盘控制器返回错误信息，
296                 bad_rw_intr();              // 则首先进行硬盘读写失败处理，
297                 do_hd_request();            // 再次请求硬盘作相应(复位)处理。
298                 return;
299         }
    // 此时说明本次写一扇区操作成功，因此将欲写扇区数减 1。若其不为 0，则说明还有扇区
    // 要写，于是把当前请求起始扇区号 +1，并调整请求项数据缓冲区指针指向下一块欲写的
    // 数据。然后再重置硬盘中断处理程序中调用的 C 函数指针 do_hd（指向本函数）。接着向
    // 控制器数据端口写入 512 字节数据，然后函数返回去等待控制器把这些数据写入硬盘后产
    // 生的中断。
300         if (--CURRENT->nr_sectors) {        // 若还有扇区要写，则
301                 CURRENT->sector++;          // 当前请求起始扇区号+1，
302                 CURRENT->buffer += 512;     // 调整请求缓冲区指针，
303                 SET_INTR(&write_intr);      // do_hd 置函数指针为 write_intr()。
304                 port_write(HD_DATA,CURRENT->buffer,256);  // 向数据端口写 256 字。
305                 return;
306         }
    // 若本次请求项的全部扇区数据已经写完，则调用 end_request()函数去处理请求项结束事宜。
    // 最后再次调用 do_hd_request()，去处理其他硬盘请求项。执行其他硬盘请求操作。
307         end_request(1);                     // 处理请求结束事宜（已设置更新标志）。
308         do_hd_request();                    // 执行其他硬盘请求操作。
309 }
310
    //// 硬盘重新校正（复位）中断调用函数。
    // 该函数会在硬盘执行重新校正操作而引发的硬盘中断中被调用。
    // 如果硬盘控制器返回错误信息，则函数首先进行硬盘读写失败处理，然后请求硬盘作相应
    // （复位）处理。 在 bad_rw_intr() 函数中，每次操作出错都会对当前请求项作出错次数
    // 累计，若出错次数不到最大允许出错次数的一半，则会先执行硬盘复位操作，然后再执行
    // 本次请求项处理。若出错次数已经大于等于最大允许出错次数 MAX_ERRORS（7 次），则结
    // 束本次请求项的处理而去处理队列中下一个请求项。do_hd_request() 中会根据当时具体
    // 的标志状态来判别是否需要先执行复位、重新校正等操作，然后再继续或处理下一请求项。
311 static void recal_intr(void)
312 {
313         if (win_result())                   // 若返回出错，则调用 bad_rw_intr()。
314                 bad_rw_intr();
315         do_hd_request();
316 }
317
    // 硬盘操作超时处理。
    // 本函数会在 do_timer()中（kernel/sched.c，第 340 行）被调用。在向硬盘控制器发送了
    // 一个命令后，若在经过了 hd_timeout 个系统滴答后控制器还没有发出一个硬盘中断信号，
    // 则说明控制器（或硬盘）操作超时。此时 do_timer()就会调用本函数设置复位标志 reset
    // 并调用 do_hd_request()执行复位处理。若在预定时间内（200 滴答）硬盘控制器发出了硬
    // 盘中断并开始执行硬盘中断处理程序，那么 ht_timeout 值就会在中断处理程序中被置 0。
    // 此时 do_timer()就会跳过本函数。
318 void hd_times_out(void)
319 {
    // 如果当前并没有请求项要处理（设备请求项指针为 NULL），则无超时可言，直接返回。否
```

```
    // 则先显示警告信息，然后判断当前请求项执行过程中发生的出错次数是否已经大于设定值
    // MAX_ERRORS（7）。如果是则以失败形式结束本次请求项的处理（不设置数据更新标志）。
    // 然后把中断过程中调用的 C 函数指针 do_hd 置空，并设置复位标志 reset，继而在请求项
    // 处理函数 do_hd_request()中去执行复位操作。
320         if (!CURRENT)
321                 return;
322         printk("HD timeout");
323         if (++CURRENT->errors >= MAX_ERRORS)
324                 end_request(0);
325         SET_INTR(NULL);                         // 令 do_hd = NULL,time_out=200。
326         reset = 1;                              // 设置复位标志。
327         do_hd_request();
328 }
329
    //// 执行硬盘读写请求操作。
    // 该函数根据设备当前请求项中的设备号和起始扇区号信息首先计算得到对应硬盘上的柱面号、
    // 当前磁道中扇区号、磁头号数据，然后再根据请求项中的命令（READ/WRITE）对硬盘发送相应
    // 读/写命令。 若控制器复位标志或硬盘重新校正标志已被置位，那么首先会去执行复位或重新
    // 校正操作。
    // 若请求项此时是块设备的第 1 个（原来设备空闲），则块设备当前请求项指针会直接指向该请
    // 求项（参见 ll_rw_blk.c，28 行），并会立刻调用本函数执行读写操作。否则在一个读写操作
    // 完成而引发的硬盘中断过程中，若还有请求项需要处理，则也会在硬盘中断过程中调用本函数。
    // 参见 kernel/sys_call.s，235 行。
330 void do_hd_request(void)
331 {
332         int i,r;
333         unsigned int block,dev;
334         unsigned int sec,head,cyl;
335         unsigned int nsect;
336
    // 函数首先检测请求项的合法性。若请求队列中已没有请求项则退出（参见 blk.h，127 行）。
    // 然后取设备号中的子设备号（见列表后对硬盘设备号的说明）以及设备当前请求项中的起始
    // 扇区号。子设备号即对应硬盘上各分区。如果子设备号不存在或者起始扇区大于该 分区扇
    // 区数-2，则结束该请求项，并跳转到标号 repeat 处（定义在 INIT_REQUEST 开始处）。因为
    // 一次要求读写一块数据（2 个扇区，即 1024 字节），所以请求的扇区号不能大于分区中最后
    // 倒数第二个扇区号。然后通过加上子设备号对应分区的起始扇区号，就把需要读写的块对应
    // 到整个硬盘的绝对扇区号 block 上。而子设备号被 5 整除即可得到对应的硬盘号。
337         INIT_REQUEST;
338         dev = MINOR(CURRENT->dev);
339         block = CURRENT->sector;                        // 请求的起始扇区。
340         if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
341                 end_request(0);
342                 goto repeat;                            // 该标号在 blk.h 最后面。
343         }
344         block += hd[dev].start_sect;
345         dev /= 5;                       // 此时 dev 代表硬盘号（硬盘 0 还是硬盘 1）。
    // 然后根据求得的绝对扇区号 block 和硬盘号 dev，我们就可以计算出对应硬盘中的磁道中扇
    // 区号（sec）、所在柱面号（cyl） 和磁头号（head）。 下面嵌入的汇编代码即用来根据硬
    // 盘信息结构中的每磁道扇区数和硬盘磁头数来计算这些数据。计算方法为：
    // 310--311 行代码初始时 eax 是扇区号 block，edx 中置 0。divl 指令把 edx:eax 组成的扇区
    // 号除以每磁道扇区数（hd_info[dev].sect），所得整数商值在 eax 中，余数在 edx 中。其
    // 中 eax 中是到指定位置的对应总磁道数（所有磁头面），edx 中是当前磁道上的扇区号。
    // 312--313 行代码初始时 eax 是计算出的对应总磁道数，edx 中置 0。divl 指令把 edx:eax
```

```
    // 的对应总磁道数除以硬盘总磁头数（hd_info[dev].head），在 eax 中得到的整除值是柱面
    // 号（cyl），edx 中得到的余数就是对应得当前磁头号（head）。
346         __asm__("divl %4":"=a" (block),"=d" (sec):"0" (block),"1" (0),
347                 "r" (hd_info[dev].sect));
348         __asm__("divl %4":"=a" (cyl),"=d" (head):"0" (block),"1" (0),
349                 "r" (hd_info[dev].head));
350         sec++;                                  // 对计算所得当前磁道扇区号进行调整。
351         nsect = CURRENT->nr_sectors;            // 欲读/写的扇区数。
    // 此时我们得到了欲读写的硬盘起始扇区 block 所对应的硬盘上柱面号（cyl）、在当前磁道
    // 上的扇区号（sec）、磁头号（head）以及欲读写的总扇区数（nsect）。 接着我们可以根
    // 据这些信息向硬盘控制器发送 I/O 操作信息了。 但在发送之前我们还需要先看看是否有复
    // 位控制器状态和重新校正硬盘的标志。通常在复位操作之后都需要重新校正硬盘磁头位置。
    // 若这些标志已被置位，则说明前面的硬盘操作可能出现了一些问题，或者现在是系统第一
    // 次硬盘读写操作等情况。 于是我们就需要重新复位硬盘或控制器并重新校正硬盘。

    // 如果此时复位标志 reset 是置位的，则需要执行复位操作。复位硬盘和控制器，并置硬盘
    // 需要重新校正标志，返回。reset_hd()将首先向硬盘控制器发送复位（重新校正）命令，
    // 然后发送硬盘控制器命令“建立驱动器参数”。
352         if (reset) {
353                 recalibrate = 1;                // 置需重新校正标志。
354                 reset_hd();
355                 return;
356         }
    // 如果此时重新校正标志（recalibrate）是置位的，则首先复位该标志，然后向硬盘控制
    // 器发送重新校正命令。该命令会执行寻道操作，让处于任何地方的磁头移动到 0 柱面。
357         if (recalibrate) {
358                 recalibrate = 0;
359                 hd_out(dev,hd_info[CURRENT_DEV].sect,0,0,0,
360                         WIN_RESTORE,&recal_intr);
361                 return;
362         }
    // 如果以上两个标志都没有置位，那么我们就可以开始向硬盘控制器发送真正的数据读/写
    // 操作命令了。如果当前请求是写扇区操作，则发送写命令，循环读取状态寄存器信息并判
    // 断请求服务标志 DRQ_STAT 是否置位。DRQ_STAT 是硬盘状态寄存器的请求服务位，表示驱
    // 动器已经准备好在主机和数据端口之间传输一个字或一个字节的数据。这方面的信息可参
    // 见程序前面的硬盘操作读/写时序图。如果请求服务 DRQ 置位则退出循环。 若等到循环结
    // 束也没有置位，则表示发送的要求写硬盘命令失败，于是跳转去处理出现的问题或继续执
    // 行下一个硬盘请求。否则我们就可以向硬盘控制器数据寄存器端口 HD_DATA 写入 1 个扇区
    // 的数据。
363         if (CURRENT->cmd == WRITE) {
364                 hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);
365                 for(i=0 ; i<10000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
366                         /* nothing */ ;
367                 if (!r) {
368                         bad_rw_intr();
369                         goto repeat;                    // 该标号在 blk.h 文件最后面。
370                 }
371                 port_write(HD_DATA,CURRENT->buffer,256);
    // 如果当前请求是读硬盘数据，则向硬盘控制器发送读扇区命令。若命令无效则停机。
372         } else if (CURRENT->cmd == READ) {
373                 hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
374         } else
375                 panic("unknown hd-command");
```

```
376 }
377
    // 硬盘系统初始化。
    // 设置硬盘中断描述符，并允许硬盘控制器发送中断请求信号。
    // 该函数设置硬盘设备的请求项处理函数指针为 do_hd_request()，然后设置硬盘中断门描述
    // 符。hd_interrupt（kernel/sys_call.s，第 235 行）是其中断处理过程地址。 硬盘中断号
    // 为 int 0x2E（46），对应 8259A 芯片的中断请求信号 IRQ13。接着复位接联的主 8259A int2
    // 的屏蔽位，允许从片发出中断请求信号。再复位硬盘的中断请求屏蔽位（在从片上），允许
    // 硬盘控制器发送中断请求信号。中断描述符表 IDT 内中断门描述符设置宏 set_intr_gate()
    // 在 include/asm/system.h 中实现。
378 void hd_init(void)
379 {
380         blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;      // do_hd_request()。
381         set_intr_gate(0x2E,&hd_interrupt);  // 设置中断门中处理函数指针。
382         outb_p(inb_p(0x21)&0xfb,0x21);      // 复位接联的主 8259A int2 的屏蔽位。
383         outb(inb_p(0xA1)&0xbf,0xA1);        // 复位硬盘中断请求屏蔽位（在从片上）。
384 }
385
```

9.3 程序 9-3 linux/kernel/blk_drv/ll_rw_blk.c

```
 1 /*
 2  *  linux/kernel/blk_dev/ll_rw.c
 3  *
 4  * (C) 1991 Linus Torvalds
 5  */
 6
 7 /*
 8  * This handles all read/write requests to block devices
 9  */
10 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
11 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
12 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
13 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
14
15 #include "blk.h"          // 块设备头文件。定义请求数据结构、块设备数据结构和宏等信息。
16
17 /*
18  * The request-struct contains all necessary data
19  * to load a nr of sectors into memory
20  */
   /*
    * 请求结构中含有加载 nr 个扇区数据到内存中去的所有必须的信息。
    */
   // 请求项数组队列。共有 NR_REQUEST = 32 个请求项。
21 struct request request[NR_REQUEST];
22
23 /*
24  * used to wait on when there are no free requests
25  */
   /*
    * 是用于在请求数组没有空闲项时进程的临时等待处。
    */
26 struct task_struct * wait_for_request = NULL;
27
28 /* blk_dev_struct is:
29  *      do_request-address
30  *      next-request
31  */
   /* blk_dev_struct 块设备结构是：（参见文件 kernel/blk_drv/blk.h，第 45 行）
    *      do_request-address      // 对应主设备号的请求处理程序指针。
    *      current-request         // 该设备的下一个请求。
    */
   // 块设备数组。该数组使用主设备号作为索引。实际内容将在各块设备驱动程序初始化时填入。
   // 例如，硬盘驱动程序初始化时（hd.c，343 行），第一条语句即用于设置 blk_dev[3]的内容。
32 struct blk_dev_struct blk_dev[NR_BLK_DEV] = {
33         { NULL, NULL },         /* no_dev */     // 0 - 无设备。
34         { NULL, NULL },         /* dev mem */    // 1 - 内存。
35         { NULL, NULL },         /* dev fd */     // 2 - 软驱设备。
36         { NULL, NULL },         /* dev hd */     // 3 - 硬盘设备。
```

```
37          { NULL, NULL },         /* dev ttyx */      // 4 - ttyx 设备。
38          { NULL, NULL },         /* dev tty */       // 5 - tty 设备。
39          { NULL, NULL }          /* dev lp */        // 6 - lp 打印机设备。
40  };
41
42  /*
43   * blk_size contains the size of all block-devices:
44   *
45   * blk_size[MAJOR][MINOR]
46   *
47   * if (!blk_size[MAJOR]) then no minor size checking is done.
48   */
    /*
     * blk_size 数组含有所有块设备的大小（块总数）：
     * blk_size[MAJOR][MINOR]
     * 如果 (!blk_size[MAJOR])，则不必检测子设备的块总数。
     */
    // 设备数据块总数指针数组。每个指针项指向指定主设备号的总块数数组。该总块数数组每一
    // 项对应子设备号确定的一个子设备上所拥有的数据块总数（1 块大小 = 1KB）。
49  int * blk_size[NR_BLK_DEV] = { NULL, NULL, };
50
    // 锁定指定缓冲块。
    // 如果指定的缓冲块已经被其他任务锁定，则使自己睡眠（不可中断地等待），直到被执行解
    // 锁缓冲块的任务明确地唤醒。
51  static inline void lock_buffer(struct buffer_head * bh)
52  {
53          cli();                          // 清中断许可。
54          while (bh->b_lock)              // 如果缓冲区已被锁定则睡眠，直到缓冲区解锁。
55                  sleep_on(&bh->b_wait);
56          bh->b_lock=1;                   // 立刻锁定该缓冲区。
57          sti();                          // 开中断。
58  }
59
    // 释放（解锁）锁定的缓冲区。
    // 该函数与 blk.h 文件中的同名函数完全一样。
60  static inline void unlock_buffer(struct buffer_head * bh)
61  {
62          if (!bh->b_lock)                // 如果该缓冲区没有被锁定，则打印出错信息。
63                  printk("ll_rw_block.c: buffer not locked\n\r");
64          bh->b_lock = 0;                 // 清锁定标志。
65          wake_up(&bh->b_wait);           // 唤醒等待该缓冲区的任务。
66  }
67
68  /*
69   * add-request adds a request to the linked list.
70   * It disables interrupts so that it can muck with the
71   * request-lists in peace.
72   *
73   * Note that swapping requests always go before other requests,
74   * and are done in the order they appear.
75   */
    /*
     * add-request()向链表中加入一项请求项。它会关闭中断，
```

```
 * 这样就能安全地处理请求链表了。
 *
 * 注意，交换请求总是在其他请求之前操作，并且以它们出
 * 现的顺序完成。
 */
//// 向链表中加入请求项。
// 参数 dev 是指定块设备结构指针，该结构中有处理请求项函数指针和当前正在请求项指针；
// req 是已设置好内容的请求项结构指针。
// 本函数把已经设置好的请求项 req 添加到指定设备的请求项链表中。如果该设备的当前请求
// 请求项指针为空，则可以设置 req 为当前请求项并立刻调用设备请求项处理函数。否则就把
// req 请求项插入到该请求项链表中。
76 static void add_request(struct blk_dev_struct * dev, struct request * req)
77 {
78         struct request * tmp;
79
// 首先再进一步对参数提供的请求项的指针和标志作初始设置。置空请求项中的下一请求项指
// 针，关中断并清除请求项相关缓冲区脏标志。
80         req->next = NULL;
81         cli();                                  // 关中断。
82         if (req->bh)
83                 req->bh->b_dirt = 0;            // 清缓冲区“脏”标志。
// 然后查看指定设备是否有当前请求项，即查看设备是否正忙。如果指定设备 dev 当前请求项
// （current_request）子段为空，则表示目前该设备没有请求项，本次是第 1 个请求项，也是
// 唯一的一个。因此可将块设备当前请求指针直接指向该请求项，并立刻执行相应设备的请求
// 函数。
84         if (!(tmp = dev->current_request)) {
85                 dev->current_request = req;
86                 sti();                          // 开中断。
87                 (dev->request_fn)();            // 执行请求函数，对于硬盘是 do_hd_request()。
88                 return;
89         }
// 如果目前该设备已经有当前请求项在处理，则首先利用电梯算法搜索最佳插入位置，然后将
// 当前请求项插入到请求链表中。在搜索过程中，如果判断出欲插入请求项的缓冲块头指针空，
// 即没有缓冲块，那么就需要找一个项，其已经有可用的缓冲块。因此若当前插入位置（tmp
// 之后）处的空闲项缓冲块头指针不空，就选择这个位置。于是退出循环并把请求项插入此处。
// 最后开中断并退出函数。电梯算法的作用是让磁盘磁头的移动距离最小，从而改善（减少）
// 硬盘访问时间。
// 下面 for 循环中 if 语句用于把 req 所指请求项与请求队列（链表）中已有的请求项作比较，
// 找出 req 插入该队列的正确位置顺序。然后中断循环，并把 req 插入到该队列正确位置处。
90         for ( ; tmp->next ; tmp=tmp->next) {
91                 if (!req->bh)
92                         if (tmp->next->bh)
93                                 break;
94                         else
95                                 continue;
96                 if ((IN_ORDER(tmp,req) ||
97                     !IN_ORDER(tmp,tmp->next)) &&
98                     IN_ORDER(req,tmp->next))
99                         break;
100        }
101        req->next=tmp->next;
102        tmp->next=req;
103        sti();
```

```
104 }
105
    //// 创建请求项并插入请求队列中。
    // 参数 major 是主设备号；rw 是指定命令；bh 是存放数据的缓冲区头指针。
106 static void make_request(int major,int rw, struct buffer_head * bh)
107 {
108         struct request * req;
109         int rw_ahead;
110
111 /* WRITEA/READA is special case - it is not really needed, so if the */
112 /* buffer is locked, we just forget about it, else it's a normal read */
    /* WRITEA/READA 是一种特殊情况 - 它们并非必要，所以如果缓冲区已经上锁，*/
    /* 我们就不用管它，否则的话它只是一个一般的读操作。 */
    // 这里'READ'和'WRITE'后面的'A'字符代表英文单词 Ahead，表示提前预读/写数据块的意思。
    // 该函数首先对命令 READA/WRITEA 的情况进行一些处理。对于这两个命令，当指定的缓冲区
    // 正在使用而已被上锁时，就放弃预读/写请求。否则就作为普通的 READ/WRITE 命令进行操作。
    // 另外，如果参数给出的命令既不是 READ 也不是 WRITE，则表示内核程序有错，显示出错信
    // 息并停机。注意，在修改命令之前这里已为参数是否是预读/写命令设置了标志 rw_ahead。
113         if (rw_ahead = (rw == READA || rw == WRITEA)) {
114                 if (bh->b_lock)
115                         return;
116                 if (rw == READA)
117                         rw = READ;
118                 else
119                         rw = WRITE;
120         }
121         if (rw!=READ && rw!=WRITE)
122                 panic("Bad block dev command, must be R/W/RA/WA");
123         lock_buffer(bh);
124         if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
125                 unlock_buffer(bh);
126                 return;
127         }
128 repeat:
129 /* we don't allow the write-requests to fill up the queue completely:
130  * we want some room for reads: they take precedence. The last third
131  * of the requests are only for reads.
132  */
    /* 我们不能让队列中全都是写请求项：我们需要为读请求保留一些空间：读操作
     * 是优先的。请求队列的后三分之一空间仅用于读请求项。
     */
    // 好，现在我们必须为本函数生成并添加读/写请求项了。首先我们需要在请求数组中寻找到
    // 一个空闲项（槽）来存放新请求项。搜索过程从请求数组末端开始。根据上述要求，对于读
    // 命令请求，我们直接从队列末尾开始搜索，而对于写请求就只能从队列 2/3 处向队列头处搜
    // 索空项填入。于是我们开始从后向前搜索，当请求结构 request 的设备字段 dev 值 = -1 时，
    // 表示该项未被占用（空闲）。如果没有一项是空闲的（此时请求项数组指针已经搜索越过头
    // 部），则查看此次请求是否是提前读/写（READA 或 WRITEA），如果是则放弃此次请求操作。
    // 否则让本次请求操作先睡眠（以等待请求队列腾出空项），过一会再来搜索请求队列。
133         if (rw == READ)
134                 req = request+NR_REQUEST;           // 对于读请求，将指针指向队列尾部。
135         else
136                 req = request+((NR_REQUEST*2)/3);   // 对于写请求，指针指向队列 2/3 处。
137 /* find an empty request */                          /* 搜索一个空请求项 */
```

```
138         while (--req >= request)
139                 if (req->dev<0)
140                         break;
141 /* if none found, sleep on new requests: check for rw_ahead */
    /* 如果没有找到空闲项，则让该次新请求操作睡眠：需检查是否提前读/写 */
142         if (req < request) {                    // 如果已搜索到头（队列无空项），
143                 if (rw_ahead) {                 // 则若是提前读/写请求，则退出。
144                         unlock_buffer(bh);
145                         return;
146                 }
147                 sleep_on(&wait_for_request);    // 否则就睡眠，过会再查看请求队列。
148                 goto repeat;                    // 跳转 110 行。
149         }
150 /* fill up the request-info, and add it to the queue */
    /* 向空闲请求项中填写请求信息，并将其加入队列中 */
    // OK，程序执行到这里表示已找到一个空闲请求项。 于是我们在设置好的新请求项后就调用
    // add_request()把它添加到请求队列中，立马退出。请求结构请参见 blk_drv/blk.h，23 行。
    // req->sector 是读写操作的起始扇区号，req->buffer 是请求项存放数据的缓冲区。
151         req->dev = bh->b_dev;               // 设备号。
152         req->cmd = rw;                      // 命令(READ/WRITE)。
153         req->errors=0;                      // 操作时产生的错误次数。
154         req->sector = bh->b_blocknr<<1;     // 起始扇区。块号转换成扇区号(1 块=2 扇区)。
155         req->nr_sectors = 2;                // 本请求项需要读写的扇区数。
156         req->buffer = bh->b_data;           // 请求项缓冲区指针指向需读写的数据缓冲区。
157         req->waiting = NULL;                // 任务等待操作执行完成的地方。
158         req->bh = bh;                       // 缓冲块头指针。
159         req->next = NULL;                   // 指向下一请求项。
160         add_request(major+blk_dev,req);     // 将请求项加入队列中(blk_dev[major],req)。
161 }
162
    //// 低级页面读写函数（Low Level Read Write Page）。
    // 以页面（4K）为单位访问块设备数据，即每次读/写 8 个扇区。参见下面 ll_rw_blk()函数。
163 void ll_rw_page(int rw, int dev, int page, char * buffer)
164 {
165         struct request * req;
166         unsigned int major = MAJOR(dev);
167
    // 首先对函数参数的合法性进行检测。如果设备主设备号不存在或者该设备的请求操作函数不
    // 存在，则显示出错信息，并返回。如果参数给出的命令既不是 READ 也不是 WRITE，则表示
    // 内核程序有错，显示出错信息并停机。
168         if (major >= NR_BLK_DEV || !(blk_dev[major].request_fn)) {
169                 printk("Trying to read nonexistent block-device\n\r");
170                 return;
171         }
172         if (rw!=READ && rw!=WRITE)
173                 panic("Bad block dev command, must be R/W");
    // 在参数检测操作完成后，我们现在需要为本次操作建立请求项。首先我们需要在请求数组中
    // 寻找到一个空闲项（槽）来存放新请求项。搜索过程从请求数组末端开始。于是我们开始从
    // 后向前搜索，当请求结构 request 的设备字段 dev 值 <0 时，表示该项未被占用（空闲）。
    // 如果没有一项是空闲的（此时请求项数组指针已经搜索越过头部），则让本次请求操作先睡
    // 眠（以等待请求队列腾出空项），过一会再来搜索请求队列。
174 repeat:
175         req = request+NR_REQUEST;           // 将指针指向队列尾部。
```

```
176         while (--req >= request)
177                 if (req->dev<0)
178                         break;
179         if (req < request) {
180                 sleep_on(&wait_for_request);        // 睡眠，过会再查看请求队列。
181                 goto repeat;                        // 跳转到 174 行去重新搜索。
182         }
183 /* fill up the request-info, and add it to the queue */
    /* 向空闲请求项中填写请求信息，并将其加入队列中 */
    // OK，程序执行到这里表示已找到一个空闲请求项。 于是我们设置好新请求项，把当前进程
    // 置为不可中断睡眠中断后，就去调用 add_request()把它添加到请求队列中，然后直接调用
    // 调度函数让当前进程睡眠等待页面从交换设备中读入。这里不象 make_request()函数那样
    // 直接退出函数而调用了 schedule()，是因为 make_request()函数仅读 2 个扇区数据。而这
    // 里需要对交换设备读/写 8 个扇区，需要花较长的时间。因此当前进程肯定需要等待而睡眠。
    // 因此这里直接就让进程去睡眠了，省得在程序其他地方还要进行这些判断操作。
184         req->dev = dev;                             // 设备号。
185         req->cmd = rw;                              // 命令(READ/WRITE)。
186         req->errors = 0;                            // 读写操作错误计数。
187         req->sector = page<<3;                      // 起始读写扇区。
188         req->nr_sectors = 8;                        // 读写扇区数。
189         req->buffer = buffer;                       // 数据缓冲区。
190         req->waiting = current;                     // 当前进程进入该请求等待队列。
191         req->bh = NULL;                             // 无缓冲块头指针（不用高速缓冲）。
192         req->next = NULL;                           // 下一个请求项指针。
193         current->state = TASK_UNINTERRUPTIBLE;      // 置为不可中断状态。
194         add_request(major+blk_dev,req);             // 将请求项加入队列中。
195         schedule();
196 }
197
    //// 低级数据块读写函数（Low Level Read Write Block）。
    // 该函数是块设备驱动程序与系统其他部分的接口函数。通常在 fs/buffer.c 程序中被调用。
    // 主要功能是创建块设备读写请求项并插入到指定块设备请求队列中。实际的读写操作则是
    // 由设备的 request_fn()函数完成。对于硬盘操作，该函数是 do_hd_request()；对于软盘
    // 操作该函数是 do_fd_request()；对于虚拟盘则是 do_rd_request()。 另外，在调用该函
    // 数之前，调用者需要首先把读/写块设备的信息保存在缓冲块头结构中，如设备号、块号。
    // 参数：rw - READ、READA、WRITE 或 WRITEA 是命令；bh - 数据缓冲块头指针。
198 void ll_rw_block(int rw, struct buffer_head * bh)
199 {
200         unsigned int major;                 // 主设备号（对于硬盘是 3）。
201
    // 如果设备主设备号不存在或者该设备的请求操作函数不存在，则显示出错信息，并返回。
    // 否则创建请求项并插入请求队列。
202         if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
203         !(blk_dev[major].request_fn)) {
204                 printk("Trying to read nonexistent block-device\n\r");
205                 return;
206         }
207         make_request(major,rw,bh);
208 }
209
    //// 块设备初始化函数，由初始化程序 main.c 调用。
    // 初始化请求数组，将所有请求项置为空闲项(dev = -1)。有 32 项(NR_REQUEST = 32)。
210 void blk_dev_init(void)
```

```
211 {
212         int i;
213
214         for (i=0 ; i<NR_REQUEST ; i++) {
215                 request[i].dev = -1;
216                 request[i].next = NULL;
217         }
218 }
219
```

9.4 程序 9-4 linux/kernel/blk_drv/ramdisk.c

```
/*
 *  linux/kernel/blk_drv/ramdisk.c
 *
 *  Written by Theodore Ts'o, 12/2/91
 */
/* 由 Theodore Ts'o 编制，12/2/91
 */
// Theodore Ts'o (Ted Ts'o)是 Linux 社区中的著名人物。Linux 在世界范围内的流行也有他很
// 大的功劳。早在 Linux 操作系统刚问世时，他就怀着极大的热情为 Linux 的发展提供了电子邮
// 件列表服务 maillist，并在北美地区最早设立了 Linux 的 ftp 服务器站点（tsx-11.mit.edu），
// 而且至今仍为广大 Linux 用户提供服务。他对 Linux 作出的最大贡献之一是提出并实现了 ext2
// 文件系统。该文件系统已成为 Linux 世界中事实上的文件系统标准。最近他又推出了 ext3 文件
// 系统，大大提高了文件系统的稳定性、可恢复性和访问效率。作为对他的推崇，第 97 期（2002
// 年 5 月）的 LinuxJournal 期刊将他作为了封面人物，并对他进行了采访。目前他为 IBM Linux
// 技术中心工作，并从事着有关 LSB (Linux Standard Base)等方面的工作。（他的个人主页是：
// http://thunk.org/tytso/)

#include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。

#include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）可选项。
#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据，
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/fs.h>     // 文件系统头文件。定义文件表结构（file、m_inode）等。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原型定义。
#include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等嵌入式汇编宏。
#include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
#include <asm/memory.h>   // 内存拷贝头文件。含有 memcpy()嵌入式汇编宏函数。

// 定义 RAM 盘主设备号符号常数。在驱动程序中主设备号必须在包含 blk.h 文件之前被定义。
// 因为 blk.h 文件中要用到这个符号常数值来确定一些列的其他常数符号和宏。
#define MAJOR_NR 1
#include "blk.h"

// 虚拟盘在内存中的起始位置。该位置会在第 52 行上初始化函数 rd_init()中确定。参见内核
// 初始化程序 init/main.c，第 124 行。'rd'是'ramdisk'的缩写。
char    *rd_start;                      // 虚拟盘在内存中的开始地址。
int     rd_length = 0;                  // 虚拟盘所占内存大小（字节）。

// 虚拟盘当前请求项操作函数。
// 该函数的程序结构与硬盘的 do_hd_request()函数类似，参见 hd.c，294 行。在低级块设备
// 接口函数 ll_rw_block()建立起虚拟盘（rd）的请求项并添加到 rd 的链表中之后，就会调
// 用该函数对 rd 当前请求项进行处理。该函数首先计算当前请求项中指定的起始扇区对应虚
// 拟盘所处内存的起始位置 addr 和要求的扇区数对应的字节长度值 len，然后根据请求项中
// 的命令进行操作。若是写命令 WRITE，就把请求项所指缓冲区中的数据直接复制到内存位置
// addr 处。若是读操作则反之。数据复制完成后即可直接调用 end_request() 对本次请求项
// 作结束处理。然后跳转到函数开始处再去处理下一个请求项。若已没有请求项则退出。
void do_rd_request(void)
{
        int     len;
```

```
26          char    *addr;
27
    // 首先检测请求项的合法性，若已没有请求项则退出（参见 blk.h，第 127 行）。然后计算请
    // 求项处理的虚拟盘中起始扇区在物理内存中对应的地址 addr 和占用的内存字节长度值 len。
    // 下句用于取得请求项中的起始扇区对应的内存起始位置和内存长度。其中 sector << 9 表示
    // sector * 512，换算成字节值。CURRENT 被定义为 (blk_dev[MAJOR_NR].current_request)。
28          INIT_REQUEST;
29          addr = rd_start + (CURRENT->sector << 9);
30          len = CURRENT->nr_sectors << 9;
    // 如果当前请求项中子设备号不为 1 或者对应内存起始位置大于虚拟盘末尾，则结束该请求项，
    // 并跳转到 repeat 处去处理下一个虚拟盘请求项。标号 repeat 定义在宏 INIT_REQUEST 内，
    // 位于宏的开始处，参见 blk.h 文件第 127 行。
31          if ((MINOR(CURRENT->dev) != 1) || (addr+len > rd_start+rd_length)) {
32                  end_request(0);
33                  goto repeat;
34          }
    // 然后进行实际的读写操作。如果是写命令（WRITE），则将请求项中缓冲区的内容复制到地址
    // addr 处，长度为 len 字节。如果是读命令（READ），则将 addr 开始的内存内容复制到请求项
    // 缓冲区中，长度为 len 字节。否则显示命令不存在，死机。
35          if (CURRENT-> cmd == WRITE) {
36                  (void ) memcpy(addr,
37                                 CURRENT->buffer,
38                                 len);
39          } else if (CURRENT->cmd == READ) {
40                  (void) memcpy(CURRENT->buffer,
41                                 addr,
42                                 len);
43          } else
44                  panic("unknown ramdisk-command");
    // 然后在请求项成功后处理，置更新标志。并继续处理本设备的下一请求项。
45          end_request(1);
46          goto repeat;
47  }
48
49  /*
50   * Returns amount of memory which needs to be reserved.
51   */
    /* 返回内存虚拟盘 ramdisk 所需的内存量 */
    // 虚拟盘初始化函数。
    // 该函数首先设置虚拟盘设备的请求项处理函数指针指向 do_rd_request()，然后确定虚拟盘
    // 在物理内存中的起始地址、占用字节长度值。并对整个虚拟盘区清零。最后返回盘区长度。
    // 当 linux/Makefile 文件中设置过 RAMDISK 值不为零时，表示系统中会创建 RAM 虚拟盘设备。
    // 在这种情况下的内核初始化过程中，本函数就会被调用（init/main.c，L151 行）。该函数
    // 的第 2 个参数 length 会被赋值成 RAMDISK * 1024，单位为字节。
52  long rd_init(long mem_start, int length)
53  {
54          int     i;
55          char    *cp;
56
57          blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;  // do_rd_request()。
58          rd_start = (char *) mem_start;                  // 对于 16MB 系统该值为 4MB。
59          rd_length = length;                             // 虚拟盘区域长度值。
60          cp = rd_start;
```

```
61          for (i=0; i < length; i++)                      // 盘区清零。
62                  *cp++ = '\0';
63          return(length);
64  }
65
66  /*
67   * If the root device is the ram disk, try to load it.
68   * In order to do this, the root device is originally set to the
69   * floppy, and we later change it to be ram disk.
70   */
    /*
     * 如果根文件系统设备(root device)是 ramdisk 的话，则尝试加载它。
     * root device 原先是指向软盘的，我们将它改成指向 ramdisk。
     */
    //// 尝试把根文件系统加载到虚拟盘中。
    // 该函数将在内核设置函数 setup()（hd.c，156 行）中被调用。另外，1 磁盘块 = 1024 字节。
    // 第 75 行上的变量 block=256 表示根文件系统映像文件被存储于 boot 盘第 256 磁盘块开始处。
71  void rd_load(void)
72  {
73          struct buffer_head *bh;         // 高速缓冲块头指针。
74          struct super_block      s;      // 文件超级块结构。
75          int             block = 256;    /* Start at block 256 */ /* 开始于 256 盘块 */
76          int             i = 1;
77          int             nblocks;        // 文件系统盘块总数。
78          char            *cp;            /* Move pointer */
79
    // 首先检查虚拟盘的有效性和完整性。如果 ramdisk 的长度为零，则退出。否则显示 ramdisk
    // 的大小以及内存起始位置。如果此时根文件设备不是软盘设备，则也退出。
80          if (!rd_length)
81                  return;
82          printk("Ram disk: %d bytes, starting at 0x%x\n", rd_length,
83                  (int) rd_start);
84          if (MAJOR(ROOT_DEV) != 2)
85                  return;
    // 然后读根文件系统的基本参数。即读软盘块 256+1、256 和 256+2。这里 block+1 是指磁盘上
    // 的超级块。breada() 用于读取指定的数据块，并标出还需要读的块，然后返回含有数据块的
    // 缓冲区指针。如果返回 NULL，则表示数据块不可读（fs/buffer.c，322）。然后把缓冲区中
    // 的磁盘超级块（d_super_block 是磁盘超级块结构）复制到 s 变量中，并释放缓冲区。 接着
    // 我们开始对超级块的有效性进行判断。 如果超级块中文件系统魔数不对，则说明加载的数据
    // 块不是 MINIX 文件系统，于是退出。有关 MINIX 超级块的结构请参见文件系统一章内容。
86          bh = breada(ROOT_DEV,block+1,block,block+2,-1);
87          if (!bh) {
88                  printk("Disk error while looking for ramdisk!\n");
89                  return;
90          }
91          *((struct d_super_block *) &s) = *((struct d_super_block *) bh->b_data);
92          brelse(bh);
93          if (s.s_magic != SUPER_MAGIC)
94                  /* No ram disk image present, assume normal floppy boot */
                    /* 磁盘中没有 ramdisk 映像文件，退出去执行通常的软盘引导 */
95                  return;
    // 然后我们试图把整个根文件系统读入到内存虚拟盘区中。对于一个文件系统来说，其超级块
    // 结构的 s_nzones 字段中保存着总逻辑块数（或称为区段数）。一个逻辑块中含有的数据块
```

```
    // 数则由字段 s_log_zone_size 指定。因此文件系统中的数据块总数 nblocks 就等于（逻辑块
    // 数 * 2^(每区段块数的次方)），即 nblocks = (s_nzones * 2^s_log_zone_size)。如果遇到
    // 文件系统中数据块总数大于内存虚拟盘所能容纳的块数的情况，则不能执行加载操作，而只
    // 能显示出错信息并返回。
 96         nblocks = s.s_nzones << s.s_log_zone_size;
 97         if (nblocks > (rd_length >> BLOCK_SIZE_BITS)) {
 98                 printk("Ram disk image too big!  (%d blocks, %d avail)\n",
 99                         nblocks, rd_length >> BLOCK_SIZE_BITS);
100                 return;
101         }
    // 否则若虚拟盘能容纳得下文件系统总数据块数，则我们显示加载数据块信息，并让 cp 指向
    // 内存虚拟盘起始处，然后开始执行循环操作将磁盘上根文件系统映像文件加载到虚拟盘上。
    // 在操作过程中，如果一次需要加载的盘块数大于 2 块，我们就是用超前预读函数 breada()，
    // 否则就使用 bread()函数进行单块读取。若在读盘过程中出现 I/O 操作错误，就只能放弃加
    // 载过程返回。所读取的磁盘块会使用 memcpy()函数从高速缓冲区中复制到内存虚拟盘相应
    // 位置处，同时显示已加载的块数。显示字符串中的八进制数'\010'表示显示一个制表符。
102         printk("Loading %d bytes into ram disk... 0000k",
103                 nblocks << BLOCK_SIZE_BITS);
104         cp = rd_start;
105         while (nblocks) {
106                 if (nblocks > 2)                // 若读取块数多于 2 块则采用超前预读。
107                         bh = breada(ROOT_DEV, block, block+1, block+2, -1);
108                 else                            // 否则就单块读取。
109                         bh = bread(ROOT_DEV, block);
110                 if (!bh) {
111                         printk("I/O error on block %d, aborting load\n",
112                                 block);
113                         return;
114                 }
115                 (void) memcpy(cp, bh->b_data, BLOCK_SIZE);      // 复制到 cp 处。
116                 brelse(bh);
117                 printk("\010\010\010\010\010%4dk",i);          // 打印加载块计数值。
118                 cp += BLOCK_SIZE;                               // 虚拟盘指针前移。
119                 block++;
120                 nblocks--;
121                 i++;
122         }
    // 当 boot 盘中从 256 盘块开始的整个根文件系统加载完毕后，我们显示“done”，并把目前
    // 根文件设备号修改成虚拟盘的设备号 0x0101，最后返回。
123         printk("\010\010\010\010\010done \n");
124         ROOT_DEV=0x0101;
125 }
126
```

9.5 程序 9-5 linux/kernel/blk_drv/floppy.c

```
 1 /*
 2  *  linux/kernel/floppy.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * 02.12.91 - Changed to static variables to indicate need for reset
 9  * and recalibrate. This makes some things easier (output_byte reset
10  * checking etc), and means less interrupt jumping in case of errors,
11  * so the code is hopefully easier to understand.
12  */
   /*
    * 02.12.91 - 修改成静态变量，以适应复位和重新校正操作。这使得某些事情
    * 做起来较为方便（output_byte 复位检查等），并且意味着在出错时中断跳转
    * 要少一些，所以也希望代码能更容易被理解。
    */
13
14 /*
15  * This file is certainly a mess. I've tried my best to get it working,
16  * but I don't like programming floppies, and I have only one anyway.
17  * Urgel. I should check for more errors, and do more graceful error
18  * recovery. Seems there are problems with several drives. I've tried to
19  * correct them. No promises.
20  */
   /*
    * 这个文件当然比较混乱。我已经尽我所能使其能够工作，但我不喜欢软驱编程，
    * 而且我也只有一个软驱。另外，我应该做更多的查错工作，以及改正更多的错误。
    * 对于某些软盘驱动器，本程序好象还存在一些问题。我已经尝试着进行纠正了，
    * 但不能保证问题已消失。
    */
21
22 /*
23  * As with hd.c, all routines within this file can (and will) be called
24  * by interrupts, so extreme caution is needed. A hardware interrupt
25  * handler may not sleep, or a kernel panic will happen. Thus I cannot
26  * call "floppy-on" directly, but have to set a special timer interrupt
27  * etc.
28  *
29  * Also, I'm not certain this works on more than 1 floppy. Bugs may
30  * abund.
31  */
   /*
    * 如同 hd.c 文件一样，该文件中的所有子程序都能够被中断调用，所以需要特别
    * 地小心。硬件中断处理程序是不能睡眠的，否则内核就会傻掉(死机)☺。因此不能
    * 直接调用"floppy-on"，而只能设置一个特殊的定时中断等。
    *
    * 另外，我不能保证该程序能在多于 1 个软驱的系统上工作，有可能存在错误。
    */
```

```
32
33 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
34 #include <linux/fs.h>     // 文件系统头文件。含文件表结构（file、m_inode）等。
35 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
36 #include <linux/fdreg.h>  // 软驱头文件。含有软盘控制器参数的一些定义。
37 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入汇编宏。
38 #include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
39 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
40
   // 定义软驱主设备号符号常数。在驱动程序中，主设备号必须在包含 blk.h 文件之前被定义。
   // 因为 blk.h 文件中要用到这个符号常数值来确定一些列其他相关符号常数和宏。
41 #define MAJOR_NR 2        // 软驱的主设备号是 2。
42 #include "blk.h"          // 块设备头文件。定义请求结构、块设备结构和宏函数等信息。
43
44 static int recalibrate = 0;       // 标志：1 表示需要重新校正磁头位置（磁头归零道）。
45 static int reset = 0;             // 标志：1 表示需要进行复位操作。
46 static int seek = 0;              // 标志：1 表示需要执行寻道操作。
47
   // 当前数字输出寄存器 DOR（Digital Output Register），定义在 kernel/sched.c，223 行。
   // 该变量含有软驱操作中的重要标志，包括选择软驱、控制电机启动、启动复位软盘控制器以
   // 及允许/禁止 DMA 和中断请求。请参见程序列表后对 DOR 寄存器的说明。
48 extern unsigned char current_DOR;
49
   // 字节直接数输出（嵌入汇编宏）。把值 val 输出到 port 端口。
50 #define immoutb_p(val,port) \
51 __asm__("outb %0,%1\n\tjmp 1f\n1:\tjmp 1f\n1:"::"a" ((char) (val)),"i" (port))
52
   // 这两个宏定义用于计算软驱的设备号。
   // 参数 x 是次设备号。次设备号 = TYPE*4 + DRIVE。计算方法参见列表后。
53 #define TYPE(x) ((x)>>2)          // 软驱类型（2--1.2Mb，7--1.44Mb）。
54 #define DRIVE(x) ((x)&0x03)       // 软驱序号（0--3 对应 A--D）。
55 /*
56  * Note that MAX_ERRORS=8 doesn't imply that we retry every bad read
57  * max 8 times - some types of errors increase the errorcount by 2,
58  * so we might actually retry only 5-6 times before giving up.
59  */
   /*
    * 注意，下面定义 MAX_ERRORS=8 并不表示对每次读错误尝试最多 8 次 - 有些类型
    * 的错误会把出错计数值乘 2，所以我们实际上在放弃操作之前只需尝试 5-6 遍即可。
    */
60 #define MAX_ERRORS 8
61
62 /*
63  * globals used by 'result()'
64  */
   /* 下面是函数'result()'使用的全局变量 */
   // 这些状态字节中各比特位的含义请参见 include/linux/fdreg.h 头文件。另参见列表后说明。
65 #define MAX_REPLIES 7                           // FDC 最多返回 7 字节的结果信息。
66 static unsigned char reply_buffer[MAX_REPLIES]; // 存放 FDC 返回的应答结果信息。
67 #define ST0 (reply_buffer[0])                   // 结果状态字节 0。
68 #define ST1 (reply_buffer[1])                   // 结果状态字节 1。
69 #define ST2 (reply_buffer[2])                   // 结果状态字节 2。
70 #define ST3 (reply_buffer[3])                   // 结果状态字节 3。
```

```
71
72 /*
73  * This struct defines the different floppy types. Unlike minix
74  * linux doesn't have a "search for right type"-type, as the code
75  * for that is convoluted and weird. I've got enough problems with
76  * this driver as it is.
77  *
78  * The 'stretch' tells if the tracks need to be boubled for some
79  * types (ie 360kB diskette in 1.2MB drive etc). Others should
80  * be self-explanatory.
81  */
   /*
    * 下面的软盘结构定义了不同的软盘类型。与 minix 不同的是，Linux 没有
    * "搜索正确的类型"-类型，因为对其处理的代码令人费解且怪怪的。本程序
    * 已经让我遇到太多的问题了。
    *
    * 对某些类型的软盘（例如在 1.2MB 驱动器中的 360kB 软盘等），'stretch'
    * 用于检测磁道是否需要特殊处理。其他参数应该是自明的。
    */
   // 定义软盘结构。软盘参数有：
   // size          大小(扇区数)；
   // sect          每磁道扇区数；
   // head          磁头数；
   // track         磁道数；
   // stretch       对磁道是否要特殊处理（标志）；
   // gap           扇区间隙长度(字节数)；
   // rate          数据传输速率；
   // spec1         参数（高 4 位步进速率，低四位磁头卸载时间）。
82 static struct floppy_struct {
83         unsigned int size, sect, head, track, stretch;
84         unsigned char gap,rate,spec1;
85 } floppy_type[] = {
86         {    0, 0,0,  0,0,0x00,0x00,0x00 },       /* no testing */
87         {  720, 9,2,40,0,0x2A,0x02,0xDF },       /* 360kB PC diskettes */
88         { 2400,15,2,80,0,0x1B,0x00,0xDF },       /* 1.2 MB AT-diskettes */
89         {  720, 9,2,40,1,0x2A,0x02,0xDF },       /* 360kB in 720kB drive */
90         { 1440, 9,2,80,0,0x2A,0x02,0xDF },       /* 3.5" 720kB diskette */
91         {  720, 9,2,40,1,0x23,0x01,0xDF },       /* 360kB in 1.2MB drive */
92         { 1440, 9,2,80,0,0x23,0x01,0xDF },       /* 720kB in 1.2MB drive */
93         { 2880,18,2,80,0,0x1B,0x00,0xCF },       /* 1.44MB diskette */
94 };
95
96 /*
97  * Rate is 0 for 500kb/s, 2 for 300kbps, 1 for 250kbps
98  * Spec1 is 0xSH, where S is stepping rate (F=1ms, E=2ms, D=3ms etc),
99  * H is head unload time (1=16ms, 2=32ms, etc)
100 *
101 * Spec2 is (HLD<<1 | ND), where HLD is head load time (1=2ms, 2=4 ms etc)
102 * and ND is set means no DMA. Hardcoded to 6 (HLD=6ms, use DMA).
103 */
   /*
    * 上面速率 rate：0 表示 500kbps，1 表示 300kbps，2 表示 250kbps。
    * 参数 spec1 是 0xSH，其中 S 是步进速率（F-1ms，E-2ms，D=3ms 等），
```

```
     * H 是磁头卸载时间（1=16ms，2=32ms 等）
     *
     * spec2 是（HLD<<1 | ND），其中 HLD 是磁头加载时间（1=2ms，2=4ms 等）
     * ND 置位表示不使用 DMA（No DMA），在程序中硬编码成 6（HLD=6ms，使用 DMA）。
     */
    // 注意，上述磁头加载时间的缩写 HLD 最好写成标准的 HLT（Head Load Time）。
104
    // floppy_interrupt()是 sys_call.s 程序中软驱中断处理过程标号。这里将在软盘初始化
    // 函数 floppy_init()（第 469 行）使用它初始化中断陷阱门描述符。
105 extern void floppy_interrupt(void);
    // 这是 boot/head.s 第 132 行处定义的临时软盘缓冲区。如果请求项的缓冲区处于内存 1MB
    // 以上某个地方，则需要将 DMA 缓冲区设在临时缓冲区域处。因为 8237A 芯片只能在 1MB 地
    // 址范围内寻址。
106 extern char tmp_floppy_area[1024];
107
108 /*
109  * These are global variables, as that's the easiest way to give
110  * information to interrupts. They are the data used for the current
111  * request.
112  */
    /*
     * 下面是一些全局变量，因为这是将信息传给中断程序最简单的方式。它们
     * 用于当前请求项的数据。
     */
    // 这些所谓的“全局变量”是指在软盘中断处理程序中调用的 C 函数使用的变量。当然这些
    // C 函数都在本程序内。
113 static int cur_spec1 = -1;                         // 当前软盘参数 spec1。
114 static int cur_rate = -1;                          // 当前软盘转速 rate。
115 static struct floppy_struct * floppy = floppy_type; // 软盘类型结构数组指针。
116 static unsigned char current_drive = 0;             // 当前驱动器号。
117 static unsigned char sector = 0;                    // 当前扇区号。
118 static unsigned char head = 0;                      // 当前磁头号。
119 static unsigned char track = 0;                     // 当前磁道号。
120 static unsigned char seek_track = 0;                // 寻道磁道号。
121 static unsigned char current_track = 255;           // 当前磁头所在磁道号。
122 static unsigned char command = 0;                   // 读/写命令。
123 unsigned char selected = 0;    // 软驱已选定标志。在处理请求项之前要首先选定软驱。
124 struct task_struct * wait_on_floppy_select = NULL;  // 等待选定软驱的任务队列。
125
    //// 取消选定软驱。
    // 如果函数参数指定的软驱 nr 当前并没有被选定，则显示警告信息。然后复位软驱已选定标志
    // selected，并唤醒等待选择该软驱的任务。数字输出寄存器(DOR)的低 2 位用于指定选择的软
    // 驱（0-3 对应 A-D）。
126 void floppy_deselect(unsigned int nr)
127 {
128         if (nr != (current_DOR & 3))
129                 printk("floppy_deselect: drive not selected\n\r");
130         selected = 0;                               // 复位软驱已选定标志。
131         wake_up(&wait_on_floppy_select);            // 唤醒等待的任务。
132 }
133
134 /*
135  * floppy-change is never called from an interrupt, so we can relax a bit
```

```
136  * here, sleep etc. Note that floppy-on tries to set current_DOR to point
137  * to the desired drive, but it will probably not survive the sleep if
138  * several floppies are used at the same time: thus the loop.
139  */
     /*
      * floppy-change()不是从中断程序中调用的，所以这里我们可以轻松一下，睡眠等。
      * 注意 floppy-on()会尝试设置 current_DOR 指向所需的驱动器，但当同时使用几个
      * 软盘时不能睡眠：因此此时只能使用循环方式。
      */
     //// 检测指定软驱中软盘更换情况。
     // 参数 nr 是软驱号。如果软盘更换了则返回 1，否则返回 0。
     // 该函数首先选定参数指定的软驱 nr，然后测试软盘控制器的数字输入寄存器 DIR 的值，以判
     // 断驱动器中的软盘是否被更换过。该函数由程序 fs/buffer.c 中的 check_disk_change()函
     // 数调用（第 119 行）。
140  int floppy_change(unsigned int nr)
141  {
     // 首先要让软驱中软盘旋转起来并达到正常工作转速。这需要花费一定时间。采用的方法是利
     // 用 kernel/sched.c 中软盘定时函数 do_floppy_timer()进行一定的延时处理。floppy_on()
     // 函数则用于判断延时是否到（mon_timer[nr]==0?），若没有到则让当前进程继续睡眠等待。
     // 若延时到则 do_floppy_timer()会唤醒当前进程。
142  repeat:
143          floppy_on(nr);          // 启动并等待指定软驱 nr（kernel/sched.c，第 251 行）。
     // 在软盘启动（旋转）之后，我们来查看一下当前选择的软驱是不是函数参数指定的软驱 nr。
     // 如果当前选择的软驱不是指定的软驱 nr，并且已经选定了其他软驱，则让当前任务进入可
     // 中断等待状态，以等待其他软驱被取消选定。参见上面 floppy_deselect()。 如果当前没
     // 有选择其他软驱或者其他软驱被取消选定而使当前任务被唤醒时， 当前软驱仍然不是指定
     // 的软驱 nr，则跳转到函数开始处重新循环等待。
144          while ((current_DOR & 3) != nr && selected)
145                  sleep_on(&wait_on_floppy_select);
146          if ((current_DOR & 3) != nr)
147                  goto repeat;
     // 现在软盘控制器已选定我们指定的软驱 nr。于是取数字输入寄存器 DIR 的值，如果其最高
     // 位（位 7）置位，则表示软盘已更换，此时即可关闭马达并返回 1 退出。 否则关闭马达返
     // 回 0 退出。表示磁盘没有被更换。
148          if (inb(FD_DIR) & 0x80) {
149                  floppy_off(nr);
150                  return 1;
151          }
152          floppy_off(nr);
153          return 0;
154  }
155
     //// 复制内存缓冲块，共 1024 字节。
     // 从内存地址 from 处复制 1024 字节数据到地址 to 处。
156  #define copy_buffer(from,to) \
157  __asm__("cld ; rep ; movsl" \
158          ::"c" (BLOCK_SIZE/4),"S" ((long)(from)),"D" ((long)(to)) \
159          :"cx","di","si")
160
     //// 设置（初始化）软盘 DMA 通道。
     // 软盘中数据读写操作是使用 DMA 进行的。因此在每次进行数据传输之前需要设置 DMA 芯片
     // 上专门用于软驱的通道 2。有关 DMA 编程方法请参见程序列表后的信息。
161  static void setup_DMA(void)
```

```
162 {
163         long addr = (long) CURRENT->buffer;       // 当前请求项缓冲区所处内存地址。
164
    // 首先检测请求项的缓冲区所在位置。如果缓冲区处于内存 1MB 以上的某个地方，则需要将
    // DMA 缓冲区设在临时缓冲区域（tmp_floppy_area）处。因为 8237A 芯片只能在 1MB 地址范
    // 围内寻址。如果是写盘命令，则还需要把数据从请求项缓冲区复制到该临时区域。
165         cli();
166         if (addr >= 0x100000) {
167                 addr = (long) tmp_floppy_area;
168                 if (command == FD_WRITE)
169                         copy_buffer(CURRENT->buffer,tmp_floppy_area);
170         }
    // 接下来我们开始设置 DMA 通道 2。在开始设置之前需要先屏蔽该通道。单通道屏蔽寄存器
    // 端口为 0x0A。位 0-1 指定 DMA 通道(0--3)，位 2：1 表示屏蔽，0 表示允许请求。 然后向
    // DMA 控制器端口 12 和 11 写入方式字（读盘是 0x46，写盘则是 0x4A）。 再写入传输使用
    // 缓冲区地址 addr 和需要传输的字节数 0x3ff（0--1023）。最后复位对 DMA 通道 2 的屏蔽，
    // 开放 DMA2 请求 DREQ 信号。
171 /* mask DMA 2 */    /* 屏蔽 DMA 通道 2 */
172         immoutb_p(4|2,10);
173 /* output command byte. I don't know why, but everyone (minix, */
174 /* sanches & canton) output this twice, first to 12 then to 11 */
    /* 输出命令字节。我是不知道为什么，但是每个人（minix，*/
    /* sanches 和 canton）都输出两次，首先是 12 口，然后是 11 口 */
    // 下面嵌入汇编代码向 DMA 控制器的“清除先后触发器”端口 12 和方式寄存器端口 11 写入
    // 方式字（读盘时是 0x46，写盘是 0x4A）。
    // 由于各通道的地址和计数寄存器都是 16 位的，因此在设置他们时都需要分 2 次进行操作。
    // 一次访问低字节，另一次访问高字节。而实际在写哪个字节则由先后触发器的状态决定。
    // 当触发器为 0 时，则访问低字节；当字节触发器为 1 时，则访问高字节。每访问一次，
    // 该触发器的状态就变化一次。而写端口 12 就可以将触发器置成 0 状态，从而对 16 位寄存
    // 器的设置从低字节开始。
175         __asm__("outb %%al,$12\n\tjmp 1f\n1:\tjmp 1f\n1:\t"
176         "outb %%al,$11\n\tjmp 1f\n1:\tjmp 1f\n1:"::
177         "a" ((char) ((command == FD_READ)?DMA_READ:DMA_WRITE)));
178 /* 8 low bits of addr */     /* 地址低 0-7 位 */
    // 向 DMA 通道 2 写入基/当前地址寄存器（端口 4）。
179         immoutb_p(addr,4);
180         addr >>= 8;
181 /* bits 8-15 of addr */    /* 地址高 8-15 位 */
182         immoutb_p(addr,4);
183         addr >>= 8;
184 /* bits 16-19 of addr */    /* 地址 16-19 位 */
    // DMA 只可以在 1MB 内存空间内寻址，其高 16-19 位地址需放入页面寄存器(端口 0x81)。
185         immoutb_p(addr,0x81);
186 /* low 8 bits of count-1 (1024-1=0x3ff) */  /* 计数器低 8 位(1024-1 = 0x3ff) */
    // 向 DMA 通道 2 写入基/当前字节计数器值（端口 5）。
187         immoutb_p(0xff,5);
188 /* high 8 bits of count-1 */    /* 计数器高 8 位 */
    // 一次共传输 1024 字节（两个扇区）。
189         immoutb_p(3,5);
190 /* activate DMA 2 */    /* 开启 DMA 通道 2 的请求 */
191         immoutb_p(0|2,10);
192         sti();
193 }
```

```
194
    //// 向软驱控制器输出一个字节命令或参数。
    // 在向控制器发送一个字节之前，控制器需要处于准备好状态，并且数据传输方向必须设置
    // 成从 CPU 到 FDC，因此函数需要首先读取控制器状态信息。这里使用了循环查询方式，以
    // 作适当延时。若出错，则会设置复位标志 reset。
195 static void output_byte(char byte)
196 {
197         int counter;
198         unsigned char status;
199
    // 循环读取主状态控制器 FD_STATUS（0x3f4）的状态。如果所读状态是 STATUS_READY 并且
    // 方向位 STATUS_DIR = 0（CPU→FDC），则向数据端口输出指定字节。
200         if (reset)
201                 return;
202         for(counter = 0 ; counter < 10000 ; counter++) {
203                 status = inb_p(FD_STATUS) & (STATUS_READY | STATUS_DIR);
204                 if (status == STATUS_READY) {
205                         outb(byte,FD_DATA);
206                         return;
207                 }
208         }
    // 如果到循环 1 万次结束还不能发送，则置复位标志，并打印出错信息。
209         reset = 1;
210         printk("Unable to send byte to FDC\n\r");
211 }
212
    //// 读取 FDC 执行的结果信息。
    // 结果信息最多 7 个字节，存放在数组 reply_buffer[]中。返回读入的结果字节数，若返回
    // 值 = -1，则表示出错。程序处理方式与上面函数类似。
213 static int result(void)
214 {
215         int i = 0, counter, status;
216
    // 若复位标志已置位，则立刻退出。去执行后续程序中的复位操作。否则循环读取主状态控
    // 制器 FD_STATUS（0x3f4）的状态。如果读取的控制器状态是 READY，表示已经没有数据可
    // 取，则返回已读取的字节数 i。如果控制器状态是方向标志置位（CPU←FDC）、已准备好、
    // 忙，表示有数据可读取。 于是把控制器中的结果数据读入到应答结果数组中。 最多读取
    // MAX_REPLIES（7）个字节。
217         if (reset)
218                 return -1;
219         for (counter = 0 ; counter < 10000 ; counter++) {
220                 status = inb_p(FD_STATUS)&(STATUS_DIR|STATUS_READY|STATUS_BUSY);
221                 if (status == STATUS_READY)
222                         return i;
223                 if (status == (STATUS_DIR|STATUS_READY|STATUS_BUSY)) {
224                         if (i >= MAX_REPLIES)
225                                 break;
226                         reply_buffer[i++] = inb_p(FD_DATA);
227                 }
228         }
    // 如果到循环 1 万次结束还不能发送，则置复位标志，并打印出错信息。
229         reset = 1;
230         printk("Getstatus times out\n\r");
```

```
231         return -1;
232 }
233
    //// 软盘读写出错处理函数。
    // 该函数根据软盘读写出错次数来确定需要采取的进一步行动。如果当前处理的请求项出错
    // 次数大于规定的最大出错次数 MAX_ERRORS（8 次），则不再对当前请求项作进一步的操作
    // 尝试。如果读/写出错次数已经超过 MAX_ERRORS/2，则需要对软驱作复位处理，于是设置
    // 复位标志 reset。否则若出错次数还不到最大值的一半，则只需重新校正一下磁头位置，
    // 于是设置重新校正标志 recalibrate。真正的复位和重新校正处理会在后续的程序中进行。
234 static void bad_flp_intr(void)
235 {
    // 首先把当前请求项出错次数增 1。如果当前请求项出错次数大于最大允许出错次数，则取
    // 消选定当前软驱，并结束该请求项（缓冲区内容没有被更新）。
236         CURRENT->errors++;
237         if (CURRENT->errors > MAX_ERRORS) {
238                 floppy_deselect(current_drive);
239                 end_request(0);
240         }
    // 如果当前请求项出错次数大于最大允许出错次数的一半，则置复位标志，需对软驱进行复
    // 位操作，然后再试。否则软驱需重新校正一下再试。
241         if (CURRENT->errors > MAX_ERRORS/2)
242                 reset = 1;
243         else
244                 recalibrate = 1;
245 }
246
247 /*
248  * Ok, this interrupt is called after a DMA read/write has succeeded,
249  * so we check the results, and copy any buffers.
250  */
    /*
     * OK，下面的中断处理函数是在 DMA 读/写成功后调用的，这样我们就可以检查
     * 执行结果，并复制缓冲区中的数据。
     */
    //// 软盘读写操作中断调用函数。
    // 该函数在软驱控制器操作结束后引发的中断处理过程中被调用。函数首先读取操作结果状
    // 态信息，据此判断操作是否出现问题并作相应处理。 如果读/写操作成功，那么若请求项
    // 是读操作并且其缓冲区在内存 1MB 以上位置，则需要把数据从软盘临时缓冲区复制到请求
    // 项的缓冲区。
251 static void rw_interrupt(void)
252 {
    // 读取 FDC 执行的结果信息。如果返回结果字节数不等于 7，或者状态字节 0、1 或 2 中存在
    // 出错标志，那么若是写保护就显示出错信息，释放当前驱动器，并结束当前请求项。否则
    // 就执行出错计数处理。然后继续执行软盘请求项操作。以下状态的含义参见 fdreg.h 文件。
    // ( 0xf8 = ST0_INTR | ST0_SE | ST0_ECE | ST0_NR )
    // ( 0xbf = ST1_EOC | ST1_CRC | ST1_OR | ST1_ND | ST1_WP | ST1_MAM，应该是 0xb7)
    // ( 0x73 = ST2_CM | ST2_CRC | ST2_WC | ST2_BC | ST2_MAM )
253         if (result() != 7 || (ST0 & 0xf8) || (ST1 & 0xbf) || (ST2 & 0x73)) {
254                 if (ST1 & 0x02) {         // 0x02 = ST1_WP - Write Protected。
255                         printk("Drive %d is write protected\n\r",current_drive);
256                         floppy_deselect(current_drive);
257                         end_request(0);
258                 } else
```

```
259                         bad_flp_intr();
260                 do_fd_request();
261                 return;
262         }
    // 如果当前请求项的缓冲区位于 1MB 地址以上，则说明此次软盘读操作的内容还放在临时缓
    // 冲区内，需要复制到当前请求项的缓冲区中（因为 DMA 只能在 1MB 地址范围寻址）。最后
    // 释放当前软驱（取消选定），执行当前请求项结束处理：唤醒等待该请求项的进行，唤醒
    // 等待空闲请求项的进程（若有的话），从软驱设备请求项链表中删除本请求项。再继续执
    // 行其他软盘请求项操作。
263         if (command == FD_READ && (unsigned long)(CURRENT->buffer) >= 0x100000)
264                 copy_buffer(tmp_floppy_area,CURRENT->buffer);
265         floppy_deselect(current_drive);
266         end_request(1);
267         do_fd_request();
268 }
269
    //// 设置 DMA 通道 2 并向软盘控制器输出命令和参数（输出 1 字节命令 + 0~7 字节参数）。
    // 若 reset 标志没有置位，那么在该函数退出并且软盘控制器执行完相应读/写操作后就会
    // 产生一个软盘中断请求，并开始执行软盘中断处理程序。
270 inline void setup_rw_floppy(void)
271 {
272         setup_DMA();                    // 初始化软盘 DMA 通道。
273         do_floppy = rw_interrupt;       // 置软盘中断调用函数指针。
274         output_byte(command);           // 发送命令字节。
275         output_byte(head<<2 | current_drive);  // 参数：磁头号+驱动器号。
276         output_byte(track);             // 参数：磁道号。
277         output_byte(head);              // 参数：磁头号。
278         output_byte(sector);            // 参数：起始扇区号。
279         output_byte(2);             /* sector size = 512 */ // 参数：(N=2)512 字节。
280         output_byte(floppy->sect);  // 参数：每磁道扇区数。
281         output_byte(floppy->gap);   // 参数：扇区间隔长度。
282         output_byte(0xFF);          /* sector size (0xff when n!=0 ?) */
                                        // 参数：当 N=0 时，扇区定义的字节长度，这里无用。
    // 若上述任何一个 output_byte()操作出错，则会设置复位标志 reset。此时即会立刻去执行
    // do_fd_request()中的复位处理代码。
283         if (reset)
284                 do_fd_request();
285 }
286
287 /*
288  * This is the routine called after every seek (or recalibrate) interrupt
289  * from the floppy controller. Note that the "unexpected interrupt" routine
290  * also does a recalibrate, but doesn't come here.
291  */
    /*
     * 该子程序是在每次软盘控制器寻道（或重新校正）中断中被调用的。注意
     * "unexpected interrupt"(意外中断)子程序也会执行重新校正操作，但不在此地。
     */
    //// 寻道处理结束后中断过程中调用的 C 函数。
    // 首先发送检测中断状态命令，获得状态信息 ST0 和磁头所在磁道信息。若出错则执行错误
    // 计数检测处理或取消本次软盘操作请求项。否则根据状态信息设置当前磁道变量，然后调
    // 用函数 setup_rw_floppy()设置 DMA 并输出软盘读写命令和参数。
292 static void seek_interrupt(void)
```

```
293 {
    // 首先发送检测中断状态命令，以获取寻道操作执行的结果。该命令不带参数。返回结果信
    // 息是两个字节：ST0 和磁头当前磁道号。然后读取 FDC 执行的结果信息。 如果返回结果字
    // 节数不等于 2，或者 ST0 不为寻道结束，或者磁头所在磁道（ST1）不等于设定磁道，则说
    // 明发生了错误。于是执行检测错误计数处理，然后继续执行软盘请求项或执行复位处理。
294 /* sense drive status */    /* 检测驱动器状态 */
295         output_byte(FD_SENSEI);
296         if (result() != 2 || (ST0 & 0xF8) != 0x20 || ST1 != seek_track) {
297                 bad_flp_intr();
298                 do_fd_request();
299                 return;
300         }
    // 若寻道操作成功，则继续执行当前请求项的软盘操作，即向软盘控制器发送命令和参数。
301         current_track = ST1;        // 设置当前磁道。
302         setup_rw_floppy();          // 设置 DMA 并输出软盘操作命令和参数。
303 }
304
305 /*
306  * This routine is called when everything should be correctly set up
307  * for the transfer (ie floppy motor is on and the correct floppy is
308  * selected).
309  */
    /*
     * 该函数是在传输操作的所有信息都正确设置好后被调用的（即软驱马达已开启
     * 并且已选择了正确的软盘（软驱）。
     */
    //// 读写数据传输函数。
310 static void transfer(void)
311 {
    // 首先检查当前驱动器参数是否就是指定驱动器的参数。若不是就发送设置驱动器参数命令
    // 及相应参数（参数 1：高 4 位步进速率，低四位磁头卸载时间；参数 2：磁头加载时间）。
    // 然后判断当前数据传输速率是否与指定驱动器的一致，若不是就发送指定软驱的速率值到
    // 数据传输速率控制寄存器(FD_DCR)。
312         if (cur_spec1 != floppy->spec1) {           // 检测当前参数。
313                 cur_spec1 = floppy->spec1;
314                 output_byte(FD_SPECIFY);            // 发送设置磁盘参数命令。
315                 output_byte(cur_spec1);             /* hut etc */  // 发送参数。
316                 output_byte(6);                     /* Head load time =6ms, DMA */
317         }
318         if (cur_rate != floppy->rate)               // 检测当前速率。
319                 outb_p(cur_rate = floppy->rate,FD_DCR);
    // 若上面任何一个 output_byte()操作执行出错，则复位标志 reset 就会被置位。因此这里
    // 我们需要检测一下 reset 标志。若 reset 真的被置位了，就立刻去执行 do_fd_request()
    // 中的复位处理代码。
320         if (reset) {
321                 do_fd_request();
322                 return;
323         }
    // 如果此时寻道标志为零（即不需要寻道），则设置 DMA 并向软盘控制器发送相应操作命令
    // 和参数后返回。否则就执行寻道处理，于是首先置软盘中断处理调用函数为寻道中断函数。
    // 如果起始磁道号不等于零则发送磁头寻道命令和参数。所使用的参数即是第 112--121 行
    // 上设置的全局变量值。如果起始磁道号 seek_track 为 0，则执行重新校正命令让磁头归零
    // 位。
```

```
324         if (!seek) {
325                 setup_rw_floppy();                      // 发送命令参数块。
326                 return;
327         }
328         do_floppy = seek_interrupt;                     // 寻道中断调用的 C 函数。
329         if (seek_track) {                               // 起始磁道号。
330                 output_byte(FD_SEEK);                   // 发送磁头寻道命令。
331                 output_byte(head<<2 | current_drive);// 发送参数：磁头号+当前软驱号。
332                 output_byte(seek_track);                // 发送参数：磁道号。
333         } else {
334                 output_byte(FD_RECALIBRATE);            // 发送重新校正命令（磁头归零）。
335                 output_byte(head<<2 | current_drive);// 发送参数：磁头号+当前软驱号。
336         }
    // 同样地，若上面任何一个 output_byte()操作执行出错，则复位标志 reset 就会被置位。
    // 若 reset 真的被置位了，就立刻去执行 do_fd_request()中的复位处理代码。
337         if (reset)
338                 do_fd_request();
339 }
340
341 /*
342  * Special case - used after a unexpected interrupt (or reset)
343  */
    /*
     * 特殊情况 - 用于意外中断（或复位）处理后。
     */
    //// 软驱重新校正中断调用函数。
    // 首先发送检测中断状态命令（无参数），如果返回结果表明出错，则置复位标志。否则重新
    // 校正标志清零。然后再次执行软盘请求项处理函数作相应操作。
344 static void recal_interrupt(void)
345 {
346         output_byte(FD_SENSEI);                         // 发送检测中断状态命令。
347         if (result()!=2 || (ST0 & 0xE0) == 0x60) // 如果返回结果字节数不等于 2 或命令
348                 reset = 1;                              // 异常结束，则置复位标志。
349         else
350                 recalibrate = 0;                        // 否则复位重新校正标志。
351         do_fd_request();                                // 作相应处理。
352 }
353
    //// 意外软盘中断请求引发的软盘中断处理程序中调用的函数。
    // 首先发送检测中断状态命令（无参数），如果返回结果表明出错，则置复位标志，否则置重新
    // 校正标志。
354 void unexpected_floppy_interrupt(void)
355 {
356         output_byte(FD_SENSEI);                         // 发送检测中断状态命令。
357         if (result()!=2 || (ST0 & 0xE0) == 0x60) // 如果返回结果字节数不等于 2 或命令
358                 reset = 1;                              // 异常结束，则置复位标志。
359         else
360                 recalibrate = 1;                        // 否则置重新校正标志。
361 }
362
    //// 软盘重新校正处理函数。
    // 向软盘控制器 FDC 发送重新校正命令和参数，并复位重新校正标志。当软盘控制器执行完
    // 重新校正命令就会再其引发的软盘中断中调用 recal_interrupt()函数。
```

```
363 static void recalibrate_floppy(void)
364 {
365         recalibrate = 0;                        // 复位重新校正标志。
366         current_track = 0;                      // 当前磁道号归零。
367         do_floppy = recal_interrupt;            // 指向重新校正中断调用的 C 函数。
368         output_byte(FD_RECALIBRATE);            // 命令：重新校正。
369         output_byte(head<<2 | current_drive);   // 参数：磁头号 + 当前驱动器号。
    // 若上面任何一个 output_byte()操作执行出错，则复位标志 reset 就会被置位。因此这里
    // 我们需要检测一下 reset 标志。若 reset 真的被置位了，就立刻去执行 do_fd_request()
    // 中的复位处理代码。
370         if (reset)
371                 do_fd_request();
372 }
373
    //// 软盘控制器 FDC 复位中断调用函数。
    // 该函数会在向控制器发送了复位操作命令后引发的软盘中断处理程序中被调用。
    // 首先发送检测中断状态命令（无参数），然后读出返回的结果字节。接着发送设定软驱
    // 参数命令和相关参数，最后再次调用请求项处理函数 do_fd_request() 去执行重新校正
    // 操作。但由于执行 output_byte() 函数出错时复位标志又会被置位，因此也可能再次去
    // 执行复位处理。
374 static void reset_interrupt(void)
375 {
376         output_byte(FD_SENSEI);                 // 发送检测中断状态命令。
377         (void) result();                        // 读取命令执行结果字节。
378         output_byte(FD_SPECIFY);                // 发送设定软驱参数命令。
379         output_byte(cur_spec1);                 /* hut etc */  // 发送参数。
380         output_byte(6);                         /* Head load time =6ms, DMA */
381         do_fd_request();                        // 调用执行软盘请求。
382 }
383
384 /*
385  * reset is done by pulling bit 2 of DOR low for a while.
386  */
    /* FDC 复位是通过将数字输出寄存器(DOR)位 2 置 0 一会儿实现的 */
    //// 复位软盘控制器。
    // 该函数首先设置参数和标志，把复位标志清 0，然后把软驱变量 cur_spec1 和 cur_rate
    // 置为无效。因为复位操作后，这两个参数就需要重新设置。接着设置需要重新校正标志，
    // 并设置 FDC 执行复位操作后引发的软盘中断中调用的 C 函数 reset_interrupt()。最后
    // 把 DOR 寄存器位 2 置 0 一会儿以对软驱执行复位操作。当前数字输出寄存器 DOR 的位 2
    // 是启动/复位软驱位。
387 static void reset_floppy(void)
388 {
389         int i;
390
391         reset = 0;                              // 复位标志置 0。
392         cur_spec1 = -1;                         // 使无效。
393         cur_rate = -1;
394         recalibrate = 1;                        // 重新校正标志置位。
395         printk("Reset-floppy called\n\r");      // 显示执行软盘复位操作信息。
396         cli();                                  // 关中断。
397         do_floppy = reset_interrupt;            // 设置在中断处理程序中调用的函数。
398         outb_p(current_DOR & ~0x04,FD_DOR);     // 对软盘控制器 FDC 执行复位操作。
399         for (i=0 ; i<100 ; i++)                 // 空操作，延迟。
```

```
400                 __asm__("nop");
401         outb(current_DOR,FD_DOR);               // 再启动软盘控制器。
402         sti();                                  // 开中断。
403 }
404
    //// 软驱启动定时中断调用函数。
    // 在执行一个请求项要求的操作之前，为了等待指定软驱马达旋转起来到达正常的工作转速，
    // do_fd_request()函数为准备好的当前请求项添加了一个延时定时器。本函数即是该定时器
    // 到期时调用的函数。它首先检查数字输出寄存器(DOR)，使其选择当前指定的驱动器。然后
    // 调用执行软盘读写传输函数 transfer()。
405 static void floppy_on_interrupt(void)          // floppy_on() interrupt。
406 {
407 /* We cannot do a floppy-select, as that might sleep. We just force it */
    /* 我们不能任意设置选择的软驱，因为这可能会引起进程睡眠。我们只是迫使它自己选择 */
    // 如果当前驱动器号与数字输出寄存器 DOR 中的不同，则需要重新设置 DOR 为当前驱动器。
    // 在向数字输出寄存器输出当前 DOR 以后，使用定时器延迟 2 个滴答时间，以让命令得到执
    // 行。然后调用软盘读写传输函数 transfer()。若当前驱动器与 DOR 中的相符，那么就可以
    // 直接调用软盘读写传输函数。
408         selected = 1;                           // 置已选定当前驱动器标志。
409         if (current_drive != (current_DOR & 3)) {
410                 current_DOR &= 0xFC;
411                 current_DOR |= current_drive;
412                 outb(current_DOR,FD_DOR);       // 向数字输出寄存器输出当前 DOR。
413                 add_timer(2,&transfer);         // 添加定时器并执行传输函数。
414         } else
415                 transfer();                     // 执行软盘读写传输函数。
416 }
417
    //// 软盘读写请求项处理函数。
    // 该函数是软盘驱动程序中最主要的函数。主要作用是：①处理有复位标志或重新校正标志置
    // 位情况；②利用请求项中的设备号计算取得请求项指定软驱的参数块；③利用内河定时器启
    // 动软盘读/写操作。
418 void do_fd_request(void)
419 {
420         unsigned int block;
421
    // 首先检查是否有复位标志或重新校正标志置位，若有则本函数仅执行相关标志的处理功能
    // 后就返回。如果复位标志已置位，则执行软盘复位操作并返回。如果重新校正标志已置位，
    // 则执行软盘重新校正操作并返回。
422         seek = 0;                               // 清寻道标志。
423         if (reset) {                            // 复位标志已置位。
424                 reset_floppy();
425                 return;
426         }
427         if (recalibrate) {                      // 重新校正标志已置位。
428                 recalibrate_floppy();
429                 return;
430         }
    // 本函数的真正功能从这里开始。首先利用 blk.h 文件中的 INIT_REQUEST 宏来检测请求项的
    // 合法性，如果已没有请求项则退出（参见 blk.h,127）。然后利用请求项中的设备号取得请
    // 求项指定软驱的参数块。这个参数块将在下面用于设置软盘操作使用的全局变量参数块（参
    // 见 112 - 122 行）。请求项设备号中的软盘类型 (MINOR(CURRENT->dev)>>2) 被用作磁盘类
    // 型数组 floppy_type[]的索引值来取得指定软驱的参数块。
```

```
431         INIT_REQUEST;
432         floppy = (MINOR(CURRENT->dev)>>2) + floppy_type;

    // 下面开始设置 112--122 行上的全局变量值。如果当前驱动器号 current_drive 不是请求项
    // 中指定的驱动器号，则置标志 seek，表示在执行读/写操作之前需要先让驱动器执行寻道处
    // 理。然后把当前驱动器号设置为请求项中指定的驱动器号。
433         if (current_drive != CURRENT_DEV)    // CURRENT_DEV 是请求项中指定的软驱号。
434                 seek = 1;
435         current_drive = CURRENT_DEV;

    // 设置读写起始扇区 block。因为每次读写是以块为单位（1 块为 2 个扇区），所以起始扇区
    // 需要起码比磁盘总扇区数小 2 个扇区。否则说明这个请求项参数无效，结束该次软盘请求项
    // 去执行下一个请求项。
436         block = CURRENT->sector;             // 取当前软盘请求项中起始扇区号。
437         if (block+2 > floppy->size) {        // 如果 block + 2 大于磁盘扇区总数，
438                 end_request(0);              // 则结束本次软盘请求项。
439                 goto repeat;
440         }
    // 再求对应在磁道上的扇区号、磁头号、磁道号、搜寻磁道号（对于软驱读不同格式的盘）。
441         sector = block % floppy->sect;   // 起始扇区对每磁道扇区数取模，得磁道上扇区号。
442         block /= floppy->sect;           // 起始扇区对每磁道扇区数取整，得起始磁道数。
443         head = block % floppy->head;     // 起始磁道数对磁头数取模，得操作的磁头号。
444         track = block / floppy->head;    // 起始磁道数对磁头数取整，得操作的磁道号。
445         seek_track = track << floppy->stretch; // 相应于软驱中盘类型进行调整，得寻道号。

    // 再看看是否还需要首先执行寻道操作。如果寻道号与当前磁头所在磁道号不同，则需要进行
    // 寻道操作，于是置需要寻道标志 seek。最后我们设置执行的软盘命令 command。
446         if (seek_track != current_track)
447                 seek = 1;
448         sector++;                            // 磁盘上实际扇区计数是从 1 算起。
449         if (CURRENT->cmd == READ)            // 如果请求项是读操作，则置读命令码。
450                 command = FD_READ;
451         else if (CURRENT->cmd == WRITE)  // 如果请求项是写操作，则置写命令码。
452                 command = FD_WRITE;
453         else
454                 panic("do_fd_request: unknown command");
    // 在上面设置好 112--122 行上所有全局变量值之后，我们可以开始执行请求项操作了。该操
    // 作利用定时器来启动。因为为了能对软驱进行读写操作，需要首先启动驱动器马达并达到正
    // 常运转速度。而这需要一定的时间。因此这里利用 ticks_to_floppy_on() 来计算启动延时
    // 时间，然后使用该延时设定一个定时器。当时间到时就调用函数 floppy_on_interrupt()。
455         add_timer(ticks_to_floppy_on(current_drive),&floppy_on_interrupt);
456 }
457
    // 各种类型软驱磁盘含有的数据块总数。
458 static int floppy_sizes[] ={
459            0,    0,    0,    0,
460          360,  360 ,360,  360,
461         1200,1200,1200,1200,
462          360,  360,  360,  360,
463          720,  720,  720,  720,
464          360,  360,  360,  360,
465          720,  720,  720,  720,
466         1440,1440,1440,1440
```

```
467 };
468
    //// 软盘系统初始化。
    // 设置软盘块设备请求项的处理函数 do_fd_request()，并设置软盘中断门（int 0x26，对应
    // 硬件中断请求信号 IRQ6）。 然后取消对该中断信号的屏蔽，以允许软盘控制器 FDC 发送中
    // 断请求信号。中断描述符表 IDT 中陷阱门描述符设置宏 set_trap_gate()定义在头文件
    // include/asm/system.h 中。
469 void floppy_init(void)
470 {
    // 设置软盘中断门描述符。floppy_interrupt（kernel/sys_call.s，267 行）是其中断处理
    // 过程。中断号为 int 0x26（38），对应 8259A 芯片中断请求信号 IRQ6。
471         blk_size[MAJOR_NR] = floppy_sizes;
472         blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;  // = do_fd_request()。
473         set_trap_gate(0x26,&floppy_interrupt);          // 设置陷阱门描述符。
474         outb(inb_p(0x21)&~0x40,0x21);                   // 复位软盘中断请求屏蔽位。
475 }
476
```

第10章 字符设备程序

10.1 程序 10-1 linux/kernel/chr_drv/keyboard.S

```
/*
 *  linux/kernel/keyboard.S
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      Thanks to Alfred Leung for US keyboard patches
 *              Wolfgang Thiel for German keyboard patches
 *              Marc Corsini for the French keyboard
 */
/*
 *      感谢 Alfred Leung 添加了 US 键盘补丁程序；
 *              Wolfgang Thiel 添加了德语键盘补丁程序；
 *              Marc Corsini 添加了法文键盘补丁程序。
 */

/* KBD_FINNISH for Finnish keyboards
 * KBD_US for US-type
 * KBD_GR for German keyboards
 * KBD_FR for Frech keyboard
 */
/*
 * KBD_FINNISH 是芬兰键盘。
 * KBD_US 是美式键盘。
 * KBD_GR 是德式键盘。
 * KBD_FR 是法式键盘。
 */
#define KBD_FINNISH         // 定义使用的键盘类型。用于后面选择采用的字符映射码表。

.text
.globl _keyboard_interrupt  // 申明为全局变量，用于在初始化时设置键盘中断描述符。

/*
 * these are for the keyboard read functions
 */
/*
 * 以下这些用于读键盘操作。
 */
// size 是键盘缓冲区（缓冲队列）长度（字节数）。
/* 值必须是 2 的次方! 并且与 tty_io.c 中的值匹配!!!! */
size    = 1024          /* must be a power of two ! And MUST be the same
                           as in tty_io.c !!!! */

// 以下是键盘缓冲队列数据结构 tty_queue 中的偏移量（include/linux/tty.h，第 16 行）。
head = 4                // 缓冲区头指针字段在 tty_queue 结构中的偏移。
```

```
29 tail = 8                 // 缓冲区尾指针字段偏移。
30 proc_list = 12           // 等待该缓冲队列的进程字段偏移。
31 buf = 16                 // 缓冲区字段偏移。
32
   // 在本程序中使用了 3 个标志字节。mode 是键盘特殊键（ctrl、alt 或 caps）的按下状态标志；
   // leds 是用于表示键盘指示灯的状态标志。 e0 是当收到扫描码 0xe0 或 0xe1 时设置的标志。
   // 每个字节标志中各位的含义见如下说明：
   // (1) mode 是键盘特殊键的按下状态标志。
   // 表示大小写转换键(caps)、交换键(alt)、控制键(ctrl)和换档键(shift)的状态。
   //     位 7 caps 键按下；
   //     位 6 caps 键的状态（应该与 leds 中对应 caps 的标志位一样）；
   //     位 5 右 alt 键按下；
   //     位 4 左 alt 键按下；
   //     位 3 右 ctrl 键按下；
   //     位 2 左 ctrl 键按下；
   //     位 1 右 shift 键按下；
   //     位 0 左 shift 键按下。
   // (2) leds 是用于表示键盘指示灯的状态标志。即表示数字锁定键（num-lock）、大小写转换
   // 键（caps-lock）和滚动锁定键（scroll-lock）的 LED 发光管状态。
   //     位 7-3 全 0 不用；
   //     位 2 caps-lock；
   //     位 1 num-lock（初始置 1，也即设置数字锁定键(num-lock)发光管为亮）；
   //     位 0 scroll-lock。
   // (3) 当扫描码是 0xe0 或 0xe1 时，置该标志。表示其后还跟随着 1 个或 2 个字符扫描码。通
   // 常若收到扫描码 0xe0 则意味着还有一个字符跟随其后；若收到扫描码 0xe1 则表示后面还跟
   // 随着 2 个字符。参见程序列表后说明。
   //     位 1 =1 收到 0xe1 标志；
   //     位 0 =1 收到 0xe0 标志。
33 mode:   .byte 0          /* caps, alt, ctrl and shift mode */
34 leds:   .byte 2          /* num-lock, caps, scroll-lock mode (nom-lock on) */
35 e0:     .byte 0
36
37 /*
38  *  con_int is the real interrupt routine that reads the
39  *  keyboard scan-code and converts it into the appropriate
40  *  ascii character(s).
41  */
   /*
    * con_int 是实际的中断处理子程序，用于读键盘扫描码并将其转换
    * 成相应的 ascii 字符。[ 注：这段英文注释已过时。]
    */
   //// 键盘中断处理程序入口点。
   // 当键盘控制器接收到用户的一个按键操作时，就会向中断控制器发出一个键盘中断请求信号
   // IRQ1。当 CPU 响应该请求时就会执行键盘中断处理程序。该中断处理程序会从键盘控制器相
   // 应端口（0x60）读入按键扫描码，并调用对应的扫描码子程序进行处理。
   // 首先从端口 0x60 读取当前按键的扫描码。然后判断该扫描码是否是 0xe0 或 0xe1，如果是
   // 的话就立刻对键盘控制器作出应答，并向中断控制器发送中断结束（EOI）信号，以允许键
   // 盘控制器能继续产生中断信号，从而让我们来接收后续的字符。 如果接收到的扫描码不是
   // 这两个特殊扫描码，我们就根据扫描码值调用按键跳转表 key_table 中相应按键处理子程
   // 序，把扫描码对应的字符放入读字符缓冲队列 read_q 中。然后，在对键盘控制器作出应答
   // 并发送 EOI 信号之后，调用函数 do_tty_interrupt()（实际上是调用 copy_to_cooked()）
   // 把 read_q 中的字符经过处理后放到 secondary 辅助队列中。
42 _keyboard_interrupt:
```

```
43          pushl %eax
44          pushl %ebx
45          pushl %ecx
46          pushl %edx
47          push %ds
48          push %es
49          movl $0x10,%eax         // 将 ds、es 段寄存器置为内核数据段。
50          mov %ax,%ds
51          mov %ax,%es
52          movl _blankinterval,%eax
53          movl %eax,_blankcount   // 预置黑屏时间计数值为 blankinterval（滴答数）。
54          xorl %eax,%eax          /* %eax is scan code */    /* eax 中是扫描码 */
55          inb $0x60,%al           // 读取扫描码➔al。
56          cmpb $0xe0,%al          // 扫描码是 0xe0 吗？若是则跳转到设置 e0 标志代码处。
57          je set_e0
58          cmpb $0xe1,%al          // 扫描码是 0xe1 吗？若是则跳转到设置 e1 标志代码处。
59          je set_e1
60          call key_table(,%eax,4) // 调用键处理程序 key_table + eax*4（参见 502 行）。
61          movb $0,e0              // 返回之后复位 e0 标志。

    // 下面这段代码（55-65 行）针对使用 8255A 的 PC 标准键盘电路进行硬件复位处理。端口 0x61
    // 是 8255A 输出口 B 的地址，该输出端口的第 7 位（PB7）用于禁止和允许对键盘数据的处理。
    // 这段程序用于对收到的扫描码做出应答。方法是首先禁止键盘，然后立刻重新允许键盘工作。
62  e0_e1:  inb $0x61,%al           // 取 PPI 端口 B 状态，其位 7 用于允许/禁止（0/1）键盘。
63          jmp 1f                  // 延迟一会。
64  1:      jmp 1f
65  1:      orb $0x80,%al           // al 位 7 置位（禁止键盘工作）。
66          jmp 1f
67  1:      jmp 1f
68  1:      outb %al,$0x61          // 使 PPI PB7 位置位。
69          jmp 1f
70  1:      jmp 1f
71  1:      andb $0x7F,%al          // al 位 7 复位。
72          outb %al,$0x61          // 使 PPI PB7 位复位（允许键盘工作）。
73          movb $0x20,%al          // 向 8259 中断芯片发送 EOI(中断结束)信号。
74          outb %al,$0x20
75          pushl $0                // 控制台 tty 号=0，作为参数入栈。
76          call _do_tty_interrupt  // 将收到数据转换成规范模式并存放在规范字符缓冲队列中。
77          addl $4,%esp            // 丢弃入栈的参数，弹出保留的寄存器，并中断返回。
78          pop %es
79          pop %ds
80          popl %edx
81          popl %ecx
82          popl %ebx
83          popl %eax
84          iret
85  set_e0: movb $1,e0              // 收到扫描前导码 0xe0 时，设置 e0 标志（位 0）。
86          jmp e0_e1
87  set_e1: movb $2,e0              // 收到扫描前导码 0xe1 时，设置 e1 标志（位 1）。
88          jmp e0_e1
89
90  /*
91   * This routine fills the buffer with max 8 bytes, taken from
```

```
92  * %ebx:%eax. (%edx is high). The bytes are written in the
93  * order %al,%ah,%eal,%eah,%bl,%bh ... until %eax is zero.
94  */
    /*
     * 下面该子程序把 ebx:eax 中的最多 8 个字符添入缓冲队列中。（ebx 是
     * 高字）所写入字符的顺序是 al,ah,eal,eah,bl,bh...直到 eax 等于 0。
     */
    // 首先从缓冲队列地址表 table_list（tty_io.c，99 行）取控制台的读缓冲队列 read_q 地址。
    // 然后把 al 寄存器中的字符复制到读队列头指针处并把头指针前移 1 字节位置。若头指针移出
    // 读缓冲区的末端，就让其回绕到缓冲区开始处。 然后再看看此时缓冲队列是否已满，即比较
    // 一下队列头指针是否与尾指针相等（相等表示满）。 如果已满，就把 ebx:eax 中可能还有的
    // 其余字符全部抛弃掉。如果缓冲区还未满，就把 ebx:eax 中数据联合右移 8 个比特（即把 ah
    // 值移到 al、bl→ah、bh→bl），然后重复上面对 al 的处理过程。直到所有字符都处理完后，
    // 就保存当前头指针值，再检查一下是否有进程等待着读队列，如果有救唤醒之。
95  put_queue:
96          pushl %ecx
97          pushl %edx                  // 下句取控制台 tty 结构中读缓冲队列指针。
98          movl _table_list,%edx        # read-queue for console
99          movl head(%edx),%ecx        // 取队列头指针→ecx。
100 1:      movb %al,buf(%edx,%ecx)     // 将 al 中的字符放入头指针位置处。
101         incl %ecx                   // 头指针前移 1 字节。
102         andl $size-1,%ecx           // 调整头指针。若超出缓冲区末端则绕回开始处。
103         cmpl tail(%edx),%ecx         # buffer full - discard everything
                                        // 头指针==尾指针吗？（即缓冲队列满了吗？）
104         je 3f                       // 如果已满，则后面未放入的字符全抛弃。
105         shrdl $8,%ebx,%eax          // 将 ebx 中 8 个比特右移 8 位到 eax 中，ebx 不变。
106         je 2f                       // 还有字符吗？若没有（等于 0）则跳转。
107         shrl $8,%ebx                // 将 ebx 值右移 8 位，并跳转到标号 1 继续操作。
108         jmp 1b
109 2:      movl %ecx,head(%edx)        // 若已将所有字符都放入队列，则保存头指针。
110         movl proc_list(%edx),%ecx   // 该队列的等待进程指针？
111         testl %ecx,%ecx             // 检测是否有等待该队列的进程。
112         je 3f                       // 无，则跳转；
113         movl $0,(%ecx)              // 有，则唤醒进程（置该进程为就绪状态）。
114 3:      popl %edx
115         popl %ecx
116         ret
117
    // 从这里开始是键跳转表 key_table 中指针对应的各个按键（或松键）处理子程序。供上面第
    // 53 行语句调用。键跳转表 key_table 在第 513 行开始。
    //
    // 下面这段代码根据 ctrl 或 alt 的扫描码，分别设置模式标志 mode 中相应位。如果在该扫描
    // 码之前收到过 0xe0 扫描码（e0 标志置位），则说明按下的是键盘右边的 ctrl 或 alt 键，则
    // 对应设置 ctrl 或 alt 在模式标志 mode 中的比特位。
118 ctrl:   movb $0x04,%al              // 0x4 是 mode 中左 ctrl 键对应的比特位（位 2）。
119         jmp 1f
120 alt:    movb $0x10,%al              // 0x10 是 mode 中左 alt 键对应的比特位（位 4）。
121 1:      cmpb $0,e0                  // e0 置位了吗（按下的是右边的 ctrl/alt 键吗）？
122         je 2f                       // 不是则转。
123         addb %al,%al                // 是，则改成置相应右键标志位（位 3 或位 5）。
124 2:      orb %al,mode                // 设置 mode 标志中对应的比特位。
125         ret
    // 这段代码处理 ctrl 或 alt 键松开时的扫描码，复位模式标志 mode 中的对应比特位。在处理
```

```
    // 时要根据 e0 标志是否置位来判断是否是键盘右边的 ctrl 或 alt 键。
126 unctrl: movb $0x04,%al              // mode 中左 ctrl 键对应的比特位（位 2）。
127         jmp 1f
128 unalt:  movb $0x10,%al              // 0x10 是 mode 中左 alt 键对应的比特位（位 4）。
129 1:      cmpb $0,e0                  // e0 置位了吗（释放的是右边的 ctrl/alt 键吗）？
130         je 2f                       // 不是，则转。
131         addb %al,%al                // 是，则改成复位相应右键的标志位（位 3 或位 5）。
132 2:      notb %al                    // 复位 mode 标志中对应的比特位。
133         andb %al,mode
134         ret
135
    // 这段代码处理左、右 shift 键按下和松开时的扫描码，分别设置和复位 mode 中的相应位。
136 lshift:
137         orb $0x01,mode              // 是左 shift 键按下，设置 mode 中位 0。
138         ret
139 unlshift:
140         andb $0xfe,mode             // 是左 shift 键松开，复位 mode 中位 0。
141         ret
142 rshift:
143         orb $0x02,mode              // 是右 shift 键按下，设置 mode 中位 1。
144         ret
145 unrshift:
146         andb $0xfd,mode             // 是右 shift 键松开，复位 mode 中位 1。
147         ret
148
    // 这段代码对收到 caps 键扫描码进行处理。通过 mode 中位 7 可以知道 caps 键当前是否正处于
    // 在按下状态。若是则返回，否则就翻转 mode 标志中 caps 键按下的比特位（位 6）和 leds 标
    // 志中 caps-lock 比特位（位 2），设置 mode 标志中 caps 键已按下标志位（位 7）。
149 caps:   testb $0x80,mode            // 测试 mode 中位 7 是否已置位（即在按下状态）。
150         jne 1f                      // 如果已处于按下状态，则返回（186 行）。
151         xorb $4,leds                // 翻转 leds 标志中 caps-lock 比特位（位 2）。
152         xorb $0x40,mode             // 翻转 mode 标志中 caps 键按下的比特位（位 6）。
153         orb $0x80,mode              // 设置 mode 标志中 caps 键已按下标志位（位 7）。
    // 这段代码根据 leds 标志，开启或关闭 LED 指示器。
154 set_leds:
155         call kb_wait                // 等待键盘控制器输入缓冲空。
156         movb $0xed,%al              /* set leds command */
157         outb %al,$0x60              // 发送键盘命令 0xed 到 0x60 端口。
158         call kb_wait
159         movb leds,%al               // 取 leds 标志，作为参数。
160         outb %al,$0x60              // 发送该参数。
161         ret
162 uncaps: andb $0x7f,mode             // caps 键松开，则复位 mode 中的对应位（位 7）。
163         ret
164 scroll:
165         testb $0x03,mode            // 若此时 ctrl 键也同时按下，则
166         je 1f
167         call _show_mem              // 显示内存状态信息（mm/memory.c，457 行）。
168         jmp 2f
169 1:      call _show_state            // 否则显示进程状态信息（kernel/sched.c，45 行）。
170 2:      xorb $1,leds                // scroll 键按下，则翻转 leds 中对应位（位 0）。
171         jmp set_leds                // 根据 leds 标志重新开启或关闭 LED 指示器。
172 num:    xorb $2,leds                // num 键按下，则翻转 leds 中的对应位（位 1）。
```

```
173         jmp set_leds                // 根据 leds 标志重新开启或关闭 LED 指示器。
174
175 /*
176  *  curosr-key/numeric keypad cursor keys are handled here.
177  *  checking for numeric keypad etc.
178  */
    /*
     * 这里处理方向键/数字小键盘方向键，检测数字小键盘等。
     */
179 cursor:
180         subb $0x47,%al          // 扫描码是数字键盘上的键（其扫描码>=0x47）发出的?
181         jb 1f                   // 如果小于则不处理，返回（198 行）。
182         cmpb $12,%al            // 如果扫描码 > 0x53（0x53 - 0x47 = 12），则
183         ja 1f                   // 表示扫描码值超过 83（0x53），不处理，返回。
184         jne cur2                /* check for ctrl-alt-del */ /* 检测 ctrl-alt-del 键*/
    // 若等于 12，说明 del 键已被按下，则继续判断 ctrl 和 alt 是否也被同时按下。
185         testb $0x0c,mode        // 有 ctrl 键按下了吗？无，则跳转。
186         je cur2
187         testb $0x30,mode        // 有 alt 键按下吗?
188         jne reboot              // 有，则跳转到重启动处理（594 行）。
189 cur2:   cmpb $0x01,e0           /* e0 forces cursor movement */ /* e0 置位指光标移动*/
                                    // e0 标志置位了吗?
190         je cur                  // 置位了，则跳转光标移动处理处 cur。
191         testb $0x02,leds        /* not num-lock forces cursor */ /* num-lock 键则不许*/
                                    // 测试 leds 中标志 num-lock 键标志是否置位。
192         je cur                  // 若没有置位（num 的 LED 不亮），则也处理光标移动。
193         testb $0x03,mode        /* shift forces cursor */ /* shift 键也使光标移动 */
                                    // 测试模式标志 mode 中 shift 按下标志。
194         jne cur                 // 如果有 shift 键按下，则也进行光标移动处理。
195         xorl %ebx,%ebx          // 否则查询小数字表（199 行），取键的数字 ASCII 码。
196         movb num_table(%eax),%al        // 以 eax 作为索引值，取对应数字字符➔al。
197         jmp put_queue           // 字符放入缓冲队列中。由于要放入队列的字符数<=4，因此
198 1:      ret                     // 在执行 put_queue 前需把 ebx 清零，见 87 行上的注释。
199
    // 这段代码处理光标移动或插入删除按键。
200 cur:    movb cur_table(%eax),%al        // 取光标字符表中相应键的代表字符➔al。
201         cmpb $'9,%al            // 若字符<='9'（5、6、2 或 3），说明是上一页、下一页、
202         ja ok_cur               // 插入或删除键，则功能字符序列中要添入字符'~'。不过
203         movb $'~,%ah            // 本内核并没有对它们进行识别和处理。
204 ok_cur: shll $16,%eax          // 将 ax 中内容移到 eax 高字中。
205         movw $0x5b1b,%ax        // 把'esc ['放入 ax，与 eax 高字中字符组成移动序列。
206         xorl %ebx,%ebx          // 由于只需把 eax 中字符放入队列，因此需要把 ebx 清零。
207         jmp put_queue           // 将该字符放入缓冲队列中。
208
209 #if defined(KBD_FR)
210 num_table:
211         .ascii "789 456 1230."  // 数字小键盘上键对应的数字 ASCII 码表。
212 #else
213 num_table:
214         .ascii "789 456 1230,"
215 #endif
216 cur_table:
217         .ascii "HA5 DGC YB623"  // 小键盘上方向键或插入删除键对应的移动表示字符表。
```

```
218
219 /*
220  * this routine handles function keys
221  */
    /*
     * 下面子程序处理功能键。
     */
    // 把功能键扫描码变换成转义字符序列并存放到读队列中。
222 func:
223         subb $0x3B,%al              // 键'F1'的扫描码是 0x3B，因此 al 中是功能键索引号。
224         jb end_func                 // 如果扫描码小于 0x3b，则不处理，返回。
225         cmpb $9,%al                 // 功能键是 F1--F10?
226         jbe ok_func                 // 是，则跳转。
227         subb $18,%al                // 是功能键 F11，F12 吗? F11、F12 扫描码是 0x57、0x58。
228         cmpb $10,%al                // 是功能键 F11?
229         jb end_func                 // 不是，则不处理，返回。
230         cmpb $11,%al                // 是功能键 F12?
231         ja end_func                 // 不是，则不处理，返回。
232 ok_func:
233         testb $0x10,mode            // 左 alt 键也同时按下了吗?
234         jne alt_func                // 是则跳转处理更换虚拟控制终端。
235         cmpl $4,%ecx                /* check that there is enough room */ /*检查空间*/
236         jl end_func                 // [??]需要放入 4 个字符，如果放不下，则返回。
237         movl func_table(,%eax,4),%eax       // 取功能键对应字符序列。
238         xorl %ebx,%ebx              // 因要放入队列字符数=4，因此执行 put_queue 之前
239         jmp put_queue               // 需把 ebx 清零。
    // 处理 alt + Fn 组合键，改变虚拟控制终端。此时 eax 中是功能键索引号（F1 -- 0），对应
    // 虚拟控制终端号。
240 alt_func:
241         pushl %eax                  // 虚拟控制终端号入栈，作为参数。
242         call _change_console        // 更改当前虚拟控制终端（chr_dev/tty_io.c，87 行）。
243         popl %eax                   // 丢弃参数。
244 end_func:
245         ret
246
247 /*
248  * function keys send F1:'esc [ [ A' F2:'esc [ [ B' etc.
249  */
    /*
     * 功能键发送的扫描码，F1 键为：'esc [ [ A'， F2 键为：'esc [ [ B'等。
     */
250 func_table:
251         .long 0x415b5b1b,0x425b5b1b,0x435b5b1b,0x445b5b1b
252         .long 0x455b5b1b,0x465b5b1b,0x475b5b1b,0x485b5b1b
253         .long 0x495b5b1b,0x4a5b5b1b,0x4b5b5b1b,0x4c5b5b1b
254
    // 扫描码-ASCII 字符映射表。
    // 根据前面定义的键盘类型(FINNISH，US，GERMEN，FRANCH)，将相应键的扫描码映射到
    // ASCII 字符。
255 #if     defined(KBD_FINNISH)    // 以下是芬兰语键盘的扫描码映射表。
256 key_map:
257         .byte 0,27              // 扫描码 0x00,0x01 对应的 ASCII 码;
258         .ascii "1234567890+'"   // 扫描码 0x02,...0x0c,0x0d 对应的 ASCII 码，以下类似。
```

```
259         .byte 127,9
260         .ascii "qwertyuiop}"
261         .byte 0,13,0
262         .ascii "asdfghjkl|{"
263         .byte 0,0
264         .ascii "'zxcvbnm,.-"
265         .byte 0,'*,0,32         /* 36-39 */    /* 扫描码 0x36-0x39 对应的 ASCII 码 */
266         .fill 16,1,0            /* 3A-49 */    /* 扫描码 0x3A-0x49 对应的 ASCII 码 */
267         .byte '-,0,0,0,'+       /* 4A-4E */    /* 扫描码 0x4A-0x4E 对应的 ASCII 码 */
268         .byte 0,0,0,0,0,0,0     /* 4F-55 */    /* 扫描码 0x4F-0x55 对应的 ASCII 码 */
269         .byte '<
270         .fill 10,1,0
271
272 shift_map:                      // shift 键同时按下时的映射表。
273         .byte 0,27
274         .ascii "!\"#$%&/()=?`"
275         .byte 127,9
276         .ascii "QWERTYUIOP]^"
277         .byte 13,0
278         .ascii "ASDFGHJKL\\["
279         .byte 0,0
280         .ascii "*ZXCVBNM;:_"
281         .byte 0,'*,0,32         /* 36-39 */
282         .fill 16,1,0            /* 3A-49 */
283         .byte '-,0,0,0,'+       /* 4A-4E */
284         .byte 0,0,0,0,0,0,0     /* 4F-55 */
285         .byte '>
286         .fill 10,1,0
287
288 alt_map:                         // alt 键同时按下时的映射表。
289         .byte 0,0
290         .ascii "\0@\0$\0\0{[]}\\\0"
291         .byte 0,0
292         .byte 0,0,0,0,0,0,0,0,0,0,0
293         .byte '~,13,0
294         .byte 0,0,0,0,0,0,0,0,0,0,0
295         .byte 0,0
296         .byte 0,0,0,0,0,0,0,0,0,0,0
297         .byte 0,0,0,0           /* 36-39 */
298         .fill 16,1,0            /* 3A-49 */
299         .byte 0,0,0,0,0         /* 4A-4E */
300         .byte 0,0,0,0,0,0,0     /* 4F-55 */
301         .byte '|
302         .fill 10,1,0
303
304 #elif defined(KBD_US)           // 以下是美式键盘的扫描码映射表。
305
306 key_map:
307         .byte 0,27
308         .ascii "1234567890-="
309         .byte 127,9
310         .ascii "qwertyuiop[]"
311         .byte 13,0
```

```
312         .ascii "asdfghjkl;'"
313         .byte '`,0
314         .ascii "\\zxcvbnm,./"
315         .byte 0,'*,0,32         /* 36-39 */
316         .fill 16,1,0            /* 3A-49 */
317         .byte '-,0,0,0,'+       /* 4A-4E */
318         .byte 0,0,0,0,0,0,0     /* 4F-55 */
319         .byte '<
320         .fill 10,1,0
321
322
323 shift_map:
324         .byte 0,27
325         .ascii "!@#$%^&*()_+"
326         .byte 127,9
327         .ascii "QWERTYUIOP{}"
328         .byte 13,0
329         .ascii "ASDFGHJKL:\""
330         .byte '~,0
331         .ascii "|ZXCVBNM<>?"
332         .byte 0,'*,0,32         /* 36-39 */
333         .fill 16,1,0            /* 3A-49 */
334         .byte '-,0,0,0,'+       /* 4A-4E */
335         .byte 0,0,0,0,0,0,0     /* 4F-55 */
336         .byte '>
337         .fill 10,1,0
338
339 alt_map:
340         .byte 0,0
341         .ascii "\0@\0$\0\0{[]}\\\0"
342         .byte 0,0
343         .byte 0,0,0,0,0,0,0,0,0,0,0
344         .byte '~,13,0
345         .byte 0,0,0,0,0,0,0,0,0,0,0
346         .byte 0,0
347         .byte 0,0,0,0,0,0,0,0,0,0,0
348         .byte 0,0,0,0           /* 36-39 */
349         .fill 16,1,0            /* 3A-49 */
350         .byte 0,0,0,0,0         /* 4A-4E */
351         .byte 0,0,0,0,0,0,0     /* 4F-55 */
352         .byte '|
353         .fill 10,1,0
354
355 #elif defined(KBD_GR)           // 以下是德语键盘的扫描码映射表。
356
357 key_map:
358         .byte 0,27
359         .ascii "1234567890\\'"
360         .byte 127,9
361         .ascii "qwertzuiop@+"
362         .byte 13,0
363         .ascii "asdfghjkl[]^"
364         .byte 0,'#
```

```
365         .ascii "yxcvbnm,.-"
366         .byte 0,'*,0,32         /* 36-39 */
367         .fill 16,1,0            /* 3A-49 */
368         .byte '-,0,0,0,'+       /* 4A-4E */
369         .byte 0,0,0,0,0,0,0     /* 4F-55 */
370         .byte '<
371         .fill 10,1,0
372
373
374 shift_map:
375         .byte 0,27
376         .ascii "!\"#$%&/()=?`"
377         .byte 127,9
378         .ascii "QWERTZUIOP\\*"
379         .byte 13,0
380         .ascii "ASDFGHJKL{}~"
381         .byte 0,''
382         .ascii "YXCVBNM;:_"
383         .byte 0,'*,0,32         /* 36-39 */
384         .fill 16,1,0            /* 3A-49 */
385         .byte '-,0,0,0,'+       /* 4A-4E */
386         .byte 0,0,0,0,0,0,0     /* 4F-55 */
387         .byte '>
388         .fill 10,1,0
389
390 alt_map:
391         .byte 0,0
392         .ascii "\0@\0$\0\0{[]}\\\0"
393         .byte 0,0
394         .byte '@,0,0,0,0,0,0,0,0,0,0
395         .byte '~,13,0
396         .byte 0,0,0,0,0,0,0,0,0,0,0
397         .byte 0,0
398         .byte 0,0,0,0,0,0,0,0,0,0,0
399         .byte 0,0,0,0           /* 36-39 */
400         .fill 16,1,0            /* 3A-49 */
401         .byte 0,0,0,0,0         /* 4A-4E */
402         .byte 0,0,0,0,0,0,0     /* 4F-55 */
403         .byte '|
404         .fill 10,1,0
405
406
407 #elif defined(KBD_FR)           // 以下是法语键盘的扫描码映射表。
408
409 key_map:
410         .byte 0,27
411         .ascii "&{\"'(-}_/@)="
412         .byte 127,9
413         .ascii "azertyuiop^$"
414         .byte 13,0
415         .ascii "qsdfghjklm|"
416         .byte '`,0,42           /* coin sup gauche, don't know, [*|mu] */
417         .ascii "wxcvbn,;:!"
```

```
418         .byte 0,'*,0,32         /* 36-39 */
419         .fill 16,1,0            /* 3A-49 */
420         .byte '-,0,0,0,'+       /* 4A-4E */
421         .byte 0,0,0,0,0,0,0     /* 4F-55 */
422         .byte '<
423         .fill 10,1,0
424
425 shift_map:
426         .byte 0,27
427         .ascii "1234567890]+"
428         .byte 127,9
429         .ascii "AZERTYUIOP<>"
430         .byte 13,0
431         .ascii "QSDFGHJKLM%"
432         .byte '~,0,'#
433         .ascii "WXCVBN?./\\"
434         .byte 0,'*,0,32         /* 36-39 */
435         .fill 16,1,0            /* 3A-49 */
436         .byte '-,0,0,0,'+       /* 4A-4E */
437         .byte 0,0,0,0,0,0,0     /* 4F-55 */
438         .byte '>
439         .fill 10,1,0
440
441 alt_map:
442         .byte 0,0
443         .ascii "\0~#{[|`\\^@]}"
444         .byte 0,0
445         .byte '@,0,0,0,0,0,0,0,0,0,0
446         .byte '~,13,0
447         .byte 0,0,0,0,0,0,0,0,0,0,0
448         .byte 0,0
449         .byte 0,0,0,0,0,0,0,0,0,0,0
450         .byte 0,0,0,0           /* 36-39 */
451         .fill 16,1,0            /* 3A-49 */
452         .byte 0,0,0,0,0         /* 4A-4E */
453         .byte 0,0,0,0,0,0,0     /* 4F-55 */
454         .byte '|
455         .fill 10,1,0
456
457 #else
458 #error "KBD-type not defined"
459 #endif
460 /*
461  * do_self handles "normal" keys, ie keys that don't change meaning
462  * and which have just one character returns.
463  */
    /*
     * do_self 用于处理“普通”键，也即含义没有变化并且只有一个字符返回的键。
     */
    // 首先根据 mode 标志选择 alt_map、shift_map 或 key_map 映射表之一。
464 do_self:
465         lea alt_map,%ebx            // 取 alt 键同时按下时的映射表基址 alt_map。
466         testb $0x20,mode            /* alt-gr */  /* 右 alt 键同时按下了？ */
```

```
467         jne 1f                          // 是，则向前跳转到标号 1 处。
468         lea shift_map,%ebx              // 取 shift 键同时按下时的映射表基址 shift_map。
469         testb $0x03,mode                // 有 shift 键同时按下了吗？
470         jne 1f                          // 有，则向前跳转到标号 1 处去映射字符。
471         lea key_map,%ebx                // 否则使用普通映射表 key_map。
    // 然后根据扫描码取映射表中对应的 ASCII 字符。若没有对应字符，则返回（转 none）。
472 1:      movb (%ebx,%eax),%al            // 将扫描码作为索引值，取对应的 ASCII 码➔al。
473         orb %al,%al                     // 检测看是否有对应的 ASCII 码。
474         je none                         // 若没有(对应的 ASCII 码=0)，则返回。
    // 若 ctrl 键已按下或 caps 键锁定，并且字符在 'a'--'}'（0x61--0x7D）范围内，则将其转成
    // 大写字符（0x41--0x5D）。
475         testb $0x4c,mode                /* ctrl or caps */  /* 控制键已按下或 caps 亮？*/
476         je 2f                           // 没有，则向前跳转标号 2 处。
477         cmpb $'a,%al                    // 将 al 中的字符与'a'比较。
478         jb 2f                           // 若 al 值<'a'，则转标号 2 处。
479         cmpb $'},%al                    // 将 al 中的字符与'}'比较。
480         ja 2f                           // 若 al 值>'}'，则转标号 2 处。
481         subb $32,%al                    // 将 al 转换为大写字符（减 0x20）。
    // 若 ctrl 键已按下，并且字符在 ``--'_'（0x40--0x5F）之间，即是大写字符，则将其转换为
    // 控制字符（0x00--0x1F）。
482 2:      testb $0x0c,mode                /* ctrl */  /* ctrl 键同时按下了吗？*/
483         je 3f                           // 若没有则转标号 3。
484         cmpb $64,%al                    // 将 al 与'@'（64）字符比较，即判断字符所属范围。
485         jb 3f                           // 若值<'@'，则转标号 3。
486         cmpb $64+32,%al                 // 将 al 与``（96）字符比较，即判断字符所属范围。
487         jae 3f                          // 若值>=``，则转标号 3。
488         subb $64,%al                    // 否则 al 减 0x40，转换为 0x00--0x1f 的控制字符。
    // 若左 alt 键同时按下，则将字符的位 7 置位。即此时生成值大于 0x7f 的扩展字符集中的字符。
489 3:      testb $0x10,mode                /* left alt */  /* 左 alt 键同时按下？*/
490         je 4f                           // 没有，则转标号 4。
491         orb $0x80,%al                   // 字符的位 7 置位。
    // 将 al 中的字符放入读缓冲队列中。
492 4:      andl $0xff,%eax                 // 清 eax 的高字和 ah。
493         xorl %ebx,%ebx                  // 由于放入队列字符数<=4，因此需把 ebx 清零。
494         call put_queue                  // 将字符放入缓冲队列中。
495 none:   ret
496
497 /*
498  * minus has a routine of it's own, as a 'E0h' before
499  * the scan code for minus means that the numeric keypad
500  * slash was pushed.
501  */
    /*
     * 减号有它自己的处理子程序，因为在减号扫描码之前的 0xe0
     * 意味着按下了数字小键盘上的斜杠键。
     */
    // 注意，对于芬兰语和德语键盘，扫描码 0x35 对应的是'-'键。参见第 264 和 365 行。
502 minus:  cmpb $1,e0                      // e0 标志置位了吗？
503         jne do_self                     // 没有，则调用 do_self 对减号符进行普通处理。
504         movl $'/,%eax                   // 否则用'/'替换减号'-'➔al。
505         xorl %ebx,%ebx                  // 由于放入队列字符数<=4，因此需把 ebx 清零。
506         jmp put_queue                   // 并将字符放入缓冲队列中。
507
```

```
/*
 * This table decides which routine to call when a scan-code has been
 * gotten. Most routines just call do_self, or none, depending if
 * they are make or break.
 */
/*
 * 下面是一张子程序地址跳转表。当取得扫描码后就根据此表调用相应的扫描码
 * 处理子程序。大多数调用的子程序是 do_self，或者是 none，这起决于是按键
 * （make）还是释放键(break)。
 */
key_table:
        .long none,do_self,do_self,do_self      /* 00-03 s0 esc 1 2 */
        .long do_self,do_self,do_self,do_self   /* 04-07 3 4 5 6 */
        .long do_self,do_self,do_self,do_self   /* 08-0B 7 8 9 0 */
        .long do_self,do_self,do_self,do_self   /* 0C-0F + ' bs tab */
        .long do_self,do_self,do_self,do_self   /* 10-13 q w e r */
        .long do_self,do_self,do_self,do_self   /* 14-17 t y u i */
        .long do_self,do_self,do_self,do_self   /* 18-1B o p } ^ */
        .long do_self,ctrl,do_self,do_self      /* 1C-1F enter ctrl a s */
        .long do_self,do_self,do_self,do_self   /* 20-23 d f g h */
        .long do_self,do_self,do_self,do_self   /* 24-27 j k l | */
        .long do_self,do_self,lshift,do_self    /* 28-2B { para lshift , */
        .long do_self,do_self,do_self,do_self   /* 2C-2F z x c v */
        .long do_self,do_self,do_self,do_self   /* 30-33 b n m , */
        .long do_self,minus,rshift,do_self      /* 34-37 . - rshift * */
        .long alt,do_self,caps,func             /* 38-3B alt sp caps f1 */
        .long func,func,func,func               /* 3C-3F f2 f3 f4 f5 */
        .long func,func,func,func               /* 40-43 f6 f7 f8 f9 */
        .long func,num,scroll,cursor            /* 44-47 f10 num scr home */
        .long cursor,cursor,do_self,cursor      /* 48-4B up pgup - left */
        .long cursor,cursor,do_self,cursor      /* 4C-4F n5 right + end */
        .long cursor,cursor,cursor,cursor       /* 50-53 dn pgdn ins del */
        .long none,none,do_self,func            /* 54-57 sysreq ? < f11 */
        .long func,none,none,none               /* 58-5B f12 ? ? ? */
        .long none,none,none,none               /* 5C-5F ? ? ? ? */
        .long none,none,none,none               /* 60-63 ? ? ? ? */
        .long none,none,none,none               /* 64-67 ? ? ? ? */
        .long none,none,none,none               /* 68-6B ? ? ? ? */
        .long none,none,none,none               /* 6C-6F ? ? ? ? */
        .long none,none,none,none               /* 70-73 ? ? ? ? */
        .long none,none,none,none               /* 74-77 ? ? ? ? */
        .long none,none,none,none               /* 78-7B ? ? ? ? */
        .long none,none,none,none               /* 7C-7F ? ? ? ? */
        .long none,none,none,none               /* 80-83 ? br br br */
        .long none,none,none,none               /* 84-87 br br br br */
        .long none,none,none,none               /* 88-8B br br br br */
        .long none,none,none,none               /* 8C-8F br br br br */
        .long none,none,none,none               /* 90-93 br br br br */
        .long none,none,none,none               /* 94-97 br br br br */
        .long none,none,none,none               /* 98-9B br br br br */
        .long none,unctrl,none,none             /* 9C-9F br unctrl br br */
        .long none,none,none,none               /* A0-A3 br br br br */
        .long none,none,none,none               /* A4-A7 br br br br */
```

```
556         .long none,none,unlshift,none           /* A8-AB br br unlshift br */
557         .long none,none,none,none               /* AC-AF br br br br */
558         .long none,none,none,none               /* B0-B3 br br br br */
559         .long none,none,unrshift,none           /* B4-B7 br br unrshift br */
560         .long unalt,none,uncaps,none            /* B8-BB unalt br uncaps br */
561         .long none,none,none,none               /* BC-BF br br br br */
562         .long none,none,none,none               /* C0-C3 br br br br */
563         .long none,none,none,none               /* C4-C7 br br br br */
564         .long none,none,none,none               /* C8-CB br br br br */
565         .long none,none,none,none               /* CC-CF br br br br */
566         .long none,none,none,none               /* D0-D3 br br br br */
567         .long none,none,none,none               /* D4-D7 br br br br */
568         .long none,none,none,none               /* D8-DB br ? ? ? */
569         .long none,none,none,none               /* DC-DF ? ? ? ? */
570         .long none,none,none,none               /* E0-E3 e0 e1 ? ? */
571         .long none,none,none,none               /* E4-E7 ? ? ? ? */
572         .long none,none,none,none               /* E8-EB ? ? ? ? */
573         .long none,none,none,none               /* EC-EF ? ? ? ? */
574         .long none,none,none,none               /* F0-F3 ? ? ? ? */
575         .long none,none,none,none               /* F4-F7 ? ? ? ? */
576         .long none,none,none,none               /* F8-FB ? ? ? ? */
577         .long none,none,none,none               /* FC-FF ? ? ? ? */
578
579 /*
580  * kb_wait waits for the keyboard controller buffer to empty.
581  * there is no timeout - if the buffer doesn't empty, we hang.
582  */
    /*
     * 子程序 kb_wait 用于等待键盘控制器缓冲空。不存在超时处理 - 如果
     * 缓冲永远不空的话，程序就会永远等待(死掉)。
     */
583 kb_wait:
584         pushl %eax
585 1:      inb $0x64,%al                   // 读键盘控制器状态。
586         testb $0x02,%al                 // 测试输入缓冲器是否为空（等于 0）。
587         jne 1b                          // 若不空，则跳转循环等待。
588         popl %eax
589         ret
590 /*
591  * This routine reboots the machine by asking the keyboard
592  * controller to pulse the reset-line low.
593  */
    /*
     * 该子程序通过设置键盘控制器，向复位线输出负脉冲，使系统复
     * 位重启（reboot）。
     */
    // 该子程序往物理内存地址 0x472 处写值 0x1234。该位置是启动模式（reboot_mode）标志字。
    // 在启动过程中 ROM BIOS 会读取该启动模式标志值并根据其值来指导下一步的执行。 如果该
    // 值是 0x1234，则 BIOS 就会跳过内存检测过程而执行热启动（Warm-boot）过程。 如果若该
    // 值为 0，则执行冷启动（Cold-boot）过程。
594 reboot:
595         call kb_wait                    // 首先等待键盘控制器输入缓冲器空。
596         movw $0x1234,0x472              /* don't do memory check */ /* 不检测内存 */
```

```
597             movb $0xfc,%al              /* pulse reset and A20 low */
598             outb %al,$0x64              // 向系统复位引脚和 A20 线输出负脉冲。
599 die:        jmp die                     // 停机。
```

10.2 程序 10-2 linux/kernel/chr_drv/console.c

```
/*
 *  linux/kernel/console.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      console.c
 *
 * This module implements the console io functions
 *      'void con_init(void)'
 *      'void con_write(struct tty_queue * queue)'
 * Hopefully this will be a rather complete VT102 implementation.
 *
 * Beeping thanks to John T Kohl.
 *
 * Virtual Consoles, Screen Blanking, Screen Dumping, Color, Graphics
 *   Chars, and VT100 enhancements by Peter MacDonald.
 */
/*
 *      console.c
 *
 * 该模块实现控制台输入输出功能
 *      'void con_init(void)'
 *      'void con_write(struct tty_queue * queue)'
 * 希望这是一个非常完整的 VT102 实现。
 *
 * 感谢 John T Kohl 实现了蜂鸣指示子程序。
 *
 * 虚拟控制台、屏幕黑屏处理、屏幕拷贝、彩色处理、图形字符显示以及
 * VT100 终端增强操作由 Peter MacDonald 编制。
 */

/*
 *  NOTE!!! We sometimes disable and enable interrupts for a short while
 * (to put a word in video IO), but this will work even for keyboard
 * interrupts. We know interrupts aren't enabled when getting a keyboard
 * interrupt, as we use trap-gates. Hopefully all is well.
 */
/*
 * 注意!!! 我们有时短暂地禁止和允许中断（当输出一个字(word) 到视频 IO），但
 * 即使对于键盘中断这也是可以工作的。因为我们使用陷阱门，所以我们知道在处理
 * 一个键盘中断过程期间中断是被禁止的。希望一切均正常。
 */

/*
 * Code to check for different video-cards mostly by Galen Hunt,
 * <g-hunt@ee.utah.edu>
 */
```

```
/*
 * 检测不同显示卡的大多数代码是 Galen Hunt 编写的，
 * <g-hunt@ee.utah.edu>
 */

#include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 数据等。
#include <linux/tty.h>    // tty 头文件，定义有关 tty_io，串行通信方面的参数、常数。
#include <linux/config.h> // 内核配置头文件。定义硬盘类型（HD_TYPE）可选项。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。

#include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
#include <asm/system.h>   // 系统头文件。定义设置或修改描述符/中断门等的汇编宏。
#include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

#include <string.h>       // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
#include <errno.h>        // 错误号头文件。包含系统中各种出错号。

// 该符号常量定义终端 IO 结构的默认数据。其中符号常数请参照 include/termios.h 文件。
#define DEF_TERMIOS \
(struct termios) { \
        ICRNL, \
        OPOST | ONLCR, \
        0, \
        IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, \
        0, \
        INIT_C_CC \
}


/*
 * These are set up by the setup-routine at boot-time:
 */
/*
 * 这些是 setup 程序在引导启动系统时设置的参数：
 */
// 参见对 boot/setup.s 的注释和 setup 程序读取并保留的系统参数表。

#define ORIG_X            (*(unsigned char *)0x90000)      // 初始光标列号。
#define ORIG_Y            (*(unsigned char *)0x90001)      // 初始光标行号。
#define ORIG_VIDEO_PAGE   (*(unsigned short *)0x90004)     // 初始显示页面。
#define ORIG_VIDEO_MODE   ((*(unsigned short *)0x90006) & 0xff)          // 显示模式。
#define ORIG_VIDEO_COLS   (((*(unsigned short *)0x90006) & 0xff00) >> 8) // 屏幕列数。
#define ORIG_VIDEO_LINES  ((*(unsigned short *)0x9000e) & 0xff)          // 屏幕行数。
#define ORIG_VIDEO_EGA_AX (*(unsigned short *)0x90008)     // [??]
#define ORIG_VIDEO_EGA_BX (*(unsigned short *)0x9000a)     // 显示内存大小和色彩模式。
#define ORIG_VIDEO_EGA_CX (*(unsigned short *)0x9000c)     // 显示卡特性参数。

// 定义显示器单色/彩色显示模式类型符号常数。
#define VIDEO_TYPE_MDA    0x10    /* Monochrome Text Display     */ /* 单色文本   */
#define VIDEO_TYPE_CGA    0x11    /* CGA Display                 */ /* CGA 显示器  */
#define VIDEO_TYPE_EGAM   0x20    /* EGA/VGA in Monochrome Mode  */ /* EGA/VGA 单色*/
#define VIDEO_TYPE_EGAC   0x21    /* EGA/VGA in Color Mode       */ /* EGA/VGA 彩色*/

```

```
75 #define NPAR 16                     // 转义字符序列中最大参数个数。
76
77 int NR_CONSOLES = 0;                // 系统实际支持的虚拟控制台数量。
78
79 extern void keyboard_interrupt(void);          // 键盘中断处理程序（keyboard.S）。
80
   // 以下这些静态变量是本文件函数中使用的一些全局变量。
   // video_type;              使用的显示类型；
   // video_num_columns;       屏幕文本列数；
   // video_mem_base;          物理显示内存基地址；
   // video_mem_term;          物理显示内存末端地址；
   // video_size_row;          屏幕每行使用的字节数；
   // video_num_lines;         屏幕文本行数；
   // video_page;              初试显示页面；
   // video_port_reg;          显示控制选择寄存器端口；
   // video_port_val;          显示控制数据寄存器端口。
81 static unsigned char    video_type;              /* Type of display being used   */
82 static unsigned long    video_num_columns;       /* Number of text columns       */
83 static unsigned long    video_mem_base;          /* Base of video memory         */
84 static unsigned long    video_mem_term;          /* End of video memory          */
85 static unsigned long    video_size_row;          /* Bytes per row                */
86 static unsigned long    video_num_lines;         /* Number of test lines         */
87 static unsigned char    video_page;              /* Initial video page           */
88 static unsigned short   video_port_reg;          /* Video register select port   */
89 static unsigned short   video_port_val;          /* Video register value port    */
90 static int can_do_colour = 0;                    // 标志：可使用彩色功能。
91
   // 虚拟控制台结构。其中包含一个虚拟控制台的当前所有信息。其中 vc_origin 和 vc_scr_end
   // 是当前正在处理的虚拟控制台执行快速滚屏操作时使用的起始行和末行对应的显示内存位置。
   // vc_video_mem_start 和 vc_video_mem_end 是当前虚拟控制台使用的显示内存区域部分。
   // vc -- Virtual Console。
92 static struct {
93         unsigned short  vc_video_erase_char;    // 擦除字符属性及字符（0x0720）
94         unsigned char   vc_attr;                // 字符属性。
95         unsigned char   vc_def_attr;            // 默认字符属性。
96         int             vc_bold_attr;           // 粗体字符属性。
97         unsigned long   vc_ques;                // 问号字符。
98         unsigned long   vc_state;               // 处理转义或控制序列的当前状态。
99         unsigned long   vc_restate;             // 处理转义或控制序列的下一状态。
100        unsigned long   vc_checkin;
101        unsigned long   vc_origin;              /* Used for EGA/VGA fast scroll */
102        unsigned long   vc_scr_end;             /* Used for EGA/VGA fast scroll */
103        unsigned long   vc_pos;                 // 当前光标对应的显示内存位置。
104        unsigned long   vc_x,vc_y;              // 当前光标列、行值。
105        unsigned long   vc_top,vc_bottom;       // 滚动时顶行行号；底行行号。
106        unsigned long   vc_npar,vc_par[NPAR];   // 转义序列参数个数和参数数组。
107        unsigned long   vc_video_mem_start;     /* Start of video RAM           */
108        unsigned long   vc_video_mem_end;       /* End of video RAM (sort of)   */
109        unsigned int    vc_saved_x;             // 保存的光标列号。
110        unsigned int    vc_saved_y;             // 保存的光标行号。
111        unsigned int    vc_iscolor;             // 彩色显示标志。
112        char *          vc_translate;           // 使用的字符集。
113 } vc_cons [MAX_CONSOLES];
```

```
114
    // 为了便于引用，以下定义当前正在处理控制台信息的符号。含义同上。其中 currcons 是使用
    // vc_cons[]结构的函数参数中的当前虚拟终端号。
115 #define origin          (vc_cons[currcons].vc_origin)  // 快速滚屏操作起始内存位置。
116 #define scr_end         (vc_cons[currcons].vc_scr_end) // 快速滚屏操作末端内存位置。
117 #define pos             (vc_cons[currcons].vc_pos)
118 #define top             (vc_cons[currcons].vc_top)
119 #define bottom          (vc_cons[currcons].vc_bottom)
120 #define x               (vc_cons[currcons].vc_x)
121 #define y               (vc_cons[currcons].vc_y)
122 #define state           (vc_cons[currcons].vc_state)
123 #define restate         (vc_cons[currcons].vc_restate)
124 #define checkin         (vc_cons[currcons].vc_checkin)
125 #define npar            (vc_cons[currcons].vc_npar)
126 #define par             (vc_cons[currcons].vc_par)
127 #define ques            (vc_cons[currcons].vc_ques)
128 #define attr            (vc_cons[currcons].vc_attr)
129 #define saved_x         (vc_cons[currcons].vc_saved_x)
130 #define saved_y         (vc_cons[currcons].vc_saved_y)
131 #define translate       (vc_cons[currcons].vc_translate)
132 #define video_mem_start (vc_cons[currcons].vc_video_mem_start)  // 使用显存的起始位置。
133 #define video_mem_end   (vc_cons[currcons].vc_video_mem_end)    // 使用显存的末端位置。
134 #define def_attr        (vc_cons[currcons].vc_def_attr)
135 #define video_erase_char  (vc_cons[currcons].vc_video_erase_char)
136 #define iscolor         (vc_cons[currcons].vc_iscolor)
137
138 int blankinterval = 0;                  // 设定的屏幕黑屏间隔时间。
139 int blankcount = 0;                     // 黑屏时间计数。
140
141 static void sysbeep(void);              // 系统蜂鸣函数。
142
143 /*
144  * this is what the terminal answers to a ESC-Z or csi0c
145  * query (= vt100 response).
146  */
    /*
     * 下面是终端回应 ESC-Z 或 csi0c 请求的应答（=vt100 响应）。
     */
    // csi - 控制序列引导码(Control Sequence Introducer)。
    // 主机通过发送不带参数或参数是 0 的设备属性（DA）控制序列（ 'ESC [c' 或 'ESC [0c' ）
    // 要求终端应答一个设备属性控制序列（ESC Z 的作用与此相同），终端则发送以下序列来响应
    // 主机。该序列（即 'ESC [?1;2c' ）表示终端是具有高级视频功能的 VT100 兼容终端。
147 #define RESPONSE "\033[?1;2c"
148
    // 定义使用的字符集。其中上半部分时普通 7 比特 ASCII 代码，即 US 字符集。下半部分对应
    // VT100 终端设备中的线条字符，即显示图表线条的字符集。
149 static char * translations[] = {
150 /* normal 7-bit ascii */
151         " !\"#$%&'()*+,-./0123456789:;<=>?"
152         "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_"
153         "`abcdefghijklmnopqrstuvwxyz{|}~ ",
154 /* vt100 graphics */
155         " !\"#$%&'()*+,-./0123456789:;<=>?"
```

```
156         "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^ "
157         "\004\261\007\007\007\007\370\361\007\007\275\267\326\323\327\304"
158         "\304\304\304\304\307\266\320\322\272\363\362\343\\007\234\007 "
159 };
160
161 #define NORM_TRANS (translations[0])
162 #define GRAF_TRANS (translations[1])
163
    //// 跟踪光标当前位置。
    // 参数：currcons - 当前虚拟终端号；new_x - 光标所在列号；new_y - 光标所在行号。
    // 更新当前光标位置变量 x,y，并修正光标在显示内存中的对应位置 pos。该函数会首先检查
    // 参数的有效性。如果给定的光标列号超出显示器最大列数，或者光标行号不低于显示的最大
    // 行数，则退出。否则就更新当前光标变量和新光标位置对应在显示内存中位置 pos。
    // 注意，函数中的所有变量实际上是 vc_cons[currcons]结构中的相应字段。以下函数相同。
164 /* NOTE! gotoxy thinks x==video_num_columns is ok */
    /* 注意！gotoxy 函数认为 x==video_num_columns 时是正确的 */
165 static inline void gotoxy(int currcons, int new_x,unsigned int new_y)
166 {
167         if (new_x > video_num_columns || new_y >= video_num_lines)
168                 return;
169         x = new_x;
170         y = new_y;
171         pos = origin + y*video_size_row + (x<<1);    // 1 列用 2 个字节表示，所以 x<<1。
172 }
173
    //// 设置滚屏起始显示内存地址。
    // 再次提醒，函数中变量基本上都是 vc_cons[currcons] 结构中的相应字段。
174 static inline void set_origin(int currcons)
175 {
    // 首先判断显示卡类型。 对于 EGA/VGA 卡，我们可以指定屏内行范围（区域）进行滚屏操作，
    // 而 MDA 单色显示卡只能进行整屏滚屏操作。因此只有 EGA/VGA 卡才需要设置滚屏起始行显示
    // 内存地址（起始行是 origin 对应的行）。即显示类型如果不是 EGA/VGA 彩色模式，也不是
    // EGA/VGA 单色模式，那么就直接返回。另外，我们只对前台控制台进行操作，因此当前控制台
    // currcons 必须是前台控制台时，我们才需要设置其滚屏起始行对应的内存起点位置。
176         if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
177                 return;
178         if (currcons != fg_console)
179                 return;
    // 然后向显示寄存器选择端口 video_port_reg 输出 12，即选择显示控制数据寄存器 r12，接着
    // 写入滚屏起始地址高字节。其中向右移动 9 位，实际上表示向右移动 8 位再除以 2（屏幕上 1
    // 个字符用 2 字节表示）。再选择显示控制数据寄存器 r13，然后写入滚屏起始地址低字节。向
    // 右移动 1 位表示除以 2，同样代表屏幕上 1 个字符用 2 字节表示。输出值相对于默认显示内存
    // 起始位置 video_mem_base 进行操作，例如对于 EGA/VGA 彩色模式，viedo_mem_base = 物理
    // 内存地址 0xb8000。
180         cli();
181         outb_p(12, video_port_reg);     // 选择数据寄存器 r12，输出滚屏起始位置高字节。
182         outb_p(0xff&((origin-video_mem_base)>>9), video_port_val);
183         outb_p(13, video_port_reg);     // 选择数据寄存器 r13，输出滚屏起始位置低字节。
184         outb_p(0xff&((origin-video_mem_base)>>1), video_port_val);
185         sti();
186 }
187
    //// 向上卷动一行。
```

```
    // 将屏幕滚动窗口向下移动一行，并在屏幕滚动区域底出现的新行上添加空格字符。滚屏区域
    // 必须大于 1 行。参见程序列表后说明。
188 static void scrup(int currcons)
189 {
    // 滚屏区域必须起码有 2 行。如果滚屏区域顶行号大于等于区域底行号，则不满足进行滚行操作
    // 的条件。另外，对于 EGA/VGA 卡，我们可以指定屏内行范围（区域）进行滚屏操作，而 MDA 单
    // 色显示卡只能进行整屏滚屏操作。该函数对 EGA 和 MDA 显示类型进行分别处理。如果显示类型
    // 是 EGA，则还分为整屏窗口移动和区域内窗口移动。这里首先处理显示卡是 EGA/VGA 显示类型
    // 的情况。
190         if (bottom<=top)
191                 return;
192         if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
193         {
    // 如果移动起始行 top=0，移动最底行 bottom = video_num_lines = 25，则表示整屏窗口向下
    // 移动。于是把整个屏幕窗口左上角对应的起始内存位置 origin 调整为向下移一行对应的内存
    // 位置，同时也跟踪调整当前光标对应的内存位置以及屏幕末行末端字符指针 scr_end 的位置。
    // 最后把新屏幕窗口内存起始位置值 origin 写入显示控制器中。
194                 if (!top && bottom == video_num_lines) {
195                         origin += video_size_row;
196                         pos += video_size_row;
197                         scr_end += video_size_row;
    // 如果屏幕窗口末端所对应的显示内存指针 scr_end 超出了实际显示内存末端，则将屏幕内容
    // 除第一行以外所有行对应的内存数据移动到显示内存的起始位置 video_mem_start 处，并在
    // 整屏窗口向下移动出现的新行上填入空格字符。然后根据屏幕内存数据移动后的情况，重新
    // 调整当前屏幕对应内存的起始指针、光标位置指针和屏幕末端对应内存指针 scr_end。
    // 这段嵌入汇编程序首先将（屏幕字符行数 - 1）行对应的内存数据移动到显示内存起始位置
    // video_mem_start 处，然后在随后的内存位置处添加一行空格（擦除）字符数据。
    // %0 -eax(擦除字符+属性)；%1 -ecx（(屏幕字符行数-1)所对应的字符数/2，以长字移动）；
    // %2 -edi(显示内存起始位置 video_mem_start)；%3 -esi(屏幕窗口内存起始位置 origin)。
    // 移动方向：[edi]➔[esi]，移动 ecx 个长字。
198                         if (scr_end > video_mem_end) {
199                                 __asm__("cld\n\t"               // 清方向位。
200                                         "rep\n\t"               // 重复操作，将当前屏幕内存
201                                         "movsl\n\t"             // 数据移动到显示内存起始处。
202                                         "movl _video_num_columns,%1\n\t"
203                                         "rep\n\t"               // 在新行上填入空格字符。
204                                         "stosw"
205                                         ::"a" (video_erase_char),
206                                         "c" ((video_num_lines-1)*video_num_columns>>1),
207                                         "D" (video_mem_start),
208                                         "S" (origin)
209                                         :"cx","di","si");
210                                 scr_end -= origin-video_mem_start;
211                                 pos -= origin-video_mem_start;
212                                 origin = video_mem_start;
    // 如果调整后的屏幕末端对应的内存指针 scr_end 没有超出显示内存的末端 video_mem_end，
    // 则只需在新行上填入擦除字符（空格字符）。
    // %0 -eax(擦除字符+属性)；%1 -ecx(屏幕行数)；%2 - edi（最后 1 行开始处对应内存位置）；
213                         } else {
214                                 __asm__("cld\n\t"
215                                         "rep\n\t"               // 重复操作，在新出现行上
216                                         "stosw"                 // 填入擦除字符(空格字符)。
217                                         ::"a" (video_erase_char),
```

```
218                                         "c" (video_num_columns),
219                                         "D" (scr_end-video_size_row)
220                                         :"cx","di");
221                         }
    // 然后把新屏幕滚动窗口内存起始位置值 origin 写入显示控制器中。
222                         set_origin(currcons);
    // 否则表示不是整屏移动。即表示从指定行 top 开始到 bottom 区域中的所有行向上移动 1 行，
    // 指定行 top 被删除。此时直接将屏幕从指定行 top 到屏幕末端所有行对应的显示内存数据向
    // 上移动 1 行，并在最下面新出现的行上填入擦除字符。
    // %0 - eax(擦除字符+属性)；%1 - ecx(top 行下 1 行开始到 bottom 行所对应的内存长字数)；
    // %2 - edi(top 行所处的内存位置)；%3 - esi(top+1 行所处的内存位置)。
223                 } else {
224                         __asm__("cld\n\t"
225                                 "rep\n\t"           // 循环操作，将 top+1 到 bottom 行
226                                 "movsl\n\t"         // 所对应的内存块移到 top 行开始处。
227                                 "movl _video_num_columns,%%ecx\n\t"
228                                 "rep\n\t"           // 在新行上填入擦除字符。
229                                 "stosw"
230                                 ::"a" (video_erase_char),
231                                 "c" ((bottom-top-1)*video_num_columns>>1),
232                                 "D" (origin+video_size_row*top),
233                                 "S" (origin+video_size_row*(top+1))
234                                 :"cx","di","si");
235                 }
236         }
    // 如果显示类型不是 EGA（而是 MDA ），则执行下面移动操作。因为 MDA 显示控制卡只能整屏滚
    // 动，并且会自动调整超出显示范围的情况，即会自动翻卷指针，所以这里不对屏幕内容对应内
    // 存超出显示内存的情况单独处理。处理方法与 EGA 非整屏移动情况完全一样。
237         else                /* Not EGA/VGA */
238         {
239                 __asm__("cld\n\t"
240                         "rep\n\t"
241                         "movsl\n\t"
242                         "movl _video_num_columns,%%ecx\n\t"
243                         "rep\n\t"
244                         "stosw"
245                         ::"a" (video_erase_char),
246                         "c" ((bottom-top-1)*video_num_columns>>1),
247                         "D" (origin+video_size_row*top),
248                         "S" (origin+video_size_row*(top+1))
249                         :"cx","di","si");
250         }
251 }
252
    //// 向下卷动一行。
    // 将屏幕滚动窗口向上移动一行，相应屏幕滚动区域内容向下移动 1 行。并在移动开始行的上
    // 方出现一新行。参见程序列表后说明。处理方法与 scrup()相似，只是为了在移动显示内存
    // 数据时不会出现数据覆盖的问题，复制操作是以逆向进行的，即先从屏幕倒数第 2 行的最后
    // 一个字符开始复制到最后一行，再将倒数第 3 行复制到倒数第 2 行等等。因为此时对 EGA/
    // VGA 显示类型和 MDA 类型的处理过程完全一样，所以该函数实际上没有必要写两段相同的代
    // 码。即这里 if 和 else 语句块中的操作完全一样！
253 static void scrdown(int currcons)
254 {
```

```
    // 同样，滚屏区域必须起码有 2 行。如果滚屏区域顶行号大于等于区域底行号，则不满足进行滚
    // 行操作的条件。另外，对于 EGA/VGA 卡，我们可以指定屏内行范围（区域）进行滚屏操作，而
    // MDA 单色显示卡只能进行整屏滚屏操作。由于窗口向上移动最多移动到当前控制台占用显示区
    // 域内存的起始位置，因此不会发生屏幕窗口末端所对应的显示内存指针 scr_end 超出实际显示
    // 内存末端的情况，所以这里只需要处理普通的内存数据移动情况。
255         if (bottom <= top)
256                 return;
257         if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
258         {
    // %0 - eax(擦除字符+属性)；%1 - ecx(top 行到 bottom-1 行的行数所对应的内存长字数)；
    // %2 - edi(窗口右下角最后一个长字位置)；%3 - esi(窗口倒数第 2 行最后一个长字位置)。
    // 移动方向：[esi]➔[edi]，移动 ecx 个长字。
259                 __asm__("std\n\t"               // 置方向位！！
260                         "rep\n\t"               // 重复操作，向下移动从 top 行到
261                         "movsl\n\t"             // bottom-1 行对应的内存数据。
262                         "addl $2,%%edi\n\t"     /* %edi has been decremented by 4 */
                                                    /* %edi 已减 4，因也是反向填擦除字符*/
263                         "movl _video_num_columns,%%ecx\n\t"
264                         "rep\n\t"               // 将擦除字符填入上方新行中。
265                         "stosw"
266                         ::"a" (video_erase_char),
267                         "c" ((bottom-top-1)*video_num_columns>>1),
268                         "D" (origin+video_size_row*bottom-4),
269                         "S" (origin+video_size_row*(bottom-1)-4)
270                         :"ax","cx","di","si");
271         }
    // 如果不是 EGA 显示类型，则执行以下操作（与上面完全一样）。
272         else            /* Not EGA/VGA */
273         {
274                 __asm__("std\n\t"
275                         "rep\n\t"
276                         "movsl\n\t"
277                         "addl $2,%%edi\n\t"     /* %edi has been decremented by 4 */
278                         "movl _video_num_columns,%%ecx\n\t"
279                         "rep\n\t"
280                         "stosw"
281                         ::"a" (video_erase_char),
282                         "c" ((bottom-top-1)*video_num_columns>>1),
283                         "D" (origin+video_size_row*bottom-4),
284                         "S" (origin+video_size_row*(bottom-1)-4)
285                         :"ax","cx","di","si");
286         }
287 }
288
    //// 光标在同列位置下移一行。
    // 如果光标没有处在最后一行上，则直接修改光标当前行变量 y++，并调整光标对应显示内存
    // 位置 pos（加上一行字符所对应的内存长度）。否则就需要将屏幕窗口内容上移一行。
    // 函数名称 lf（line feed 换行）是指处理控制字符 LF。
289 static void lf(int currcons)
290 {
291         if (y+1<bottom) {
292                 y++;
293                 pos += video_size_row;          // 加上屏幕一行占用内存的字节数。
```

```
294                 return;
295         }
296         scrup(currcons);                        // 将屏幕窗口内容上移一行。
297 }
298
    //// 光标在同列上移一行。
    // 如果光标不在屏幕第一行上，则直接修改光标当前行标量 y--，并调整光标对应显示内存位置
    // pos，减去屏幕上一行字符所对应的内存长度字节数。否则需要将屏幕窗口内容下移一行。
    // 函数名称 ri（reverse index 反向索引）是指控制字符 RI 或转义序列“ESC M”。
299 static void ri(int currcons)
300 {
301         if (y>top) {
302                 y--;
303                 pos -= video_size_row;          // 减去屏幕一行占用内存的字节数。
304                 return;
305         }
306         scrdown(currcons);                      // 将屏幕窗口内容下移一行。
307 }
308
    // 光标回到第 1 列（0 列）。
    // 调整光标对应内存位置 pos。光标所在列号*2 即是 0 列到光标所在列对应的内存字节长度。
    // 函数名称 cr（carriage return 回车）指明处理的控制字符是回车字符。
309 static void cr(int currcons)
310 {
311         pos -= x<<1;                            // 减去 0 列到光标处占用的内存字节数。
312         x=0;
313 }
314
    // 擦除光标前一字符（用空格替代）（del - delete 删除）。
    // 如果光标没有处在 0 列，则将光标对应内存位置 pos 后退 2 字节（对应屏幕上一个字符），
    // 然后将当前光标变量列值减 1，并将光标所在位置处字符擦除。
315 static void del(int currcons)
316 {
317         if (x) {
318                 pos -= 2;
319                 x--;
320                 *(unsigned short *)pos = video_erase_char;
321         }
322 }
323
    //// 删除屏幕上与光标位置相关的部分。
    // ANSI 控制序列：’ESC [ Ps J’（Ps =0 -删除光标处到屏幕底端；1 -删除屏幕开始到光标处；
    // 2 - 整屏删除）。本函数根据指定的控制序列具体参数值，执行与光标位置相关的删除操作，
    // 并且在擦除字符或行时光标位置不变。
    // 函数名称 csi_J （CSI - Control Sequence Introducer，即控制序列引导码）指明对控制
    // 序列“CSI Ps J”进行处理。
    // 参数：vpar - 对应上面控制序列中 Ps 的值。
324 static void csi_J(int currcons, int vpar)
325 {
326         long count __asm__("cx");               // 设为寄存器变量。
327         long start __asm__("di");
328
    // 首先根据三种情况分别设置需要删除的字符数和删除开始的显示内存位置。
```

```
329         switch (vpar) {
330                 case 0: /* erase from cursor to end of display */
331                         count = (scr_end-pos)>>1;    /* 擦除光标到屏幕底端所有字符 */
332                         start = pos;
333                         break;
334                 case 1: /* erase from start to cursor */
335                         count = (pos-origin)>>1;  /* 删除从屏幕开始到光标处的字符 */
336                         start = origin;
337                         break;
338                 case 2: /* erase whole display */     /* 删除整个屏幕上的所有字符 */
339                         count = video_num_columns * video_num_lines;
340                         start = origin;
341                         break;
342                 default:
343                         return;
344         }
    // 然后使用擦除字符填写被删除字符的地方。
    // %0 -ecx(删除的字符数 count)；%1 -edi(删除操作开始地址)；%2 -eax（填入的擦除字符）。
345         __asm__("cld\n\t"
346                 "rep\n\t"
347                 "stosw\n\t"
348                 ::"c" (count),
349                 "D" (start),"a" (video_erase_char)
350                 :"cx","di");
351 }
352
    //// 删除一行上与光标位置相关的部分。
    // ANSI 转义字符序列：’ESC [ Ps K’（Ps = 0 删除到行尾；1 从开始删除；2 整行都删除）。
    // 本函数根据参数擦除光标所在行的部分或所有字符。擦除操作从屏幕上移走字符但不影响其
    // 他字符。擦除的字符被丢弃。在擦除字符或行时光标位置不变。
    // 参数：par - 对应上面控制序列中 Ps 的值。
353 static void csi_K(int currcons, int vpar)
354 {
355         long count __asm__("cx");          // 设置寄存器变量。
356         long start __asm__("di");
357
    // 首先根据三种情况分别设置需要删除的字符数和删除开始的显示内存位置。
358         switch (vpar) {
359                 case 0: /* erase from cursor to end of line */
360                         if (x>=video_num_columns)     /* 删除光标到行尾所有字符 */
361                                 return;
362                         count = video_num_columns-x;
363                         start = pos;
364                         break;
365                 case 1: /* erase from start of line to cursor */
366                         start = pos - (x<<1);           /* 删除从行开始到光标处 */
367                         count = (x<video_num_columns)?x:video_num_columns;
368                         break;
369                 case 2: /* erase whole line */            /* 将整行字符全删除 */
370                         start = pos - (x<<1);
371                         count = video_num_columns;
372                         break;
373                 default:
```

```
374                         return;
375         }
    // 然后使用擦除字符填写删除字符的地方。
    // %0 - ecx(删除字符数 count)；%1 -edi(删除操作开始地址)；%2 -eax（填入的擦除字符）。
376         __asm__("cld\n\t"
377                 "rep\n\t"
378                 "stosw\n\t"
379                 ::"c" (count),
380                 "D" (start),"a" (video_erase_char)
381                 :"cx","di");
382 }
383
    //// 设置显示字符属性。
    // ANSI 转义序列：'ESC [ Ps;Ps m'。Ps = 0 - 默认属性；1 - 粗体并增亮；4 - 下划线；
    // 5 - 闪烁；7 - 反显；22 - 非粗体；24 - 无下划线；25 - 无闪烁；27 - 正显；
    // 30--38 - 设置前景色彩；39 - 默认前景色（White）；40--48 - 设置背景色彩；
    // 49 - 默认背景色（Black）。
    // 该控制序列根据参数设置字符显示属性。以后所有发送到终端的字符都将使用这里指定的属
    // 性，直到再次执行本控制序列重新设置字符显示的属性。
384 void csi_m(int currcons )
385 {
386         int i;
387
    // 一个控制序列中可以带有多个不同参数。参数存储在数组 par[]中。下面就根据接收到的参数
    // 个数 npar，循环处理各个参数 Ps。
    // 如果 Ps = 0，则把当前虚拟控制台随后显示的字符属性设置为默认属性 def_attr。初始化时
    // def_attr 已被设置成 0x07（黑底白字）。
    // 如果 Ps = 1，则把当前虚拟控制台随后显示的字符属性设置为粗体或增亮显示。 如果是彩色
    // 显示，则把字符属性或上 0x08 让字符高亮度显示；如果是单色显示，则让字符带下划线显示。
    // 如果 Ps = 4，则对彩色和单色显示进行不同的处理。若此时不是彩色显示方式，则让字符带
    // 下划线显示。如果是彩色显示，那么若原来 vc_bold_attr 不等于-1 时就复位其背景色；否则
    // 的话就把前景色取反。若取反后前景色与背景色相同，就把前景色增 1 而取另一种颜色。
388         for (i=0;i<=npar;i++)
389                 switch (par[i]) {
390                         case 0: attr=def_attr;break;  /* default */
391                         case 1: attr=(iscolor?attr|0x08:attr|0x0f);break;  /* bold */
392                         /*case 4: attr=attr|0x01;break;*/  /* underline */
393                         case 4: /* bold */
394                           if (!iscolor)
395                             attr |= 0x01;             // 单色则带下划线显示。
396                           else
397                           { /* check if forground == background */
398                             if (vc_cons[currcons].vc_bold_attr != -1)
399                               attr = (vc_cons[currcons].vc_bold_attr&0x0f)|(0xf0&(attr));
400                             else
401                             { short newattr = (attr&0xf0)|(0xf&(~attr));
402                               attr = ((newattr&0xf)==((attr>>4)&0xf)?
403                                 (attr&0xf0)|(((attr&0xf)+1)%0xf):
404                                 newattr);
405                             }
406                           }
407                           break;
    // 如果 Ps = 5，则把当前虚拟控制台随后显示的字符设置为闪烁，即把属性字节比特位 7 置 1。
```

```
    // 如果 Ps = 7，则把当前虚拟控制台随后显示的字符设置为反显，即把前景和背景色交换。
    // 如果 Ps = 22，则取消随后字符的高亮度显示（取消粗体显示）。
    // 如果 Ps = 24，则对于单色显示是取消随后字符的下划线显示，对于彩色显示则是取消绿色。
    // 如果 Ps = 25，则取消随后字符的闪烁显示。
    // 如果 Ps = 27，则取消随后字符的反显。
    // 如果 Ps = 39，则复位随后字符的前景色为默认前景色（白色）。
    // 如果 Ps = 49，则复位随后字符的背景色为默认背景色（黑色）。
408                         case 5: attr=attr|0x80;break;  /* blinking */
409                         case 7: attr=(attr<<4)|(attr>>4);break;  /* negative */
410                         case 22: attr=attr&0xf7;break; /* not bold */
411                         case 24: attr=attr&0xfe;break;  /* not underline */
412                         case 25: attr=attr&0x7f;break;  /* not blinking */
413                         case 27: attr=def_attr;break; /* positive image */
414                         case 39: attr=(attr & 0xf0)|(def_attr & 0x0f); break;
415                         case 49: attr=(attr & 0x0f)|(def_attr & 0xf0); break;
    // 当 Ps（par[i]）为其他值时，则是设置指定的前景色或背景色。如果 Ps = 30..37，则是设置
    // 前景色；如果 Ps=40..47，则是设置背景色。有关颜色值请参见程序后说明。
416                         default:
417                           if (!can_do_colour)
418                             break;
419                           iscolor = 1;
420                           if ((par[i]>=30) && (par[i]<=38))        // 设置前景色。
421                             attr = (attr & 0xf0) | (par[i]-30);
422                           else  /* Background color */
423                             if ((par[i]>=40) && (par[i]<=48))      // 设置背景色。
424                               attr = (attr & 0x0f) | ((par[i]-40)<<4);
425                             else
426                                 break;
427                 }
428 }
429
    //// 设置显示光标。
    // 根据光标对应显示内存位置 pos，设置显示控制器光标的显示位置。
430 static inline void set_cursor(int currcons)
431 {
    // 既然我们需要设置显示光标，说明有键盘操作，因此需要恢复进行黑屏操作的延时计数值。
    // 另外，显示光标的控制台必须是前台控制台，因此若当前处理的台号 currcons 不是前台控
    // 制台就立刻返回。
432         blankcount = blankinterval;             // 复位黑屏操作的计数值。
433         if (currcons != fg_console)
434                 return;
    // 然后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），接着
    // 写入光标当前位置高字节（向右移动 9 位表示高字节移到低字节再除以 2）。是相对于默认
    // 显示内存操作的。再使用索引寄存器选择 r15，并将光标当前位置低字节写入其中。
435         cli();
436         outb_p(14, video_port_reg);
437         outb_p(0xff&((pos-video_mem_base)>>9), video_port_val);
438         outb_p(15, video_port_reg);
439         outb_p(0xff&((pos-video_mem_base)>>1), video_port_val);
440         sti();
441 }
442
    // 隐藏光标。
```

```
    // 把光标设置到当前虚拟控制台窗口的末端，起到隐藏光标的作用。
443 static inline void hide_cursor(int currcons)
444 {
    // 首先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（向右移动 9 位表示高字节移到低字节再除以 2）。是相对于默认
    // 显示内存操作的。再使用索引寄存器选择 r15，并将光标当前位置低字节写入其中。
445         outb_p(14, video_port_reg);
446         outb_p(0xff&((scr_end-video_mem_base)>>9), video_port_val);
447         outb_p(15, video_port_reg);
448         outb_p(0xff&((scr_end-video_mem_base)>>1), video_port_val);
449 }
450
    //// 发送对 VT100 的响应序列。
    // 即为响应主机请求终端向主机发送设备属性（DA）。主机通过发送不带参数或参数是 0 的 DA
    // 控制序列（'ESC [ 0c' 或 'ESC Z'）要求终端发送一个设备属性（DA）控制序列，终端则发
    // 送 85 行上定义的应答序列（即 'ESC [?1;2c'）来响应主机的序列，该序列告诉主机本终端
    // 是具有高级视频功能的 VT100 兼容终端。处理过程是将应答序列放入读缓冲队列中，并使用
    // copy_to_cooked()函数处理后放入辅助队列中。
451 static void respond(int currcons, struct tty_struct * tty)
452 {
453         char * p = RESPONSE;                    // 定义在第 147 行上。
454
455         cli();
456         while (*p) {                            // 将应答序列放入读队列。
457                 PUTCH(*p,tty->read_q);          // 逐字符放入。include/linux/tty.h，46 行。
458                 p++;
459         }
460         sti();                                  // 转换成规范模式（放入辅助队列中）。
461         copy_to_cooked(tty);                    // tty_io.c，120 行。
462 }
463
    //// 在光标处插入一空格字符。
    // 把光标开始处的所有字符右移一格，并将擦除字符插入在光标所在处。
464 static void insert_char(int currcons)
465 {
466         int i=x;
467         unsigned short tmp, old = video_erase_char;     // 擦除字符（加属性）。
468         unsigned short * p = (unsigned short *) pos;    // 光标对应内存位置。
469
470         while (i++<video_num_columns) {
471                 tmp=*p;
472                 *p=old;
473                 old=tmp;
474                 p++;
475         }
476 }
477
    //// 在光标处插入一行。
    // 将屏幕窗口从光标所在行到窗口底的内容向下卷动一行。光标将处在新的空行上。
478 static void insert_line(int currcons)
479 {
480         int oldtop,oldbottom;
481
```

```
    // 首先保存屏幕窗口卷动开始行 top 和最后行 bottom 值，然后从光标所在行让屏幕内容向下
    // 滚动一行。最后恢复屏幕窗口卷动开始行 top 和最后行 bottom 的原来值。
482         oldtop=top;
483         oldbottom=bottom;
484         top=y;                              // 设置屏幕卷动开始行和结束行。
485         bottom = video_num_lines;
486         scrdown(currcons);                  // 从光标开始处，屏幕内容向下滚动一行。
487         top=oldtop;
488         bottom=oldbottom;
489 }
490
    //// 删除一个字符。
    // 删除光标处的一个字符，光标右边的所有字符左移一格。
491 static void delete_char(int currcons)
492 {
493         int i;
494         unsigned short * p = (unsigned short *) pos;
495
    // 如果光标的当前列位置 x 超出屏幕最右列，则返回。否则从光标右一个字符开始到行末所有
    // 字符左移一格。然后在最后一个字符处填入擦除字符。
496         if (x>=video_num_columns)
497                 return;
498         i = x;
499         while (++i < video_num_columns) {         // 光标右所有字符左移 1 格。
500                 *p = *(p+1);
501                 p++;
502         }
503         *p = video_erase_char;                    // 最后填入擦除字符。
504 }
505
    //// 删除光标所在行。
    // 删除光标所在的一行，并从光标所在行开始屏幕内容上卷一行。
506 static void delete_line(int currcons)
507 {
508         int oldtop,oldbottom;
509
    // 首先保存屏幕卷动开始行 top 和最后行 bottom 值，然后从光标所在行让屏幕内容向上滚动
    // 一行。最后恢复屏幕卷动开始行 top 和最后行 bottom 的原来值。
510         oldtop=top;
511         oldbottom=bottom;
512         top=y;                                    // 设置屏幕卷动开始行和最后行。
513         bottom = video_num_lines;
514         scrup(currcons);                          // 从光标开始处，屏幕内容向上滚动一行。
515         top=oldtop;
516         bottom=oldbottom;
517 }
518
    //// 在光标处插入 nr 个字符。
    // ANSI 转义字符序列：'ESC [ Pn @'。在当前光标处插入 1 个或多个空格字符。Pn 是插入的字
    // 符数。默认是 1。光标将仍然处于第 1 个插入的空格字符处。在光标与右边界的字符将右移。
    // 超过右边界的字符将被丢失。
    // 参数 nr = 转义字符序列中的参数 Pn。
519 static void csi_at(int currcons, unsigned int nr)
```

```
520 {
    // 如果插入的字符数大于一行字符数，则截为一行字符数；若插入字符数 nr 为 0，则插入 1 个
    // 字符。然后循环插入指定个空格字符。
521         if (nr > video_num_columns)
522                 nr = video_num_columns;
523         else if (!nr)
524                 nr = 1;
525         while (nr--)
526                 insert_char(currcons);
527 }
528
    //// 在光标位置处插入 nr 行。
    // ANSI 转义字符序列：'ESC [ Pn L'。该控制序列在光标处插入 1 行或多行空行。操作完成后
    // 光标位置不变。当空行被插入时，光标以下滚动区域内的行向下移动。滚动出显示页的行就
    // 丢失。
    // 参数 nr = 转义字符序列中的参数 Pn。
529 static void csi_L(int currcons, unsigned int nr)
530 {
    // 如果插入的行数大于屏幕最多行数，则截为屏幕显示行数；若插入行数 nr 为 0，则插入 1 行。
    // 然后循环插入指定行数 nr 的空行。
531         if (nr > video_num_lines)
532                 nr = video_num_lines;
533         else if (!nr)
534                 nr = 1;
535         while (nr--)
536                 insert_line(currcons);
537 }
538
    //// 删除光标处的 nr 个字符。
    // ANSI 转义序列：'ESC [ Pn P'。该控制序列从光标处删除 Pn 个字符。当一个字符被删除时，
    // 光标右所有字符都左移。这会在右边界处产生一个空字符。其属性应该与最后一个左移字符
    // 相同，但这里作了简化处理，仅使用字符的默认属性（黑底白字空格 0x0720）来设置空字符。
    // 参数 nr = 转义字符序列中的参数 Pn。
539 static void csi_P(int currcons, unsigned int nr)
540 {
    // 如果删除的字符数大于一行字符数，则截为一行字符数；若删除字符数 nr 为 0，则删除 1 个
    // 字符。然后循环删除光标处指定字符数 nr。
541         if (nr > video_num_columns)
542                 nr = video_num_columns;
543         else if (!nr)
544                 nr = 1;
545         while (nr--)
546                 delete_char(currcons);
547 }
548
    //// 删除光标处的 nr 行。
    // ANSI 转义序列：'ESC [ Pn M'。该控制序列在滚动区域内，从光标所在行开始删除 1 行或多
    // 行。当行被删除时，滚动区域内的被删行以下的行会向上移动，并且会在最底行添加 1 空行。
    // 若 Pn 大于显示页上剩余行数，则本序列仅删除这些剩余行，并对滚动区域外不起作用。
    // 参数 nr = 转义字符序列中的参数 Pn。
549 static void csi_M(int currcons, unsigned int nr)
550 {
    // 如果删除的行数大于屏幕最多行数，则截为屏幕显示行数；若欲删除的行数 nr 为 0，则删除
```

```
    // 1 行。然后循环删除指定行数 nr。
551         if (nr > video_num_lines)
552                 nr = video_num_lines;
553         else if (!nr)
554                 nr=1;
555         while (nr--)
556                 delete_line(currcons);
557 }
558
    //// 保存当前光标位置。
559 static void save_cur(int currcons)
560 {
561         saved_x=x;
562         saved_y=y;
563 }
564
    //// 恢复保存的光标位置。
565 static void restore_cur(int currcons)
566 {
567         gotoxy(currcons,saved_x, saved_y);
568 }
569
570
    // 这个枚举定义用于下面 con_write()函数中处理转义序列或控制序列的解析。ESnormal 是初
    // 始进入状态，也是转义或控制序列处理完毕时的状态。
    // ESnormal -  表示处于初始正常状态。此时若接收到的是普通显示字符，则把字符直接显示
    //             在屏幕上；若接收到的是控制字符（例如回车字符），则对光标位置进行设置。
    //             当刚处理完一个转义或控制序列，程序也会返回到本状态。
    // ESesc    -  表示接收到转义序列引导字符 ESC（0x1b = 033 = 27）；如果在此状态下接收
    //             到一个'['字符，则说明转义序列引导码，于是跳转到 ESsquare 去处理。否则
    //             就把接收到的字符作为转义序列来处理。对于选择字符集转义序列'ESC (' 和
    //             'ESC )'，我们使用单独的状态 ESsetgraph 来处理；对于设备控制字符串序列
    //             'ESC P'，我们使用单独的状态 ESsetterm 来处理。
    // ESsquare -  表示已经接收到一个控制序列引导码（'ESC ['），表示接收到的是一个控制序
    //             列。于是本状态执行参数数组 par[]清零初始化工作。如果此时接收到的又是一
    //             个'['字符，则表示收到了'ESC [['序列。该序列是键盘功能键发出的序列，于
    //             是跳转到 Esfunckey 去处理。否则我们需要准备接收控制序列的参数，于是置
    //             状态 Esgetpars 并直接进入该状态去接收并保存序列的参数字符。
    // ESgetpars - 该状态表示我们此时要接收控制序列的参数值。参数用十进制数表示，我们把
    //             接收到的数字字符转换成数值并保存到 par[]数组中。如果收到一个分号 ';'，
    //             则还是维持在本状态，并把接收到的参数值保存在数据 par[]下一项中。若不是
    //             数字字符或分号，说明已取得所有参数，那么就转移到状态 ESgotpars 去处理。
    // ESgotpars - 表示我们已经接收到一个完整的控制序列。此时我们可以根据本状态接收到的结
    //             尾字符对相应控制序列进行处理。不过在处理之前，如果我们在 ESsquare 状态
    //             收到过 '?'，说明这个序列是终端设备私有序列。本内核不对支持对这种序列的
    //             处理，于是我们直接恢复到 ESnormal 状态。否则就去执行相应控制序列。待序
    //             列处理完后就把状态恢复到 ESnormal。
    // ESfunckey - 表示我们接收到了键盘上功能键发出的一个序列。我们不用显示。于是恢复到正
    //             常状态 ESnormal。
    // ESsetterm - 表示处于设备控制字符串序列状态（DCS）。此时若收到字符 'S'，则恢复初始
    //             的显示字符属性。若收到的字符是'L'或'l'，则开启或关闭折行显示方式。
    // ESsetgraph -表示收到设置字符集转移序列'ESC (' 或 'ESC )'。它们分别用于指定 G0 和 G1
    //             所用的字符集。此时若收到字符 '0'，则选择图形字符集作为 G0 和 G1，若收到
```

```
    //              的字符是 'B'，这选择普通 ASCII 字符集作为 G0 和 G1 的字符集。
571 enum { ESnormal, ESesc, ESsquare, ESgetpars, ESgotpars, ESfunckey,
572         ESsetterm, ESsetgraph };
573
    //// 控制台写函数。
    // 从终端对应的 tty 写缓冲队列中取字符，针对每个字符进行分析。若是控制字符或转义或控制
    // 序列，则进行光标定位、字符删除等的控制处理；对于普通字符就直接在光标处显示。
    // 参数 tty 是当前控制台使用的 tty 结构指针。
574 void con_write(struct tty_struct * tty)
575 {
576         int nr;
577         char c;
578         int currcons;
579
    // 该函数首先根据当前控制台使用的 tty 在 tty 表中的项位置取得对应的控制台号 currcons，
    // 然后计算出（CHARS()）目前 tty 写队列中含有的字符数 nr，并循环取出其中的每个字符进行
    // 处理。不过如果当前控制台由于接收到键盘或程序发出的暂停命令（如按键 Ctrl-S）而处于
    // 停止状态，那么本函数就停止处理写队列中的字符，退出函数。另外，如果取出的是控制字符
    // CAN（24）或 SUB（26），那么若是在转义或控制序列期间收到的，则序列不会执行而立刻终
    // 止，同时显示随后的字符。注意，con_write()函数只处理取队列字符数时写队列中当前含有
    // 的字符。这有可能在一个序列被放到写队列期间读取字符数，因此本函数前一次退出时 state
    // 有可能正处于处理转义或控制序列的其他状态上。
580         currcons = tty - tty_table;
581         if ((currcons>=MAX_CONSOLES) || (currcons<0))
582                 panic("con_write: illegal tty");
583
584         nr = CHARS(tty->write_q);               // 取写队列中字符数。在 tty.h 文件中。
585         while (nr--) {
586                 if (tty->stopped)
587                         break;
588                 GETCH(tty->write_q,c);          // 取 1 字符到 c 中。
589                 if (c == 24 || c == 26)         // 控制字符 CAN、SUB -  取消、替换。
590                         state = ESnormal;
591                 switch(state) {
    // 如果从写队列中取出的字符是普通显示字符代码，就直接从当前映射字符集中取出对应的显示
    // 字符，并放到当前光标所处的显示内存位置处，即直接显示该字符。然后把光标位置右移一个
    // 字符位置。具体地，如果字符不是控制字符也不是扩展字符，即(31<c<127)，那么，若当前光
    // 标处在行末端或末端以外，则将光标移到下行头列。并调整光标位置对应的内存指针 pos。然
    // 后将字符 c 写到显示内存中 pos 处，并将光标右移 1 列，同时也将 pos 对应地移动 2 个字节。
592                         case ESnormal:
593                                 if (c>31 && c<127) {            // 是普通显示字符。
594                                         if (x>=video_num_columns) {     // 要换行？
595                                                 x -= video_num_columns;
596                                                 pos -= video_size_row;
597                                                 lf(currcons);
598                                         }
599                                         __asm__("movb %2,%%ah\n\t"      // 写字符。
600                                                 "movw %%ax,%1\n\t"
601                                                 ::"a" (translate[c-32]),
602                                                 "m" (*(short *)pos),
603                                                 "m" (attr)
604                                                 :"ax");
605                                         pos += 2;
```

```
606                                     x++;
    // 如果字符 c 是转义字符 ESC，则转换状态 state 到 ESesc（637 行）。
607                             } else if (c==27)               // ESC - 转义控制字符。
608                                     state=ESesc;
    // 如果 c 是换行符 LF(10)，或垂直制表符 VT(11)，或换页符 FF(12)，则光标移动到下 1 行。
609                             else if (c==10 || c==11 || c==12)
610                                     lf(currcons);
    // 如果 c 是回车符 CR(13)，则将光标移动到头列（0 列）。
611                             else if (c==13)                 // CR - 回车。
612                                     cr(currcons);
    // 如果 c 是 DEL(127)，则将光标左边字符擦除(用空格字符替代)，并将光标移到被擦除位置。
613                             else if (c==ERASE_CHAR(tty))
614                                     del(currcons);
    // 如果 c 是 BS(backspace,8)，则将光标左移 1 格，并相应调整光标对应内存位置指针 pos。
615                             else if (c==8) {                // BS - 后退。
616                                     if (x) {
617                                             x--;
618                                             pos -= 2;
619                                     }
    // 如果字符 c 是水平制表符 HT(9)，则将光标移到 8 的倍数列上。若此时光标列数超出屏幕最大
    // 列数，则将光标移到下一行上。
620                             } else if (c==9) {              // HT - 水平制表。
621                                     c=8-(x&7);
622                                     x += c;
623                                     pos += c<<1;
624                                     if (x>video_num_columns) {
625                                             x -= video_num_columns;
626                                             pos -= video_size_row;
627                                             lf(currcons);
628                                     }
629                                     c=9;
    // 如果字符 c 是响铃符 BEL(7)，则调用蜂鸣函数，是扬声器发声。
630                             } else if (c==7)                // BEL - 响铃。
631                                     sysbeep();
    // 如果 c 是控制字符 SO（14）或 SI（15），则相应选择字符集 G1 或 G0 作为显示字符集。
632                             else if (c == 14)               // SO - 换出，使用 G1。
633                                     translate = GRAF_TRANS;
634                             else if (c == 15)               // SI - 换进，使用 G0。
635                                     translate = NORM_TRANS;
636                             break;
    // 如果在 ESnormal 状态收到转义字符 ESC(0x1b = 033 = 27)，则转到本状态处理。该状态对 C1
    // 中控制字符或转义字符进行处理。处理完后默认的状态将是 ESnormal。
637                     case ESesc:
638                             state = ESnormal;
639                             switch (c)
640                             {
641                               case '[':             // ESC [ - 是 CSI 序列。
642                                     state=ESsquare;
643                                     break;
644                               case 'E':             // ESC E - 光标下移 1 行回 0 列。
645                                     gotoxy(currcons,0,y+1);
646                                     break;
647                               case 'M':             // ESC M - 光标下移 1 行。
```

```
648                                     ri(currcons);
649                                     break;
650                             case 'D':               // ESC D - 光标下移 1 行。
651                                     lf(currcons);
652                                     break;
653                             case 'Z':               // ESC Z - 设备属性查询。
654                                     respond(currcons,tty);
655                                     break;
656                             case '7':               // ESC 7 - 保存光标位置。
657                                     save_cur(currcons);
658                                     break;
659                             case '8':               // ESC 8 - 恢复保存的光标原位置。
660                                     restore_cur(currcons);
661                                     break;
662                             case '(':  case ')': // ESC (、ESC ) - 选择字符集。
663                                     state = ESsetgraph;
664                                     break;
665                             case 'P':               // ESC P - 设置终端参数。
666                                     state = ESsetterm;
667                                     break;
668                             case '#':               // ESC # - 修改整行属性。
669                                     state = -1;
670                                     break;
671                             case 'c':               // ESC c - 复位到终端初始设置。
672                                     tty->termios = DEF_TERMIOS;
673                                     state = restate = ESnormal;
674                                     checkin = 0;
675                                     top = 0;
676                                     bottom = video_num_lines;
677                                     break;
678                             /* case '>':   Numeric keypad */
679                             /* case '=':   Appl. keypad */
680                             }
681                             break;
    // 如果在状态 ESesc（是转义字符 ESC）时收到字符'['，则表明是 CSI 控制序列，于是转到状
    // 态 ESsequare 来处理。首先对 ESC 转义序列保存参数的数组 par[]清零，索引变量 npar 指向
    // 首项，并且设置我们开始处于取参数状态 ESgetpars。如果接收到的字符不是'?'，则直接转
    // 到状态 ESgetpars 去处理，若接收到的字符是'?'，说明这个序列是终端设备私有序列，后面
    // 会有一个功能字符。于是去读下一字符，再到状态 ESgetpars 去处理代码处。如果此时接收
    // 到的字符还是'['，那么表明收到了键盘功能键发出的序列，于是设置下一状态为 ESfunckey。
    // 否则直接进入 ESgetpars 状态继续处理。
682                     case ESsquare:
683                             for(npar=0;npar<NPAR;npar++)      // 初始化参数数组。
684                                     par[npar]=0;
685                             npar=0;
686                             state=ESgetpars;
687                             if (c =='[')  /* Function key */ // 'ESC [['是功能键。
688                             { state=ESfunckey;
689                               break;
690                             }
691                             if (ques=(c=='?'))
692                                     break;
    // 该状态表示我们此时要接收控制序列的参数值。参数用十进制数表示，我们把接收到的数字字
```

```
    // 符转换成数值并保存到 par[]数组中。如果收到一个分号 ';'，则还是维持在本状态，并把接
    // 收到的参数值保存在数据 par[]下一项中。若不是数字字符或分号，说明已取得所有参数，那
    // 么就转移到状态 ESgotpars 去处理。
693                     case ESgetpars:
694                             if (c==';' && npar<NPAR-1) {
695                                     npar++;
696                                     break;
697                             } else if (c>='0' && c<='9') {
698                                     par[npar]=10*par[npar]+c-'0';
699                                     break;
700                             } else state=ESgotpars;
    // ESgotpars 状态表示我们已经接收到一个完整的控制序列。此时我们可以根据本状态接收到的
    // 结尾字符对相应控制序列进行处理。不过在处理之前，如果我们在 ESsquare 状态收到过'?'，
    // 说明这个序列是终端设备私有序列。本内核不支持对这种序列的处理，于是我们直接恢复到
    // ESnormal 状态。否则就去执行相应控制序列。待序列处理完后就把状态恢复到 ESnormal。
701                     case ESgotpars:
702                             state = ESnormal;
703                             if (ques)
704                             { ques =0;
705                               break;
706                             }
707                             switch(c) {
    // 如果 c 是字符'G'或'`'，则 par[]中第 1 个参数代表列号。若列号不为零，则将光标左移 1 格。
708                                     case 'G': case '`':  // CSI Pn G -光标水平移动。
709                                             if (par[0]) par[0]--;
710                                             gotoxy(currcons,par[0],y);
711                                             break;
    // 如果 c 是'A'，则第 1 个参数代表光标上移的行数。若参数为 0 则上移 1 行。
712                                     case 'A':           // CSI Pn A - 光标上移。
713                                             if (!par[0]) par[0]++;
714                                             gotoxy(currcons,x,y-par[0]);
715                                             break;
    // 如果 c 是'B'或'e'，则第 1 个参数代表光标下移的行数。若参数为 0 则下移 1 行。
716                                     case 'B': case 'e':  // CSI Pn B - 光标下移。
717                                             if (!par[0]) par[0]++;
718                                             gotoxy(currcons,x,y+par[0]);
719                                             break;
    // 如果 c 是'C'或'a'，则第 1 个参数代表光标右移的格数。若参数为 0 则右移 1 格。
720                                     case 'C': case 'a':  // CSI Pn C - 光标右移。
721                                             if (!par[0]) par[0]++;
722                                             gotoxy(currcons,x+par[0],y);
723                                             break;
    // 如果 c 是'D'，则第 1 个参数代表光标左移的格数。若参数为 0 则左移 1 格。
724                                     case 'D':           // CSI Pn D - 光标左移。
725                                             if (!par[0]) par[0]++;
726                                             gotoxy(currcons,x-par[0],y);
727                                             break;
    // 如果 c 是'E'，则第 1 个参数代表光标向下移动的行数，并回到 0 列。若参数为 0 则下移 1 行。
728                                     case 'E':       // CSI Pn E - 光标下移回 0 列。
729                                             if (!par[0]) par[0]++;
730                                             gotoxy(currcons,0,y+par[0]);
731                                             break;
    // 如果 c 是'F'，则第 1 个参数代表光标向上移动的行数，并回到 0 列。若参数为 0 则上移 1 行。
```

```
732                                     case 'F':       // CSI Pn F - 光标上移回 0 列。
733                                             if (!par[0]) par[0]++;
734                                             gotoxy(currcons,0,y-par[0]);
735                                             break;
    // 如果 c 是'd'，则第 1 个参数代表光标所需在的行号（从 0 计数）。
736                                     case 'd':       // CSI Pn d - 在当前列置行位置。
737                                             if (par[0]) par[0]--;
738                                             gotoxy(currcons,x,par[0]);
739                                             break;
    // 如果 c 是'H'或'f'，则第 1 个参数代表光标移到的行号，第 2 个参数代表光标移到的列号。
740                                     case 'H': case 'f':  // CSI Pn H - 光标定位。
741                                             if (par[0]) par[0]--;
742                                             if (par[1]) par[1]--;
743                                             gotoxy(currcons,par[1],par[0]);
744                                             break;
    // 如果字符 c 是'J'，则第 1 个参数代表以光标所处位置清屏的方式：
    // 序列：'ESC [ Ps J'（Ps=0 删除光标到屏幕底端；1 删除屏幕开始到光标处；2 整屏删除）。
745                                     case 'J':       // CSI Pn J - 屏幕擦除字符。
746                                             csi_J(currcons,par[0]);
747                                             break;
    // 如果字符 c 是'K'，则第一个参数代表以光标所在位置对行中字符进行删除处理的方式。
    // 转义序列：'ESC [ Ps K'（Ps = 0 删除到行尾；1 从开始删除；2 整行都删除）。
748                                     case 'K':       // CSI Pn K - 行内擦除字符。
749                                             csi_K(currcons,par[0]);
750                                             break;
    // 如果字符 c 是'L'，表示在光标位置处插入 n 行（控制序列 'ESC [ Pn L'）。
751                                     case 'L':       // CSI Pn L - 插入行。
752                                             csi_L(currcons,par[0]);
753                                             break;
    // 如果字符 c 是'M'，表示在光标位置处删除 n 行（控制序列 'ESC [ Pn M'）。
754                                     case 'M':       // CSI Pn M - 删除行。
755                                             csi_M(currcons,par[0]);
756                                             break;
    // 如果字符 c 是'P'，表示在光标位置处删除 n 个字符（控制序列 'ESC [ Pn P'）。
757                                     case 'P':       // CSI Pn P - 删除字符。
758                                             csi_P(currcons,par[0]);
759                                             break;
    // 如果字符 c 是'@'，表示在光标位置处插入 n 个字符（控制序列 'ESC [ Pn @' ）。
760                                     case '@':       // CSI Pn @ - 插入字符。
761                                             csi_at(currcons,par[0]);
762                                             break;
    // 如果字符 c 是'm'，表示改变光标处字符的显示属性，比如加粗、加下划线、闪烁、反显等。
    // 转义序列：'ESC [ Pn m'。n=0 正常显示；1 加粗；4 加下划线；7 反显；27 正常显示等。
763                                     case 'm':       // CSI Ps m - 设置显示字符属性。
764                                             csi_m(currcons);
765                                             break;
    // 如果字符 c 是'r'，则表示用两个参数设置滚屏的起始行号和终止行号。
766                                     case 'r':       // CSI Pn;Pn r - 设置滚屏上下界。
767                                             if (par[0]) par[0]--;
768                                             if (!par[1]) par[1] = video_num_lines;
769                                             if (par[0] < par[1] &&
770                                                 par[1] <= video_num_lines) {
771                                                     top=par[0];
```

```
772                                             bottom=par[1];
773                                     }
774                                     break;
    // 如果字符 c 是's'，则表示保存当前光标所在位置。
775                             case 's':       // CSI s - 保存光标位置。
776                                     save_cur(currcons);
777                                     break;
    // 如果字符 c 是'u'，则表示恢复光标到原保存的位置处。
778                             case 'u':       // CSI u - 恢复保存的光标位置。
779                                     restore_cur(currcons);
780                                     break;
    // 如果字符 c 是'l'或'b'，则分别表示设置屏幕黑屏间隔时间和设置粗体字符显示。此时参数数
    // 组中 par[1]和 par[2]是特征值，它们分别必须为 par[1]= par[0]+13；par[2]= par[0]+17。
    // 在这个条件下，如果 c 是字符'l'，那么 par[0]中是开始黑屏时说延迟的分钟数；如果 c 是
    // 字符'b'，那么 par[0]中是设置的粗体字符属性值。
781                             case 'l': /* blank interval */
782                             case 'b': /* bold attribute */
783                                       if (!((npar >= 2) &&
784                                       ((par[1]-13) == par[0]) &&
785                                       ((par[2]-17) == par[0])))
786                                         break;
787                                     if ((c=='l')&&(par[0]>=0)&&(par[0]<=60))
788                                     {
789                                       blankinterval = HZ*60*par[0];
790                                       blankcount = blankinterval;
791                                     }
792                                     if (c=='b')
793                                       vc_cons[currcons].vc_bold_attr
794                                         = par[0];
795                             }
796                             break;
    // 状态 ESfunckey 表示我们接收到了键盘上功能键发出的一个序列。我们不用显示。于是恢复到
    // 正常状态 ESnormal。
797                     case ESfunckey:             // 键盘功能键码。
798                             state = ESnormal;
799                             break;
    // 状态 ESsetterm 表示处于设备控制字符串序列状态（DCS）。此时若收到字符 'S'，则恢复初
    // 始的显示字符属性。若收到的字符是'L'或'l'，则开启或关闭折行显示方式。
800                     case ESsetterm:  /* Setterm functions. */
801                             state = ESnormal;
802                             if (c == 'S') {
803                                     def_attr = attr;
804                                     video_erase_char = (video_erase_char&0x0ff) |
                                                        (def_attr<<8);
805                             } else if (c == 'L')
806                                     ; /*linewrap on*/
807                             else if (c == 'l')
808                                     ; /*linewrap off*/
809                             break;
    // 状态 ESsetgraph 表示收到设置字符集转移序列'ESC (' 或 'ESC )'。它们分别用于指定 G0 和
    // G1 所用的字符集。此时若收到字符'0'，则选择图形字符集作为 G0 和 G1，若收到的字符是'B'，
    // 则选择普通 ASCII 字符集作为 G0 和 G1 的字符集。
810                     case ESsetgraph:        // 'CSI ( 0'或'CSI ( B' - 选择字符集。
```

```
811                                 state = ESnormal;
812                                 if (c == '0')
813                                         translate = GRAF_TRANS;
814                                 else if (c == 'B')
815                                         translate = NORM_TRANS;
816                                 break;
817                         default:
818                                 state = ESnormal;
819                 }
820         }
821         set_cursor(currcons);  // 最后根据上面设置的光标位置，设置显示控制器中光标位置。
822 }
823
824 /*
825  *  void con_init(void);
826  *
827  * This routine initalizes console interrupts, and does nothing
828  * else. If you want the screen to clear, call tty_write with
829  * the appropriate escape-sequece.
830  *
831  * Reads the information preserved by setup.s to determine the current display
832  * type and sets everything accordingly.
833  */
    /*
     * void con_init(void);
     *
     * 这个子程序初始化控制台中断，其他什么都不做。如果你想让屏幕干净的话，就使用
     * 适当的转义字符序列调用 tty_write()函数。
     *
     * 读取 setup.s 程序保存的信息，用以确定当前显示器类型，并且设置所有相关参数。
     */
834 void con_init(void)
835 {
836         register unsigned char a;
837         char *display_desc = "????";
838         char *display_ptr;
839         int currcons = 0;                       // 当前虚拟控制台号。
840         long base, term;
841         long video_memory;
842
    // 首先根据 setup.s 程序取得的系统硬件参数（见本程序第 60--68 行）初始化几个本函数专用
    // 的静态全局变量。
843         video_num_columns = ORIG_VIDEO_COLS;    // 显示器显示字符列数。
844         video_size_row = video_num_columns * 2; // 每行字符需使用的字节数。
845         video_num_lines = ORIG_VIDEO_LINES;     // 显示器显示字符行数。
846         video_page = ORIG_VIDEO_PAGE;           // 当前显示页面。
847         video_erase_char = 0x0720;              // 擦除字符（0x20 是字符，0x07 属性）。
848         blankcount = blankinterval;             // 默认的黑屏间隔时间（嘀嗒数）。
849
    // 然后根据显示模式是单色还是彩色分别设置所使用的显示内存起始位置以及显示寄存器索引
    // 端口号和显示寄存器数据端口号。如果获得的 BIOS 显示方式等于 7，则表示是单色显示卡。
850         if (ORIG_VIDEO_MODE == 7)               /* Is this a monochrome display? */
851         {
```

```
852                 video_mem_base = 0xb0000;       // 设置单显映像内存起始地址。
853                 video_port_reg = 0x3b4;         // 设置单显索引寄存器端口。
854                 video_port_val = 0x3b5;         // 设置单显数据寄存器端口。

    // 接着我们根据 BIOS 中断 int 0x10 功能 0x12 获得的显示模式信息，判断显示卡是单色显示卡
    // 还是彩色显示卡。若使用上述中断功能所得到的 BX 寄存器返回值不等于 0x10，则说明是 EGA
    // 卡。因此初始显示类型为 EGA 单色。虽然 EGA 卡上有较多显示内存，但在单色方式下最多只
    // 能利用地址范围在 0xb0000--0xb8000 之间的显示内存。然后置显示器描述字符串为'EGAm'。
    // 并会在系统初始化期间显示器描述字符串将显示在屏幕的右上角。
    // 注意，这里使用了 bx 在调用中断 int 0x10 前后是否被改变的方法来判断卡的类型。若 BL 在
    // 中断调用后值被改变，表示显示卡支持 Ah=12h 功能调用，是 EGA 或后推出来的 VGA 等类型的
    // 显示卡。若中断调用返回值未变，表示显示卡不支持这个功能，则说明是一般单色显示卡。
855                 if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
856                 {
857                         video_type = VIDEO_TYPE_EGAM;  // 设置显示类型（EGA 单色）。
858                         video_mem_term = 0xb8000;      // 设置显示内存末端地址。
859                         display_desc = "EGAm";         // 设置显示描述字符串。
860                 }
    // 如果 BX 寄存器的值等于 0x10，则说明是单色显示卡 MDA，仅有 8KB 显示内存。
861                 else
862                 {
863                         video_type = VIDEO_TYPE_MDA;   // 设置显示类型(MDA 单色)。
864                         video_mem_term = 0xb2000;      // 设置显示内存末端地址。
865                         display_desc = "*MDA";         // 设置显示描述字符串。
866                 }
867         }
    // 如果显示方式不为 7，说明是彩色显示卡。此时文本方式下所用显示内存起始地址为 0xb8000；
    // 显示控制索引寄存器端口地址为 0x3d4；数据寄存器端口地址为 0x3d5。
868         else                            /* If not, it is color. */
869         {
870                 can_do_colour = 1;                     // 设置彩色显示标志。
871                 video_mem_base = 0xb8000;              // 显示内存起始地址。
872                 video_port_reg  = 0x3d4;               // 设置彩色显示索引寄存器端口。
873                 video_port_val  = 0x3d5;               // 设置彩色显示数据寄存器端口。
    // 再判断显示卡类别。如果 BX 不等于 0x10，则说明是 EGA 显示卡，此时共有 32KB 显示内存可用
    // （0xb8000-0xc0000）。否则说明是 CGA 显示卡，只能使用 8KB 显示内存（0xb8000-0xba000）。
874                 if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
875                 {
876                         video_type = VIDEO_TYPE_EGAC;  // 设置显示类型（EGA 彩色）。
877                         video_mem_term = 0xc0000;      // 设置显示内存末端地址。
878                         display_desc = "EGAc";         // 设置显示描述字符串。
879                 }
880                 else
881                 {
882                         video_type = VIDEO_TYPE_CGA;   // 设置显示类型（CGA）。
883                         video_mem_term = 0xba000;      // 设置显示内存末端地址。
884                         display_desc = "*CGA";         // 设置显示描述字符串。
885                 }
886         }
    // 现在我们来计算当前显示卡内存上可以开设的虚拟控制台数量。硬件允许开设的虚拟控制台数
    // 量等于总显示内存量 video_memory 除以每个虚拟控制台占用的字节数。每个虚拟控制台占用的
    // 显示内存数等于屏幕显示行数 video_num_lines 乘上每行字符占有的字节数 video_size_row。
    // 如果硬件允许开设的虚拟控制台数量大于系统限定的最大数量 MAX_CONSOLES，就把虚拟控制台
```

```
    // 数量设置为 MAX_CONSOLES。若这样计算出的虚拟控制台数量为 0，则设置为 1（不可能吧！）。
    // 最后总显示内存数除以判断出的虚拟控制台数即得到每个虚拟控制台占用显示内存字节数。
887         video_memory = video_mem_term - video_mem_base;
888         NR_CONSOLES = video_memory / (video_num_lines * video_size_row);
889         if (NR_CONSOLES > MAX_CONSOLES)             // MAX_CONSOLES = 8。
890                 NR_CONSOLES = MAX_CONSOLES;
891         if (!NR_CONSOLES)
892                 NR_CONSOLES = 1;
893         video_memory /= NR_CONSOLES;                // 每个虚拟控制台占用显示内存字节数。
894
895         /* Let the user known what kind of display driver we are using */
            /* 初始化用于滚屏的变量（主要用于 EGA/VGA） */
896
    // 然后我们在屏幕的右上角显示描述字符串。采用的方法是直接将字符串写到显示内存的相应
    // 位置处。首先将显示指针 display_ptr 指到屏幕第 1 行右端差 4 个字符处（每个字符需 2 个
    // 字节，因此减 8），然后循环复制字符串的字符，并且每复制 1 个字符都空开 1 个属性字节。
897         display_ptr = ((char *)video_mem_base) + video_size_row - 8;
898         while (*display_desc)
899         {
900                 *display_ptr++ = *display_desc++;
901                 display_ptr++;
902         }
903
904         /* Initialize the variables used for scrolling (mostly EGA/VGA) */
            /* 初始化用于滚屏的变量(主要用于 EGA/VGA) */
905
    // 注意，此时当前虚拟控制台号 currcons 已被初始化位 0。因此下面实际上是初始化 0 号虚拟控
    // 制台的结构 vc_cons[0]中的所有字段值。例如，这里符号 origin 在前面第 115 行上已被定义为
    // vc_cons[0].vc_origin。下面首先设置 0 号控制台的默认滚屏开始内存位置 video_mem_start
    // 和默认滚屏末行内存位置，实际上它们也就是 0 号控制台占用的部分显示内存区域。然后初始
    // 设置 0 号虚拟控制台的其他属性和标志值。
906         base = origin = video_mem_start = video_mem_base;  // 默认滚屏开始内存位置。
907         term = video_mem_end = base + video_memory;        // 0 号屏幕内存末端位置。
908         scr_end = video_mem_start + video_num_lines * video_size_row; // 滚屏末端位置。
909         top     = 0;                                // 初始设置滚动时顶行行号和底行行号。
910         bottom  = video_num_lines;
911         attr = 0x07;                                // 初始设置显示字符属性（黑底白字）。
912         def_attr = 0x07;                            // 设置默认显示字符属性。
913         restate = state = ESnormal;                 // 初始化转义序列操作的当前和下一状态。
914         checkin = 0;
915         ques = 0;                                   // 收到问号字符标志。
916         iscolor = 0;                                // 彩色显示标志。
917         translate = NORM_TRANS;                     // 使用的字符集（普通 ASCII 码表）。
918         vc_cons[0].vc_bold_attr = -1;               // 粗体字符属性标志（-1 表示不用）。
919
    // 在设置了 0 号控制台当前光标所在位置和光标对应的内存位置 pos 后，我们循环设置其余的几
    // 个虚拟控制台结构的参数值。除了各自占用的显示内存开始和结束位置不同，它们的初始值基
    // 本上都与 0 号控制台相同。
920         gotoxy(currcons,ORIG_X,ORIG_Y);
921         for (currcons = 1; currcons<NR_CONSOLES; currcons++) {
922                 vc_cons[currcons] = vc_cons[0];     // 复制 0 号结构的参数。
923                 origin = video_mem_start = (base += video_memory);
924                 scr_end = origin + video_num_lines * video_size_row;
```

```
925                 video_mem_end = (term += video_memory);
926                 gotoxy(currcons,0,0);                   // 光标都初始化在屏幕左上角位置。
927         }
    // 最后设置当前前台控制台的屏幕原点（左上角）位置和显示控制器中光标显示位置，并设置键
    // 盘中断 0x21 陷阱门描述符（&keyboard_interrupt 是键盘中断处理过程地址）。然后取消中断
    // 控制芯片 8259A 中对键盘中断的屏蔽，允许响应键盘发出的 IRQ1 请求信号。最后复位键盘控
    // 制器以允许键盘开始正常工作。
928         update_screen();                            // 更新前台原点和设置光标位置。
929         set_trap_gate(0x21,&keyboard_interrupt); // 参见 system.h，第 36 行开始。
930         outb_p(inb_p(0x21)&0xfd,0x21);              // 取消对键盘中断的屏蔽，允许 IRQ1。
931         a=inb_p(0x61);                              // 读取键盘端口 0x61（8255A 端口 PB）。
932         outb_p(a|0x80,0x61);                        // 设置禁止键盘工作（位 7 置位），
933         outb_p(a,0x61);                             // 再允许键盘工作，用以复位键盘。
934 }
935
    // 更新当前前台控制台。
    // 把前台控制台转换为 fg_console 指定的虚拟控制台。fg_console 是设置的前台虚拟控制台号。
936 void update_screen(void)
937 {
938         set_origin(fg_console);                  // 设置滚屏起始显示内存地址。
939         set_cursor(fg_console);                  // 设置显示控制器中光标显示内存位置。
940 }
941
942 /* from bsd-net-2: */
943
    //// 停止蜂鸣。
    // 复位 8255A PB 端口的位 1 和位 0。参见 kernel/sched.c 程序后的定时器编程说明。
944 void sysbeepstop(void)
945 {
946         /* disable counter 2 */ /* 禁止定时器 2 */
947         outb(inb_p(0x61)&0xFC, 0x61);
948 }
949
950 int beepcount = 0;              // 蜂鸣时间嘀嗒计数。
951
    // 开通蜂鸣。
    // 8255A 芯片 PB 端口的位 1 用作扬声器的开门信号；位 0 用作 8253 定时器 2 的门信号，该定时
    // 器的输出脉冲送往扬声器，作为扬声器发声的频率。因此要使扬声器蜂鸣，需要两步：首先开
    // 启 PB 端口（0x61）位 1 和 位 0（置位），然后设置定时器 2 通道发送一定的定时频率即可。
    // 参见 boot/setup.s 程序后 8259A 芯片编程方法和 kernel/sched.c 程序后的定时器编程说明。
952 static void sysbeep(void)
953 {
954         /* enable counter 2 */          /* 开启定时器 2 */
955         outb_p(inb_p(0x61)|3, 0x61);
956         /* set command for counter 2, 2 byte write */    /* 送设置定时器 2 命令 */
957         outb_p(0xB6, 0x43);             // 定时器芯片控制字寄存器端口。
958         /* send 0x637 for 750 HZ */     /* 设置频率为 750HZ，因此送定时值 0x637 */
959         outb_p(0x37, 0x42);             // 通道 2 数据端口分别送计数高低字节。
960         outb(0x06, 0x42);
961         /* 1/8 second */                /* 蜂鸣时间为 1/8 秒 */
962         beepcount = HZ/8;
963 }
964
```

```
    //// 拷贝屏幕。
    // 把屏幕内容复制到参数指定的用户缓冲区 arg 中。
    // 参数 arg 有两个用途，一是用于传递控制台号，二是作为用户缓冲区指针。
965 int do_screendump(int arg)
966 {
967         char *sptr, *buf = (char *)arg;
968         int currcons, l;
969
    // 函数首先验证用户提供的缓冲区容量，若不够则进行适当扩展。然后从其开始处取出控制台
    // 号 currcons。在判断控制台号有效之后，就把该控制台屏幕的所有内存内容复制到用户缓冲
    // 区中。
970         verify_area(buf,video_num_columns*video_num_lines);
971         currcons = get_fs_byte(buf);
972         if ((currcons<1) || (currcons>NR_CONSOLES))
973                 return -EIO;
974         currcons--;
975         sptr = (char *) origin;
976         for (l=video_num_lines*video_num_columns; l>0 ; l--)
977                 put_fs_byte(*sptr++,buf++);
978         return(0);
979 }
980
    // 黑屏处理。
    // 当用户在 blankInterval 时间间隔内没有按任何按键时就让屏幕黑屏，以保护屏幕。
981 void blank_screen()
982 {
983         if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
984                 return;
985 /* blank here. I can't find out how to do it, though */
986 }
987
    // 恢复黑屏的屏幕。
    // 当用户按下任何按键时，就恢复处于黑屏状态的屏幕显示内容。
988 void unblank_screen()
989 {
990         if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
991                 return;
992 /* unblank here */
993 }
994
    //// 控制台显示函数。
    // 该函数仅用于内核显示函数 printk()（kernel/printk.c），用于在当前前台控制台上显示
    // 内核信息。处理方法是循环取出缓冲区中的字符，并根据字符的特性控制光标移动或直接显
    // 示在屏幕上。
    // 参数 b 是 null 结尾的字符串缓冲区指针。
995 void console_print(const char * b)
996 {
997         int currcons = fg_console;
998         char c;
999
    // 循环读取缓冲区 b 中的字符。如果当前字符 c 是换行符，则对光标执行回车换行操作；然后
    // 去处理下一个字符。如果是回车符，就直接执行回车动作。然后去处理下一个字符。
1000         while (c = *(b++)) {
```

```
1001                    if (c == 10) {
1002                            cr(currcons);
1003                            lf(currcons);
1004                            continue;
1005                    }
1006                    if (c == 13) {
1007                            cr(currcons);
1008                            continue;
1009                    }
        // 在读取了一个不是回车或换行字符后，如果发现当前光标列位置 x 已经到达屏幕右末端，则让
        // 光标折返到下一行开始处。然后把字符放到光标所处显示内存位置处，即在屏幕上显示出来。
        // 再把光标右移一格位置，为显示下一个字符作准备。
1010                    if (x>=video_num_columns) {
1011                            x -= video_num_columns;
1012                            pos -= video_size_row;
1013                            lf(currcons);
1014                    }
        // 寄存器 al 中是需要显示的字符，这里把属性字节放到 ah 中，然后把 ax 内容存储到光标内存
        // 位置 pos 处，即在光标处显示字符。
1015                    __asm__("movb %2,%%ah\n\t"              // 属性字节放到 ah 中。
1016                            "movw %%ax,%1\n\t"              // ax 内容放到 pos 处。
1017                            ::"a" (c),
1018                            "m" (*(short *)pos),
1019                            "m" (attr)
1020                            :"ax");
1021                    pos += 2;
1022                    x++;
1023            }
1024            set_cursor(currcons); // 最后设置的光标内存位置，设置显示控制器中光标位置。
1025 }
1026
```

10.3 程序 10-3 linux/kernel/chr_drv/serial.c

```
/*
 *  linux/kernel/serial.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      serial.c
 *
 * This module implements the rs232 io functions
 *      void rs_write(struct tty_struct * queue);
 *      void rs_init(void);
 * and all interrupts pertaining to serial IO.
 */
/*
 *     serial.c
 * 该程序用于实现 rs232 的输入输出函数
 *     void rs_write(struct tty_struct *queue);
 *     void rs_init(void);
 * 以及与串行 IO 有关系的所有中断处理程序。
 */

#include <linux/tty.h>    // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
#include <asm/system.h>   // 系统头文件。定义设置或修改描述符/中断门等的嵌入式汇编宏。
#include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。

#define WAKEUP_CHARS (TTY_BUF_SIZE/4)  // 当写队列中含有 WAKEUP_CHARS 个字符时就开始发送。

extern void rs1_interrupt(void);       // 串行口 1 的中断处理程序（rs_io.s, 34 行）。
extern void rs2_interrupt(void);       // 串行口 2 的中断处理程序（rs_io.s, 38 行）。

//// 初始化串行端口
// 设置指定串行端口的传输波特率（2400bps）并允许除了写保持寄存器空以外的所有中断源。
// 另外，在输出 2 字节的波特率因子时，须首先设置线路控制寄存器的 DLAB 位（位 7）。
// 参数：port 是串行端口基地址，串口 1 - 0x3F8；串口 2 - 0x2F8。
static void init(int port)
{
        outb_p(0x80,port+3);    /* set DLAB of line control reg */
        outb_p(0x30,port);      /* LS of divisor (48 -> 2400 bps */
        outb_p(0x00,port+1);    /* MS of divisor */
        outb_p(0x03,port+3);    /* reset DLAB */
        outb_p(0x0b,port+4);    /* set DTR,RTS, OUT_2 */
        outb_p(0x0d,port+1);    /* enable all intrs but writes */
        (void)inb(port);        /* read data port to reset things (?) */
}

//// 初始化串行中断程序和串行接口。
// 中断描述符表 IDT 中的门描述符设置宏 set_intr_gate()在 include/asm/system.h 中实现。
```

```
37 void rs_init(void)
38 {
   // 下面两句用于设置两个串行口的中断门描述符。rs1_interrupt 是串口 1 的中断处理过程指针。
   // 串口 1 使用的中断是 int 0x24，串口 2 的是 int 0x23。参见表 2-2 和 system.h 文件。
39         set_intr_gate(0x24,rs1_interrupt);    // 设置串行口 1 的中断门向量(IRQ4 信号)。
40         set_intr_gate(0x23,rs2_interrupt);    // 设置串行口 2 的中断门向量(IRQ3 信号)。
41         init(tty_table[64].read_q->data);     // 初始化串行口 1(.data 是端口基地址)。
42         init(tty_table[65].read_q->data);     // 初始化串行口 2。
43         outb(inb_p(0x21)&0xE7,0x21);          // 允许主 8259A 响应 IRQ3、IRQ4 中断请求。
44 }
45
46 /*
47  * This routine gets called when tty_write has put something into
48  * the write_queue. It must check wheter the queue is empty, and
49  * set the interrupt register accordingly
50  *
51  *      void _rs_write(struct tty_struct * tty);
52  */
   /*
    * 在 tty_write()已将数据放入输出(写)队列时会调用下面的子程序。在该
    * 子程序中必须首先检查写队列是否为空，然后设置相应中断寄存器。
    */
   //// 串行数据发送输出。
   // 该函数实际上只是开启发送保持寄存器已空中断标志。此后当发送保持寄存器空时，UART 就会
   // 产生中断请求。而在该串行中断处理过程中，程序会取出写队列尾指针处的字符，并输出到发
   // 送保持寄存器中。一旦 UART 把该字符发送了出去，发送保持寄存器又会变空而引发中断请求。
   // 于是只要写队列中还有字符，系统就会重复这个处理过程，把字符一个一个地发送出去。当写
   // 队列中所有字符都发送了出去，写队列变空了，中断处理程序就会把中断允许寄存器中的发送
   // 保持寄存器中断允许标志复位掉，从而再次禁止发送保持寄存器空引发中断请求。此次“循环”
   // 发送操作也随之结束。
53 void rs_write(struct tty_struct * tty)
54 {
   // 如果写队列不空，则首先从 0x3f9（或 0x2f9）读取中断允许寄存器内容，添上发送保持寄存器
   // 中断允许标志（位 1）后，再写回该寄存器。这样，当发送保持寄存器空时 UART 就能够因期望
   // 获得欲发送的字符而引发中断。write_q.data 中是串行端口基地址。
55         cli();
56         if (!EMPTY(tty->write_q))
57                 outb(inb_p(tty->write_q->data+1)|0x02,tty->write_q->data+1);
58         sti();
59 }
60
```

10.4 程序 10-4 linux/kernel/chr_drv/rs_io.s

```
/*
 *  linux/kernel/rs_io.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      rs_io.s
 *
 * This module implements the rs232 io interrupts.
 */
/*
 * 该模块实现 rs232 输入输出中断处理程序。
 */

.text
.globl _rs1_interrupt,_rs2_interrupt

// size 是读写队列缓冲区的字节长度。该值必须是 2 的次方，并且必须与 tty_io.c 中的匹配。
size    = 1024                          /* must be power of two !
                                           and must match the value
                                           in tty_io.c!!! */

/* these are the offsets into the read/write buffer structures */
/* 以下这些是读写缓冲队列结构中的偏移量 */
// 对应 include/linux/tty.h 文件中 tty_queue 结构中各字段的字节偏移量。其中 rs_addr
// 对应 tty_queue 结构的 data 字段。对于串行终端缓冲队列，该字段存放着串行端口基地址。
rs_addr = 0                     // 串行端口号字段偏移（端口是 0x3f8 或 0x2f8）。
head = 4                        // 缓冲区中头指针字段偏移。
tail = 8                        // 缓冲区中尾指针字段偏移。
proc_list = 12                  // 等待该缓冲的进程字段偏移。
buf = 16                        // 缓冲区字段偏移。

// 当一个写缓冲队列满后，内核就会把要往写队列填字符的进程设置为等待状态。当写缓冲队列
// 中还剩余最多 256 个字符时，中断处理程序就可以唤醒这些等待进程继续往写队列中放字符。
startup = 256                   /* chars left in write queue when we restart it */
                                /* 当我们重新开始写时，队列里最多还剩余字符个数。*/

/*
 * These are the actual interrupt routines. They look where
 * the interrupt is coming from, and take appropriate action.
 */
/*
 * 这些是实际的中断处理程序。程序首先检查中断的来源，然后执行
 * 相应的处理。
 */
//// 串行端口 1 中断处理程序入口点。
// 初始化时 rs1_interrupt 地址被放入中断描述符 0x24 中，对应 8259A 的中断请求 IRQ4 引脚。
// 这里首先把 tty 表中串行终端 1（串口 1）读写缓冲队列指针的地址入栈（tty_io.c，81），
```

```
   // 然后跳转到 rs_int 继续处理。这样做可以让串口 1 和串口 2 的处理代码公用。字符缓冲队列
   // 结构 tty_queue 格式请参见 include/linux/tty.h，第 22 行。
33 .align 2
34 _rs1_interrupt:
35         pushl $_table_list+8    // tty 表中串口 1 读写缓冲队列指针地址入栈。
36         jmp rs_int
37 .align 2
   //// 串行端口 2 中断处理程序入口点。
38 _rs2_interrupt:
39         pushl $_table_list+16   // tty 表中串口 2 读写缓冲队列指针地址入栈。

   // 这段代码首先让段寄存器 ds、es 指向内核数据段，然后从对应读写缓冲队列 data 字段取出
   // 串行端口基地址。该地址加 2 即是中断标识寄存器 IIR 的端口地址。若位 0 = 0，表示有需
   // 要处理的中断。于是根据位 2、位 1 使用指针跳转表调用相应中断源类型处理子程序。在每
   // 个子程序中会在处理完后复位 UART 的相应中断源。在子程序返回后这段代码会循环判断是
   // 否还有其他中断源（位 0 = 0？）。如果本次中断还有其他中断源，则 IIR 的位 0 仍然是 0。
   // 于是中断处理程序会再调用相应中断源子程序继续处理。直到引起本次中断的所有中断源都
   // 被处理并复位，此时 UART 会自动地设置 IIR 的位 0 =1，表示已无待处理的中断，于是中断
   // 处理程序即可退出。
40 rs_int:
41         pushl %edx
42         pushl %ecx
43         pushl %ebx
44         pushl %eax
45         push %es
46         push %ds                /* as this is an interrupt, we cannot */
47         pushl $0x10             /* know that bs is ok. Load it */
48         pop %ds                 /* 由于这是一个中断程序，我们不知道 ds 是否正确，*/
49         pushl $0x10             /* 所以加载它们（让 ds、es 指向内核数据段） */
50         pop %es
51         movl 24(%esp),%edx      // 取上面 35 或 39 行入栈的相应串口缓冲队列指针地址。
52         movl (%edx),%edx        // 取读缓冲队列结构指针（地址）➔edx。
53         movl rs_addr(%edx),%edx // 取串口 1（或串口 2）端口基地址➔edx。
54         addl $2,%edx            /* interrupt ident. reg */  /* 指向中断标识寄存器 */
                                   // 中断标识寄存器端口地址是 0x3fa（0x2fa）。
55 rep_int:
56         xorl %eax,%eax
57         inb %dx,%al             // 取中断标识字节，以判断中断来源（有 4 种中断情况）。
58         testb $1,%al            // 首先判断有无待处理中断（位 0 = 0 有中断）。
59         jne end                 // 若无待处理中断，则跳转至退出处理处 end。
60         cmpb $6,%al             /* this shouldn't happen, but ... */ /*这不会发生，但…*/
61         ja end                  // al 值大于 6，则跳转至 end（没有这种状态）。
62         movl 24(%esp),%ecx      // 调用子程序之前把缓冲队列指针地址放入 ecx。
63         pushl %edx              // 临时保存中断标识寄存器端口地址。
64         subl $2,%edx            // edx 中恢复串口基地址值 0x3f8（0x2f8）。
65         call jmp_table(,%eax,2) /* NOTE! not *4, bit0 is 0 already */
   // 上面语句是指，当有待处理中断时，al 中位 0=0，位 2、位 1 是中断类型，因此相当于已经将
   // 中断类型乘了 2，这里再乘 2，获得跳转表（第 79 行）对应各中断类型地址，并跳转到那里去
   // 作相应处理。中断来源有 4 种：modem 状态发生变化；要写（发送）字符；要读（接收）字符；
   // 线路状态发生变化。允许发送字符中断通过设置发送保持寄存器标志实现。在 serial.c 程序
   // 中，当写缓冲队列中有数据时，rs_write()函数就会修改中断允许寄存器内容，添加上发送保
   // 持寄存器中断允许标志，从而在系统需要发送字符时引起串行中断发生。
66         popl %edx               // 恢复中断标识寄存器端口地址 0x3fa（或 0x2fa）。
```

```
67          jmp rep_int             // 跳转，继续判断有无待处理中断并作相应处理。

68  end:    movb $0x20,%al          // 中断退出处理。向中断控制器发送结束中断指令 EOI。
69          outb %al,$0x20          /* EOI */
70          pop %ds
71          pop %es
72          popl %eax
73          popl %ebx
74          popl %ecx
75          popl %edx
76          addl $4,%esp            # jump over _table_list entry   # 丢弃队列指针地址。
77          iret
78
    // 各中断类型处理子程序地址跳转表，共有 4 种中断来源：
    // modem 状态变化中断，写字符中断，读字符中断，线路状态有问题中断。
79  jmp_table:
80          .long modem_status,write_char,read_char,line_status
81
    // 由于 modem 状态发生变化而引发此次中断。通过读 modem 状态寄存器 MSR 对其进行复位操作。
82  .align 2
83  modem_status:
84          addl $6,%edx            /* clear intr by reading modem status reg */
85          inb %dx,%al             /* 通过读 modem 状态寄存器进行复位（0x3fe） */
86          ret
87
    // 由于线路状态发生变化而引起这次串行中断。通过读线路状态寄存器 LSR 对其进行复位操作。
88  .align 2
89  line_status:
90          addl $5,%edx            /* clear intr by reading line status reg. */
91          inb %dx,%al             /* 通过读线路状态寄存器进行复位（0x3fd） */
92          ret
93
    // 由于 UART 芯片接收到字符而引起这次中断。对接收缓冲寄存器执行读操作可复位该中断源。
    // 这个子程序将接收到的字符放到读缓冲队列 read_q 头指针（head）处，并且让该指针前移一
    // 个字符位置。若 head 指针已经到达缓冲区末端，则让其折返到缓冲区开始处。最后调用 C 函
    // 数 do_tty_interrupt()（也即 copy_to_cooked()），把读入的字符经过处理放入规范模式缓
    // 冲队列（辅助缓冲队列 secondary）中。
94  .align 2
95  read_char:
96          inb %dx,%al             // 读取接收缓冲寄存器 RBR 中字符➔al。
97          movl %ecx,%edx          // 当前串口缓冲队列指针地址➔edx。
98          subl $_table_list,%edx  // 当前串口队列指针地址 - 缓冲队列指针表首址 ➔edx，
99          shrl $3,%edx            // 差值/8，得串口号。对于串口 1 是 1，对于串口 2 是 2。
100         movl (%ecx),%ecx         # read-queue     // 取读缓冲队列结构地址➔ecx。
101         movl head(%ecx),%ebx    // 取读队列中缓冲头指针➔ebx。
102         movb %al,buf(%ecx,%ebx) // 将字符放在缓冲区中头指针所指位置处。
103         incl %ebx               // 将头指针前移（右移）一字节。
104         andl $size-1,%ebx       // 用缓冲区长度对头指针取模操作。
105         cmpl tail(%ecx),%ebx    // 缓冲区头指针与尾指针比较。
106         je 1f                   // 若指针移动后相等，表示缓冲区满，不保存头指针，跳转。
107         movl %ebx,head(%ecx)    // 保存修改过的头指针。
108 1:      addl $63,%edx           // 串口号转换成 tty 号（63 或 64）并作为参数入栈。
109         pushl %edx
```

```
110         call _do_tty_interrupt  // 调用 tty 中断处理 C 函数（tty_io.c，342 行）。
111         addl $4,%esp            // 丢弃入栈参数，并返回。
112         ret
113
    // 由于设置了发送保持寄存器允许中断标志而引起此次中断。说明对应串行终端的写字符缓冲队
    // 列中有字符需要发送。于是计算出写队列中当前所含字符数，若字符数已小于 256 个，则唤醒
    // 等待写操作进程。然后从写缓冲队列尾部取出一个字符发送，并调整和保存尾指针。如果写缓
    // 冲队列已空，则跳转到 write_buffer_empty 处处理写缓冲队列空的情况。
114 .align 2
115 write_char:
116         movl 4(%ecx),%ecx           # write-queue  // 取写缓冲队列结构地址➔ecx。
117         movl head(%ecx),%ebx        // 取写队列头指针➔ebx。
118         subl tail(%ecx),%ebx        // 头指针 - 尾指针 = 队列中字符数。
119         andl $size-1,%ebx           # nr chars in queue
120         je write_buffer_empty       // 若头指针 = 尾指针，说明写队列空，跳转处理。
121         cmpl $startup,%ebx          // 队列中字符数还超过 256 个？
122         ja 1f                       // 超过则跳转处理。
123         movl proc_list(%ecx),%ebx   # wake up sleeping process  # 唤醒等待的进程。
                                        // 取等待该队列的进程指针，并判断是否为空。
124         testl %ebx,%ebx             # is there any?    # 有等待写的进程吗？
125         je 1f                       // 是空的，则向前跳转到标号 1 处。
126         movl $0,(%ebx)              // 否则将进程置为可运行状态（唤醒进程）。
127 1:      movl tail(%ecx),%ebx        // 取尾指针。
128         movb buf(%ecx,%ebx),%al     // 从缓冲中尾指针处取一字符➔al。
129         outb %al,%dx                // 向端口 0x3f8（0x2f8）写到发送保持寄存器中。
130         incl %ebx                   // 尾指针前移。
131         andl $size-1,%ebx           // 尾指针若到缓冲区末端，则折回。
132         movl %ebx,tail(%ecx)        // 保存已修改过的尾指针。
133         cmpl head(%ecx),%ebx        // 尾指针与头指针比较，
134         je write_buffer_empty       // 若相等，表示队列已空，则跳转。
135         ret
    // 处理写缓冲队列 write_q 已空的情况。若有等待写该串行终端的进程则唤醒之，然后屏蔽发
    // 送保持寄存器空中断，不让发送保持寄存器空时产生中断。
    // 如果此时写缓冲队列 write_q 已空，表示当前无字符需要发送。于是我们应该做两件事情。
    // 首先看看有没有进程正等待写队列空出来，如果有就唤醒之。另外，因为现在系统已无字符
    // 需要发送，所以此时我们要暂时禁止发送保持寄存器 THR 空时产生中断。当再有字符被放入
    // 写缓冲队列中时，serial.c 中的 rs_write()函数会再次允许发送保持寄存器空时产生中断，
    // 因此 UART 就又会“自动”地来取写缓冲队列中的字符，并发送出去。
136 .align 2
137 write_buffer_empty:
138         movl proc_list(%ecx),%ebx   # wake up sleeping process # 唤醒等待的进程。
                                        // 取等待该队列的进程的指针，并判断是否为空。
139         testl %ebx,%ebx             # is there any?    # 有等待的进程吗？
140         je 1f                       // 无，则向前跳转到标号 1 处。
141         movl $0,(%ebx)              // 否则将进程置为可运行状态（唤醒进程）。
142 1:      incl %edx                   // 指向端口 0x3f9（0x2f9）。
143         inb %dx,%al                 // 读取中断允许寄存器 IER。
144         jmp 1f                      // 稍作延迟。
145 1:      jmp 1f                      /* 屏蔽发送保持寄存器空中断（位 1） */
146 1:      andb $0xd,%al               /* disable transmit interrupt */
147         outb %al,%dx                // 写入 0x3f9(0x2f9)。
148         ret
```

10.5 程序 10-5 linux/kernel/chr_drv/tty_io.c

```
 1 /*
 2  *  linux/kernel/tty_io.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * 'tty_io.c' gives an orthogonal feeling to tty's, be they consoles
 9  * or rs-channels. It also implements echoing, cooked mode etc.
10  *
11  * Kill-line thanks to John T Kohl, who also corrected VMIN = VTIME = 0.
12  */
   /*
   * 'tty_io.c'给 tty 终端一种非相关的感觉，不管它们是控制台还是串行终端。
   * 该程序同样实现了回显、规范(熟)模式等。
   *
   * Kill-line 问题，要谢谢 John T Kohl。他同时还纠正了当 VMIN = VTIME = 0 时的问题。
   */
13
14 #include <ctype.h>          // 字符类型头文件。定义了一些有关字符类型判断和转换的宏。
15 #include <errno.h>          // 错误号头文件。包含系统中各种出错号。
16 #include <signal.h>         // 信号头文件。定义信号符号常量，信号结构及其操作函数原型。
17 #include <unistd.h>         // unistd.h 是标准符号常数与类型文件，并声明了各种函数。
18
   // 给出定时警告（alarm）信号在信号位图中对应的比特屏蔽位。
19 #define ALRMMASK (1<<(SIGALRM-1))
20
21 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
22 #include <linux/tty.h>    // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
23 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
24 #include <asm/system.h>   // 系统头文件。定义设置或修改描述符/中断门等嵌入式汇编宏。
25
   // 终止进程组（向进程组发送信号）。参数 pgrp 指定进程组号；sig 指定信号；priv 权限。
   // 即向指定进程组 pgrp 中的每个进程发送指定信号 sig。只要向一个进程发送成功最后就会
   // 返回 0，否则如果没有找到指定进程组号 pgrp 的任何一个进程，则返回出错号-ESRCH，若
   // 找到进程组号是 pgrp 的进程，但是发送信号失败，则返回发送失败的错误码。
26 int kill_pg(int pgrp, int sig, int priv);               // kernel/exit.c，171 行。
   // 判断一个进程组是否是孤儿进程。如果不是则返回 0；如果是则返回 1。
27 int is_orphaned_pgrp(int pgrp);                        // kernel/exit.c，232 行。
28
   // 获取 termios 结构中三个模式标志集之一，或者用于判断一个标志集是否有置位标志。
29 #define _L_FLAG(tty,f)  ((tty)->termios.c_lflag & f)    // 本地模式标志。
30 #define _I_FLAG(tty,f)  ((tty)->termios.c_iflag & f)    // 输入模式标志。
31 #define _O_FLAG(tty,f)  ((tty)->termios.c_oflag & f)    // 输出模式标志。
32
   // 取 termios 结构终端特殊（本地）模式标志集中的一个标志。
33 #define L_CANON(tty)    _L_FLAG((tty),ICANON)    // 取规范模式标志。
34 #define L_ISIG(tty)     _L_FLAG((tty),ISIG)      // 取信号标志。
35 #define L_ECHO(tty)     _L_FLAG((tty),ECHO)      // 取回显字符标志。
```

```
36 #define L_ECHOE(tty)     _L_FLAG((tty),ECHOE)     // 规范模式时取回显擦出标志。
37 #define L_ECHOK(tty)     _L_FLAG((tty),ECHOK)     // 规范模式时取 KILL 擦除当前行标志。
38 #define L_ECHOCTL(tty)   _L_FLAG((tty),ECHOCTL)   // 取回显控制字符标志。
39 #define L_ECHOKE(tty)    _L_FLAG((tty),ECHOKE)    // 规范模式时取 KILL 擦除行并回显标志。
40 #define L_TOSTOP(tty)    _L_FLAG((tty),TOSTOP)    // 对于后台输出发送 SIGTTOU 信号。
41
   // 取 termios 结构输入模式标志集中的一个标志。
42 #define I_UCLC(tty)      _I_FLAG((tty),IUCLC)     // 取大写到小写转换标志。
43 #define I_NLCR(tty)      _I_FLAG((tty),INLCR)     // 取换行符 NL 转回车符 CR 标志。
44 #define I_CRNL(tty)      _I_FLAG((tty),ICRNL)     // 取回车符 CR 转换行符 NL 标志。
45 #define I_NOCR(tty)      _I_FLAG((tty),IGNCR)     // 取忽略回车符 CR 标志。
46 #define I_IXON(tty)      _I_FLAG((tty),IXON)      // 取输入控制流标志 XON。
47
   // 取 termios 结构输出模式标志集中的一个标志。
48 #define O_POST(tty)      _O_FLAG((tty),OPOST)     // 取执行输出处理标志。
49 #define O_NLCR(tty)      _O_FLAG((tty),ONLCR)     // 取换行符 NL 转回车换行符 CR-NL 标志。
50 #define O_CRNL(tty)      _O_FLAG((tty),OCRNL)     // 取回车符 CR 转换行符 NL 标志。
51 #define O_NLRET(tty)     _O_FLAG((tty),ONLRET)    // 取换行符 NL 执行回车功能的标志。
52 #define O_LCUC(tty)      _O_FLAG((tty),OLCUC)     // 取小写转大写字符标志。
53
   // 取 termios 结构控制标志集中波特率。CBAUD 是波特率屏蔽码（0000017）。
54 #define C_SPEED(tty)     ((tty)->termios.c_cflag & CBAUD)
   // 判断 tty 终端是否已挂线（hang up），即其传输波特率是否为 B0（0）。
55 #define C_HUP(tty)       (C_SPEED((tty)) == B0)
56
   // 取最小值宏。
57 #ifndef MIN
58 #define MIN(a,b) ((a) < (b) ? (a) : (b))
59 #endif
60
   // 下面定义 tty 终端使用的缓冲队列结构数组 tty_queues 和 tty 终端表结构数组 tty_table。
   // QUEUES 是 tty 终端使用的缓冲队列最大数量。伪终端分主从两种（master 和 slave）。每个
   // tty 终端使用 3 个 tty 缓冲队列，它们分别是用于缓冲键盘或串行输入的读队列 read_queue、
   // 用于缓冲屏幕或串行输出的写队列 write_queue，以及用于保存规范模式字符的辅助缓冲队列
   // secondary。
61 #define QUEUES  (3*(MAX_CONSOLES+NR_SERIALS+2*NR_PTYS))  // 共 54 项。
62 static struct tty_queue tty_queues[QUEUES];              // tty 缓冲队列数组。
63 struct tty_struct tty_table[256];                        // tty 表结构数组。
64
   // 下面设定各种类型的 tty 终端所使用缓冲队列结构在 tty_queues[]数组中的起始项位置。
   // 8 个虚拟控制台终端占用 tty_queues[]数组开头 24 项（3 X MAX_CONSOLES）（0 -- 23）；
   // 两个串行终端占用随后的 6 项（3 X NR_SERIALS）（24 -- 29）。
   // 4 个主伪终端占用随后的 12 项（3 X NR_PTYS）（30 -- 41）。
   // 4 个从伪终端占用随后的 12 项（3 X NR_PTYS）（42 -- 53）。
65 #define con_queues tty_queues
66 #define rs_queues ((3*MAX_CONSOLES) + tty_queues)
67 #define mpty_queues ((3*(MAX_CONSOLES+NR_SERIALS)) + tty_queues)
68 #define spty_queues ((3*(MAX_CONSOLES+NR_SERIALS+NR_PTYS)) + tty_queues)
69
   // 下面设定各种类型 tty 终端所使用的 tty 结构在 tty_table[]数组中的起始项位置。
   // 8 个虚拟控制台终端可用 tty_table[]数组开头 64 项（0 -- 63）；
   // 两个串行终端使用随后的 2 项（64 -- 65）。
   // 4 个主伪终端使用从 128 开始的项，最多 64 项（128 -- 191）。
```

```
   // 4 个从伪终端使用从 192 开始的项，最多 64 项（192 -- 255）。
70 #define con_table tty_table                 // 定义控制台终端 tty 表符号常数。
71 #define rs_table (64+tty_table)             // 串行终端 tty 表。
72 #define mpty_table (128+tty_table)          // 主伪终端 tty 表。
73 #define spty_table (192+tty_table)          // 从伪终端 tty 表。
74
75 int fg_console = 0;                 // 当前前台控制台号（范围 0--7）。
76
77 /*
78  * these are the tables used by the machine code handlers.
79  * you can implement virtual consoles.
80  */
   /*
    * 下面是汇编程序中使用的缓冲队列结构地址表。通过修改这个表，
    * 你可以实现虚拟控制台。
    */
   // tty 读写缓冲队列结构地址表。供 rs_io.s 程序使用，用于取得读写缓冲队列结构的地址。
81 struct tty_queue * table_list[]={
82         con_queues + 0, con_queues + 1,       // 前台控制台读、写队列结构地址。
83         rs_queues + 0, rs_queues + 1,         // 串行终端 1 读、写队列结构地址。
84         rs_queues + 3, rs_queues + 4          // 串行终端 2 读、写队列结构地址。
85         };
86
   //// 改变前台控制台。
   // 将前台控制台设定为指定的虚拟控制台。
   // 参数：new_console - 指定的新控制台号。
87 void change_console(unsigned int new_console)
88 {
   // 如果参数指定的控制台已经在前台或者参数无效，则退出。否则设置当前前台控制台号，同
   // 时更新 table_list[]中的前台控制台读、写队列结构地址。最后更新当前前台控制台屏幕。
89         if (new_console == fg_console || new_console >= NR_CONSOLES)
90                 return;
91         fg_console = new_console;
92         table_list[0] = con_queues + 0 + fg_console*3;
93         table_list[1] = con_queues + 1 + fg_console*3;
94         update_screen();                    // kernel/chr_drv/console.c，936 行。
95 }
96
   //// 如果队列缓冲区空则让进程进入可中断睡眠状态。
   // 参数：queue - 指定队列的指针。
   // 进程在取队列缓冲区中字符之前需要调用此函数加以验证。如果当前进程没有信号要处理，
   // 并且指定的队列缓冲区空，则让进程进入可中断睡眠状态，并让队列的进程等待指针指向
   // 该进程。
97 static void sleep_if_empty(struct tty_queue * queue)
98 {
99         cli();
100        while (!(current->signal & ~current->blocked) && EMPTY(queue))
101                interruptible_sleep_on(&queue->proc_list);
102        sti();
103 }
104
   //// 若队列缓冲区满则让进程进入可中断的睡眠状态。
   // 参数：queue - 指定队列的指针。
```

```
    // 进程在往队列缓冲区中写入字符之前需要调用此函数判断队列情况。
105 static void sleep_if_full(struct tty_queue * queue)
106 {
    // 如果队列缓冲区不满则返回退出。否则若进程没有信号需要处理，并且队列缓冲区中空闲剩
    // 余区长度 < 128，则让进程进入可中断睡眠状态，并让该队列的进程等待指针指向该进程。
107         if (!FULL(queue))
108                 return;
109         cli();
110         while (!(current->signal & ~current->blocked) && LEFT(queue)<128)
111                 interruptible_sleep_on(&queue->proc_list);
112         sti();
113 }
114
    //// 等待按键。
    // 如果前台控制台读队列缓冲区空，则让进程进入可中断睡眠状态。
115 void wait_for_keypress(void)
116 {
117         sleep_if_empty(tty_table[fg_console].secondary);
118 }
119
    //// 复制成规范模式字符序列。
    // 根据终端 termios 结构中设置的各种标志，将指定 tty 终端读队列缓冲区中的字符复制转换
    // 成规范模式（熟模式）字符并存放在辅助队列（规范模式队列）中。
    // 参数：tty - 指定终端的 tty 结构指针。
120 void copy_to_cooked(struct tty_struct * tty)
121 {
122         signed char c;
123
    // 首先检查当前终端 tty 结构中缓冲队列指针是否有效。如果三个队列指针都是 NULL，则说明
    // 内核 tty 初始化函数有问题。
124         if (!(tty->read_q || tty->write_q || tty->secondary)) {
125                 printk("copy_to_cooked: missing queues\n\r");
126                 return;
127         }
    // 否则我们根据终端 termios 结构中的输入和本地标志，对从 tty 读队列缓冲区中取出的每个
    // 字符进行适当的处理，然后放入辅助队列 secondary 中。在下面循环体中，如果此时读队列
    // 缓冲区已经取空或者辅助队列缓冲区已经放满字符，就退出循环体。否则程序就从读队列缓
    // 冲区尾指针处取一字符，并把尾指针前移一个字符位置。然后根据该字符代码值进行处理。
    // 另外，如果定义了_POSIX_VDISABLE（\0），那么在对字符处理过程忠，若字符代码值等于
    // _POSIX_VDISABLE 的值时，表示禁止使用相应特殊控制字符的功能。
128         while (1) {
129                 if (EMPTY(tty->read_q))
130                         break;
131                 if (FULL(tty->secondary))
132                         break;
133                 GETCH(tty->read_q,c);           // 取一字符到 c，并前移尾指针。
    // 如果该字符是回车符 CR（13），那么若回车转换行标志 CRNL 置位，则将字符转换为换行符
    // NL（10）。否则如果忽略回车标志 NOCR 置位，则忽略该字符，继续处理其他字符。如果字
    // 符是换行符 NL（10），并且换行转回车标志 NLCR 置位，则将其转换为回车符 CR（13）。
134                 if (c==13) {
135                         if (I_CRNL(tty))
136                                 c=10;
137                         else if (I_NOCR(tty))
```

```
138                             continue;
139                 } else if (c==10 && I_NLCR(tty))
140                         c=13;
    // 如果大写转小写输入标志 UCLC 置位，则将该字符转换为小写字符。
141                 if (I_UCLC(tty))
142                         c=tolower(c);
    // 如果本地模式标志集中规范模式标志 CANON 已置位，则对读取的字符进行以下处理。 首先，
    // 如果该字符是键盘终止控制字符 KILL(^U)，则对已输入的当前行执行删除处理。删除一行字
    // 符的循环过程如是：如果 tty 辅助队列不空，并且取出的辅助队列中最后一个字符不是换行
    // 符 NL（10），并且该字符不是文件结束字符（^D），则循环执行下列代码：
    // 如果本地回显标志 ECHO 置位，那么：若字符是控制字符（值 < 32），则往 tty 写队列中放
    // 入擦除控制字符 ERASE（^H）。然后再放入一个擦除字符 ERASE，并且调用该 tty 写函数，把
    // 写队列中的所有字符输出到终端屏幕上。 另外，因为控制字符在放入写队列时需要用 2 个字
    // 节表示（例如^V），因此要求特别对控制字符多放入一个 ERASE。最后将 tty 辅助队列头指针
    // 后退 1 字节。另外，如果定义了_POSIX_VDISABLE（\0），那么在对字符处理过程忠，若字符
    // 代码值等于 _POSIX_VDISABLE 的值时，表示禁止使用相应特殊控制字符的功能。
143                 if (L_CANON(tty)) {
144                         if ((KILL_CHAR(tty) != _POSIX_VDISABLE) &&
145                             (c==KILL_CHAR(tty))) {
146                                 /* deal with killing the input line */
147                                 while(!(EMPTY(tty->secondary) ||
148                                         (c=LAST(tty->secondary))==10 ||
149                                         ((EOF_CHAR(tty) != _POSIX_VDISABLE) &&
150                                          (c==EOF_CHAR(tty))))) {
151                                         if (L_ECHO(tty)) {    // 若本地回显标志置位。
152                                                 if (c<32)     // 控制字符要删 2 字节。
153                                                         PUTCH(127,tty->write_q);
154                                                 PUTCH(127,tty->write_q);
155                                                 tty->write(tty);
156                                         }
157                                         DEC(tty->secondary->head);
158                                 }
159                                 continue;          // 继续读取读队列中字符进行处理。
160                         }
    // 如果该字符是删除控制字符 ERASE（^H），那么：如果 tty 的辅助队列为空，或者其最后一个
    // 字符是换行符 NL(10)，或者是文件结束符，则继续处理其他字符。如果本地回显标志 ECHO 置
    // 位，那么：若字符是控制字符（值< 32），则往 tty 的写队列中放入擦除字符 ERASE。再放入
    // 一个擦除字符 ERASE，并且调用该 tty 的写函数。最后将 tty 辅助队列头指针后退 1 字节，继
    // 续处理其他字符。同样地，如果定义了_POSIX_VDISABLE（\0），那么在对字符处理过程忠，
    // 若字符代码值等于 _POSIX_VDISABLE 的值时，表示禁止使用相应特殊控制字符的功能。
161                         if ((ERASE_CHAR(tty) != _POSIX_VDISABLE) &&
162                             (c==ERASE_CHAR(tty))) {
163                                 if (EMPTY(tty->secondary) ||
164                                    (c=LAST(tty->secondary))==10 ||
165                                    ((EOF_CHAR(tty) != _POSIX_VDISABLE) &&
166                                     (c==EOF_CHAR(tty))))
167                                         continue;
168                                 if (L_ECHO(tty)) {          // 若本地回显标志置位。
169                                         if (c<32)
170                                                 PUTCH(127,tty->write_q);
171                                         PUTCH(127,tty->write_q);
172                                         tty->write(tty);
173                                 }
```

```
174                             DEC(tty->secondary->head);
175                             continue;
176                         }
177                 }
    // 如果设置了 IXON 标志，则使终端停止/开始输出控制字符起作用。如果没有设置此标志，那
    // 么停止和开始字符将被作为一般字符供进程读取。在这段代码中，如果读取的字符是停止字
    // 符 STOP（^S），则置 tty 停止标志，让 tty 暂停输出。同时丢弃该特殊控制字符（不放入
    // 辅助队列中），并继续处理其他字符。如果字符是开始字符 START（^Q），则复位 tty 停止
    // 标志，恢复 tty 输出。同时丢弃该控制字符，并继续处理其他字符。
    // 对于控制台来说，这里的 tty->write()是 console.c 中的 con_write()函数。因此控制台将
    // 由于发现 stopped=1 而会立刻暂停在屏幕上显示新字符（chr_drv/console.c，第 586 行）。
    // 对于伪终端也是由于设置了终端 stopped 标志而会暂停写操作（chr_drv/pty.c，第 24 行）。
    // 对于串行终端，也应该在发送终端过程中根据终端 stopped 标志暂停发送，但本版未实现。
178                 if (I_IXON(tty)) {
179                         if ((STOP_CHAR(tty) != _POSIX_VDISABLE) &&
180                             (c==STOP_CHAR(tty))) {
181                                 tty->stopped=1;
182                                 tty->write(tty);
183                                 continue;
184                         }
185                         if ((START_CHAR(tty) != _POSIX_VDISABLE) &&
186                             (c==START_CHAR(tty))) {
187                                 tty->stopped=0;
188                                 tty->write(tty);
189                                 continue;
190                         }
191                 }
    // 若输入模式标志集中 ISIG 标志置位，表示终端键盘可以产生信号，则在收到控制字符 INTR、
    // QUIT、SUSP 或 DSUSP 时，需要为进程产生相应的信号。 如果该字符是键盘中断符（^C），
    // 则向当前进程之进程组中所有进程发送键盘中断信号 SIGINT，并继续处理下一字符。 如果该
    // 字符是退出符（^\），则向当前进程之进程组中所有进程发送键盘退出信号 SIGQUIT，并继续
    // 处理下一字符。如果字符是暂停符（^Z），则向当前进程发送暂停信号 SIGTSTP。同样，若定
    // 义了_POSIX_VDISABLE（\0），那么在对字符处理过程忠，若字符代码值等于_POSIX_VDISABLE
    // 的值时，表示禁止使用相应特殊控制字符的功能。以下不再啰嗦了 :-)
192                 if (L_ISIG(tty)) {
193                         if ((INTR_CHAR(tty) != _POSIX_VDISABLE) &&
194                             (c==INTR_CHAR(tty))) {
195                                 kill_pg(tty->pgrp, SIGINT, 1);
196                                 continue;
197                         }
198                         if ((QUIT_CHAR(tty) != _POSIX_VDISABLE) &&
199                             (c==QUIT_CHAR(tty))) {
200                                 kill_pg(tty->pgrp, SIGQUIT, 1);
201                                 continue;
202                         }
203                         if ((SUSPEND_CHAR(tty) != _POSIX_VDISABLE) &&
204                             (c==SUSPEND_CHAR(tty))) {
205                                 if (!is_orphaned_pgrp(tty->pgrp))
206                                         kill_pg(tty->pgrp, SIGTSTP, 1);
207                                 continue;
208                         }
209                 }
    // 如果该字符是换行符 NL（10），或者是文件结束符 EOF（4，^D），表示一行字符已处理完，
```

```
    // 则把辅助缓冲队列中当前含有字符行数值 secondary.data 增 1。如果在函数 tty_read()中取
    // 走一行字符，该值即会被减 1，参见 315 行。
210                 if (c==10 || (EOF_CHAR(tty) != _POSIX_VDISABLE &&
211                               c==EOF_CHAR(tty)))
212                         tty->secondary->data++;
    // 如果本地模式标志集中回显标志 ECHO 在置位状态，那么，如果字符是换行符 NL（10），则将
    // 换行符 NL（10）和回车符 CR（13）放入 tty 写队列缓冲区中；如果字符是控制字符（值<32）
    // 并且回显控制字符标志 ECHOCTL 置位，则将字符'^'和字符 c+64 放入 tty 写队列中（也即会
    // 显示^C、^H 等）；否则将该字符直接放入 tty 写缓冲队列中。最后调用该 tty 写操作函数。
213                 if (L_ECHO(tty)) {
214                         if (c==10) {
215                                 PUTCH(10,tty->write_q);
216                                 PUTCH(13,tty->write_q);
217                         } else if (c<32) {
218                                 if (L_ECHOCTL(tty)) {
219                                         PUTCH('^',tty->write_q);
220                                         PUTCH(c+64,tty->write_q);
221                                 }
222                         } else
223                                 PUTCH(c,tty->write_q);
224                         tty->write(tty);
225                 }
    // 每一次循环末将处理过的字符放入辅助队列中。
226                 PUTCH(c,tty->secondary);
227         }
    // 最后在退出循环体后唤醒等待该辅助缓冲队列的进程（如果有的话）。
228         wake_up(&tty->secondary->proc_list);
229 }
230
231 /*
232  * Called when we need to send a SIGTTIN or SIGTTOU to our process
233  * group
234  *
235  * We only request that a system call be restarted if there was if the
236  * default signal handler is being used.  The reason for this is that if
237  * a job is catching SIGTTIN or SIGTTOU, the signal handler may not want
238  * the system call to be restarted blindly.  If there is no way to reset the
239  * terminal pgrp back to the current pgrp (perhaps because the controlling
240  * tty has been released on logout), we don't want to be in an infinite loop
241  * while restarting the system call, and have it always generate a SIGTTIN
242  * or SIGTTOU.  The default signal handler will cause the process to stop
243  * thus avoiding the infinite loop problem.  Presumably the job-control
244  * cognizant parent will fix things up before continuging its child process.
245  */
    /* 当需要发送信号 SIGTTIN 或 SIGTTOU 到我们进程组中所有进程时就会调用该函数。
     *
     * 在进程使用默认信号处理句柄情况下，我们仅要求一个系统调用被重新启动，如果
     * 有系统调用因本信号而被中断。这样做的原因是，如果一个作业正在捕获 SIGTTIN
     * 或 SIGTTOU 信号，那么相应信号句柄并不会希望系统调用被盲目地重新启动。如果
     * 没有其他方法把终端的 pgrp 复位到当前 pgrp（例如可能由于在 logout 时控制终端
     * 已被释放），那么我们并不希望在重新启动系统调用时掉入一个无限循环中，并且
     * 总是产生 SIGTTIN 或 SIGTTOU 信号。默认的信号句柄会使得进程停止，因而可以
     * 避免无限循环问题。这里假设可识别作业控制的父进程会在继续执行其子进程之前
```

```
     * 把问题搞定。
     */
    //// 向使用终端的进程组中所有进程发送信号。
    // 在后台进程组中的一个进程访问控制终端时，该函数用于向后台进程组中的所有进程发送
    // SIGTTIN 或 SIGTTOU 信号。无论后台进程组中的进程是否已经阻塞或忽略掉了这两个信号，
    // 当前进程都将立刻退出读写操作而返回。
246 int tty_signal(int sig, struct tty_struct *tty)
247 {
    // 我们不希望停止一个孤儿进程组中的进程（参见文件 kernel/exit.c 中第 232 行上的说明）。
    // 因此如果当前进程组是孤儿进程组，就出错返回。否则就向当前进程组所有进程发送指定信
    // 号 sig。
248         if (is_orphaned_pgrp(current->pgrp))
249                 return -EIO;            /* don't stop an orphaned pgrp */
250         (void) kill_pg(current->pgrp,sig,1);    // 发送信号 sig。
    // 如果这个信号被当前进程阻塞（屏蔽），或者被当前进程忽略掉，则出错返回。否则，如果
    // 当前进程对信号 sig 设置了新的处理句柄，那么就返回我们可被中断的信息。否则就返回在
    // 系统调用重新启动后可以继续执行的信息。
251         if ((current->blocked & (1<<(sig-1))) ||
252             ((int) current->sigaction[sig-1].sa_handler == 1))
253                 return -EIO;            /* Our signal will be ignored */
254         else if (current->sigaction[sig-1].sa_handler)
255                 return -EINTR;          /* We _will_ be interrupted :-) */
256         else
257                 return -ERESTARTSYS;    /* We _will_ be interrupted :-) */
258                                         /* (but restart after we continue) */
259 }
260
    //// tty 读函数。
    // 从终端辅助缓冲队列中读取指定数量的字符，放到用户指定的缓冲区中。
    // 参数：channel - 子设备号；buf - 用户缓冲区指针；nr - 欲读字节数。
    // 返回已读字节数。
261 int tty_read(unsigned channel, char * buf, int nr)
262 {
263         struct tty_struct * tty;
264         struct tty_struct * other_tty = NULL;
265         char c, * b=buf;
266         int minimum,time;
267
    // 首先判断参数有效性并取终端的 tty 结构指针。如果 tty 终端的三个缓冲队列指针都是 NULL，
    // 则返回 EIO 出错信息。如果 tty 终端是一个伪终端，则再取得另一个对应伪终端的 tty 结构
    // other_tty。
268         if (channel > 255)
269                 return -EIO;
270         tty = TTY_TABLE(channel);
271         if (!(tty->write_q || tty->read_q || tty->secondary))
272                 return -EIO;
    // 如果当前进程使用的是这里正在处理的 tty 终端，但该终端的进程组号却与当前进程组号不
    // 同，表示当前进程是后台进程组中的一个进程，即进程不在前台。于是我们要停止当前进程
    // 组的所有进程。因此这里就需要向当前进程组发送 SIGTTIN 信号，并返回等待成为前台进程
    // 组后再执行读操作。
273         if ((current->tty == channel) && (tty->pgrp != current->pgrp))
274                 return(tty_signal(SIGTTIN, tty));
    // 如果当前终端是伪终端，那么对应的另一个伪终端就是 other_tty。若这里 tty 是主伪终端，
```

```
    // 那么 other_tty 就是对应的从伪终端，反之也然。
275         if (channel & 0x80)
276                 other_tty = tty_table + (channel ^ 0x40);

    // 然后根据 VTIME 和 VMIN 对应的控制字符数组值设置读字符操作超时定时值 time 和最少需
    // 要读取的字符个数 minimum。在非规范模式下，这两个是超时定时值。VMIN 表示为了满足读
    // 操作而需要读取的最少字符个数。VTIME 是一个 1/10 秒计数计时值。
277         time = 10L*tty->termios.c_cc[VTIME];        // 设置读操作超时定时值。
278         minimum = tty->termios.c_cc[VMIN];          // 最少需要读取的字符个数。
    // 如果 tty 终端处于规范模式，则设置最小要读取字符数 minimum 等于进程欲读字符数 nr。同
    // 时把进程读取 nr 字符的超时时间值设置为极大值（不会超时）。否则说明终端处于非规范模
    // 式下，若此时设置了最少读取字符数 minimum，则先临时设置进城读超时定时值为无限大，以
    // 让进程先读取辅助队列中已有字符。如果读到的字符数不足 minimum 的话，后面代码会根据
    // 指定的超时值 time 来设置进程的读超时值 timeout，并会等待读取其余字符。参见 328 行。
    // 若此时没有设置最少读取字符数 minimum（为 0），则将其设置为进程欲读字符数 nr，并且如
    // 果设置了超时定时值 time 的话，就把进程读字符超时定时值 timeout 设置为系统当前时间值
    //  + 指定的超时值 time，同时复位 time。 另外，如果以上设置的最少读取字符数 minimum 大
    // 于进程欲读取的字符数 nr，则让 minimum=nr。即对于规范模式下的读取操作，它不受 VTIME
    // 和 VMIN 对应控制字符值的约束和控制，它们仅在非规范模式（生模式）操作中起作用。
279         if (L_CANON(tty)) {
280                 minimum = nr;
281                 current->timeout = 0xffffffff;
282                 time = 0;
283         } else if (minimum)
284                 current->timeout = 0xffffffff;
285         else {
286                 minimum = nr;
287                 if (time)
288                         current->timeout = time + jiffies;
289                 time = 0;
290         }
291         if (minimum>nr)
292                 minimum = nr;                   // 最多读取要求的字符数。

    // 现在我们开始从辅助队列中循环取出字符并放到用户缓冲区 buf 中。当欲读的字节数大于 0，
    // 则执行以下循环操作。在循环过程中，如果当前终端是伪终端，那么我们就执行其对应的另
    // 一个伪终端的写操作函数，让另一个伪终端把字符写入当前伪终端辅助队列缓冲区中。即让
    // 另一终端把写队列缓冲区中字符复制到当前伪终端读队列缓冲区中，并经行规程函数转换后
    // 放入当前伪终端辅助队列中。
293         while (nr>0) {
294                 if (other_tty)
295                         other_tty->write(other_tty);
    // 如果 tty 辅助缓冲队列为空，或者设置了规范模式标志并且 tty 读队列缓冲区未满，并且辅
    // 助队列中字符行数为 0，那么，如果没有设置过进程读字符超时值（为 0），或者当前进程
    // 目前收到信号，就先退出循环体。否则如果本终端是一个从伪终端，并且其对应的主伪终端
    // 已经挂断，那么我们也退出循环体。如果不是以上这两种情况，我们就让当前进程进入可中
    // 断睡眠状态，返回后继续处理。由于规范模式时内核以行为单位为用户提供数据，因此在该
    // 模式下辅助队列中必须起码有一行字符可供取用，即 secondary.data 起码是 1 才行。
296                 cli();
297                 if (EMPTY(tty->secondary) || (L_CANON(tty) &&
298                     !FULL(tty->read_q) && !tty->secondary->data)) {
299                         if (!current->timeout ||
300                           (current->signal & ~current->blocked)) {
```

```
301                             sti();
302                             break;
303                     }
304                     if (IS_A_PTY_SLAVE(channel) && C_HUP(other_tty))
305                             break;
306                     interruptible_sleep_on(&tty->secondary->proc_list);
307                     sti();
308                     continue;
309             }
310             sti();
    // 下面开始正式执行取字符操作。需读字符数 nr 依次递减，直到 nr=0 或者辅助缓冲队列为空。
    // 在这个循环过程中，首先取辅助缓冲队列字符 c，并且把缓冲队列尾指针 tail 向右移动一个
    // 字符位置。如果所取字符是文件结束符（^D）或者是换行符 NL（10），则把辅助缓冲队列中
    // 含有字符行数值减 1。 如果该字符是文件结束符（^D）并且规范模式标志成置位状态，则中
    // 断本循环，否则说明现在还没有遇到文件结束符或者正处于原始（非规范）模式。在这种模
    // 式中用户以字符流作为读取对象，也不识别其中的控制字符（如文件结束符）。于是将字符
    // 直接放入用户数据缓冲区 buf 中，并把欲读字符数减 1。此时如果欲读字符数已为 0 则中断
    // 循环。另外，如果终端处于规范模式并且读取的字符是换行符 NL（10），则也退出循环。
    // 除此之外，只要还没有取完欲读字符数 nr 并且辅助队列不空，就继续取队列中的字符。
311             do {
312                     GETCH(tty->secondary,c);
313                     if ((EOF_CHAR(tty) != _POSIX_VDISABLE &&
314                          c==EOF_CHAR(tty)) || c==10)
315                             tty->secondary->data--;
316                     if ((EOF_CHAR(tty) != _POSIX_VDISABLE &&
317                          c==EOF_CHAR(tty)) && L_CANON(tty))
318                             break;
319                     else {
320                             put_fs_byte(c,b++);
321                             if (!--nr)
322                                     break;
323                     }
324                     if (c==10 && L_CANON(tty))
325                             break;
326             } while (nr>0 && !EMPTY(tty->secondary));
    // 执行到此，那么如果 tty 终端处于规范模式下，说明我们可能读到了换行符或者遇到了文件
    // 结束符。如果是处于非规范模式下，那么说明我们已经读取了 nr 个字符，或者辅助队列已经
    // 被取空了。于是我们首先唤醒等待读队列的进程，然后看看是否设置过超时定时值 time。如
    // 果超时定时值 time 不为 0，我们就要求等待一定的时间让其他进程可以把字符写入读队列中。
    // 于是设置进程读超时定时值为系统当前时间 jiffies + 读超时值 time。当然，如果终端处于
    // 规范模式，或者已经读取了 nr 个字符，我们就可以直接退出这个大循环了。
327             wake_up(&tty->read_q->proc_list);
328             if (time)
329                     current->timeout = time+jiffies;
330             if (L_CANON(tty) || b-buf >= minimum)
331                     break;
332     }
    // 此时读取 tty 字符循环操作结束，因此复位进程的读取超时定时值 timeout。如果此时当前进
    // 程已收到信号并且还没有读取到任何字符，则以重新启动系统调用号返回。否则就返回已读取
    // 的字符数(b-buf)。
333     current->timeout = 0;
334     if ((current->signal & ~current->blocked) && !(b-buf))
335             return -ERESTARTSYS;
```

```
336         return (b-buf);
337 }
338
    //// tty 写函数。
    // 把用户缓冲区中的字符放入 tty 写队列缓冲区中。
    // 参数：channel - 子设备号；buf - 缓冲区指针；nr - 写字节数。
    // 返回已写字节数。
339 int tty_write(unsigned channel, char * buf, int nr)
340 {
341         static cr_flag=0;
342         struct tty_struct * tty;
343         char c, *b=buf;
344
    // 首先判断参数有效性并取终端的 tty 结构指针。如果 tty 终端的三个缓冲队列指针都是 NULL，
    // 则返回 EIO 出错信息。
345         if (channel > 255)
346                 return -EIO;
347         tty = TTY_TABLE(channel);
348         if (!(tty->write_q || tty->read_q || tty->secondary))
349                 return -EIO;
    // 如果若终端本地模式标志集中设置了 TOSTOP，表示后台进程输出时需要发送信号 SIGTTOU。
    // 如果当前进程使用的是这里正在处理的 tty 终端，但该终端的进程组号却与当前进程组号不
    // 同，即表示当前进程是后台进程组中的一个进程，即进程不在前台。于是我们要停止当前进
    // 程组的所有进程。因此这里就需要向当前进程组发送 SIGTTOU 信号，并返回等待成为前台进
    // 程组后再执行写操作。
350         if (L_TOSTOP(tty) &&
351             (current->tty == channel) && (tty->pgrp != current->pgrp))
352                 return(tty_signal(SIGTTOU, tty));
    // 现在我们开始从用户缓冲区 buf 中循环取出字符并放到写队列缓冲区中。当欲写字节数大于 0，
    // 则执行以下循环操作。在循环过程中，如果此时 tty 写队列已满，则当前进程进入可中断的睡
    // 眠状态。如果当前进程有信号要处理，则退出循环体。
353         while (nr>0) {
354                 sleep_if_full(tty->write_q);
355                 if (current->signal & ~current->blocked)
356                         break;
    // 当要写的字符数 nr 还大于 0 并且 tty 写队列缓冲区不满，则循环执行以下操作。首先从用户
    // 缓冲区中取 1 字节。如果终端输出模式标志集中的执行输出处理标志 OPOST 置位，则执行对
    // 字符的后处理操作。
357                 while (nr>0 && !FULL(tty->write_q)) {
358                         c=get_fs_byte(b);
359                         if (O_POST(tty)) {
    // 如果该字符是回车符'\r'（CR，13）并且回车符转换行符标志 OCRNL 置位，则将该字符换成
    // 换行符'\n'（NL，10）；否则如果该字符是换行符'\n'（NL，10）并且换行转回车功能标志
    // ONLRET 置位的话，则将该字符换成回车符'\r'（CR，13）。
360                                 if (c=='\r' && O_CRNL(tty))
361                                         c='\n';
362                                 else if (c=='\n' && O_NLRET(tty))
363                                         c='\r';
    // 如果该字符是换行符'\n' 并且回车标志 cr_flag 没有置位，但换行转回车-换行标志 ONLCR
    // 置位的话，则将 cr_flag 标志置位，并将一回车符放入写队列中。然后继续处理下一个字符。
    // 如果小写转大写标志 OLCUC 置位的话，就将该字符转成大写字符。
364                                 if (c=='\n' && !cr_flag && O_NLCR(tty)) {
365                                         cr_flag = 1;
```

```
366                                     PUTCH(13,tty->write_q);
367                                     continue;
368                             }
369                             if (O_LCUC(tty))                // 小写转成大写字符。
370                                     c=toupper(c);
371                     }
    // 接着把用户数据缓冲指针 b 前移 1 字节；欲写字节数减 1 字节；复位 cr_flag 标志，并将该
    // 字节放入 tty 写队列中。
372                     b++; nr--;
373                     cr_flag = 0;
374                     PUTCH(c,tty->write_q);
375             }
    // 若要求的字符全部写完，或者写队列已满，则程序退出循环。此时会调用对应 tty 写函数，
    // 把写队列缓冲区中的字符显示在控制台屏幕上，或者通过串行端口发送出去。如果当前处
    // 理的 tty 是控制台终端，那么 tty->write()调用的是 con_write()；如果 tty 是串行终端，
    // 则 tty->write()调用的是 rs_write()函数。若还有字节要写，则等待写队列中字符取走。
    // 所以这里调用调度程序，先去执行其他任务。
376             tty->write(tty);
377             if (nr>0)
378                     schedule();
379         }
380         return (b-buf);                 // 最后返回写入的字节数。
381 }
382
383 /*
384  * Jeh, sometimes I really like the 386.
385  * This routine is called from an interrupt,
386  * and there should be absolutely no problem
387  * with sleeping even in an interrupt (I hope).
388  * Of course, if somebody proves me wrong, I'll
389  * hate intel for all time :-). We'll have to
390  * be careful and see to reinstating the interrupt
391  * chips before calling this, though.
392  *
393  * I don't think we sleep here under normal circumstances
394  * anyway, which is good, as the task sleeping might be
395  * totally innocent.
396  */
    /*
     * 呵，有时我真得很喜欢 386。该子程序被从一个中断处理程序中
     * 调用，并且即使在中断处理程序中睡眠也应该绝对没有问题（我
     * 希望如此）。当然，如果有人证明我是错的，那么我将憎恨 intel
     * 一辈子☺。但是我们必须小心，在调用该子程序之前需要恢复中断。
     *
     * 我不认为在通常环境下会处在这里睡眠，这样很好，因为任务睡眠
     * 是完全任意的。
     */
    //// tty 中断处理调用函数 - 字符规范模式处理。
    // 参数：tty - 指定的 tty 终端号。
    // 将指定 tty 终端队列缓冲区中的字符复制或转换成规范(熟)模式字符并存放在辅助队列中。
    // 该函数会在串口读字符中断（rs_io.s，109）和键盘中断（kerboard.S，69）中被调用。
397 void do_tty_interrupt(int tty)
398 {
```

```
399         copy_to_cooked(TTY_TABLE(tty));
400 }
401
    //// 字符设备初始化函数。空，为以后扩展做准备。
402 void chr_dev_init(void)
403 {
404 }
405
    //// tty 终端初始化函数。
    // 初始化所有终端缓冲队列，初始化串口终端和控制台终端。
406 void tty_init(void)
407 {
408         int i;
409
    // 首先初始化所有终端的缓冲队列结构，设置初值。对于串行终端的读/写缓冲队列，将它们的
    // data 字段设置为串行端口基地址值。串口 1 是 0x3f8，串口 2 是 0x2f8。然后先初步设置所有
    // 终端的 tty 结构。其中特殊字符数组 c_cc[]设置的初值定义在 include/linux/tty.h 文件中。
410         for (i=0 ; i < QUEUES ; i++)
411                 tty_queues[i] = (struct tty_queue) {0,0,0,0,""};
412         rs_queues[0] = (struct tty_queue) {0x3f8,0,0,0,""};
413         rs_queues[1] = (struct tty_queue) {0x3f8,0,0,0,""};
414         rs_queues[3] = (struct tty_queue) {0x2f8,0,0,0,""};
415         rs_queues[4] = (struct tty_queue) {0x2f8,0,0,0,""};
416         for (i=0 ; i<256 ; i++) {
417                 tty_table[i] =  (struct tty_struct) {
418                         {0, 0, 0, 0, 0, INIT_C_CC},
419                         0, 0, 0, NULL, NULL, NULL, NULL
420                 };
421         }
    // 接着初始化控制台终端（console.c，834 行）。把 con_init()放在这里，是因为我们需要根
    // 据显示卡类型和显示内存容量来确定系统中虚拟控制台的数量 NR_CONSOLES。该值被用于随后
    // 的控制台 tty 结构初始化循环中。对于控制台的 tty 结构，425--430 行是 tty 结构中包含的
    // termios 结构字段。其中输入模式标志集被初始化为 ICRNL 标志；输出模式标志被初始化为含
    // 有后处理标志 OPOST 和把 NL 转换成 CRNL 的标志 ONLCR；本地模式标志集被初始化含有 IXON、
    // ICANON、ECHO、ECHOCTL 和 ECHOKE 标志；控制字符数组 c_cc[]被设置含有初始值 INIT_C_CC。
    // 435 行上初始化控制台终端 tty 结构中的读缓冲、写缓冲和辅助缓冲队列结构，它们分别指向
    // tty 缓冲队列结构数组 tty_table[]中的相应结构项。参见 61--73 行上的相关说明。
422         con_init();
423         for (i = 0 ; i<NR_CONSOLES ; i++) {
424                 con_table[i] = (struct tty_struct) {
425                         {ICRNL,         /* change incoming CR to NL */   /* CR 转 NL */
426                         OPOST|ONLCR,    /* change outgoing NL to CRNL */ /*NL 转 CRNL*/
427                         0,                                               // 控制模式标志集。
428                         IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, // 本地标志集。
429                         0,              /* console termio */             // 线路规程，0 -- TTY。
430                         INIT_C_CC},                                      // 控制字符数组 c_cc[]。
431                         0,              /* initial pgrp */               // 所属初始进程组 pgrp。
432                         0,              /* initial session */            // 初始会话组 session。
433                         0,              /* initial stopped */            // 初始停止标志 stopped。
434                         con_write,                                       // 控制台写函数。
435                         con_queues+0+i*3,con_queues+1+i*3,con_queues+2+i*3
436                 };
437         }
```

```
    // 然后初始化串行终端的 tty 结构各字段。450 行初始化串行终端 tty 结构中的读/写和辅助缓
    // 冲队列结构，它们分别指向 tty 缓冲队列结构数组 tty_table[]中的相应结构项。参见 61--
    // 73 行上的相关说明。
438         for (i = 0 ; i<NR_SERIALS ; i++) {
439                 rs_table[i] = (struct tty_struct) {
440                         {0, /* no translation */  // 输入模式标志集。0，无须转换。
441                         0,  /* no translation */  // 输出模式标志集。0，无须转换。
442                         B2400 | CS8,        // 控制模式标志集。2400bps，8 位数据位。
443                         0,                          // 本地模式标志集。
444                         0,                          // 线路规程，0 -- TTY。
445                         INIT_C_CC},                 // 控制字符数组。
446                         0,                          // 所属初始进程组。
447                         0,                          // 初始会话组。
448                         0,                          // 初始停止标志。
449                         rs_write,                   // 串口终端写函数。
450                         rs_queues+0+i*3,rs_queues+1+i*3,rs_queues+2+i*3  // 三个队列。
451                 };
452         }
    // 然后再初始化伪终端使用的 tty 结构。伪终端是配对使用的，即一个主（master）伪终端配
    // 有一个从（slave）伪终端。因此对它们都要进行初始化设置。在循环中，我们首先初始化
    // 每个主伪终端的 tty 结构，然后再初始化其对应的从伪终端的 tty 结构。
453         for (i = 0 ; i<NR_PTYS ; i++) {
454                 mpty_table[i] = (struct tty_struct) {
455                         {0, /* no translation */  // 输入模式标志集。0，无须转换。
456                         0,  /* no translation */  // 输出模式标志集。0，无须转换。
457                         B9600 | CS8,        // 控制模式标志集。9600bps，8 位数据位。
458                         0,                          // 本地模式标志集。
459                         0,                          // 线路规程，0 -- TTY。
460                         INIT_C_CC},                 // 控制字符数组。
461                         0,                          // 所属初始进程组。
462                         0,                          // 所属初始会话组。
463                         0,                          // 初始停止标志。
464                         mpty_write,                 // 主伪终端写函数。
465                         mpty_queues+0+i*3,mpty_queues+1+i*3,mpty_queues+2+i*3
466                 };
467                 spty_table[i] = (struct tty_struct) {
468                         {0, /* no translation */  // 输入模式标志集。0，无须转换。
469                         0,  /* no translation */  // 输出模式标志集。0，无须转换。
470                         B9600 | CS8,        // 控制模式标志集。9600bps，8 位数据位。
471                         IXON | ISIG | ICANON,       // 本地模式标志集。
472                         0,                          // 线路规程，0 -- TTY。
473                         INIT_C_CC},                 // 控制字符数组。
474                         0,                          // 所属初始进程组。
475                         0,                          // 所属初始会话组。
476                         0,                          // 初始停止标志。
477                         spty_write,                 // 从伪终端写函数。
478                         spty_queues+0+i*3,spty_queues+1+i*3,spty_queues+2+i*3
479                 };
480         }
    // 最后初始化串行中断处理程序和串行接口 1 和 2（serial.c，37 行），并显示系统含有的虚拟
    // 控制台数 NR_CONSOLES 和伪终端数 NR_PTYS。
481         rs_init();
482         printk("%d virtual consoles\n\r",NR_CONSOLES);
```

```
483             printk("%d pty's\n\r",NR_PTYS);
484 }
485
```

10.6 程序 10-6 linux/kernel/chr_drv/tty_ioctl.c

```
 1 /*
 2  *  linux/kernel/chr_drv/tty_ioctl.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
 8 #include <termios.h>      // 终端输入输出函数头文件。主要定义控制异步通信口的终端接口。
 9
10 #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 的数据等。
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <linux/tty.h>    // tty 头文件，定义有关 tty_io、串行通信方面参数、常数。
13
14 #include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
15 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
16 #include <asm/system.h>   // 系统头文件。定义设置或修改描述符/中断门等的嵌入式汇编宏。
17
   // 根据进程组号 pgrp 取得进程组所属的会话号。定义在 kernel/exit.c，161 行。
18 extern int session_of_pgrp(int pgrp);
   // 向使用指定 tty 终端的进程组中所有进程发送信号。定义在 chr_drv/tty_io.c，246 行。
19 extern int tty_signal(int sig, struct tty_struct *tty);
20
   // 这是波特率因子数组（或称为除数数组）。波特率与波特率因子的对应关系参见列表后说明。
   // 例如波特率是 2400bps 时，对应的因子是 48（0x30）；9600bps 的因子是 12（0x1c）。
21 static unsigned short quotient[] = {
22         0, 2304, 1536, 1047, 857,
23         768, 576, 384, 192, 96,
24         64, 48, 24, 12, 6, 3
25 };
26
   //// 修改传输波特率。
   // 参数：tty - 终端对应的 tty 数据结构。
   // 在除数锁存标志 DLAB 置位情况下，通过端口 0x3f8 和 0x3f9 向 UART 分别写入波特率因子低
   // 字节和高字节。写完后再复位 DLAB 位。对于串口 2，这两个端口分别是 0x2f8 和 0x2f9。
27 static void change_speed(struct tty_struct * tty)
28 {
29         unsigned short port,quot;
30
   // 函数首先检查参数 tty 指定的终端是否是串行终端，若不是则退出。对于串口终端的 tty 结
   // 构，其读缓冲队列 data 字段存放着串行端口基址（0x3f8 或 0x2f8），而一般控制台终端的
   // tty 结构的 read_q.data 字段值为 0。然后从终端 termios 结构的控制模式标志集中取得已设
   // 置的波特率索引号，并据此从波特率因子数组 quotient[]中取得对应的波特率因子值 quot。
   // CBAUD 是控制模式标志集中波特率位屏蔽码。
31         if (!(port = tty->read_q->data))
32                 return;
33         quot = quotient[tty->termios.c_cflag & CBAUD];
   // 接着把波特率因子 quot 写入串行端口对应 UART 芯片的波特率因子锁存器中。在写之前我们
   // 先要把线路控制寄存器 LCR 的除数锁存访问比特位 DLAB（位 7）置 1。然后把 16 位的波特
   // 率因子低高字节分别写入端口 0x3f8、0x3f9 （分别对应波特率因子低、高字节锁存器）。
```

```
    // 最后再复位 LCR 的 DLAB 标志位。
34          cli();
35          outb_p(0x80,port+3);        /* set DLAB */   // 首先设置除数锁定标志 DLAB。
36          outb_p(quot & 0xff,port);   /* LS of divisor */   // 输出因子低字节。
37          outb_p(quot >> 8,port+1);   /* MS of divisor */   // 输出因子高字节。
38          outb(0x03,port+3);          /* reset DLAB */      // 复位 DLAB。
39          sti();
40  }
41
    //// 刷新 tty 缓冲队列。
    // 参数：queue - 指定的缓冲队列指针。
    // 令缓冲队列的头指针等于尾指针，从而达到清空缓冲区的目的。
42  static void flush(struct tty_queue * queue)
43  {
44          cli();
45          queue->head = queue->tail;
46          sti();
47  }
48
    //// 等待字符发送出去。
49  static void wait_until_sent(struct tty_struct * tty)
50  {
51          /* do nothing - not implemented */  /* 什么都没做 - 还未实现 */
52  }
53
    //// 发送 BREAK 控制符。
54  static void send_break(struct tty_struct * tty)
55  {
56          /* do nothing - not implemented */  /* 什么都没做 - 还未实现 */
57  }
58
    //// 取终端 termios 结构信息。
    // 参数：tty - 指定终端的 tty 结构指针；termios - 存放 termios 结构的用户缓冲区。
59  static int get_termios(struct tty_struct * tty, struct termios * termios)
60  {
61          int i;
62
    // 首先验证用户缓冲区指针所指内存区容量是否足够，如不够则分配内存。然后复制指定终端
    // 的 termios 结构信息到用户缓冲区中。最后返回 0。
63          verify_area(termios, sizeof (*termios));        // kernel/fork.c，24 行。
64          for (i=0 ; i< (sizeof (*termios)) ; i++)
65                  put_fs_byte( ((char *)&tty->termios)[i] , i+(char *)termios );
66          return 0;
67  }
68
    //// 设置终端 termios 结构信息。
    // 参数：tty - 指定终端的 tty 结构指针；termios - 用户数据区 termios 结构指针。
69  static int set_termios(struct tty_struct * tty, struct termios * termios,
70                         int channel)
71  {
72          int i, retsig;
73
74          /* If we try to set the state of terminal and we're not in the
```

```
75              foreground, send a SIGTTOU.  If the signal is blocked or
76              ignored, go ahead and perform the operation.  POSIX 7.2) */
            /* 如果试图设置终端的状态但此时终端不在前台，那么我们就需要发送
               一个 SIGTTOU 信号。如果该信号被进程屏蔽或者忽略了，就直接执行
               本次操作。  POSIX 7.2  */
    // 如果当前进程使用的 tty 终端的进程组号与进程的进程组号不同，即当前进程终端不在前台，
    // 表示当前进程试图修改不受控制的终端的 termios 结构。因此根据 POSIX 标准的要求这里需
    // 要发送 SIGTTOU 信号让使用这个终端的进程先暂时停止执行，以让我们先修改 termios 结构。
    // 如果发送信号函数 tty_signal()返回值是 ERESTARTSYS 或 EINTR，则等一会再执行本次操作。
77          if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
78                  retsig = tty_signal(SIGTTOU, tty);      // chr_drv/tty_io.c，246 行。
79                  if (retsig == -ERESTARTSYS || retsig == -EINTR)
80                          return retsig;
81          }
    // 接着把用户数据区中 termios 结构信息复制到指定终端 tty 结构的 termios 结构中。因为用
    // 户有可能已修改了终端串行口传输波特率，所以这里再根据 termios 结构中的控制模式标志
    // c_cflag 中的波特率信息修改串行 UART 芯片内的传输波特率。最后返回 0。
82          for (i=0 ; i< (sizeof (*termios)) ; i++)
83                  ((char *)&tty->termios)[i]=get_fs_byte(i+(char *)termios);
84          change_speed(tty);
85          return 0;
86  }
87
    //// 读取 termio 结构中的信息。
    // 参数：tty - 指定终端的 tty 结构指针；termio - 保存 termio 结构信息的用户缓冲区。
88  static int get_termio(struct tty_struct * tty, struct termio * termio)
89  {
90          int i;
91          struct termio tmp_termio;
92
    // 首先验证用户的缓冲区指针所指内存区容量是否足够，如不够则分配内存。然后将 termios
    // 结构的信息复制到临时 termio 结构中。这两个结构基本相同，但输入、输出、控制和本地
    // 标志集数据类型不同。前者的是 long，而后者的是 short。因此先复制到临时 termio 结构
    // 中目的是为了进行数据类型转换。
93          verify_area(termio, sizeof (*termio));
94          tmp_termio.c_iflag = tty->termios.c_iflag;
95          tmp_termio.c_oflag = tty->termios.c_oflag;
96          tmp_termio.c_cflag = tty->termios.c_cflag;
97          tmp_termio.c_lflag = tty->termios.c_lflag;
98          tmp_termio.c_line = tty->termios.c_line;
99          for(i=0 ; i < NCC ; i++)
100                 tmp_termio.c_cc[i] = tty->termios.c_cc[i];
    // 最后逐字节地把临时 termio 结构中的信息复制到用户 termio 结构缓冲区中。并返回 0。
101         for (i=0 ; i< (sizeof (*termio)) ; i++)
102                 put_fs_byte( ((char *)&tmp_termio)[i] , i+(char *)termio );
103         return 0;
104 }
105
106 /*
107  * This only works as the 386 is low-byt-first
108  */
    /*
     * 下面 termio 设置函数仅适用于低字节在前的 386 CPU。
```

```
 */
//// 设置终端 termio 结构信息。
// 参数：tty - 指定终端的 tty 结构指针；termio - 用户数据区中 termio 结构。
// 将用户缓冲区 termio 的信息复制到终端的 termios 结构中。返回 0 。
109 static int set_termio(struct tty_struct * tty, struct termio * termio,
110                        int channel)
111 {
112         int i, retsig;
113         struct termio tmp_termio;
114
// 与 set_termios()一样，如果进程使用的终端的进程组号与进程的进程组号不同，即当前进
// 程终端不在前台，表示当前进程试图修改不受控制的终端的 termios 结构。因此根据 POSIX
// 标准的要求这里需要发送 SIGTTOU 信号让使用这个终端的进程先暂时停止执行，以让我们先
// 修改 termios 结构。如果发送信号函数 tty_signal()返回值是 ERESTARTSYS 或 EINTR，则等
// 一会再执行本次操作。
115         if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
116                 retsig = tty_signal(SIGTTOU, tty);
117                 if (retsig == -ERESTARTSYS || retsig == -EINTR)
118                         return retsig;
119         }
// 接着复制用户数据区中 termio 结构信息到临时 termio 结构中。然后再将 termio 结构的信息
// 复制到 tty 的 termios 结构中。这样做的目的是为了对其中模式标志集的类型进行转换，即
// 从 termio 的短整数类型转换成 termios 的长整数类型。但两种结构的 c_line 和 c_cc[]字段
// 是完全相同的。
120         for (i=0 ; i< (sizeof (*termio)) ; i++)
121                 ((char *)&tmp_termio)[i]=get_fs_byte(i+(char *)termio);
122         *(unsigned short *)&tty->termios.c_iflag = tmp_termio.c_iflag;
123         *(unsigned short *)&tty->termios.c_oflag = tmp_termio.c_oflag;
124         *(unsigned short *)&tty->termios.c_cflag = tmp_termio.c_cflag;
125         *(unsigned short *)&tty->termios.c_lflag = tmp_termio.c_lflag;
126         tty->termios.c_line = tmp_termio.c_line;
127         for(i=0 ; i < NCC ; i++)
128                 tty->termios.c_cc[i] = tmp_termio.c_cc[i];
// 最后因为用户有可能已修改了终端串行口传输波特率，所以这里再根据 termios 结构中的控制
// 模式标志 c_cflag 中的波特率信息修改串行 UART 芯片内的传输波特率，并返回 0。
129         change_speed(tty);
130         return 0;
131 }
132
//// tty 终端设备输入输出控制函数。
// 参数：dev - 设备号；cmd - ioctl 命令；arg - 操作参数指针。
// 该函数首先根据参数给出的设备号找出对应终端的 tty 结构，然后根据控制命令 cmd 分别进行
// 处理。
133 int tty_ioctl(int dev, int cmd, int arg)
134 {
135         struct tty_struct * tty;
136         int     pgrp;
137
// 首先根据设备号取得 tty 子设备号，从而取得终端的 tty 结构。若主设备号是 5（控制终端），
// 则进程的 tty 字段即是 tty 子设备号。此时如果进程的 tty 子设备号是负数，表明该进程没有
// 控制终端，即不能发出该 ioctl 调用，于是显示出错信息并停机。如果主设备号不是 5 而是 4，
// 我们就可以从设备号中取出子设备号。子设备号可以是 0（控制台终端）、1（串口 1 终端）、
// 2（串口 2 终端）。
```

```
138         if (MAJOR(dev) == 5) {
139                 dev=current->tty;
140                 if (dev<0)
141                         panic("tty_ioctl: dev<0");
142         } else
143                 dev=MINOR(dev);
    // 然后根据子设备号和 tty 表，我们可取得对应终端的 tty 结构。于是让 tty 指向对应子设备
    // 号的 tty 结构。然后再根据参数提供的 ioctl 命令 cmd 进行分别处理。144 行后半部分用于
    // 根据子设备号 dev 在 tty_table[]表中选择对应的 tty 结构。如果 dev = 0，表示正在使用
    // 前台终端，因此直接使用终端号 fg_console 作为 tty_table[] 项索引取 tty 结构。如果
    // dev 大于 0，那么就要分两种情况考虑：① dev 是虚拟终端号；② dev 是串行终端号或者
    // 伪终端号。对于虚拟终端其 tty 结构在 tty_table[]中索引项是 dev-1（0 -- 63）。对于
    // 其它类型终端，则它们的 tty 结构索引项就是 dev。例如，如果 dev = 64，表示是一个串
    // 行终端 1，则其 tty 结构就是 ttb_table[dev]。 如果 dev = 1，则对应终端的 tty 结构是
    // tty_table[0]。参见 tty_io.c 程序第 70 -- 73 行。
144         tty = tty_table + (dev ? ((dev < 64)? dev-1:dev) : fg_console);
145         switch (cmd) {
    // 取相应终端 termios 结构信息。此时参数 arg 是用户缓冲区指针。
146                 case TCGETS:
147                         return get_termios(tty,(struct termios *) arg);
    // 在设置 termios 结构信息之前，需要先等待输出队列中所有数据处理完毕，并且刷新（清空）
    // 输入队列。再接着执行下面设置终端 termios 结构的操作。
148                 case TCSETSF:
149                         flush(tty->read_q); /* fallthrough */ /* 接着继续执行 */
    // 在设置终端 termios 的信息之前，需要先等待输出队列中所有数据处理完（耗尽）。对于修改
    // 参数会影响输出的情况，就需要使用这种形式。
150                 case TCSETSW:
151                         wait_until_sent(tty); /* fallthrough */
    // 设置相应终端 termios 结构信息。此时参数 arg 是保存 termios 结构的用户缓冲区指针。
152                 case TCSETS:
153                         return set_termios(tty,(struct termios *) arg, dev);
    // 取相应终端 termio 结构中的信息。此时参数 arg 是用户缓冲区指针。
154                 case TCGETA:
155                         return get_termio(tty,(struct termio *) arg);
    // 在设置 termio 结构信息之前，需要先等待输出队列中所有数据处理完毕，并且刷新（清空）
    // 输入队列。再接着执行下面设置终端 termio 结构的操作。
156                 case TCSETAF:
157                         flush(tty->read_q); /* fallthrough */ /* 接着继续执行 */
    // 在设置终端 termios 的信息之前，需要先等待输出队列中所有数据处理完（耗尽）。对于修改
    // 参数会影响输出的情况，就需要使用这种形式。
158                 case TCSETAW:
159                         wait_until_sent(tty); /* fallthrough */
    // 设置相应终端 termio 结构信息。此时参数 arg 是保存 termio 结构的用户缓冲区指针。
160                 case TCSETA:
161                         return set_termio(tty,(struct termio *) arg, dev);
    // 如果参数 arg 值是 0，则等待输出队列处理完毕（空），并发送一个 break。
162                 case TCSBRK:
163                         if (!arg) {
164                                 wait_until_sent(tty);
165                                 send_break(tty);
166                         }
167                         return 0;
    // 开始/停止流控制。如果参数 arg 是 TCOOFF（Terminal Control Output OFF），则挂起输出；
```

```
    // 如果是 TCOON，则恢复挂起的输出。在挂起或恢复输出同时需要把写队列中的字符输出，以
    // 加快用户交互响应速度。如果 arg 是 TCIOFF（Terminal Control Input ON），则挂起输入；
    // 如果是 TCION，则重新开启挂起的输入。
168                 case TCXONC:
169                         switch (arg) {
170                         case TCOOFF:
171                                 tty->stopped = 1;       // 停止终端输出。
172                                 tty->write(tty);        // 写缓冲队列输出。
173                                 return 0;
174                         case TCOON:
175                                 tty->stopped = 0;       // 恢复终端输出。
176                                 tty->write(tty);
177                                 return 0;
    // 如果参数 arg 是 TCIOFF，表示要求终端停止输入，于是我们往终端写队列中放入 STOP 字符。
    // 当终端收到该字符时就会暂停输入。如果参数是 TCION，表示要发送一个 START 字符，让终
    // 端恢复传输。
    // STOP_CHAR(tty)定义为 ((tty)->termios.c_cc[VSTOP])，即取终端 termios 结构控制字符数
    // 组对应项值。若内核定义了_POSIX_VDISABLE（\0），那么当某一项值等于_POSIX_VDISABLE
    // 的值时，表示禁止使用相应的特殊字符。因此这里直接判断该值是否为 0 来确定要不要把停
    // 止控制字符放入终端写队列中。以下同。
178                         case TCIOFF:
179                                 if (STOP_CHAR(tty))
180                                         PUTCH(STOP_CHAR(tty),tty->write_q);
181                                 return 0;
182                         case TCION:
183                                 if (START_CHAR(tty))
184                                         PUTCH(START_CHAR(tty),tty->write_q);
185                                 return 0;
186                         }
187                         return -EINVAL; /* not implemented */
    // 刷新已写输出但还没有发送、或已收但还没有读的数据。如果参数 arg 是 0，则刷新（清空）
    // 输入队列；如果是 1，则刷新输出队列；如果是 2，则刷新输入和输出队列。
188                 case TCFLSH:
189                         if (arg==0)
190                                 flush(tty->read_q);
191                         else if (arg==1)
192                                 flush(tty->write_q);
193                         else if (arg==2) {
194                                 flush(tty->read_q);
195                                 flush(tty->write_q);
196                         } else
197                                 return -EINVAL;
198                         return 0;
    // 设置终端串行线路专用模式。
199                 case TIOCEXCL:
200                         return -EINVAL; /* not implemented */   /* 未实现 */
    // 复位终端串行线路专用模式。
201                 case TIOCNXCL:
202                         return -EINVAL; /* not implemented */
    // 设置 tty 为控制终端。（TIOCNOTTY - 不要控制终端）。
203                 case TIOCSCTTY:
204                         return -EINVAL; /* set controlling term NI */  /* 未实现 */
    // 读取终端进程组号（即读取前台进程组号）。 首先验证用户缓冲区长度，然后复制终端 tty
```

```
    // 的 pgrp 字段到用户缓冲区。此时参数 arg 是用户缓冲区指针。
205                 case TIOCGPGRP:                         // 实现库函数 tcgetpgrp()。
206                         verify_area((void *) arg,4);
207                         put_fs_long(tty->pgrp,(unsigned long *) arg);
208                         return 0;
    // 设置终端进程组号 pgrp（即设置前台进程组号）。 此时参数 arg 是用户缓冲区中进程组号
    // pgrp 的指针。执行该命令的前提条件是进程必须有控制终端。 如果当前进程没有控制终端，
    // 或者 dev 不是其控制终端，或者控制终端现在的确是正在处理的终端 dev，但进程的会话号
    // 与该终端 dev 的会话号不同，则返回无终端错误信息。
209                 case TIOCSPGRP:                         // 实现库函数 tcsetpgrp()。
210                         if ((current->tty < 0) ||
211                             (current->tty != dev) ||
212                             (tty->session != current->session))
213                                 return -ENOTTY;
    // 然后我们就从用户缓冲区中取得欲设置的进程组号，并对该组号的有效性进行验证。如果组
    // 号 pgrp 小于 0，则返回无效组号错误信息；如果 pgrp 的会话号与当前进程的不同，则返回
    // 许可错误信息。否则我们可以设中终端的进程组号为 prgp。此时 prgp 成为前台进程组。
214                         pgrp=get_fs_long((unsigned long *) arg);
215                         if (pgrp < 0)
216                                 return -EINVAL;
217                         if (session_of_pgrp(pgrp) != current->session)
218                                 return -EPERM;
219                         tty->pgrp = pgrp;
220                         return 0;
    // 返回输出队列中还未送出的字符数。首先验证用户缓冲区长度，然后复制队列中字符数给用户。
    // 此时参数 arg 是用户缓冲区指针。
221                 case TIOCOUTQ:
222                         verify_area((void *) arg,4);
223                         put_fs_long(CHARS(tty->write_q),(unsigned long *) arg);
224                         return 0;
    // 返回输入队列中还未读取的字符数。首先验证用户缓冲区长度，然后复制队列中字符数给用户。
    // 此时参数 arg 是用户缓冲区指针。
225                 case TIOCINQ:
226                         verify_area((void *) arg,4);
227                         put_fs_long(CHARS(tty->secondary),
228                                 (unsigned long *) arg);
229                         return 0;
    // 模拟终端输入操作。该命令以一个指向字符的指针作为参数，并假设该字符是在终端上键入的。
    // 用户必须在该控制终端上具有超级用户权限或具有读许可权限。
230                 case TIOCSTI:
231                         return -EINVAL; /* not implemented */   /* 未实现 */
    // 读取终端设备窗口大小信息（参见 termios.h 中的 winsize 结构）。
232                 case TIOCGWINSZ:
233                         return -EINVAL; /* not implemented */
    // 设置终端设备窗口大小信息（参见 winsize 结构）。
234                 case TIOCSWINSZ:
235                         return -EINVAL; /* not implemented */
    // 返回 MODEM 状态控制引线的当前状态比特位标志集（参见 termios.h 中 185 -- 196 行）。
236                 case TIOCMGET:
237                         return -EINVAL; /* not implemented */
    // 设置单个 modem 状态控制引线的状态（true 或 false）。
238                 case TIOCMBIS:
239                         return -EINVAL; /* not implemented */
```

```
    // 复位单个 MODEM 状态控制引线的状态。
240                 case TIOCMBIC:
241                         return -EINVAL; /* not implemented */
    // 设置 MODEM 状态引线的状态。如果某一比特位置位，则 modem 对应的状态引线将置为有效。
242                 case TIOCMSET:
243                         return -EINVAL; /* not implemented */
    // 读取软件载波检测标志（1 - 开启；0 - 关闭）。
244                 case TIOCGSOFTCAR:
245                         return -EINVAL; /* not implemented */
    // 设置软件载波检测标志（1 - 开启；0 - 关闭）。
246                 case TIOCSSOFTCAR:
247                         return -EINVAL; /* not implemented */
248                 default:
249                         return -EINVAL;
250         }
251 }
252
```

第11章 协处理器仿真程序

11.1 程序 11-1 linux/kernel/math/math_emulate.c

```
1  /*
2   * linux/kernel/math/math_emulate.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  /*
8   * Limited emulation 27.12.91 - mostly loads/stores, which gcc wants
9   * even for soft-float, unless you use bruce evans' patches. The patches
10  * are great, but they have to be re-applied for every version, and the
11  * library is different for soft-float and 80387. So emulation is more
12  * practical, even though it's slower.
13  *
14  * 28.12.91 - loads/stores work, even BCD. I'll have to start thinking
15  * about add/sub/mul/div. Urgel. I should find some good source, but I'll
16  * just fake up something.
17  *
18  * 30.12.91 - add/sub/mul/div/com seem to work mostly. I should really
19  * test every possible combination.
20  */
    /*
     * 仿真范围有限的程序 91.12.27 - 绝大多数是一些加载/存储指令。除非你使用
     * 了 Bruce Evans 的补丁程序，否则即使使用软件执行浮点运算，gcc 也需要这些
     * 指令。Bruce 的补丁程序非常好，但每次更换 gcc 版本你都得用这个补丁程序。
     * 而且对于软件浮点实现和 80387，所使用的库是不同的。因此使用仿真是更为实
     * 际的方法，尽管仿真方法更慢。
     *
     * 91.12.28 - 加载/存储协处理器指令可以用了，即使是 BCD 码的也能使用。我将
     * 开始考虑实现 add/sub/mul/div 指令。唉，我应该找一些好的资料，不过现在
     * 我会先仿造一些操作。
     *
     * 91.12.30 - add/sub/mul/div/com 这些指令好像大多数都可以使用了。我真应
     * 该测试每种指令可能的组合操作。
     */
21
22 /*
23  * This file is full of ugly macros etc: one problem was that gcc simply
24  * didn't want to make the structures as they should be: it has to try to
25  * align them. Sickening code, but at least I've hidden the ugly things
26  * in this one file: the other files don't need to know about these things.
27  *
28  * The other files also don't care about ST(x) etc - they just get addresses
29  * to 80-bit temporary reals, and do with them as they please. I wanted to
30  * hide most of the 387-specific things here.
31  */
    /*
```

```
 * 这个程序中到处都是些别扭的宏：问题之一是 gcc 就是不想把结构建立成其应该
 * 成为的样子：gcc 企图对结构进行对齐处理。真是讨厌，不过我起码已经把所有
 * 蹩脚的代码都隐藏在这么一个文件中了：其他程序文件不需要了解这些信息。
 *
 * 其他的程序也不需要知道 ST(x)等 80387 内部结构 - 它们只需要得到 80 位临时
 * 实数的地址就可以随意操作。我想尽可能在这里隐藏所有 387 专有信息。
 */

#include <signal.h>        // 信号头文件。定义信号符号，信号结构及信号操作函数原型。

#define __ALIGNED_TEMP_REAL 1
#include <linux/math_emu.h> // 协处理器头文件。定义临时实数结构和 387 寄存器操作宏等。
#include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

#define bswapw(x) __asm__("xchgb %%al,%%ah":"=a" (x):"" ((short)x)) // 交换 2 字节位置。
#define ST(x) (*__st((x)))                          // 取仿真的 ST(x)累加器值。
#define PST(x) ((const temp_real *) __st((x)))      // 取仿真的 ST(x)累加器的指针。

/*
 * We don't want these inlined - it gets too messy in the machine-code.
 */
/*
 * 我们不想让这些成为嵌入的语句 - 因为这会使得到的机器码太混乱。
 */
// 以下这些是相同名称浮点指令的仿真函数。
static void fpop(void);
static void fpush(void);
static void fxchg(temp_real_unaligned * a, temp_real_unaligned * b);
static temp_real_unaligned * __st(int i);

// 执行浮点指令仿真。
// 该函数首先检测仿真的 I387 结构状态字寄存器中是否有未屏蔽的异常标志置位。若有则对状
// 态字中忙标志 B 进行设置。然后把指令指针保存起来，并取出代码指针 EIP 处的 2 字节浮点
// 指令代码 code。接着分析代码 code，并根据其含义进行处理。针对不同代码类型值，Linus
// 使用了几个不同的 switch 程序块进行仿真处理。
// 参数是 info 结构的指针。
static void do_emu(struct info * info)
{
        unsigned short code;
        temp_real tmp;
        char * address;

// 该函数首先检测仿真的 I387 结构状态字寄存器中是否有未屏蔽的异常标志置位。若有就设置
// 状态字中的忙标志 B（位 15），否则复位 B 标志。然后我们把指令指针保存起来。再看看执
// 行本函数的代码是否是用户代码。如果不是，即调用者的代码段选择符不等于 0x0f，则说明
// 内核中有代码使用了浮点指令。于是在显示出浮点指令出的 CS、EIP 值和信息“内核中需要
// 数学仿真”后停机。
        if (I387.cwd & I387.swd & 0x3f)
                I387.swd |= 0x8000;                 // 设置忙标志 B。
        else
                I387.swd &= 0x7fff;                 // 清忙标志 B。
        ORIG_EIP = EIP;                             // 保存浮点指令指针。
```

```
63 /* 0x0007 means user code space */
64         if (CS != 0x000F) {                             // 不是用户代码则停机。
65                 printk("math_emulate: %04x:%08x\n\r",CS,EIP);
66                 panic("Math emulation needed in kernel");
67         }
   // 然后我们取出代码指针 EIP 处的 2 字节浮点指令代码 code。 由于 Intel CPU 存储数据时是
   // “小头”（Little endien）在前的，此时取出的代码正好与指令的第 1、第 2 字节顺序颠倒。
   // 因此我们需要交换一下 code 中两个字节的顺序。然后再屏蔽掉第 1 个代码字节中的 ESC 位
   // （二进制 11011）。接着把浮点指令指针 EIP 保存到 TSS 段 i387 结构中的 fip 字段中，而 CS
   // 保存到 fcs 字段中，同时把略微处理过的浮点指令代码 code 放到 fcs 字段的高 16 位中。
   // 保存这些值是为了在出现仿真的处理器异常时程序可以像使用真实的协处理器一样进行处理。
   // 最后让 EIP 指向随后的浮点指令或操作数。
68         code = get_fs_word((unsigned short *) EIP);  // 取 2 字节的浮点指令代码。
69         bswapw(code);                                   // 交换高低字节。
70         code &= 0x7ff;                                  // 屏蔽代码中的 ESC 码。
71         I387.fip = EIP;                                 // 保存指令指针。
72         *(unsigned short *) &I387.fcs = CS;             // 保存代码段选择符。
73         *(1+(unsigned short *) &I387.fcs) = code;       // 保存代码。
74         EIP += 2;                                       // 指令指针指向下一个字节。
   // 然后分析代码值 code，并根据其含义进行处理。针对不同代码类型值，Linus 使用了几个不同
   // 的 switch 程序块进行处理。首先，若指令操作码是具有固定代码值（与寄存器等无关），则
   // 在下面处理。
75         switch (code) {
76                 case 0x1d0: /* fnop */    /* 空操作指令 FNOP */
77                         return;
78                 case 0x1d1: case 0x1d2: case 0x1d3:  // 无效指令代码。发信号，退出。
79                 case 0x1d4: case 0x1d5: case 0x1d6: case 0x1d7:
80                         math_abort(info,1<<(SIGILL-1));
81                 case 0x1e0:                 // FCHS - 改变 ST 符号位。即 ST = -ST。
82                         ST(0).exponent ^= 0x8000;
83                         return;
84                 case 0x1e1:                 // FABS - 取绝对值。即 ST = |ST|。
85                         ST(0).exponent &= 0x7fff;
86                         return;
87                 case 0x1e2: case 0x1e3:   // 无效指令代码。发信号，退出。
88                         math_abort(info,1<<(SIGILL-1));
89                 case 0x1e4:                 // FTST - 测试 TS，同时设置状态字中 Cn。
90                         ftst(PST(0));
91                         return;
92                 case 0x1e5:                 // FXAM - 检查 TS 值，同时修改状态字中 Cn。
93                         printk("fxam not implemented\n\r");  // 未实现。发信号退出。
94                         math_abort(info,1<<(SIGILL-1));
95                 case 0x1e6: case 0x1e7:   // 无效指令代码。发信号，退出。
96                         math_abort(info,1<<(SIGILL-1));
97                 case 0x1e8:                 // FLD1 - 加载常数 1.0 到累加器 ST。
98                         fpush();
99                         ST(0) = CONST1;
100                        return;
101                case 0x1e9:                 // FLDL2T - 加载常数 Log2(10) 到累加器 ST。
102                        fpush();
103                        ST(0) = CONSTL2T;
104                        return;
105                case 0x1ea:                 // FLDL2E - 加载常数 Log2(e) 到累加器 ST。
```

```
106                     fpush();
107                     ST(0) = CONSTL2E;
108                     return;
109             case 0x1eb:             // FLDPI - 加载常数 Pi 到累加器 ST。
110                     fpush();
111                     ST(0) = CONSTPI;
112                     return;
113             case 0x1ec:             // FLDLG2 - 加载常数 Log10(2) 到累加器 ST。
114                     fpush();
115                     ST(0) = CONSTLG2;
116                     return;
117             case 0x1ed:             // FLDLN2 - 加载常数 Loge(2) 到累加器 ST。
118                     fpush();
119                     ST(0) = CONSTLN2;
120                     return;
121             case 0x1ee:             // FLDZ - 加载常数 0.0 到累加器 ST。
122                     fpush();
123                     ST(0) = CONSTZ;
124                     return;
125             case 0x1ef:             // 无效和未实现仿真指令代码。发信号，退出。
126                     math_abort(info,1<<(SIGILL-1));
127             case 0x1f0: case 0x1f1: case 0x1f2: case 0x1f3:
128             case 0x1f4: case 0x1f5: case 0x1f6: case 0x1f7:
129             case 0x1f8: case 0x1f9: case 0x1fa: case 0x1fb:
130             case 0x1fc: case 0x1fd: case 0x1fe: case 0x1ff:
131                     printk("%04x fxxx not implemented\n\r",code + 0xc800);
132                     math_abort(info,1<<(SIGILL-1));
133             case 0x2e9:             // FUCOMPP - 无次序比较。
134                     fucom(PST(1),PST(0));
135                     fpop(); fpop();
136                     return;
137             case 0x3d0: case 0x3d1:   // FNOP - 对 387。!!应该是 0x3e0, 0x3e1。
138                     return;
139             case 0x3e2:             // FCLEX - 清状态字中异常标志。
140                     I387.swd &= 0x7f00;
141                     return;
142             case 0x3e3:             // FINIT - 初始化协处理器。
143                     I387.cwd = 0x037f;
144                     I387.swd = 0x0000;
145                     I387.twd = 0x0000;
146                     return;
147             case 0x3e4:             // FNOP - 对 80387。
148                     return;
149             case 0x6d9:             // FCOMPP - ST(1)与 ST 比较，出栈操作两次。
150                     fcom(PST(1),PST(0));
151                     fpop(); fpop();
152                     return;
153             case 0x7e0:             // FSTSW AX - 保存当前状态字到 AX 寄存器中。
154                     *(short *) &EAX = I387.swd;
155                     return;
156     }

// 下面开始处理第 2 字节最后 3 比特是 REG 的指令。即 11011,XXXXXXXX,REG 形式的代码。
```

```
157         switch (code >> 3) {
158                 case 0x18:                  // FADD ST, ST(i)。
159                         fadd(PST(0),PST(code & 7),&tmp);
160                         real_to_real(&tmp,&ST(0));
161                         return;
162                 case 0x19:                  // FMUL ST, ST(i)。
163                         fmul(PST(0),PST(code & 7),&tmp);
164                         real_to_real(&tmp,&ST(0));
165                         return;
166                 case 0x1a:                  // FCOM ST(i)。
167                         fcom(PST(code & 7),&tmp);
168                         real_to_real(&tmp,&ST(0));
169                         return;
170                 case 0x1b:                  // FCOMP ST(i)。
171                         fcom(PST(code & 7),&tmp);
172                         real_to_real(&tmp,&ST(0));
173                         fpop();
174                         return;
175                 case 0x1c:                  // FSUB ST, ST(i)。
176                         real_to_real(&ST(code & 7),&tmp);
177                         tmp.exponent ^= 0x8000;
178                         fadd(PST(0),&tmp,&tmp);
179                         real_to_real(&tmp,&ST(0));
180                         return;
181                 case 0x1d:                  // FSUBR ST, ST(i)。
182                         ST(0).exponent ^= 0x8000;
183                         fadd(PST(0),PST(code & 7),&tmp);
184                         real_to_real(&tmp,&ST(0));
185                         return;
186                 case 0x1e:                  // FDIV ST, ST(i)。
187                         fdiv(PST(0),PST(code & 7),&tmp);
188                         real_to_real(&tmp,&ST(0));
189                         return;
190                 case 0x1f:                  // FDIVR ST, ST(i)。
191                         fdiv(PST(code & 7),PST(0),&tmp);
192                         real_to_real(&tmp,&ST(0));
193                         return;
194                 case 0x38:                  // FLD ST(i)。
195                         fpush();
196                         ST(0) = ST((code & 7)+1);
197                         return;
198                 case 0x39:                  // FXCH ST(i)。
199                         fxchg(&ST(0),&ST(code & 7));
200                         return;
201                 case 0x3b:                  // FSTP ST(i)。
202                         ST(code & 7) = ST(0);
203                         fpop();
204                         return;
205                 case 0x98:                  // FADD ST(i), ST。
206                         fadd(PST(0),PST(code & 7),&tmp);
207                         real_to_real(&tmp,&ST(code & 7));
208                         return;
209                 case 0x99:                  // FMUL ST(i), ST。
```

```
210                     fmul(PST(0),PST(code & 7),&tmp);
211                     real_to_real(&tmp,&ST(code & 7));
212                     return;
213             case 0x9a:              // FCOM ST(i)。
214                     fcom(PST(code & 7),PST(0));
215                     return;
216             case 0x9b:              // FCOMP ST(i)。
217                     fcom(PST(code & 7),PST(0));
218                     fpop();
219                     return;
220             case 0x9c:              // FSUBR ST(i), ST。
221                     ST(code & 7).exponent ^= 0x8000;
222                     fadd(PST(0),PST(code & 7),&tmp);
223                     real_to_real(&tmp,&ST(code & 7));
224                     return;
225             case 0x9d:              // FSUB ST(i), ST。
226                     real_to_real(&ST(0),&tmp);
227                     tmp.exponent ^= 0x8000;
228                     fadd(PST(code & 7),&tmp,&tmp);
229                     real_to_real(&tmp,&ST(code & 7));
230                     return;
231             case 0x9e:              // FDIVR ST(i), ST。
232                     fdiv(PST(0),PST(code & 7),&tmp);
233                     real_to_real(&tmp,&ST(code & 7));
234                     return;
235             case 0x9f:              // FDIV ST(i), ST。
236                     fdiv(PST(code & 7),PST(0),&tmp);
237                     real_to_real(&tmp,&ST(code & 7));
238                     return;
239             case 0xb8:              // FFREE ST(i)。未实现。
240                     printk("ffree not implemented\n\r");
241                     math_abort(info,1<<(SIGILL-1));
242             case 0xb9:              // FXCH ST(i)。
243                     fxchg(&ST(0),&ST(code & 7));
244                     return;
245             case 0xba:              // FST ST(i)。
246                     ST(code & 7) = ST(0);
247                     return;
248             case 0xbb:              // FSTP ST(i)。
249                     ST(code & 7) = ST(0);
250                     fpop();
251                     return;
252             case 0xbc:              // FUCOM ST(i)。
253                     fucom(PST(code & 7),PST(0));
254                     return;
255             case 0xbd:              // FUCOMP ST(i)。
256                     fucom(PST(code & 7),PST(0));
257                     fpop();
258                     return;
259             case 0xd8:              // FADDP ST(i), ST。
260                     fadd(PST(code & 7),PST(0),&tmp);
261                     real_to_real(&tmp,&ST(code & 7));
262                     fpop();
```

```
263                     return;
264             case 0xd9:              // FMULP ST(i), ST。
265                     fmul(PST(code & 7),PST(0),&tmp);
266                     real_to_real(&tmp,&ST(code & 7));
267                     fpop();
268                     return;
269             case 0xda:              // FCOMP ST(i)。
270                     fcom(PST(code & 7),PST(0));
271                     fpop();
272                     return;
273             case 0xdc:              // FSUBRP ST(i), ST。
274                     ST(code & 7).exponent ^= 0x8000;
275                     fadd(PST(0),PST(code & 7),&tmp);
276                     real_to_real(&tmp,&ST(code & 7));
277                     fpop();
278                     return;
279             case 0xdd:              // FSUBP ST(i), ST。
280                     real_to_real(&ST(0),&tmp);
281                     tmp.exponent ^= 0x8000;
282                     fadd(PST(code & 7),&tmp,&tmp);
283                     real_to_real(&tmp,&ST(code & 7));
284                     fpop();
285                     return;
286             case 0xde:              // FDIVRP ST(i), ST。
287                     fdiv(PST(0),PST(code & 7),&tmp);
288                     real_to_real(&tmp,&ST(code & 7));
289                     fpop();
290                     return;
291             case 0xdf:              // FDIVP ST(i), ST。
292                     fdiv(PST(code & 7),PST(0),&tmp);
293                     real_to_real(&tmp,&ST(code & 7));
294                     fpop();
295                     return;
296             case 0xf8:              // FFREE ST(i)。未实现。
297                     printk("ffree not implemented\n\r");
298                     math_abort(info,1<<(SIGILL-1));
299                     fpop();
300                     return;
301             case 0xf9:              // FXCH ST(i)。
302                     fxchg(&ST(0),&ST(code & 7));
303                     return;
304             case 0xfa:              // FSTP ST(i)。
305             case 0xfb:              // FSTP ST(i)。
306                     ST(code & 7) = ST(0);
307                     fpop();
308                     return;
309     }

    // 处理第 2 个字节位 7--6 是 MOD、位 2--0 是 R/M 的指令，即 11011,XXX,MOD,XXX,R/M 形式的
    // 代码。MOD 在各子程序中处理，因此这里首先让代码与上 0xe7（0b11100111）屏蔽掉 MOD。
310     switch ((code>>3) & 0xe7) {
311             case 0x22:              // FST  - 保存单精度实数（短实数）。
312                     put_short_real(PST(0),info,code);
```

```
313                 return;
314         case 0x23:              // FSTP  - 保存单精度实数（短实数）。
315                 put_short_real(PST(0),info,code);
316                 fpop();
317                 return;
318         case 0x24:              // FLDENV - 加载协处理器状态和控制寄存器等。
319                 address = ea(info,code);
320                 for (code = 0 ; code < 7 ; code++) {
321                         ((long *) & I387)[code] =
322                            get_fs_long((unsigned long *) address);
323                         address += 4;
324                 }
325                 return;
326         case 0x25:              // FLDCW - 加载控制字。
327                 address = ea(info,code);
328                 *(unsigned short *) &I387.cwd =
329                         get_fs_word((unsigned short *) address);
330                 return;
331         case 0x26:              // FSTENV - 储存协处理器状态和控制寄存器等。
332                 address = ea(info,code);
333                 verify_area(address,28);
334                 for (code = 0 ; code < 7 ; code++) {
335                         put_fs_long( ((long *) & I387)[code],
336                                 (unsigned long *) address);
337                         address += 4;
338                 }
339                 return;
340         case 0x27:              // FSTCW - 储存控制字。
341                 address = ea(info,code);
342                 verify_area(address,2);
343                 put_fs_word(I387.cwd,(short *) address);
344                 return;
345         case 0x62:              // FIST - 储存短整形数。
346                 put_long_int(PST(0),info,code);
347                 return;
348         case 0x63:              // FISTP - 储存短整形数。
349                 put_long_int(PST(0),info,code);
350                 fpop();
351                 return;
352         case 0x65:              // FLD - 加载扩展（临时）实数。
353                 fpush();
354                 get_temp_real(&tmp,info,code);
355                 real_to_real(&tmp,&ST(0));
356                 return;
357         case 0x67:              // FSTP - 储存扩展实数。
358                 put_temp_real(PST(0),info,code);
359                 fpop();
360                 return;
361         case 0xa2:              // FST - 储存双精度实数。
362                 put_long_real(PST(0),info,code);
363                 return;
364         case 0xa3:              // FSTP - 储存双精度实数。
365                 put_long_real(PST(0),info,code);
```

```
366                     fpop();
367                     return;
368             case 0xa4:              // FRSTOR - 恢复所有 108 字节的寄存器内容。
369                     address = ea(info,code);
370                     for (code = 0 ; code < 27 ; code++) {
371                             ((long *) & I387)[code] =
372                                get_fs_long((unsigned long *) address);
373                             address += 4;
374                     }
375                     return;
376             case 0xa6:              // FSAVE - 保存所有 108 字节寄存器内容。
377                     address = ea(info,code);
378                     verify_area(address,108);
379                     for (code = 0 ; code < 27 ; code++) {
380                             put_fs_long( ((long *) & I387)[code],
381                                     (unsigned long *) address);
382                             address += 4;
383                     }
384                     I387.cwd = 0x037f;
385                     I387.swd = 0x0000;
386                     I387.twd = 0x0000;
387                     return;
388             case 0xa7:              // FSTSW - 保存状态字。
389                     address = ea(info,code);
390                     verify_area(address,2);
391                     put_fs_word(I387.swd,(short *) address);
392                     return;
393             case 0xe2:              // FIST - 保存短整型数。
394                     put_short_int(PST(0),info,code);
395                     return;
396             case 0xe3:              // FISTP - 保存短整型数。
397                     put_short_int(PST(0),info,code);
398                     fpop();
399                     return;
400             case 0xe4:              // FBLD - 加载 BCD 类型数。
401                     fpush();
402                     get_BCD(&tmp,info,code);
403                     real_to_real(&tmp,&ST(0));
404                     return;
405             case 0xe5:              // FILD - 加载长整型数。
406                     fpush();
407                     get_longlong_int(&tmp,info,code);
408                     real_to_real(&tmp,&ST(0));
409                     return;
410             case 0xe6:              // FBSTP - 保存 BCD 类型数。
411                     put_BCD(PST(0),info,code);
412                     fpop();
413                     return;
414             case 0xe7:              // BISTP - 保存长整型数。
415                     put_longlong_int(PST(0),info,code);
416                     fpop();
417                     return;
418     }
```

```
    // 下面处理第 2 类浮点指令。首先根据指令代码的位 10--9 的 MF 值取指定类型的数，然后根据
    // OPA 和 OPB 的组合值进行分别处理。即处理 11011,MF,000,XXX,R/M 形式的指令代码。
419         switch (code >> 9) {
420                 case 0:                         // MF = 00，短实数（32 位实数）。
421                         get_short_real(&tmp,info,code);
422                         break;
423                 case 1:                         // MF = 01，短整数（32 位整数）。
424                         get_long_int(&tmp,info,code);
425                         break;
426                 case 2:                         // MF = 10，长实数（64 位实数）。
427                         get_long_real(&tmp,info,code);
428                         break;
429                 case 4:                         // MF = 11，长整数（64 位整数）。！应是 case 3。
430                         get_short_int(&tmp,info,code);
431         }
    // 处理浮点指令第 2 字节中的 OPB 代码。
432         switch ((code>>3) & 0x27) {
433                 case 0:                         // FADD。
434                         fadd(&tmp,PST(0),&tmp);
435                         real_to_real(&tmp,&ST(0));
436                         return;
437                 case 1:                         // FMUL。
438                         fmul(&tmp,PST(0),&tmp);
439                         real_to_real(&tmp,&ST(0));
440                         return;
441                 case 2:                         // FCOM。
442                         fcom(&tmp,PST(0));
443                         return;
444                 case 3:                         // FCOMP。
445                         fcom(&tmp,PST(0));
446                         fpop();
447                         return;
448                 case 4:                         // FSUB。
449                         tmp.exponent ^= 0x8000;
450                         fadd(&tmp,PST(0),&tmp);
451                         real_to_real(&tmp,&ST(0));
452                         return;
453                 case 5:                         // FSUBR。
454                         ST(0).exponent ^= 0x8000;
455                         fadd(&tmp,PST(0),&tmp);
456                         real_to_real(&tmp,&ST(0));
457                         return;
458                 case 6:                         // FDIV。
459                         fdiv(PST(0),&tmp,&tmp);
460                         real_to_real(&tmp,&ST(0));
461                         return;
462                 case 7:                         // FDIVR。
463                         fdiv(&tmp,PST(0),&tmp);
464                         real_to_real(&tmp,&ST(0));
465                         return;
466         }
    // 处理形如 11011,XX,1,XX,000,R/M 的指令代码。
467         if ((code & 0x138) == 0x100) {          // FLD、FILD。
```

```
468                         fpush();
469                         real_to_real(&tmp,&ST(0));
470                         return;
471         }
    // 其余均为无效指令。
472         printk("Unknown math-insns: %04x:%08x %04x\n\r",CS,EIP,code);
473         math_abort(info,1<<(SIGFPE-1));
474 }
475
    // CPU 异常中断 int7 调用的 80387 仿真接口函数。
    // 若当前进程没有使用过协处理器，就设置使用协处理器标志 used_math，然后初始化 80387
    // 的控制字、状态字和特征字。最后使用中断 int7 调用本函数的返回地址指针作为参数调用
    // 浮点指令仿真主函数 do_emu()。
    // 参数 ___false 是 _orig_eip。
476 void math_emulate(long ___false)
477 {
478         if (!current->used_math) {
479                 current->used_math = 1;
480                 I387.cwd = 0x037f;
481                 I387.swd = 0x0000;
482                 I387.twd = 0x0000;
483         }
484 /* &___false points to info->___orig_eip, so subtract 1 to get info */
485         do_emu((struct info *) ((&___false) - 1));
486 }
487
    // 终止仿真操作。
    // 当处理到无效指令代码或者未实现的指令代码时，该函数首先恢复程序的原 EIP，并发送指定
    // 信号给当前进程。最后将栈指针指向中断 int7 处理过程调用本函数的返回地址，直接返回到
    // 中断处理过程中。
488 void __math_abort(struct info * info, unsigned int signal)
489 {
490         EIP = ORIG_EIP;
491         current->signal |= signal;
492         __asm__("movl %0,%%esp ; ret"::"g" ((long) info));
493 }
494
    // 累加器栈弹出操作。
    // 将状态字 TOP 字段值加 1，并以 7 取模。
495 static void fpop(void)
496 {
497         unsigned long tmp;
498
499         tmp = I387.swd & 0xffffc7ff;
500         I387.swd += 0x00000800;
501         I387.swd &= 0x00003800;
502         I387.swd |= tmp;
503 }
504
    // 累加器栈入栈操作。
    // 将状态字 TOP 字段减 1（即加 7），并以 7 取模。
505 static void fpush(void)
506 {
```

```
507         unsigned long tmp;
508
509         tmp = I387.swd & 0xffffc7ff;
510         I387.swd += 0x00003800;
511         I387.swd &= 0x00003800;
512         I387.swd |= tmp;
513 }
514
    // 交换两个累加器寄存器的值。
515 static void fxchg(temp_real_unaligned * a, temp_real_unaligned * b)
516 {
517         temp_real_unaligned c;
518
519         c = *a;
520         *a = *b;
521         *b = c;
522 }
523
    // 取 ST(i)的内存指针。
    // 取状态字中 TOP 字段值。加上指定的物理数据寄存器号并取模，最后返回 ST(i)对应的指针。
524 static temp_real_unaligned * __st(int i)
525 {
526         i += I387.swd >> 11;            // 取状态字中 TOP 字段值。
527         i &= 7;
528         return (temp_real_unaligned *) (i*10 + (char *)(I387.st_space));
529 }
530
```

11.2 程序 11-2 linux/kernel/math/error.c

```
 1 /*
 2  * linux/kernel/math/error.c
 3  *
 4  * (C) 1991 Linus Torvalds
 5  */
 6
 7 #include <signal.h>        // 信号头文件。定义信号符号常量，信号结构及信号操作函数原型。
 8
 9 #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 数据等。
10
   // 协处理器错误中断 int16 调用的处理函数。
   // 当协处理器检测到自己发生错误时，就会通过 ERROR 引脚通知 CPU。下面代码用于处理协处理
   // 器发出的出错信号。并跳转去执行 math_error()。返回后将跳转到标号 ret_from_sys_call
   // 处继续执行。
11 void math_error(void)
12 {
13         __asm__("fnclex");           // 让 80387 清除状态字中所有异常标志位和忙位。
14         if (last_task_used_math)     // 若使用了协处理器，则设置协处理器出错信号。
15                 last_task_used_math->signal |= 1<<(SIGFPE-1);
16 }
17
```

11.3 程序 11-3 linux/kernel/math/ea.c

```
 1 /*
 2  * linux/kernel/math/ea.c
 3  *
 4  * (C) 1991 Linus Torvalds
 5  */
 6
 7 /*
 8  * Calculate the effective address.
 9  */
   /*
    * 计算有效地址。
    */
10
11 #include <stddef.h>          // 标准定义头文件。本程序使用了其中的 offsetof()定义。
12
13 #include <linux/math_emu.h> // 协处理器头文件。定义临时实数结构和 387 寄存器操作宏等。
14 #include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
15
   // info 结构中各个寄存器在结构中的偏移位置。offsetof()用于求指定字段在结构中的偏移位
   // 置。参见 include/stddef.h 文件。
16 static int __regoffset[] = {
17         offsetof(struct info,___eax),
18         offsetof(struct info,___ecx),
19         offsetof(struct info,___edx),
20         offsetof(struct info,___ebx),
21         offsetof(struct info,___esp),
22         offsetof(struct info,___ebp),
23         offsetof(struct info,___esi),
24         offsetof(struct info,___edi)
25 };
26
   // 取 info 结构中指定位置处寄存器内容。
27 #define REG(x) (*(long *)(__regoffset[(x)]+(char *) info))
28
   // 求 2 字节寻址模式中第 2 操作数指示字节 SIB（Scale，Index，Base）的值。
29 static char * sib(struct info * info, int mod)
30 {
31         unsigned char ss,index,base;
32         long offset = 0;
33
   // 首先从用户代码段中取得 SIB 字节，然后取出各个字段比特位值。
34         base = get_fs_byte((char *) EIP);
35         EIP++;
36         ss = base >> 6;                     // 比例因子大小 ss。
37         index = (base >> 3) & 7;            // 索引值索引代号 index。
38         base &= 7;                          // 基地址代号 base。
   // 如果索引代号为 0b100，表示无索引偏移值。否则索引偏移值 offset=对应寄存器内容*比例因子。
39         if (index == 4)
40                 offset = 0;
```

```
41          else
42                  offset = REG(index);
43          offset <<= ss;
    // 如果上一 MODRM 字节中的 MOD 不为零，或者 Base 不等于 0b101，则表示有偏移值在 base 指定的
    // 寄存器中。因此偏移 offset 需要再加上 base 对应寄存器中的内容。
44          if (mod || base != 5)
45                  offset += REG(base);
    // 如果 MOD=1，则表示偏移值为 1 字节。否则，若 MOD=2，或者 base=0b101，则偏移值为 4 字节。
46          if (mod == 1) {
47                  offset += (signed char) get_fs_byte((char *) EIP);
48                  EIP++;
49          } else if (mod == 2 || base == 5) {
50                  offset += (signed) get_fs_long((unsigned long *) EIP);
51                  EIP += 4;
52          }
    // 最后保存并返回偏移值。
53          I387.foo = offset;
54          I387.fos = 0x17;
55          return (char *) offset;
56  }
57
    // 根据指令中寻址模式字节计算有效地址值。
58  char * ea(struct info * info, unsigned short code)
59  {
60          unsigned char mod,rm;
61          long * tmp = &EAX;
62          int offset = 0;
63
    // 首先取代码中的 MOD 字段和 R/M 字段值。如果 MOD=0b11，表示是单字节指令，没有偏移字段。
    // 如果 R/M 字段=0b100，并且 MOD 不为 0b11，表示是 2 字节地址模式寻址，因此调用 sib()求
    // 出偏移值并返回即可。
64          mod = (code >> 6) & 3;          // MOD 字段。
65          rm = code & 7;                  // R/M 字段。
66          if (rm == 4 && mod != 3)
67                  return sib(info,mod);
    // 如果 R/M 字段为 0b101，并且 MOD 为 0，表示是单字节地址模式编码且后随 32 字节偏移值。
    // 于是取出用户代码中 4 字节偏移值，保存并返回之。
68          if (rm == 5 && !mod) {
69                  offset = get_fs_long((unsigned long *) EIP);
70                  EIP += 4;
71                  I387.foo = offset;
72                  I387.fos = 0x17;
73                  return (char *) offset;
74          }
    // 对于其余情况，则根据 MOD 进行处理。首先取出 R/M 代码对应寄存器内容的值作为指针 tmp。
    // 对于 MOD=0，无偏移值。对于 MOD=1，代码后随 1 字节偏移值。对于 MOD=2，代码后有 4 字节
    // 偏移值。最后保存并返回有效地址值。
75          tmp = & REG(rm);
76          switch (mod) {
77                  case 0: offset = 0; break;
78                  case 1:
79                          offset = (signed char) get_fs_byte((char *) EIP);
80                          EIP++;
```

```
81                          break;
82                  case 2:
83                          offset = (signed) get_fs_long((unsigned long *) EIP);
84                          EIP += 4;
85                          break;
86                  case 3:
87                          math_abort(info,1<<(SIGILL-1));
88          }
89          I387.foo = offset;
90          I387.fos = 0x17;
91          return offset + (char *) *tmp;
92  }
93
```

11.4 程序 11-4 linux/kernel/math/convert.c

```
1  /*
2   * linux/kernel/math/convert.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  #include <linux/math_emu.h> // 协处理器头文件。定义临时实数结构和 387 寄存器操作宏等。
8
9  /*
10  * NOTE!!! There is some "non-obvious" optimisations in the temp_to_long
11  * and temp_to_short conversion routines: don't touch them if you don't
12  * know what's going on. They are the adding of one in the rounding: the
13  * overflow bit is also used for adding one into the exponent. Thus it
14  * looks like the overflow would be incorrectly handled, but due to the
15  * way the IEEE numbers work, things are correct.
16  *
17  * There is no checking for total overflow in the conversions, though (ie
18  * if the temp-real number simply won't fit in a short- or long-real.)
19  */
    /*
     * 注意!!! 在 temp_to_long 和 temp_to_short 数据类型转换子程序中有些“不明显”
     * 的优化处理：如果不理解就不要随意修改。它们是舍入操作中的加 1：溢出位也同
     * 样被用于向指数中加 1。因此看上去溢出好像没有被正确地处理，但是由于 IEEE
     * 浮点数标准所规定数据格式的操作方式，这些做法是正确的。
     *
     * 不过这里没有对转换过程中总体溢出作检测（也即临时实数是否能够简单地放入短
     * 实数或长实数格式中）。
     */
20
    // 短实数转换成临时实数格式。
    // 短实数长度是 32 位，其有效数（尾数）长度是 23 位，指数是 8 位，还有 1 个符号位。
21  void short_to_temp(const short_real * a, temp_real * b)
22  {
    // 首先处理被转换的短实数是 0 的情况。若为 0，则设置对应临时实数 b 的有效数为 0。然后根
    // 据短实数符号位设置临时实数的符号位，即 exponent 的最高有效位。
23          if (!(*a & 0x7fffffff)) {
24                  b->a = b->b = 0;                        // 置临时实数的有效数 = 0。
25                  if (*a)
26                          b->exponent = 0x8000;           // 设置符号位。
27                  else
28                          b->exponent = 0;
29                  return;
30          }
    // 对于一般短实数，先确定对应临时实数的指数值。这里需要用到整型数偏置表示方法的概念。
    // 短实数指数的偏置量是 127，而临时实数指数的偏置量是 16383。因此在取出短实数中指数值
    // 后需要变更其中的偏置量为 16383。 此时就形成了临时实数格式的指数值 exponent。另外，
    // 如果短实数是负数，则需要设置临时实数的符号位（位 79）。下一步设置尾数值。方法是把
    // 短实数左移 8 位，让 23 位尾数最高有效位处于临时实数的位 62 处。而临时实数尾数的位 63
    // 处需要恒置一个 1，即需要或上 0x80000000。最后清掉临时实数低 32 位有效数。
```

```
31          b->exponent = ((*a>>23) & 0xff)-127+16383;    // 取出短实数指数位，更换偏置量。
32          if (*a<0)
33                  b->exponent |= 0x8000;                // 若为负数则设置符号位。
34          b->b = (*a<<8) | 0x80000000;                  // 放置尾数，添加固定 1 值。
35          b->a = 0;
36  }
37
    // 长实数转换成临时实数格式。
    // 方法与 short_to_temp()完全一样。不过长实数指数偏置量是 1023。
38  void long_to_temp(const long_real * a, temp_real * b)
39  {
40          if (!a->a && !(a->b & 0x7fffffff)) {
41                  b->a = b->b = 0;                      // 置临时实数的有效数 = 0。
42                  if (a->b)
43                          b->exponent = 0x8000;         // 设置符号位。
44                  else
45                          b->exponent = 0;
46                  return;
47          }
48          b->exponent = ((a->b >> 20) & 0x7ff)-1023+16383;  // 取长实数指数，更换偏置量。
49          if (a->b<0)
50                  b->exponent |= 0x8000;                // 若为负数则设置符号位。
51          b->b = 0x80000000 | (a->b<<11) | (((unsigned long)a->a)>>21); // 放置尾数，添 1。
52          b->a = a->a<<11;
53  }
54
    // 临时实数转换成短实数格式。
    // 过程与 short_to_temp()相反，但需要处理精度和舍入问题。
55  void temp_to_short(const temp_real * a, short_real * b)
56  {
    // 如果指数部分为 0，则根据有无符号位设置短实数为-0 或 0。
57          if (!(a->exponent & 0x7fff)) {
58                  *b = (a->exponent)?0x80000000:0;
59                  return;
60          }
    // 先处理指数部分。即更换临时实数指数偏置量（16383）为短实数的偏置量 127。
61          *b = ((((long) a->exponent)-16383+127) << 23) & 0x7f800000;
62          if (a->exponent < 0)                          // 若是负数则设置符号位。
63                  *b |= 0x80000000;
64          *b |= (a->b >> 8) & 0x007fffff;               // 取临时实数有效数高 23 位。
    // 根据控制字中的舍入设置执行舍入操作。
65          switch (ROUNDING) {
66                  case ROUND_NEAREST:
67                          if ((a->b & 0xff) > 0x80)
68                                  ++*b;
69                          break;
70                  case ROUND_DOWN:
71                          if ((a->exponent & 0x8000) && (a->b & 0xff))
72                                  ++*b;
73                          break;
74                  case ROUND_UP:
75                          if (!(a->exponent & 0x8000) && (a->b & 0xff))
76                                  ++*b;
```

```
77                          break;
78          }
79  }
80
    // 临时实数转换成长实数。
81  void temp_to_long(const temp_real * a, long_real * b)
82  {
83          if (!(a->exponent & 0x7fff)) {
84                  b->a = 0;
85                  b->b = (a->exponent)?0x80000000:0;
86                  return;
87          }
88          b->b = (((0x7fff & (long) a->exponent)-16383+1023) << 20) & 0x7ff00000;
89          if (a->exponent < 0)
90                  b->b |= 0x80000000;
91          b->b |= (a->b >> 11) & 0x000fffff;
92          b->a = a->b << 21;
93          b->a |= (a->a >> 11) & 0x001fffff;
94          switch (ROUNDING) {
95                  case ROUND_NEAREST:
96                          if ((a->a & 0x7ff) > 0x400)
97                                  __asm__("add1 $1,%0 ; adc1 $0,%1"
98                                          :"=r" (b->a), "=r" (b->b)
99                                          :"" (b->a), "1" (b->b));
100                         break;
101                 case ROUND_DOWN:
102                         if ((a->exponent & 0x8000) && (a->b & 0xff))
103                                 __asm__("add1 $1,%0 ; adc1 $0,%1"
104                                         :"=r" (b->a), "=r" (b->b)
105                                         :"" (b->a), "1" (b->b));
106                         break;
107                 case ROUND_UP:
108                         if (!(a->exponent & 0x8000) && (a->b & 0xff))
109                                 __asm__("add1 $1,%0 ; adc1 $0,%1"
110                                         :"=r" (b->a), "=r" (b->b)
111                                         :"" (b->a), "1" (b->b));
112                         break;
113         }
114 }
115
    // 临时实数转换成临时整数格式。
    // 临时整数也用 10 字节表示。其中低 8 字节是无符号整数值，高 2 字节表示指数值和符号位。
    // 如果高 2 字节最高有效位为 1，则表示是负数；若位 0，表示是正数。
116 void real_to_int(const temp_real * a, temp_int * b)
117 {
118         int shift =  16383 + 63 - (a->exponent & 0x7fff);
119         unsigned long underflow;
120
121         b->a = b->b = underflow = 0;
122         b->sign = (a->exponent < 0);
123         if (shift < 0) {
124                 set_OE();
125                 return;
```

```
126         }
127         if (shift < 32) {
128                 b->b = a->b; b->a = a->a;
129         } else if (shift < 64) {
130                 b->a = a->b; underflow = a->a;
131                 shift -= 32;
132         } else if (shift < 96) {
133                 underflow = a->b;
134                 shift -= 64;
135         } else
136                 return;
137         __asm__("shrdl %2,%1,%0"
138                 :"=r" (underflow), "=r" (b->a)
139                 :"c" ((char) shift), "" (underflow), "1" (b->a));
140         __asm__("shrdl %2,%1,%0"
141                 :"=r" (b->a), "=r" (b->b)
142                 :"c" ((char) shift), "" (b->a), "1" (b->b));
143         __asm__("shrl %1,%0"
144                 :"=r" (b->b)
145                 :"c" ((char) shift), "" (b->b));
146         switch (ROUNDING) {
147                 case ROUND_NEAREST:
148                         __asm__("addl %4,%5 ; adcl $0,%0 ; adcl $0,%1"
149                                 :"=r" (b->a), "=r" (b->b)
150                                 :"" (b->a), "1" (b->b)
151                                 ,"r" (0x7fffffff + (b->a & 1))
152                                 ,"m" (*&underflow));
153                         break;
154                 case ROUND_UP:
155                         if (!b->sign && underflow)
156                                 __asm__("addl $1,%0 ; adcl $0,%1"
157                                         :"=r" (b->a), "=r" (b->b)
158                                         :"" (b->a), "1" (b->b));
159                         break;
160                 case ROUND_DOWN:
161                         if (b->sign && underflow)
162                                 __asm__("addl $1,%0 ; adcl $0,%1"
163                                         :"=r" (b->a), "=r" (b->b)
164                                         :"" (b->a), "1" (b->b));
165                         break;
166         }
167 }
168
    // 临时整数转换成临时实数格式。
169 void int_to_real(const temp_int * a, temp_real * b)
170 {
    // 由于原值是整数，所以转换成临时实数时指数除了需要加上偏置量 16383 外，还要加上 63。
    // 表示有效数要乘上 2 的 63 次方，即都是整数值。
171         b->a = a->a;
172         b->b = a->b;
173         if (b->a || b->b)
174                 b->exponent = 16383 + 63 + (a->sign? 0x8000:0);
175         else {
```

```
176                 b->exponent = 0;
177                 return;
178         }
    // 对格式转换后的临时实数进行规格化处理，即让有效数最高有效位不是 0。
179         while (b->b >= 0) {
180                 b->exponent--;
181                 __asm__("addl %0,%0 ; adcl %1,%1"
182                         :"=r" (b->a),"=r" (b->b)
183                         :"" (b->a),"1" (b->b));
184         }
185 }
186
```

11.5 程序 11-5 linux/kernel/math/add.c

```
 1 /*
 2  * linux/kernel/math/add.c
 3  *
 4  * (C) 1991 Linus Torvalds
 5  */
 6
 7 /*
 8  * temporary real addition routine.
 9  *
10  * NOTE! These aren't exact: they are only 62 bits wide, and don't do
11  * correct rounding. Fast hack. The reason is that we shift right the
12  * values by two, in order not to have overflow (1 bit), and to be able
13  * to move the sign into the mantissa (1 bit). Much simpler algorithms,
14  * and 62 bits (61 really - no rounding) accuracy is usually enough. The
15  * only time you should notice anything weird is when adding 64-bit
16  * integers together. When using doubles (52 bits accuracy), the
17  * 61-bit accuracy never shows at all.
18  */
   /*
    * 临时实数加法子程序。
    *
    * 注意！ 这些并不精确：它们只有 62 比特宽度，并且不能进行正确地舍入操作。
    * 这些仅是草就之作。原因是为了不会溢出（1 比特），我们把值右移了 2 位，
    * 并且使得符号位（1 比特）能够移入尾数中。这是非常简单的算法，而且 62 位
    * （实际上是 61 位 - 没有舍入）的精度通常也足够了。只有当你把 64 位的整数
    * 相加时才会发觉一些奇怪的问题。当使用双精度（52 比特精度）数据时，是永
    * 远不可能超过 61 比特精度的。
    */
19
20 #include <linux/math_emu.h> // 协处理器头文件。定义临时实数结构和 387 寄存器操作宏等。
21
   // 求一个数的负数（二进制补码）表示。
   // 把临时实数尾数（有效数）取反后再加 1。
   // 参数 a 是临时实数结构。其中 a、b 字段组合是实数的有效数。
22 #define NEGINT(a) \
23 __asm__("notl %0 ; notl %1 ; addl $1,%0 ; adcl $0,%1" \
24         :"=r" (a->a),"=r" (a->b) \
25         :"" (a->a),"1" (a->b))
26
   // 尾数符号化。
   // 即把临时实数变换成指数和整数表示形式，以便于仿真运算。因此我们这里称其为仿真格式。
27 static void signify(temp_real * a)
28 {
   // 把 64 位二进制尾数右移 2 位（因此指数需要加 2）。因为指数字段 exponent 的最高比特位是
   // 符号位，所以若指数值小于零，说明该数是负数。于是则把尾数用补码表示（取负）。然后把
   // 指数取正值。此时尾数中不仅包含移过 2 位的有效数，而且还包含数值的符号位。
   // 30 行上：%0 - a->a；%1 - a->b。汇编指令“shrdl $2, %1, %0”执行双精度（64 位）右移，
   // 即把组合尾数<b,a>右移 2 位。由于该移动操作不会改变%1（a->b）中的值，因此还需要单独
   // 对其右移 2 位。
```

```
29          a->exponent += 2;
30          __asm__("shrd1 $2,%1,%0 ; shr1 $2,%1"   // 使用双精度指令把尾数右移 2 位。
31                  :"=r" (a->a), "=r" (a->b)
32                  :"" (a->a), "1" (a->b));
33          if (a->exponent < 0)                    // 是负数，则尾数用补码表示（取负值）。
34                  NEGINT(a);
35          a->exponent &= 0x7fff;                  // 去掉符号位（若有）。
36  }
37
    // 尾数非符号化。
    // 将仿真格式转换为临时实数格式。即把指数和整数表示的实数转换为临时实数格式。
38  static void unsignify(temp_real * a)
39  {
    // 对于值为 0 的数不用处理，直接返回。否则，我们先复位临时实数格式的符号位。然后判断
    // 尾数的高位 long 字段 a->b 是否带有符号位。若有，则在 exponent 字段添加符号位，同时
    // 把尾数用无符号数形式表示（取补）。最后对尾数进行规格化处理，同时指数值作相应递减。
    // 即执行左移操作，使得尾数最高有效位不为 0（最后 a->b 值表现为负值）。
40          if (!(a->a || a->b)) {                  // 若值为 0 就返回。
41                  a->exponent = 0;
42                  return;
43          }
44          a->exponent &= 0x7fff;                  // 去掉符号位（若有）。
45          if (a->b < 0) {                         // 是负数，则尾数取正值。
46                  NEGINT(a);
47                  a->exponent |= 0x8000;          // 临时实数添加置符号位。
48          }
49          while (a->b >= 0) {
50                  a->exponent--;                  // 对尾数进行规格化处理。
51                  __asm__("add1 %0,%0 ; adc1 %1,%1"
52                          :"=r" (a->a), "=r" (a->b)
53                          :"" (a->a), "1" (a->b));
54          }
55  }
56
    // 仿真浮点加法指令运算。
    // 临时实数参数 src1 + src2 ➔ result。
57  void fadd(const temp_real * src1, const temp_real * src2, temp_real * result)
58  {
59          temp_real a,b;
60          int x1,x2,shift;
61
    // 首先取两个数的指数值 x1、x2（去掉符号位）。然后让变量 a 等于其中最大值，shift 为指数
    // 差值（即相差 2 的倍数值）。
62          x1 = src1->exponent & 0x7fff;
63          x2 = src2->exponent & 0x7fff;
64          if (x1 > x2) {
65                  a = *src1;
66                  b = *src2;
67                  shift = x1-x2;
68          } else {
69                  a = *src2;
70                  b = *src1;
71                  shift = x2-x1;
```

```
72          }
    // 若两者相差太大，大于等于 2 的 64 次方，则我们可以忽略小的那个数，即 b 值。于是直接返
    // 回 a 值即可。否则，若相差大于等于 2 的 32 次方，那么我们可以忽略小值 b 中的低 32 位值。
    // 于是我们把 b 的高 long 字段值 b.b 右移 32 位，即放到 b.a 中。然后把 b 的指数值相应地增
    // 加 32 次方。即指数差值减去 32。这样调整之后，相加的两个数的尾数基本上落在相同区域中。
73          if (shift >= 64) {
74                  *result = a;
75                  return;
76          }
77          if (shift >= 32) {
78                  b.a = b.b;
79                  b.b = 0;
80                  shift -= 32;
81          }
    // 接着再进行细致地调整，以将相加两者调整成相同。调整方法是把小值 b 的尾数右移 shift
    // 各比特位。这样两者的指数相同，处于同一个数量级下。我们就可以对尾数进行相加运算了。
    // 相加之前我们需要先把它们转换成仿真运算格式。在加法运算后再变换会临时实数格式。
82          __asm__("shrd1 %4,%1,%0 ; shr1 %4,%1"                   // 双精度（64 位）右移。
83                  :"=r" (b.a), "=r" (b.b)
84                  :"" (b.a), "1" (b.b), "c" ((char) shift));
85          signify(&a);                                            // 变换格式。
86          signify(&b);
87          __asm__("add1 %4,%0 ; adc1 %5,%1"                       // 执行加法运算。
88                  :"=r" (a.a), "=r" (a.b)
89                  :"" (a.a), "1" (a.b), "g" (b.a), "g" (b.b));
90          unsignify(&a);                                          // 再变换会临时实数格式。
91          *result = a;
92  }
93
```

11.6 程序 11-6 linux/kernel/math/compare.c

```
 1 /*
 2  * linux/kernel/math/compare.c
 3  *
 4  * (C) 1991 Linus Torvalds
 5  */
 6
 7 /*
 8  * temporary real comparison routines
 9  */
   /*
    * 累加器中临时实数比较子程序。
    */
10
11 #include <linux/math_emu.h> // 协处理器头文件。定义临时实数结构和 387 寄存器操作宏等。
12
   // 复位状态字中的 C3、C2、C1 和 C0 条件位。
13 #define clear_Cx() (I387.swd &= ~0x4500)
14
   // 对临时实数 a 进行规格化处理。即表示成指数、有效数形式。
   // 例如：102.345 表示成 1.02345 X 10^2。 0.0001234 表示成 1.234 X 10^-4。当然，函数中是
   // 二进制表示。
15 static void normalize(temp_real * a)
16 {
17         int i = a->exponent & 0x7fff;           // 取指数值（略去符号位）。
18         int sign = a->exponent & 0x8000;        // 取符号位。
19
   // 如果临时实数 a 的 64 位有效数（尾数）为 0，那么说明 a 等于 0。于是清 a 的指数，并返回。
20         if (!(a->a || a->b)) {
21                 a->exponent = 0;
22                 return;
23         }
   // 如果 a 的尾数最左端有 0 值比特位，那么将尾数左移，同时调整指数值（递减）。直到尾数
   // 的 b 字段最高有效位 MSB 是 1 位置（此时 b 表现为负值）。最后再添加上符号位。
24         while (i && a->b >= 0) {
25                 i--;
26                 __asm__("addl %0,%0 ; adcl %1,%1"
27                         :"=r" (a->a), "=r" (a->b)
28                         :"" (a->a), "1" (a->b));
29         }
30         a->exponent = i | sign;
31 }
32
   // 仿真浮点指令 FTST。
   // 即栈定累加器 ST(0)与 0 比较，并根据比较结果设置条件位。若 ST > 0.0，则 C3，C2，C0
   // 分别为 000；若 ST < 0.0，则条件位为 001；若 ST == 0.0，则条件位是 100；若不可比较，
   // 则条件位为 111。
33 void ftst(const temp_real * a)
34 {
35         temp_real b;
```

```
36
   // 首先清状态字中条件标志位，并对比较值 b（ST）进行规格化处理。若 b 不等于零并且设置
   // 了符号位（是负数），则设置条件位 C0。否则设置条件位 C3。
37         clear_Cx();
38         b = *a;
39         normalize(&b);
40         if (b.a || b.b || b.exponent) {
41                 if (b.exponent < 0)
42                         set_C0();
43         } else
44                 set_C3();
45 }
46
   // 仿真浮点指令 FCOM。
   // 比较两个参数 src1、src2。并根据比较结果设置条件位。若 src1 > src2，则 C3，C2，C0
   // 分别为 000；若 src1 < src2，则条件位为 001；若两者相等，则条件位是 100。
47 void fcom(const temp_real * src1, const temp_real * src2)
48 {
49         temp_real a;
50
51         a = *src1;
52         a.exponent ^= 0x8000;                   // 符号位取反。
53         fadd(&a,src2,&a);                       // 两者相加（即相减）。
54         ftst(&a);                               // 测试结果并设置条件位。
55 }
56
   // 仿真浮点指令 FUCOM（无次序比较）。
   // 用于操作数之一是 NaN 的比较。
57 void fucom(const temp_real * src1, const temp_real * src2)
58 {
59         fcom(src1,src2);
60 }
61
```

11.7 程序 11-7 linux/kernel/math/get_put.c

```
 1 /*
 2  * linux/kernel/math/get_put.c
 3  *
 4  * (C) 1991 Linus Torvalds
 5  */
 6
 7 /*
 8  * This file handles all accesses to user memory: getting and putting
 9  * ints/reals/BCD etc. This is the only part that concerns itself with
10  * other than temporary real format. All other cals are strictly temp_real.
11  */
   /*
    * 本程序处理所有对用户内存的访问：获取和存入指令/实数值/BCD 数值等。这是
    * 涉及临时实数以外其他格式仅有的部分。所有其他运算全都使用临时实数格式。
    */
12 #include <signal.h>        // 信号头文件。定义信号符号，信号结构及信号操作函数原型。
13
14 #include <linux/math_emu.h> // 协处理器头文件。定义临时实数结构和 387 寄存器操作宏等。
15 #include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
16 #include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
17
   // 取用户内存中的短实数（单精度实数）。
   // 根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，取得短实数
   // 所在有效地址（math/ea.c），然后从用户数据区读取相应实数值。最后把用户短实数转换成
   // 临时实数（math/convert.c）。
   // 参数：tmp - 转换成临时实数后的指针；info - info 结构指针；code - 指令代码。
18 void get_short_real(temp_real * tmp,
19         struct info * info, unsigned short code)
20 {
21         char * addr;
22         short_real sr;
23
24         addr = ea(info,code);                          // 计算有效地址。
25         sr = get_fs_long((unsigned long *) addr);      // 取用户数据区中的值。
26         short_to_temp(&sr,tmp);                        // 转换成临时实数格式。
27 }
28
   // 取用户内存中的长实数（双精度实数）。
   // 首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，取得长
   // 实数所在有效地址（math/ea.c），然后从用户数据区读取相应实数值。最后把用户实数值转
   // 换成临时实数（math/convert.c）。
   // 参数：tmp - 转换成临时实数后的指针；info - info 结构指针；code - 指令代码。
29 void get_long_real(temp_real * tmp,
30         struct info * info, unsigned short code)
31 {
32         char * addr;
33         long_real lr;
34
35         addr = ea(info,code);                          // 取指令中的有效地址值。
```

```
36          lr.a = get_fs_long((unsigned long *) addr);       // 取长 8 字节实数。
37          lr.b = get_fs_long(1 + (unsigned long *) addr);
38          long_to_temp(&lr,tmp);                            // 转换成临时实数格式。
39  }
40
    // 取用户内存中的临时实数。
    // 首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，取得临
    // 时实数所在有效地址（math/ea.c），然后从用户数据区读取相应临时实数值。
    // 参数：tmp - 转换成临时实数后的指针；info - info 结构指针；code - 指令代码。
41  void get_temp_real(temp_real * tmp,
42          struct info * info, unsigned short code)
43  {
44          char * addr;
45
46          addr = ea(info,code);                             // 取指令中的有效地址值。
47          tmp->a = get_fs_long((unsigned long *) addr);
48          tmp->b = get_fs_long(1 + (unsigned long *) addr);
49          tmp->exponent = get_fs_word(4 + (unsigned short *) addr);
50  }
51
    // 取用户内存中的短整数并转换成临时实数格式。
    // 临时整数也用 10 字节表示。其中低 8 字节是无符号整数值，高 2 字节表示指数值和符号位。
    // 如果高 2 字节最高有效位为 1，则表示是负数；若最高有效位是 0，表示是正数。
    // 该函数首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，
    // 取得短整数所在有效地址（math/ea.c），然后从用户数据区读取相应整数值，并保存为临时
    // 整数格式。最后把临时整数值转换成临时实数（math/convert.c）。
    // 参数：tmp - 转换成临时实数后的指针；info - info 结构指针；code - 指令代码。
52  void get_short_int(temp_real * tmp,
53          struct info * info, unsigned short code)
54  {
55          char * addr;
56          temp_int ti;
57
58          addr = ea(info,code);                 // 取指令中的有效地址值。
59          ti.a = (signed short) get_fs_word((unsigned short *) addr);
60          ti.b = 0;
61          if (ti.sign = (ti.a < 0))             // 若是负数，则设置临时整数符号位。
62                  ti.a = - ti.a;                // 临时整数“尾数”部分为无符号数。
63          int_to_real(&ti,tmp);                 // 把临时整数转换成临时实数格式。
64  }
65
    // 取用户内存中的长整数并转换成临时实数格式。
    // 首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，取得长
    // 整数所在有效地址（math/ea.c），然后从用户数据区读取相应整数值，并保存为临时整数格
    // 式。最后把临时整数值转换成临时实数（math/convert.c）。
    // 参数：tmp - 转换成临时实数后的指针；info - info 结构指针；code - 指令代码。
66  void get_long_int(temp_real * tmp,
67          struct info * info, unsigned short code)
68  {
69          char * addr;
70          temp_int ti;
71
72          addr = ea(info,code);                 // 取指令中的有效地址值。
```

```
73          ti.a = get_fs_long((unsigned long *) addr);
74          ti.b = 0;
75          if (ti.sign = (ti.a < 0))           // 若是负数，则设置临时整数符号位。
76                  ti.a = - ti.a;              // 临时整数“尾数”部分为无符号数。
77          int_to_real(&ti,tmp);               // 把临时整数转换成临时实数格式。
78  }
79
    // 取用户内存中的 64 位长整数并转换成临时实数格式。
    // 首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，取得
    // 64 位长整数所在有效地址（math/ea.c），然后从用户数据区读取相应整数值，并保存为临
    // 时整数格式。最后再把临时整数值转换成临时实数（math/convert.c）。
    // 参数：tmp - 转换成临时实数后的指针；info - info 结构指针；code - 指令代码。
80  void get_longlong_int(temp_real * tmp,
81          struct info * info, unsigned short code)
82  {
83          char * addr;
84          temp_int ti;
85
86          addr = ea(info,code);                           // 取指令中的有效地址值。
87          ti.a = get_fs_long((unsigned long *) addr);     // 取用户 64 位长整数。
88          ti.b = get_fs_long(1 + (unsigned long *) addr);
89          if (ti.sign = (ti.b < 0))                   // 若是负数则设置临时整数符号位。
90                  __asm__("notl %0 ; notl %1\n\t"     // 同时取反加 1 和进位调整。
91                          "addl $1,%0 ; adcl $0,%1"
92                          :"=r" (ti.a),"=r" (ti.b)
93                          :"" (ti.a),"1" (ti.b));
94          int_to_real(&ti,tmp);                       // 把临时整数转换成临时实数格式。
95  }
96
    // 将一个 64 位整数（例如 N）乘 10。
    // 这个宏用于下面 BCD 码数值转换成临时实数格式过程中。方法是：N<<1 + N<<3。
97  #define MUL10(low,high) \
98  __asm__("addl %0,%0 ; adcl %1,%1\n\t" \
99  "movl %0,%%ecx ; movl %1,%%ebx\n\t" \
100 "addl %0,%0 ; adcl %1,%1\n\t" \
101 "addl %0,%0 ; adcl %1,%1\n\t" \
102 "addl %%ecx,%0 ; adcl %%ebx,%1" \
103 :"=a" (low),"=d" (high) \
104 :"" (low),"1" (high):"cx","bx")
105
    // 64 位加法。
    // 把 32 位的无符号数 val 加到 64 位数 <high,low> 中。
106 #define ADD64(val,low,high) \
107 __asm__("addl %4,%0 ; adcl $0,%1":"=r" (low),"=r" (high) \
108 :"" (low),"1" (high),"r" ((unsigned long) (val)))
109
    // 取用户内存中的 BCD 码数值并转换成临时实数格式。
    // 该函数首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，
    // 取得 BCD 码所在有效地址（math/ea.c），然后从用户数据区读取 10 字节相应 BCD 码值（其
    // 中 1 字节用于符号），同时转换成临时整数形式。最后把临时整数值转换成临时实数。
    // 参数：tmp - 转换成临时实数后的指针；info - info 结构指针；code - 指令代码。
110 void get_BCD(temp_real * tmp, struct info * info, unsigned short code)
111 {
```

```
112         int k;
113         char * addr;
114         temp_int i;
115         unsigned char c;
116
    // 取得 BCD 码数值所在内存有效地址。然后从最后 1 个 BCD 码字节（最高有效位）开始处理。
    // 先取得 BCD 码数值的符号位，并设置临时整数的符号位。然后把 9 字节的 BCD 码值转换成
    // 临时整数格式，最后再把临时整数值转换成临时实数。
117         addr = ea(info,code);                       // 取有效地址。
118         addr += 9;                                  // 指向最后一个（第 10 个）字节。
119         i.sign = 0x80 & get_fs_byte(addr--);        // 取其中符号位。
120         i.a = i.b = 0;
121         for (k = 0; k < 9; k++) {                   // 转换成临时整数格式。
122                 c = get_fs_byte(addr--);
123                 MUL10(i.a, i.b);
124                 ADD64((c>>4), i.a, i.b);
125                 MUL10(i.a, i.b);
126                 ADD64((c&0xf), i.a, i.b);
127         }
128         int_to_real(&i,tmp);                        // 转换成临时实数格式。
129 }
130
    // 把运算结果以短（单精度）实数格式保存到用户数据区中。
    // 该函数首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，
    // 取得保存结果的有效地址 addr，然后把临时实数格式的结果转换成短实数格式并存储到有效
    // 地址 addr 处。
    // 参数：tmp - 临时实数格式结果值；info - info 结构指针；code - 指令代码。
131 void put_short_real(const temp_real * tmp,
132         struct info * info, unsigned short code)
133 {
134         char * addr;
135         short real sr;
136
137         addr = ea(info,code);                       // 取有效地址。
138         verify_area(addr,4);                        // 为保存结果验证或分配内存。
139         temp_to_short(tmp,&sr);                     // 结果转换成短实数格式。
140         put_fs_long(sr,(unsigned long *) addr);     // 存储数据到用户内存区。
141 }
142
    // 把运算结果以长（双精度）实数格式保存到用户数据区中。
    // 该函数首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，
    // 取得保存结果的有效地址 addr，然后把临时实数格式的结果转换成长实数格式，并存储到有
    // 效地址 addr 处。
    // 参数：tmp - 临时实数格式结果值；info - info 结构指针；code - 指令代码。
143 void put_long_real(const temp_real * tmp,
144         struct info * info, unsigned short code)
145 {
146         char * addr;
147         long real lr;
148
149         addr = ea(info,code);                       // 取有效地址。
150         verify_area(addr,8);                        // 为保存结果验证或分配内存。
151         temp_to_long(tmp,&lr);                      // 结果转换成长实数格式。
```

```
152         put_fs_long(lr.a, (unsigned long *) addr);   // 存储数据到用户内存区。
153         put_fs_long(lr.b, 1 + (unsigned long *) addr);
154 }
155
    // 把运算结果以临时实数格式保存到用户数据区中。
    // 该函数首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，
    // 取得保存结果的有效地址 addr，然后把临时实数存储到有效地址 addr 处。
    // 参数：tmp - 临时实数格式结果值；info - info 结构指针；code - 指令代码。
156 void put_temp_real(const temp_real * tmp,
157         struct info * info, unsigned short code)
158 {
159         char * addr;
160
161         addr = ea(info,code);                          // 取有效地址。
162         verify_area(addr,10);                          // 为保存结果验证或分配内存。
163         put_fs_long(tmp->a, (unsigned long *) addr); // 存储数据到用户内存区。
164         put_fs_long(tmp->b, 1 + (unsigned long *) addr);
165         put_fs_word(tmp->exponent, 4 + (short *) addr);
166 }
167
    // 把运算结果以短整数格式保存到用户数据区中。
    // 该函数首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，
    // 取得保存结果的有效地址 addr，然后把临时实数格式的结果转换成临时整数格式。如果是负
    // 数则设置整数符号位。最后把整数保存到用户内存中。
    // 参数：tmp - 临时实数格式结果值；info - info 结构指针；code - 指令代码。
168 void put_short_int(const temp_real * tmp,
169         struct info * info, unsigned short code)
170 {
171         char * addr;
172         temp_int ti;
173
174         addr = ea(info,code);                          // 取有效地址。
175         real_to_int(tmp,&ti);                          // 转换成临时整数格式。
176         verify_area(addr,2);                           // 验证或分配存储内存。
177         if (ti.sign)                                   // 若有符号位，则取负数值。
178                 ti.a = -ti.a;
179         put_fs_word(ti.a,(short *) addr);              // 存储到用户数据区中。
180 }
181
    // 把运算结果以长整数格式保存到用户数据区中。
    // 该函数首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，
    // 取得保存结果的有效地址 addr，然后把临时实数格式的结果转换成临时整数格式。如果是负
    // 数则设置整数符号位。最后把整数保存到用户内存中。
    // 参数：tmp - 临时实数格式结果值；info - info 结构指针；code - 指令代码。
182 void put_long_int(const temp_real * tmp,
183         struct info * info, unsigned short code)
184 {
185         char * addr;
186         temp_int ti;
187
188         addr = ea(info,code);                          // 取有效地址。
189         real_to_int(tmp,&ti);                          // 转换成临时整数格式。
190         verify_area(addr,4);                           // 验证或分配存储内存。
```

```
191         if (ti.sign)                                    // 若有符号位，则取负数值。
192                 ti.a = -ti.a;
193         put_fs_long(ti.a,(unsigned long *) addr);       // 存储到用户数据区中。
194 }
195
    // 把运算结果以 64 位整数格式保存到用户数据区中。
    // 该函数首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，
    // 取得保存结果的有效地址 addr，然后把临时实数格式的结果转换成临时整数格式。如果是负
    // 数则设置整数符号位。最后把整数保存到用户内存中。
    // 参数：tmp - 临时实数格式结果值；info - info 结构指针；code - 指令代码。
196 void put_longlong_int(const temp_real * tmp,
197         struct info * info, unsigned short code)
198 {
199         char * addr;
200         temp_int ti;
201
202         addr = ea(info,code);                           // 取有效地址。
203         real_to_int(tmp,&ti);                           // 转换成临时整数格式。
204         verify_area(addr,8);                            // 验证存储区域。
205         if (ti.sign)                                    // 若是负数，则取反加 1。
206                 __asm__("notl %0 ; notl %1\n\t"
207                         "addl $1,%0 ; adcl $0,%1"
208                         :"=r" (ti.a),"=r" (ti.b)
209                         :"" (ti.a),"1" (ti.b));
210         put_fs_long(ti.a,(unsigned long *) addr);       // 存储到用户数据区中。
211         put_fs_long(ti.b,1 + (unsigned long *) addr);
212 }
213
    // 无符号数<high, low>除以 10，余数放在 rem 中。
214 #define DIV10(low,high,rem) \
215 __asm__("divl %6 ; xchgl %1,%2 ; divl %6" \
216         :"=d" (rem),"=a" (low),"=b" (high) \
217         :"" (0),"1" (high),"2" (low),"c" (10))
218
    // 把运算结果以 BCD 码格式保存到用户数据区中。
    // 该函数首先根据浮点指令代码中寻址模式字节中的内容和 info 结构中当前寄存器中的内容，
    // 取得保存结果的有效地址 addr，并验证保存 10 字节 BCD 码的用户空间。然后把临时实数格式
    // 的结果转换成 BCD 码格式的数据并保存到用户内存中。如果是负数则设置最高存储字节的最高
    // 有效位。
    // 参数：tmp - 临时实数格式结果值；info - info 结构指针；code - 指令代码。
219 void put_BCD(const temp_real * tmp,struct info * info, unsigned short code)
220 {
221         int k,rem;
222         char * addr;
223         temp_int i;
224         unsigned char c;
225
226         addr = ea(info,code);                           // 取有效地址。
227         verify_area(addr,10);                           // 验证存储空间容量。
228         real_to_int(tmp,&i);                            // 转换成临时整数格式。
229         if (i.sign)                                   // 若是负数，则设置符号字节最高有效位。
230                 put_fs_byte(0x80, addr+9);
231         else                                            // 否则符号字节设置为 0。
```

```
232                 put_fs_byte(0, addr+9);
233         for (k = 0; k < 9; k++) {                       // 临时整数转换成 BCD 码并保存。
234                 DIV10(i.a,i.b,rem);
235                 c = rem;
236                 DIV10(i.a,i.b,rem);
237                 c += rem<<4;
238                 put_fs_byte(c,addr++);
239         }
240 }
241
```

11.8 程序 11-8 linux/kernel/math/mul.c

```
1  /*
2   * linux/kernel/math/mul.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  /*
8   * temporary real multiplication routine.
9   */
   /*
    * 临时实数乘法子程序。
    */
10
11 #include <linux/math_emu.h> // 协处理器头文件。定义临时实数结构和 387 寄存器操作宏等。
12
   // 把 c 指针处的 16 字节值左移 1 位（乘 2）。
13 static void shift(int * c)
14 {
15         __asm__("movl (%0),%%eax ; addl %%eax,(%0)\n\t"
16                 "movl 4(%0),%%eax ; adcl %%eax,4(%0)\n\t"
17                 "movl 8(%0),%%eax ; adcl %%eax,8(%0)\n\t"
18                 "movl 12(%0),%%eax ; adcl %%eax,12(%0)"
19                 ::"r" ((long) c):"ax");
20 }
21
   // 2 个临时实数相乘，结果放在 c 指针处（16 字节）。
22 static void mul64(const temp_real * a, const temp_real * b, int * c)
23 {
24         __asm__("movl (%0),%%eax\n\t"
25                 "mull (%1)\n\t"
26                 "movl %%eax,(%2)\n\t"
27                 "movl %%edx,4(%2)\n\t"
28                 "movl 4(%0),%%eax\n\t"
29                 "mull 4(%1)\n\t"
30                 "movl %%eax,8(%2)\n\t"
31                 "movl %%edx,12(%2)\n\t"
32                 "movl (%0),%%eax\n\t"
33                 "mull 4(%1)\n\t"
34                 "addl %%eax,4(%2)\n\t"
35                 "adcl %%edx,8(%2)\n\t"
36                 "adcl $0,12(%2)\n\t"
37                 "movl 4(%0),%%eax\n\t"
38                 "mull (%1)\n\t"
39                 "addl %%eax,4(%2)\n\t"
40                 "adcl %%edx,8(%2)\n\t"
41                 "adcl $0,12(%2)"
42                 ::"b" ((long) a), "c" ((long) b), "D" ((long) c)
43                 :"ax", "dx");
44 }
```

```
45
   // 仿真浮点指令 FMUL。
   // 临时实数 src1 * scr2 ➔ result 处。
46 void fmul(const temp real * src1, const temp real * src2, temp real * result)
47 {
48         int i,sign;
49         int tmp[4] = {0,0,0,0};
50
   // 首先确定两数相乘的符号。符号值等于两者符号位异或值。然后计算乘后的指数值。相乘时
   // 指数值需要相加。但是由于指数使用偏执数格式保存，两个数的指数相加时偏置量也被加了
   // 两次，因此需要减掉一个偏置量值（临时实数的偏置量是 16383）。
51         sign = (src1->exponent ^ src2->exponent) & 0x8000;
52         i = (src1->exponent & 0x7fff) + (src2->exponent & 0x7fff) - 16383 + 1;
   // 如果结果指数变成了负值，表示两数相乘后产生下溢。于是直接返回带符号的零值。
   // 如果结果指数大于 0x7fff，表示产生上溢，于是设置状态字溢出异常标志位，并返回。
53         if (i<0) {
54                 result->exponent = sign;
55                 result->a = result->b = 0;
56                 return;
57         }
58         if (i>0x7fff) {
59                 set OE();
60                 return;
61         }
   // 如果两数尾数相乘后结果不为 0，则对结果尾数进行规格化处理。即左移结果尾数值，使得
   // 最高有效位为 1。同时相应地调整指数值。如果两数的尾数相乘后 16 字节的结果尾数为 0，
   // 则也设置指数值为 0。最后把相乘结果保存在临时实数变量 result 中。
62         mul64(src1,src2,tmp);
63         if (tmp[0] || tmp[1] || tmp[2] || tmp[3])
64                 while (i && tmp[3] >= 0) {
65                         i--;
66                         shift(tmp);
67                 }
68         else
69                 i = 0;
70         result->exponent = i | sign;
71         result->a = tmp[2];
72         result->b = tmp[3];
73 }
74
```

11.9 程序 11-9 linux/kernel/math/div.c

```
 1 /*
 2  * linux/kernel/math/div.c
 3  *
 4  * (C) 1991 Linus Torvalds
 5  */
 6
 7 /*
 8  * temporary real division routine.
 9  */
10
11 #include <linux/math_emu.h> // 协处理器头文件。定义临时实数结构和 387 寄存器操作宏等。
12
   // 将指针 c 指向的 4 字节中内容左移 1 位。
13 static void shift_left(int * c)
14 {
15         __asm__ __volatile__("movl (%0),%%eax ; addl %%eax,(%0)\n\t"
16                 "movl 4(%0),%%eax ; adcl %%eax,4(%0)\n\t"
17                 "movl 8(%0),%%eax ; adcl %%eax,8(%0)\n\t"
18                 "movl 12(%0),%%eax ; adcl %%eax,12(%0)"
19                 ::"r" ((long) c):"ax");
20 }
21
   // 将指针 c 指向的 4 字节中内容右移 1 位。
22 static void shift_right(int * c)
23 {
24         __asm__("shrl $1,12(%0) ; rcrl $1,8(%0) ; rcrl $1,4(%0) ; rcrl $1,(%0)"
25                 ::"r" ((long) c));
26 }
27
   // 减法运算。
   // 16 字节减法运算，a - b ➔ a。最后根据是否有借位（CF=1）设置 ok。若无借位（CF=0）
   // 则 ok = 1。否则 ok=0。
28 static int try_sub(int * a, int * b)
29 {
30         char ok;
31
32         __asm__ __volatile__("movl (%1),%%eax ; subl %%eax,(%2)\n\t"
33                 "movl 4(%1),%%eax ; sbbl %%eax,4(%2)\n\t"
34                 "movl 8(%1),%%eax ; sbbl %%eax,8(%2)\n\t"
35                 "movl 12(%1),%%eax ; sbbl %%eax,12(%2)\n\t"
36                 "setae %%al":"=a" (ok):"c" ((long) a),"d" ((long) b));
37         return ok;
38 }
39
   // 16 字节除法。
   // 参数 a /b ➔ c。利用减法模拟多字节除法。
40 static void div64(int * a, int * b, int * c)
41 {
42         int tmp[4];
```

```
43          int i;
44          unsigned int mask = 0;
45
46          c += 4;
47          for (i = 0 ; i<64 ; i++) {
48                  if (!(mask >>= 1)) {
49                          c--;
50                          mask = 0x80000000;
51                  }
52                  tmp[0] = a[0]; tmp[1] = a[1];
53                  tmp[2] = a[2]; tmp[3] = a[3];
54                  if (try_sub(b,tmp)) {
55                          *c |= mask;
56                          a[0] = tmp[0]; a[1] = tmp[1];
57                          a[2] = tmp[2]; a[3] = tmp[3];
58                  }
59                  shift_right(b);
60          }
61  }
62
    // 仿真浮点指令 FDIV。
63  void fdiv(const temp_real * src1, const temp_real * src2, temp_real * result)
64  {
65          int i,sign;
66          int a[4],b[4],tmp[4] = {0,0,0,0};
67
68          sign = (src1->exponent ^ src2->exponent) & 0x8000;
69          if (!(src2->a || src2->b)) {
70                  set_ZE();
71                  return;
72          }
73          i = (src1->exponent & 0x7fff) - (src2->exponent & 0x7fff) + 16383;
74          if (i<0) {
75                  set_UE();
76                  result->exponent = sign;
77                  result->a = result->b = 0;
78                  return;
79          }
80          a[0] = a[1] = 0;
81          a[2] = src1->a;
82          a[3] = src1->b;
83          b[0] = b[1] = 0;
84          b[2] = src2->a;
85          b[3] = src2->b;
86          while (b[3] >= 0) {
87                  i++;
88                  shift_left(b);
89          }
90          div64(a,b,tmp);
91          if (tmp[0] || tmp[1] || tmp[2] || tmp[3]) {
92                  while (i && tmp[3] >= 0) {
93                          i--;
94                          shift_left(tmp);
```

```
95                  }
96                  if (tmp[3] >= 0)
97                          set_DE();
98          } else
99                  i = 0;
100         if (i>0x7fff) {
101                 set_OE();
102                 return;
103         }
104         if (tmp[0] || tmp[1])
105                 set_PE();
106         result->exponent = i | sign;
107         result->a = tmp[2];
108         result->b = tmp[3];
109 }
110
```

第12章 文件系统程序

12.1 程序 12-1 linux/fs/buffer.c

```
 1 /*
 2  *  linux/fs/buffer.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  *  'buffer.c' implements the buffer-cache functions. Race-conditions have
 9  * been avoided by NEVER letting a interrupt change a buffer (except for the
10  * data, of course), but instead letting the caller do it. NOTE! As interrupts
11  * can wake up a caller, some cli-sti sequences are needed to check for
12  * sleep-on-calls. These should be extremely quick, though (I hope).
13  */
   /*
    * 'buffer.c'用于实现缓冲区高速缓存功能。通过不让中断处理过程改变缓冲区，而是让调
    * 用者来执行，避免了竞争条件（当然除改变数据以外）。注意！由于中断可以唤醒一个调
    * 用者，因此就需要开关中断指令（cli-sti）序列来检测由于调用而睡眠。但需要非常地快
    * （我希望是这样）。
    */
14
15 /*
16  * NOTE! There is one discordant note here: checking floppies for
17  * disk change. This is where it fits best, I think, as it should
18  * invalidate changed floppy-disk-caches.
19  */
   /*
    * 注意！有一个程序应不属于这里：检测软盘是否更换。但我想这里是放置
    * 该程序最好的地方了，因为它需要使已更换软盘缓冲失效。
    */
20
21 #include <stdarg.h>       // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了-个
                             // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于
                             // vsprintf、vprintf、vfprintf 函数。
22
23 #include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）可选项。
24 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据，
                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
25 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
26 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入汇编宏。
27 #include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
28
   // 变量 end 是由编译时的连接程序 ld 生成，用于表明内核代码的末端，即指明内核模块末端
   // 位置，参见错误!未找到引用源。。也可以从编译内核时生成的 System.map 文件中查出。这里用它来表
   // 明高速缓冲区开始于内核代码末端位置。
   // 第 33 行上的 buffer_wait 变量是等待空闲缓冲块而睡眠的任务队列头指针。它与缓冲块头
   // 部结构中 b_wait 指针的作用不同。当任务申请一个缓冲块而正好遇到系统缺乏可用空闲缓
```

```
   // 冲块时，当前任务就会被添加到 buffer_wait 睡眠等待队列中。而 b_wait 则是专门供等待
   // 指定缓冲块（即 b_wait 对应的缓冲块）的任务使用的等待队列头指针。
29 extern int end;
30 struct buffer_head * start_buffer = (struct buffer_head *) &end;
31 struct buffer_head * hash_table[NR_HASH];        // NR_HASH = 307 项。
32 static struct buffer_head * free_list;           // 空闲缓冲块链表头指针。
33 static struct task_struct * buffer_wait = NULL;  // 等待空闲缓冲块而睡眠的任务队列。

   // 下面定义系统缓冲区中含有的缓冲块个数。这里，NR_BUFFERS 是一个定义在 linux/fs.h 头
   // 文件第 48 行的宏，其值即是变量名 nr_buffers，并且在 fs.h 文件第 172 行声明为全局变量。
   // 大写名称通常都是一个宏名称，Linus 这样编写代码是为了利用这个大写名称来隐含地表示
   // nr_buffers 是一个在内核初始化之后不再改变的“常量”。它将在初始化函数 buffer_init()
   // 中被设置（第 371 行）。
34 int NR_BUFFERS = 0;                             // 系统含有缓冲块个数。
35
   //// 等待指定缓冲块解锁。
   // 如果指定的缓冲块 bh 已经上锁就让进程不可中断地睡眠在该缓冲块的等待队列 b_wait 中。
   // 在缓冲块解锁时，其等待队列上的所有进程将被唤醒。虽然是在关闭中断（cli）之后去睡
   // 眠的，但这样做并不会影响在其他进程上下文中响应中断。因为每个进程都在自己的 TSS 段
   // 中保存了标志寄存器 EFLAGS 的值，所以在进程切换时 CPU 中当前 EFLAGS 的值也随之改变。
   // 使用 sleep_on()进入睡眠状态的进程需要用 wake_up()明确地唤醒。
36 static inline void wait_on_buffer(struct buffer_head * bh)
37 {
38         cli();                                  // 关中断。
39         while (bh->b_lock)                      // 如果已被上锁则进程进入睡眠，等待其解锁。
40                 sleep_on(&bh->b_wait);
41         sti();                                  // 开中断。
42 }
43
   //// 设备数据同步。
   // 同步设备和内存高速缓冲中数据。其中，sync_inodes()定义在 inode.c，59 行。
44 int sys_sync(void)
45 {
46         int i;
47         struct buffer_head * bh;
48
   // 首先调用 i 节点同步函数，把内存 i 节点表中所有修改过的 i 节点写入高速缓冲中。然后
   // 扫描所有高速缓冲区，对已被修改的缓冲块产生写盘请求，将缓冲中数据写入盘中，做到
   // 高速缓冲中的数据与设备中的同步。
49         sync_inodes();                          /* write out inodes into buffers */
50         bh = start_buffer;                      // bh 指向缓冲区开始处。
51         for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
52                 wait_on_buffer(bh);             // 等待缓冲区解锁（如果已上锁的话）。
53                 if (bh->b_dirt)
54                         ll_rw_block(WRITE,bh);  // 产生写设备块请求。
55         }
56         return 0;
57 }
58
   //// 对指定设备进行高速缓冲数据与设备上数据的同步操作。
   // 该函数首先搜索高速缓冲区中所有缓冲块。对于指定设备 dev 的缓冲块，若其数据已被修改
   // 过就写入盘中（同步操作）。然后把内存中 i 节点表数据写入高速缓冲中。之后再对指定设
   // 备 dev 执行一次与上述相同的写盘操作。
```

```
59 int sync_dev(int dev)
60 {
61         int i;
62         struct buffer_head * bh;
63
   // 首先对参数指定的设备执行数据同步操作，让设备上的数据与高速缓冲区中的数据同步。
   // 方法是扫描高速缓冲区中所有缓冲块，对指定设备 dev 的缓冲块，先检测其是否已被上锁，
   // 若已被上锁就睡眠等待其解锁。然后再判断一次该缓冲块是否还是指定设备的缓冲块并且
   // 已修改过（b_dirt 标志置位），若是就对其执行写盘操作。因为在我们睡眠期间该缓冲块
   // 有可能已被释放或者被挪作它用，所以在继续执行前需要再次判断一下该缓冲块是否还是
   // 指定设备的缓冲块，
64         bh = start_buffer;                          // bh 指向缓冲区开始处。
65         for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
66                 if (bh->b_dev != dev)               // 不是设备 dev 的缓冲块则继续。
67                         continue;
68                 wait_on_buffer(bh);                 // 等待缓冲区解锁（如果已上锁的话）。
69                 if (bh->b_dev == dev && bh->b_dirt)
70                         ll_rw_block(WRITE,bh);
71         }
   // 再将 i 节点数据写入高速缓冲。让 i 节点表 inode_table 中的 inode 与缓冲中的信息同步。
72         sync_inodes();
   // 然后在高速缓冲中的数据更新之后，再把它们与设备中的数据同步。这里采用两遍同步操作
   // 是为了提高内核执行效率。第一遍缓冲区同步操作可以让内核中许多“脏块”变干净，使得
   // i 节点的同步操作能够高效执行。本次缓冲区同步操作则把那些由于 i 节点同步操作而又变
   // 脏的缓冲块与设备中数据同步。
73         bh = start_buffer;
74         for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
75                 if (bh->b_dev != dev)
76                         continue;
77                 wait_on_buffer(bh);
78                 if (bh->b_dev == dev && bh->b_dirt)
79                         ll_rw_block(WRITE,bh);
80         }
81         return 0;
82 }
83
   //// 使指定设备在高速缓冲区中的数据无效。
   // 扫描高速缓冲区中所有缓冲块。对指定设备的缓冲块复位其有效(更新)标志和已修改标志。
84 void inline invalidate_buffers(int dev)
85 {
86         int i;
87         struct buffer_head * bh;
88
89         bh = start_buffer;
90         for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
91                 if (bh->b_dev != dev)               // 如果不是指定设备的缓冲块，则
92                         continue;                   // 继续扫描下一块。
93                 wait_on_buffer(bh);                 // 等待该缓冲区解锁（如果已被上锁）。
   // 由于进程执行过睡眠等待，所以需要再判断一下缓冲区是否是指定设备的。
94                 if (bh->b_dev == dev)
95                         bh->b_uptodate = bh->b_dirt = 0;
96         }
97 }
```

```
 98
 99 /*
100  * This routine checks whether a floppy has been changed, and
101  * invalidates all buffer-cache-entries in that case. This
102  * is a relatively slow routine, so we have to try to minimize using
103  * it. Thus it is called only upon a 'mount' or 'open'. This
104  * is the best way of combining speed and utility, I think.
105  * People changing diskettes in the middle of an operation deserve
106  * to loose :-)
107  *
108  * NOTE! Although currently this is only for floppies, the idea is
109  * that any additional removable block-device will use this routine,
110  * and that mount/open needn't know that floppies/whatever are
111  * special.
112  */
    /*
     * 该子程序检查一个软盘是否已被更换，如果已经更换就使高速缓冲中与该软驱
     * 对应的所有缓冲区无效。该子程序相对来说较慢，所以我们要尽量少使用它。
     * 所以仅在执行'mount'或'open'时才调用它。我想这是将速度和实用性相结合的
     * 最好方法。若在操作过程中更换软盘，就会导致数据的丢失。这是咎由自取☺。
     *
     * 注意！尽管目前该子程序仅用于软盘，以后任何可移动介质的块设备都将使用该
     * 程序，mount/open 操作不需要知道是软盘还是其他什么特殊介质。
     */
    //// 检查磁盘是否更换，如果已更换就使对应高速缓冲区无效。
113 void check_disk_change(int dev)
114 {
115         int i;
116
    // 首先检测一下是不是软盘设备。因为现在仅支持软盘可移动介质。如果不是则退出。然后
    // 测试软盘是否已更换，如果没有则退出。floppy_change()在 blk_drv/floppy.c 第 139 行。
117         if (MAJOR(dev) != 2)
118                 return;
119         if (!floppy_change(dev & 0x03))
120                 return;
    // 软盘已经更换，所以释放对应设备的 i 节点位图和逻辑块位图所占的高速缓冲区；并使该
    // 设备的 i 节点和数据块信息所占踞的高速缓冲块无效。
121         for (i=0 ; i<NR_SUPER ; i++)
122                 if (super_block[i].s_dev == dev)
123                         put_super(super_block[i].s_dev);
124         invalidate_inodes(dev);
125         invalidate_buffers(dev);
126 }
127
    // 下面两行代码是 hash（散列）函数定义和 hash 表项的计算宏。
    // hash 表的主要作用是减少查找比较元素所花费的时间。通过在元素的存储位置与关键字之间
    // 建立一个对应关系（hash 函数），我们就可以直接通过函数计算立刻查询到指定的元素。建
    // 立 hash 函数的指导条件主要是尽量确保散列到任何数组项的概率基本相等。建立函数的方法
    // 有多种，这里 Linux 0.12 主要采用了关键字除留余数法。因为我们寻找的缓冲块有两个条件，
    // 即设备号 dev 和缓冲块号 block，因此设计的 hash 函数肯定需要包含这两个关键值。这里两个
    // 关键字的异或操作只是计算关键值的一种方法。再对关键值进行 MOD 运算就可以保证函数所计
    // 算得到的值都处于函数数组项范围内。
128 #define _hashfn(dev,block) (((unsigned)(dev^block))%NR_HASH)
```

```
#define hash(dev,block) hash_table[_hashfn(dev,block)]

//// 从 hash 队列和空闲缓冲队列中移走缓冲块。
// hash 队列是双向链表结构，空闲缓冲块队列是双向循环链表结构。
static inline void remove_from_queues(struct buffer_head * bh)
{
/* remove from hash-queue */
/* 从 hash 队列中移除缓冲块 */
        if (bh->b_next)
                bh->b_next->b_prev = bh->b_prev;
        if (bh->b_prev)
                bh->b_prev->b_next = bh->b_next;
// 如果该缓冲区是该队列的头一个块，则让 hash 表的对应项指向本队列中的下一个缓冲区。
        if (hash(bh->b_dev,bh->b_blocknr) == bh)
                hash(bh->b_dev,bh->b_blocknr) = bh->b_next;
/* remove from free list */
/* 从空闲缓冲块表中移除缓冲块 */
        if (!(bh->b_prev_free) || !(bh->b_next_free))
                panic("Free block list corrupted");
        bh->b_prev_free->b_next_free = bh->b_next_free;
        bh->b_next_free->b_prev_free = bh->b_prev_free;
// 如果空闲链表头指向本缓冲区，则让其指向下一缓冲区。
        if (free_list == bh)
                free_list = bh->b_next_free;
}

//// 将缓冲块插入空闲链表尾部，同时放入 hash 队列中。
static inline void insert_into_queues(struct buffer_head * bh)
{
/* put at end of free list */
/* 放在空闲链表末尾处 */
        bh->b_next_free = free_list;
        bh->b_prev_free = free_list->b_prev_free;
        free_list->b_prev_free->b_next_free = bh;
        free_list->b_prev_free = bh;
/* put the buffer in new hash-queue if it has a device */
/* 如果该缓冲块对应一个设备，则将其插入新 hash 队列中 */
// 请注意当 hash 表某项第 1 次插入项时，hash()计算值肯定为 NULL，因此此时第 161 行上
// 得到的 bh->b_next 肯定是 NULL，所以第 163 行上应该在 bh->b_next 不为 NULL 时才能给
// b_prev 赋 bh 值。即第 163 行前应该增加判断“if (bh->b_next)”。该错误到 0.96 版后
// 才被纠正。
        bh->b_prev = NULL;
        bh->b_next = NULL;
        if (!bh->b_dev)
                return;
        bh->b_next = hash(bh->b_dev,bh->b_blocknr);
        hash(bh->b_dev,bh->b_blocknr) = bh;
        bh->b_next->b_prev = bh;          // 此句前应添加“if (bh->b_next)”判断。
}

//// 利用 hash 表在高速缓冲中寻找给定设备和指定块号的缓冲区块。
// 如果找到则返回缓冲区块的指针，否则返回 NULL。
static struct buffer_head * find_buffer(int dev, int block)
```

```
167 {
168         struct buffer_head * tmp;
169 
    // 搜索 hash 表，寻找指定设备号和块号的缓冲块。
170         for (tmp = hash(dev,block) ; tmp != NULL ; tmp = tmp->b_next)
171                 if (tmp->b_dev==dev && tmp->b_blocknr==block)
172                         return tmp;
173         return NULL;
174 }
175 
176 /*
177  * Why like this, I hear you say... The reason is race-conditions.
178  * As we don't lock buffers (unless we are readint them, that is),
179  * something might happen to it while we sleep (ie a read-error
180  * will force it bad). This shouldn't really happen currently, but
181  * the code is ready.
182  */
    /*
     * 代码为什么会是这样子的？我听见你问... 原因是竞争条件。由于我们没有对
     * 缓冲块上锁（除非我们正在读取它们中的数据），那么当我们（进程）睡眠时
     * 缓冲块可能会发生一些问题（例如一个读错误将导致该缓冲块出错）。目前
     * 这种情况实际上是不会发生的，但处理的代码已经准备好了。
     */
    //// 利用 hash 表在高速缓冲区中寻找指定的缓冲块。若找到则对该缓冲块上锁并返回块头指针。
183 struct buffer_head * get_hash_table(int dev, int block)
184 {
185         struct buffer_head * bh;
186 
187         for (;;) {
    // 在高速缓冲中寻找给定设备和指定块的缓冲区块，如果没有找到则返回 NULL，退出。
188                 if (!(bh=find_buffer(dev,block)))
189                         return NULL;
    // 对该缓冲块增加引用计数，并等待该缓冲块解锁（如果已被上锁）。由于经过了睡眠状态，
    // 因此有必要再验证该缓冲块的正确性，并返回缓冲块头指针。
190                 bh->b_count++;
191                 wait_on_buffer(bh);
192                 if (bh->b_dev == dev && bh->b_blocknr == block)
193                         return bh;
    // 如果在睡眠时该缓冲块所属的设备号或块号发生了改变，则撤消对它的引用计数，重新寻找。
194                 bh->b_count--;
195         }
196 }
197 
198 /*
199  * Ok, this is getblk, and it isn't very clear, again to hinder
200  * race-conditions. Most of the code is seldom used, (ie repeating),
201  * so it should be much more efficient than it looks.
202  *
203  * The algoritm is changed: hopefully better, and an elusive bug removed.
204  */
    /*
     * OK，下面是 getblk 函数，该函数的逻辑并不是很清晰，同样也是因为要考虑
     * 竞争条件问题。其中大部分代码很少用到，(例如重复操作语句)，因此它应该
```

```
 * 比看上去的样子有效得多。
 *
 * 算法已经作了改变：希望能更好，而且一个难以琢磨的错误已经去除。
 */
// 下面宏用于同时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重要比锁定标志
// 大。
#define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)
//// 取高速缓冲中指定的缓冲块。
// 检查指定（设备号和块号）的缓冲区是否已经在高速缓冲中。如果指定块已经在高速缓冲中，
// 则返回对应缓冲区头指针退出；如果不在，就需要在高速缓冲中设置一个对应设备号和块号的
// 新项。返回相应缓冲区头指针。
struct buffer_head * getblk(int dev,int block)
{
        struct buffer_head * tmp, * bh;

repeat:
// 搜索 hash 表，如果指定块已经在高速缓冲中，则返回对应缓冲区头指针，退出。
        if (bh = get_hash_table(dev,block))
                return bh;
// 扫描空闲数据块链表，寻找空闲缓冲区。
// 首先让 tmp 指向空闲链表的第一个空闲缓冲区头。
        tmp = free_list;
        do {
// 如果该缓冲区正被使用（引用计数不等于 0），则继续扫描下一项。对于 b_count=0 的块，
// 即高速缓冲中当前没有引用的块不一定就是干净的（b_dirt=0）或没有锁定的（b_lock=0）。
// 因此，我们还是需要继续下面的判断和选择。例如当一个任务改写过一块内容后就释放了，
// 于是该块 b_count = 0，但 b_lock 不等于 0；当一个任务执行 breada()预读几个块时，只要
// ll_rw_block()命令发出后，它就会递减 b_count；但此时实际上硬盘访问操作可能还在进行，
// 因此此时 b_lock=1，但 b_count=0。
                if (tmp->b_count)
                        continue;
// 如果缓冲头指针 bh 为空，或者 tmp 所指缓冲头的标志(修改、锁定)权重小于 bh 头标志的权
// 重，则让 bh 指向 tmp 缓冲块头。 如果该 tmp 缓冲块头表明缓冲块既没有修改也没有锁定标
// 志置位，则说明已为指定设备上的块取得对应的高速缓冲块，则退出循环。否则我们就继续
// 执行本循环，看看能否找到一个 BADNESS()最小的缓冲快。
                if (!bh || BADNESS(tmp)<BADNESS(bh)) {
                        bh = tmp;
                        if (!BADNESS(tmp))
                                break;
                }
/* and repeat until we find something good */ /* 重复操作直到找到适合的缓冲块 */
        } while ((tmp = tmp->b_next_free) != free_list);
// 如果循环检查发现所有缓冲块都正在被使用（所有缓冲块的头部引用计数都>0）中，则睡眠
// 等待有空闲缓冲块可用。当有空闲缓冲块可用时本进程会被明确地唤醒。然后我们就跳转到
// 函数开始处重新查找空闲缓冲块。
        if (!bh) {
                sleep_on(&buffer_wait);
                goto repeat;                    // 跳转至 210 行。
        }
// 执行到这里，说明我们已经找到了一个比较适合的空闲缓冲块了。于是先等待该缓冲区解锁
//（如果已被上锁的话）。如果在我们睡眠阶段该缓冲区又被其他任务使用的话，只好重复上述
// 寻找过程。
        wait_on_buffer(bh);
```

```
229         if (bh->b_count)                        // 又被占用？？
230                 goto repeat;
    // 如果该缓冲区已被修改，则将数据写盘，并再次等待缓冲区解锁。同样地，若该缓冲区又被
    // 其他任务使用的话，只好再重复上述寻找过程。
231         while (bh->b_dirt) {
232                 sync_dev(bh->b_dev);
233                 wait_on_buffer(bh);
234                 if (bh->b_count)                // 又被占用？？
235                         goto repeat;
236         }
237 /* NOTE!! While we slept waiting for this block, somebody else might */
238 /* 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 */
     * 加入进高速缓冲中，所以我们也要对此进行检查。*/
    // 在高速缓冲 hash 表中检查指定设备和块的缓冲块是否乘我们睡眠之即已经被加入进去。如果
    // 是的话，就再次重复上述寻找过程。
239         if (find_buffer(dev,block))
240                 goto repeat;
241 /* OK, FINALLY we know that this buffer is the only one of it's kind, */
242 /* and that it's unused (b_count=0), unlocked (b_lock=0), and clean */
    /* OK，最终我们知道该缓冲块是指定参数的唯一一块，而且目前还没有被占用 */
    /* (b_count=0)，也未被上锁(b_lock=0)，并且是干净的（未被修改的）*/
    // 于是让我们占用此缓冲块。置引用计数为 1，复位修改标志和有效(更新)标志。
243         bh->b_count=1;
244         bh->b_dirt=0;
245         bh->b_uptodate=0;
    // 从 hash 队列和空闲块链表中移出该缓冲区头，让该缓冲区用于指定设备和其上的指定块。
    // 然后根据此新的设备号和块号重新插入空闲链表和 hash 队列新位置处。并最终返回缓冲
    // 头指针。
246         remove_from_queues(bh);
247         bh->b_dev=dev;
248         bh->b_blocknr=block;
249         insert_into_queues(bh);
250         return bh;
251 }
252
    //// 释放指定缓冲块。
    // 等待该缓冲块解锁。然后引用计数递减 1，并明确地唤醒等待空闲缓冲块的进程。
253 void brelse(struct buffer_head * buf)
254 {
255         if (!buf)                       // 如果缓冲头指针无效则返回。
256                 return;
257         wait_on_buffer(buf);
258         if (!(buf->b_count--))
259                 panic("Trying to free free buffer");
260         wake_up(&buffer_wait);
261 }
262
263 /*
264  * bread() reads a specified block and returns the buffer that contains
265  * it. It returns NULL if the block was unreadable.
266  */
    /*
```

```
 * 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。
 */
//// 从设备上读取数据块。
// 该函数根据指定的设备号 dev 和数据块号 block，首先在高速缓冲区中申请一块缓冲块。
// 如果该缓冲块中已经包含有有效的数据就直接返回该缓冲块指针，否则就从设备中读取
// 指定的数据块到该缓冲块中并返回缓冲块指针。
struct buffer_head * bread(int dev,int block)
{
        struct buffer_head * bh;

// 在高速缓冲区中申请一块缓冲块。如果返回值是 NULL，则表示内核出错，停机。然后我们
// 判断其中是否已有可用数据。 如果该缓冲块中数据是有效的（已更新的）可以直接使用，
// 则返回。
        if (!(bh=getblk(dev,block)))
                panic("bread: getblk returned NULL\n");
        if (bh->b_uptodate)
                return bh;
// 否则我们就调用底层块设备读写 ll_rw_block()函数，产生读设备块请求。然后等待指定
// 数据块被读入，并等待缓冲区解锁。在睡眠醒来之后，如果该缓冲区已更新，则返回缓冲
// 区头指针，退出。否则表明读设备操作失败，于是释放该缓冲区，返回 NULL，退出。
        ll_rw_block(READ,bh);
        wait_on_buffer(bh);
        if (bh->b_uptodate)
                return bh;
        brelse(bh);
        return NULL;
}

//// 复制内存块。
// 从 from 地址复制一块（1024 字节）数据到 to 位置。
#define COPYBLK(from,to) \
__asm__("cld\n\t" \
        "rep\n\t" \
        "movsl\n\t" \
        ::"c" (BLOCK_SIZE/4),"S" (from),"D" (to) \
        :"cx","di","si")

/*
 * bread_page reads four buffers into memory at the desired address. It's
 * a function of its own, as there is some speed to be got by reading them
 * all at the same time, not waiting for one to be read, and then another
 * etc.
 */
/*
 * bread_page 一次读四个缓冲块数据读到内存指定的地址处。它是一个完整的函数，
 * 因为同时读取四块可以获得速度上的好处，不用等着读一块，再读一块了。
 */
//// 读设备上一个页面（4 个缓冲块）的内容到指定内存地址处。
// 参数 address 是保存页面数据的地址；dev 是指定的设备号；b[4]是含有 4 个设备数据块号
// 的数组。该函数仅用于 mm/memory.c 文件的 do_no_page()函数中（第 386 行）。
void bread_page(unsigned long address,int dev,int b[4])
{
```

```
298         struct buffer_head * bh[4];
299         int i;
300
    // 该函数循环执行 4 次，根据放在数组 b[]中的 4 个块号从设备 dev 中读取一页内容放到指定
    // 内存位置 address 处。 对于参数 b[i]给出的有效块号，函数首先从高速缓冲中取指定设备
    // 和块号的缓冲块。如果缓冲块中数据无效（未更新）则产生读设备请求从设备上读取相应数
    // 据块。对于 b[i]无效的块号则不用去理它了。因此本函数其实可以根据指定的 b[]中的块号
    // 随意读取 1—4 个数据块。
301         for (i=0 ; i<4 ; i++)
302                 if (b[i]) {                     // 若块号有效。
303                         if (bh[i] = getblk(dev,b[i]))
304                                 if (!bh[i]->b_uptodate)
305                                         ll_rw_block(READ,bh[i]);
306                 } else
307                         bh[i] = NULL;
    // 随后将 4 个缓冲块上的内容顺序复制到指定地址处。在进行复制（使用）缓冲块之前我们
    // 先要睡眠等待缓冲块解锁（若被上锁的话）。另外，因为可能睡眠过了，所以我们还需要
    // 在复制之前再检查一下缓冲块中的数据是否是有效的。复制完后我们还需要释放缓冲块。
308         for (i=0 ; i<4 ; i++,address += BLOCK_SIZE)
309                 if (bh[i]) {
310                         wait_on_buffer(bh[i]);    // 等待缓冲块解锁(若被上锁的话)。
311                         if (bh[i]->b_uptodate)    // 若缓冲块中数据有效的话则复制。
312                                 COPYBLK((unsigned long) bh[i]->b_data,address);
313                         brelse(bh[i]);            // 释放该缓冲区。
314                 }
315 }
316
317 /*
318  * Ok, breada can be used as bread, but additionally to mark other
319  * blocks for reading as well. End the argument list with a negative
320  * number.
321  */
    /*
     * OK，breada 可以象 bread 一样使用，但会另外预读一些块。该函数参数列表
     * 需要使用一个负数来表明参数列表的结束。
     */
    //// 从指定设备读取指定的一些块。
    // 函数参数个数可变，是一系列指定的块号。成功时返回第 1 块的缓冲块头指针，否则返回
    // NULL。
322 struct buffer_head * breada(int dev,int first, ...)
323 {
324         va_list args;
325         struct buffer_head * bh, *tmp;
326
    // 首先取可变参数表中第 1 个参数（块号）。接着从高速缓冲区中取指定设备和块号的缓冲
    // 块。如果该缓冲块数据无效（更新标志未置位），则发出读设备数据块请求。
327         va_start(args,first);
328         if (!(bh=getblk(dev,first)))
329                 panic("bread: getblk returned NULL\n");
330         if (!bh->b_uptodate)
331                 ll_rw_block(READ,bh);
    // 然后顺序取可变参数表中其他预读块号，并作与上面同样处理，但不引用。注意，336 行上
    // 有一个 bug。其中的 bh 应该是 tmp。这个 bug 直到在 0.96 版的内核代码中才被纠正过来。
```

```
    // 另外，因为这里是预读随后的数据块，只需读进高速缓冲区但并不马上就使用，所以第 337
    // 行语句需要将其引用计数递减释放掉该块（因为 getblk()函数会增加缓冲块引用计数值）。
332         while ((first=va_arg(args,int))>=0) {
333                 tmp=getblk(dev,first);
334                 if (tmp) {
335                         if (!tmp->b_uptodate)
336                                 ll_rw_block(READA,bh);     // bh 应该是 tmp。
337                         tmp->b_count--;                    // 暂时释放掉该预读块。
338                 }
339         }
    // 此时可变参数表中所有参数处理完毕。于是等待第 1 个缓冲区解锁（如果已被上锁）。在等
    // 待退出之后如果缓冲区中数据仍然有效，则返回缓冲区头指针退出。否则释放该缓冲区返回
    // NULL，退出。
340         va_end(args);
341         wait_on_buffer(bh);
342         if (bh->b_uptodate)
343                 return bh;
344         brelse(bh);
345         return (NULL);
346 }
347
    //// 缓冲区初始化函数。
    // 参数 buffer_end 是缓冲区内存末端。对于具有 16MB 内存的系统，缓冲区末端被设置为 4MB。
    // 对于有 8MB 内存的系统，缓冲区末端被设置为 2MB。该函数从缓冲区开始位置 start_buffer
    // 处和缓冲区末端 buffer_end 处分别同时设置（初始化）缓冲块头结构和对应的数据块。直到
    // 缓冲区中所有内存被分配完毕。参见程序列表前面的示意图。
348 void buffer_init(long buffer_end)
349 {
350         struct buffer_head * h = start_buffer;
351         void * b;
352         int i;
353
    // 首先根据参数提供的缓冲区高端位置确定实际缓冲区高端位置 b。如果缓冲区高端等于 1Mb，
    // 则因为从 640KB - 1MB 被显示内存和 BIOS 占用，所以实际可用缓冲区内存高端位置应该是
    // 640KB。否则缓冲区内存高端一定大于 1MB。
354         if (buffer_end == 1<<20)
355                 b = (void *) (640*1024);
356         else
357                 b = (void *) buffer_end;
    // 这段代码用于初始化缓冲区，建立空闲缓冲块循环链表，并获取系统中缓冲块数目。操作的
    // 过程是从缓冲区高端开始划分 1KB 大小的缓冲块，与此同时在缓冲区低端建立描述该缓冲块
    // 的结构 buffer_head，并将这些 buffer_head 组成双向链表。
    // h 是指向缓冲头结构的指针，而 h+1 是指向内存地址连续的下一个缓冲头地址，也可以说是
    // 指向 h 缓冲头的末端外。为了保证有足够长度的内存来存储一个缓冲头结构，需要 b 所指向
    // 的内存块地址 >= h 缓冲头的末端，即要求 >= h+1。
358         while ( (b -= BLOCK_SIZE) >= ((void *) (h+1)) ) {
359                 h->b_dev = 0;                   // 使用该缓冲块的设备号。
360                 h->b_dirt = 0;                  // 脏标志，即缓冲块修改标志。
361                 h->b_count = 0;                 // 缓冲块引用计数。
362                 h->b_lock = 0;                  // 缓冲块锁定标志。
363                 h->b_uptodate = 0;              // 缓冲块更新标志（或称数据有效标志）。
364                 h->b_wait = NULL;               // 指向等待该缓冲块解锁的进程。
365                 h->b_next = NULL;               // 指向具有相同 hash 值的下一个缓冲头。
```

```
366                 h->b_prev = NULL;           // 指向具有相同 hash 值的前一个缓冲头。
367                 h->b_data = (char *) b;     // 指向对应缓冲块数据块（1024 字节）。
368                 h->b_prev_free = h-1;       // 指向链表中前一项。
369                 h->b_next_free = h+1;       // 指向链表中下一项。
370                 h++;                           // h 指向下一新缓冲头位置。
371                 NR_BUFFERS++;                  // 缓冲区块数累加。
372                 if (b == (void *) 0x100000)    // 若 b 递减到等于 1MB，则跳过 384KB，
373                         b = (void *) 0xA0000;  // 让 b 指向地址 0xA0000(640KB)处。
374         }
375         h--;                                   // 让 h 指向最后一个有效缓冲块头。
376         free_list = start_buffer;              // 让空闲链表头指向头一个缓冲块。
377         free_list->b_prev_free = h;  // 链表头的 b_prev_free 指向前一项（即最后一项）。
378         h->b_next_free = free_list;            // h 的下一项指针指向第一项，形成一个环链。
    // 最后初始化 hash 表（哈希表、散列表），置表中所有指针为 NULL。
379         for (i=0;i<NR_HASH;i++)
380                 hash_table[i]=NULL;
381 }
382
```

12.2 程序 12-2　linux/fs/bitmap.c

```
/*
 *  linux/fs/bitmap.c
 *
 *  (C) 1991  Linus Torvalds
 */

/* bitmap.c contains the code that handles the inode and block bitmaps */
/* bitmap.c 程序含有处理 i 节点和磁盘块位图的代码 */
#include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
                           // 这里使用了其中的 memset()函数。
#include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 数据。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。

//// 将指定地址（addr）处的一块 1024 字节内存清零。
// 输入：eax = 0；ecx = 以长字为单位的数据块长度（BLOCK_SIZE/4）；edi = 指定起始地
// 址 addr。
#define clear_block(addr) \
__asm__("cld\n\t" \                    // 清方向位。
        "rep\n\t" \                    // 重复执行存储数据（0）。
        "stosl" \
        ::"a" (0),"c" (BLOCK_SIZE/4),"D" ((long) (addr)):"cx","di")

//// 把指定地址开始的第 nr 个位偏移处的比特位置位（nr 可大于 32！）。返回原比特位值。
// 输入：%0 -eax（返回值）；%1 -eax(0)；%2 -nr，位偏移值；%3 -(addr)，addr 的内容。
// 第 20 行定义了一个局部寄存器变量 res。该变量将被保存在指定的 eax 寄存器中，以便于
// 高效访问和操作。这种定义变量的方法主要用于内嵌汇编程序中。详细说明参见 gcc 手册
// “在指定寄存器中的变量”。整个宏定义是一个语句表达式，该表达式值是最后 res 的值。
// 第 21 行上的 btsl 指令用于测试并设置比特位（Bit Test and Set）。把基地址（%3）和
// 比特位偏移值（%2）所指定的比特位值先保存到进位标志 CF 中，然后设置该比特位为 1。
// 指令 setb 用于根据进位标志 CF 设置操作数（%al）。如果 CF=1 则%al =1，否则%al =0。
#define set_bit(nr,addr) ({\
register int res __asm__("ax"); \
__asm__ __volatile__("btsl %2,%3\n\tsetb %%al": \
"=a" (res):"" (0),"r" (nr),"m" (*(addr))); \
res;})

//// 复位指定地址开始的第 nr 位偏移处的比特位。返回原比特位值的反码。
// 输入：%0 -eax（返回值）；%1 -eax(0)；%2 -nr，位偏移值；%3 -(addr)，addr 的内容。
// 第 27 行上的 btrl 指令用于测试并复位比特位（Bit Test and Reset）。其作用与上面的
// btsl 类似，但是复位指定比特位。指令 setnb 用于根据进位标志 CF 设置操作数（%al）。
// 如果 CF = 1 则%al = 0，否则%al = 1。
#define clear_bit(nr,addr) ({\
register int res __asm__("ax"); \
__asm__ __volatile__("btrl %2,%3\n\tsetnb %%al": \
"=a" (res):"" (0),"r" (nr),"m" (*(addr))); \
res;})

//// 从 addr 开始寻找第 1 个 0 值比特位。
// 输入：%0 - ecx(返回值)；%1 - ecx(0)；%2 - esi(addr)。
```

```
   // 在 addr 指定地址开始的位图中寻找第 1 个是 0 的比特位，并将其距离 addr 的比特位偏移
   // 值返回。addr 是缓冲块数据区的地址，扫描寻找的范围是 1024 字节（8192 比特位）。
31 #define find_first_zero(addr) ({ \
32 int __res; \
33 __asm__("cld\n" \                    // 清方向位。
34         "1:\tlodsl\n\t" \            // 取[esi]➔eax。
35         "notl %%eax\n\t" \           // eax 中每位取反。
36         "bsfl %%eax,%%edx\n\t" \     // 从位 0 扫描 eax 中是 1 的第 1 个位，其偏移值➔edx。
37         "je 2f\n\t" \                // 如果 eax 中全是 0，则向前跳转到标号 2 处(40 行)。
38         "addl %%edx,%%ecx\n\t" \     // 偏移值加入 ecx（ecx 是位图首个 0 值位的偏移值）。
39         "jmp 3f\n" \                 // 向前跳转到标号 3 处（结束）。
40         "2:\taddl $32,%%ecx\n\t" \   // 未找到 0 值位，则将 ecx 加 1 个长字的位偏移量 32。
41         "cmpl $8192,%%ecx\n\t" \     // 已经扫描了 8192 比特位（1024 字节）了吗？
42         "jl 1b\n" \                  // 若还没有扫描完 1 块数据，则向前跳转到标号 1 处。
43         "3:" \                       // 结束。此时 ecx 中是位偏移量。
44         :"=c" (__res):"c" (0),"S" (addr):"ax","dx","si"); \
45 __res;})
46
   //// 释放设备 dev 上数据区中的逻辑块 block。
   // 复位指定逻辑块 block 对应的逻辑块位图比特位。成功则返回 1，否则返回 0。
   // 参数：dev 是设备号，block 是逻辑块号（盘块号）。
47 int free_block(int dev, int block)
48 {
49         struct super_block * sb;
50         struct buffer_head * bh;
51
   // 首先取设备 dev 上文件系统的超级块信息，根据其中数据区开始逻辑块号和文件系统中逻辑
   // 块总数信息判断参数 block 的有效性。如果指定设备超级块不存在，则出错停机。若逻辑块
   // 号小于盘上数据区第 1 个逻辑块的块号或者大于设备上总逻辑块数，也出错停机。
52         if (!(sb = get_super(dev)))                    // fs/super.c，第 56 行。
53                 panic("trying to free block on nonexistent device");
54         if (block < sb->s_firstdatazone || block >= sb->s_nzones)
55                 panic("trying to free block not in datazone");
56         bh = get_hash_table(dev,block);
   // 然后从 hash 表中寻找该块数据。若找到了则判断其有效性，并清已修改和更新标志，释放
   // 该数据块。该段代码的主要用途是如果该逻辑块目前存在于高速缓冲区中，就释放对应的缓
   // 冲块。
57         if (bh) {
58                 if (bh->b_count > 1) {      // 如果引用次数大于 1，则调用 brelse()，
59                         brelse(bh);         // b_count--后即退出，该块还有人用。
60                         return 0;
61                 }
62                 bh->b_dirt=0;               // 否则复位已修改和已更新标志。
63                 bh->b_uptodate=0;
64                 if (bh->b_count)            // 若此时 b_count 为 1，则调用 brelse()释放之。
65                         brelse(bh);
66         }
   // 接着我们复位 block 在逻辑块位图中的比特位（置 0）。先计算 block 在数据区开始算起的
   // 数据逻辑块号（从 1 开始计数）。然后对逻辑块(区块)位图进行操作，复位对应的比特位。
   // 如果对应比特位原来就是 0，则出错停机。由于 1 个缓冲块有 1024 字节，即 8192 比特位，
   // 因此 block/8192 即可计算出指定块 block 在逻辑位图中的哪个块上。而 block&8191 可
   // 以得到 block 在逻辑块位图当前块中的比特偏移位置。
67         block -= sb->s_firstdatazone - 1 ; // 即 block = block - ( s_firstdatazone -1);
```

```
68          if (clear_bit(block&8191,sb->s_zmap[block/8192]->b_data)) {
69                  printk("block (%04x:%d) ",dev,block+sb->s_firstdatazone-1);
70                  printk("free_block: bit already cleared\n");
71          }
    // 最后置相应逻辑块位图所在缓冲区已修改标志。
72          sb->s_zmap[block/8192]->b_dirt = 1;
73          return 1;
74  }
75
    ////向设备申请一个逻辑块（盘块，区块）。
    // 函数首先取得设备的超级块，并在超级块中的逻辑块位图中寻找第一个 0 值比特位（代表
    // 一个空闲逻辑块）。然后置位对应逻辑块在逻辑块位图中的比特位。接着为该逻辑块在缓
    // 冲区中取得一块对应缓冲块。最后将该缓冲块清零，并设置其已更新标志和已修改标志。
    // 并返回逻辑块号。函数执行成功则返回逻辑块号（盘块号），否则返回 0。
76  int new_block(int dev)
77  {
78          struct buffer_head * bh;
79          struct super_block * sb;
80          int i,j;
81
    // 首先获取设备 dev 的超级块。如果指定设备的超级块不存在，则出错停机。然后扫描文件
    // 系统的 8 块逻辑块位图，寻找首个 0 值比特位，以寻找空闲逻辑块，获取放置该逻辑块的
    // 块号。 如果全部扫描完 8 块逻辑块位图的所有比特位（i >= 8 或 j >= 8192）还没找到
    // 0 值比特位或者位图所在的缓冲块指针无效(bh = NULL)则 返回 0 退出（没有空闲逻辑块）。
82          if (!(sb = get_super(dev)))
83                  panic("trying to get new block from nonexistant device");
84          j = 8192;
85          for (i=0 ; i<8 ; i++)
86                  if (bh=sb->s_zmap[i])
87                          if ((j=find_first_zero(bh->b_data))<8192)
88                                  break;
89          if (i>=8 || !bh || j>=8192)
90                  return 0;
    // 接着设置找到的新逻辑块 j 对应逻辑块位图中的比特位。若对应比特位已经置位，则出错
    // 停机。否则置存放位图的对应缓冲区块已修改标志。因为逻辑块位图仅表示盘上数据区中
    // 逻辑块的占用情况，即逻辑块位图中比特位偏移值表示从数据区开始处算起的块号，因此
    // 这里需要加上数据区第 1 个逻辑块的块号，把 j 转换成逻辑块号。此时如果新逻辑块大于
    // 该设备上的总逻辑块数，则说明指定逻辑块在对应设备上不存在。申请失败，返回 0 退出。
91          if (set_bit(j,bh->b_data))
92                  panic("new_block: bit already set");
93          bh->b_dirt = 1;
94          j += i*8192 + sb->s_firstdatazone-1;
95          if (j >= sb->s_nzones)
96                  return 0;
    // 然后在高速缓冲区中为该设备上指定的逻辑块号取得一个缓冲块，并返回缓冲块头指针。
    // 因为刚取得的逻辑块其引用次数一定为 1（getblk()中会设置），因此若不为 1 则停机。
    // 最后将新逻辑块清零，并设置其已更新标志和已修改标志。然后释放对应缓冲块，返回
    // 逻辑块号。
97          if (!(bh=getblk(dev,j)))
98                  panic("new_block: cannot get block");
99          if (bh->b_count != 1)
100                 panic("new block: count is != 1");
101         clear_block(bh->b_data);
```

```
102         bh->b_uptodate = 1;
103         bh->b_dirt = 1;
104         brelse(bh);
105         return j;
106 }
107
    //// 释放指定的 i 节点。
    // 该函数首先判断参数给出的 i 节点号的有效性和可释放性。若 i 节点仍然在使用中则不能
    // 被释放。然后利用超级块信息对 i 节点位图进行操作，复位 i 节点号对应的 i 节点位图中
    // 比特位，并清空 i 节点结构。
108 void free_inode(struct m_inode * inode)
109 {
110         struct super_block * sb;
111         struct buffer_head * bh;
112
    // 首先判断参数给出的需要释放的 i 节点有效性或合法性。如果 i 节点指针=NULL，则退出。
    // 如果 i 节点上的设备号字段为 0，说明该节点没有使用。于是用 0 清空对应 i 节点所占内存
    // 区并返回。memset()定义在 include/string.h 第 395 行开始处。这里表示用 0 填写 inode
    // 指针指定处、长度是 sizeof(*inode) 的内存块。
113         if (!inode)
114                 return;
115         if (!inode->i_dev) {
116                 memset(inode,0,sizeof(*inode));
117                 return;
118         }
    // 如果此 i 节点还有其他程序引用，则不能释放，说明内核有问题，停机。如果文件连接数
    // 不为 0，则表示还有其他文件目录项在使用该节点，因此也不应释放，而应该放回等。
119         if (inode->i_count>1) {
120                 printk("trying to free inode with count=%d\n",inode->i_count);
121                 panic("free_inode");
122         }
123         if (inode->i_nlinks)
124                 panic("trying to free inode with links");
    // 在判断完 i 节点的合理性之后，我们开始利用其超级块信息对其中的 i 节点位图进行操作。
    // 首先取 i 节点所在设备的超级块，测试设备是否存在。然后判断 i 节点号的范围是否正确，
    // 如果 i 节点号等于 0 或 大于该设备上 i 节点总数，则出错（0 号 i 节点保留没有使用）。
    // 如果该 i 节点对应的节点位图不存在，则出错。因为一个缓冲块的 i 节点位图有 8192 比
    // 特位。因此 i_num>>13（即 i_num/8192）可以得到当前 i 节点号所在的 s_imap[]项，即所
    // 在盘块。
125         if (!(sb = get_super(inode->i_dev)))
126                 panic("trying to free inode on nonexistent device");
127         if (inode->i_num < 1 || inode->i_num > sb->s_ninodes)
128                 panic("trying to free inode 0 or nonexistant inode");
129         if (!(bh=sb->s_imap[inode->i_num>>13]))
130                 panic("nonexistent imap in superblock");
    // 现在我们复位 i 节点对应的节点位图中的比特位。如果该比特位已经等于 0，则显示出错
    // 警告信息。最后置 i 节点位图所在缓冲区已修改标志，并清空该 i 节点结构所占内存区。
131         if (clear_bit(inode->i_num&8191,bh->b_data))
132                 printk("free_inode: bit already cleared.\n\r");
133         bh->b_dirt = 1;
134         memset(inode,0,sizeof(*inode));
135 }
136
```

```
    //// 为设备 dev 建立一个新 i 节点。初始化并返回该新 i 节点的指针。
    // 在内存 i 节点表中获取一个空闲 i 节点表项，并从 i 节点位图中找一个空闲 i 节点。
137 struct m_inode * new_inode(int dev)
138 {
139         struct m_inode * inode;
140         struct super_block * sb;
141         struct buffer_head * bh;
142         int i,j;
143
    // 首先从内存 i 节点表（inode_table）中获取一个空闲 i 节点项，并读取指定设备的超级块
    // 结构。 然后扫描超级块中 8 块 i 节点位图，寻找首个 0 比特位，寻找空闲节点，获取放置
    // 该 i 节点的节点号。如果全部扫描完还没找到，或者位图所在的缓冲块无效（bh = NULL），
    // 则放回先前申请的 i 节点表中的 i 节点，并返回空指针退出（没有空闲 i 节点）。
144         if (!(inode=get_empty_inode()))             // fs/inode.c，第 197 行。
145                 return NULL;
146         if (!(sb = get_super(dev)))                 // fs/super.c，第 56 行。
147                 panic("new_inode with unknown device");
148         j = 8192;
149         for (i=0 ; i<8 ; i++)
150                 if (bh=sb->s_imap[i])
151                         if ((j=find_first_zero(bh->b_data))<8192)
152                                 break;
153         if (!bh || j >= 8192 || j+i*8192 > sb->s_ninodes) {
154                 iput(inode);
155                 return NULL;
156         }
    // 现在我们已经找到了还未使用的 i 节点号 j。于是置位 i 节点 j 对应的 i 节点位图相应比
    // 特位（如果已经置位，则出错）。然后置 i 节点位图所在缓冲块已修改标志。最后初始化
    // 该 i 节点结构（i_ctime 是 i 节点内容改变时间）。
157         if (set_bit(j,bh->b_data))
158                 panic("new_inode: bit already set");
159         bh->b_dirt = 1;
160         inode->i_count=1;                       // 引用计数。
161         inode->i_nlinks=1;                      // 文件目录项链接数。
162         inode->i_dev=dev;                       // i 节点所在的设备号。
163         inode->i_uid=current->euid;             // i 节点所属用户 id。
164         inode->i_gid=current->egid;             // 组 id。
165         inode->i_dirt=1;                        // 已修改标志置位。
166         inode->i_num = j + i*8192;              // 对应设备中的 i 节点号。
167         inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;  // 设置时间。
168         return inode;                           // 返回该 i 节点指针。
169 }
170
```

12.3 程序 12-3 linux/fs/truncate.c

```
 1 /*
 2  *  linux/fs/truncate.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
 8
 9 #include <sys/stat.h>     // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。
10
   //// 释放所有一次间接块。（内部函数）
   // 参数 dev 是文件系统所在设备的设备号；block 是逻辑块号。成功则返回 1，否则返回 0。
11 static int free_ind(int dev,int block)
12 {
13         struct buffer_head * bh;
14         unsigned short * p;
15         int i;
16         int block_busy;                              // 有逻辑块没有被释放的标志。
17
   // 首先判断参数的有效性。如果逻辑块号为 0，则返回。然后读取一次间接块，并释放其上表
   // 明使用的所有逻辑块，然后释放该一次间接块的缓冲块。 函数 free_block()用于释放设备
   // 上指定逻辑块号的磁盘块（fs/bitmap.c 第 47 行）。
18         if (!block)
19                 return 1;
20         block_busy = 0;
21         if (bh=bread(dev,block)) {
22                 p = (unsigned short *) bh->b_data;  // 指向缓冲块数据区。
23                 for (i=0;i<512;i++,p++)             // 每个逻辑块上可有 512 个块号。
24                         if (*p)
25                                 if (free_block(dev,*p)) {  // 释放指定的设备逻辑块。
26                                         *p = 0;            // 清零。
27                                         bh->b_dirt = 1;    // 设置已修改标志。
28                                 } else
29                                         block_busy = 1;    // 设置逻辑块没有释放标志。
30                 brelse(bh);                              // 然后释放间接块占用的缓冲块。
31         }
   // 最后释放设备上的一次间接块。但如果其中有逻辑块没有被释放，则返回 0（失败）。
32         if (block_busy)
33                 return 0;
34         else
35                 return free_block(dev,block);      // 成功则返回 1，否则返回 0。
36 }
37
   //// 释放所有二次间接块。
   // 参数 dev 是文件系统所在设备的设备号；block 是逻辑块号。
38 static int free_dind(int dev,int block)
39 {
40         struct buffer_head * bh;
41         unsigned short * p;
```

```
42          int i;
43          int block_busy;                             // 有逻辑块没有被释放的标志。
44
    // 首先判断参数的有效性。如果逻辑块号为0，则返回。然后读取二次间接块的一级块，并释
    // 放其上表明使用的所有逻辑块，然后释放该一级块的缓冲块。
45          if (!block)
46                  return 1;
47          block_busy = 0;
48          if (bh=bread(dev,block)) {
49                  p = (unsigned short *) bh->b_data;  // 指向缓冲块数据区。
50                  for (i=0;i<512;i++,p++)             // 每个逻辑块上可连接512个二级块。
51                          if (*p)
52                                  if (free_ind(dev,*p)) {   // 释放所有一次间接块。
53                                          *p = 0;           // 清零。
54                                          bh->b_dirt = 1;   // 设置已修改标志。
55                                  } else
56                                          block_busy = 1;   // 设置逻辑块没有释放标志。
57                  brelse(bh);                         // 释放二次间接块占用的缓冲块。
58          }
    // 最后释放设备上的二次间接块。但如果其中有逻辑块没有被释放，则返回0（失败）。
59          if (block_busy)
60                  return 0;
61          else
62                  return free_block(dev,block);
63  }
64
    //// 截断文件数据函数。
    // 将节点对应的文件长度截为0，并释放占用的设备空间。
65  void truncate(struct m_inode * inode)
66  {
67          int i;
68          int block_busy;                             // 有逻辑块没有被释放的标志。
69
    // 首先判断指定i节点有效性。如果不是常规文件、目录文件或链接项，则返回。
70          if (!(S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode) ||
71               S_ISLNK(inode->i_mode)))
72                  return;
    // 然后释放i节点的7个直接逻辑块，并将这7个逻辑块项全置零。函数free_block()用于
    // 释放设备上指定逻辑块号的磁盘块（fs/bitmap.c 第47行）。若有逻辑块忙而没有被释放
    // 则置块忙标志block_busy。
73  repeat:
74          block_busy = 0;
75          for (i=0;i<7;i++)
76                  if (inode->i_zone[i]) {             // 如果块号不为0，则释放之。
77                          if (free_block(inode->i_dev,inode->i_zone[i]))
78                                  inode->i_zone[i]=0;   // 块指针置0。
79                          else
80                                  block_busy = 1;       // 若没有释放掉则置标志。
81                  }
82          if (free_ind(inode->i_dev,inode->i_zone[7]))  // 释放所有一次间接块。
83                  inode->i_zone[7] = 0;                 // 块指针置0。
84          else
85                  block_busy = 1;                       // 若没有释放掉则置标志。
```

```
86          if (free_dind(inode->i_dev,inode->i_zone[8])) // 释放所有二次间接块。
87                  inode->i_zone[8] = 0;                 // 块指针置 0。
88          else
89                  block_busy = 1;                       // 若没有释放掉则置标志。
    // 此后设置 i 节点已修改标志，并且如果还有逻辑块由于“忙”而没有被释放，则把当前进程
    // 运行时间片置 0，以让当前进程先被切换去运行其他进程，稍等一会再重新执行释放操作。
90          inode->i_dirt = 1;
91          if (block_busy) {
92                  current->counter = 0;                  // 当前进程时间片置 0。
93                  schedule();
94                  goto repeat;
95          }
96          inode->i_size = 0;                             // 文件大小置零。
    // 最后重新置文件修改时间和 i 节点改变时间为当前时间。宏 CURRENT_TIME 定义在头文件
    // include/linux/sched.h 第 142 行处，定义为(startup_time + jiffies/HZ)。用于取得从
    // 1970:0:0:0 开始到现在为止经过的秒数。
97          inode->i_mtime = inode->i_ctime = CURRENT_TIME;
98  }
99
100
```

12.4 程序 12-4 linux/fs/inode.c

```
 1 /*
 2  *  linux/fs/inode.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
 8 #include <sys/stat.h>      // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。
 9
10 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据，
                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <linux/mm.h>     // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
13 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
14
   // 设备数据块总数指针数组。每个指针项指向指定主设备号的总块数数组 hd_sizes[]。该总
   // 块数数组每一项对应子设备号确定的一个子设备上所拥有的数据块总数（1 块大小 = 1KB）。
15 extern int *blk_size[];
16
17 struct m_inode inode_table[NR_INODE]={{0,},};     // 内存中 i 节点表（NR_INODE=32 项）。
18
19 static void read_inode(struct m_inode * inode);  // 读指定 i 节点号的 i 节点信息，297 行。
20 static void write_inode(struct m_inode * inode); // 写 i 节点信息到高速缓冲中，324 行。
21
   //// 等待指定的 i 节点可用。
   // 如果 i 节点已被锁定，则将当前任务置为不可中断的等待状态，并添加到该 i 节点的等待队
   // 列 i_wait 中。直到该 i 节点解锁并明确地唤醒本任务。
22 static inline void wait_on_inode(struct m_inode * inode)
23 {
24         cli();
25         while (inode->i_lock)
26                 sleep_on(&inode->i_wait);          // kernel/sched.c，第 199 行。
27         sti();
28 }
29
   //// 对 i 节点上锁（锁定指定的 i 节点）。
   // 如果 i 节点已被锁定，则将当前任务置为不可中断的等待状态，并添加到该 i 节点的等待队
   // 列 i_wait 中。直到该 i 节点解锁并明确地唤醒本任务。然后对其上锁。
30 static inline void lock_inode(struct m_inode * inode)
31 {
32         cli();
33         while (inode->i_lock)
34                 sleep_on(&inode->i_wait);
35         inode->i_lock=1;                           // 置锁定标志。
36         sti();
37 }
38
   //// 对指定的 i 节点解锁。
   // 复位 i 节点的锁定标志，并明确地唤醒等待在此 i 节点等待队列 i_wait 上的所有进程。
```

```
39 static inline void unlock_inode(struct m_inode * inode)
40 {
41         inode->i_lock=0;
42         wake_up(&inode->i_wait);                // kernel/sched.c，第 204 行。
43 }
44
   //// 释放设备 dev 在内存 i 节点表中的所有 i 节点。
   // 扫描内存中的 i 节点表数组，如果是指定设备使用的 i 节点就释放之。
45 void invalidate_inodes(int dev)
46 {
47         int i;
48         struct m_inode * inode;
49
   // 首先让指针指向内存 i 节点表数组首项。然后扫描 i 节点表指针数组中的所有 i 节点。针对
   // 其中每个 i 节点，先等待该 i 节点解锁可用（若目前正被上锁的话），再判断是否属于指定
   // 设备的 i 节点。如果是指定设备的 i 节点，则看看它是否还被使用着，即其引用计数是否不
   // 为 0。若是则显示警告信息。然后释放之，即把 i 节点的设备号字段 i_dev 置 0。第 50 行上
   // 的指针赋值 "0+inode_table " 等同于 "inode_table"、"&inode_table[0] "。不过这样写
   // 可能更明了一些。
50         inode = 0+inode_table;                  // 指向 i 节点表指针数组首项。
51         for(i=0 ; i<NR_INODE ; i++,inode++) {
52                 wait_on_inode(inode);           // 等待该 i 节点可用（解锁）。
53                 if (inode->i_dev == dev) {
54                         if (inode->i_count)     // 若其引用数不为 0，则显示出错警告。
55                                 printk("inode in use on removed disk\n\r");
56                         inode->i_dev = inode->i_dirt = 0; // 释放 i 节点(置设备号为 0)。
57                 }
58         }
59 }
60
   //// 同步所有 i 节点。
   // 把内存 i 节点表中所有 i 节点与设备上 i 节点作同步操作。
61 void sync_inodes(void)
62 {
63         int i;
64         struct m_inode * inode;
65
   // 首先让内存 i 节点类型的指针指向 i 节点表首项，然后扫描整个 i 节点表中的节点。针对
   // 其中每个 i 节点，先等待该 i 节点解锁可用（若目前正被上锁的话），然后判断该 i 节点
   // 是否已被修改并且不是管道节点。若是这种情况则将该 i 节点写入高速缓冲区中。缓冲区
   // 管理程序 buffer.c 会在适当时机将它们写入盘中。
66         inode = 0+inode_table;                  // 让指针首先指向 i 节点表指针数组首项。
67         for(i=0 ; i<NR_INODE ; i++,inode++) {   // 扫描 i 节点表指针数组。
68                 wait_on_inode(inode);           // 等待该 i 节点可用（解锁）。
69                 if (inode->i_dirt && !inode->i_pipe)  // 若 i 节点已修改且不是管道节点，
70                         write_inode(inode);             // 则写盘（实际是写入缓冲区中）。
71         }
72 }
73
   //// 文件数据块映射到盘块的处理操作。（block 位图处理函数，bmap - block map）
   // 参数：inode - 文件的 i 节点指针；block - 文件中的数据块号；create - 创建块标志。
   // 该函数把指定的文件数据块 block 对应到设备上逻辑块上，并返回逻辑块号。如果创建标志
   // 置位，则在设备上对应逻辑块不存在时就申请新磁盘块，返回文件数据块 block 对应在设备
```

```
    // 上的逻辑块号（盘块号）。
74  static int _bmap(struct m_inode * inode,int block,int create)
75  {
76          struct buffer_head * bh;
77          int i;
78
    // 首先判断参数文件数据块号 block 的有效性。如果块号小于 0，则停机。如果块号大于直接
    // 块数 + 间接块数 + 二次间接块数，超出文件系统表示范围，则停机。
79          if (block<0)
80                  panic("_bmap: block<0");
81          if (block >= 7+512+512*512)
82                  panic("_bmap: block>big");
    // 然后根据文件块号的大小值和是否设置了创建标志分别进行处理。如果该块号小于 7，则使
    // 用直接块表示。如果创建标志置位，并且 i 节点中对应该块的逻辑块（区段）字段为 0，则
    // 向相应设备申请一磁盘块（逻辑块），并且将盘上逻辑块号（盘块号）填入逻辑块字段中。
    // 然后设置 i 节点改变时间，置 i 节点已修改标志。 最后返回逻辑块号。 函数 new_block()
    // 定义在 bitmap.c 程序中第 76 行开始处。
83          if (block<7) {
84                  if (create && !inode->i_zone[block])
85                          if (inode->i_zone[block]=new_block(inode->i_dev)) {
86                                  inode->i_ctime=CURRENT_TIME; // ctime - Change time。
87                                  inode->i_dirt=1;             // 设置已修改标志。
88                          }
89                  return inode->i_zone[block];
90          }
    // 如果该块号>=7，且小于 7+512，则说明使用的是一次间接块。下面对一次间接块进行处理。
    // 如果是创建，并且该 i 节点中对应间接块字段 i_zone[7]是 0，表明文件是首次使用间接块，
    // 则需申请一磁盘块用于存放间接块信息，并将此实际磁盘块号填入间接块字段中。 然后设
    // 置 i 节点已修改标志和修改时间。 如果创建时申请磁盘块失败，则此时 i 节点间接块字段
    // i_zone[7]为 0，则返回 0。或者不是创建，但 i_zone[7]原来就为 0，表明 i 节点中没有间
    // 接块，于是映射磁盘块失败，返回 0 退出。
91          block -= 7;
92          if (block<512) {
93                  if (create && !inode->i_zone[7])
94                          if (inode->i_zone[7]=new_block(inode->i_dev)) {
95                                  inode->i_dirt=1;
96                                  inode->i_ctime=CURRENT_TIME;
97                          }
98                  if (!inode->i_zone[7])
99                          return 0;
    // 现在读取设备上该 i 节点的一次间接块。并取该间接块上第 block 项中的逻辑块号（盘块
    // 号）i。每一项占 2 个字节。如果是创建并且间接块的第 block 项中的逻辑块号为 0 的话，
    // 则申请一磁盘块，并让间接块中的第 block 项等于该新逻辑块块号。然后置位间接块的已
    // 修改标志。如果不是创建，则 i 就是需要映射（寻找）的逻辑块号。
100                 if (!(bh = bread(inode->i_dev,inode->i_zone[7])))
101                         return 0;
102                 i = ((unsigned short *) (bh->b_data))[block];
103                 if (create && !i)
104                         if (i=new_block(inode->i_dev)) {
105                                 ((unsigned short *) (bh->b_data))[block]=i;
106                                 bh->b_dirt=1;
107                         }
    // 最后释放该间接块占用的缓冲块，并返回磁盘上新申请或原有的对应 block 的逻辑块块号。
```

```
108                 brelse(bh);
109                 return i;
110         }
    // 若程序运行到此，则表明数据块属于二次间接块。其处理过程与一次间接块类似。下面是对
    // 二次间接块的处理。首先将 block 再减去间接块所容纳的块数（512）。然后根据是否设置
    // 了创建标志进行创建或寻找处理。如果是新创建并且 i 节点的二次间接块字段为 0，则需申
    // 请一磁盘块用于存放二次间接块的一级块信息，并将此实际磁盘块号填入二次间接块字段
    // 中。之后，置 i 节点已修改编制和修改时间。同样地，如果创建时申请磁盘块失败，则此
    // 时 i 节点二次间接块字段 i_zone[8]为 0，则返回 0。或者不是创建，但 i_zone[8]原来就
    // 为 0，表明 i 节点中没有间接块，于是映射磁盘块失败，返回 0 退出。
111         block -= 512;
112         if (create && !inode->i_zone[8])
113                 if (inode->i_zone[8]=new_block(inode->i_dev)) {
114                         inode->i_dirt=1;
115                         inode->i_ctime=CURRENT_TIME;
116                 }
117         if (!inode->i_zone[8])
118                 return 0;
    // 现在读取设备上该 i 节点的二次间接块。并取该二次间接块的一级块上第 (block/512)
    // 项中的逻辑块号 i。如果是创建并且二次间接块的一级块上第 (block/512) 项中的逻辑
    // 块号为 0 的话，则需申请一磁盘块（逻辑块）作为二次间接块的二级块 i，并让二次间接
    // 块的一级块中第 (block/512)项等于该二级块的块号 i。然后置位二次间接块的一级块已
    // 修改标志。并释放二次间接块的一级块。如果不是创建，则 i 就是需要映射（寻找）的逻
    // 辑块号。
119         if (!(bh=bread(inode->i_dev,inode->i_zone[8])))
120                 return 0;
121         i = ((unsigned short *)bh->b_data)[block>>9];
122         if (create && !i)
123                 if (i=new_block(inode->i_dev)) {
124                         ((unsigned short *) (bh->b_data))[block>>9]=i;
125                         bh->b_dirt=1;
126                 }
127         brelse(bh);
    // 如果二次间接块的二级块块号为 0，表示申请磁盘块失败或者原来对应块号就为 0，则返
    // 回 0 退出。否则就从设备上读取二次间接块的二级块，并取该二级块上第 block 项中的逻
    // 辑块号（与上 511 是为了限定 block 值不超过 511）。
128         if (!i)
129                 return 0;
130         if (!(bh=bread(inode->i_dev,i)))
131                 return 0;
132         i = ((unsigned short *)bh->b_data)[block&511];
    // 如果是创建并且二级块的第 block 项中逻辑块号为 0 的话，则申请一磁盘块（逻辑块），
    // 作为最终存放数据信息的块。并让二级块中的第 block 项等于该新逻辑块块号(i)。然后
    // 置位二级块的已修改标志。
133         if (create && !i)
134                 if (i=new_block(inode->i_dev)) {
135                         ((unsigned short *) (bh->b_data))[block&511]=i;
136                         bh->b_dirt=1;
137                 }
    // 最后释放该二次间接块的二级块，返回磁盘上新申请的或原有的对应 block 的逻辑块块号。
138         brelse(bh);
139         return i;
140 }
```

```
141
    //// 取文件数据块 block 在设备上对应的逻辑块号。
    // 参数：inode - 文件的内存 i 节点指针；block - 文件中的数据块号。
    // 若操作成功则返回对应的逻辑块号，否则返回 0。
142 int bmap(struct m_inode * inode,int block)
143 {
144         return _bmap(inode,block,0);
145 }
146
    //// 取文件数据块 block 在设备上对应的逻辑块号。如果对应的逻辑块不存在就创建一块。
    // 并返回设备上对应的逻辑块号。
    // 参数：inode - 文件的内存 i 节点指针；block - 文件中的数据块号。
    // 若操作成功则返回对应的逻辑块号，否则返回 0。
147 int create_block(struct m_inode * inode, int block)
148 {
149         return _bmap(inode,block,1);
150 }
151
    //// 放回（放置）一个 i 节点(回写入设备)。
    // 该函数主要用于把 i 节点引用计数值递减 1，并且若是管道 i 节点，则唤醒等待的进程。
    // 若是块设备文件 i 节点则刷新设备。并且若 i 节点的链接计数为 0，则释放该 i 节点占用
    // 的所有磁盘逻辑块，并释放该 i 节点。
152 void iput(struct m_inode * inode)
153 {
    // 首先判断参数给出的 i 节点的有效性，并等待 inode 节点解锁（如果已上锁的话）。如果 i
    // 节点的引用计数为 0，表示该 i 节点已经是空闲的。内核再要求对其进行放回操作，说明内
    // 核中其他代码有问题。于是显示错误信息并停机。
154         if (!inode)
155                 return;
156         wait_on_inode(inode);
157         if (!inode->i_count)
158                 panic("iput: trying to free free inode");
    // 如果是管道 i 节点，则唤醒等待该管道的进程，引用次数减 1，如果还有引用则返回。否则
    // 释放管道占用的内存页面，并复位该节点的引用计数值、已修改标志和管道标志，并返回。
    // 对于管道节点，inode->i_size 存放着内存页地址。参见 get_pipe_inode()，231，237 行。
159         if (inode->i_pipe) {
160                 wake_up(&inode->i_wait);
161                 wake_up(&inode->i_wait2);          //
162                 if (--inode->i_count)
163                         return;
164                 free_page(inode->i_size);
165                 inode->i_count=0;
166                 inode->i_dirt=0;
167                 inode->i_pipe=0;
168                 return;
169         }
    // 如果 i 节点对应的设备号 = 0，则将此节点的引用计数递减 1，返回。例如用于管道操作的
    // i 节点，其 i 节点的设备号为 0。
170         if (!inode->i_dev) {
171                 inode->i_count--;
172                 return;
173         }
    // 如果是块设备文件的 i 节点，此时逻辑块字段 0（i_zone[0]）中是设备号，则刷新该设备。
```

```
    // 并等待 i 节点解锁。
174         if (S_ISBLK(inode->i_mode)) {
175                 sync_dev(inode->i_zone[0]);
176                 wait_on_inode(inode);
177         }
    // 如果 i 节点的引用计数大于 1，则计数递减 1 后就直接返回（因为该 i 节点还有人在用，不能
    // 释放），否则就说明 i 节点的引用计数值为 1（因为第 157 行已经判断过引用计数是否为零）。
    // 如果 i 节点的链接数为 0，则说明 i 节点对应文件被删除。于是释放该 i 节点的所有逻辑块，
    // 并释放该 i 节点。函数 free_inode()用于实际释放 i 节点操作，即复位 i 节点对应的 i 节点位
    // 图比特位，清空 i 节点结构内容。
178 repeat:
179         if (inode->i_count>1) {
180                 inode->i_count--;
181                 return;
182         }
183         if (!inode->i_nlinks) {
184                 truncate(inode);
185                 free_inode(inode);          // bitmap.c 第 108 行开始处。
186                 return;
187         }
    // 如果该 i 节点已作过修改，则回写更新该 i 节点，并等待该 i 节点解锁。由于这里在写 i 节
    // 点时需要等待睡眠，此时其他进程有可能修改该 i 节点，因此在进程被唤醒后需要再次重复
    // 进行上述判断过程（repeat）。
188         if (inode->i_dirt) {
189                 write_inode(inode);         /* we can sleep - so do again */
190                 wait_on_inode(inode);       /* 因为我们睡眠了，所以需要重复判断 */
191                 goto repeat;
192         }
    // 程序若能执行到此，则说明该 i 节点的引用计数值 i_count 是 1、链接数不为零，并且内容
    // 没有被修改过。因此此时只要把 i 节点引用计数递减 1，返回。此时该 i 节点的 i_count=0，
    // 表示已释放。
193         inode->i_count--;
194         return;
195 }
196
    //// 从 i 节点表（inode_table）中获取一个空闲 i 节点项。
    // 寻找引用计数 count 为 0 的 i 节点，并将其写盘后清零，返回其指针。引用计数被置 1。
197 struct m_inode * get_empty_inode(void)
198 {
199         struct m_inode * inode;
200         static struct m_inode * last_inode = inode_table;   // 指向 i 节点表第 1 项。
201         int i;
202
    // 在初始化 last_inode 指针指向 i 节点表头一项后循环扫描整个 i 节点表。如果 last_inode
    // 已经指向 i 节点表的最后 1 项之后，则让其重新指向 i 节点表开始处，以继续循环寻找空闲
    // i 节点项。如果 last_inode 所指向的 i 节点的计数值为 0，则说明可能找到空闲 i 节点项。
    // 让 inode 指向该 i 节点。如果该 i 节点的已修改标志和锁定标志均为 0，则我们可以使用该
    // i 节点，于是退出 for 循环。
203         do {
204                 inode = NULL;
205                 for (i = NR_INODE; i ; i--) {               // NR_INODE = 32。
206                         if (++last_inode >= inode_table + NR_INODE)
207                                 last_inode = inode_table;
```

```
208                         if (!last_inode->i_count) {
209                                 inode = last_inode;
210                                 if (!inode->i_dirt && !inode->i_lock)
211                                         break;
212                         }
213                 }
    // 如果没有找到空闲 i 节点（inode = NULL），则将 i 节点表打印出来供调试使用，并停机。
214                 if (!inode) {
215                         for (i=0 ; i<NR_INODE ; i++)
216                                 printk("%04x: %6d\t",inode_table[i].i_dev,
217                                         inode_table[i].i_num);
218                         panic("No free inodes in mem");
219                 }
    // 等待该 i 节点解锁（如果又被上锁的话）。如果该 i 节点已修改标志被置位的话，则将该
    // i 节点刷新（同步）。因为刷新时可能会睡眠，因此需要再次循环等待该 i 节点解锁。
220                 wait_on_inode(inode);
221                 while (inode->i_dirt) {
222                         write_inode(inode);
223                         wait_on_inode(inode);
224                 }
225         } while (inode->i_count);
    // 如果 i 节点又被其他占用的话（i 节点的计数值不为 0 了），则重新寻找空闲 i 节点。否则
    // 说明已找到符合要求的空闲 i 节点项。则将该 i 节点项内容清零，并置引用计数为 1，返回
    // 该 i 节点指针。
226         memset(inode,0,sizeof(*inode));
227         inode->i_count = 1;
228         return inode;
229 }
230
    //// 获取管道节点。
    // 首先扫描 i 节点表，寻找一个空闲 i 节点项，然后取得一页空闲内存供管道使用。然后将得
    // 到的 i 节点的引用计数置为 2(读者和写者)，初始化管道头和尾，置 i 节点的管道类型表示。
    // 返回为 i 节点指针，如果失败则返回 NULL。
231 struct m_inode * get_pipe_inode(void)
232 {
233         struct m_inode * inode;
234
    // 首先从内存 i 节点表中取得一个空闲 i 节点。如果找不到空闲 i 节点则返回 NULL。然后为该
    // i 节点申请一页内存，并让节点的 i_size 字段指向该页面。如果已没有空闲内存，则释放该
    // i 节点，并返回 NULL。
235         if (!(inode = get_empty_inode()))
236                 return NULL;
237         if (!(inode->i_size=get_free_page())) {     // 节点的 i_size 字段指向缓冲区。
238                 inode->i_count = 0;
239                 return NULL;
240         }
    // 然后设置该 i 节点的引用计数为 2，并复位复位管道头尾指针。i 节点逻辑块号数组 i_zone[]
    // 的 i_zone[0]和 i_zone[1]中分别用来存放管道头和管道尾指针。最后设置 i 节点是管道 i 节
    // 点标志并返回该 i 节点号。
241         inode->i_count = 2;     /* sum of readers/writers */   /* 读/写两者总计 */
242         PIPE_HEAD(*inode) = PIPE_TAIL(*inode) = 0;  // 复位管道头尾指针。
243         inode->i_pipe = 1;                          // 置节点为管道使用的标志。
244         return inode;
```

```
245 }
246
    //// 取得一个 i 节点。
    // 参数：dev - 设备号；nr - i 节点号。
    // 从设备上读取指定节点号的 i 节点结构内容到内存 i 节点表中，并且返回该 i 节点指针。
    // 首先在位于高速缓冲区中的 i 节点表中搜寻，若找到指定节点号的 i 节点则在经过一些判断
    // 处理后返回该 i 节点指针。否则从设备 dev 上读取指定 i 节点号的 i 节点信息放入 i 节点表
    // 中，并返回该 i 节点指针。
247 struct m_inode * iget(int dev,int nr)
248 {
249         struct m_inode * inode, * empty;
250
    // 首先判断参数有效性。若设备号是 0，则表明内核代码问题，显示出错信息并停机。然后预
    // 先从 i 节点表中取一个空闲 i 节点备用。
251         if (!dev)
252                 panic("iget with dev==0");
253         empty = get_empty_inode();
    // 接着扫描 i 节点表。寻找参数指定节点号 nr 的 i 节点。并递增该节点的引用次数。如果当
    // 前扫描 i 节点的设备号不等于指定的设备号或者节点号不等于指定的节点号，则继续扫描。
254         inode = inode_table;
255         while (inode < NR_INODE+inode_table) {
256                 if (inode->i_dev != dev || inode->i_num != nr) {
257                         inode++;
258                         continue;
259                 }
    // 如果找到指定设备号 dev 和节点号 nr 的 i 节点，则等待该节点解锁（如果已上锁的话）。
    // 在等待该节点解锁过程中，i 节点表可能会发生变化。所以再次进行上述相同判断。如果发
    // 生了变化，则再次重新扫描整个 i 节点表。
260                 wait_on_inode(inode);
261                 if (inode->i_dev != dev || inode->i_num != nr) {
262                         inode = inode_table;
263                         continue;
264                 }
    // 到这里表示找到相应的 i 节点。于是将该 i 节点引用计数增 1。然后再作进一步检查，看它
    // 是否是另一个文件系统的安装点。若是则寻找被安装文件系统根节点并返回。如果该 i 节点
    // 的确是其他文件系统的安装点，则在超级块表中搜寻安装在此 i 节点的超级块。如果没有找
    // 到，则显示出错信息，并放回本函数开始时获取的空闲节点 empty，返回该 i 节点指针。
265                 inode->i_count++;
266                 if (inode->i_mount) {
267                         int i;
268
269                         for (i = 0 ; i<NR_SUPER ; i++)
270                                 if (super_block[i].s_imount==inode)
271                                         break;
272                         if (i >= NR_SUPER) {
273                                 printk("Mounted inode hasn't got sb\n");
274                                 if (empty)
275                                         iput(empty);
276                                 return inode;
277                         }
    // 执行到这里表示已经找到安装到 inode 节点的文件系统超级块。于是将该 i 节点写盘放回，
    // 并从安装在此 i 节点上的文件系统超级块中取设备号，并令 i 节点号为 ROOT_INO，即为 1。
    // 然后重新扫描整个 i 节点表，以获取该被安装文件系统的根 i 节点信息。
```

```
278                         iput(inode);
279                         dev = super_block[i].s_dev;
280                         nr = ROOT_INO;
281                         inode = inode_table;
282                         continue;
283                 }
    // 最终我们找到了相应的 i 节点。因此可以放弃本函数开始处临时申请的空闲 i 节点，返回
    // 找到的 i 节点指针。
284                 if (empty)
285                         iput(empty);
286                 return inode;
287         }
    // 如果我们在 i 节点表中没有找到指定的 i 节点，则利用前面申请的空闲 i 节点 empty 在 i
    // 节点表中建立该 i 节点。并从相应设备上读取该 i 节点信息，返回该 i 节点指针。
288         if (!empty)
289                 return (NULL);
290         inode=empty;
291         inode->i_dev = dev;                 // 设置 i 节点的设备。
292         inode->i_num = nr;                  // 设置 i 节点号。
293         read_inode(inode);
294         return inode;
295 }
296
    //// 读取指定 i 节点信息。
    // 从设备上读取含有指定 i 节点信息的 i 节点盘块，然后复制到指定的 i 节点结构中。 为了
    // 确定 i 节点所在的设备逻辑块号（或缓冲块），必须首先读取相应设备上的超级块，以获取
    // 用于计算逻辑块号的每块 i 节点数信息 INODES_PER_BLOCK。 在计算出 i 节点所在的逻辑块
    // 号后，就把该逻辑块读入一缓冲块中。然后把缓冲块中相应位置处的 i 节点内容复制到参数
    // 指定的位置处。
297 static void read_inode(struct m_inode * inode)
298 {
299         struct super_block * sb;
300         struct buffer_head * bh;
301         int block;
302
    // 首先锁定该 i 节点，并取该节点所在设备的超级块。
303         lock_inode(inode);
304         if (!(sb=get_super(inode->i_dev)))
305                 panic("trying to read inode without dev");
    // 该 i 节点所在的设备逻辑块号 = (启动块 + 超级块) + i 节点位图占用的块数 + 逻辑块位
    // 图占用的块数 + (i 节点号-1)/每块含有的 i 节点数。虽然 i 节点号从 0 开始编号，但第 1
    // 个 0 号 i 节点不用，并且磁盘上也不保存对应的 0 号 i 节点结构。因此存放 i 节点的盘块的
    // 第 1 块上保存的是 i 节点号是是 1--16 的 i 节点结构而不是 0--15 的。因此在上面计算 i 节
    // 点号对应的 i 节点结构所在盘块时需要减 1，即：B =(i 节点号-1)/每块含有 i 节点结构数。
    // 例如，节点号 16 的 i 节点结构应该在 B=(16-1)/16 = 0 的块上。 这里我们从设备上读取该
    // i 节点所在的逻辑块，并复制指定 i 节点内容到 inode 指针所指位置处。
306         block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
307                 (inode->i_num-1)/INODES_PER_BLOCK;
308         if (!(bh=bread(inode->i_dev,block)))
309                 panic("unable to read i-node block");
310         *(struct d_inode *)inode =
311                 ((struct d_inode *)bh->b_data)
312                         [(inode->i_num-1)%INODES_PER_BLOCK];
```

```
    // 最后释放读入的缓冲块，并解锁该 i 节点。对于块设备文件，还需要设置 i 节点的文件最大
    // 长度值。
313         brelse(bh);
314         if (S_ISBLK(inode->i_mode)) {
315                 int i = inode->i_zone[0];   // 对于块设备文件，i_zone[0]中是设备号。
316                 if (blk_size[MAJOR(i)])
317                         inode->i_size = 1024*blk_size[MAJOR(i)][MINOR(i)];
318                 else
319                         inode->i_size = 0x7fffffff;
320         }
321         unlock_inode(inode);
322 }
323
    //// 将 i 节点信息写入缓冲区中。
    // 该函数把参数指定的 i 节点写入缓冲区相应的缓冲块中，待缓冲区刷新时会写入盘中。为了
    // 确定 i 节点所在的设备逻辑块号（或缓冲块），必须首先读取相应设备上的超级块，以获取
    // 用于计算逻辑块号的每块 i 节点数信息 INODES_PER_BLOCK。 在计算出 i 节点所在的逻辑块
    // 号后，就把该逻辑块读入一缓冲块中。然后把 i 节点内容复制到缓冲块的相应位置处。
324 static void write_inode(struct m_inode * inode)
325 {
326         struct super_block * sb;
327         struct buffer_head * bh;
328         int block;
329
    // 首先锁定该 i 节点，如果该 i 节点没有被修改过或者该 i 节点的设备号等于零，则解锁该
    // i 节点，并退出。对于没有被修改过的 i 节点，其内容与缓冲区中或设备中的相同。 然后
    // 获取该 i 节点的超级块。
330         lock_inode(inode);
331         if (!inode->i_dirt || !inode->i_dev) {
332                 unlock_inode(inode);
333                 return;
334         }
335         if (!(sb=get_super(inode->i_dev)))
336                 panic("trying to write inode without device");
    // 该 i 节点所在的设备逻辑块号 = (启动块 + 超级块) + i 节点位图占用的块数 + 逻辑块位
    // 图占用的块数 + (i 节点号-1)/每块含有的 i 节点数。  我们从设备上读取该 i 节点所在的
    // 逻辑块，并将该 i 节点信息复制到逻辑块对应该 i 节点的项位置处。
337         block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
338                 (inode->i_num-1)/INODES_PER_BLOCK;
339         if (!(bh=bread(inode->i_dev,block)))
340                 panic("unable to read i-node block");
341         ((struct d_inode *)bh->b_data)
342                 [(inode->i_num-1)%INODES_PER_BLOCK] =
343                         *(struct d_inode *)inode;
    // 然后置缓冲区已修改标志，而 i 节点内容已经与缓冲区中的一致，因此修改标志置零。然后
    // 释放该含有 i 节点的缓冲区，并解锁该 i 节点。
344         bh->b_dirt=1;
345         inode->i_dirt=0;
346         brelse(bh);
347         unlock_inode(inode);
348 }
349
```

12.5 程序 12-5 linux/fs/super.c

```
 1 /*
 2  *  linux/fs/super.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * super.c contains code to handle the super-block tables.
 9  */
   /*
    * super.c 程序中含有处理超级块表的代码。
    */
10 #include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）可选项。
11 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据，
                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
12 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
13 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
14
15 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
16 #include <sys/stat.h>     // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。
17
   // 对指定设备执行高速缓冲与设备上数据的同步操作（fs/buffer.c，59 行）。
18 int sync_dev(int dev);
   // 等待击键（kernel/chr_drv/tty_io.c，140 行）。
19 void wait_for_keypress(void);
20
21 /* set_bit uses setb, as gas doesn't recognize setc */
   /* set_bit()使用了 setb 指令，因为汇编编译器 gas 不能识别指令 setc */
   //// 测试指定位偏移处比特位的值，并返回该原比特位值（应该取名为 test_bit()更妥帖）。
   // 嵌入式汇编宏。参数 bitnr 是比特位偏移值，addr 是测试比特位操作的起始地址。
   // %0 - ax(__res)，%1 - 0，%2 - bitnr，%3 - addr
   // 第 23 行定义了一个局部寄存器变量。该变量将被保存在 eax 寄存器中，以便于高效访问和
   // 操作。第 24 行上指令 bt 用于对比特位进行测试（Bit Test）。它会把地址 addr（%3）和
   // 比特位偏移量 bitnr（%2）指定的比特位的值放入进位标志 CF 中。 指令 setb 用于根据进
   // 位标志 CF 设置操作数%al。如果 CF = 1 则%al = 1，否则%al = 0。
22 #define set_bit(bitnr,addr) ({ \
23 register int __res __asm__("ax"); \
24 __asm__("bt %2,%3;setb %%al":"=a" (__res):"a" (0),"r" (bitnr),"m" (*(addr))); \
25 __res; })
26
27 struct super_block super_block[NR_SUPER];   // 超级块结构表数组（NR_SUPER = 8）。
28 /* this is initialized in init/main.c */
   /* ROOT_DEV 已在 init/main.c 中被初始化 */
29 int ROOT_DEV = 0;                           // 根文件系统设备号。
30
   // 以下 3 个函数（lock_super()、free_super()和 wait_on_super()）的作用与 inode.c 文
   // 件中头 3 个函数的作用雷同，只是这里操作的对象换成了超级块。
   //// 锁定超级块。
   // 如果超级块已被锁定，则将当前任务置为不可中断的等待状态，并添加到该超级块等待队
```

```
   // 列 s_wait 中。直到该超级块解锁并明确地唤醒本任务。然后对其上锁。
31 static void lock_super(struct super_block * sb)
32 {
33         cli();                                // 关中断。
34         while (sb->s_lock)                    // 如果该超级块已经上锁，则睡眠等待。
35                 sleep_on(&(sb->s_wait));      // kernel/sched.c，第 199 行。
36         sb->s_lock = 1;                       // 给该超级块加锁（置锁定标志）。
37         sti();                                // 开中断。
38 }
39
   //// 对指定超级块解锁。
   // 复位超级块的锁定标志，并明确地唤醒等待在此超级块等待队列 s_wait 上的所有进程。
   // 如果使用 ulock_super 这个名称则可能更妥帖。
40 static void free_super(struct super_block * sb)
41 {
42         cli();
43         sb->s_lock = 0;                       // 复位锁定标志。
44         wake_up(&(sb->s_wait));               // 唤醒等待该超级块的进程。
45         sti();                                // wake_up()在 kernel/sched.c，第 188 行。
46 }
47
   //// 睡眠等待超级块解锁。
   // 如果超级块已被锁定，则将当前任务置为不可中断的等待状态，并添加到该超级块的等待队
   // 列 s_wait 中。直到该超级块解锁并明确地唤醒本任务。
48 static void wait_on_super(struct super_block * sb)
49 {
50         cli();
51         while (sb->s_lock)                    // 如果超级块已经上锁，则睡眠等待。
52                 sleep_on(&(sb->s_wait));
53         sti();
54 }
55
   //// 取指定设备的超级块。
   // 在超级块表（数组）中搜索指定设备 dev 的超级块结构信息。若找到则返回超级块的指针，
   // 否则返回空指针。
56 struct super_block * get_super(int dev)
57 {
58         struct super_block * s;               // s 是超级块数据结构指针。
59
   // 首先判断参数给出设备的有效性。若设备号为 0 则返回空指针。然后让 s 指向超级块数组
   // 起始处，开始搜索整个超级块数组，以寻找指定设备 dev 的超级块。 第 62 行上的指针赋
   // 值语句"s = 0+super_block" 等同于 "s = super_block"、"s = &super_block[0]"。
60         if (!dev)
61                 return NULL;
62         s = 0+super_block;
63         while (s < NR_SUPER+super_block)
   // 如果当前搜索项是指定设备的超级块，即该超级块的设备号字段值与函数参数指定的相同，
   // 则先等待该超级块解锁（若已被其他进程上锁的话）。在等待期间，该超级块项有可能被
   // 其他设备使用，因此等待返回之后需再判断一次是否是指定设备的超级块，如果是则返回
   // 该超级块的指针。否则就重新对超级块数组再搜索一遍，因此此时 s 需重又指向超级块数
   // 组开始处。
64                 if (s->s_dev == dev) {
65                         wait_on_super(s);
```

```
66                          if (s->s_dev == dev)
67                                  return s;
68                          s = 0+super_block;
   // 如果当前搜索项不是，则检查下一项。如果没有找到指定的超级块，则返回空指针。
69                  } else
70                          s++;
71          return NULL;
72  }
73
   //// 释放（放回）指定设备的超级块。
   // 释放设备所使用的超级块数组项（置 s_dev=0），并释放该设备 i 节点位图和逻辑块位图所
   // 占用的高速缓冲块。如果超级块对应的文件系统是根文件系统，或者其某个 i 节点上已经安
   // 装有其他的文件系统，则不能释放该超级块。
74  void put_super(int dev)
75  {
76          struct super_block * sb;
77          int i;
78
   // 首先判断参数的有效性和合法性。如果指定设备是根文件系统设备，则显示警告信息“根系
   // 统盘改变了，准备生死决战吧”，并返回。然后在超级块表中寻找指定设备号的文件系统超
   // 级块。如果找不到指定设备的超级块，则返回。另外，如果该超级块指明该文件系统所安装
   // 到的 i 节点还没有被处理过，则显示警告信息并返回。在文件系统卸载（umount）操作中，
   // s_imount 会先被置成 Null 以后才会调用本函数，参见第 192 行。
79          if (dev == ROOT_DEV) {
80                  printk("root diskette changed: prepare for armageddon\n\r");
81                  return;
82          }
83          if (!(sb = get_super(dev)))
84                  return;
85          if (sb->s_imount) {
86                  printk("Mounted disk changed - tssk, tssk\n\r");
87                  return;
88          }
   // 然后在找到指定设备的超级块之后，我们先锁定该超级块，再置该超级块对应的设备号字段
   // s_dev 为 0，也即释放该设备上的文件系统超级块。然后释放该超级块占用的其他内核资源，
   // 即释放该设备上文件系统 i 节点位图和逻辑块位图在缓冲区中所占用的缓冲块。下面常数符
   // 号 I_MAP_SLOTS 和 Z_MAP_SLOTS 均等于 8，用于分别指明 i 节点位图和逻辑块位图占用的磁
   // 盘逻辑块数。注意，若这些缓冲块内容被修改过，则需要作同步操作才能把缓冲块中的数据
   // 写入设备中。函数最后对该超级块解锁，并返回。
89          lock_super(sb);
90          sb->s_dev = 0;                          // 置超级块空闲。
91          for(i=0;i<I_MAP_SLOTS;i++)
92                  brelse(sb->s_imap[i]);
93          for(i=0;i<Z_MAP_SLOTS;i++)
94                  brelse(sb->s_zmap[i]);
95          free_super(sb);
96          return;
97  }
98
   //// 读取指定设备的超级块。
   // 如果指定设备 dev 上的文件系统超级块已经在超级块表中，则直接返回该超级块项的指针。
   // 否则就从设备 dev 上读取超级块到缓冲块中，并复制到超级块表中。并返回超级块指针。
99  static struct super_block * read_super(int dev)
```

```
100 {
101         struct super_block * s;
102         struct buffer_head * bh;
103         int i,block;
104
    // 首先判断参数的有效性。如果没有指明设备，则返回空指针。然后检查该设备是否可更换
    // 过盘片（也即是否是软盘设备）。如果更换过盘，则高速缓冲区有关该设备的所有缓冲块
    // 均失效，需要进行失效处理，即释放原来加载的文件系统。
105         if (!dev)
106                 return NULL;
107         check_disk_change(dev);
    // 如果该设备的超级块已经在超级块表中，则直接返回该超级块的指针。否则，首先在超级
    // 块数组中找出一个空项(也即字段 s_dev=0 的项)。如果数组已经占满则返回空指针。
108         if (s = get_super(dev))
109                 return s;
110         for (s = 0+super_block ;; s++) {
111                 if (s >= NR_SUPER+super_block)
112                         return NULL;
113                 if (!s->s_dev)
114                         break;
115         }
    // 在超级块数组中找到空项之后，就将该超级块项用于指定设备 dev 上的文件系统。于是对
    // 该超级块结构中的内存字段进行部分初始化处理。
116         s->s_dev = dev;                         // 用于 dev 设备上的文件系统。
117         s->s_isup = NULL;
118         s->s_imount = NULL;
119         s->s_time = 0;
120         s->s_rd_only = 0;
121         s->s_dirt = 0;
    // 然后锁定该超级块，并从设备上读取超级块信息到 bh 指向的缓冲块中。超级块位于块设备
    // 的第 2 个逻辑块（1 号块）中，（第 1 个是引导盘块）。如果读超级块操作失败，则释放上
    // 面选定的超级块数组中的项（即置 s_dev=0），并解锁该项，返回空指针退出。否则就将设
    // 备上读取的超级块信息从缓冲块数据区复制到超级块数组相应项结构中。并释放存放读取信
    // 息的高速缓冲块。
122         lock_super(s);
123         if (!(bh = bread(dev,1))) {
124                 s->s_dev=0;
125                 free_super(s);
126                 return NULL;
127         }
128         *((struct d_super_block *) s) =
129                 *((struct d_super_block *) bh->b_data);
130         brelse(bh);
    // 现在我们从设备 dev 上得到了文件系统的超级块，于是开始检查这个超级块的有效性并从设
    // 备上读取 i 节点位图和逻辑块位图等信息。如果所读取的超级块的文件系统魔数字段不对，
    // 说明设备上不是正确的文件系统，因此同上面一样，释放上面选定的超级块数组中的项，并
    // 解锁该项，返回空指针退出。对于该版 Linux 内核，只支持 MINIX 文件系统 1.0 版本，其魔
    // 数是 0x137f。
131         if (s->s_magic != SUPER_MAGIC) {
132                 s->s_dev = 0;
133                 free_super(s);
134                 return NULL;
135         }
```

```
    // 下面开始读取设备上 i 节点位图和逻辑块位图数据。首先初始化内存超级块结构中位图空间。
    // 然后从设备上读取 i 节点位图和逻辑块位图信息，并存放在超级块对应字段中。i 节点位图
    // 保存在设备上 2 号块开始的逻辑块中，共占用 s_imap_blocks 个块。逻辑块位图在 i 节点位
    // 图所在块的后续块中，共占用 s_zmap_blocks 个块。
136         for (i=0;i<I_MAP_SLOTS;i++)                     // 初始化操作。
137                 s->s_imap[i] = NULL;
138         for (i=0;i<Z_MAP_SLOTS;i++)
139                 s->s_zmap[i] = NULL;
140         block=2;
141         for (i=0 ; i < s->s_imap_blocks ; i++)          // 读取设备中 i 节点位图。
142                 if (s->s_imap[i]=bread(dev,block))
143                         block++;
144                 else
145                         break;
146         for (i=0 ; i < s->s_zmap_blocks ; i++)          // 读取设备中逻辑块位图。
147                 if (s->s_zmap[i]=bread(dev,block))
148                         block++;
149                 else
150                         break;
    // 如果读出的位图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。因此只能释放前面申请并占用的所有资源，即释放 i 节点位图和逻辑块位图
    // 占用的高速缓冲块、释放上面选定的超级块数组项、解锁该超级块项，并返回空指针退出。
151         if (block != 2+s->s_imap_blocks+s->s_zmap_blocks) {
152                 for(i=0;i<I_MAP_SLOTS;i++)              // 释放位图占用的高速缓冲块。
153                         brelse(s->s_imap[i]);
154                 for(i=0;i<Z_MAP_SLOTS;i++)
155                         brelse(s->s_zmap[i]);
156                 s->s_dev=0;                             // 释放选定的超级块数组项。
157                 free_super(s);                          // 解锁该超级块项。
158                 return NULL;
159         }
    // 否则一切成功。另外，由于对于申请空闲 i 节点的函数来讲，如果设备上所有的 i 节点已经
    // 全被使用，则查找函数会返回 0 值。因此 0 号 i 节点是不能用的，所以这里将位图中第 1 块
    // 的最低比特位设置为 1，以防止文件系统分配 0 号 i 节点。同样的道理，也将逻辑块位图的
    // 最低位设置为 1。最后函数解锁该超级块，并返回超级块指针。
160         s->s_imap[0]->b_data[0] |= 1;
161         s->s_zmap[0]->b_data[0] |= 1;
162         free_super(s);
163         return s;
164 }
165
    //// 卸载文件系统（系统调用）。
    // 参数 dev_name 是文件系统所在设备的设备文件名。
    // 该函数首先根据参数给出的块设备文件名获得设备号，然后复位文件系统超级块中的相应字
    // 段，释放超级块和位图占用的缓冲块，最后对该设备执行高速缓冲与设备上数据的同步操作。
    // 若卸载操作成功则返回 0，否则返回出错码。
166 int sys_umount(char * dev_name)
167 {
168         struct m_inode * inode;
169         struct super_block * sb;
170         int dev;
171
    // 首先根据设备文件名找到对应的 i 节点，并取其中的设备号。设备文件所定义设备的设备号
```

```
    // 是保存在其 i 节点的 i_zone[0]中的。 参见后面 namei.c 程序中系统调用 sys_mknod()的代
    // 码第 445 行。另外，由于文件系统需要存放在块设备上，因此如果不是块设备文件，则放回
    // 刚申请的 i 节点 dev_i，返回出错码。
172         if (!(inode=namei(dev_name)))
173                 return -ENOENT;
174         dev = inode->i_zone[0];
175         if (!S_ISBLK(inode->i_mode)) {
176                 iput(inode);                            // fs/inode.c，第 150 行。
177                 return -ENOTBLK;
178         }
    // OK，现在上面为了得到设备号而取得的 i 节点已完成了它的使命，因此这里放回该设备文件
    // 的 i 节点。接着我们来检查一下卸载该文件系统的条件是否满足。如果设备上是根文件系统，
    // 则不能被卸载，返回忙出错号。
179         iput(inode);
180         if (dev==ROOT_DEV)
181                 return -EBUSY;
    // 如果在超级块表中没有找到该设备上文件系统的超级块，或者已找到但是该设备上文件系统
    // 没有安装过，则返回出错码。如果超级块所指明的被安装到的 i 节点并没有置位其安装标志
    // i_mount，则显示警告信息。 然后查找一下 i 节点表，看看是否有进程在使用该设备上的文
    // 件，如果有则返回忙出错码。
182         if (!(sb=get_super(dev)) || !(sb->s_imount))
183                 return -ENOENT;
184         if (!sb->s_imount->i_mount)
185                 printk("Mounted inode has i_mount=0\n");
186         for (inode=inode_table+0 ; inode<inode_table+NR_INODE ; inode++)
187                 if (inode->i_dev==dev && inode->i_count)
188                                 return -EBUSY;
    // 现在该设备上文件系统的卸载条件均得到满足，因此我们可以开始实施真正的卸载操作了。
    // 首先复位被安装到的 i 节点的安装标志，释放该 i 节点。然后置超级块中被安装 i 节点字段
    // 为空，并放回设备文件系统的根 i 节点，接着置超级块中被安装系统根 i 节点指针为空。
189         sb->s_imount->i_mount=0;
190         iput(sb->s_imount);
191         sb->s_imount = NULL;
192         iput(sb->s_isup);
193         sb->s_isup = NULL;
    // 最后我们释放该设备上的超级块以及位图占用的高速缓冲块，并对该设备执行高速缓冲与设
    // 备上数据的同步操作。然后返回 0（卸载成功）。
194         put_super(dev);
195         sync_dev(dev);
196         return 0;
197 }
198
    //// 安装文件系统（系统调用）。
    // 参数 dev_name 是设备文件名，dir_name 是安装到的目录名，rw_flag 被安装文件系统的可
    // 读写标志。将被加载的地方必须是一个目录名，并且对应的 i 节点没有被其他程序占用。
    // 若操作成功则返回 0，否则返回出错号。
199 int sys_mount(char * dev_name, char * dir_name, int rw_flag)
200 {
201         struct m_inode * dev_i, * dir_i;
202         struct super_block * sb;
203         int dev;
204
    // 首先根据设备文件名找到对应的 i 节点，以取得其中的设备号。对于块特殊设备文件，设备
```

```
    // 号在其 i 节点的 i_zone[0]中。另外，由于文件系统必须在块设备中，因此如果不是块设备
    // 文件，则放回刚取得的 i 节点 dev_i，返回出错码。
205         if (!(dev_i=namei(dev_name)))
206                 return -ENOENT;
207         dev = dev_i->i_zone[0];
208         if (!S_ISBLK(dev_i->i_mode)) {
209                 iput(dev_i);
210                 return -EPERM;
211         }
    // OK，现在上面为了得到设备号而取得的 i 节点 dev_i 已完成了它的使命，因此这里放回该设
    // 备文件的 i 节点。接着我们来检查一下文件系统安装到的目录名是否有效。于是根据给定的
    // 目录文件名找到对应的 i 节点 dir_i。 如果该 i 节点的引用计数不为 1（仅在这里引用），
    // 或者该 i 节点的节点号是根文件系统的节点号 1，则放回该 i 节点返回出错码。另外，如果
    // 该节点不是一个目录文件节点，则也放回该 i 节点，返回出错码。因为文件系统只能安装在
    // 一个目录名上。
212         iput(dev_i);
213         if (!(dir_i=namei(dir_name)))
214                 return -ENOENT;
215         if (dir_i->i_count != 1 || dir_i->i_num == ROOT_INO) {
216                 iput(dir_i);
217                 return -EBUSY;
218         }
219         if (!S_ISDIR(dir_i->i_mode)) {                  // 安装点需要是一个目录名。
220                 iput(dir_i);
221                 return -EPERM;
222         }
    // 现在安装点也检查完毕，我们开始读取要安装文件系统的超级块信息。如果读超级块操作失
    // 败，则放回该安装点 i 节点 dir_i 并返回出错码。一个文件系统的超级块会首先从超级块表
    // 中进行搜索，如果不在超级块表中就从设备上读取。
223         if (!(sb=read_super(dev))) {
224                 iput(dir_i);
225                 return -EBUSY;
226         }
    // 在得到了文件系统超级块之后，我们对它先进行检测一番。如果将要被安装的文件系统已经
    // 安装在其他地方，则放回该 i 节点，返回出错码。如果将要安装到的 i 节点已经安装了文件
    // 系统（安装标志已经置位），则放回该 i 节点，也返回出错码。
227         if (sb->s_imount) {
228                 iput(dir_i);
229                 return -EBUSY;
230         }
231         if (dir_i->i_mount) {
232                 iput(dir_i);
233                 return -EPERM;
234         }
    // 最后设置被安装文件系统超级块的“被安装到 i 节点”字段指向安装到的目录名的 i 节点。
    // 并设置安装位置 i 节点的安装标志和节点已修改标志。然后返回 0（安装成功）。
235         sb->s_imount=dir_i;
236         dir_i->i_mount=1;
237         dir_i->i_dirt=1;/* NOTE! we don't iput(dir_i) */ /*注意!这里没用 iput(dir_i)*/
238         return 0;       /* we do that in umount */      /* 这将在 umount 内操作 */
239 }
240
    //// 安装根文件系统。
```

```
    // 该函数属于系统初始化操作的一部分。函数首先初始化文件表数组 file_table[]和超级块表
    // （数组），然后读取根文件系统超级块，并取得文件系统根 i 节点。 最后统计并显示出根文
    // 件系统上的可用资源（空闲块数和空闲 i 节点数）。 该函数会在系统开机进行初始化设置时
    // （sys_setup()）调用（blk_drv/hd.c，157 行）。
241 void mount_root(void)
242 {
243         int i,free;
244         struct super_block * p;
245         struct m_inode * mi;
246
    // 若磁盘 i 节点结构不是 32 字节，则出错停机。该判断用于防止修改代码时出现不一致情况。
247         if (32 != sizeof (struct d_inode))
248                 panic("bad i-node size");
    // 首先初始化文件表数组（共 64 项，即系统同时只能打开 64 个文件）和超级块表。这里将所
    // 有文件结构中的引用计数设置为 0（表示空闲），并把超级块表中各项结构的设备字段初始
    // 化为 0（也表示空闲）。如果根文件系统所在设备是软盘的话，就提示“插入根文件系统盘，
    // 并按回车键”，并等待按键。
249         for(i=0;i<NR_FILE;i++)                          // 初始化文件表。
250                 file_table[i].f_count=0;
251         if (MAJOR(ROOT_DEV) == 2) {                     // 提示插入根文件系统盘。
252                 printk("Insert root floppy and press ENTER");
253                 wait_for_keypress();
254         }
255         for(p = &super_block[0] ; p < &super_block[NR_SUPER] ; p++) {
256                 p->s_dev = 0;                           // 初始化超级块表。
257                 p->s_lock = 0;
258                 p->s_wait = NULL;
259         }
    // 做好以上“份外”的初始化工作之后，我们开始安装根文件系统。于是从根设备上读取文件
    // 系统超级块，并取得文件系统的根 i 节点（1 号节点）在内存 i 节点表中的指针。如果读根
    // 设备上超级块失败或取根节点失败，则都显示信息并停机。
260         if (!(p=read_super(ROOT_DEV)))
261                 panic("Unable to mount root");
262         if (!(mi=iget(ROOT_DEV,ROOT_INO)))              // 在 fs.h 中 ROOT_INO 定义为 1。
263                 panic("Unable to read root i-node");
    // 现在我们对超级块和根 i 节点进行设置。把根 i 节点引用次数递增 3 次。因为下面 266 行上
    // 也引用了该 i 节点。另外，iget() 函数中 i 节点引用计数已被设置为 1。然后置该超级块的
    // 被安装文件系统 i 节点和被安装到 i 节点字段为该 i 节点。再设置当前进程的当前工作目录
    // 和根目录 i 节点。此时当前进程是 1 号进程（init 进程）。
264         mi->i_count += 3 ;      /* NOTE! it is logically used 4 times, not 1 */
                                    /* 注意！从逻辑上讲，它已被引用了 4 次，而不是 1 次 */
265         p->s_isup = p->s_imount = mi;
266         current->pwd = mi;
267         current->root = mi;
    // 然后我们对根文件系统上的资源作统计工作。统计该设备上空闲块数和空闲 i 节点数。首先
    // 令 i 等于超级块中表明的设备逻辑块总数。然后根据逻辑块位图中相应比特位的占用情况统
    // 计出空闲块数。这里宏函数 set_bit()只是在测试比特位，而非设置比特位。"i&8191" 用于
    // 取得 i 节点号在当前位图块中对应的比特位偏移值。"i>>13" 是将 i 除以 8192，也即除一个
    // 磁盘块包含的比特位数。
268         free=0;
269         i=p->s_nzones;
270         while (-- i >= 0)
271                 if (!set_bit(i&8191,p->s_zmap[i>>13]->b_data))
```

```
272                         free++;
    // 在显示过设备上空闲逻辑块数/逻辑块总数之后。我们再统计设备上空闲 i 节点数。首先令 i
    // 等于超级块中表明的设备上 i 节点总数+1。加 1 是将 0 节点也统计进去。然后根据 i 节点位
    // 图中相应比特位的占用情况计算出空闲 i 节点数。最后再显示设备上可用空闲 i 节点数和 i
    // 节点总数。
273         printk("%d/%d free blocks\n\r",free,p->s_nzones);
274         free=0;
275         i=p->s_ninodes+1;
276         while (-- i >= 0)
277                 if (!set_bit(i&8191,p->s_imap[i>>13]->b_data))
278                         free++;
279         printk("%d/%d free inodes\n\r",free,p->s_ninodes);
280 }
281
```

12.6 程序 12-6 linux/fs/namei.c

```
 1 /*
 2  *  linux/fs/namei.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * Some corrections by tytso.
 9  */
10
11 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据等。
12 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
13 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14
15 #include <string.h>       // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
16 #include <fcntl.h>        // 文件控制头文件。文件及其描述符的操作控制常数符号的定义。
17 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
18 #include <const.h>        // 常数符号头文件。目前仅定义 i 节点中 i_mode 字段的各标志位。
19 #include <sys/stat.h>     // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。
20
   // 由文件名查找对应 i 节点的内部函数。
21 static struct m_inode * _namei(const char * filename, struct m_inode * base,
22         int follow_links);
23
   // 下面宏中右侧表达式是访问数组的一种特殊使用方法。它基于这样的一个事实，即用数组名和
   // 数组下标所表示的数组项（例如 a[b]）的值等同于使用数组首指针（地址）加上该项偏移地址
   // 的形式的值 *(a + b)，同时可知项 a[b]也可以表示成 b[a]的形式。因此对于字符数组项形式
   // 为 "LoveYou"[2]（或者 2["LoveYou"]）就等同于*("LoveYou" + 2)。另外，字符串"LoveYou"
   // 在内存中被存储的位置就是其地址，因此数组项"LoveYou"[2]的值就是该字符串中索引值为 2
   // 的字符"v"所对应的 ASCII 码值 0x76，或用八进制表示就是 0166。在 C 语言中，字符也可以用
   // 其 ASCII 码值来表示，方法是在字符的 ASCII 码值前面加一个反斜杠。例如字符 "v"可以表示
   // 成"\x76"或者"\166"。因此对于不可显示的字符（例如 ASCII 码值为 0x00--0x1f 的控制字符）
   // 就可用其 ASCII 码值来表示。
   //
   // 下面是访问模式宏。x 是头文件 include/fcntl.h 中第 7 行开始定义的文件访问（打开）标志。
   // 这个宏根据文件访问标志 x 的值来索引双引号中对应的数值。双引号中有 4 个八进制数值（实
   // 际表示 4 个控制字符）："\004\002\006\377"，分别表示读、写和执行的权限为：r、w、rw
   // 和 wxrwxrwx，并且分别对应 x 的索引值 0--3。 例如，如果 x 为 2，则该宏返回八进制值 006，
   // 表示可读可写（rw）。另外，其中 O_ACCMODE = 00003，是索引值 x 的屏蔽码。
24 #define ACC_MODE(x) ("\004\002\006\377"[(x)&O_ACCMODE])
25
26 /*
27  * comment out this line if you want names > NAME_LEN chars to be
28  * truncated. Else they will be disallowed.
29  */
    /*
    * 如果想让文件名长度 > NAME_LEN 个的字符被截掉，就将下面定义注释掉。
    */
30 /* #define NO_TRUNCATE */
```

```
31
32 #define MAY_EXEC 1          // 可执行(可进入)。
33 #define MAY_WRITE 2         // 可写。
34 #define MAY_READ 4          // 可读。
35
36 /*
37  *      permission()
38  *
39  * is used to check for read/write/execute permissions on a file.
40  * I don't know if we should look at just the euid or both euid and
41  * uid, but that should be easily changed.
42  */
   /*
    *     permission()
    *
    * 该函数用于检测一个文件的读/写/执行权限。我不知道是否只需检查 euid，
    * 还是需要检查 euid 和 uid 两者，不过这很容易修改。
    */
   //// 检测文件访问许可权限。
   // 参数：inode - 文件的 i 节点指针；mask - 访问属性屏蔽码。
   // 返回：访问许可返回 1，否则返回 0。
43 static int permission(struct m_inode * inode,int mask)
44 {
45         int mode = inode->i_mode;                    // 文件访问属性。
46
47 /* special case: not even root can read/write a deleted file */
   /* 特殊情况：即使是超级用户（root）也不能读/写一个已被删除的文件 */
   // 如果 i 节点有对应的设备，但该 i 节点的链接计数值等于 0，表示该文件已被删除，则返回。
   // 否则，如果进程的有效用户 id（euid）与 i 节点的用户 id 相同，则取文件宿主的访问权限。
   // 否则，如果进程的有效组 id（egid）与 i 节点的组 id 相同，则取组用户的访问权限。
48         if (inode->i_dev && !inode->i_nlinks)
49                 return 0;
50         else if (current->euid==inode->i_uid)
51                 mode >>= 6;
52         else if (in_group_p(inode->i_gid))
53                 mode >>= 3;
   // 最后判断如果所取的的访问权限与屏蔽码相同，或者是超级用户，则返回 1，否则返回 0。
54         if (((mode & mask & 0007) == mask) || suser())
55                 return 1;
56         return 0;
57 }
58
59 /*
60  * ok, we cannot use strncmp, as the name is not in our data space.
61  * Thus we'll have to use match. No big problem. Match also makes
62  * some sanity tests.
63  *
64  * NOTE! unlike strncmp, match returns 1 for success, 0 for failure.
65  */
   /*
    * ok，我们不能使用 strncmp 字符串比较函数，因为名称不在我们的数据空间
    * （不在内核空间）。 因而我们只能使用 match()。问题不大，match()同样
    * 也处理一些完整的测试。
```

```
 *
 * 注意！与 strncmp 不同的是 match()成功时返回 1，失败时返回 0。
 */
//// 指定长度字符串比较函数。
// 参数：len - 比较的字符串长度；name - 文件名指针；de - 目录项结构。
// 返回：相同返回 1，不同返回 0。
// 第 68 行上定义了一个局部寄存器变量 same。该变量将被保存在 eax 寄存器中，以便于高效
// 访问。
66 static int match(int len,const char * name,struct dir_entry * de)
67 {
68         register int same __asm__("ax");
69
// 首先判断函数参数的有效性。如果目录项指针空，或者目录项 i 节点等于 0，或者要比较的
// 字符串长度超过文件名长度，则返回 0（不匹配）。如果比较的长度 len 等于 0 并且目录项
// 中文件名的第 1 个字符是 '.'，并且只有这么一个字符，那么我们就认为是相同的，因此返
// 回 1（匹配）。如果要比较的长度 len 小于 NAME_LEN，但是目录项中文件名长度超过 len，
// 则也返回 0（不匹配）。
// 第 75 行上对目录项中文件名长度是否超过 len 的判断方法是检测 name[len] 是否为 NULL。
// 若长度超过 len，则 name[len]处就是一个不是 NULL 的普通字符。而对于长度为 len 的字符
// 串 name，字符 name[len]就应该是 NULL。
70         if (!de || !de->inode || len > NAME_LEN)
71                 return 0;
72         /* "" means "." ---> so paths like "/usr/lib//libc.a" work */
           /* "" 当作 "." 来看待 ---> 这样就能处理象 "/usr/lib//libc.a" 那样的路径名 */
73         if (!len && (de->name[0]=='.') && (de->name[1]=='\0'))
74                 return 1;
75         if (len < NAME_LEN && de->name[len])
76                 return 0;
// 然后使用嵌入汇编语句进行快速比较操作。它会在用户数据空间（fs 段）执行字符串的比较
// 操作。%0 - eax（比较结果 same）；%1 - eax（eax 初值 0）；%2 - esi（名字指针）；
// %3 - edi（目录项名指针）；%4 - ecx(比较的字节长度值 len)。
77         __asm__("cld\n\t"                       // 清方向标志位。
78                 "fs ; repe ; cmpsb\n\t"         // 用户空间执行循环比较[esi++]和[edi++]操作，
79                 "setz %%al"                     // 若比较结果一样（zf=0）则置 al=1（same=eax）。
80                 :"=a" (same)
81                 :"" (0),"S" ((long) name),"D" ((long) de->name),"c" (len)
82                 :"cx","di","si");
83         return same;                            // 返回比较结果。
84 }
85
86 /*
87  *      find_entry()
88  *
89  * finds an entry in the specified directory with the wanted name. It
90  * returns the cache buffer in which the entry was found, and the entry
91  * itself (as a parameter - res_dir). It does NOT read the inode of the
92  * entry - you'll have to do that yourself if you want to.
93  *
94  * This also takes care of the few special cases due to '..'-traversal
95  * over a pseudo-root and a mount point.
96  */
/*
 *      find_entry()
```

```
 *
 * 在指定目录中寻找一个与名字匹配的目录项。返回一个含有找到目录项的高速
 * 缓冲块以及目录项本身（作为一个参数 - res_dir）。该函数并不读取目录项
 * 的 i 节点 - 如果需要的话则自己操作。
 *
 * 由于有'..'目录项，因此在操作期间也会对几种特殊情况分别处理 - 比如横越
 * 一个伪根目录以及安装点。
 */
//// 查找指定目录和文件名的目录项。
// 参数：*dir - 指定目录 i 节点的指针；name - 文件名；namelen - 文件名长度；
// 该函数在指定目录的数据（文件）中搜索指定文件名的目录项。并对指定文件名是'..'的
// 情况根据当前进行的相关设置进行特殊处理。关于函数参数传递指针的指针的作用，请参
// 见 linux/sched.c 第 151 行前的注释。
// 返回：成功则函数高速缓冲区指针，并在*res_dir 处返回的目录项结构指针。失败则返回
// 空指针 NULL。
97  static struct buffer_head * find_entry(struct m_inode ** dir,
98          const char * name, int namelen, struct dir_entry ** res_dir)
99  {
100         int entries;
101         int block,i;
102         struct buffer_head * bh;
103         struct dir_entry * de;
104         struct super_block * sb;
105
// 同样，本函数一上来也需要对函数参数的有效性进行判断和验证。如果我们在前面第 30 行
// 定义了符号常数 NO_TRUNCATE，那么如果文件名长度超过最大长度 NAME_LEN，则不予处理。
// 如果没有定义过 NO_TRUNCATE，那么在文件名长度超过最大长度 NAME_LEN 时截短之。
106 #ifdef NO_TRUNCATE
107         if (namelen > NAME_LEN)
108                 return NULL;
109 #else
110         if (namelen > NAME_LEN)
111                 namelen = NAME_LEN;
112 #endif

// 首先计算本目录中目录项项数 entries。 目录 i 节点 i_size 字段中含有本目录包含的数据
// 长度，因此其除以一个目录项的长度（16 字节）即可得到该目录中目录项数。然后置空返回
// 目录项结构指针。
113         entries = (*dir)->i_size / (sizeof (struct dir_entry));
114         *res_dir = NULL;
// 接下来我们对目录项文件名是'..'的情况进行特殊处理。如果当前进程指定的根 i 节点就是
// 函数参数指定的目录，则说明对于本进程来说，这个目录就是它的伪根目录，即进程只能访
// 问该目录中的项而不能后退到其父目录中去。也即对于该进程本目录就如同是文件系统的根
// 目录。因此我们需要将文件名修改为'.'。
// 否则，如果该目录的 i 节点号等于 ROOT_INO（1 号）的话，说明确实是文件系统的根 i 节点。
// 则取文件系统的超级块。如果被安装到的 i 节点存在，则先放回原 i 节点，然后对被安装到
// 的 i 节点进行处理。于是我们让*dir 指向该被安装到的 i 节点；并且该 i 节点的引用数加 1。
// 即针对这种情况，我们悄悄地进行了“偷梁换柱”工程:)
115 /* check for '..', as we might have to do some "magic" for it */
/* 检查目录项 '..'，因为我们可能需要对其进行特殊处理 */
116         if (namelen==2 && get_fs_byte(name)=='.' && get_fs_byte(name+1)=='.') {
117 /* '..' in a pseudo-root results in a faked '.' (just change namelen) */
/* 伪根中的 '..' 如同一个假 '.'（只需改变名字长度） */
```

```
118                 if ((*dir) == current->root)
119                         namelen=1;
120                 else if ((*dir)->i_num == ROOT_INO) {
121 /* '..' over a mount-point results in 'dir' being exchanged for the mounted
122    directory-inode. NOTE! We set mounted, so that we can iput the new dir */
    /* 在一个安装点上的 '..' 将导致目录交换到被安装文件系统的目录 i 节点上。注意!
       由于我们设置了 mounted 标志，因而我们能够放回该新目录 */
123                         sb=get_super((*dir)->i_dev);
124                         if (sb->s_imount) {
125                                 iput(*dir);
126                                 (*dir)=sb->s_imount;
127                                 (*dir)->i_count++;
128                         }
129                 }
130         }

    // 现在我们开始正常操作，查找指定文件名的目录项在什么地方。因此我们需要读取目录的数
    // 据，即取出目录 i 节点对应块设备数据区中的数据块（逻辑块）信息。这些逻辑块的块号保
    // 存在 i 节点结构的 i_zone[9]数组中。我们先取其中第 1 个块号。如果目录 i 节点指向的第
    // 一个直接磁盘块号为 0，则说明该目录竟然不含数据，这不正常。于是返回 NULL 退出。否则
    // 我们就从节点所在设备读取指定的目录项数据块。当然，如果不成功，则也返回 NULL 退出。
131         if (!(block = (*dir)->i_zone[0]))
132                 return NULL;
133         if (!(bh = bread((*dir)->i_dev,block)))
134                 return NULL;

    // 此时我们就在这个读取的目录 i 节点数据块中搜索匹配指定文件名的目录项。首先让 de 指
    // 向缓冲块中的数据块部分，并在不超过目录中目录项数的条件下，循环执行搜索。其中 i 是
    // 目录中的目录项索引号，在循环开始时初始化为 0。
135         i = 0;
136         de = (struct dir_entry *) bh->b_data;
137         while (i < entries) {
    // 如果当前目录项数据块已经搜索完，还没有找到匹配的目录项，则释放当前目录项数据块。
    // 再读入目录的下一个逻辑块。若这块为空，则只要还没有搜索完目录中的所有目录项，就
    // 跳过该块，继续读目录的下一逻辑块。若该块不空，就让 de 指向该数据块，然后在其中
    // 继续搜索。其中 141 行上 i/DIR_ENTRIES_PER_BLOCK 可得到当前搜索的目录项所在目录文
    // 件中的块号，而 bmap()函数（inode.c，第 142 行）则可计算出在设备上对应的逻辑块号。
138                 if ((char *)de >= BLOCK_SIZE+bh->b_data) {
139                         brelse(bh);
140                         bh = NULL;
141                         if (!(block = bmap(*dir,i/DIR_ENTRIES_PER_BLOCK)) ||
142                             !(bh = bread((*dir)->i_dev,block))) {
143                                 i += DIR_ENTRIES_PER_BLOCK;
144                                 continue;
145                         }
146                         de = (struct dir_entry *) bh->b_data;
147                 }
    // 如果找到匹配的目录项的话，则返回该目录项结构指针 de 和该目录项 i 节点指针*dir 以
    // 及该目录项数据块指针 bh，并退出函数。否则继续在目录项数据块中比较下一个目录项。
148                 if (match(namelen,name,de)) {
149                         *res_dir = de;
150                         return bh;
151                 }
```

```
152                 de++;
153                 i++;
154         }
    // 如果指定目录中的所有目录项都搜索完后，还没有找到相应的目录项，则释放目录的数据
    // 块，最后返回 NULL（失败）。
155         brelse(bh);
156         return NULL;
157 }
158
159 /*
160  *      add_entry()
161  *
162  * adds a file entry to the specified directory, using the same
163  * semantics as find_entry(). It returns NULL if it failed.
164  *
165  * NOTE!! The inode part of 'de' is left at 0 - which means you
166  * may not sleep between calling this and putting something into
167  * the entry, as someone else might have used it while you slept.
168  */
    /*
     *     add_entry()
     * 使用与 find_entry()同样的方法，往指定目录中添加一指定文件名的目
     * 录项。如果失败则返回 NULL。
     *
     * 注意！！'de'（指定目录项结构指针）的 i 节点部分被设置为 0 - 这表
     * 示在调用该函数和往目录项中添加信息之间不能去睡眠。 因为如果睡眠，
     * 那么其他人(进程)可能会使用了该目录项。
     */
    //// 根据指定的目录和文件名添加目录项。
    // 参数：dir - 指定目录的 i 节点；name - 文件名；namelen - 文件名长度；
    // 返回：高速缓冲区指针；res_dir - 返回的目录项结构指针；
169 static struct buffer_head * add_entry(struct m_inode * dir,
170         const char * name, int namelen, struct dir_entry ** res_dir)
171 {
172         int block,i;
173         struct buffer_head * bh;
174         struct dir_entry * de;
175
    // 同样，本函数一上来也需要对函数参数的有效性进行判断和验证。如果我们在前面第 30 行
    // 定义了符号常数 NO_TRUNCATE，那么如果文件名长度超过最大长度 NAME_LEN，则不予处理。
    // 如果没有定义过 NO_TRUNCATE，那么在文件名长度超过最大长度 NAME_LEN 时截短之。
176         *res_dir = NULL;                    // 用于返回目录项结构指针。
177 #ifdef NO_TRUNCATE
178         if (namelen > NAME_LEN)
179                 return NULL;
180 #else
181         if (namelen > NAME_LEN)
182                 namelen = NAME_LEN;
183 #endif
    // 现在我们开始操作，向指定目录中添加一个指定文件名的目录项。因此我们需要先读取目录
    // 的数据，即取出目录 i 节点对应块设备数据区中的数据块（逻辑块）信息。这些逻辑块的块
    // 号保存在 i 节点结构的 i_zone[9]数组中。我们先取其中第 1 个块号。如果目录 i 节点指向
    // 的第一个直接磁盘块号为 0，则说明该目录竟然不含数据，这不正常。于是返回 NULL 退出。
```

```
    // 否则我们就从节点所在设备读取指定的目录项数据块。当然，如果不成功，则也返回 NULL
    // 退出。另外，如果参数提供的文件名长度等于 0，则也返回 NULL 退出。
184         if (!namelen)
185                 return NULL;
186         if (!(block = dir->i_zone[0]))
187                 return NULL;
188         if (!(bh = bread(dir->i_dev,block)))
189                 return NULL;
    // 此时我们就在这个目录 i 节点数据块中循环查找最后未使用的空目录项。首先让目录项结构
    // 指针 de 指向缓冲块中的数据块部分，即第一个目录项处。其中 i 是目录中的目录项索引号，
    // 在循环开始时初始化为 0。
190         i = 0;
191         de = (struct dir_entry *) bh->b_data;
192         while (1) {
    // 如果当前目录项数据块已经搜索完毕，但还没有找到需要的空目录项，则释放当前目录项数
    // 据块，再读入目录的下一个逻辑块。如果对应的逻辑块不存在就创建一块。若读取或创建操
    // 作失败则返回空。如果此次读取的磁盘逻辑块数据返回的缓冲块指针为空，说明这块逻辑块
    // 可能是因为不存在而新创建的空块，则把目录项索引值加上一块逻辑块所能容纳的目录项数
    // DIR_ENTRIES_PER_BLOCK，用以跳过该块并继续搜索。否则说明新读入的块上有目录项数据，
    // 于是让目录项结构指针 de 指向该块的缓冲块数据部分，然后在其中继续搜索。其中 196 行
    // 上的 i/DIR_ENTRIES_PER_BLOCK 可计算得到当前搜索的目录项 i 所在目录文件中的块号，
    // 而 create_block()函数（inode.c，第 147 行）则可读取或创建出在设备上对应的逻辑块。
193                 if ((char *)de >= BLOCK_SIZE+bh->b_data) {
194                         brelse(bh);
195                         bh = NULL;
196                         block = create_block(dir,i/DIR_ENTRIES_PER_BLOCK);
197                         if (!block)
198                                 return NULL;
199                         if (!(bh = bread(dir->i_dev,block))) { // 若空则跳过该块继续。
200                                 i += DIR_ENTRIES_PER_BLOCK;
201                                 continue;
202                         }
203                         de = (struct dir_entry *) bh->b_data;
204                 }
    // 如果当前所操作的目录项序号 i 乘上目录结构大小所的长度值已经超过了该目录 i 节点信息
    // 所指出的目录数据长度值 i_size ，则说明整个目录文件数据中没有由于删除文件留下的空
    // 目录项，因此我们只能把需要添加的新目录项附加到目录文件数据的末端处。于是对该处目
    // 录项进行设置（置该目录项的 i 节点指针为空），并更新该目录文件的长度值（加上一个目
    // 录项的长度），然后设置目录的 i 节点已修改标志，再更新该目录的改变时间为当前时间。
205                 if (i*sizeof(struct dir_entry) >= dir->i_size) {
206                         de->inode=0;
207                         dir->i_size = (i+1)*sizeof(struct dir_entry);
208                         dir->i_dirt = 1;
209                         dir->i_ctime = CURRENT_TIME;
210                 }
    // 若当前搜索的目录项 de 的 i 节点为空，则表示找到一个还未使用的空闲目录项或是添加的
    // 新目录项。于是更新目录的修改时间为当前时间，并从用户数据区复制文件名到该目录项的
    // 文件名字段，置含有本目录项的相应高速缓冲块已修改标志。返回该目录项的指针以及该高
    // 速缓冲块的指针，退出。
211                 if (!de->inode) {
212                         dir->i_mtime = CURRENT_TIME;
213                         for (i=0; i < NAME_LEN ; i++)
214                                 de->name[i]=(i<namelen)?get_fs_byte(name+i):0;
```

```
215                         bh->b_dirt = 1;
216                         *res_dir = de;
217                         return bh;
218                 }
219                 de++;           // 如果该目录项已经被使用，则继续检测下一个目录项。
220                 i++;
221         }
    // 本函数执行不到这里。这也许是 Linus 在写这段代码时，先复制了上面 find_entry()函数
    // 的代码，而后修改成本函数的☺。
222         brelse(bh);
223         return NULL;
224 }
225
    //// 查找符号链接的 i 节点。
    // 参数：dir - 目录 i 节点；inode - 目录项 i 节点。
    // 返回：返回符号链接到文件的 i 节点指针。出错返回 NULL。
226 static struct m_inode * follow_link(struct m_inode * dir, struct m_inode * inode)
227 {
228         unsigned short fs;                      // 用于临时保存 fs 段寄存器值。
229         struct buffer_head * bh;
230
    // 首先判断函数参数的有效性。如果没有给出目录 i 节点，我们就使用进程任务结构中设置的
    // 根 i 节点，并把链接数增 1。如果没有给出目录项 i 节点，则放回目录 i 节点后返回 NULL。
    // 如果指定目录项不是一个符号链接，就直接返回目录项对应的 i 节点 inode。
231         if (!dir) {
232                 dir = current->root;
233                 dir->i_count++;
234         }
235         if (!inode) {
236                 iput(dir);
237                 return NULL;
238         }
239         if (!S_ISLNK(inode->i_mode)) {
240                 iput(dir);
241                 return inode;
242         }
    // 然后取 fs 段寄存器值。fs 通常保存着指向任务数据段的选择符 0x17。如果 fs 没有指向用户
    // 数据段，或者给出的目录项 i 节点第 1 个直接块块号等于 0，或者是读取第 1 个直接块出错，
    // 则放回 dir 和 inode 两个 i 节点并返回 NULL 退出。否则说明现在 fs 正指向用户数据段、并且
    // 我们已经成功地读取了这个符号链接目录项的文件内容，并且文件内容已经在 bh 指向的缓冲
    // 块数据区中。实际上，这个缓冲块数据区中仅包含一个链接指向的文件路径名字符串。
243         __asm__("mov %%fs,%0":"=r" (fs));
244         if (fs != 0x17 || !inode->i_zone[0] ||
245            !(bh = bread(inode->i_dev, inode->i_zone[0]))) {
246                 iput(dir);
247                 iput(inode);
248                 return NULL;
249         }
    // 此时我们已经不需要符号链接目录项的 i 节点了，于是把它放回。现在碰到一个问题，那就
    // 是内核函数处理的用户数据都是存放在用户数据空间中的，并使用了 fs 段寄存器来从用户
    // 空间传递数据到内核空间中。而这里需要处理的数据却在内核空间中。因此为了正确地处理
    // 位于内核中的用户数据，我们需要让 fs 段寄存器临时指向内核空间，即让 fs =0x10。并在
    // 调用函数处理完后再恢复原 fs 的值。最后释放相应缓冲块，并返回 _namei()解析得到的符
```

```
    // 号链接指向的文件 i 节点。
250         iput(inode);
251         __asm__("mov %0,%%fs"::"r" ((unsigned short) 0x10));
252         inode = _namei(bh->b_data,dir,0);
253         __asm__("mov %0,%%fs"::"r" (fs));
254         brelse(bh);
255         return inode;
256 }
257
258 /*
259  *      get_dir()
260  *
261  * Getdir traverses the pathname until it hits the topmost directory.
262  * It returns NULL on failure.
263  */
    /*
     *     get_dir()
     *
     * 该函数根据给出的路径名进行搜索，直到达到最顶端的目录。
     * 如果失败则返回 NULL。
     */
    //// 从指定目录开始搜寻指定路径名的目录（或文件名）的 i 节点。
    // 参数：pathname - 路径名；inode - 指定起始目录的 i 节点。
    // 返回：目录或文件的 i 节点指针。失败时返回 NULL。
264 static struct m_inode * get_dir(const char * pathname, struct m_inode * inode)
265 {
266         char c;
267         const char * thisname;
268         struct buffer_head * bh;
269         int namelen,inr;
270         struct dir_entry * de;
271         struct m_inode * dir;
272
    // 首先判断参数有效性。如果给出的指定目录的 i 节点指针 inode 为空，则使用当前进程的当
    // 前工作目录 i 节点。如果用户指定路径名的第 1 个字符是'/'，则说明路径名是绝对路径名。
    // 则应该从当前进程任务结构中设置的根（或伪根）i 节点开始操作。于是我们需要先放回参
    // 数指定的或者设定的目录 i 节点，并取得进程使用的根 i 节点。然后把该 i 节点的引用计数
    // 加 1，并删除路径名的第 1 个字符 '/'。这样就可以保证当前进程只能以其设定的根 i 节点
    // 作为搜索的起点。
273         if (!inode) {
274                 inode = current->pwd;                   // 进程的当前工作目录 i 节点。
275                 inode->i_count++;
276         }
277         if ((c=get_fs_byte(pathname))=='/') {
278                 iput(inode);                            // 放回原 i 节点。
279                 inode = current->root;                  // 为进程指定的根 i 节点。
280                 pathname++;
281                 inode->i_count++;
282         }
    // 然后针对路径名中的各个目录名部分和文件名进行循环处理。在循环处理过程中，我们先要
    // 对当前正在处理的目录名部分的 i 节点进行有效性判断，并且把变量 thisname 指向当前正
    // 在处理的目录名部分。如果该 i 节点表明当前处理的目录名部分不是目录类型，或者没有可
    // 进入该目录的访问许可，则放回该 i 节点，并返回 NULL 退出。 当然在刚进入循环时，当前
```

```
    // 目录的 i 节点 inode 就是进程根 i 节点或者是当前工作目录的 i 节点，或者是参数指定的某
    // 个搜索起始目录的 i 节点。
283         while (1) {
284                 thisname = pathname;
285                 if (!S_ISDIR(inode->i_mode) || !permission(inode,MAY_EXEC)) {
286                         iput(inode);
287                         return NULL;
288                 }
    // 每次循环我们处理路径名中一个目录名（或文件名）部分。因此在每次循环中我们都要从路
    // 径名字符串中分离出一个目录名（或文件名）。方法是从当前路径名指针 pathname 开始处
    // 搜索检测字符，直到字符是一个结尾符（NULL）或者是一个'/'字符。此时变量 namelen 正
    // 好是当前处理目录名部分的长度，而变量 thisname 正指向该目录名部分的开始处。此时如
    // 果字符是结尾符 NULL，则表明已经搜索到路径名末尾，并已到达最后指定目录名或文件名，
    // 则返回该 i 节点指针退出。
    // 注意！如果路径名中最后一个名称也是一个目录名，但其后面没有加上 '/'字符，则函数不
    // 会返回该最后目录名的 i 节点！例如：对于路径名/usr/src/linux，该函数将只返回 src/目
    // 录名的 i 节点。
289                 for(namelen=0;(c=get_fs_byte(pathname++))&&(c!='/');namelen++)
290                         /* nothing */ ;
291                 if (!c)
292                         return inode;
    // 在得到当前目录名部分（或文件名）后，我们调用查找目录项函数 find_entry()在当前处
    // 理的目录中寻找指定名称的目录项。如果没有找到，则放回该 i 节点，并返回 NULL 退出。
    // 然后在找到的目录项中取出其 i 节点号 inr 和设备号 idev，释放包含该目录项的高速缓冲
    // 块并放回该 i 节点。 然后取节点号 inr 的 i 节点 inode，并以该目录项为当前目录继续循
    // 环处理路径名中的下一目录名部分（或文件名）。如果当前处理的目录项是一个符号链接
    // 名，则使用 follow_link()就可以得到其指向的目录项名的 i 节点。
293                 if (!(bh = find_entry(&inode,thisname,namelen,&de))) {
294                         iput(inode);
295                         return NULL;
296                 }
297                 inr = de->inode;                        // 当前目录名部分的 i 节点号。
298                 brelse(bh);
299                 dir = inode;
300                 if (!(inode = iget(dir->i_dev,inr))) {      // 取 i 节点内容。
301                         iput(dir);
302                         return NULL;
303                 }
304                 if (!(inode = follow_link(dir,inode)))
305                         return NULL;
306         }
307 }
308
309 /*
310  *      dir_namei()
311  *
312  * dir_namei() returns the inode of the directory of the
313  * specified name, and the name within that directory.
314  */
    /*
     *     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。
     */
    // 参数：pathname - 目录路径名；namelen - 路径名长度；name - 返回的最顶层目录名。
    // base - 搜索起始目录的 i 节点。
    // 返回：指定目录名最顶层目录的 i 节点指针和最顶层目录名称及长度。出错时返回 NULL。
    // 注意！！这里“最顶层目录”是指路径名中最靠近末端的目录。
315 static struct m_inode * dir_namei(const char * pathname,
316         int * namelen, const char ** name, struct m_inode * base)
317 {
318         char c;
319         const char * basename;
320         struct m_inode * dir;
321
    // 首先取得指定路径名最顶层目录的 i 节点。然后对路径名 pathname 进行搜索检测，查出最后
    // 一个'/'字符后面的名字字符串，计算其长度，并且返回最顶层目录的 i 节点指针。注意！如
    // 果路径名最后一个字符是斜杠字符'/'，那么返回的目录名为空，并且长度为 0。但返回的 i
    // 节点指针仍然指向最后一个'/'字符前目录名的 i 节点。参见第 289 行上的“注意”说明。
322         if (!(dir = get_dir(pathname,base)))      // base 是指定的起始目录 i 节点。
323                 return NULL;
324         basename = pathname;
325         while (c=get_fs_byte(pathname++))
326                 if (c=='/')
327                         basename=pathname;
328         *namelen = pathname-basename-1;
329         *name = basename;
330         return dir;
331 }
332
    //// 取指定路径名的 i 节点内部函数。
    // 参数：pathname - 路径名；base - 搜索起点目录 i 节点；follow_links - 是否跟随
    // 符号链接的标志，1 - 需要，0 不需要。
    // 返回：对应的 i 节点。
333 struct m_inode * _namei(const char * pathname, struct m_inode * base,
334         int follow_links)
335 {
336         const char * basename;
337         int inr,namelen;
338         struct m_inode * inode;
339         struct buffer_head * bh;
340         struct dir_entry * de;
341
    // 首先查找指定路径名中最顶层目录的目录名并得到其 i 节点。若不存在，则返回 NULL 退出。
    // 如果返回的最顶层名字的长度是 0，则表示该路径名以一个目录名为最后一项。因此说明我
    // 们已经找到对应目录的 i 节点，可以直接返回该 i 节点退出。如果返回的名字长度不是 0，
    // 则我们以指定的起始目录 base，再次调用 dir_namei()函数来搜索顶层目录名，并根据返回
    // 的信息作类似判断。
311         if (!(dir = dir_namei(pathname,&namelen,&basename)))
312                 return NULL;
313         if (!namelen)                   /* special case: '/usr/' etc */
314                 return dir;             /* 对应于'/usr/'等情况 */
342         if (!(base = dir_namei(pathname,&namelen,&basename,base)))
343                 return NULL;
344         if (!namelen)                   /* special case: '/usr/' etc */
```

```
345                 return base;
    // 然后在返回的顶层目录中寻找指定文件名目录项的 i 节点。注意！因为如果最后也是一个目
    // 录名，但其后没有加'/'，则不会返回该最后目录的 i 节点！ 例如：/usr/src/linux，将只
    // 返回 src/目录名的 i 节点。因为函数 dir_namei() 将不以'/'结束的最后一个名字当作一个
    // 文件名来看待，因此这里需要单独对这种情况使用寻找目录项 i 节点函数 find_entry()进行
    // 处理。此时 de 中含有寻找到的目录项指针，而 dir 是包含该目录项的目录的 i 节点指针。
346         bh = find_entry(&base,basename,namelen,&de);
347         if (!bh) {
348                 iput(base);
349                 return NULL;
350         }
    // 接着取该目录项的 i 节点号，并释放包含该目录项的高速缓冲块并放回目录 i 节点。然后取
    // 对应节点号的 i 节点，修改其被访问时间为当前时间，并置已修改标志。最后返回该 i 节点
    // 指针 inode。如果当前处理的目录项是一个符号链接名，则使用 follow_link()得到其指向的
    // 目录项名的 i 节点。
351         inr = de->inode;
352         brelse(bh);
353         if (!(inode = iget(base->i_dev,inr))) {
354                 iput(base);
355                 return NULL;
356         }
357         if (follow_links)
358                 inode = follow_link(base,inode);
359         else
360                 iput(base);
361         inode->i_atime=CURRENT_TIME;
362         inode->i_dirt=1;
363         return inode;
364 }
365
    //// 取指定路径名的 i 节点，不跟随符号链接。
    // 参数：pathname - 路径名。
    // 返回：对应的 i 节点。
366 struct m_inode * lnamei(const char * pathname)
367 {
368         return _namei(pathname, NULL, 0);
369 }
370
371 /*
372  *      namei()
373  *
374  * is used by most simple commands to get the inode of a specified name.
375  * Open, link etc use their own routines, but this is enough for things
376  * like 'chmod' etc.
377  */
    /*
     *     namei()
     *
     * 该函数被许多简单命令用于取得指定路径名称的 i 节点。open、link 等则使用它们
     * 自己的相应函数。但对于象修改模式'chmod'等这样的命令，该函数已足够用了。
     */
    //// 取指定路径名的 i 节点，跟随符号链接。
    // 参数：pathname - 路径名。
```

```
    // 返回：对应的 i 节点。
378 struct m_inode * namei(const char * pathname)
379 {
380         return _namei(pathname,NULL,1);
381 }
382
383 /*
384  *      open_namei()
385  *
386  * namei for open - this is in fact almost the whole open-routine.
387  */
    /*
     *     open_namei()
     *
     * open()函数使用的 namei 函数 - 这其实几乎是完整的打开文件程序。
     */
    //// 文件打开 namei 函数。
    // 参数 filename 是文件路径名，flag 是打开文件标志，可取值 O_RDONLY（只读）、O_WRONLY
    // （只写）或 O_RDWR（读写），以及 O_CREAT（创建）、O_EXCL（被创建文件必须不存在）、
    // O_APPEND（在文件尾添加数据）等其他一些标志的组合。如果本调用创建了一个新文件，则
    // mode 就用于指定文件的许可属性。这些属性有 S_IRWXU（文件宿主具有读、写和执行权限）、
    // S_IRUSR（用户具有读文件权限）、S_IRWXG（组成员具有读、写和执行权限）等等。对于新
    // 创建的文件，这些属性只应用于将来对文件的访问，创建了只读文件的打开调用也将返回一
    // 个可读写的文件句柄。参见包含文件 sys/stat.h、fcntl.h。
    // 返回：成功返回 0，否则返回出错码；res_inode - 返回对应文件路径名的 i 节点指针。
388 int open_namei(const char * pathname, int flag, int mode,
389         struct m_inode ** res_inode)
390 {
391         const char * basename;
392         int inr,dev,namelen;
393         struct m_inode * dir, *inode;
394         struct buffer_head * bh;
395         struct dir_entry * de;
396
    // 首先对函数参数进行合理的处理。如果文件访问模式标志是只读（0），但是文件截零标志
    // O_TRUNC 却置位了，则在文件打开标志中添加只写标志 O_WRONLY。这样做的原因是由于截零
    // 标志 O_TRUNC 必须在文件可写情况下才有效。然后使用当前进程的文件访问许可屏蔽码，屏
    // 蔽掉给定模式中的相应位，并添上普通文件标志 I_REGULAR。该标志将用于打开的文件不存
    // 在而需要创建文件时，作为新文件的默认属性。参见下面 411 行上的注释。
397         if ((flag & O_TRUNC) && !(flag & O_ACCMODE))
398                 flag |= O_WRONLY;
399         mode &= 0777 & ~current->umask;
400         mode |= I_REGULAR;              // 常规文件标志。见参见 include/const.h 文件）。
    // 然后根据指定的路径名寻找到对应的 i 节点，以及最顶端目录名及其长度。此时如果最顶端
    // 目录名长度为 0（ 例如'/usr/' 这种路径名的情况），那么若操作不是读写、创建和文件长
    // 度截 0，则表示是在打开一个目录名文件操作。于是直接返回该目录的 i 节点并返回 0 退出。
    // 否则说明进程操作非法，于是放回该 i 节点，返回出错码。
401         if (!(dir = dir_namei(pathname,&namelen,&basename,NULL)))
402                 return -ENOENT;
403         if (!namelen) {                 /* special case: '/usr/' etc */
404                 if (!(flag & (O_ACCMODE|O_CREAT|O_TRUNC))) {
405                         *res_inode=dir;
406                         return 0;
```

```
407                 }
408                 iput(dir);
409                 return -EISDIR;
410         }
    // 接着根据上面得到的最顶层目录名的 i 节点 dir，在其中查找取得路径名字符串中最后的文
    // 件名对应的目录项结构 de，并同时得到该目录项所在的高速缓冲区指针。 如果该高速缓冲
    // 指针为 NULL，则表示没有找到对应文件名的目录项，因此只可能是创建文件操作。 此时如
    // 果不是创建文件，则放回该目录的 i 节点，返回出错号退出。如果用户在该目录没有写的权
    // 力，则放回该目录的 i 节点，返回出错号退出。
411         bh = find_entry(&dir,basename,namelen,&de);
412         if (!bh) {
413                 if (!(flag & O_CREAT)) {
414                         iput(dir);
415                         return -ENOENT;
416                 }
417                 if (!permission(dir,MAY_WRITE)) {
418                         iput(dir);
419                         return -EACCES;
420                 }
    // 现在我们确定了是创建操作并且有写操作许可。 因此我们就在目录 i 节点对应设备上申请
    // 一个新的 i 节点给路径名上指定的文件使用。 若失败则放回目录的 i 节点，并返回没有空
    // 间出错码。否则使用该新 i 节点，对其进行初始设置：置节点的用户 id；对应节点访问模
    // 式；置已修改标志。然后并在指定目录 dir 中添加一个新目录项。
421                 inode = new_inode(dir->i_dev);
422                 if (!inode) {
423                         iput(dir);
424                         return -ENOSPC;
425                 }
426                 inode->i_uid = current->euid;
427                 inode->i_mode = mode;
428                 inode->i_dirt = 1;
429                 bh = add_entry(dir,basename,namelen,&de);
    // 如果返回的应该含有新目录项的高速缓冲区指针为 NULL，则表示添加目录项操作失败。于是
    // 将该新 i 节点的引用连接计数减 1，放回该 i 节点与目录的 i 节点并返回出错码退出。 否则
    // 说明添加目录项操作成功。 于是我们来设置该新目录项的一些初始值：置 i 节点号为新申请
    // 到的 i 节点的号码；并置高速缓冲区已修改标志。 然后释放该高速缓冲区，放回目录的 i 节
    // 点。返回新目录项的 i 节点指针，并成功退出。
430                 if (!bh) {
431                         inode->i_nlinks--;
432                         iput(inode);
433                         iput(dir);
434                         return -ENOSPC;
435                 }
436                 de->inode = inode->i_num;
437                 bh->b_dirt = 1;
438                 brelse(bh);
439                 iput(dir);
440                 *res_inode = inode;
441                 return 0;
442         }
    // 若上面（411 行）在目录中取文件名对应目录项结构的操作成功（即 bh 不为 NULL），则说
    // 明指定打开的文件已经存在。于是取出该目录项的 i 节点号和其所在设备号，并释放该高速
    // 缓冲区以及放回目录的 i 节点。如果此时独占操作标志 O_EXCL 置位，但现在文件已经存在，
```

```
    // 则返回文件已存在出错码退出。
443         inr = de->inode;
444         dev = dir->i_dev;
445         brelse(bh);
446         if (flag & O_EXCL) {
447                 iput(dir);
448                 return -EEXIST;
449         }
    // 然后我们读取该目录项的 i 节点内容。若该 i 节点是一个目录的 i 节点并且访问模式是只
    // 写或读写，或者没有访问的许可权限，则放回该 i 节点，返回访问权限出错码退出。
450         if (!(inode = follow_link(dir,iget(dev,inr))))
451                 return -EACCES;
452         if ((S_ISDIR(inode->i_mode) && (flag & O_ACCMODE)) ||
453             !permission(inode,ACC_MODE(flag))) {
454                 iput(inode);
455                 return -EPERM;
456         }
    // 接着我们更新该 i 节点的访问时间字段值为当前时间。如果设立了截 0 标志，则将该 i 节
    // 点的文件长度截为 0。最后返回该目录项 i 节点的指针，并返回 0（成功）。
457         inode->i_atime = CURRENT_TIME;
458         if (flag & O_TRUNC)
459                 truncate(inode);
460         *res_inode = inode;
461         return 0;
462 }
463
    //// 创建一个设备特殊文件或普通文件节点（node）。
    // 该函数创建名称为 filename，由 mode 和 dev 指定的文件系统节点（普通文件、设备特殊文
    // 件或命名管道）。
    // 参数：filename - 路径名；mode - 指定使用许可以及所创建节点的类型；dev - 设备号。
    // 返回：成功则返回 0，否则返回出错码。
464 int sys_mknod(const char * filename, int mode, int dev)
465 {
466         const char * basename;
467         int namelen;
468         struct m_inode * dir, * inode;
469         struct buffer_head * bh;
470         struct dir_entry * de;
471
    // 首先检查操作许可和参数的有效性并取路径名中顶层目录的 i 节点。如果不是超级用户，则
    // 返回访问许可出错码。如果找不到对应路径名中顶层目录的 i 节点，则返回出错码。如果最
    // 顶端的文件名长度为 0，则说明给出的路径名最后没有指定文件名，放回该目录 i 节点，返
    // 回出错码退出。如果在该目录中没有写的权限，则放回该目录的 i 节点，返回访问许可出错
    // 码退出。如果不是超级用户，则返回访问许可出错码。
472         if (!suser())
473                 return -EPERM;
474         if (!(dir = dir_namei(filename,&namelen,&basename, NULL)))
475                 return -ENOENT;
476         if (!namelen) {
477                 iput(dir);
478                 return -ENOENT;
479         }
480         if (!permission(dir,MAY_WRITE)) {
```

```
481                 iput(dir);
482                 return -EPERM;
483         }
    // 然后我们搜索一下路径名指定的文件是否已经存在。若已经存在则不能创建同名文件节点。
    // 如果对应路径名上最后的文件名的目录项已经存在，则释放包含该目录项的缓冲区块并放回
    // 目录的 i 节点，返回文件已经存在的出错码退出。
484         bh = find_entry(&dir,basename,namelen,&de);
485         if (bh) {
486                 brelse(bh);
487                 iput(dir);
488                 return -EEXIST;
489         }
    // 否则我们就申请一个新的 i 节点，并设置该 i 节点的属性模式。如果要创建的是块设备文件
    // 或者是字符设备文件，则令 i 节点的直接逻辑块指针 0 等于设备号。即对于设备文件来说，
    // 其 i 节点的 i_zone[0]中存放的是该设备文件所定义设备的设备号。然后设置该 i 节点的修
    // 改时间、访问时间为当前时间，并设置 i 节点已修改标志。
490         inode = new_inode(dir->i_dev);
491         if (!inode) {           // 若不成功则放回目录 i 节点，返回无空间出错码退出。
492                 iput(dir);
493                 return -ENOSPC;
494         }
495         inode->i_mode = mode;
496         if (S_ISBLK(mode) || S_ISCHR(mode))
497                 inode->i_zone[0] = dev;
498         inode->i_mtime = inode->i_atime = CURRENT_TIME;
499         inode->i_dirt = 1;
    // 接着为这个新的 i 节点在目录中新添加一个目录项。如果失败（包含该目录项的高速缓冲
    // 块指针为 NULL），则放回目录的 i 节点；把所申请的 i 节点引用连接计数复位，并放回该
    // i 节点，返回出错码退出。
500         bh = add_entry(dir,basename,namelen,&de);
501         if (!bh) {
502                 iput(dir);
503                 inode->i_nlinks=0;
504                 iput(inode);
505                 return -ENOSPC;
506         }
    // 现在添加目录项操作也成功了，于是我们来设置这个目录项内容。令该目录项的 i 节点字
    // 段等于新 i 节点号，并置高速缓冲区已修改标志，放回目录和新的 i 节点，释放高速缓冲
    // 区，最后返回 0(成功)。
507         de->inode = inode->i_num;
508         bh->b_dirt = 1;
509         iput(dir);
510         iput(inode);
511         brelse(bh);
512         return 0;
513 }
514
    ////  创建一个目录。
    // 参数：pathname - 路径名；mode - 目录使用的权限属性。
    // 返回：成功则返回 0，否则返回出错码。
515 int sys_mkdir(const char * pathname, int mode)
516 {
517         const char * basename;
```

```
518         int namelen;
519         struct m_inode * dir, * inode;
520         struct buffer_head * bh, *dir_block;
521         struct dir_entry * de;
522
    // 首先检查参数的有效性并取路径名中顶层目录的 i 节点。如果找不到对应路径名中顶层目录
    // 的 i 节点，则返回出错码。如果最顶端的文件名长度为 0，则说明给出的路径名最后没有指
    // 定文件名，放回该目录 i 节点，返回出错码退出。如果在该目录中没有写的权限，则放回该
    // 目录的 i 节点，返回访问许可出错码退出。如果不是超级用户，则返回访问许可出错码。
523         if (!(dir = dir_namei(pathname,&namelen,&basename, NULL)))
524                 return -ENOENT;
525         if (!namelen) {
526                 iput(dir);
527                 return -ENOENT;
528         }
529         if (!permission(dir,MAY_WRITE)) {
530                 iput(dir);
531                 return -EPERM;
532         }
    // 然后我们搜索一下路径名指定的目录名是否已经存在。若已经存在则不能创建同名目录节点。
    // 如果对应路径名上最后的目录名的目录项已经存在，则释放包含该目录项的缓冲区块并放回
    // 目录的 i 节点，返回文件已经存在的出错码退出。否则我们就申请一个新的 i 节点，并设置
    // 该 i 节点的属性模式：置该新 i 节点对应的文件长度为 32 字节 （2 个目录项的大小）、置
    // 节点已修改标志，以及节点的修改时间和访问时间。2 个目录项分别用于'.'和'..'目录。
533         bh = find_entry(&dir,basename,namelen,&de);
534         if (bh) {
535                 brelse(bh);
536                 iput(dir);
537                 return -EEXIST;
538         }
539         inode = new_inode(dir->i_dev);
540         if (!inode) {               // 若不成功则放回目录的 i 节点，返回无空间出错码。
541                 iput(dir);
542                 return -ENOSPC;
543         }
544         inode->i_size = 32;
545         inode->i_dirt = 1;
546         inode->i_mtime = inode->i_atime = CURRENT_TIME;
    // 接着为该新 i 节点申请一用于保存目录项数据的磁盘块，并令 i 节点的第一个直接块指针等
    // 于该块号。如果申请失败则放回对应目录的 i 节点；复位新申请的 i 节点连接计数；放回该
    // 新的 i 节点，返回没有空间出错码退出。否则置该新的 i 节点已修改标志。
547         if (!(inode->i_zone[0]=new_block(inode->i_dev))) {
548                 iput(dir);
549                 inode->i_nlinks--;
550                 iput(inode);
551                 return -ENOSPC;
552         }
553         inode->i_dirt = 1;
    // 从设备上读取新申请的磁盘块（目的是把对应块放到高速缓冲区中）。若出错，则放回对应
    // 目录的 i 节点；释放申请的磁盘块；复位新申请的 i 节点连接计数；放回该新的 i 节点，返
    // 回没有空间出错码退出。
554         if (!(dir_block=bread(inode->i_dev,inode->i_zone[0]))) {
555                 iput(dir);
```

```
556                 inode->i_nlinks--;
557                 iput(inode);
558                 return -ERROR;
559         }
    // 然后我们在缓冲块中建立起所创建目录文件中的 2 个默认的新目录项（'.'和'..'）结构数
    // 据。首先令 de 指向存放目录项的数据块，然后置该目录项的 i 节点号字段等于新申请的 i
    // 节点号，名字字段等于"."。 然后 de 指向下一个目录项结构，并在该结构中存放上级目录
    // 的 i 节点号和名字".."。然后设置该高速缓冲块已修改标志，并释放该缓冲块。再初始化
    // 设置新 i 节点的模式字段，并置该 i 节点已修改标志。
560         de = (struct dir_entry *) dir_block->b_data;
561         de->inode=inode->i_num;                     // 设置'.'目录项。
562         strcpy(de->name,".");
563         de++;
564         de->inode = dir->i_num;                     // 设置'..'目录项。
565         strcpy(de->name,"..");
566         inode->i_nlinks = 2;
567         dir_block->b_dirt = 1;
568         brelse(dir_block);
569         inode->i_mode = I_DIRECTORY | (mode & 0777 & ~current->umask);
570         inode->i_dirt = 1;
    // 现在我们在指定目录中新添加一个目录项，用于存放新建目录的 i 节点和目录名。如果失
    // 败（包含该目录项的高速缓冲区指针为 NULL），则放回目录的 i 节点；所申请的 i 节点引
    // 用连接计数复位，并放回该 i 节点。返回出错码退出。
571         bh = add_entry(dir,basename,namelen,&de);
572         if (!bh) {
573                 iput(dir);
574                 inode->i_nlinks=0;
575                 iput(inode);
576                 return -ENOSPC;
577         }
    // 最后令该新目录项的 i 节点字段等于新 i 节点号，并置高速缓冲块已修改标志，放回目录
    // 和新的 i 节点，释放高速缓冲区，最后返回 0（成功）。
578         de->inode = inode->i_num;
579         bh->b_dirt = 1;
580         dir->i_nlinks++;
581         dir->i_dirt = 1;
582         iput(dir);
583         iput(inode);
584         brelse(bh);
585         return 0;
586 }
587
588 /*
589  * routine to check that the specified directory is empty (for rmdir)
590  */
    /*
     * 用于检查指定的目录是否为空的子程序（用于 rmdir 系统调用）。
     */
    //// 检查指定目录是否空。
    // 参数：inode - 指定目录的 i 节点指针。
    // 返回：1 - 目录中是空的；0 - 不空。
591 static int empty_dir(struct m_inode * inode)
592 {
```

```
593         int nr,block;
594         int len;
595         struct buffer_head * bh;
596         struct dir_entry * de;
597
    // 首先计算指定目录中现有目录项个数并检查开始两个特定目录项中信息是否正确。一个目录
    // 中应该起码有 2 个目录项：即"."和".."。 如果目录项个数少于 2 个或者该目录 i 节点的第
    // 1 个直接块没有指向任何磁盘块号，或者该直接块读不出，则显示警告信息“设备 dev 上目
    // 录错”，返回 0(失败)。
598         len = inode->i_size / sizeof (struct dir_entry);       // 目录中目录项个数。
599         if (len<2 || !inode->i_zone[0] ||
600             !(bh=bread(inode->i_dev,inode->i_zone[0]))) {
601                 printk("warning - bad directory on dev %04x\n",inode->i_dev);
602                 return 0;
603         }
    // 此时 bh 所指缓冲块中含有目录项数据。我们让目录项指针 de 指向缓冲块中第 1 个目录项。
    // 对于第 1 个目录项（"."），它的 i 节点号字段 inode 应该等于当前目录的 i 节点号。对于
    // 第 2 个目录项（".."），它的 i 节点号字段 inode 应该等于上一层目录的 i 节点号，不会
    // 为 0。因此如果第 1 个目录项的 i 节点号字段值不等于该目录的 i 节点号，或者第 2 个目录
    // 项的 i 节点号字段为零，或者两个目录项的名字字段不分别等于"."和".."，则显示出错警
    // 告信息“设备 dev 上目录错”，并返回 0。
604         de = (struct dir_entry *) bh->b_data;
605         if (de[0].inode != inode->i_num || !de[1].inode ||
606             strcmp(".",de[0].name) || strcmp("..",de[1].name)) {
607                 printk("warning - bad directory on dev %04x\n",inode->i_dev);
608                 return 0;
609         }
    // 然后我们令 nr 等于目录项序号（从 0 开始计）；de 指向第三个目录项。并循环检测该目录
    // 中其余所有的（len - 2）个目录项，看有没有目录项的 i 节点号字段不为 0（被使用）。
610         nr = 2;
611         de += 2;
612         while (nr<len) {
    // 如果该块磁盘块中的目录项已经全部检测完毕，则释放该磁盘块的缓冲块，并读取目录数据
    // 文件中下一块含有目录项的磁盘块。读取的方法是根据当前检测的目录项序号 nr 计算出对
    // 应目录项在目录数据文件中的数据块号（nr/DIR_ENTRIES_PER_BLOCK），然后使用 bmap()
    // 函数取得对应的盘块号 block，再使用读设备盘块函数 bread() 把相应盘块读入缓冲块中，
    // 并返回该缓冲块的指针。若所读取的相应盘块没有使用（或已经不用，如文件已经删除等），
    // 则继续读下一块，若读不出，则出错返回 0。否则让 de 指向读出块的首个目录项。
613                 if ((void *) de >= (void *) (bh->b_data+BLOCK_SIZE)) {
614                         brelse(bh);
615                         block=bmap(inode,nr/DIR_ENTRIES_PER_BLOCK);
616                         if (!block) {
617                                 nr += DIR_ENTRIES_PER_BLOCK;
618                                 continue;
619                         }
620                         if (!(bh=bread(inode->i_dev,block)))
621                                 return 0;
622                         de = (struct dir_entry *) bh->b_data;
623                 }
    // 对于 de 指向的当前目录项，如果该目录项的 i 节点号字段不等于 0，则表示该目录项目前正
    // 被使用，则释放该高速缓冲区，返回 0 退出。否则，若还没有查询完该目录中的所有目录项，
    // 则把目录项序号 nr 增 1、de 指向下一个目录项，继续检测。
624                 if (de->inode) {
```

```
625                         brelse(bh);
626                         return 0;
627                 }
628                 de++;
629                 nr++;
630         }
    // 执行到这里说明该目录中没有找到已用的目录项(当然除了头两个以外)，则释放缓冲块返回 1。
631         brelse(bh);
632         return 1;
633 }
634
    //// 删除目录。
    // 参数： name - 目录名（路径名）。
    // 返回：返回 0 表示成功，否则返回出错号。
635 int sys_rmdir(const char * name)
636 {
637         const char * basename;
638         int namelen;
639         struct m_inode * dir, * inode;
640         struct buffer_head * bh;
641         struct dir_entry * de;
642
    // 首先检查参数的有效性并取路径名中顶层目录的 i 节点。如果找不到对应路径名中顶层目录
    // 的 i 节点，则返回出错码。如果最顶端的文件名长度为 0，则说明给出的路径名最后没有指
    // 定文件名，放回该目录 i 节点，返回出错码退出。如果在该目录中没有写的权限，则放回该
    // 目录的 i 节点，返回访问许可出错码退出。如果不是超级用户，则返回访问许可出错码。
643         if (!(dir = dir_namei(name,&namelen,&basename, NULL)))
644                 return -ENOENT;
645         if (!namelen) {
646                 iput(dir);
647                 return -ENOENT;
648         }
649         if (!permission(dir,MAY_WRITE)) {
650                 iput(dir);
651                 return -EPERM;
652         }
    // 然后根据指定目录的 i 节点和目录名利用函数 find_entry()寻找对应目录项，并返回包含该
    // 目录项的缓冲块指针 bh、包含该目录项的目录的 i 节点指针 dir 和该目录项指针 de。再根据
    // 该目录项 de 中的 i 节点号利用 iget()函数得到对应的 i 节点 inode。如果对应路径名上最
    // 后目录名的目录项不存在，则释放包含该目录项的高速缓冲区，放回目录的 i 节点，返回文
    // 件已经存在出错码，并退出。如果取目录项的 i 节点出错，则放回目录的 i 节点，并释放含
    // 有目录项的高速缓冲区，返回出错号。
653         bh = find_entry(&dir,basename,namelen,&de);
654         if (!bh) {
655                 iput(dir);
656                 return -ENOENT;
657         }
658         if (!(inode = iget(dir->i_dev, de->inode))) {
659                 iput(dir);
660                 brelse(bh);
661                 return -EPERM;
662         }
    // 此时我们已有包含要被删除目录项的目录 i 节点 dir、要被删除目录项的 i 节点 inode 和要
```

```
    // 被删除目录项指针 de。下面我们通过对这 3 个对象中信息的检查来验证删除操作的可行性。

    // 若该目录设置了受限删除标志并且进程的有效用户 id（euid）不是 root，并且进程的有效
    // 用户 id（euid）不等于该 i 节点的用户 id，则表示当前进程没有权限删除该目录，于是放
    // 回包含要删除目录名的目录 i 节点和该要删除目录的 i 节点，然后释放高速缓冲区，返回
    // 出错码。
663         if ((dir->i_mode & S_ISVTX) && current->euid &&
664             inode->i_uid != current->euid) {
665                 iput(dir);
666                 iput(inode);
667                 brelse(bh);
668                 return -EPERM;
669         }
    // 如果要被删除的目录项 i 节点的设备号不等于包含该目录项的目录的设备号，或者该被删除
    // 目录的引用连接计数大于 1（表示有符号连接等），则不能删除该目录。于是释放包含要删
    // 除目录名的目录 i 节点和该要删除目录的 i 节点，释放高速缓冲块，返回出错码。
670         if (inode->i_dev != dir->i_dev || inode->i_count>1) {
671                 iput(dir);
672                 iput(inode);
673                 brelse(bh);
674                 return -EPERM;
675         }
    // 如果要被删除目录的目录项 i 节点就等于包含该需删除目录的目录 i 节点，则表示试图删除
    // "."目录，这是不允许的。于是放回包含要删除目录名的目录 i 节点和要删除目录的 i 节点，
    // 释放高速缓冲块，返回出错码。
676         if (inode == dir) {     /* we may not delete ".", but "../dir" is ok */
677                 iput(inode);
678                 iput(dir);
679                 brelse(bh);
680                 return -EPERM;
681         }
    // 若要被删除目录 i 节点的属性表明这不是一个目录，则本删除操作的前提完全不存在。于是
    // 放回包含删除目录名的目录 i 节点和该要删除目录的 i 节点，释放高速缓冲块，返回出错码。
682         if (!S_ISDIR(inode->i_mode)) {
683                 iput(inode);
684                 iput(dir);
685                 brelse(bh);
686                 return -ENOTDIR;
687         }
    // 若该需被删除的目录不空，则也不能删除。于是放回包含要删除目录名的目录 i 节点和该要
    // 删除目录的 i 节点，释放高速缓冲块，返回出错码。
688         if (!empty_dir(inode)) {
689                 iput(inode);
690                 iput(dir);
691                 brelse(bh);
692                 return -ENOTEMPTY;
693         }
    // 对于一个空目录，其目录项链接数应该为 2（链接到上层目录和本目录）。若该需被删除目
    // 录的 i 节点的连接数不等于 2，则显示警告信息。但删除操作仍然继续执行。于是置该需被
    // 删除目录的目录项的 i 节点号字段为 0，表示该目录项不再使用，并置含有该目录项的高速
    // 缓冲块已修改标志，并释放该缓冲块。然后再置被删除目录 i 节点的链接数为 0（表示空闲），
    // 并置 i 节点已修改标志。
694         if (inode->i_nlinks != 2)
```

```
695                 printk("empty directory has nlink!=2 (%d)",inode->i_nlinks);
696         de->inode = 0;
697         bh->b_dirt = 1;
698         brelse(bh);
699         inode->i_nlinks=0;
700         inode->i_dirt=1;
    // 再将包含被删除目录名的目录的 i 节点链接计数减 1，修改其改变时间和修改时间为当前时
    // 间，并置该节点已修改标志。最后放回包含要删除目录名的目录 i 节点和该要删除目录的 i
    // 节点，返回 0（删除操作成功）。
701         dir->i_nlinks--;
702         dir->i_ctime = dir->i_mtime = CURRENT_TIME;
703         dir->i_dirt=1;
704         iput(dir);
705         iput(inode);
706         return 0;
707 }
708
    //// 删除（释放）文件名对应的目录项。
    // 从文件系统删除一个名字。如果是文件的最后一个链接，并且没有进程正打开该文件，则该
    // 文件也将被删除，并释放所占用的设备空间。
    // 参数：name - 文件名（路径名）。
    // 返回：成功则返回 0，否则返回出错号。
709 int sys_unlink(const char * name)
710 {
711         const char * basename;
712         int namelen;
713         struct m_inode * dir, * inode;
714         struct buffer_head * bh;
715         struct dir_entry * de;
716
    // 首先检查参数的有效性并取路径名中顶层目录的 i 节点。如果找不到对应路径名中顶层目录
    // 的 i 节点，则返回出错码。如果最顶端的文件名长度为 0，则说明给出的路径名最后没有指
    // 定文件名，放回该目录 i 节点，返回出错码退出。如果在该目录中没有写的权限，则放回该
    // 目录的 i 节点，返回访问许可出错码退出。如果找不到对应路径名顶层目录的 i 节点，则返
    // 回出错码。
717         if (!(dir = dir_namei(name,&namelen,&basename, NULL)))
718                 return -ENOENT;
719         if (!namelen) {
720                 iput(dir);
721                 return -ENOENT;
722         }
723         if (!permission(dir,MAY_WRITE)) {
724                 iput(dir);
725                 return -EPERM;
726         }
    // 然后根据指定目录的 i 节点和目录名利用函数 find_entry()寻找对应目录项，并返回包含该
    // 目录项的缓冲块指针 bh、包含该目录项的目录的 i 节点指针 dir 和该目录项指针 de。再根据
    // 该目录项 de 中的 i 节点号利用 iget()函数得到对应的 i 节点 inode。如果对应路径名上最
    // 后目录名的目录项不存在，则释放包含该目录项的高速缓冲区，放回目录的 i 节点，返回文
    // 件已经存在出错码，并退出。如果取目录项的 i 节点出错，则放回目录的 i 节点，并释放含
    // 有目录项的高速缓冲区，返回出错号。
727         bh = find_entry(&dir,basename,namelen,&de);
728         if (!bh) {
```

```
729                 iput(dir);
730                 return -ENOENT;
731         }
732         if (!(inode = iget(dir->i_dev, de->inode))) {
733                 iput(dir);
734                 brelse(bh);
735                 return -ENOENT;
736         }
    // 此时我们已有包含要被删除目录项的目录 i 节点 dir、要被删除目录项的 i 节点 inode 和要
    // 被删除目录项指针 de。下面我们通过对这 3 个对象中信息的检查来验证删除操作的可行性。

    // 若该目录设置了受限删除标志并且进程的有效用户 id（euid）不是 root，并且进程的 euid
    // 不等于该 i 节点的用户 id，并且进程的 euid 也不等于目录 i 节点的用户 id，则表示当前进
    // 程没有权限删除该目录，于是放回包含要删除目录名的目录 i 节点和该要删除目录的 i 节点，
    // 然后释放高速缓冲块，返回出错码。
737         if ((dir->i_mode & S_ISVTX) && !suser() &&
738             current->euid != inode->i_uid &&
739             current->euid != dir->i_uid) {
740                 iput(dir);
741                 iput(inode);
742                 brelse(bh);
743                 return -EPERM;
744         }
    // 如果该指定文件名是一个目录，则也不能删除。放回该目录 i 节点和该文件名目录项的 i 节
    // 点，释放包含该目录项的缓冲块，返回出错号。
745         if (S_ISDIR(inode->i_mode)) {
746                 iput(inode);
747                 iput(dir);
748                 brelse(bh);
749                 return -EPERM;
750         }
    // 如果该 i 节点的链接计数值已经为 0，则显示警告信息，并修正其为 1。
751         if (!inode->i_nlinks) {
752                 printk("Deleting nonexistent file (%04x:%d), %d\n",
753                         inode->i_dev,inode->i_num,inode->i_nlinks);
754                 inode->i_nlinks=1;
755         }
    // 现在我们可以删除文件名对应的目录项了。于是将该文件名目录项中的 i 节点号字段置为 0，
    // 表示释放该目录项，并设置包含该目录项的缓冲块已修改标志，释放该高速缓冲块。
756         de->inode = 0;
757         bh->b_dirt = 1;
758         brelse(bh);
    // 然后把文件名对应 i 节点的链接数减 1，置已修改标志，更新改变时间为当前时间。最后放
    // 回该 i 节点和目录的 i 节点，返回 0（成功）。如果是文件的最后一个链接，即 i 节点链接
    // 数减 1 后等于 0，并且此时没有进程正打开该文件，那么在调用 iput()放回 i 节点时，该文
    // 件也将被删除，并释放所占用的设备空间。参见 fs/inode.c，第 183 行。
759         inode->i_nlinks--;
760         inode->i_dirt = 1;
761         inode->i_ctime = CURRENT_TIME;
762         iput(inode);
763         iput(dir);
764         return 0;
765 }
```

```
766
    //// 建立符号链接。
    // 为一个已存在文件创建一个符号链接（也称为软连接 - hard link）。
    // 参数：oldname - 原路径名；newname - 新的路径名。
    // 返回：若成功则返回 0，否则返回出错号。
767 int sys_symlink(const char * oldname, const char * newname)
768 {
769         struct dir_entry * de;
770         struct m_inode * dir, * inode;
771         struct buffer_head * bh, * name_block;
772         const char * basename;
773         int namelen, i;
774         char c;
775
    // 首先查找新路径名的最顶层目录的 i 节点 dir，并返回最后的文件名及其长度。如果目录的
    // i 节点没有找到，则返回出错号。如果新路径名中不包括文件名，则放回新路径名目录的 i
    // 节点，返回出错号。另外，如果用户没有在新目录中写的权限，则也不能建立连接，于是放
    // 回新路径名目录的 i 节点，返回出错号。
776         dir = dir_namei(newname,&namelen,&basename, NULL);
777         if (!dir)
778                 return -EACCES;
779         if (!namelen) {
780                 iput(dir);
781                 return -EPERM;
782         }
783         if (!permission(dir,MAY_WRITE)) {
784                 iput(dir);
785                 return -EACCES;
786         }
    // 现在我们在目录指定设备上申请一个新的 i 节点，并设置该 i 节点模式为符号链接类型以及
    // 进程规定的模式屏蔽码。并且设置该 i 节点已修改标志。
787         if (!(inode = new_inode(dir->i_dev))) {
788                 iput(dir);
789                 return -ENOSPC;
790         }
791         inode->i_mode = S_IFLNK | (0777 & ~current->umask);
792         inode->i_dirt = 1;
    // 为了保存符号链接路径名字符串信息，我们需要为该 i 节点申请一个磁盘块，并让 i 节点的
    // 第 1 个直接块号 i_zone[0]等于得到的逻辑块号。然后置 i 节点已修改标志。如果申请失败
    // 则放回对应目录的 i 节点；复位新申请的 i 节点链接计数；放回该新的 i 节点，返回没有空
    // 间出错码退出。
793         if (!(inode->i_zone[0]=new_block(inode->i_dev))) {
794                 iput(dir);
795                 inode->i_nlinks--;
796                 iput(inode);
797                 return -ENOSPC;
798         }
799         inode->i_dirt = 1;
    // 然后从设备上读取新申请的磁盘块（目的是把对应块放到高速缓冲区中）。若出错，则放回
    // 对应目录的 i 节点；复位新申请的 i 节点链接计数；放回该新的 i 节点，返回没有空间出错
    // 码退出。
800         if (!(name_block=bread(inode->i_dev,inode->i_zone[0]))) {
801                 iput(dir);
```

```
802                 inode->i_nlinks--;
803                 iput(inode);
804                 return -ERROR;
805         }
    // 现在我们可以把符号链接名字符串放入这个盘块中了。盘块长度为 1024 字节，因此默认符号
    // 链接名长度最大也只能是 1024 字节。我们把用户空间中的符号链接名字符串复制到盘块所在
    // 的缓冲块中，并置缓冲块已修改标志。为防止用户提供的字符串没有以 null 结尾，我们在缓
    // 冲块数据区最后一个字节处放上一个 NULL。然后释放该缓冲块，并设置 i 节点对应文件中数
    // 据长度等于符号链接名字符串长度，并置 i 节点已修改标志。
806         i = 0;
807         while (i < 1023 && (c=get_fs_byte(oldname++)))
808                 name_block->b_data[i++] = c;
809         name_block->b_data[i] = 0;
810         name_block->b_dirt = 1;
811         brelse(name_block);
812         inode->i_size = i;
813         inode->i_dirt = 1;
    // 然后我们搜索一下路径名指定的符号链接文件名是否已经存在。若已经存在则不能创建同名
    // 目录项 i 节点。如果对应符号链接文件名已经存在，则释放包含该目录项的缓冲区块，复位
    // 新申请的 i 节点连接计数，并放回目录的 i 节点，返回文件已经存在的出错码退出。
814         bh = find_entry(&dir,basename,namelen,&de);
815         if (bh) {
816                 inode->i_nlinks--;
817                 iput(inode);
818                 brelse(bh);
819                 iput(dir);
820                 return -EEXIST;
821         }
    // 现在我们在指定目录中新添加一个目录项，用于存放新建符号链接文件名的 i 节点号和目录
    // 名。如果失败（包含该目录项的高速缓冲区指针为 NULL），则放回目录的 i 节点；所申请的
    // i 节点引用连接计数复位，并放回该 i 节点。返回出错码退出。
822         bh = add_entry(dir,basename,namelen,&de);
823         if (!bh) {
824                 inode->i_nlinks--;
825                 iput(inode);
826                 iput(dir);
827                 return -ENOSPC;
828         }
    // 最后令该新目录项的 i 节点字段等于新 i 节点号，并置高速缓冲块已修改标志，释放高速缓
    // 冲块，放回目录和新的 i 节点，最后返回 0（成功）。
829         de->inode = inode->i_num;
830         bh->b_dirt = 1;
831         brelse(bh);
832         iput(dir);
833         iput(inode);
834         return 0;
835 }
836
    //// 为文件建立一个文件名目录项。
    // 为一个已存在的文件创建一个新链接（也称为硬连接 - hard link）。
    // 参数：oldname - 原路径名；newname - 新的路径名。
    // 返回：若成功则返回 0，否则返回出错号。
837 int sys_link(const char * oldname, const char * newname)
```

```
838 {
839         struct dir_entry * de;
840         struct m_inode * oldinode, * dir;
841         struct buffer_head * bh;
842         const char * basename;
843         int namelen;
844
    // 首先对原文件名进行有效性验证，它应该存在并且不是一个目录名。所以我们先取原文件路
    // 径名对应的 i 节点 oldinode。如果为 0，则表示出错，返回出错号。如果原路径名对应的是
    // 一个目录名，则放回该 i 节点，也返回出错号。
845         oldinode=namei(oldname);
846         if (!oldinode)
847                 return -ENOENT;
848         if (S_ISDIR(oldinode->i_mode)) {
849                 iput(oldinode);
850                 return -EPERM;
851         }
    // 然后查找新路径名的最顶层目录的 i 节点 dir，并返回最后的文件名及其长度。如果目录的
    // i 节点没有找到，则放回原路径名的 i 节点，返回出错号。如果新路径名中不包括文件名，
    // 则放回原路径名 i 节点和新路径名目录的 i 节点，返回出错号。
852         dir = dir_namei(newname,&namelen,&basename, NULL);
853         if (!dir) {
854                 iput(oldinode);
855                 return -EACCES;
856         }
857         if (!namelen) {
858                 iput(oldinode);
859                 iput(dir);
860                 return -EPERM;
861         }
    // 我们不能跨设备建立硬链接。因此如果新路径名顶层目录的设备号与原路径名的设备号
    // 不一样，则放回新路径名目录的 i 节点和原路径名的 i 节点，返回出错号。另外，如果用户
    // 没有在新目录中写的权限，则也不能建立连接，于是放回新路径名目录的 i 节点和原路径名
    // 的 i 节点，返回出错号。
862         if (dir->i_dev != oldinode->i_dev) {
863                 iput(dir);
864                 iput(oldinode);
865                 return -EXDEV;
866         }
867         if (!permission(dir,MAY_WRITE)) {
868                 iput(dir);
869                 iput(oldinode);
870                 return -EACCES;
871         }
    // 现在查询该新路径名是否已经存在，如果存在则也不能建立链接。于是释放包含该已存在目
    // 录项的高速缓冲块，放回新路径名目录的 i 节点和原路径名的 i 节点，返回出错号。
872         bh = find_entry(&dir,basename,namelen,&de);
873         if (bh) {
874                 brelse(bh);
875                 iput(dir);
876                 iput(oldinode);
877                 return -EEXIST;
878         }
```

```
    // 现在所有条件都满足了，于是我们在新目录中添加一个目录项。若失败则放回该目录的 i 节
    // 点和原路径名的 i 节点，返回出错号。否则初始设置该目录项的 i 节点号等于原路径名的 i
    // 节点号，并置包含该新添目录项的缓冲块已修改标志，释放该缓冲块，放回目录的 i 节点。
879         bh = add_entry(dir,basename,namelen,&de);
880         if (!bh) {
881                 iput(dir);
882                 iput(oldinode);
883                 return -ENOSPC;
884         }
885         de->inode = oldinode->i_num;
886         bh->b_dirt = 1;
887         brelse(bh);
888         iput(dir);
    // 再将原节点的链接计数加 1，修改其改变时间为当前时间，并设置 i 节点已修改标志。最后
    // 放回原路径名的 i 节点，并返回 0（成功）。
889         oldinode->i_nlinks++;
890         oldinode->i_ctime = CURRENT_TIME;
891         oldinode->i_dirt = 1;
892         iput(oldinode);
893         return 0;
894 }
895
```

12.7 程序 12-7 linux/fs/file_table.c

```
 1 /*
 2  *  linux/fs/file_table.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <linux/fs.h>     // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
 8
 9 struct file file_table[NR_FILE];  // 文件表数组(64 项)。
10
```

12.8 程序 12-8 linux/fs/block_dev.c

```
 1 /*
 2  *  linux/fs/block_dev.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
 8
 9 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据，
                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
10 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
11 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
12 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
13
   // 设备数据块总数指针数组。每个指针项指向指定主设备号的总块数数组 hd_sizes[]。该总
   // 块数数组每一项对应子设备号确定的一个子设备上所拥有的数据块总数（1 块大小 = 1KB）。
14 extern int *blk_size[];              // blk_drv/ll_rw_blk.c，49 行。
15
   //// 数据块写函数 - 向指定设备从给定偏移处写入指定长度数据。
   // 参数：dev - 设备号；pos - 设备文件中偏移量指针；buf - 用户空间中缓冲区地址；
   // count - 要传送的字节数。
   // 返回已写入字节数。若没有写入任何字节或出错，则返回出错号。
   // 对于内核来说，写操作是向高速缓冲区中写入数据。什么时候数据最终写入设备是由高速缓
   // 冲管理程序决定并处理的。另外，因为块设备是以块为单位进行读写，因此对于写开始位置
   // 不处于块起始处时，需要先将开始字节所在的整个块读出，然后将需要写的数据从写开始处
   // 填写满该块，再将完整的一块数据写盘（即交由高速缓冲程序去处理）。
16 int block_write(int dev, long * pos, char * buf, int count)
17 {
   // 首先由文件中位置 pos 换算成开始读写盘块的块序号 block，并求出需写第 1 字节在该块中
   // 的偏移位置 offset。
18         int block = *pos >> BLOCK_SIZE_BITS;          // pos 所在文件数据块号。
19         int offset = *pos & (BLOCK_SIZE-1);           // pos 在数据块中偏移值。
20         int chars;
21         int written = 0;
22         int size;
23         struct buffer_head * bh;
24         register char * p;                    // 局部寄存器变量，被存放在寄存器中。
25
   // 在写一个块设备文件时，要求写的总数据块数当然不能超过指定设备上容许的最大数据块总
   // 数。因此这里首先取出指定设备的块总数 size 来比较和限制函数参数给定的写入数据长度。
   // 如果系统中没有对设备指定长度，就使用默认长度 0x7fffffff（2GB 个块）。
26         if (blk_size[MAJOR(dev)])
27                 size = blk_size[MAJOR(dev)][MINOR(dev)];
28         else
29                 size = 0x7fffffff;
   // 然后针对要写入的字节数 count，循环执行以下操作，直到数据全部写入。在循环执行过程
   // 中，若当前写入数据的块号已经大于或等于指定设备的总块数，则返回已写字节数并退出。
   // 然后再计算在当前处理的数据块中可写入的字节数。如果需要写入的字节数填不满一块，那
   // 么就只需写 count 字节。如果正好要写 1 块数据内容，则直接申请 1 块高速缓冲块，并把用
```

```
    // 户数据放入即可。否则就需要读入将被写入部分数据的数据块，并预读下两块数据。然后将
    // 块号递增 1，为下次操作做好准备。如果缓冲块操作失败，则返回已写字节数，如果没有写
    // 入任何字节，则返回出错号（负数）。
30          while (count>0) {
31                  if (block >= size)
32                          return written?written:-EIO;
33                  chars = BLOCK_SIZE - offset;            // 本块可写入的字节数。
34                  if (chars > count)
35                          chars=count;
36                  if (chars == BLOCK_SIZE)
37                          bh = getblk(dev,block);         // buffer.c 第 206、322 行。
38                  else
39                          bh = breada(dev,block,block+1,block+2,-1);
40                  block++;
41                  if (!bh)
42                          return written?written:-EIO;
    // 接着先把指针 p 指向读出数据的缓冲块中开始写入数据的位置处。若最后一次循环写入的数
    // 据不足一块，则需从块开始处填写（修改）所需的字节，因此这里需预先设置 offset 为零。
    // 此后将文件中偏移指针 pos 前移此次将要写的字节数 chars，并累加这些要写的字节数到统
    // 计值 written 中。再把还需要写的计数值 count 减去此次要写的字节数 chars。然后我们从
    // 用户缓冲区复制 chars 个字节到 p 指向的高速缓冲块中开始写入的位置处。复制完后就设置
    // 该缓冲区块已修改标志，并释放该缓冲区（也即该缓冲区引用计数递减 1）。
43                  p = offset + bh->b_data;
44                  offset = 0;
45                  *pos += chars;
46                  written += chars;                       // 累计写入字节数。
47                  count -= chars;
48                  while (chars-->0)
49                          *(p++) = get_fs_byte(buf++);
50                  bh->b_dirt = 1;
51                  brelse(bh);
52          }
53          return written;                                 // 返回已写入的字节数，正常退出。
54  }
55
    //// 数据块读函数 - 从指定设备和位置处读入指定长度数据到用户缓冲区中。
    // 参数：dev - 设备号；pos - 设备文件中偏移量指针；buf - 用户空间中缓冲区地址；
    // count - 要传送的字节数。
    // 返回已读入字节数。若没有读入任何字节或出错，则返回出错号。
56  int block_read(int dev, unsigned long * pos, char * buf, int count)
57  {
58          int block = *pos >> BLOCK_SIZE_BITS;
59          int offset = *pos & (BLOCK_SIZE-1);
60          int chars;
61          int size;
62          int read = 0;
63          struct buffer_head * bh;
64          register char * p;                      // 局部寄存器变量，被存放在寄存器中。
65
    // 在读一个块设备文件时，要求读的总数据块数当然不能超过指定设备上容许的最大数据块总
    // 数。因此这里首先取出指定设备的块总数 size 来比较和限制函数参数给定的读入数据长度。
    // 如果系统中没有对设备指定长度，就使用默认长度 0x7fffffff（2GB 个块）。
66          if (blk_size[MAJOR(dev)])
```

```
67                  size = blk_size[MAJOR(dev)][MINOR(dev)];
68          else
69                  size = 0x7fffffff;
    // 然后针对要读入的字节数 count，循环执行以下操作，直到数据全部读入。在循环执行过程
    // 中，若当前读入数据的块号已经大于或等于指定设备的总块数，则返回已读字节数并退出。
    // 然后再计算在当前处理的数据块中需读入的字节数。如果需要读入的字节数还不满一块，那
    // 么就只需读 count 字节。然后调用读块函数 breada()读入需要的数据块，并预读下两块数据，
    // 如果读操作出错，则返回已读字节数，如果没有读入任何字节，则返回出错号。然后将块号
    // 递增 1。为下次操作做好准备。如果缓冲块操作失败，则返回已写字节数，如果没有读入任
    // 何字节，则返回出错号（负数）。
70          while (count>0) {
71                  if (block >= size)
72                          return read?read:-EIO;
73                  chars = BLOCK_SIZE-offset;
74                  if (chars > count)
75                          chars = count;
76                  if (!(bh = breada(dev,block,block+1,block+2,-1)))
77                          return read?read:-EIO;
78                  block++;
    // 接着先把指针 p 指向读出盘块的缓冲块中开始读入数据的位置处。若最后一次循环读操作的
    // 数据不足一块，则需从块起始处读取所需字节，因此这里需预先设置 offset 为零。 此后将
    // 文件中偏移指针 pos 前移此次将要读的字节数 chars，并且累加这些要读的字节数到统计值
    // read 中。再把还需要读的计数值 count 减去此次要读的字节数 chars。然后我们从高速缓冲
    // 块中 p 指向的开始读的位置处复制 chars 个字节到用户缓冲区中，同时把用户缓冲区指针前
    // 移。本次复制完后就释放该缓冲块。
79                  p = offset + bh->b_data;
80                  offset = 0;
81                  *pos += chars;
82                  read += chars;                  // 累计读入字节数。
83                  count -= chars;
84                  while (chars-->0)
85                          put_fs_byte(*(p++),buf++);
86                  brelse(bh);
87          }
88          return read;                            // 返回已读取的字节数，正常退出。
89  }
90
```

12.9 程序 12-9 linux/fs/file_dev.c

```
 1 /*
 2  *  linux/fs/file_dev.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
 8 #include <fcntl.h>        // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
 9
10 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据等。
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
13
14 #define MIN(a,b) (((a)<(b))?(a):(b))          // 取 a,b 中的最小值。
15 #define MAX(a,b) (((a)>(b))?(a):(b))          // 取 a,b 中的最大值。
16
   //// 文件读函数 - 根据 i 节点和文件结构，读取文件中数据。
   // 由 i 节点我们可以知道设备号，由 filp 结构可以知道文件中当前读写指针位置。buf 指定用
   // 户空间中缓冲区的位置，count 是需要读取的字节数。 返回值是实际读取的字节数，或出错
   // 号（小于 0）。
17 int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
18 {
19         int left,chars,nr;
20         struct buffer_head * bh;
21
   // 首先判断参数的有效性。若需要读取的字节计数 count 小于等于零，则返回 0。若还需要读
   // 取的字节数不等于 0，就循环执行下面操作，直到数据全部读出或遇到问题。在读循环操作
   // 过程中，我们根据 i 节点和文件表结构信息，并利用 bmap() 得到包含文件当前读写位置的
   // 数据块在设备上对应的逻辑块号 nr。若 nr 不为 0，则从 i 节点指定的设备上读取该逻辑块。
   // 如果读操作失败则退出循环。若 nr 为 0，表示指定的数据块不存在，置缓冲块指针为 NULL。
   // (filp->f_pos)/BLOCK_SIZE 用于计算出文件当前指针所在数据块号。
22         if ((left=count)<=0)
23                 return 0;
24         while (left) {
25                 if (nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE)) { // inode.c 第 140 行。
26                         if (!(bh=bread(inode->i_dev,nr)))
27                                 break;
28                 } else
29                         bh = NULL;
   // 接着我们计算文件读写指针在数据块中的偏移值 nr，则在该数据块中我们希望读取的字节数
   // 为 (BLOCK_SIZE - nr)。然后和现在还需读取的字节数 left 作比较，其中小值即为本次操作
   // 需读取的字节数 chars。如果 (BLOCK_SIZE - nr) > left，则说明该块是需要读取的最后一
   // 块数据，反之则还需要读取下一块数据。之后调整读写文件指针。 指针前移此次将读取的字
   // 节数 chars。剩余字节计数 left 相应减去 chars。
30                 nr = filp->f_pos % BLOCK_SIZE;
31                 chars = MIN( BLOCK_SIZE-nr , left );
32                 filp->f_pos += chars;
33                 left -= chars;
   // 若上面从设备上读到了数据，则将 p 指向缓冲块中开始读取数据的位置，并且复制 chars 字节
```

```
// 到用户缓冲区 buf 中。否则往用户缓冲区中填入 chars 个 0 值字节。
34                  if (bh) {
35                          char * p = nr + bh->b_data;
36                          while (chars-->0)
37                                  put_fs_byte(*(p++),buf++);
38                          brelse(bh);
39                  } else {
40                          while (chars-->0)
41                                  put_fs_byte(0,buf++);
42                  }
43          }
// 修改该 i 节点的访问时间为当前时间。返回读取的字节数，若读取字节数为 0，则返回出错号。
// CURRENT_TIME 是定义在 include/linux/sched.h 第 142 行上的宏，用于计算 UNIX 时间。即从
// 1970 年 1 月 1 日 0 时 0 秒开始，到当前的时间。单位是秒。
44          inode->i_atime = CURRENT_TIME;
45          return (count-left)?(count-left):-ERROR;
46  }
47
//// 文件写函数 - 根据 i 节点和文件结构信息，将用户数据写入文件中。
// 由 i 节点我们可以知道设备号，而由 file 结构可以知道文件中当前读写指针位置。buf 指定
// 用户态中缓冲区的位置，count 为需要写入的字节数。 返回值是实际写入的字节数，或出错
// 号（小于 0）。
48  int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
49  {
50          off_t pos;
51          int block,c;
52          struct buffer_head * bh;
53          char * p;
54          int i=0;
55
56  /*
57   * ok, append may not work when many processes are writing at the same time
58   * but so what. That way leads to madness anyway.
59   */
/*
 * ok，当许多进程同时写时，append 操作可能不行，但那又怎样。不管怎样那样做会
 * 导致混乱一团。
 */
// 首先确定数据写入文件的位置。如果是要向文件后添加数据，则将文件读写指针移到文件尾
// 部。否则就将在文件当前读写指针处写入。
60          if (filp->f_flags & O_APPEND)
61                  pos = inode->i_size;
62          else
63                  pos = filp->f_pos;
// 然后在已写入字节数 i（刚开始时为 0）小于指定写入字节数 count 时，循环执行以下操作。
// 在循环操作过程中，我们先取文件数据块号 （ pos/BLOCK_SIZE ） 在设备上对应的逻辑块号
// block。如果对应的逻辑块不存在就创建一块。如果得到的逻辑块号 = 0，则表示创建失败，
// 于是退出循环。否则我们根据该逻辑块号读取设备上的相应逻辑块，若出错也退出循环。
64          while (i<count) {
65                  if (!(block = create_block(inode,pos/BLOCK_SIZE)))
66                          break;
67                  if (!(bh=bread(inode->i_dev,block)))
68                          break;
```

```
    // 此时缓冲块指针 bh 正指向刚读入的文件数据块。现在再求出文件当前读写指针在该数据块中
    // 的偏移值 c，并将指针 p 指向缓冲块中开始写入数据的位置，并置该缓冲块已修改标志。对于
    // 块中当前指针，从开始读写位置到块末共可写入 c =(BLOCK_SIZE - c)个字节。若 c 大于剩余
    // 还需写入的字节数 (count - i)，则此次只需再写入 c = (count - i)个字节即可。
69                  c = pos % BLOCK_SIZE;
70                  p = c + bh->b_data;
71                  bh->b_dirt = 1;
72                  c = BLOCK_SIZE-c;
73                  if (c > count-i) c = count-i;
    // 在写入数据之前，我们先预先设置好下一次循环操作要读写文件中的位置。因此我们把 pos
    // 指针前移此次需写入的字节数。如果此时 pos 位置值超过了文件当前长度，则修改 i 节点中
    // 文件长度字段，并置 i 节点已修改标志。然后把此次要写入的字节数 c 累加到已写入字节计
    // 数值 i 中，供循环判断使用。接着从用户缓冲区 buf 中复制 c 个字节到高速缓冲块中 p 指向
    // 的开始位置处。复制完后就释放该缓冲块。
74                  pos += c;
75                  if (pos > inode->i_size) {
76                          inode->i_size = pos;
77                          inode->i_dirt = 1;
78                  }
79                  i += c;
80                  while (c-->0)
81                          *(p++) = get_fs_byte(buf++);
82                  brelse(bh);
83          }
    // 当数据已经全部写入文件或者在写操作过程中发生问题时就会退出循环。此时我们更改文件
    // 修改时间为当前时间，并调整文件读写指针。如果此次操作不是在文件尾添加数据，则把文
    // 件读写指针调整到当前读写位置 pos 处，并更改文件 i 节点的修改时间为当前时间。最后返
    // 回写入的字节数，若写入字节数为 0，则返回出错号-1。
84          inode->i_mtime = CURRENT_TIME;
85          if (!(filp->f_flags & O_APPEND)) {
86                  filp->f_pos = pos;
87                  inode->i_ctime = CURRENT_TIME;
88          }
89          return (i?i:-1);
90  }
91
```

12.10 程序 12-10 linux/fs/pipe.c

```
 1 /*
 2  *  linux/fs/pipe.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <signal.h>        // 信号头文件。定义信号符号常量，信号结构及操作函数原型。
 8 #include <errno.h>         // 错误号头文件。包含系统中各种出错号。
 9 #include <termios.h>       // 终端输入输出函数头文件。主要定义控制异步通信口的终端接口。
10
11 #include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
12 #include <linux/mm.h>      /* for get_free_page */     /* 使用其中的 get_free_page */
13 #include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
15
   //// 管道读操作函数。
   // 参数 inode 是管道对应的 i 节点，buf 是用户数据缓冲区指针，count 是读取的字节数。
16 int read_pipe(struct m_inode * inode, char * buf, int count)
17 {
18         int chars, size, read = 0;
19
   // 如果需要读取的字节计数 count 大于 0，我们就循环执行以下操作。在循环读操作过程中，
   // 若当前管道中没有数据（size=0），则唤醒等待该节点的进程，这通常是写管道进程。如果
   // 已没有写管道者，即 i 节点引用计数值小于 2，则返回已读字节数退出。如果当前收到有非
   // 阻塞信号，则立刻返回已读取字节数退出；若还没有收到任何数据，则返回重新启动系统
   // 调用号退出。否则就让进程在该管道上睡眠，用以等待信息的到来。宏 PIPE_SIZE 定义在
   // include/linux/fs.h 中。关于“重新启动系统调用”，请参见 kernel/signal.c 程序。
20         while (count>0) {
21                 while (!(size=PIPE_SIZE(*inode))) {      // 取管道中数据长度值。
22                         wake_up(& PIPE_WRITE_WAIT(*inode));
23                         if (inode->i_count != 2)    /* are there any writers? */
24                                 return read;
25                         if (current->signal & ~current->blocked)
26                                 return read?read:-ERESTARTSYS;
27                         interruptible_sleep_on(& PIPE_READ_WAIT(*inode));
28                 }
   // 此时说明管道（缓冲区）中有数据。于是我们取管道尾指针到缓冲区末端的字节数 chars。
   // 如果其大于还需要读取的字节数 count，则令其等于 count。如果 chars 大于当前管道中含
   // 有数据的长度 size，则令其等于 size。 然后把需读字节数 count 减去此次可读的字节数
   // chars，并累加已读字节数 read。
29                 chars = PAGE_SIZE-PIPE_TAIL(*inode);
30                 if (chars > count)
31                         chars = count;
32                 if (chars > size)
33                         chars = size;
34                 count -= chars;
35                 read += chars;
   // 再令 size 指向管道尾指针处，并调整当前管道尾指针（前移 chars 字节）。若尾指针超过
   // 管道末端则绕回。然后将管道中的数据复制到用户缓冲区中。对于管道 i 节点，其 i_size
```

```
// 字段中是管道缓冲块指针。
36                 size = PIPE_TAIL(*inode);
37                 PIPE_TAIL(*inode) += chars;
38                 PIPE_TAIL(*inode) &= (PAGE_SIZE-1);
39                 while (chars-->0)
40                         put_fs_byte(((char *)inode->i_size)[size++],buf++);
41         }
// 当此次读管道操作结束，则唤醒等待该管道的进程，并返回读取的字节数。
42         wake_up(&PIPE_WRITE_WAIT(*inode));
43         return read;
44 }
45
//// 管道写操作函数。
// 参数 inode 是管道对应的 i 节点，buf 是数据缓冲区指针，count 是将写入管道的字节数。
46 int write_pipe(struct m_inode * inode, char * buf, int count)
47 {
48         int chars, size, written = 0;
49
// 如果要写入的字节数 count 还大于 0，那么我们就循环执行以下操作。在循环操作过程中，
// 如果当前管道中已经满了（空闲空间 size = 0），则唤醒等待该管道的进程，通常唤醒
// 的是读管道进程。 如果已没有读管道者，即 i 节点引用计数值小于 2，则向当前进程发送
// SIGPIPE 信号，并返回已写入的字节数退出；若写入 0 字节，则返回 -1。否则让当前进程
// 在该管道上睡眠，以等待读管道进程来读取数据，从而让管道腾出空间。宏 PIPE_SIZE()、
// PIPE_HEAD()等定义在文件 include/linux/fs.h 中。
50         while (count>0) {
51                 while (!(size=(PAGE_SIZE-1)-PIPE_SIZE(*inode))) {
52                         wake_up(&PIPE_READ_WAIT(*inode));
53                         if (inode->i_count != 2) { /* no readers */
54                                 current->signal |= (1<<(SIGPIPE-1));
55                                 return written?written:-1;
56                         }
57                         sleep_on(&PIPE_WRITE_WAIT(*inode));
58                 }
// 程序执行到这里表示管道缓冲区中有可写空间 size。于是我们取管道头指针到缓冲区末端空
// 间字节数 chars。写管道操作是从管道头指针处开始写的。如果 chars 大于还需要写入的字节
// 数 count，则令其等于 count。 如果 chars 大于当前管道中空闲空间长度 size，则令其等于
// size。然后把需要写入字节数 count 减去此次可写入的字节数 chars，并把写入字节数累加到
// written 中。
59                 chars = PAGE_SIZE-PIPE_HEAD(*inode);
60                 if (chars > count)
61                         chars = count;
62                 if (chars > size)
63                         chars = size;
64                 count -= chars;
65                 written += chars;
// 再令 size 指向管道数据头指针处，并调整当前管道数据头部指针（前移 chars 字节）。若头
// 指针超过管道末端则绕回。然后从用户缓冲区复制 chars 个字节到管道头指针开始处。 对于
// 管道 i 节点，其 i_size 字段中是管道缓冲块指针。
66                 size = PIPE_HEAD(*inode);
67                 PIPE_HEAD(*inode) += chars;
68                 PIPE_HEAD(*inode) &= (PAGE_SIZE-1);
69                 while (chars-->0)
70                         ((char *)inode->i_size)[size++]=get_fs_byte(buf++);
```

```
71          }
    // 当此次写管道操作结束，则唤醒等待管道的进程，返回已写入的字节数，退出。
72          wake_up(& PIPE_READ_WAIT(*inode));
73          return written;
74  }
75
    //// 创建管道系统调用。
    // 在 fildes 所指的数组中创建一对文件句柄（描述符）。这对文件句柄指向一管道 i 节点。
    // 参数：filedes -文件句柄数组。fildes[0]用于读管道数据，fildes[1]向管道写入数据。
    // 成功时返回 0，出错时返回-1。
76  int sys_pipe(unsigned long * fildes)
77  {
78          struct m_inode * inode;
79          struct file * f[2];                     // 文件结构数组。
80          int fd[2];                              // 文件句柄数组。
81          int i,j;
82
    // 首先从系统文件表中取两个空闲项（引用计数字段为 0 的项），并分别设置引用计数为 1。
    // 若只有 1 个空闲项，则释放该项（引用计数复位）。若没有找到两个空闲项，则返回 -1。
83          j=0;
84          for(i=0;j<2 && i<NR_FILE;i++)
85                  if (!file_table[i].f_count)
86                          (f[j++]=i+file_table)->f_count++;
87          if (j==1)
88                  f[0]->f_count=0;
89          if (j<2)
90                  return -1;
    // 针对上面取得的两个文件表结构项，分别分配一文件句柄号，并使进程文件结构指针数组的
    // 两项分别指向这两个文件结构。而文件句柄即是该数组的索引号。类似地，如果只有一个空
    // 闲文件句柄，则释放该句柄（置空相应数组项）。如果没有找到两个空闲句柄，则释放上面
    // 获取的两个文件结构项（复位引用计数值），并返回 -1。
91          j=0;
92          for(i=0;j<2 && i<NR_OPEN;i++)
93                  if (!current->filp[i]) {
94                          current->filp[ fd[j]=i ] = f[j];
95                          j++;
96                  }
97          if (j==1)
98                  current->filp[fd[0]]=NULL;
99          if (j<2) {
100                 f[0]->f_count=f[1]->f_count=0;
101                 return -1;
102         }
    // 然后利用函数 get_pipe_inode()申请一个管道使用的 i 节点，并为管道分配一页内存作为缓
    // 冲区。如果不成功，则相应释放两个文件句柄和文件结构项，并返回-1。
103         if (!(inode=get_pipe_inode())) {          // fs/inode.c，第 231 行开始处。
104                 current->filp[fd[0]] =
105                         current->filp[fd[1]] = NULL;
106                 f[0]->f_count = f[1]->f_count = 0;
107                 return -1;
108         }
    // 如果管道 i 节点申请成功，则对两个文件结构进行初始化操作，让它们都指向同一个管道 i 节
    // 点，并把读写指针都置零。第 1 个文件结构的文件模式置为读，第 2 个文件结构的文件模式置
```

```
   // 为写。最后将文件句柄数组复制到对应的用户空间数组中，成功返回 0，退出。
109         f[0]->f_inode = f[1]->f_inode = inode;
110         f[0]->f_pos = f[1]->f_pos = 0;
111         f[0]->f_mode = 1;               /* read */
112         f[1]->f_mode = 2;               /* write */
113         put_fs_long(fd[0],0+fildes);
114         put_fs_long(fd[1],1+fildes);
115         return 0;
116 }
117
    //// 管道 io 控制函数。
    // 参数：pino - 管道 i 节点指针；cmd - 控制命令；arg - 参数。
    // 函数返回 0 表示执行成功，否则返回出错码。
118 int pipe_ioctl(struct m_inode *pino, int cmd, int arg)
119 {
    // 如果命令是取管道中当前可读数据长度，则把管道数据长度值添入用户参数指定的位置处，
    // 并返回 0。否则返回无效命令错误码。
120         switch (cmd) {
121                 case FIONREAD:
122                         verify_area((void *) arg,4);
123                         put_fs_long(PIPE_SIZE(*pino),(unsigned long *) arg);
124                         return 0;
125                 default:
126                         return -EINVAL;
127         }
128 }
129
```

12.11 程序 12-11 linux/fs/char_dev.c

```
 1 /*
 2  *  linux/fs/char_dev.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
 8 #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
 9
10 #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 数据等。
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12
13 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14 #include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
15
16 extern int tty_read(unsigned minor,char * buf,int count);     // 终端读。
17 extern int tty_write(unsigned minor,char * buf,int count);    // 终端写。
18
   // 定义字符设备读写函数指针类型。
19 typedef (*crw_ptr)(int rw,unsigned minor,char * buf,int count,off_t * pos);
20
   //// 串口终端读写操作函数。
   // 参数：rw - 读写命令；minor - 终端子设备号；buf - 缓冲区；cout - 读写字节数；
   // pos - 读写操作当前指针，对于终端操作，该指针无用。
   // 返回：实际读写的字节数。若失败则返回出错码。
21 static int rw_ttyx(int rw,unsigned minor,char * buf,int count,off_t * pos)
22 {
23         return ((rw==READ)?tty_read(minor,buf,count):
24                 tty_write(minor,buf,count));
25 }
26
   //// 终端读写操作函数。
   // 同上 rw_ttyx()，只是增加了对进程是否有控制终端的检测。
27 static int rw_tty(int rw,unsigned minor,char * buf,int count, off_t * pos)
28 {
   // 若进程没有对应的控制终端，则返回出错号。否则调用终端读写函数 rw_ttyx()，并返回
   // 实际读写字节数。
29         if (current->tty<0)
30                 return -EPERM;
31         return rw_ttyx(rw,current->tty,buf,count,pos);
32 }
33
   //// 内存数据读写。未实现。
34 static int rw_ram(int rw,char * buf, int count, off_t *pos)
35 {
36         return -EIO;
37 }
38
   //// 物理内存数据读写操作函数。未实现。
```

```
39 static int rw_mem(int rw,char * buf, int count, off_t * pos)
40 {
41         return -EIO;
42 }
43
   //// 内核虚拟内存数据读写函数。未实现。
44 static int rw_kmem(int rw,char * buf, int count, off_t * pos)
45 {
46         return -EIO;
47 }
48
   // 端口读写操作函数。
   // 参数：rw - 读写命令；buf - 缓冲区；cout - 读写字节数；pos - 端口地址。
   // 返回：实际读写的字节数。
49 static int rw_port(int rw,char * buf, int count, off_t * pos)
50 {
51         int i=*pos;
52
   // 对于所要求读写的字节数，并且端口地址小于 64k 时，循环执行单个字节的读写操作。
   // 若是读命令，则从端口 i 中读取一字节内容并放到用户缓冲区中。若是写命令，则从用
   // 户数据缓冲区中取一字节输出到端口 i。
53         while (count-->0 && i<65536) {
54                 if (rw==READ)
55                         put_fs_byte(inb(i),buf++);
56                 else
57                         outb(get_fs_byte(buf++),i);
58                 i++;                                    // 前移一个端口。[??]
59         }
   // 然后计算读/写的字节数，调整相应读写指针，并返回读/写的字节数。
60         i -= *pos;
61         *pos += i;
62         return i;
63 }
64
   //// 内存读写操作函数。内存主设备号是 1。这里仅给出对 0-5 子设备的处理。
65 static int rw_memory(int rw, unsigned minor, char * buf, int count, off_t * pos)
66 {
   // 根据内存设备子设备号，分别调用不同的内存读写函数。
67         switch(minor) {
68                 case 0:         // 对应设备文件名是 /dev/ram0 或/dev/ramdisk。
69                         return rw_ram(rw,buf,count,pos);
70                 case 1:         // /dev/ram1 或/dev/mem 或 ram。
71                         return rw_mem(rw,buf,count,pos);
72                 case 2:         // /dev/ram2 或/dev/kmem。
73                         return rw_kmem(rw,buf,count,pos);
74                 case 3:         // /dev/null。
75                         return (rw==READ)?0:count;      /* rw_null */
76                 case 4:         // /dev/port。
77                         return rw_port(rw,buf,count,pos);
78                 default:
79                         return -EIO;
80         }
81 }
```

```
82
   // 定义系统中设备种数。
83 #define NRDEVS ((sizeof (crw_table))/(sizeof (crw_ptr)))
84
   // 字符设备读写函数指针表。
85 static crw_ptr crw_table[]={
86         NULL,           /* nodev */             /* 无设备(空设备) */
87         rw_memory,      /* /dev/mem etc */      /* /dev/mem 等 */
88         NULL,           /* /dev/fd */           /* /dev/fd 软驱 */
89         NULL,           /* /dev/hd */           /* /dev/hd 硬盘 */
90         rw_ttyx,        /* /dev/ttyx */         /* /dev/ttyx 串口终端 */
91         rw_tty,         /* /dev/tty */          /* /dev/tty 终端 */
92         NULL,           /* /dev/lp */           /* /dev/lp 打印机 */
93         NULL};          /* unnamed pipes */     /* 未命名管道 */
94
   //// 字符设备读写操作函数。
   // 参数：rw -读写命令；dev -设备号；buf -缓冲区；count -读写字节数；pos -读写指针。
   // 返回：实际读/写字节数。
95 int rw_char(int rw,int dev, char * buf, int count, off_t * pos)
96 {
97         crw_ptr call_addr;
98
   // 如果设备号超出系统设备数，则返回出错码。如果该设备没有对应的读/写函数，也返回出
   // 错码。否则调用对应设备的读写操作函数，并返回实际读/写的字节数。
99         if (MAJOR(dev)>=NRDEVS)
100                return -ENODEV;
101        if (!(call_addr=crw_table[MAJOR(dev)]))
102                return -ENODEV;
103        return call_addr(rw,MINOR(dev),buf,count,pos);
104 }
105
```

12.12 程序 12-12 linux/fs/read_write.c

```
 1 /*
 2  *  linux/fs/read_write.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <sys/stat.h>     // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。
 8 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
 9 #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
10
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 数据等。
13 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14
   // 字符设备读写函数。fs/char_dev.c，第 95 行。
15 extern int rw_char(int rw,int dev, char * buf, int count, off_t * pos);
   // 读管道操作函数。fs/pipe.c，第 13 行。
16 extern int read_pipe(struct m_inode * inode, char * buf, int count);
   // 写管道操作函数。fs/pipe.c，第 41 行。
17 extern int write_pipe(struct m_inode * inode, char * buf, int count);
   // 块设备读操作函数。fs/block_dev.c，第 47 行。
18 extern int block_read(int dev, off_t * pos, char * buf, int count);
   // 块设备写操作函数。fs/block_dev.c，第 14 行。
19 extern int block_write(int dev, off_t * pos, char * buf, int count);
   // 读文件操作函数。fs/file_dev.c，第 17 行。
20 extern int file_read(struct m_inode * inode, struct file * filp,
21                 char * buf, int count);
   // 写文件操作函数。fs/file_dev.c，第 48 行。
22 extern int file_write(struct m_inode * inode, struct file * filp,
23                 char * buf, int count);
24
   //// 重定位文件读写指针系统调用。
   // 参数 fd 是文件句柄，offset 是新的文件读写指针偏移值，origin 是偏移的起始位置，可有
   // 三种选择：SEEK_SET（0，从文件开始处）、SEEK_CUR（1，从当前读写位置）、SEEK_END（
   // 2，从文件尾处）。
25 int sys_lseek(unsigned int fd,off_t offset, int origin)
26 {
27         struct file * file;
28         int tmp;
29
   // 首先判断函数提供的参数有效性。如果文件句柄值大于程序最多打开文件数 NR_OPEN(20)，
   // 或者该句柄的文件结构指针为空，或者对应文件结构的 i 节点字段为空，或者指定设备文件
   // 指针是不可定位的，则返回出错码并退出。如果文件对应的 i 节点是管道节点，则返回出错
   // 码退出。因为管道头尾指针不可随意移动！
30         if (fd >= NR_OPEN || !(file=current->filp[fd]) || !(file->f_inode)
31            || !IS_SEEKABLE(MAJOR(file->f_inode->i_dev)))
32                 return -EBADF;
33         if (file->f_inode->i_pipe)
34                 return -ESPIPE;
```

```
    // 然后根据设置的定位标志，分别重新定位文件读写指针。
35          switch (origin) {
    // origin = SEEK_SET，要求以文件起始处作为原点设置文件读写指针。若偏移值小于零，则出
    // 错返回错误码。否则设置文件读写指针等于 offset。
36                  case 0:
37                          if (offset<0) return -EINVAL;
38                          file->f_pos=offset;
39                          break;
    // origin = SEEK_CUR，要求以文件当前读写指针处作为原点重定位读写指针。如果文件当前指
    // 针加上偏移值小于 0，则返回出错码退出。否则在当前读写指针上加上偏移值。
40                  case 1:
41                          if (file->f_pos+offset<0) return -EINVAL;
42                          file->f_pos += offset;
43                          break;
    // origin = SEEK_END，要求以文件末尾作为原点重定位读写指针。此时若文件大小加上偏移值
    // 小于零则返回出错码退出。否则重定位读写指针为文件长度加上偏移值。
44                  case 2:
45                          if ((tmp=file->f_inode->i_size+offset) < 0)
46                                  return -EINVAL;
47                          file->f_pos = tmp;
48                          break;
    // 若 origin 设置无效，返回出错码退出。
49                  default:
50                          return -EINVAL;
51          }
52          return file->f_pos;                 // 最后返回重定位后的文件读写指针值。
53  }
54
    //// 读文件系统调用。
    // 参数 fd 是文件句柄，buf 是缓冲区，count 是欲读字节数。
55  int sys_read(unsigned int fd,char * buf,int count)
56  {
57          struct file * file;
58          struct m_inode * inode;
59
    // 函数首先对参数有效性进行判断。如果文件句柄值大于程序最多打开文件数 NR_OPEN，或者
    // 需要读取的字节计数值小于 0，或者该句柄的文件结构指针为空，则返回出错码并退出。若
    // 需读取的字节数 count 等于 0，则返回 0 退出
60          if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
61                  return -EINVAL;
62          if (!count)
63                  return 0;
    // 然后验证存放数据的缓冲区内存限制。并取文件的 i 节点。用于根据该 i 节点的属性，分别
    // 调用相应的读操作函数。若是管道文件，并且是读管道文件模式，则进行读管道操作，若成
    // 功则返回读取的字节数，否则返回出错码，退出。如果是字符型文件，则进行读字符设备操
    // 作，并返回读取的字符数。如果是块设备文件，则执行块设备读操作，并返回读取的字节数。
64          verify_area(buf,count);
65          inode = file->f_inode;
66          if (inode->i_pipe)
67                  return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
68          if (S_ISCHR(inode->i_mode))
69                  return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
70          if (S_ISBLK(inode->i_mode))
```

```
71                  return block_read(inode->i_zone[0],&file->f_pos,buf,count);
    // 如果是目录文件或者是常规文件，则首先验证读取字节数 count 的有效性并进行调整（若读
    // 取字节数加上文件当前读写指针值大于文件长度，则重新设置读取字节数为 文件长度-当前
    // 读写指针值，若读取数等于 0，则返回 0 退出），然后执行文件读操作，返回读取的字节数
    // 并退出。
72          if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
73                  if (count+file->f_pos > inode->i_size)
74                          count = inode->i_size - file->f_pos;
75                  if (count<=0)
76                          return 0;
77                  return file_read(inode,file,buf,count);
78          }
    // 执行到这里，说明我们无法判断文件的属性。则打印节点文件属性，并返回出错码退出。
79          printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);
80          return -EINVAL;
81  }
82
    //// 写文件系统调用。
    // 参数 fd 是文件句柄，buf 是用户缓冲区，count 是欲写字节数。
83  int sys_write(unsigned int fd,char * buf,int count)
84  {
85          struct file * file;
86          struct m_inode * inode;
87
    // 同样地，我们首先判断函数参数的有效性。如果进程文件句柄值大于程序最多打开文件数
    // NR_OPEN，或者需要写入的字节计数小于 0，或者该句柄的文件结构指针为空，则返回出错
    // 码并退出。如果需读取的字节数 count 等于 0，则返回 0 退出
88          if (fd>=NR_OPEN || count <0 || !(file=current->filp[fd]))
89                  return -EINVAL;
90          if (!count)
91                  return 0;
    // 然后验证存放数据的缓冲区内存限制。并取文件的 i 节点。用于根据该 i 节点的属性，分别
    // 调用相应的读操作函数。若是管道文件，并且是写管道文件模式，则进行写管道操作，若成
    // 功则返回写入的字节数，否则返回出错码退出。如果是字符设备文件，则进行写字符设备操
    // 作，返回写入的字符数退出。如果是块设备文件，则进行块设备写操作，并返回写入的字节
    // 数退出。若是常规文件，则执行文件写操作，并返回写入的字节数，退出。
92          inode=file->f_inode;
93          if (inode->i_pipe)
94                  return (file->f_mode&2)?write_pipe(inode,buf,count):-EIO;
95          if (S_ISCHR(inode->i_mode))
96                  return rw_char(WRITE,inode->i_zone[0],buf,count,&file->f_pos);
97          if (S_ISBLK(inode->i_mode))
98                  return block_write(inode->i_zone[0],&file->f_pos,buf,count);
99          if (S_ISREG(inode->i_mode))
100                 return file_write(inode,file,buf,count);
    // 执行到这里，说明我们无法判断文件的属性。则打印节点文件属性，并返回出错码退出。
101         printk("(Write)inode->i_mode=%06o\n\r",inode->i_mode);
102         return -EINVAL;
103 }
104
```

12.13 程序 12-13 linux/fs/open.c

```
 1 /*
 2  *  linux/fs/open.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <string.h>       // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
 8 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
 9 #include <fcntl.h>        // 文件控制头文件。用于文件及其描述符操作控制常数符号定义。
10 #include <sys/types.h>    // 类型头文件。定义基本的系统和文件系统统计信息结构和类型。
11 #include <utime.h>        // 用户时间头文件。定义访问和修改时间结构以及 utime()原型。
12 #include <sys/stat.h>     // 文件状态头文件。含有文件状态结构 stat{}和符号常量等。
13
14 #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 数据等。
15 #include <linux/tty.h>    // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
16 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
17
18 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
19
   //// 取文件系统信息。
   // 参数 dev 是含有已安装文件系统的设备号。ubuf 是一个 ustat 结构缓冲区指针，用于存放
   // 系统返回的文件系统信息。该系统调用用于返回已安装（mounted）文件系统的统计信息。
   // 成功时返回 0，并且 ubuf 指向的 ustate 结构被添入文件系统总空闲块数和空闲 i 节点数。
   // ustat 结构定义在 include/sys/types.h 中。
20 int sys_ustat(int dev, struct ustat * ubuf)
21 {
22         return -ENOSYS;                 // 出错码：功能还未实现。
23 }
24
   //// 设置文件访问和修改时间。
   // 参数 filename 是文件名，times 是访问和修改时间结构指针。
   // 如果 times 指针不为 NULL，则取 utimbuf 结构中的时间信息来设置文件的访问和修改时间。
   // 如果 times 指针是 NULL，则取系统当前时间来设置指定文件的访问和修改时间域。
25 int sys_utime(char * filename, struct utimbuf * times)
26 {
27         struct m_inode * inode;
28         long actime,modtime;
29
   // 文件的时间信息保存在其 i 节点钟。因此我们首先根据文件名取得对应 i 节点。如果没有找
   // 到，则返回出错码。如果提供的访问和修改时间结构指针 times 不为 NULL，则从结构中读取
   // 用户设置的时间值。否则就用系统当前时间来设置文件的访问和修改时间。
30         if (!(inode=namei(filename)))
31                 return -ENOENT;
32         if (times) {
33                 actime = get_fs_long((unsigned long *) &times->actime);
34                 modtime = get_fs_long((unsigned long *) &times->modtime);
35         } else
36                 actime = modtime = CURRENT_TIME;
   // 然后修改 i 节点中的访问时间字段和修改时间字段。再设置 i 节点已修改标志，放回该 i 节
```

```
// 点，并返回 0。
37          inode->i_atime = actime;
38          inode->i_mtime = modtime;
39          inode->i_dirt = 1;
40          iput(inode);
41          return 0;
42  }
43
44  /*
45   * XXX should we use the real or effective uid?  BSD uses the real uid,
46   * so as to make this call useful to setuid programs.
47   */
    /*
     * XXX 我们该用真实用户 id（ruid）还是有效用户 id（euid）？BSD 系统使用了
     * 真实用户 id，以使该调用可以供 setuid 程序使用。
     * （注：POSIX 标准建议使用真实用户 ID）。
     * （注 1：英文注释开始的 'XXX' 表示重要提示）。
     */
    //// 检查文件的访问权限。
    // 参数 filename 是文件名，mode 是检查的访问属性，它有 3 个有效比特位组成：R_OK(值 4)、
    // W_OK(2)、X_OK(1) 和 F_OK(0) 组成，分别表示检测文件是否可读、可写、可执行和文件是
    // 否存在。如果访问允许的话，则返回 0，否则返回出错码。
48  int sys_access(const char * filename,int mode)
49  {
50          struct m_inode * inode;
51          int res, i_mode;
52
    // 文件的访问权限信息也同样保存在文件的 i 节点结构中，因此我们要先取得对应文件名的 i
    // 节点。检测的访问属性 mode 由低 3 位组成，因此需要与上八进制 0007 来清除所有高比特位。
    // 如果文件名对应的 i 节点不存在，则返回没有许可权限出错码。若 i 节点存在，则取 i 节点
    // 钟文件属性码，并放回该 i 节点。另外，57 行上语句“iput(inode);”最后放在 61 行之后。
53          mode &= 0007;
54          if (!(inode=namei(filename)))
55                  return -EACCES;                     // 出错码：无访问权限。
56          i_mode = res = inode->i_mode & 0777;
57          iput(inode);
    // 如果当前进程用户是该文件的宿主，则取文件宿主属性。否则如果当前进程用户与该文件宿
    // 主同属一组，则取文件组属性。否则，此时 res 最低 3 比特是其他人访问该文件的许可属性。
    // [[?? 这里应 res >>3 ??]
58          if (current->uid == inode->i_uid)
59                  res >>= 6;
60          else if (current->gid == inode->i_gid)
61                  res >>= 6;
    // 此时 res 的最低 3 比特是根据当前进程用户与文件的关系选择出来的访问属性位。现在我们
    // 来判断这 3 比特。如果文件属性具有参数所查询的属性位 mode，则访问许可，返回 0
62          if ((res & 0007 & mode) == mode)
63                  return 0;
64          /*
65           * XXX we are doing this test last because we really should be
66           * swapping the effective with the real user id (temporarily),
67           * and then calling suser() routine.  If we do call the
68           * suser() routine, it needs to be called last.
69           */
```

```
        /*
         * XXX 我们最后才做下面的测试，因为我们实际上需要交换有效用户 ID 和
         * 真实用户 ID（临时地），然后才调用 suser()函数。如果我们确实要调用
         * suser()函数，则需要最后才被调用。
         */
// 如果当前用户 ID 为 0（超级用户）并且屏蔽码执行位是 0 或者文件可以被任何人执行、搜
// 索，则返回 0。否则返回出错码。
70      if ((!current->uid) &&
71          (!(mode & 1) || (i_mode & 0111)))
72              return 0;
73      return -EACCES;                         // 出错码：无访问权限。
74  }
75
//// 改变当前工作目录系统调用。
// 参数 filename 是目录名。
// 操作成功则返回 0，否则返回出错码。
76  int sys_chdir(const char * filename)
77  {
78      struct m_inode * inode;
79
// 改变当前工作目录就是要求把进程任务结构的当前工作目录字段指向给定目录名的 i 节点。
// 因此我们首先取目录名的 i 节点。如果目录名对应的 i 节点不存在，则返回出错码。如果该
// i 节点不是一个目录 i 节点，则放回该 i 节点，并返回出错码。
80      if (!(inode = namei(filename)))
81              return -ENOENT;                 // 出错码：文件或目录不存在。
82      if (!S_ISDIR(inode->i_mode)) {
83              iput(inode);
84              return -ENOTDIR;                // 出错码：不是目录名。
85      }
// 然后释放进程原工作目录 i 节点，并使其指向新设置的工作目录 i 节点。返回 0。
86      iput(current->pwd);
87      current->pwd = inode;
88      return (0);
89  }
90
//// 改变根目录系统调用。
// 把指定的目录名设置成为当前进程的根目录'/'。
// 如果操作成功则返回 0，否则返回出错码。
91  int sys_chroot(const char * filename)
92  {
93      struct m_inode * inode;
94
// 该调用用于改变当前进程任务结构中的根目录字段 root，让其指向参数给定目录名的 i 节点。
// 如果目录名对应的 i 节点不存在，则返回出错码。如果该 i 节点不是目录 i 节点，则放回该
// i 节点，也返回出错码。
95      if (!(inode=namei(filename)))
96              return -ENOENT;
97      if (!S_ISDIR(inode->i_mode)) {
98              iput(inode);
99              return -ENOTDIR;
100     }
// 然后释放当前进程的根目录 i 节点，并重新设置为指定目录名的 i 节点，返回 0。
101     iput(current->root);
```

```
102         current->root = inode;
103         return (0);
104 }
105
    //// 修改文件属性系统调用。
    // 参数 filename 是文件名，mode 是新的文件属性。
    // 若操作成功则返回 0，否则返回出错码。
106 int sys_chmod(const char * filename,int mode)
107 {
108         struct m_inode * inode;
109
    // 该调用为指定文件设置新的访问属性 mode。文件的访问属性在文件名对应的 i 节点中，因此
    // 我们首先取文件名对应的 i 节点。如果 i 节点不存在，则返回出错码（文件或目录不存在）。
    // 如果当前进程的有效用户 id 与文件 i 节点的用户 id 不同，并且也不是超级用户，则放回该
    // 文件 i 节点，返回出错码（没有访问权限）。
110         if (!(inode=namei(filename)))
111                 return -ENOENT;
112         if ((current->euid != inode->i_uid) && !suser()) {
113                 iput(inode);
114                 return -EACCES;
115         }
    // 否则就重新设置该 i 节点的文件属性，并置该 i 节点已修改标志。放回该 i 节点，返回 0。
116         inode->i_mode = (mode & 07777) | (inode->i_mode & ~07777);
117         inode->i_dirt = 1;
118         iput(inode);
119         return 0;
120 }
121
    //// 修改文件宿主系统调用。
    // 参数 filename 是文件名，uid 是用户标识符(用户 ID)，gid 是组 ID。
    // 若操作成功则返回 0，否则返回出错码。
122 int sys_chown(const char * filename,int uid,int gid)
123 {
124         struct m_inode * inode;
125
    // 该调用用于设置文件 i 节点中的用户和组 ID，因此首先要取得给定文件名的 i 节点。如果文
    // 件名的 i 节点不存在，则返回出错码（文件或目录不存在）。如果当前进程不是超级用户，
    // 则放回该 i 节点，并返回出错码（没有访问权限）。
126         if (!(inode=namei(filename)))
127                 return -ENOENT;
128         if (!suser()) {
129                 iput(inode);
130                 return -EACCES;
131         }
    // 否则我们就用参数提供的值来设置文件 i 节点的用户 ID 和组 ID，并置 i 节点已经修改标志，
    // 放回该 i 节点，返回 0。
132         inode->i_uid=uid;
133         inode->i_gid=gid;
134         inode->i_dirt=1;
135         iput(inode);
136         return 0;
137 }
138
```

```
    //// 检查字符设备类型。
    // 该函数仅用于下面文件打开系统调用 sys_open()，用于检查若打开的文件是 tty 终端字符设
    // 备时，需要对当前进程的设置和对 tty 表的设置。
    // 返回 0 检测处理成功，返回-1 表示失败，对应字符设备不能打开。
139 static int check_char_dev(struct m_inode * inode, int dev, int flag)
140 {
141         struct tty_struct *tty;
142         int min;                                        // 子设备号。
143
    // 只处理主设备号是 4（/dev/ttyxx 文件）或 5（/dev/tty 文件）的情况。/dev/tty 的子设备
    // 号是 0。如果一个进程有控制终端，则它是进程控制终端设备的同义名。即/dev/tty 设备是
    // 一个虚拟设备，它对应到进程实际使用的/dev/ttyxx 设备之一。对于一个进程来说，若其有
    // 控制终端，那么它的任务结构中的 tty 字段将是 4 号设备的某一个子设备号。
    // 如果打开操作的文件是 /dev/tty（即 MAJOR(dev) = 5），那么我们令 min = 进程任务结构
    // 中的 tty 字段，即取 4 号设备的子设备号。否则如果打开的是某个 4 号设备，则直接取其子
    // 设备号。如果得到的 4 号设备子设备号小于 0，那么说明进程没有控制终端，或者设备号错
    // 误，则返回 -1，表示由于进程没有控制终端，或者不能打开这个设备。
144         if (MAJOR(dev) == 4 || MAJOR(dev) == 5) {
145                 if (MAJOR(dev) == 5)
146                         min = current->tty;
147                 else
148                         min = MINOR(dev);
149                 if (min < 0)
150                         return -1;
    // 主伪终端设备文件只能被进程独占使用。如果子设备号表明是一个主伪终端，并且该打开文件
    // i 节点引用计数大于 1，则说明该设备已被其他进程使用。因此不能再打开该字符设备文件，
    // 于是返回 -1。否则，我们让 tty 结构指针 tty 指向 tty 表中对应结构项。若打开文件操作标
    // 志 flag 中不含无需控制终端标志 O_NOCTTY，并且进程是进程组首领，并且当前进程没有控制
    // 终端，并且 tty 结构中 session 字端为 0（表示该终端还不是任何进程组的控制终端），那么
    // 就允许为进程设置这个终端设备 min 为其控制终端。于是设置进程任务结构终端设备号字段
    // tty 值等于 min，并且设置对应 tty 结构的会话号 session 和进程组号 pgrp 分别等于进程的会
    // 话号和进程组号。
151                 if ((IS_A_PTY_MASTER(min)) && (inode->i_count>1))
152                         return -1;
153                 tty = TTY_TABLE(min);
154                 if (!(flag & O_NOCTTY) &&
155                     current->leader &&
156                     current->tty<0 &&
157                     tty->session==0) {
158                         current->tty = min;
159                         tty->session= current->session;
160                         tty->pgrp = current->pgrp;
161                 }
    // 如果打开文件操作标志 flag 中含有 O_NONBLOCK（非阻塞）标志，则我们需要对该字符终端
    // 设备进行相关设置，设置为满足读操作需要读取的最少字符数为 0，设置超时定时值为 0，
    // 并把终端设备设置成非规范模式。非阻塞方式只能工作于非规范模式。在此模式下当 VMIN
    // 和 VTIME 均设置为 0 时，辅助队列中有多少支付进程就读取多少字符，并立刻返回。参见
    // include/termios.h 文件后的说明。
162                 if (flag & O_NONBLOCK) {
163                         TTY_TABLE(min)->termios.c_cc[VMIN] =0;
164                         TTY_TABLE(min)->termios.c_cc[VTIME] =0;
165                         TTY_TABLE(min)->termios.c_lflag &= ~ICANON;
166                 }
```

```
167         }
168         return 0;
169 }
170
    //// 打开（或创建）文件系统调用。
    // 参数 filename 是文件名，flag 是打开文件标志，它可取值：O_RDONLY（只读）、O_WRONLY
    // （只写）或 O_RDWR（读写），以及 O_CREAT（创建）、O_EXCL（被创建文件必须不存在）、
    // O_APPEND（在文件尾添加数据）等其他一些标志的组合，如果本调用创建了一个新文件，则
    // mode 就用于指定文件的许可属性。这些属性有 S_IRWXU（文件宿主具有读、写和执行权限）、
    // S_IRUSR（用户具有读文件权限）、S_IRWXG（组成员具有读、写和执行权限）等等。对于新
    // 创建的文件，这些属性只应用于将来对文件的访问，创建了只读文件的打开调用也将返回一
    // 个可读写的文件句柄。如果调用操作成功，则返回文件句柄(文件描述符)，否则返回出错码。
    // 参见 sys/stat.h、fcntl.h。
171 int sys_open(const char * filename,int flag,int mode)
172 {
173         struct m_inode * inode;
174         struct file * f;
175         int i,fd;
176
    // 首先对参数进行处理。将用户设置的文件模式和进程模式屏蔽码相与，产生许可的文件模式。
    // 为了为打开文件建立一个文件句柄，需要搜索进程结构中文件结构指针数组，以查找一个空
    // 闲项。空闲项的索引号 fd 即是句柄值。若已经没有空闲项，则返回出错码（参数无效）。
177         mode &= 0777 & ~current->umask;
178         for(fd=0 ; fd<NR_OPEN ; fd++)
179                 if (!current->filp[fd])                         // 找到空闲项。
180                         break;
181         if (fd>=NR_OPEN)
182                 return -EINVAL;
    // 然后我们设置当前进程的执行时关闭文件句柄（close_on_exec）位图，复位对应的比特位。
    // close_on_exec 是一个进程所有文件句柄的位图标志。每个比特位代表一个打开着的文件描
    // 述符，用于确定在调用系统调用 execve()时需要关闭的文件句柄。当程序使用 fork()函数
    // 创建了一个子进程时，通常会在该子进程中调用 execve()函数加载执行另一个新程序。此时
    // 子进程中开始执行新程序。若一个文件句柄在 close_on_exec 中的对应比特位被置位，那么
    // 在执行 execve()时该对应文件句柄将被关闭，否则该文件句柄将始终处于打开状态。当打开
    // 一个文件时，默认情况下文件句柄在子进程中也处于打开状态。因此这里要复位对应比特位。
    // 然后为打开文件在文件表中寻找一个空闲结构项。我们令 f 指向文件表数组开始处。搜索空
    // 闲文件结构项（引用计数为 0 的项），若已经没有空闲文件表结构项，则返回出错码。另外，
    // 第 184 行上的指针赋值 "0+file_table " 等同于 "file_table" 和 "&file_table[0]"。
    // 不过这样写可能更能明了一些。
183         current->close_on_exec &= ~(1<<fd);
184         f=0+file_table;
185         for (i=0 ; i<NR_FILE ; i++,f++)
186                 if (!f->f_count) break;
187         if (i>=NR_FILE)
188                 return -EINVAL;
    // 此时我们让进程对应文件句柄 fd 的文件结构指针指向搜索到的文件结构，并令文件引用计数
    // 递增 1。然后调用函数 open_namei()执行打开操作，若返回值小于 0，则说明出错，于是释放
    // 刚申请到的文件结构，返回出错码 i。若文件打开操作成功，则 inode 是已打开文件的 i 节点
    // 指针。
189         (current->filp[fd]=f)->f_count++;
190         if ((i=open_namei(filename,flag,mode,&inode))<0) {
191                 current->filp[fd]=NULL;
192                 f->f_count=0;
```

```
193                 return i;
194         }
    // 根据已打开文件 i 节点的属性字段，我们可以知道文件的类型。对于不同类型的文件，我们
    // 需要作一些特别处理。 如果打开的是字符设备文件，那么我们就要调用 check_char_dev()
    // 函数来检查当前进程是否能打开这个字符设备文件。如果允许（函数返回 0），那么在
    // check_char_dev()中会根据具体文件打开标志为进程设置控制终端。 如果不允许打开
    // 使用该字符设备文件，那么我们只能释放上面申请的文件项和句柄资源，返回出错码。
195 /* ttys are somewhat special (ttyxx major==4, tty major==5) */
    /* ttys 有些特殊（ttyxx 的主设备号==4，tty 的主设备号==5）*/
196         if (S_ISCHR(inode->i_mode))
197                 if (check_char_dev(inode,inode->i_zone[0],flag)) {
198                         iput(inode);
199                         current->filp[fd]=NULL;
200                         f->f_count=0;
201                         return -EAGAIN;          // 出错号：资源暂时不可用。
202                 }
    // 如果打开的是块设备文件，则检查盘片是否更换过。若更换过则需要让高速缓冲区中该设备
    // 的所有缓冲块失效。
203 /* Likewise with block-devices: check for floppy_change */
    /* 同样对于块设备文件：需要检查盘片是否被更换 */
204         if (S_ISBLK(inode->i_mode))
205                 check_disk_change(inode->i_zone[0]);
    // 现在我们初始化打开文件的文件结构。设置文件结构属性和标志，置句柄引用计数为 1，并
    // 设置 i 节点字段为打开文件的 i 节点，初始化文件读写指针为 0。最后返回文件句柄号。
206         f->f_mode = inode->i_mode;
207         f->f_flags = flag;
208         f->f_count = 1;
209         f->f_inode = inode;
210         f->f_pos = 0;
211         return (fd);
212 }
213
    //// 创建文件系统调用。
    // 参数 pathname 是路径名，mode 与上面的 sys_open()函数相同。
    // 成功则返回文件句柄，否则返回出错码。
214 int sys_creat(const char * pathname, int mode)
215 {
216         return sys_open(pathname, O_CREAT | O_TRUNC, mode);
217 }
218
    // 关闭文件系统调用。
    // 参数 fd 是文件句柄。
    // 成功则返回 0，否则返回出错码。
219 int sys_close(unsigned int fd)
220 {
221         struct file * filp;
222
    // 首先检查参数有效性。若给出的文件句柄值大于程序同时能打开的文件数 NR_OPEN，则返回
    // 出错码（参数无效）。然后复位进程的执行时关闭文件句柄位图对应位。若该文件句柄对应
    // 的文件结构指针是 NULL，则返回出错码。
223         if (fd >= NR_OPEN)
224                 return -EINVAL;
225         current->close_on_exec &= ~(1<<fd);
```

```
226         if (!(filp = current->filp[fd]))
227                 return -EINVAL;
    // 现在置该文件句柄的文件结构指针为 NULL。若在关闭文件之前，对应文件结构中的句柄引用
    // 计数已经为 0，则说明内核出错，停机。否则将对应文件结构的引用计数减 1。此时如果它还
    // 不为 0，则说明有其他进程正在使用该文件，于是返回 0（成功）。如果引用计数已等于 0，
    // 说明该文件已经没有进程引用，该文件结构已变为空闲。则释放该文件 i 节点，返回 0。
228         current->filp[fd] = NULL;
229         if (filp->f_count == 0)
230                 panic("Close: file count is 0");
231         if (--filp->f_count)
232                 return (0);
233         iput(filp->f_inode);
234         return (0);
235 }
236
```

12.14 程序 12-14 linux/fs/exec.c

```
 1 /*
 2  *  linux/fs/exec.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * #!-checking implemented by tytso.
 9  */
   /*
    * #!开始的脚本程序的检测代码部分是由 tytso 实现的。
    */
10
11 /*
12  * Demand-loading implemented 01.12.91 - no need to read anything but
13  * the header into memory. The inode of the executable is put into
14  * "current->executable", and page faults do the actual loading. Clean.
15  *
16  * Once more I can proudly say that linux stood up to being changed: it
17  * was less than 2 hours work to get demand-loading completely implemented.
18  */
   /*
    * 需求时加载实现于 1991.12.1 - 只需将执行文件头部读进内存而无须将整个
    * 执行文件都加载进内存。执行文件的 i 节点被放在当前进程的可执行字段中
    * "current->executable"，页异常会进行执行文件的实际加载操作。这很完美。
    *
    * 我可以再一次自豪地说，linux 经得起修改：只用了不到 2 小时的工作时间就
    * 完全实现了需求加载处理。
    */
19
20 #include <signal.h>       // 信号头文件。定义信号符号常量，信号结构及信号操作函数原型。
21 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
22 #include <string.h>       // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
23 #include <sys/stat.h>     // 文件状态头文件。含有文件状态结构 stat{}和符号常量等。
24 #include <a.out.h>        // a.out 头文件。定义了 a.out 执行文件格式和一些宏。
25
26 #include <linux/fs.h>     // 文件系统头文件。定义文件表结构（file、m_inode）等。
27 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
28 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
29 #include <linux/mm.h>     // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
30 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
31
32 extern int sys_exit(int exit_code);     // 退出程序系统调用。
33 extern int sys_close(int fd);           // 关闭文件系统调用。
34
35 /*
36  * MAX_ARG_PAGES defines the number of pages allocated for arguments
37  * and envelope for the new program. 32 should suffice, this gives
38  * a maximum env+arg of 128kB !
```

```
39  */
    /*
     * MAX_ARG_PAGES 定义了为新程序分配的给参数和环境变量使用的最大内存页数。
     * 32 页内存应该足够了，这使得环境和参数（env+arg）空间的总和达到 128kB!
     */

40  #define MAX_ARG_PAGES 32
41
    //// 使用库文件系统调用。
    // 参数：library - 库文件名。
    // 为进程选择一个库文件，并替换进程当前库文件 i 节点字段值为这里指定库文件名的 i 节点
    // 指针。如果 library 指针为空，则把进程当前的库文件释放掉。
    // 返回：成功返回 0，否则返回出错码。
42  int sys_uselib(const char * library)
43  {
44          struct m_inode * inode;
45          unsigned long base;
46
    // 首先判断当前进程是否普通进程。这是通过查看当前进程的空间长度来做到的。因为普通进
    // 程的空间长度被设置为 TASK_SIZE（64MB）。因此若进程逻辑地址空间长度不等于 TASK_SIZE
    // 则返回出错码（无效参数）。否则取库文件 i 节点 inode。若库文件名指针空，则设置 inode
    // 等于 NULL。
47          if (get_limit(0x17) != TASK_SIZE)
48                  return -EINVAL;
49          if (library) {
50                  if (!(inode=namei(library)))            /* get library inode */
51                          return -ENOENT;                 /* 取库文件 i 节点 */
52          } else
53                  inode = NULL;
54  /* we should check filetypes (headers etc), but we don't */
    /* 我们应该检查一下文件类型（如头部信息等），但是我们还没有这样做。 */
    // 然后放回进程原库文件 i 节点，并预置进程库 i 节点字段为空。接着取得进程的库代码所在
    // 位置，并释放原库代码的页表和所占用的内存页面。最后让进程库 i 节点字段指向新库 i 节
    // 点，并返回 0（成功）。
55          iput(current->library);
56          current->library = NULL;
57          base = get_base(current->ldt[2]);
58          base += LIBRARY_OFFSET;
59          free_page_tables(base,LIBRARY_SIZE);
60          current->library = inode;
61          return 0;
62  }
63
64  /*
65   * create_tables() parses the env- and arg-strings in new user
66   * memory and creates the pointer tables from them, and puts their
67   * addresses on the "stack", returning the new stack pointer value.
68   */
    /*
     * create_tables()函数在新任务内存中解析环境变量和参数字符串，由此
     * 创建指针表，并将它们的地址放到"栈"上，然后返回新栈的指针值。
     */
    //// 在新任务栈中创建参数和环境变量指针表。
```

```
   // 参数：p -数据段中参数和环境信息偏移指针；argc - 参数个数；envc - 环境变量个数。
   // 返回：栈指针值。
69 static unsigned long * create_tables(char * p,int argc,int envc)
70 {
71         unsigned long *argv,*envp;
72         unsigned long * sp;
73
   // 栈指针是以 4 字节（1 节）为边界进行寻址的，因此这里需让 sp 为 4 的整数倍值。此时 sp
   // 位于参数环境表的末端。然后我们先把 sp 向下（低地址方向）移动，在栈中空出环境变量
   // 指针占用的空间，并让环境变量指针 envp 指向该处。多空出的一个位置用于在最后存放一
   // 个 NULL 值。下面指针加 1，sp 将递增指针宽度字节值（4 字节）。再把 sp 向下移动，空出
   // 命令行参数指针占用的空间，并让 argv 指针指向该处。同样，多空处的一个位置用于存放
   // 一个 NULL 值。此时 sp 指向参数指针块的起始处，我们将环境参数块指针 envp 和命令行参
   // 数块指针以及命令行参数个数值分别压入栈中。
74         sp = (unsigned long *) (0xfffffffc & (unsigned long) p);
75         sp -= envc+1;                                   // 即 sp = sp - (envc+1);
76         envp = sp;
77         sp -= argc+1;
78         argv = sp;
79         put_fs_long((unsigned long)envp,--sp);
80         put_fs_long((unsigned long)argv,--sp);
81         put_fs_long((unsigned long)argc,--sp);
   // 再将命令行各参数指针和环境变量各指针分别放入前面空出来的相应地方，最后分别放置一
   // 个 NULL 指针。
82         while (argc-->0) {
83                 put_fs_long((unsigned long) p,argv++);
84                 while (get_fs_byte(p++)) /* nothing */ ; // p 指针指向下一个参数串。
85         }
86         put_fs_long(0,argv);
87         while (envc-->0) {
88                 put_fs_long((unsigned long) p,envp++);
89                 while (get_fs_byte(p++)) /* nothing */ ; // p 指针指向下一个参数串。
90         }
91         put_fs_long(0,envp);
92         return sp;                              // 返回构造的当前新栈指针。
93 }
94
95 /*
96  * count() counts the number of arguments/envelopes
97  */
   /*
    * count()函数计算命令行参数/环境变量的个数。
    */
   //// 计算参数个数。
   // 参数：argv - 参数指针数组，最后一个指针项是 NULL。
   // 统计参数指针数组中指针的个数。关于函数参数传递指针的指针的作用，请参见程序
   // kernel/sched.c 中第 171 行前的注释。
   // 返回：参数个数。
98 static int count(char ** argv)
99 {
100         int i=0;
101         char ** tmp;
102
```

```
103         if (tmp = argv)
104                 while (get_fs_long((unsigned long *) (tmp++)))
105                         i++;
106
107         return i;
108 }
109
110 /*
111  * 'copy_string()' copies argument/envelope strings from user
112  * memory to free pages in kernel mem. These are in a format ready
113  * to be put directly into the top of new user memory.
114  *
115  * Modified by TYT, 11/24/91 to add the from_kmem argument, which specifies
116  * whether the string and the string array are from user or kernel segments:
117  *
118  * from_kmem     argv *        argv **
119  *    0          user space    user space
120  *    1          kernel space  user space
121  *    2          kernel space  kernel space
122  *
123  * We do this by playing games with the fs segment register.  Since it
124  * it is expensive to load a segment register, we try to avoid calling
125  * set_fs() unless we absolutely have to.
126  */
    /*
     * 'copy_string()'函数从用户内存空间拷贝参数/环境字符串到内核空闲页面中。
     * 这些已具有直接放到新用户内存中的格式。
     *
     * 由 TYT(Tytso)于 1991.11.24 日修改，增加了 from_kmem 参数，该参数指明了
     * 字符串或字符串数组是来自用户段还是内核段。
     *
     * from_kmem   指针 argv *    字符串 argv **
     *    0          用户空间       用户空间
     *    1          内核空间       用户空间
     *    2          内核空间       内核空间
     *
     * 我们是通过巧妙处理 fs 段寄存器来操作的。由于加载一个段寄存器代价太高，
     * 所以我们尽量避免调用 set_fs()，除非实在必要。
     */
    //// 复制指定个数的参数字符串到参数和环境空间中。
    // 参数：argc - 欲添加的参数个数；argv - 参数指针数组；page - 参数和环境空间页面指针
    // 数组。p -参数表空间中偏移指针，始终指向已复制串的头部；from_kmem -字符串来源标志。
    // 在 do_execve()函数中，p 初始化为指向参数表(128kB)空间的最后一个长字处，参数字符串
    // 是以堆栈操作方式逆向往其中复制存放的。因此 p 指针会随着复制信息的增加而逐渐减小，
    // 并始终指向参数字符串的头部。字符串来源标志 from_kmem 应该是 TYT 为了给 execve()增添
    // 执行脚本文件的功能而新加的参数。当没有运行脚本文件的功能时，所有参数字符串都在用
    // 户数据空间中。
    // 返回：参数和环境空间当前头部指针。若出错则返回 0。
127 static unsigned long copy_strings(int argc,char ** argv,unsigned long *page,
128                 unsigned long p, int from_kmem)
129 {
130         char *tmp, *pag;
131         int len, offset = 0;
```

```
132         unsigned long old_fs, new_fs;
133
    // 首先取当前段寄存器 ds（指向内核数据段）和 fs 值，分别保存到变量 new_fs 和 old_fs 中。
    // 如果字符串和字符串数组（指针）来自内核空间，则设置 fs 段寄存器指向内核数据段。
134         if (!p)
135                 return 0;       /* bullet-proofing */  /* 偏移指针验证 */
136         new_fs = get_ds();
137         old_fs = get_fs();
138         if (from_kmem==2)           // 若串及其指针在内核空间则设置 fs 指向内核空间。
139                 set_fs(new_fs);
    // 然后循环处理各个参数，从最后一个参数逆向开始复制，复制到指定偏移地址处。在循环中，
    // 首先取需要复制的当前字符串指针。如果字符串在用户空间而字符串数组（字符串指针）在
    // 内核空间，则设置 fs 段寄存器指向内核数据段（ds）。并在内核数据空间中取了字符串指针
    // tmp 之后就立刻恢复 fs 段寄存器原值（fs 再指回用户空间）。否则不用修改 fs 值而直接从
    // 用户空间取字符串指针到 tmp。
140         while (argc-- > 0) {
141                 if (from_kmem == 1)      // 若串指针在内核空间，则 fs 指向内核空间。
142                         set_fs(new_fs);
143                 if (!(tmp = (char *)get_fs_long(((unsigned long *)argv)+argc)))
144                         panic("argc is wrong");
145                 if (from_kmem == 1)      // 若串指针在内核空间，则 fs 指回用户空间。
146                         set_fs(old_fs);
    // 然后从用户空间取该字符串，并计算该参数字符串长度 len。 此后 tmp 指向该字符串末端。
    // 如果该字符串长度超过此时参数和环境空间中还剩余的空闲长度，则空间不够了。于是恢复
    // fs 段寄存器值（如果被改变的话）并返回 0。不过因为参数和环境空间留有 128KB，所以通
    // 常不可能发生这种情况。
147                 len=0;                          /* remember zero-padding */
148                 do {                            /* 我们知道串是以 NULL 字节结尾的 */
149                         len++;
150                 } while (get_fs_byte(tmp++));
151                 if (p-len < 0) {                /* this shouldn't happen - 128kB */
152                         set_fs(old_fs);         /* 不会发生-因为有 128kB 的空间 */
153                         return 0;
154                 }
    // 接着我们逆向逐个字符地把字符串复制到参数和环境空间末端处。在循环复制字符串的字符
    // 过程中，我们首先要判断参数和环境空间中相应位置处是否已经有内存页面。如果还没有就
    // 先为其申请 1 页内存页面。偏移量 offset 被用作为在一个页面中的当前指针偏移值。 因为
    // 刚开始执行本函数时，偏移变量 offset 被初始化为 0，所以(offset-1 < 0)肯定成立而使得
    // offset 重新被设置为当前 p 指针在页面范围内的偏移值。
155                 while (len) {
156                         --p; --tmp; --len;
157                         if (--offset < 0) {
158                                 offset = p % PAGE_SIZE;
159                                 if (from_kmem==2) // 若串在内核空间则 fs 指回用户空间。
160                                         set_fs(old_fs);
    // 如果当前偏移值 p 所在的串空间页面指针数组项 page[p/PAGE_SIZE] ==0，表示此时 p 指针
    // 所处的空间内存页面还不存在，则需申请一空闲内存页，并将该页面指针填入指针数组，同
    // 时也使页面指针 pag 指向该新页面。若申请不到空闲页面则返回 0。
161                                 if (!(pag = (char *) page[p/PAGE_SIZE]) &&
162                                     !(pag = (char *) page[p/PAGE_SIZE] =
163                                       (unsigned long *) get_free_page()))
164                                         return 0;
165                                 if (from_kmem==2) // 若串在内核空间则 fs 指向内核空间。
```

```
166                                     set_fs(new_fs);
167
168                             }
    // 然后从 fs 段中复制字符串的 1 字节到参数和环境空间内存页面 pag 的 offset 处。
169                             *(pag + offset) = get_fs_byte(tmp);
170                     }
171             }
    // 如果字符串和字符串数组在内核空间，则恢复 fs 段寄存器原值。最后，返回参数和环境空
    // 间中已复制参数的头部偏移值。
172             if (from_kmem==2)
173                     set_fs(old_fs);
174             return p;
175 }
176
    //// 修改任务的局部描述符表内容。
    // 修改局部描述符表 LDT 中描述符的段基址和段限长，并将参数和环境空间页面放置在数据段
    // 末端。
    // 参数：text_size - 执行文件头部中 a_text 字段给出的代码段长度值；
    // page - 参数和环境空间页面指针数组。
    // 返回：数据段限长值（64MB）。
177 static unsigned long change_ldt(unsigned long text_size,unsigned long * page)
178 {
179             unsigned long code_limit,data_limit,code_base,data_base;
180             int i;
181
    // 首先把代码和数据段长度均设置为 64MB。 然后取当前进程局部描述符表代码段描述符中代
    // 码段基址。代码段基址与数据段基址相同。再使用这些新值重新设置局部表中代码段和数据
    // 段描述符中的基址和段限长。这里请注意，由于被加载的新程序的代码和数据段基址与原程
    // 序的相同，因此没有必要再重复去设置它们，即第 186 和 188 行上的两条设置段基址的语句
    // 多余，可省略。
182             code_limit = TASK_SIZE;
183             data_limit = TASK_SIZE;
184             code_base = get_base(current->ldt[1]);  // include/linux/sched.h，第 226 行。
185             data_base = code_base;
186             set_base(current->ldt[1],code_base);
187             set_limit(current->ldt[1],code_limit);
188             set_base(current->ldt[2],data_base);
189             set_limit(current->ldt[2],data_limit);
190 /* make sure fs points to the NEW data segment */
    /* 要确信 fs 段寄存器已指向新的数据段 */
    // fs 段寄存器中放入局部表数据段描述符的选择符(0x17)。即默认情况下 fs 都指向任务数据段。
191             __asm__("pushl $0x17\n\tpop %%fs"::);
    // 然后将参数和环境空间已存放数据的页面（最多有 MAX_ARG_PAGES 页，128kB）放到数据段末
    // 端。 方法是从进程空间库代码位置开始处逆向一页一页地放。库文件代码占用进程空间最后
    // 4MB。函数 put_dirty_page()用于把物理页面映射到进程逻辑空间中。在 mm/memory.c 中。
192             data_base += data_limit - LIBRARY_SIZE;
193             for (i=MAX_ARG_PAGES-1 ; i>=0 ; i--) {
194                     data_base -= PAGE_SIZE;
195                     if (page[i])                            // 若该页面存在，就放置该页面。
196                             put_dirty_page(page[i],data_base);
197             }
198             return data_limit;                              // 最后返回数据段限长（64MB）。
199 }
```

```
200
201 /*
202  * 'do_execve()' executes a new program.
203  *
204  * NOTE! We leave 4MB free at the top of the data-area for a loadable
205  * library.
206  */
    /*
     * 'do_execve()'函数执行一个新程序。
     */
    //// execve()系统中断调用函数。加载并执行子进程（其他程序）。
    // 该函数是系统中断调用（int 0x80）功能号__NR_execve 调用的函数。函数的参数是进入系统
    // 调用处理过程后直到调用本系统调用处理过程（system_call.s 第 200 行）和调用本函数之前
    // （system_call.s，第 203 行）逐步压入栈中的值。这些值包括：
    // ① 第 86--88 行入堆的 edx、ecx 和 ebx 寄存器值，分别对应**envp、**argv 和*filename；
    // ② 第 94 行调用 sys_call_table 中 sys_execve 函数（指针）时压入栈的函数返回地址（tmp）；
    // ③ 第 202 行在调用本函数 do_execve 前入栈的指向栈中调用系统中断的程序代码指针 eip。
    // 参数：
    // eip - 调用系统中断的程序代码指针，参见 kernel/system_call.s 程序开始部分的说明；
    // tmp - 系统中断中在调用_sys_execve 时的返回地址，无用；
    // filename - 被执行程序文件名指针；
    // argv - 命令行参数指针数组的指针；
    // envp - 环境变量指针数组的指针。
    // 返回：如果调用成功，则不返回；否则设置出错号，并返回-1。
207 int do_execve(unsigned long * eip,long tmp,char * filename,
208         char ** argv, char ** envp)
209 {
210         struct m_inode * inode;
211         struct buffer_head * bh;
212         struct exec ex;
213         unsigned long page[MAX_ARG_PAGES];          // 参数和环境串空间页面指针数组。
214         int i,argc,envc;
215         int e_uid, e_gid;                           // 有效用户 ID 和有效组 ID。
216         int retval;
217         int sh_bang = 0;                            // 控制是否需要执行脚本程序。
218         unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4; // p 指向参数和环境空间的最后部。
219
    // 在正式设置执行文件的运行环境之前，让我们先干些杂事。内核准备了 128KB（32 个页面）
    // 空间来存放化执行文件的命令行参数和环境字符串。上行把 p 初始设置成位于 128KB 空间的
    // 最后 1 个长字处。在初始参数和环境空间的操作过程中，p 将用来指明在 128KB 空间中的当
    // 前位置。
    // 另外，参数 eip[1]是调用本次系统调用的原用户程序代码段寄存器 CS 值，其中的段选择符
    // 当然必须是当前任务的代码段选择符（0x000f）。若不是该值，那么 CS 只能会是内核代码
    // 段的选择符 0x0008。 但这是绝对不允许的，因为内核代码是常驻内存而不能被替换掉的。
    // 因此下面根据 eip[1] 的值确认是否符合正常情况。然后再初始化 128KB 的参数和环境串空
    // 间，把所有字节清零，并取出执行文件的 i 节点。再根据函数参数分别计算出命令行参数和
    // 环境字符串的个数 argc 和 envc。另外，执行文件必须是常规文件。
220         if ((0xffff & eip[1]) != 0x000f)
221                 panic("execve called from supervisor mode");
222         for (i=0 ; i<MAX_ARG_PAGES ; i++)       /* clear page-table */
223                 page[i]=0;
224         if (!(inode=namei(filename)))           /* get executables inode */
225                 return -ENOENT;
```

```
226         argc = count(argv);                     // 命令行参数个数。
227         envc = count(envp);                     // 环境字符串变量个数。
228
229 restart_interp:
230         if (!S_ISREG(inode->i_mode)) {    /* must be regular file */
231                 retval = -EACCES;
232                 goto exec_error2;         //若不是常规文件则置出错码，跳转到 376 行。
233         }
    // 下面检查当前进程是否有权运行指定的执行文件。即根据执行文件 i 节点中的属性，看看本
    // 进程是否有权执行它。在把执行文件 i 节点的属性字段值取到 i 中后，我们首先查看属性中
    // 是否设置了“设置-用户-ID”（set-user_id）标志 和“设置-组-ID”（set-group-id）标
    // 志。这两个标志主要是让一般用户能够执行特权用户（如超级用户 root）的程序，例如改变
    // 密码的程序 passwd 等。如果 set-user-id 标志置位，则后面执行进程的有效用户 ID（euid）
    // 就设置成执行文件的用户 ID，否则设置成当前进程的 euid。如果执行文件 set-group-id 被
    // 置位的话，则执行进程的有效组 ID（egid）就设置为执行文件的组 ID。否则设置成当前进程
    // 的 egid。这里暂时把这两个判断出来的值保存在变量 e_uid 和 e_gid 中。
234         i = inode->i_mode;
235         e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
236         e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;

    // 现在根据进程的 euid 和 egid 和执行文件的访问属性进行比较。如果执行文件属于运行进程
    // 的用户，则把文件属性值 i 右移 6 位，此时其最低 3 位是文件宿主的访问权限标志。否则的
    // 话如果执行文件与当前进程的用户属于同组，则使属性值最低 3 位是执行文件组用户的访问
    // 权限标志。否则此时属性字最低 3 位就是其他用户访问该执行文件的权限。
    // 然后我们根据属性字 i 的最低 3 比特值来判断当前进程是否有权限运行这个执行文件。如果
    // 选出的相应用户没有运行改文件的权力（位 0 是执行权限），并且其他用户也没有任何权限
    // 或者当前进程用户不是超级用户，则表明当前进程没有权力运行这个执行文件。于是置不可
    // 执行出错码，并跳转到 exec_error2 处去作退出处理。
237         if (current->euid == inode->i_uid)
238                 i >>= 6;
239         else if (in_group_p(inode->i_gid))
240                 i >>= 3;
241         if (!(i & 1) &&
242             !((inode->i_mode & 0111) && suser())) {
243                 retval = -ENOEXEC;
244                 goto exec_error2;
245         }
    // 程序执行到这里，说明当前进程有运行指定执行文件的权限。因此从这里开始我们需要取出
    // 执行文件头部数据并根据其中的信息来分析设置运行环境，或者运行另一个 shell 程序来执
    // 行脚本程序。首先读取执行文件第 1 块数据到高速缓冲块中。 并复制缓冲块数据到 ex 中。
    // 如果执行文件开始的两个字节是字符'#!'，则说明执行文件是一个脚本文本文件。如果想运
    // 行脚本文件，我们就需要执行脚本文件的解释程序（例如 shell 程序）。通常脚本文件的第
    // 一行文本为“#! /bin/bash”。它指明了运行脚本文件需要的解释程序。运行方法是从脚本
    // 文件第 1 行（带字符'#!'）中取出其中的解释程序名及后面的参数（若有的话），然后将这
    // 些参数和脚本文件名放进执行文件（此时是解释程序）的命令行参数空间中。在这之前我们
    // 当然需要先把函数指定的原有命令行参数和环境字符串放到 128KB 空间中，而这里建立起来
    // 的命令行参数则放到它们前面位置处（因为是逆向放置）。最后让内核执行脚本文件的解释
    // 程序。下面就是在设置好解释程序的脚本文件名等参数后，取出解释程序的 i 节点并跳转到
    // 229 行去执行解释程序。由于我们需要跳转到执行过的代码 229 行去，因此在下面确认并处
    // 理了脚本文件之后需要设置一个禁止再次执行下面的脚本处理代码标志 sh_bang 。在后面的
    // 代码中该标志也用来表示我们已经设置好执行文件的命令行参数，不要重复设置。
246         if (!(bh = bread(inode->i_dev,inode->i_zone[0]))) {
247                 retval = -EACCES;
```

```
248                 goto exec_error2;
249         }
250         ex = *((struct exec *) bh->b_data);     /* read exec-header */
251         if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) {
252                 /*
253                  * This section does the #! interpretation.
254                  * Sorta complicated, but hopefully it will work.  -TYT
255                  */
                    /*
                     * 这部分处理对'#!'的解释，有些复杂，但希望能工作。-TYT
                     */
256
257                 char buf[128], *cp, *interp, *i_name, *i_arg;
258                 unsigned long old_fs;
259
    // 从这里开始，我们从脚本文件中提取解释程序名及其参数，并把解释程序名、解释程序的参数
    // 和脚本文件名组合放入环境参数块中。首先复制脚本文件头 1 行字符'#!'后面的字符串到 buf
    // 中，其中含有脚本解释程序名（例如/bin/sh），也可能还包含解释程序的几个参数。然后对
    // buf 中的内容进行处理。删除开始的空格、制表符。
260                 strncpy(buf, bh->b_data+2, 127);
261                 brelse(bh);
262                 iput(inode);                    // 释放缓冲块并放回脚本文件 i 节点。
263                 buf[127] = '\0';
264                 if (cp = strchr(buf, '\n')) {
265                         *cp = '\0';             // 第 1 个换行符换成 NULL 并去掉空格制表符。
266                         for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
267                 }
268                 if (!cp || *cp == '\0') {       // 若该行没有其他内容，则出错。
269                         retval = -ENOEXEC;      /* No interpreter name found */
270                         goto exec_error1;       /* 没有找到脚本解释程序名 */
271                 }
    // 此时我们得到了开头是脚本解释程序名的一行内容（字符串）。下面分析该行。首先取第一个
    // 字符串，它应该是解释程序名，此时 i_name 指向该名称。若解释程序名后还有字符，则它们应
    // 该是解释程序的参数串，于是令 i_arg 指向该串。
272                 interp = i_name = cp;
273                 i_arg = 0;
274                 for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++) {
275                         if (*cp == '/')
276                                 i_name = cp+1;
277                 }
278                 if (*cp) {
279                         *cp++ = '\0';           // 解释程序名尾添加 NULL 字符。
280                         i_arg = cp;             // i_arg 指向解释程序参数。
281                 }
282                 /*
283                  * OK, we've parsed out the interpreter name and
284                  * (optional) argument.
285                  */
                    /*
                     * OK，我们已经解析出解释程序的文件名以及(可选的)参数。
                     */
    // 现在我们要把上面解析出来的解释程序名 i_name 及其参数 i_arg 和脚本文件名作为解释程
    // 序的参数放进环境和参数块中。不过首先我们需要把函数提供的原来一些参数和环境字符串
```

```
    // 先放进去，然后再放这里解析出来的。例如对于命令行参数来说，如果原来的参数是"-arg1
    //  -arg2"、解释程序名是"bash"、其参数是"-iarg1 -iarg2"、脚本文件名（即原来的执行文
    // 件名）是"example.sh"，那么在放入这里的参数之后，新的命令行类似于这样：
    //          “bash -iarg1 -iarg2 example.sh -arg1 -arg2”
    // 这里我们把 sh_bang 标志置上，然后把函数参数提供的原有参数和环境字符串放入到空间中。
    // 环境字符串和参数个数分别是 envc 和 argc-1 个。少复制的一个原有参数是原来的执行文件
    // 名，即这里的脚本文件名。[[?? 可以看出，实际上我们不需要去另行处理脚本文件名，即这
    // 里完全可以复制 argc 个参数，包括原来执行文件名（即现在的脚本文件名）。因为它位于同
    // 一个位置上 ]]。注意！这里指针 p 随着复制信息增加而逐渐向小地址方向移动，因此这两个
    // 复制串函数执行完后，环境参数串信息块位于程序命令行参数串信息块的上方，并且 p 指向
    // 程序的第 1 个参数串。copy_strings()最后一个参数（0）指明参数字符串在用户空间。
286                 if (sh_bang++ == 0) {
287                         p = copy_strings(envc, envp, page, p, 0);
288                         p = copy_strings(--argc, argv+1, page, p, 0);
289                 }
290                 /*
291                  * Splice in (1) the interpreter's name for argv[0]
292                  *           (2) (optional) argument to interpreter
293                  *           (3) filename of shell script
294                  *
295                  * This is done in reverse order, because of how the
296                  * user environment and arguments are stored.
297                  */
                    /*
                     * 拼接 (1) argv[0]中放解释程序的名称
                     *      (2) (可选的)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，是由于用户环境和参数的存放方式造成的。
                     */
    // 接着我们逆向复制脚本文件名、解释程序的参数和解释程序文件名到参数和环境空间中。
    // 若出错，则置出错码，跳转到 exec_error1。另外，由于本函数参数提供的脚本文件名
    // filename 在用户空间，但这里赋予 copy_strings()的脚本文件名的指针在内核空间，因
    // 此这个复制字符串函数的最后一个参数（字符串来源标志）需要被设置成 1。若字符串在
    // 内核空间，则 copy_strings()的最后一个参数要设置成 2，如下面的第 301、304 行。
298                 p = copy_strings(1, &filename, page, p, 1);
299                 argc++;
300                 if (i_arg) {                        // 复制解释程序的多个参数。
301                         p = copy_strings(1, &i_arg, page, p, 2);
302                         argc++;
303                 }
304                 p = copy_strings(1, &i_name, page, p, 2);
305                 argc++;
306                 if (!p) {
307                         retval = -ENOMEM;
308                         goto exec_error1;
309                 }
310                 /*
311                  * OK, now restart the process with the interpreter's inode.
312                  */
                    /*
                     * OK，现在使用解释程序的 i 节点重启进程。
                     */
```

```
    // 最后我们取得解释程序的 i 节点指针，然后跳转到 204 行去执行解释程序。为了获得解释程
    // 序的 i 节点，我们需要使用 namei()函数，但是该函数所使用的参数（文件名）是从用户数
    // 据空间得到的，即从段寄存器 fs 所指空间中取得。因此在调用 namei()函数之前我们需要
    // 先临时让 fs 指向内核数据空间，以让函数能从内核空间得到解释程序名，并在 namei()
    // 返回后恢复 fs 的默认设置。因此这里我们先临时保存原 fs 段寄存器（原指向用户数据段）
    // 的值，将其设置成指向内核数据段，然后取解释程序的 i 节点。之后再恢复 fs 的原值。并
    // 跳转到 restart_interp（204 行）处重新处理新的执行文件 -- 脚本文件的解释程序。
313                 old_fs = get_fs();
314                 set_fs(get_ds());
315                 if (!(inode=namei(interp))) {        /* get executables inode */
316                         set_fs(old_fs);              /* 取得解释程序的 i 节点 */
317                         retval = -ENOENT;
318                         goto exec_error1;
319                 }
320                 set_fs(old_fs);
321                 goto restart_interp;
322         }

    // 此时缓冲块中的执行文件头结构数据已经复制到了 ex 中。 于是先释放该缓冲块，并开始对
    // ex 中的执行头信息进行判断处理。对于 Linux 0.12 内核来说，它仅支持 ZMAGIC 执行文件格
    // 式，并且执行文件代码都从逻辑地址 0 开始执行，因此不支持含有代码或数据重定位信息的
    // 执行文件。当然，如果执行文件实在太大或者执行文件残缺不全，那么我们也不能运行它。
    // 因此对于下列情况将不执行程序：如果执行文件不是需求页可执行文件（ZMAGIC）、或者代
    // 码和数据重定位部分长度不等于 0、或者(代码段+数据段+堆) 长度超过 50MB、或者执行文件
    // 长度小于 (代码段+数据段+符号表长度+执行头部分) 长度的总和。
323         brelse(bh);
324         if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
325                 ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
326                 inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
327                 retval = -ENOEXEC;
328                 goto exec_error2;
329         }
    // 另外，如果执行文件中代码开始处没有位于 1 个页面（1024 字节）边界处，则也不能执行。
    // 因为需求页（Demand paging）技术要求加载执行文件内容时以页面为单位，因此要求执行
    // 文件映像中代码和数据都从页面边界处开始。
330         if (N_TXTOFF(ex) != BLOCK_SIZE) {
331                 printk("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename);
332                 retval = -ENOEXEC;
333                 goto exec_error2;
334         }
    // 如果 sh_bang 标志没有设置，则复制指定个数的命令行参数和环境字符串到参数和环境空间
    // 中。若 sh_bang 标志已经设置，则表明是将运行脚本解释程序，此时环境变量页面已经复制，
    // 无须再复制。同样，若 sh_bang 没有置位而需要复制的话，那么此时指针 p 随着复制信息增
    // 加而逐渐向小地址方向移动，因此这两个复制串函数执行完后，环境参数串信息块位于程序
    // 参数串信息块的上方，并且 p 指向程序的第 1 个参数串。事实上，p 是 128KB 参数和环境空
    // 间中的偏移值。因此如果 p=0，则表示环境变量与参数空间页面已经被占满，容纳不下了。
335         if (!sh_bang) {
336                 p = copy_strings(envc,envp,page,p,0);
337                 p = copy_strings(argc,argv,page,p,0);
338                 if (!p) {
339                         retval = -ENOMEM;
340                         goto exec_error2;
341                 }
```

```
342         }
343 /* OK, This is the point of no return */
344 /* note that current->library stays unchanged by an exec */
    /* OK，下面开始就没有返回的地方了 */
    // 前面我们针对函数参数提供的信息对需要运行执行文件的命令行参数和环境空间进行了设置，
    // 但还没有为执行文件做过什么实质性的工作，即还没有做过为执行文件初始化进程任务结构
    // 信息、建立页表等工作。现在我们就来做这些工作。由于执行文件直接使用当前进程的“躯
    // 壳”，即当前进程将被改造成执行文件的进程，因此我们需要首先释放当前进程占用的某些
    // 系统资源，包括关闭指定的已打开文件、占用的页表和内存页面等。然后根据执行文件头结
    // 构信息修改当前进程使用的局部描述符表 LDT 中描述符的内容，重新设置代码段和数据段描
    // 述符的限长，再利用前面处理得到的 e_uid 和 e_gid 等信息来设置进程任务结构中相关的字
    // 段。最后把执行本次系统调用程序的返回地址 eip[]指向执行文件中代码的起始位置处。这
    // 样当本系统调用退出返回后就会去运行新执行文件的代码了。注意，虽然此时新执行文件代
    // 码和数据还没有从文件中加载到内存中，但其参数和环境块已经在 copy_strings() 中使用
    // get_free_page()分配了物理内存页来保存数据，并在 change_ldt()函数中使用 put_page()
    // 到了进程逻辑空间的末端处。另外，在 create_tables()中也会由于在用户栈上存放参数
    // 和环境指针表而引起缺页异常，从而内存管理程序也会就此为用户栈空间映射物理内存页。
    //
    // 这里我们首先放回进程原执行程序的 i 节点，并且让进程 executable 字段指向新执行文件
    // 的 i 节点。然后复位原进程的所有信号处理句柄，但对于 SIG_IGN 句柄无须复位。再根据设
    // 定的执行时关闭文件句柄（close_on_exec）位图标志，关闭指定的打开文件并复位该标志。
345         if (current->executable)
346                 iput(current->executable);
347         current->executable = inode;
348         current->signal = 0;
349         for (i=0 ; i<32 ; i++) {
350                 current->sigaction[i].sa_mask = 0;
351                 current->sigaction[i].sa_flags = 0;
352                 if (current->sigaction[i].sa_handler != SIG_IGN)
353                         current->sigaction[i].sa_handler = NULL;
354         }
355         for (i=0 ; i<NR_OPEN ; i++)
356                 if ((current->close_on_exec>>i)&1)
357                         sys_close(i);
358         current->close_on_exec = 0;
    // 然后根据当前进程指定的基地址和限长，释放原来程序的代码段和数据段所对应的内存页表
    // 指定的物理内存页面及页表本身。此时新执行文件并没有占用主内存区任何页面，因此在处
    // 理器真正运行新执行文件代码时就会引起缺页异常中断，此时内存管理程序即会执行缺页处
    // 理而为新执行文件申请内存页面和设置相关页表项，并且把相关执行文件页面读入内存中。
    // 如果“上次任务使用了协处理器”指向的是当前进程，则将其置空，并复位使用了协处理器
    // 的标志。
359         free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
360         free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
361         if (last_task_used_math == current)
362                 last_task_used_math = NULL;
363         current->used_math = 0;
    // 然后我们根据新执行文件头结构中的代码长度字段 a_text 的值修改局部表中描述符基址和
    // 段限长，并将 128KB 的参数和环境空间页面放置在数据段末端。执行下面语句之后，p 此时
    // 更改成以数据段起始处为原点的偏移值，但仍指向参数和环境空间数据开始处，即已转换成
    // 为栈指针值。然后调用内部函数 create_tables() 在栈空间中创建环境和参数变量指针表，
    // 供程序的 main()作为参数使用，并返回该栈指针。
364         p += change_ldt(ex.a_text,page);
365         p -= LIBRARY_SIZE + MAX_ARG_PAGES*PAGE_SIZE;
```

```
366         p = (unsigned long) create_tables((char *)p,argc,envc);

    // 接着再修改进程各字段值为新执行文件的信息。即令进程任务结构代码尾字段 end_code 等
    // 于执行文件的代码段长度 a_text；数据尾字段 end_data 等于执行文件的代码段长度加数
    // 据段长度（a_data + a_text）；并令进程堆结尾字段 brk = a_text + a_data + a_bss。
    // brk 用于指明进程当前数据段（包括未初始化数据部分）末端位置，供内核为进程分配内存
    // 时指定分配开始位置。然后设置进程栈开始字段为栈指针所在页面，并重新设置进程的有效
    // 用户 id 和有效组 id。
367         current->brk = ex.a_bss +
368                 (current->end_data = ex.a_data +
369                 (current->end_code = ex.a_text));
370         current->start_stack = p & 0xfffff000;
371         current->suid = current->euid = e_uid;
372         current->sgid = current->egid = e_gid;
    // 最后将原调用系统中断的程序在堆栈上的代码指针替换为指向新执行程序的入口点，并将栈
    // 指针替换为新执行文件的栈指针。此后返回指令将弹出这些栈数据并使得 CPU 去执行新执行
    // 文件，因此不会返回到原调用系统中断的程序中去了。
373         eip[0] = ex.a_entry;    /* eip, magic happens :-) */  /* eip，魔法起作用了*/
374         eip[3] = p;             /* stack pointer */           /* esp，堆栈指针 */
375         return 0;
376 exec_error2:
377         iput(inode);                        // 放回 i 节点。
378 exec_error1:
379         for (i=0 ; i<MAX_ARG_PAGES ; i++)
380                 free_page(page[i]);         // 释放存放参数和环境串的内存页面。
381         return(retval);                     // 返回出错码。
382 }
383
```

12.15 程序 12-15 linux/fs/stat.c

```
 1 /*
 2  *  linux/fs/stat.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
 8 #include <sys/stat.h>     // 文件状态头文件。含有文件状态结构 stat{}和常量。
 9
10 #include <linux/fs.h>     // 文件系统头文件。定义文件表结构（file、m_inode）等。
11 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
12 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
13 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14
   //// 复制文件状态信息。
   // 参数 inode 是文件 i 节点，statbuf 是用户数据空间中 stat 文件状态结构指针，用于存放取
   // 得的状态信息。
15 static void cp_stat(struct m_inode * inode, struct stat * statbuf)
16 {
17         struct stat tmp;
18         int i;
19
   // 首先验证(或分配)存放数据的内存空间。然后临时复制相应节点上的信息。
20         verify_area(statbuf,sizeof (struct stat));
21         tmp.st_dev = inode->i_dev;        // 文件所在的设备号。
22         tmp.st_ino = inode->i_num;        // 文件 i 节点号。
23         tmp.st_mode = inode->i_mode;      // 文件属性。
24         tmp.st_nlink = inode->i_nlinks;   // 文件连接数。
25         tmp.st_uid = inode->i_uid;        // 文件的用户 ID。
26         tmp.st_gid = inode->i_gid;        // 文件的组 ID。
27         tmp.st_rdev = inode->i_zone[0];   // 设备号（若是特殊字符文件或块设备文件）。
28         tmp.st_size = inode->i_size;      // 文件字节长度（如果文件是常规文件）。
29         tmp.st_atime = inode->i_atime;    // 最后访问时间。
30         tmp.st_mtime = inode->i_mtime;    // 最后修改时间。
31         tmp.st_ctime = inode->i_ctime;    // 最后 i 节点修改时间。
   // 最后将这些状态信息复制到用户缓冲区中。
32         for (i=0 ; i<sizeof (tmp) ; i++)
33                 put_fs_byte(((char *) &tmp)[i],i + (char *) statbuf);
34 }
35
   //// 文件状态系统调用。
   // 根据给定的文件名获取相关文件状态信息。
   // 参数 filename 是指定的文件名，statbuf 是存放状态信息的缓冲区指针。
   // 返回：成功返回 0，若出错则返回出错码。
36 int sys_stat(char * filename, struct stat * statbuf)
37 {
38         struct m_inode * inode;
39
   // 首先根据文件名找出对应的 i 节点。然后将 i 节点上的文件状态信息复制到用户缓冲区中，
```

```
    // 并放回该 i 节点。
40          if (!(inode=namei(filename)))
41                  return -ENOENT;
42          cp_stat(inode,statbuf);
43          iput(inode);
44          return 0;
45  }
46
    //// 文件状态系统调用。
    // 根据给定的文件名获取相关文件状态信息。文件路径名中有符号链接文件名，则取符号文件
    // 的状态。
    // 参数 filename 是指定的文件名，statbuf 是存放状态信息的缓冲区指针。
    // 返回：成功返回 0，若出错则返回出错码。
47  int sys_lstat(char * filename, struct stat * statbuf)
48  {
49          struct m_inode * inode;
50
    // 首先根据文件名找出对应的 i 节点。然后将 i 节点上的文件状态信息复制到用户缓冲区中，
    // 并放回该 i 节点。
51          if (!(inode = lnamei(filename)))      // 取指定路径名 i 节点，不跟随符号链接。
52                  return -ENOENT;
53          cp_stat(inode,statbuf);
54          iput(inode);
55          return 0;
56  }
57
    //// 文件状态系统调用。
    // 根据给定的文件句柄获取相关文件状态信息。
    // 参数 fd 是指定文件的句柄(描述符)，statbuf 是存放状态信息的缓冲区指针。
    // 返回：成功返回 0，若出错则返回出错码。
58  int sys_fstat(unsigned int fd, struct stat * statbuf)
59  {
60          struct file * f;
61          struct m_inode * inode;
62
    // 首先取文件句柄对应的文件结构，然后从中得到文件的 i 节点。然后将 i 节点上的文件状
    // 态信息复制到用户缓冲区中。如果文件句柄值大于一个程序最多打开文件数 NR_OPEN，或
    // 者该句柄的文件结构指针为空，或者对应文件结构的 i 节点字段为空，则出错，返回出错
    // 码并退出。
63          if (fd >= NR_OPEN || !(f=current->filp[fd]) || !(inode=f->f_inode))
64                  return -EBADF;
65          cp_stat(inode,statbuf);
66          return 0;
67  }
68
    //// 读符号链接文件系统调用。
    // 该调用读取符号链接文件的内容（即该符号链接所指向文件的路径名字符串），并放到指定
    // 长度的用户缓冲区中。若缓冲区太小，就会截断符号链接的内容。
    // 参数：path -- 符号链接文件路径名；buf -- 用户缓冲区；bufsiz -- 缓冲区长度。
    // 返回：成功则返回放入缓冲区中的字符数；若失败则返回出错码。
69  int sys_readlink(const char * path, char * buf, int bufsiz)
70  {
71          struct m_inode * inode;
```

```
72          struct buffer_head * bh;
73          int i;
74          char c;
75
    // 首先检查和验证函数参数的有效性，并对其进行调整。用户缓冲区字节长度 bufsi 必须在
    // 1--1023 之间。然后取得符号链接文件名的 i 节点，并读取该文件的第 1 块数据内容。之
    // 后放回 i 节点。
76          if (bufsiz <= 0)
77                  return -EBADF;
78          if (bufsiz > 1023)
79                  bufsiz = 1023;
80          verify_area(buf,bufsiz);
81          if (!(inode = lnamei(path)))
82                  return -ENOENT;
83          if (inode->i_zone[0])
84                  bh = bread(inode->i_dev, inode->i_zone[0]);
85          else
86                  bh = NULL;
87          iput(inode);
    // 如果读取文件数据内容成功，则从内容中复制最多 bufsiz 个字符到用户缓冲区中。不复制
    // NULL 字符。最后释放缓冲块，并返回复制的字节数。
88          if (!bh)
89                  return 0;
90          i = 0;
91          while (i<bufsiz && (c = bh->b_data[i])) {
92                  i++;
93                  put_fs_byte(c,buf++);
94          }
95          brelse(bh);
96          return i;
97  }
98
```

12.16 程序 12-16 linux/fs/fcntl.c

```
 1 /*
 2  *  linux/fs/fcntl.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <string.h>       // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
 8 #include <errno.h>        // 错误号头文件。包含系统中各种出错号。
 9 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
10 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
11 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
12
13 #include <fcntl.h>        // 文件控制头文件。定义文件及其描述符的操作控制常数符号。
14 #include <sys/stat.h>     // 文件状态头文件。含有文件状态结构 stat{}和常量。
15
16 extern int sys_close(int fd);     // 关闭文件系统调用。(fs/open.c, 192)
17
   //// 复制文件句柄（文件描述符）。
   // 参数 fd 是欲复制的文件句柄，arg 指定新文件句柄的最小数值。
   // 返回新文件句柄或出错码。
18 static int dupfd(unsigned int fd, unsigned int arg)
19 {
   // 首先检查函数参数的有效性。如果文件句柄值大于一个程序最多打开文件数 NR_OPEN，或者
   // 该句柄的文件结构不存在，则返回出错码并退出。如果指定的新句柄值 arg 大于最多打开文
   // 件数，也返回出错码并退出。注意，实际上文件句柄就是进程文件结构指针数组项索引号。
20         if (fd >= NR_OPEN || !current->filp[fd])
21                 return -EBADF;
22         if (arg >= NR_OPEN)
23                 return -EINVAL;
   // 然后在当前进程的文件结构指针数组中寻找索引号等于或大于 arg，但还没有使用的项。若
   // 找到的新句柄值 arg 大于最多打开文件数（即没有空闲项），则返回出错码并退出。
24         while (arg < NR_OPEN)
25                 if (current->filp[arg])
26                         arg++;
27                 else
28                         break;
29         if (arg >= NR_OPEN)
30                 return -EMFILE;
   // 否则针对找到的空闲项（句柄），在执行时关闭标志位图 close_on_exec 中复位该句柄位。
   // 即在运行 exec()类函数时，不会关闭用 dup()创建的句柄。并令该文件结构指针等于原句
   // 柄 fd 的指针，并且将文件引用计数增 1。最后返回新的文件句柄 arg。
31         current->close_on_exec &= ~(1<<arg);
32         (current->filp[arg] = current->filp[fd])->f_count++;
33         return arg;
34 }
35
   //// 复制文件句柄系统调用。
   // 复制指定文件句柄 oldfd，新文件句柄值等于 newfd。如果 newfd 已打开，则首先关闭之。
   // 参数：oldfd -- 原文件句柄；newfd - 新文件句柄。
```

```
    // 返回新文件句柄值。
36  int sys_dup2(unsigned int oldfd, unsigned int newfd)
37  {
38          sys_close(newfd);               // 若句柄 newfd 已经打开，则首先关闭之。
39          return dupfd(oldfd,newfd);      // 复制并返回新句柄。
40  }
41
    //// 复制文件句柄系统调用。
    // 复制指定文件句柄 oldfd，新句柄的值是当前最小的未用句柄值。
    // 参数：fildes -- 被复制的文件句柄。
    // 返回新文件句柄值。
42  int sys_dup(unsigned int fildes)
43  {
44          return dupfd(fildes,0);
45  }
46
    //// 文件控制系统调用函数。
    // 参数 fd 是文件句柄；cmd 是控制命令（参见 include/fcntl.h，23-30 行）；arg 则针对不
    // 同的命令有不同的含义。对于复制句柄命令 F_DUPFD，arg 是新文件句柄可取的最小值；对
    // 于设置文件操作和访问标志命令 F_SETFL，arg 是新的文件操作和访问模式。对于文件上锁
    // 命令 F_GETLK、F_SETLK 和 F_SETLKW，arg 是指向 flock 结构的指针。但本内核中没有实现
    // 文件上锁功能。
    // 返回：若出错，则所有操作都返回-1。若成功，那么 F_DUPFD 返回新文件句柄； F_GETFD
    // 返回文件句柄的当前执行时关闭标志 close_on_exec；F_GETFL 返回文件操作和访问标志。
47  int sys_fcntl(unsigned int fd, unsigned int cmd, unsigned long arg)
48  {
49          struct file * filp;
50
    // 首先检查给出的文件句柄的有效性。然后根据不同命令 cmd 进行分别处理。 如果文件句柄
    // 值大于一个进程最多打开文件数 NR_OPEN，或者该句柄的文件结构指针为空，则返回出错码
    // 并退出。
51          if (fd >= NR_OPEN || !(filp = current->filp[fd]))
52                  return -EBADF;
53          switch (cmd) {
54                  case F_DUPFD:       // 复制文件句柄。
55                          return dupfd(fd,arg);
56                  case F_GETFD:       // 取文件句柄的执行时关闭标志。
57                          return (current->close_on_exec>>fd)&1;
58                  case F_SETFD:  // 设置执行时关闭标志。arg 位 0 置位是设置，否则关闭。
59                          if (arg&1)
60                                  current->close_on_exec |= (1<<fd);
61                          else
62                                  current->close_on_exec &= ~(1<<fd);
63                          return 0;
64                  case F_GETFL:       // 取文件状态标志和访问模式。
65                          return filp->f_flags;
66                  case F_SETFL: // 设置文件状态和访问模式（根据 arg 设置添加、非阻塞标志）。
67                          filp->f_flags &= ~(O_APPEND | O_NONBLOCK);
68                          filp->f_flags |= arg & (O_APPEND | O_NONBLOCK);
69                          return 0;
70                  case F_GETLK:   case F_SETLK:   case F_SETLKW:     // 未实现。
71                          return -1;
72                  default:
```

```
73                          return -1;
74          }
75 }
76
```

12.17 程序 12-17 linux/fs/ioctl.c

```
 1 /*
 2  *  linux/fs/ioctl.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
 8 #include <errno.h>         // 错误号头文件。包含系统中各种出错号。
 9 #include <sys/stat.h>      // 文件状态头文件。含有文件状态结构 stat{}和常量。
10
11 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 数据等。
12
13 extern int tty_ioctl(int dev, int cmd, int arg);    // chr_drv/tty_ioctl.c，第 133 行。
14 extern int pipe_ioctl(struct m_inode *pino, int cmd, int arg);  // fs/pipe.c，第 118 行。
15
   // 定义输入输出控制(ioctl)函数指针类型。
16 typedef int (*ioctl_ptr)(int dev,int cmd,int arg);
17
   // 取系统中设备种数的宏。
18 #define NRDEVS ((sizeof (ioctl_table))/(sizeof (ioctl_ptr)))
19
   // ioctl 操作函数指针表。
20 static ioctl_ptr ioctl_table[]={
21         NULL,           /* nodev */
22         NULL,           /* /dev/mem */
23         NULL,           /* /dev/fd */
24         NULL,           /* /dev/hd */
25         tty_ioctl,      /* /dev/ttyx */
26         tty_ioctl,      /* /dev/tty */
27         NULL,           /* /dev/lp */
28         NULL};          /* named pipes */
29
30
   //// 系统调用函数 - 输入输出控制函数。
   // 该函数首先判断参数给出的文件描述符是否有效。然后根据对应 i 节点中文件属性判断文件
   // 类型，并根据具体文件类型调用相关的处理函数。
   // 参数：fd - 文件描述符；cmd - 命令码；arg - 参数。
   // 返回：成功则返回 0，否则返回出错码。
31 int sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
32 {
33         struct file * filp;
34         int dev,mode;
35
   // 首先判断给出的文件描述符的有效性。如果文件描述符超出可打开的文件数，或者对应描述
   // 符的文件结构指针为空，则返回出错码退出。
36         if (fd >= NR_OPEN || !(filp = current->filp[fd]))
37                 return -EBADF;
   // 如果文件结构对应的是管道 i 节点，则根据进程是否有权操作该管道确定是否执行管道 IO
   // 控制操作。若有权执行则调用 pipe_ioctl()，否则返回无效文件错误码。
```

```
38          if (filp->f_inode->i_pipe)
39                  return (filp->f_mode&1)?pipe_ioctl(filp->f_inode,cmd,arg):-EBADF;
    // 对于其他类型文件，取对应文件的属性，并据此判断文件的类型。如果该文件既不是字符设
    // 备文件，也不是块设备文件，则返回出错码退出。若是字符或块设备文件，则从文件的 i 节
    // 点中取设备号。如果设备号大于系统现有的设备数，则返回出错号。
40          mode=filp->f_inode->i_mode;
41          if (!S_ISCHR(mode) && !S_ISBLK(mode))
42                  return -EINVAL;
43          dev = filp->f_inode->i_zone[0];
44          if (MAJOR(dev) >= NRDEVS)
45                  return -ENODEV;
    // 然后根据 IO 控制表 ioctl_table 查得对应设备的 ioctl 函数指针，并调用该函数。如果该设
    // 备在 ioctl 函数指针表中没有对应函数，则返回出错码。
46          if (!ioctl_table[MAJOR(dev)])
47                  return -ENOTTY;
48          return ioctl_table[MAJOR(dev)](dev,cmd,arg);
49  }
50
```

12.18 程序 12-18 linux/fs/select.c

```
 1 /*
 2  * This file contains the procedures for the handling of select
 3  *
 4  * Created for Linux based loosely upon Mathius Lattner's minix
 5  * patches by Peter MacDonald. Heavily edited by Linus.
 6  */
   /*
    * 本文件含有处理 select()系统调用的过程。
    *
    * 这是 Peter MacDonald 基于 Mathius Lattner 提供给 MINIX 系统的补丁
    * 程序修改而成。
    */
 7
 8 #include <linux/fs.h>
 9 #include <linux/kernel.h>
10 #include <linux/tty.h>
11 #include <linux/sched.h>
12
13 #include <asm/segment.h>
14 #include <asm/system.h>
15
16 #include <sys/stat.h>
17 #include <sys/types.h>
18 #include <string.h>
19 #include <const.h>
20 #include <errno.h>
21 #include <sys/time.h>
22 #include <signal.h>
23
24 /*
25  * Ok, Peter made a complicated, but straightforward multiple_wait() function.
26  * I have rewritten this, taking some shortcuts: This code may not be easy to
27  * follow, but it should be free of race-conditions, and it's practical. If you
28  * understand what I'm doing here, then you understand how the linux sleep/wakeup
29  * mechanism works.
30  *
31  * Two very simple procedures, add_wait() and free_wait() make all the work. We
32  * have to have interrupts disabled throughout the select, but that's not really
33  * such a loss: sleeping automatically frees interrupts when we aren't in this
34  * task.
35  */
   /*
    * OK，Peter 编制了复杂但很直观的多个_wait()函数。我对这些函数进行了改写，以使之
    * 更简洁：这些代码可能不容易看懂，但是其中应该不会存在竞争条件问题，并且很实际。
    * 如果你能理解这里编制的代码，那么就说明你已经理解 Linux 中睡眠/唤醒的工作机制。
    *
    * 两个很简单的过程，add_wait()和 free_wait()执行了主要操作。在整个 select 处理
    * 过程中我们不得不禁止中断。但是这样做并不会带来太多的损失：因为当我们不在执行
    * 本任务时睡眠状态会自动地释放中断（即其他任务会使用自己 EFLAGS 中的中断标志）。
```

```
 */
36
37 typedef struct {
38         struct task_struct * old_task;
39         struct task_struct ** wait_address;
40 } wait_entry;
41
42 typedef struct {
43         int nr;
44         wait_entry entry[NR_OPEN*3];
45 } select_table;
46
   // 把未准备好描述符的等待队列指针加入等待表 wait_table 中。参数*wait_address 是与描述
   // 符相关的等待队列头指针。例如 tty 读缓冲队列 secondary 的等待队列头指针是 proc_list。
   // 参数 p 是 do_select()中定义的等待表结构指针。
47 static void add_wait(struct task_struct ** wait_address, select_table * p)
48 {
49         int i;
50
   // 首先判断描述符是否有对应的等待队列，若无则返回。然后在等待表中搜索参数指定的等待
   // 队列指针是否已经在等待表中设置过，若设置过也立刻返回。这个判断主要是针对管道文件
   // 描述符。例如若一个管道在等待可以进行读操作，那么其必定可以立刻进行写操作。
51         if (!wait_address)
52                 return;
53         for (i = 0 ; i < p->nr ; i++)
54                 if (p->entry[i].wait_address == wait_address)
55                         return;
   // 然后我们把描述符对应等待队列的头指针保存在等待表 wait_table 中，同时让等待表项的
   // old_task 字段指向等待队列头指针指向的任务（若无则为 NULL），在让等待队列头指针指
   // 向当前任务。最后把等待表有效项计数值 nr 增 1（其在第 179 行被初始化为 0）。
56         p->entry[p->nr].wait_address = wait_address;
57         p->entry[p->nr].old_task = * wait_address;
58         *wait_address = current;
59         p->nr++;
60 }
61
   // 清空等待表。参数是等待表结构指针。本函数在 do_select()函数中睡眠后被唤醒返回时被调用
   // （第 204、207 行），用于唤醒等待表中处于各个等待队列上的其他任务。它与 kernel/sched.c
   // 中 sleep_on()函数的后半部分代码几乎完全相同，请参考对 sleep_on()函数的说明。
62 static void free_wait(select_table * p)
63 {
64         int i;
65         struct task_struct ** tpp;
66
   // 如果等待表中各项（供 nr 个有效项）记录的等待队列头指针表明还有其他后来添加进的等待
   // 任务（例如其他进程调用 sleep_on() 函数而睡眠在该等待队列上），则此时等待队列头指针
   // 指向的不是当前进程，那么我们就需要先唤醒这些任务。操作方法是将等待队列头所指任务先
   // 置为就绪状态（state = 0），并把自己设置为不可中断等待状态，即自己要等待这些后续进队
   // 列的任务被唤醒而执行时来唤醒本任务。然后重新执行调度程序。
67         for (i = 0; i < p->nr ; i++) {
68                 tpp = p->entry[i].wait_address;
69                 while (*tpp && *tpp != current) {
70                         (*tpp)->state = 0;
```

```
71                      current->state = TASK_UNINTERRUPTIBLE;
72                      schedule();
73              }
    // 执行到这里，说明等待表当前处理项中的等待队列头指针字段 wait_address 指向当前任务，
    // 若它为空，则表明调度有问题，于是显示警告信息。然后我们让等待队列头指针指向在我们
    // 前面进入队列的任务（第 76 行）。若此时该头指针确实指向一个任务而不是 NULL，则说明
    // 队列中还有任务（*tpp 不为空），于是将该任务设置成就绪状态，唤醒之。最后把等待表的
    // 有效表项计数字段 nr 清零。
74              if (!*tpp)
75                      printk("free_wait: NULL");
76              if (*tpp = p->entry[i].old_task)
77                      (**tpp).state = 0;
78      }
79      p->nr = 0;
80 }
81
    // 根据文件 i 节点判断文件是否是字符终端设备文件。若是则返回其 tty 结构指针，否则返回 NULL。
82 static struct tty_struct * get_tty(struct m_inode * inode)
83 {
84      int major, minor;
85
    // 如果不是字符设备文件则返回 NULL。如果主设备号不是 5（控制终端）或 4，则返回 NULL。
86      if (!S_ISCHR(inode->i_mode))
87              return NULL;
88      if ((major = MAJOR(inode->i_zone[0])) != 5 && major != 4)
89              return NULL;
    // 如果主设备号是 5，那么其终端设备号等于进程的 tty 字段值，否则就等于字符设备文件次设备号。
    // 如果终端设备号小于 0，表示进程没有控制终端或没有使用终端，于是返回 NULL。否则返回对应的
    // tty 结构指针。
90      if (major == 5)
91              minor = current->tty;
92      else
93              minor = MINOR(inode->i_zone[0]);
94      if (minor < 0)
95              return NULL;
96      return TTY_TABLE(minor);
97 }
98
99 /*
100  * The check_XX functions check out a file. We know it's either
101  * a pipe, a character device or a fifo (fifo's not implemented)
102  */
    /*
     * check_XX 函数用于检查一个文件。我们知道该文件要么是管道文件、
     * 要么是字符设备文件，或者要么是一个 FIFO（FIFO 还未实现）。
     */
    // 检查读文件操作是否准备好，即终端读缓冲队列 secondary 是否有字符可读，或者管道文件是否
    // 不空。参数 wait 是等待表指针；inode 是文件 i 节点指针。若描述符可进行读操作则返回 1，否
    // 则返回 0。
103 static int check_in(select_table * wait, struct m_inode * inode)
104 {
105      struct tty_struct * tty;
106
```

```
    // 首先根据文件 i 节点调用 get_tty()检测文件是否是一个 tty 终端（字符）设备文件，如果是则
    // 检查该终端读缓冲队列 secondary 中是否有字符可供读取，若有则返回 1，若此时 secondary 为
    // 空则把当前任务添加到 secondary 的等待队列 proc_list 上并返回 0。如果是管道文件，则判断
    // 目前管道中是否有字符可读，若有则返回 1，若没有（管道空）则把当前任务添加到管道 i 节点
    // 的等待队列上并返回 0。注意，PIPE_EMPTY()宏使用管道当前头尾指针位置来判断管道是否空。
    // 管道 i 节点的 i_zone[0]和 i_zone[1]字段分别存放着管道当前的头尾指针。
107         if (tty = get_tty(inode))
108                 if (!EMPTY(tty->secondary))
109                         return 1;
110                 else
111                         add_wait(&tty->secondary->proc_list, wait);
112         else if (inode->i_pipe)
113                 if (!PIPE_EMPTY(*inode))
114                         return 1;
115                 else
116                         add_wait(&inode->i_wait, wait);
117         return 0;
118 }
119
    // 检查文件写操作是否准备好，即终端写缓冲队列 write_q 中是否还有空闲位置可写，或者此时管
    // 道文件是否不满。参数 wait 是等待表指针；inode 是文件 i 节点指针。若描述符可进行写操作则
    // 返回 1，否则返回 0。
120 static int check_out(select_table * wait, struct m_inode * inode)
121 {
122         struct tty_struct * tty;
123
    // 首先根据文件 i 节点调用 get_tty()检测文件是否是一个 tty 终端（字符）设备文件，如果是则
    // 检查该终端写缓冲队列 write_q 中是否有空间可写入，若有则返回 1，若没有空空间则把当前任
    // 务添加到 write_q 的等待队列 proc_list 上并返回 0。如果是管道文件则判断目前管道中是否有
    // 空闲空间可写入字符，若有则返回 1，若没有（管道满）则把当前任务添加到管道 i 节点的等待
    // 队列上并返回 0。
124         if (tty = get_tty(inode))
125                 if (!FULL(tty->write_q))
126                         return 1;
127                 else
128                         add_wait(&tty->write_q->proc_list, wait);
129         else if (inode->i_pipe)
130                 if (!PIPE_FULL(*inode))
131                         return 1;
132                 else
133                         add_wait(&inode->i_wait, wait);
134         return 0;
135 }
136
    // 检查文件是否处于异常状态。对于终端设备文件，目前内核总是返回 0。对于管道文件，如果
    // 此时两个管道描述符中有一个或都已被关闭，则返回 1，否则就把当前任务添加到管道 i 节点
    // 的等待队列上并返回 0。返回 0。参数 wait 是等待表指针；inode 是文件 i 节点指针。若出现
    // 异常条件则返回 1，否则返回 0。
137 static int check_ex(select_table * wait, struct m_inode * inode)
138 {
139         struct tty_struct * tty;
140
141         if (tty = get_tty(inode))
```

```
142                 if (!FULL(tty->write_q))
143                         return 0;
144                 else
145                         return 0;
146         else if (inode->i_pipe)
147                 if (inode->i_count < 2)
148                         return 1;
149                 else
150                         add_wait(&inode->i_wait,wait);
151         return 0;
152 }
153
    // do_select() 是内核执行 select()系统调用的实际处理函数。该函数首先检查描述符集中各个
    // 描述符的有效性，然后分别调用相关描述符集描述符检查函数 check_XX()对每个描述符进行检
    // 查，同时统计描述符集中当前已经准备好的描述符个数。若有任何一个描述符已经准备好，本
    // 函数就会立刻返回，否则进程就会在本函数中进入睡眠状态，并在过了超时时间或者由于某个
    // 描述符所在等待队列上的进程被唤醒而使本进程继续运行。
154 int do_select(fd_set in, fd_set out, fd_set ex,
155         fd_set *inp, fd_set *outp, fd_set *exp)
156 {
157         int count;                                  // 已准备好的描述符个数计数值。
158         select_table wait_table;                    // 等待表结构。
159         int i;
160         fd_set mask;
161
    // 首先把 3 个描述符集进行或操作，在 mask 中得到描述符集中有效描述符比特位屏蔽码。 然后
    // 循环判断当前进程各个描述符是否有效并且包含在描述符集内。在循环中，每判断完一个描述
    // 符就会把 mask 右移 1 位，因此根据 mask 的最低有效比特位我们就可以判断相应描述符是否在
    // 用户给定的描述符集中。有效的描述符应该是一个管道文件描述符，或者是一个字符设备文件
    // 描述符，或者是一个 FIFO 描述符，其余类型的都作为无效描述符而返回 EBADF 错误。
162         mask = in | out | ex;
163         for (i = 0 ; i < NR_OPEN ; i++,mask >>= 1) {
164                 if (!(mask & 1))                    // 若不在描述符集中则继续判断下一个。
165                         continue;
166                 if (!current->filp[i])              // 若该文件未打开，则返回描述符错。
167                         return -EBADF;
168                 if (!current->filp[i]->f_inode)     // 若文件 i 节点指针空，则返回错误号。
169                         return -EBADF;
170                 if (current->filp[i]->f_inode->i_pipe) // 若是管道文件描述符，则有效。
171                         continue;
172                 if (S_ISCHR(current->filp[i]->f_inode->i_mode)) // 字符设备文件有效。
173                         continue;
174                 if (S_ISFIFO(current->filp[i]->f_inode->i_mode)) // FIFO 也有效。
175                         continue;
176                 return -EBADF;                      // 其余都作为无效描述符而返回。
177         }
    // 下面开始循环检查 3 个描述符集中的各个描述符是否准备好（可以操作）。此时 mask 用作当前
    // 正在处理描述符的屏蔽码。循环中的 3 个函数 check_in()、check_out()和 check_ex()分别用
    // 来判断描述符是否已经准备好。若一个描述符已经准备好，则在相关描述符集中设置对应比特
    // 位，并且把已准备好描述符个数计数值 count 增 1。第 183 行 for 循环语句中的 mask += mask
    // 等效于 mask<< 1。
178 repeat:
179         wait_table.nr = 0;
```

```
180         *inp = *outp = *exp = 0;
181         count = 0;
182         mask = 1;
183         for (i = 0 ; i < NR_OPEN ; i++, mask += mask) {
    // 如果此时判断的描述符在读操作描述符集中，并且该描述符已经准备好可以进行读操作，则把
    // 该描述符在描述符集 in 中对应比特位置为 1，同时把已准备好描述符个数计数值 count 增 1。
184                 if (mask & in)
185                         if (check_in(&wait_table,current->filp[i]->f_inode)) {
186                                 *inp |= mask;           // 描述符集中设置对应比特位。
187                                 count++;                // 已准备好描述符个数计数。
188                         }
    // 如果此时判断的描述符在写操作描述符集中，并且该描述符已经准备好可以进行写操作，则把
    // 该描述符在描述符集 out 中对应比特位置为 1，同时把已准备好描述符个数计数值 count 增 1。
189                 if (mask & out)
190                         if (check_out(&wait_table,current->filp[i]->f_inode)) {
191                                 *outp |= mask;
192                                 count++;
193                         }
    // 如果此时判断的描述符在异常描述符集中，并且该描述符已经有异常出现，则把该描述符在描
    // 述符集 ex 中对应比特位置为 1，同时把已准备好描述符个数计数值 count 增 1。
194                 if (mask & ex)
195                         if (check_ex(&wait_table,current->filp[i]->f_inode)) {
196                                 *exp |= mask;
197                                 count++;
198                         }
199         }
    // 在对进程所有描述符判断处理过后，若没有发现有已准备好的描述符（count==0），并且此时
    // 进程没有收到任何非阻塞信号，并且此时有等待着的描述符或者等待时间还没有超时，那么我
    // 们就把当前进程状态设置成可中断睡眠状态，然后执行调度函数去执行其他任务。当内核又一
    // 次调度执行本任务时就调用 free_wait()唤醒相关等待队列上本任务前后的任务，然后跳转到
    // repeat 标号处（178 行）再次重新检测是否有我们关心的（描述符集中的）描述符已准备好。
200         if (!(current->signal & ~current->blocked) &&
201             (wait_table.nr || current->timeout) && !count) {
202                 current->state = TASK_INTERRUPTIBLE;
203                 schedule();
204                 free_wait(&wait_table);         // 本任务被唤醒返回后从这里开始执行。
205                 goto repeat;
206         }
    // 如果此时 count 不等于 0，或者接收到了信号，或者等待时间到并且没有需要等待的描述符，
    // 那么我们就调用 free_wait()唤醒等待队列上的任务，然后返回已准备好的描述符个数值。
207         free_wait(&wait_table);
208         return count;
209 }
210
211 /*
212  * Note that we cannot return -ERESTARTSYS, as we change our input
213  * parameters. Sad, but there you are. We could do some tweaking in
214  * the library function ...
215  */
    /*
     * 注意我们不能返回-ERESTARTSYS，因为我们会在 select 运行过程中改变
     * 输入参数值（*timeout）。很不幸，但你也只能接受这个事实。不过我们
     * 可以在库函数中做些处理...
```

```
 */
// select 系统调用函数。该函数中的代码主要负责进行 select 功能操作前后的参数复制和转换
// 工作。select 主要的工作由 do_select()函数来完成。sys_select()会首先根据参数传递来的
// 缓冲区指针从用户数据空间把 select() 函数调用的参数分解复制到内核空间，然后设置需要
// 等待的超时时间值 timeout，接着调用 do_select() 执行 select 功能，返回后就把处理结果
// 再复制会用户空间中。
// 参数 buffer 指向用户数据区中 select()函数的第 1 个参数处。如果返回值小于 0 表示执行时
// 出现错误；如果返回值等于 0，则表示在规定等待时间内没有描述符准备好操作；如果返回值
// 大于 0，则表示已准备好的描述符数量。
int sys_select( unsigned long *buffer )
{
/* Perform the select(nd, in, out, ex, tv) system call. */
/* 执行 select(nd, in, out, ex, tv)系统调用 */
// 首先定义几个局部变量，用于把指针参数传递来的 select()函数参数分解开来。
        int i;
        fd_set res_in, in = 0, *inp;                    // 读操作描述符集。
        fd_set res_out, out = 0, *outp;                 // 写操作描述符集。
        fd_set res_ex, ex = 0, *exp;                    // 异常条件描述符集。
        fd_set mask;                                    // 处理的描述符数值范围（nd）屏蔽码。
        struct timeval *tvp;                            // 等待时间结构指针。
        unsigned long timeout;

// 然后从用户数据区把参数分别隔离复制到局部指针变量中，并根据描述符集指针是否有效分别
// 取得 3 个描述符集 in（读）、out（写）和 ex（异常）。其中 mask 也是一个描述符集变量，
// 根据 3 个描述符集中最大描述符数值+1（即第 1 个参数 nd 的值），它被设置成用户程序关心的
// 所有描述符的屏蔽码。例如，若 nd = 4，则 mask = 0b00001111（共 32 比特）。
        mask = ~((~0) << get_fs_long(buffer++));
        inp = (fd_set *) get_fs_long(buffer++);
        outp = (fd_set *) get_fs_long(buffer++);
        exp = (fd_set *) get_fs_long(buffer++);
        tvp = (struct timeval *) get_fs_long(buffer);

        if (inp)                                        // 若指针有效，则取读操作描述符集。
                in = mask & get_fs_long(inp);
        if (outp)                                       // 若指针有效，则取写操作描述符集。
                out = mask & get_fs_long(outp);
        if (exp)                                        // 若指针有效，则取异常描述符集。
                ex = mask & get_fs_long(exp);
// 接下来我们尝试从时间结构中取出等待（睡眠）时间值 timeout。首先把 timeout 初始化成最大
// （无限）值，然后从用户数据空间取得该时间结构中设置的时间值，经转换和加上系统当前滴答
// 值 jiffies，最后得到需要等待的时间滴答数值 timeout。我们用此值来设置当前进程应该等待
// 的延时。另外，第 241 行上 tv_usec 字段是微秒值，把它除以 1000000 后可得到对应秒数，再乘
// 以系统每秒滴答数 HZ，即把 tv_usec 转换成滴答值。
        timeout = 0xffffffff;
        if (tvp) {
                timeout = get_fs_long((unsigned long *)&tvp->tv_usec)/(1000000/HZ);
                timeout += get_fs_long((unsigned long *)&tvp->tv_sec) * HZ;
                timeout += jiffies;
        }
        current->timeout = timeout;                     // 设置当前进程应该延时的滴答值。
// select()函数的主要工作在 do_select()中完成。在调用该函数之后的代码用于把处理结果复制
// 到用户数据区中，返回给用户。为了避免出现竞争条件，在调用 do_select()前需要禁止中断，
// 并在该函数返回后再开启中断。
```

```
    // 如果在 do_select()返回之后进程的等待延时字段 timeout 还大于当前系统计时滴答值 jiffies，
    // 说明在超时之前已经有描述符准备好，于是这里我们先记下到超时还剩余的时间值，随后我们会
    // 把这个值返回给用户。如果进程的等待延时字段 timeout 已经小于或等于当前系统 jiffies，表
    // 示 do_select()可能是由于超时而返回，因此把剩余时间值设置为 0。
246         cli();                                  // 禁止响应中断。
247         i = do_select(in, out, ex, &res_in, &res_out, &res_ex);
248         if (current->timeout > jiffies)
249                 timeout = current->timeout - jiffies;
250         else
251                 timeout = 0;
252         sti();                                  // 开启中断响应。
    // 接下来我们把进程的超时字段清零。如果 do_select()返回的已准备好描述符个数小于 0，表示
    // 执行出错，于是返回这个错误号。然后我们把处理过的描述符集内容和延迟时间结构内容写回到
    // 用户数据缓冲空间。在写时间结构内容时还需要先将滴答时间单位表示的剩余延迟时间转换成秒
    // 和微秒值。
253         current->timeout = 0;
254         if (i < 0)
255                 return i;
256         if (inp) {
257                 verify_area(inp, 4);
258                 put_fs_long(res_in,inp);        // 可读描述符集。
259         }
260         if (outp) {
261                 verify_area(outp,4);
262                 put_fs_long(res_out,outp);      // 可写描述符集。
263         }
264         if (exp) {
265                 verify_area(exp,4);
266                 put_fs_long(res_ex,exp);        // 出现异常条件描述符集。
267         }
268         if (tvp) {
269                 verify_area(tvp, sizeof(*tvp));
270                 put_fs_long(timeout/HZ, (unsigned long *) &tvp->tv_sec);  // 秒。
271                 timeout %= HZ;
272                 timeout *= (1000000/HZ);
273                 put_fs_long(timeout, (unsigned long *) &tvp->tv_usec);  // 微秒。
274         }
    // 如果此时并没有已准备好的描述符，并且收到了某个非阻塞信号，则返回被中断错误号。
    // 否则返回已准备好的描述符个数值。
275         if (!i && (current->signal & ~current->blocked))
276                 return -EINTR;
277         return i;
278 }
279
```

第13章 内存管理程序

13.1 程序 13-1 linux/mm/memory.c

```
 1 /*
 2  *  linux/mm/memory.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * demand-loading started 01.12.91 - seems it is high on the list of
 9  * things wanted, and it should be easy to implement. - Linus
10  */
   /*
    * 需求加载是从 91.12.1 开始编写的 - 在程序编制表中似乎是最重要的程序，
    * 并且应该是很容易编制的 - Linus
    */
11
12 /*
13  * Ok, demand-loading was easy, shared pages a little bit tricker. Shared
14  * pages started 02.12.91, seems to work. - Linus.
15  *
16  * Tested sharing by executing about 30 /bin/sh: under the old kernel it
17  * would have taken more than the 6M I have free, but it worked well as
18  * far as I could see.
19  *
20  * Also corrected some "invalidate()"s - I wasn't doing enough of them.
21  */
   /*
    * OK，需求加载是比较容易编写的，而共享页面却需要有点技巧。共享页面程序是
    * 91.12.2 开始编写的，好象能够工作 - Linus。
    *
    * 通过执行大约 30 个/bin/sh 对共享操作进行了测试：在老内核当中需要占用多于
    * 6M 的内存，而目前却不用。现在看来工作得很好。
    *
    * 对"invalidate()"函数也进行了修正 - 在这方面我还做的不够。
    */
22
23 /*
24  * Real VM (paging to/from disk) started 18.12.91. Much more work and
25  * thought has to go into this. Oh, well..
26  * 19.12.91  -  works, somewhat. Sometimes I get faults, don't know why.
27  *              Found it. Everything seems to work now.
28  * 20.12.91  -  Ok, making the swap-device changeable like the root.
29  */
   /*
    * 91.12.18 开始编写真正的虚拟内存管理 VM（交换页面到/从磁盘）。需要对此
    * 考虑很多并且需要作很多工作。呵呵，也只能这样了。
    * 91.12.19  -  在某种程度上可以工作了，但有时会出错，不知道怎么回事。
```

```
 *              找到错误了，现在好像一切都能工作了。
 * 91.12.20  -  OK，把交换设备修改成可更改的了，就像根文件设备那样。
 */

#include <signal.h>       // 信号头文件。定义信号符号常量，信号结构及信号函数原型。

#include <asm/system.h>   // 系统头文件。定义设置或修改描述符/中断门等嵌入汇编宏。

#include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 的数据。
#include <linux/head.h>   // head 头文件，定义段描述符的简单结构，和几个选择符常量。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。

// CODE_SPACE(addr) ((((addr)+0xfff)&~0xfff)<current->start_code+current->end_code)。
// 该宏用于判断给定线性地址是否位于当前进程的代码段中，“(((addr)+4095)&~4095)”用于
// 取得线性地址 addr 所在内存页面的末端地址。参见 265 行。
#define CODE_SPACE(addr) ((((addr)+4095)&~4095) < \
current->start_code + current->end_code)

unsigned long HIGH_MEMORY = 0;          // 全局变量，存放实际物理内存最高端地址。

// 从 from 处复制 1 页内存到 to 处（4K 字节）。
#define copy_page(from,to) \
__asm__("cld ; rep ; movsl"::"S" (from),"D" (to),"c" (1024):"cx","di","si")

// 物理内存映射字节图（1 字节代表 1 页内存）。每个页面对应的字节用于标志页面当前被引用
// （占用）次数。它最大可以映射 15Mb 的内存空间。在初始化函数 mem_init()中，对于不能用
// 作主内存区页面的位置均都预先被设置成 USED（100）。
unsigned char mem_map [ PAGING_PAGES ] = {0,};

/*
 * Free a page of memory at physical address 'addr'. Used by
 * 'free_page_tables()'
 */
/*
 * 释放物理地址'addr'处的一页内存。用于函数'free_page_tables()'。
 */
//// 释放物理地址 addr 开始的 1 页面内存。
// 物理地址 1MB 以下的内存空间用于内核程序和缓冲，不作为分配页面的内存空间。因此
// 参数 addr 需要大于 1MB。
void free_page(unsigned long addr)
{
// 首先判断参数给定的物理地址 addr 的合理性。如果物理地址 addr 小于内存低端（1MB），
// 则表示在内核程序或高速缓冲中，对此不予处理。如果物理地址 addr >= 系统所含物理
// 内存最高端，则显示出错信息并且内核停止工作。
        if (addr < LOW_MEM) return;
        if (addr >= HIGH_MEMORY)
                panic("trying to free nonexistent page");
// 如果对参数 addr 验证通过，那么就根据这个物理地址换算出从内存低端开始计起的内存
// 页面号。页面号 = (addr - LOW_MEM)/4096。可见页面号从 0 号开始计起。此时 addr
// 中存放着页面号。如果该页面号对应的页面映射字节不等于 0，则减 1 返回。此时该映射
// 字节值应该为 0，表示页面已释放。如果对应页面字节原本就是 0，表示该物理页面本来
// 就是空闲的，说明内核代码出问题。于是显示出错信息并停机。
        addr -= LOW_MEM;
```

```
59          addr >>= 12;
60          if (mem_map[addr]--) return;
61          mem_map[addr]=0;
62          panic("trying to free free page");
63  }
64
65  /*
66   * This function frees a continuos block of page tables, as needed
67   * by 'exit()'. As does copy_page_tables(), this handles only 4Mb blocks.
68   */
    /*
     * 下面函数释放页表连续的内存块，'exit()'需要该函数。与 copy_page_tables()
     * 类似，该函数仅处理 4Mb 长度的内存块。
     */
    //// 根据指定的线性地址和限长（页表个数），释放对应内存页表指定的内存块并置表项空闲。
    // 页目录位于物理地址 0 开始处，共 1024 项，每项 4 字节，共占 4K 字节。每个目录项指定一
    // 个页表。内核页表从物理地址 0x1000 处开始（紧接着目录空间），共 4 个页表。每个页表有
    // 1024 项，每项 4 字节。因此也占 4K（1 页）内存。各进程（除了在内核代码中的进程 0 和 1）
    // 的页表所占据的页面在进程被创建时由内核为其在主内存区申请得到。每个页表项对应 1 页
    // 物理内存，因此一个页表最多可映射 4MB 的物理内存。
    // 参数：from – 起始线性基地址；size – 释放的字节长度。
69  int free_page_tables(unsigned long from,unsigned long size)
70  {
71          unsigned long *pg_table;
72          unsigned long * dir, nr;
73
    // 首先检测参数 from 给出的线性基地址是否在 4MB 的边界处。因为该函数只能处理这种情况。
    // 若 from = 0，则出错。说明试图释放内核和缓冲所占空间。
74          if (from & 0x3fffff)
75                  panic("free_page_tables called with wrong alignment");
76          if (!from)
77                  panic("Trying to free up swapper memory space");
    // 然后计算参数 size 给出的长度所占的页目录项数（4MB 的进位整数倍），也即所占页表数。
    // 因为 1 个页表可管理 4MB 物理内存，所以这里用右移 22 位的方式把需要复制的内存长度值
    // 除以 4MB。其中加上 0x3fffff（即 4Mb -1）用于得到进位整数倍结果，即除操作若有余数
    // 则进 1。例如，如果原 size = 4.01Mb，那么可得到结果 size = 2。 接着计算给出的线性
    // 基地址对应的起始目录项。对应的目录项号 = from >> 22。因为每项占 4 字节，并且由于
    // 页目录表从物理地址 0 开始存放，因此实际目录项指针 = 目录项号<<2，也即(from>>20)。
    // “与”上 0xffc 确保目录项指针范围有效。
78          size = (size + 0x3fffff) >> 22;
79          dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */

    // 此时 size 是释放的页表个数，即页目录项数，而 dir 是起始目录项指针。现在开始循环
    // 操作页目录项，依次释放每个页表中的页表项。如果当前目录项无效（P 位=0），表示该
    // 目录项没有使用（对应的页表不存在），则继续处理下一个目录项。否则从目录项中取出
    // 页表地址 pg_table，并对该页表中的 1024 个表项进行处理，释放有效页表项（P 位=1）
    // 对应的物理内存页面，或者从交换设备中释放无效页表项（P 位=0）对应的页面，即释放
    // 交换设备中对应的内存页面（因为页面可能已经交换出去）。然后把该页表项清零，并继
    // 续处理下一页表项。当一个页表所有表项都处理完毕就释放该页表自身占据的内存页面，
    // 并继续处理下一页目录项。最后刷新页变换高速缓冲，并返回 0。
80          for ( ; size-->0 ; dir++) {
81                  if (!(1 & *dir))
82                          continue;
```

```
83                  pg_table = (unsigned long *) (0xfffff000 & *dir);  // 取页表地址。
84                  for (nr=0 ; nr<1024 ; nr++) {
85                          if (*pg_table) {                // 若所指页表项内容不为 0，则
86                                  if (1 & *pg_table)      // 若该项有效，则释放对应页。
87                                          free_page(0xfffff000 & *pg_table);
88                                  else                    // 否则释放交换设备中对应页。
89                                          swap_free(*pg_table >> 1);
90                                  *pg_table = 0;          // 该页表项内容清零。
91                          }
92                          pg_table++;                     // 指向页表中下一项。
93                  }
94                  free_page(0xfffff000 & *dir);           // 释放该页表所占内存页面。
95                  *dir = 0;                               // 对应页表的目录项清零。
96          }
97          invalidate();                                   // 刷新 CPU 页变换高速缓冲。
98          return 0;
99  }
100
101 /*
102  *  Well, here is one of the most complicated functions in mm. It
103  * copies a range of linerar addresses by copying only the pages.
104  * Let's hope this is bug-free, 'cause this one I don't want to debug :-)
105  *
106  * Note! We don't copy just any chunks of memory - addresses have to
107  * be divisible by 4Mb (one page-directory entry), as this makes the
108  * function easier. It's used only by fork anyway.
109  *
110  * NOTE 2!! When from==0 we are copying kernel space for the first
111  * fork(). Then we DONT want to copy a full page-directory entry, as
112  * that would lead to some serious memory waste - we just copy the
113  * first 160 pages - 640kB. Even that is more than we need, but it
114  * doesn't take any more memory - we don't copy-on-write in the low
115  * 1 Mb-range, so the pages can be shared with the kernel. Thus the
116  * special case for nr=xxxx.
117  */
    /*
     * 好了，下面是内存管理 mm 中最为复杂的程序之一。它通过只复制内存页面
     * 来拷贝一定范围内线性地址中的内容。 希望代码中没有错误，因为我不想
     * 再调试这块代码了:-)。
     *
     * 注意！我们并不复制任何内存块 - 内存块的地址需要是 4Mb 的倍数（正好
     * 一个页目录项对应的内存长度），因为这样处理可使函数很简单。 不管怎
     * 样，它仅被 fork()使用。
     *
     * 注意 2！！ 当 from==0 时，说明是在为第一次 fork()调用复制内核空间。
     * 此时我们就不想复制整个页目录项对应的内存，因为这样做会导致内存严
     * 重浪费 - 我们只须复制开头 160 个页面 - 对应 640kB。即使是复制这些
     * 页面也已经超出我们的需求，但这不会占用更多的内存 - 在低 1Mb 内存
     * 范围内我们不执行写时复制操作，所以这些页面可以与内核共享。因此这
     * 是 nr=xxxx 的特殊情况（nr 在程序中指页面数）。
     */
    //// 复制页目录表项和页表项。
    // 复制指定线性地址和长度内存对应的页目录项和页表项，从而被复制的页目录和页表对应
```

```
    // 的原物理内存页面区被两套页表映射而共享使用。复制时，需申请新页面来存放新页表，
    // 原物理内存区将被共享。此后两个进程（父进程和其子进程）将共享内存区，直到有一个
    // 进程执行写操作时，内核才会为写操作进程分配新的内存页（写时复制机制）。
    // 参数 from、to 是线性地址，size 是需要复制（共享）的内存长度，单位是字节。
118 int copy_page_tables(unsigned long from,unsigned long to,long size)
119 {
120         unsigned long * from_page_table;
121         unsigned long * to_page_table;
122         unsigned long this_page;
123         unsigned long * from_dir, * to_dir;
124         unsigned long new_page;
125         unsigned long nr;
126
    // 首先检测参数给出的源地址 from 和目的地址 to 的有效性。源地址和目的地址都需要在 4Mb
    // 内存边界地址上。否则出错死机。作这样的要求是因为一个页表的 1024 项可管理 4Mb 内存。
    // 源地址 from 和目的地址 to 只有满足这个要求才能保证从一个页表的第 1 项开始复制页表
    // 项，并且新页表的最初所有项都是有效的。 然后取得源地址和目的地址的起始目录项指针
    // （from_dir 和 to_dir）。再根据参数给出的长度 size 计算要复制的内存块占用的页表数
    // （即目录项数）。参见前面对 78、79 行的解释。
127         if ((from&0x3fffff) || (to&0x3fffff))
128                 panic("copy_page_tables called with wrong alignment");
129         from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
130         to_dir = (unsigned long *) ((to>>20) & 0xffc);
131         size = ((unsigned) (size+0x3fffff)) >> 22;
    // 在得到了源起始目录项指针 from_dir 和目的起始目录项指针 to_dir 以及需要复制的页表
    // 个数 size 后，下面开始对每个页目录项依次申请 1 页内存来保存对应的页表，并且开始
    // 页表项复制操作。如果目的目录项指定的页表已经存在（P=1），则出错死机。 如果源目
    // 录项无效，即指定的页表不存在（P=0），则继续循环处理下一个页目录项。
132         for( ; size-->0 ; from_dir++,to_dir++) {
133                 if (1 & *to_dir)
134                         panic("copy_page_tables: already exist");
135                 if (!(1 & *from_dir))
136                         continue;

    // 在验证了当前源目录项和目的项正常之后，我们取源目录项中页表地址 from_page_table。
    // 为了保存目的目录项对应的页表，需要在主内存区中申请 1 页空闲内存页。如果取空闲页面
    // 函数 get_free_page() 返回 0，则说明没有申请到空闲内存页面，可能是内存不够。于是返
    // 回-1 值退出。
137                 from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
138                 if (!(to_page_table = (unsigned long *) get_free_page()))
139                         return -1;      /* Out of memory, see freeing */

    // 否则我们设置目的目录项信息，把最后 3 位置位，即当前目的目录项“或”上 7，表示对应
    // 页表映射的内存页面是用户级的，并且可读写、存在（Usr, R/W, Present）。 （如果 U/S
    // 位是 0，则 R/W 就没有作用。如果 U/S 是 1，而 R/W 是 0，那么运行在用户层的代码就只能
    // 读页面。如果 U/S 和 R/W 都置位，则就有读写的权限）。然后针对当前处理的页目录项对应
    // 的页表，设置需要复制的页面项数。如果是在内核空间，则仅需复制头 160 页对应的页表项
    // （nr= 160），对应于开始 640KB 物理内存。否则需要复制一个页表中的所有 1024 个页表项
    // （nr= 1024），可映射 4MB 物理内存。
140                 *to_dir = ((unsigned long) to_page_table) | 7;
141                 nr = (from==0)?0xA0:1024;

    // 此时对于当前页表，开始循环复制指定的 nr 个内存页面表项。先取出源页表项内容，如果
```

```
    // 当前源页面没有使用（项内容为 0），则不用复制该表项，继续处理下一项。
142                 for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
143                         this_page = *from_page_table;
144                         if (!this_page)
145                                 continue;
    // 如果该表项有内容，但是其存在位 P=0，则该表项对应的页面可能在交换设备中。于是先申
    // 请 1 页内存，并从交换设备中读入该页面（若交换设备中有的话）。然后将该页表项复制到
    // 目的页表项中。并修改源页表项内容指向该新申请的内存页，并设置表项标志为“页面脏”
    // 加上 7。然后继续处理下一页表项。否则复位页表项中 R/W 标志（位 1 置 0），即让页表项
    // 对应的内存页面只读，然后将该页表项复制到目的页表中。
146                         if (!(1 & this_page)) {
147                                 if (!(new_page = get_free_page()))
148                                         return -1;
149                                 read_swap_page(this_page>>1, (char *) new_page);
150                                 *to_page_table = this_page;
151                                 *from_page_table = new_page | (PAGE_DIRTY | 7);
152                                 continue;
153                         }
154                         this_page &= ~2;
155                         *to_page_table = this_page;

    // 如果该页表项所指物理页面的地址在 1MB 以上，则需要设置内存页面映射数组 mem_map[]，
    // 于是计算页面号，并以它为索引在页面映射数组相应项中增加引用次数。而对于位于 1MB
    // 以下的页面，说明是内核页面，因此不需要对 mem_map[]进行设置。因为 mem_map[]仅用
    // 于管理主内存区中的页面使用情况。 因此对于内核移动到任务 0 中并且调用 fork()创建
    // 任务 1 时（用于运行 init()），由于此时复制的页面还仍然都在内核代码区域，因此以下
    // 判断中的语句不会执行，任务 0 的页面仍然可以随时读写。只有当调用 fork() 的父进程
    // 代码处于主内存区（页面位置大于 1MB）时才会执行。这种情况需要在进程调用 execve()，
    // 并装载执行了新程序代码时才会出现。
    // 157 行语句含义是令源页表项所指内存页也为只读。因为现在开始已有两个进程共用内存
    // 区了。若其中 1 个进程需要进行写操作，则可以通过页异常写保护处理为执行写操作的进
    // 程分配 1 页新空闲页面，也即进行写时复制（copy on write）操作。
156                         if (this_page > LOW_MEM) {
157                                 *from_page_table = this_page; // 令源页表项也只读。
158                                 this_page -= LOW_MEM;
159                                 this_page >>= 12;
160                                 mem_map[this_page]++;
161                         }
162                 }
163         }
164         invalidate();                           // 刷新页变换高速缓冲。
165         return 0;
166 }
167
168 /*
169  * This function puts a page in memory at the wanted address.
170  * It returns the physical address of the page gotten, 0 if
171  * out of memory (either when trying to access page-table or
172  * page.)
173  */
    /*
     * 下面函数将一内存页面放置（映射）到指定线性地址处。它返回页面
     * 的物理地址，如果内存不够(在访问页表或页面时)，则返回 0。
```

```
 */
// 把一物理内存页面映射到线性地址空间指定处。
// 或者说是把线性地址空间中指定地址 address 处的页面映射到主内存区页面 page 上。主要
// 工作是在相关页目录项和页表项中设置指定页面的信息。若成功则返回物理页面地址。 在
// 处理缺页异常的 C 函数 do_no_page() 中会调用此函数。对于缺页引起的异常，由于任何缺
// 页缘故而对页表作修改时，并不需要刷新 CPU 的页变换缓冲（或称 Translation Lookaside
// Buffer - TLB），即使页表项中标志 P 被从 0 修改成 1。因为无效页项不会被缓冲，因此当
// 修改了一个无效的页表项时不需要刷新。在此就表现为不用调用 Invalidate()函数。
// 参数 page 是分配的主内存区中某一页面（页帧，页框）的指针；address 是线性地址。
174 static unsigned long put_page(unsigned long page,unsigned long address)
175 {
176         unsigned long tmp, *page_table;
177
178 /* NOTE !!! This uses the fact that _pg_dir=0 */
    /* 注意!!! 这里使用了页目录表基地址_pg_dir=0 的条件 */
179
    // 首先判断参数给定物理内存页面 page 的有效性。如果该页面位置低于 LOW_MEM（1MB）或
    // 超出系统实际含有内存高端 HIGH_MEMORY，则发出警告。LOW_MEM 是主内存区可能有的最
    // 小起始位置。当系统物理内存小于或等于 6MB 时，主内存区起始于 LOW_MEM 处。再查看一
    // 下该 page 页面是否是已经申请的页面，即判断其在内存页面映射字节图 mem_map[]中相
    // 应字节是否已经置位。若没有则需发出警告。
180         if (page < LOW_MEM || page >= HIGH_MEMORY)
181                 printk("Trying to put page %p at %p\n",page,address);
182         if (mem_map[(page-LOW_MEM)>>12] != 1)
183                 printk("mem_map disagrees with %p at %p\n",page,address);

    // 然后根据参数指定的线性地址 address 计算其在页目录表中对应的目录项指针，并从中取得
    // 二级页表地址。 如果该目录项有效（P=1），即指定的页表在内存中，则从中取得指定页表
    // 地址放到 page_table 变量中。否则申请一空闲页面给页表使用，并在对应目录项中置相应
    // 标志（7 - User、U/S、R/W）。然后将该页表地址放到 page_table 变量中。
184         page_table = (unsigned long *) ((address>>20) & 0xffc);
185         if ((*page_table)&1)
186                 page_table = (unsigned long *) (0xfffff000 & *page_table);
187         else {
188                 if (!(tmp=get_free_page()))
189                         return 0;
190                 *page_table = tmp | 7;
191                 page_table = (unsigned long *) tmp;
192         }
    // 最后在找到的页表 page_table 中设置相关页表项内容，即把物理页面 page 的地址填入表
    // 项同时置位 3 个标志（U/S、W/R、P）。该页表项在页表中的索引值等于线性地址位 21 --
    // 位 12 组成的 10 比特的值。每个页表共可有 1024 项（0 -- 0x3ff）。
193         page_table[(address>>12) & 0x3ff] = page | 7;
194 /* no need for invalidate */
    /* 不需要刷新页变换高速缓冲 */
195         return page;                    // 返回物理页面地址。
196 }
197
198 /*
199  * The previous function doesn't work very well if you also want to mark
200  * the page dirty: exec.c wants this, as it has earlier changed the page,
201  * and we want the dirty-status to be correct (for VM). Thus the same
202  * routine, but this time we mark it dirty too.
```

```
203  */
     /*
      * 如果你也想设置页面已修改标志，则上一个函数工作得不是很好：exec.c 程序
      * 需要这种设置。因为 exec.c 中函数会在放置页面之前修改过页面内容。为了实
      * 现 VM，我们需要能正确设置已修改状态标志。因而下面就有了与上面相同的函
      * 数，但是该函数在放置页面时会把页面标志为已修改状态。
      */
     // 把一内容已修改过的物理内存页面映射到线性地址空间指定处。
     // 该函数与上一个函数 put_page()几乎完全一样，除了本函数在 223 行设置页表项内容时，
     // 同时还设置了页面已修改标志（位 6，PAGE_DIRTY）。
204  unsigned long put_dirty_page(unsigned long page, unsigned long address)
205  {
206          unsigned long tmp, *page_table;
207
208  /* NOTE !!! This uses the fact that _pg_dir=0 */
209
210          if (page < LOW_MEM || page >= HIGH_MEMORY)
211                  printk("Trying to put page %p at %p\n",page,address);
212          if (mem_map[(page-LOW_MEM)>>12] != 1)
213                  printk("mem_map disagrees with %p at %p\n",page,address);
214          page_table = (unsigned long *) ((address>>20) & 0xffc);
215          if ((*page_table)&1)
216                  page_table = (unsigned long *) (0xfffff000 & *page_table);
217          else {
218                  if (!(tmp=get_free_page()))
219                          return 0;
220                  *page_table = tmp|7;
221                  page_table = (unsigned long *) tmp;
222          }
223          page_table[(address>>12) & 0x3ff] = page | (PAGE_DIRTY | 7);
224  /* no need for invalidate */
225          return page;
226  }
227
     //// 取消写保护页面函数。用于页异常中断过程中写保护异常的处理（写时复制）。
     // 在内核创建进程时，新进程与父进程被设置成共享代码和数据内存页面，并且所有这些页面
     // 均被设置成只读页面。而当新进程或原进程需要向内存页面写数据时，CPU 就会检测到这个
     // 情况并产生页面写保护异常。于是在这个函数中内核就会首先判断要写的页面是否被共享。
     // 若没有则把页面设置成可写然后退出。若页面是出于共享状态，则需要重新申请一新页面并
     // 复制被写页面内容，以供写进程单独使用。共享被取消。
     // 输入参数为页表项指针，是物理地址。[ un_wp_page -- Un-Write Protect Page]
228  void un_wp_page(unsigned long * table_entry)
229  {
230          unsigned long old_page,new_page;
231
     // 首先取参数指定的页表项中物理页面位置（地址）并判断该页面是否是共享页面。如果原
     // 页面地址大于内存低端 LOW_MEM（表示在主内存区中），并且其在页面映射字节图数组中
     // 值为 1（表示页面仅被引用 1 次，页面没有被共享），则在该页面的页表项中置 R/W 标志
     // （可写），并刷新页变换高速缓冲，然后返回。即如果该内存页面此时只被一个进程使用，
     // 并且不是内核中的进程，就直接把属性改为可写即可，不用再重新申请一个新页面。
232          old_page = 0xfffff000 & *table_entry;       // 取指定页表项中物理页面地址。
233          if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
234                  *table_entry |= 2;
```

```
235                 invalidate();
236                 return;
237         }
    // 否则就需要在主内存区内申请一页空闲页面给执行写操作的进程单独使用，取消页面共享。
    // 如果原页面大于内存低端（则意味着 mem_map[] > 1，页面是共享的），则将原页面的页
    // 面映射字节数组值递减 1。然后将指定页表项内容更新为新页面地址，并置可读写等标志
    // （U/S、R/W、P）。在刷新页变换高速缓冲之后，最后将原页面内容复制到新页面。
238         if (!(new_page=get_free_page()))
239                 oom();                          // Out of Memory。内存不够处理。
240         if (old_page >= LOW_MEM)
241                 mem_map[MAP_NR(old_page)]--;
242         copy_page(old_page,new_page);
243         *table_entry = new_page | 7;
244         invalidate();
245 }
246
247 /*
248  * This routine handles present pages, when users try to write
249  * to a shared page. It is done by copying the page to a new address
250  * and decrementing the shared-page counter for the old page.
251  *
252  * If it's in code space we exit with a segment error.
253  */
    /*
     * 当用户试图往一共享页面上写时，该函数处理已存在的内存页面（写时复制），
     * 它是通过将页面复制到一个新地址上并且递减原页面的共享计数值实现的。
     *
     * 如果它在代码空间，我们就显示段出错信息并退出。
     */
    //// 执行写保护页面处理。
    // 是写共享页面处理函数。是页异常中断处理过程中调用的 C 函数。在 page.s 程序中被调用。
    // 函数参数 error_code 和 address 是进程在写写保护页面时由 CPU 产生异常而自动生成的。
    // error_code 指出出错类型，参见本章开始处的“内存页面出错异常”一节；address 是产生
    // 异常的页面线性地址。写共享页面时需复制页面（写时复制）。
254 void do_wp_page(unsigned long error_code,unsigned long address)
255 {
    // 首先判断 CPU 控制寄存器 CR2 给出的引起页面异常的线性地址在什么范围中。如果 address
    // 小于 TASK_SIZE（0x4000000，即 64MB），表示异常页面位置在内核或任务 0 和任务 1 所处
    // 的线性地址范围内，于是发出警告信息“内核范围内存被写保护”；如果 (address - 当前
    // 进程代码起始地址)大于一个进程的长度（64MB），表示 address 所指的线性地址不在引起
    // 异常的进程线性地址空间范围内，则在发出出错信息后退出。
256         if (address < TASK_SIZE)
257                 printk("\n\rBAD! KERNEL MEMORY WP-ERR!\n\r");
258         if (address - current->start_code > TASK_SIZE) {
259                 printk("Bad things happen: page error in do_wp_page\n\r");
260                 do_exit(SIGSEGV);
261         }
262 #if 0
263 /* we cannot do this yet: the estdio library writes to code space */
264 /* stupid, stupid. I really want the libc.a from GNU */
    /* 我们现在还不能这样做：因为 estdio 库会在代码空间执行写操作 */
    /* 真是太愚蠢了。我真想从 GNU 得到 libc.a 库。*/
    // 如果线性地址位于进程的代码空间中，则终止执行程序。因为代码是只读的。
```

```
265         if (CODE_SPACE(address))
266                 do_exit(SIGSEGV);
267 #endif
    // 调用上面函数 un_wp_page()来处理取消页面保护。但首先需要为其准备好参数。参数是
    // 线性地址 address 指定页面在页表中的页表项指针，其计算方法是：
    // ① ((address>>10) & 0xffc)：计算指定线性地址中页表项在页表中的偏移地址；因为
    // 根据线性地址结构，(address>>12) 就是页表项中的索引，但每项占 4 个字节，因此乘
    // 4 后：(address>>12)<<2 = (address>>10)&0xffc 就可得到页表项在表中的偏移地址。
    // 与操作&0xffc 用于限制地址范围在一个页面内。 又因为只移动了 10 位，因此最后 2 位
    // 是线性地址低 12 位中的最高 2 位，也应屏蔽掉。 因此求线性地址中页表项在页表中偏
    // 移地址直观一些的表示方法是(((address>>12) & 0x3ff)<<2 )。
    // ② (0xfffff000 & *((address>>20) &0xffc))：用于取目录项中页表的地址值；其中，
    // ((address>>20) &0xffc)用于取线性地址中的目录索引项在目录表中的偏移位置。因为
    // address>>22 是目录项索引值，但每项 4 个字节，因此乘以 4 后： (address>>22)<<2
    // = (address>>20) 就是指定项在目录表中的偏移地址。 &0xffc 用于屏蔽目录项索引值
    // 中最后 2 位。因为只移动了 20 位，因此最后 2 位是页表索引的内容，应该屏蔽掉。而
    // *((address>>20) &0xffc) 则是取指定目录表项内容中对应页表的物理地址。最后与上
    // 0xfffff000 用于屏蔽掉页目录项内容中的一些标志位（目录项低 12 位）。直观表示为
    // (0xfffff000 & *((unsigned long *) (((address>>22) & 0x3ff)<<2)))。
    // ③ 由①中页表项在页表中偏移地址加上 ②中目录表项内容中对应页表的物理地址即可
    // 得到页表项的指针（物理地址）。这里对共享的页面进行复制。
268         un_wp_page((unsigned long *)
269                 (((address>>10) & 0xffc) + (0xfffff000 &
270                 *((unsigned long *) ((address>>20) &0xffc)))));
271
272 }
273
    //// 写页面验证。
    // 若页面不可写，则复制页面。在 fork.c 中第 34 行被内存验证通用函数 verify_area()调用。
    // 参数 address 是指定页面在 4G 空间中的线性地址。
274 void write_verify(unsigned long address)
275 {
276         unsigned long page;
277
    // 首先取指定线性地址对应的页目录项，根据目录项中的存在位（P）判断目录项对应的页表
    // 是否存在（存在位 P=1?），若不存在（P=0）则返回。这样处理是因为对于不存在的页面没
    // 有共享和写时复制可言，并且若程序对此不存在的页面执行写操作时，系统就会因为缺页异
    // 常而去执行 do_no_page()，并为这个地方使用 put_page()函数映射一个物理页面。
    // 接着程序从目录项中取页表地址，加上指定页面在页表中的页表项偏移值，得对应地址的页
    // 表项指针。在该表项中包含着给定线性地址对应的物理页面。
278         if (!( (page = *((unsigned long *) ((address>>20) & 0xffc)) )&1))
279                 return;
280         page &= 0xfffff000;
281         page += ((address>>10) & 0xffc);
    // 然后判断该页表项中的位 1（R/W）、位 0（P）标志。如果该页面不可写（R/W=0）且存在，
    // 那么就执行共享检验和复制页面操作（写时复制）。否则什么也不做，直接退出。
282         if ((3 & *(unsigned long *) page) == 1)  /* non-writeable, present */
283                 un_wp_page((unsigned long *) page);
284         return;
285 }
286
    //// 取得一页空闲内存页并映射到指定线性地址处。
    // get_free_page()仅是申请取得了主内存区的一页物理内存。而本函数则不仅是获取到一页
```

```
    // 物理内存页面，还进一步调用 put_page()，将物理页面映射到指定的线性地址处。
    // 参数 address 是指定页面的线性地址。
287 void get_empty_page(unsigned long address)
288 {
289         unsigned long tmp;
290
    // 若不能取得一空闲页面，或者不能将所取页面放置到指定地址处，则显示内存不够的信息。
    // 292 行上英文注释的含义是：free_page()函数的参数 tmp 是 0 也没有关系，该函数会忽略
    // 它并能正常返回。
291         if (!(tmp=get_free_page()) || !put_page(tmp,address)) {
292                 free_page(tmp);         /* 0 is ok - ignored */
293                 oom();
294         }
295 }
296
297 /*
298  * try_to_share() checks the page at address "address" in the task "p",
299  * to see if it exists, and if it is clean. If so, share it with the current
300  * task.
301  *
302  * NOTE! This assumes we have checked that p != current, and that they
303  * share the same executable or library.
304  */
    /*
     * try_to_share()在任务"p"中检查位于地址"address"处的页面，看页面是否存在，
     * 是否干净。如果是干净的话，就与当前任务共享。
     *
     * 注意！这里我们已假定 p !=当前任务，并且它们共享同一个执行程序或库程序。
     */
    //// 尝试对当前进程指定地址处的页面进行共享处理。
    // 当前进程与进程 p 是同一执行代码，也可以认为当前进程是由 p 进程执行 fork 操作产生的
    // 进程，因此它们的代码内容一样。如果未对数据段内容作过修改那么数据段内容也应一样。
    // 参数 address 是进程中的逻辑地址，即是当前进程欲与 p 进程共享页面的逻辑页面地址。
    // 进程 p 是将被共享页面的进程。如果 p 进程 address 处的页面存在并且没有被修改过的话，
    // 就让当前进程与 p 进程共享之。同时还需要验证指定的地址处是否已经申请了页面，若是
    // 则出错，死机。返回：1 - 页面共享处理成功；0 - 失败。
305 static int try_to_share(unsigned long address, struct task_struct * p)
306 {
307         unsigned long from;
308         unsigned long to;
309         unsigned long from_page;
310         unsigned long to_page;
311         unsigned long phys_addr;
312
    // 首先分别求得指定进程 p 中和当前进程中逻辑地址 address 对应的页目录项。为了计算方便
    // 先求出指定逻辑地址 address 处的'逻辑'页目录项号，即以进程空间（0 - 64MB）算出的页
    // 目录项号。该'逻辑'页目录项号加上进程 p 在 CPU 4G 线性空间中起始地址对应的页目录项，
    // 即得到进程 p 中地址 address 处页面所对应的 4G 线性空间中的实际页目录项 from_page。
    // 而'逻辑'页目录项号加上当前进程 CPU 4G 线性空间中起始地址对应的页目录项，即可最后
    // 得到当前进程中地址 address 处页面所对应的 4G 线性空间中的实际页目录项 to_page。
313         from_page = to_page = ((address>>20) & 0xffc);
314         from_page += ((p->start_code>>20) & 0xffc);      // p 进程目录项。
315         to_page += ((current->start_code>>20) & 0xffc);  // 当前进程目录项。
```

```
    // 在得到 p 进程和当前进程 address 对应的目录项后，下面分别对进程 p 和当前进程进行处理。
    // 下面首先对 p 进程的表项进行操作。目标是取得 p 进程中 address 对应的物理内存页面地址，
    // 并且该物理页面存在，而且干净（没有被修改过，不脏）。
    // 方法是先取目录项内容。如果该目录项无效（P=0），表示目录项对应的二级页表不存在，
    // 于是返回。否则取该目录项对应页表地址 from，从而计算出逻辑地址 address 对应的页表项
    // 指针，并取出该页表项内容临时保存在 phys_addr 中。
316 /* is there a page-directory at from? */
    /* 在 from 处是否存在页目录项？ */
317         from = *(unsigned long *) from_page;            // p 进程目录项内容。
318         if (!(from & 1))
319                 return 0;
320         from &= 0xfffff000;                             // 页表地址。
321         from_page = from + ((address>>10) & 0xffc);     // 页表项指针。
322         phys_addr = *(unsigned long *) from_page;       // 页表项内容。
    // 接着看看页表项映射的物理页面是否存在并且干净。 0x41 对应页表项中的 D（Dirty）和
    // P（Present）标志。如果页面不干净或无效则返回。然后我们从该表项中取出物理页面地址
    // 再保存在 phys_addr 中。最后我们再检查一下这个物理页面地址的有效性，即它不应该超过
    // 机器最大物理地址值，也不应该小于内存低端(1MB)。
323 /* is the page clean and present? */
    /* 物理页面干净并且存在吗？ */
324         if ((phys_addr & 0x41) != 0x01)
325                 return 0;
326         phys_addr &= 0xfffff000;                        // 物理页面地址。
327         if (phys_addr >= HIGH_MEMORY || phys_addr < LOW_MEM)
328                 return 0;

    // 下面首先对当前进程的表项进行操作。目标是取得当前进程中 address 对应的页表项地址，
    // 并且该页表项还没有映射物理页面，即其 P=0。
    // 首先取当前进程页目录项内容➔to。如果该目录项无效（P=0），即目录项对应的二级页表
    // 不存在，则申请一空闲页面来存放页表，并更新目录项 to_page 内容，让其指向该内存页面。
329         to = *(unsigned long *) to_page;                // 当前进程目录项内容。
330         if (!(to & 1))
331                 if (to = get_free_page())
332                         *(unsigned long *) to_page = to | 7;
333                 else
334                         oom();
    // 否则取目录项中的页表地址➔to，加上页表项索引值<<2，即页表项在表中偏移地址，得到
    // 页表项地址➔to_page。针对该页表项，如果此时我们检查出其对应的物理页面已经存在，
    // 即页表项的存在位 P=1，则说明原本我们想共享进程 p 中对应的物理页面，但现在我们自己
    // 已经占有了（映射有）物理页面。于是说明内核出错，死机。
335         to &= 0xfffff000;                               // 页表地址。
336         to_page = to + ((address>>10) & 0xffc);         // 页表项地址。
337         if (1 & *(unsigned long *) to_page)
338                 panic("try_to_share: to_page already exists");

    // 在找到了进程 p 中逻辑地址 address 处对应的干净并且存在的物理页面，而且也确定了当前
    // 进程中逻辑地址 address 所对应的二级页表项地址之后，我们现在对他们进行共享处理。
    // 方法很简单，就是首先对 p 进程的页表项进行修改，设置其写保护（R/W=0，只读）标志，
    // 然后让当前进程复制 p 进程的这个页表项。此时当前进程逻辑地址 address 处页面即被
    // 映射到 p 进程逻辑地址 address 处页面映射的物理页面上。
339 /* share them: write-protect */
    /* 对它们进行共享处理：写保护 */
```

```
340         *(unsigned long *) from_page &= ~2;
341         *(unsigned long *) to_page = *(unsigned long *) from_page;
    // 随后刷新页变换高速缓冲。计算所操作物理页面的页面号，并将对应页面映射字节数组项中
    // 的引用递增 1。最后返回 1，表示共享处理成功。
342         invalidate();
343         phys_addr -= LOW_MEM;
344         phys_addr >>= 12;                               // 得页面号。
345         mem_map[phys_addr]++;
346         return 1;
347 }
348
349 /*
350  * share_page() tries to find a process that could share a page with
351  * the current one. Address is the address of the wanted page relative
352  * to the current data space.
353  *
354  * We first check if it is at all feasible by checking executable->i_count.
355  * It should be >1 if there are other tasks sharing this inode.
356  */
    /*
     * share_page()试图找到一个进程，它可以与当前进程共享页面。参数 address 是
     * 当前进程数据空间中期望共享的某页面地址。
     *
     * 首先我们通过检测 executable->i_count 来查证是否可行。如果有其他任务已共享
     * 该 inode，则它应该大于 1。
     */
    //// 共享页面处理。
    // 在发生缺页异常时，首先看看能否与运行同一个执行文件的其他进程进行页面共享处理。
    // 该函数首先判断系统中是否有另一个进程也在运行当前进程一样的执行文件。若有，则在
    // 系统当前所有任务中寻找这样的任务。若找到了这样的任务就尝试与其共享指定地址处的
    // 页面。若系统中没有其他任务正在运行与当前进程相同的执行文件，那么共享页面操作的
    // 前提条件不存在，因此函数立刻退出。判断系统中是否有另一个进程也在执行同一个执行
    // 文件的方法是利用进程任务数据结构中的 executable 字段（或 library 字段）。该字段
    // 指向进程正在执行程序（或使用的库文件）在内存中的 i 节点。根据该 i 节点的引用次数
    // i_count 我们可以进行这种判断。 若节点的 i_count 值大于 1，则表明系统中有两个进程
    // 正在运行同一个执行文件（或库文件），于是可以再对任务结构数组中所有任务比较是否
    // 有相同的 executable 字段（或 library 字段）来最后确定多个进程运行着相同执行文件
    // 的情况。
    // 参数 inode 是欲进行共享页面进程执行文件的内存 i 节点。address 是进程中的逻辑地址，
    // 即是当前进程欲与 p 进程共享页面的逻辑页面地址。返回 1 - 共享操作成功，0 - 失败。
357 static int share_page(struct m_inode * inode, unsigned long address)
358 {
359         struct task_struct ** p;
360
    // 首先检查一下参数指定的内存 i 节点引用计数值。如果该内存 i 节点的引用计数值等于
    // 1（executable->i_count =1）或者 i 节点指针空，表示当前系统中只有 1 个进程在运行
    // 该执行文件或者提供的 i 节点无效。因此无共享可言，直接退出函数。
361         if (inode->i_count < 2 || !inode)
362                 return 0;
    // 否则搜索任务数组中所有任务。寻找与当前进程可共享页面的进程，即运行相同执行文件
    // 的另一个进程，并尝试对指定地址的页面进行共享。若进程逻辑地址 address 小于进程库
    // 文件在逻辑地址空间的起始地址 LIBRARY_OFFSET，则表明共享的页面在进程执行文件对应
    // 的逻辑地址空间范围内，于是检查一下指定 i 节点是否与进程的执行文件 i 节点（即进程
```

```
    // 的 executable 相同，若不相同则继续寻找。若进程逻辑地址 address 大于等于进程库文件
    // 在逻辑地址空间的起始地址 LIBRARY_OFFSET，则表明想要共享的页面在进程使用的库文件
    // 中，于是检查指定节点 inode 是否与进程的库文件 i 节点相同，若不相同则继续寻找。
    // 如果找到某个进程 p，其 executable 或 library 与指定的节点 inode 相同，则调用页面
    // 试探函数 try_to_share() 尝试页面共享。若共享操作成功，则函数返回 1。否则返回 0，
    // 表示共享页面操作失败。
363         for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
364                 if (!*p)                            // 如果该任务项空闲，则继续寻找。
365                         continue;
366                 if (current == *p)                  // 如果就是当前任务，也继续寻找。
367                         continue;
368                 if (address < LIBRARY_OFFSET) {
369                         if (inode != (*p)->executable)    // 进程执行文件 i 节点。
370                                 continue;
371                 } else {
372                         if (inode != (*p)->library)       // 进程使用库文件 i 节点。
373                                 continue;
374                 }
375                 if (try_to_share(address,*p))     // 尝试共享页面。
376                         return 1;
377         }
378         return 0;
379 }
380
    //// 执行缺页处理。
    // 访问不存在页面的处理函数，页异常中断处理过程中调用此函数。在 page.s 程序中被调用。
    // 函数参数 error_code 和 address 是进程在访问页面时由 CPU 因缺页产生异常而自动生成。
    // error_code 指出出错类型，参见本章开始处的“内存页面出错异常”一节；address 是产生
    // 异常的页面线性地址。
    // 该函数首先查看所缺页是否在交换设备中，若是则交换进来。否则尝试与已加载的相同文件
    // 进行页面共享，或者只是由于进程动态申请内存页面而只需映射一页物理内存页即可。若共
    // 享操作不成功，那么只能从相应文件中读入所缺的数据页面到指定线性地址处。
381 void do_no_page(unsigned long error_code,unsigned long address)
382 {
383         int nr[4];
384         unsigned long tmp;
385         unsigned long page;
386         int block,i;
387         struct m_inode * inode;
388
    // 首先判断 CPU 控制寄存器 CR2 给出的引起页面异常的线性地址在什么范围中。如果 address
    // 小于 TASK_SIZE（0x4000000，即 64MB），表示异常页面位置在内核或任务 0 和任务 1 所处
    // 的线性地址范围内，于是发出警告信息“内核范围内存被写保护”；如果 (address - 当前
    // 进程代码起始地址)大于一个进程的长度（64MB），表示 address 所指的线性地址不在引起
    // 异常的进程线性地址空间范围内，则在发出出错信息后退出。
389         if (address < TASK_SIZE)
390                 printk("\n\rBAD!! KERNEL PAGE MISSING\n\r");
391         if (address - current->start_code > TASK_SIZE) {
392                 printk("Bad things happen: nonexistent page error in do_no_page\n\r");
393                 do_exit(SIGSEGV);
394         }
    // 然后根据指定的线性地址 address 求出其对应的二级页表项指针，并根据该页表项内容判断
    // address 处的页面是否在交换设备中。若是则调入页面并退出。方法是首先取指定线性地址
```

```
    // address 对应的目录项内容。如果对应的二级页表存在，则取出该目录项中二级页表的地址，
    // 加上页表项偏移值即得到线性地址 address 处页面对应的页面表项指针，从而获得页表项内
    // 容。若页表项内容不为 0 并且页表项存在位 P=0，则说明该页表项指定的物理页面应该在交
    // 换设备中。于是从交换设备中调入指定页面后退出函数。
395         page = *(unsigned long *) ((address >> 20) & 0xffc); // 取目录项内容。
396         if (page & 1) {
397                 page &= 0xfffff000;                         // 二级页表地址。
398                 page += (address >> 10) & 0xffc;            // 页表项指针。
399                 tmp = *(unsigned long *) page;              // 页表项内容。
400                 if (tmp && !(1 & tmp)) {
401                         swap_in((unsigned long *) page);    // 从交换设备读页面。
402                         return;
403                 }
404         }
    // 否则取线性空间中指定地址 address 处页面地址，并算出指定线性地址在进程空间中相对于
    // 进程基址的偏移长度值 tmp，即对应的逻辑地址。从而可以算出缺页页面在执行文件映像中
    // 或在库文件中的具体起始数据块号。
405         address &= 0xfffff000;                              // address 处缺页页面地址。
406         tmp = address - current->start_code;                // 缺页页面对应逻辑地址。

    // 如果缺页对应的逻辑地址 tmp 大于库映像文件在进程逻辑空间中的起始位置，说明缺少的页
    // 面在库映像文件中。于是从当前进程任务数据结构中可以取得库映像文件的 i 节点 library，
    // 并计算出该缺页在库文件中的起始数据块号 block。如果缺页对应的逻辑地址 tmp 小于进程
    // 的执行映像文件在逻辑地址空间的末端位置，则说明缺少的页面在进程执行文件映像中，于
    // 是可以从当前进程任务数据机构中取得执行文件的 i 节点号 executable，并计算出该缺页
    // 在执行文件映像中的起始数据块号 block。若逻辑地址 tmp 既不在执行文件映像的地址范围
    // 内，也不在库文件空间范围内，则说明缺页是进程访问动态申请的内存页面数据所致，因此
    // 没有对应 i 节点和数据块号（都置空）。
    // 因为块设备上存放的执行文件映像第 1 块数据是程序头结构，因此在读取该文件时需要跳过
    // 第 1 块数据。所以需要首先计算缺页所在的数据块号。因为每块数据长度为 BLOCK_SIZE =
    // 1KB，因此一页内存可存放 4 个数据块。进程逻辑地址 tmp 除以数据块大小再加 1 即可得出
    // 缺少的页面在执行映像文件中的起始块号 block。
407         if (tmp >= LIBRARY_OFFSET ) {
408                 inode = current->library;           // 库文件 i 节点和缺页起始块号。
409                 block = 1 + (tmp-LIBRARY_OFFSET) / BLOCK_SIZE;
410         } else if (tmp < current->end_data) {
411                 inode = current->executable;        // 执行文件 i 节点和缺页起始块号。
412                 block = 1 + tmp / BLOCK_SIZE;
413         } else {
414                 inode = NULL;                       // 是动态申请的数据或栈内存页面。
415                 block = 0;
416         }
    // 若是进程访问其动态申请的页面或为了存放栈信息而引起的缺页异常，则直接申请一页物
    // 理内存页面并映射到线性地址 address 处即可。否则说明所缺页面在进程执行文件或库文
    // 件范围内，于是就尝试共享页面操作，若成功则退出。若不成功就只能申请一页物理内存
    // 页面 page，然后从设备上读取执行文件中的相应页面并放置（映射）到进程页面逻辑地址
    // tmp 处。
417         if (!inode) {                               // 是动态申请的数据内存页面。
418                 get_empty_page(address);
419                 return;
420         }
421         if (share_page(inode,tmp))                  // 尝试逻辑地址 tmp 处页面的共享。
422                 return;
```

```
423         if (!(page = get_free_page()))              // 申请一页物理内存。
424                 oom();
425 /* remember that 1 block is used for header */
    /* 记住，（程序）头要使用 1 个数据块 */
    // 根据这个块号和执行文件的 i 节点，我们就可以从映射位图中找到对应块设备中对应的设备
    // 逻辑块号（保存在 nr[]数组中）。利用 bread_page()即可把这 4 个逻辑块读入到物理页面
    // page 中。
426         for (i=0 ; i<4 ; block++,i++)
427                 nr[i] = bmap(inode,block);
428         bread_page(page,inode->i_dev,nr);

    // 在读设备逻辑块操作时，可能会出现这样一种情况，即在执行文件中的读取页面位置可能离
    // 文件尾不到 1 个页面的长度。因此就可能读入一些无用的信息。下面的操作就是把这部分超
    // 出执行文件 end_data 以后的部分进行清零处理。当然，若该页面离末端超过 1 页，说明不
    // 是从执行文件映像中读取的页面，而是从库文件中读取的，因此不用执行清零操作。
429         i = tmp + 4096 - current->end_data;          // 超出的字节长度值。
430         if (i>4095)                                  // 离末端超过 1 页则不用清零。
431                 i = 0;
432         tmp = page + 4096;                           // tmp 指向页面末端。
433         while (i-- > 0) {                            // 页面末端 i 字节清零。
434                 tmp--;
435                 *(char *)tmp = 0;
436         }
    // 最后把引起缺页异常的一页物理页面映射到指定线性地址 address 处。若操作成功就返回。
    // 否则就释放内存页，显示内存不够。
437         if (put_page(page,address))
438                 return;
439         free_page(page);
440         oom();
441 }
442
    //// 物理内存管理初始化。
    // 该函数对 1MB 以上内存区域以页面为单位进行管理前的初始化设置工作。一个页面长度为
    // 4KB 字节。该函数把 1MB 以上所有物理内存划分成一个个页面，并使用一个页面映射字节
    // 数组 mem_map[] 来管理所有这些页面。对于具有 16MB 内存容量的机器，该数组共有 3840
    // 项 ((16MB - 1MB)/4KB)，即可管理 3840 个物理页面。每当一个物理内存页面被占用时就
    // 把 mem_map[]中对应的的字节值增 1；若释放一个物理页面，就把对应字节值减 1。 若字
    // 节值为 0，则表示对应页面空闲； 若字节值大于或等于 1，则表示对应页面被占用或被不
    // 同程序共享占用。
    // 在该版本的 Linux 内核中，最多能管理 16MB 的物理内存，大于 16MB 的内存将弃置不用。
    // 对于具有 16MB 内存的 PC 机系统，在没有设置虚拟盘 RAMDISK 的情况下 start_mem 通常
    // 是 4MB，end_mem 是 16MB。因此此时主内存区范围是 4MB—16MB，共有 3072 个物理页面可
    // 供分配。而范围 0 - 1MB 内存空间用于内核系统（其实内核只使用 0 —640Kb，剩下的部
    // 分被部分高速缓冲和设备内存占用）。
    // 参数 start_mem 是可用作页面分配的主内存区起始地址（已去除 RAMDISK 所占内存空间）。
    // end_mem 是实际物理内存最大地址。而地址范围 start_mem 到 end_mem 是主内存区。
443 void mem_init(long start_mem, long end_mem)
444 {
445         int i;
446
    // 首先将 1MB 到 16MB 范围内所有内存页面对应的内存映射字节数组项置为已占用状态，即各
    // 项字节值全部设置成 USED（100）。PAGING_PAGES 被定义为(PAGING_MEMORY>>12)，即 1MB
    // 以上所有物理内存分页后的内存页面数(15MB/4KB = 3840)。
```

```
447         HIGH_MEMORY = end_mem;                      // 设置内存最高端（16MB）。
448         for (i=0 ; i<PAGING_PAGES ; i++)
449                 mem_map[i] = USED;
    // 然后计算主内存区起始内存 start_mem 处页面对应内存映射字节数组中项号 i 和主内存区
    // 页面数。此时 mem_map[] 数组的第 i 项正对应主内存区中第 1 个页面。最后将主内存区中
    // 页面对应的数组项清零（表示空闲）。对于具有 16MB 物理内存的系统，mem_map[] 中对应
    // 4Mb--16Mb 主内存区的项被清零。
450         i = MAP_NR(start_mem);                      // 主内存区起始位置处页面号。
451         end_mem -= start_mem;
452         end_mem >>= 12;                             // 主内存区中的总页面数。
453         while (end_mem-->0)
454                 mem_map[i++]=0;                     // 主内存区页面对应字节值清零。
455 }
456
    // 显示系统内存信息。
    // 根据内存映射字节数组 mem_map[] 中的信息以及页目录和页表内容统计系统中使用的内存页
    // 面数和主内存区中总物理内存页面数。该函数在 chr_drv/keyboard.S 程序第 186 行被调用。
    // 即当按下“Shift + Scroll Lock”组合键时会显示系统内存统计信息。
457 void show_mem(void)
458 {
459         int i,j,k,free=0,total=0;
460         int shared=0;
461         unsigned long * pg_tbl;
462
    // 根据内存映射字节数组 mem_map[]，统计系统主内存区页面总数 total，以及其中空闲页面
    // 数 free 和被共享的页面数 shared。并这些信息显示。
463         printk("Mem-info:\n\r");
464         for(i=0 ; i<PAGING_PAGES ; i++) {
465                 if (mem_map[i] == USED)             // 1MB 以上内存系统占用的页面。
466                         continue;
467                 total++;
468                 if (!mem_map[i])
469                         free++;                     // 主内存区空闲页面统计。
470                 else
471                         shared += mem_map[i]-1;     // 共享的页面数（字节值>1）。
472         }
473         printk("%d free pages of %d\n\r",free,total);
474         printk("%d pages shared\n\r",shared);

    // 统计处理器分页管理逻辑页面数。页目录表前 4 项供内核代码使用，不列为统计范围，因此
    // 扫描处理的页目录项从第 5 项开始。方法是循环处理所有页目录项（除前 4 个项），若对应
    // 的二级页表存在，那么先统计二级页表本身占用的内存页面（484 行），然后对该页表中所
    // 有页表项对应页面情况进行统计。
475         k = 0;                                      // 一个进程占用页面统计值。
476         for(i=4 ; i<1024 ;) {
477                 if (1&pg_dir[i]) {
    // （如果页目录项对应二级页表地址大于机器最高物理内存地址 HIGH_MEMORY，则说明该目录项
    // 有问题。于是显示该目录项信息并继续处理下一个目录项。）
478                         if (pg_dir[i]>HIGH_MEMORY) {        // 目录项内容不正常。
479                                 printk("page directory[%d]: %08X\n\r",
480                                         i,pg_dir[i]);
481                                 continue;           // continue 之前需插入 i++;
482                         }
```

```
    // 如果页目录项对应二级页表的“地址”大于 LOW_MEM（即 1MB），则把一个进程占用的物理
    // 内存页统计值 k 增 1，把系统占用的所有物理内存页统计值 free 增 1。然后取对应页表地址
    // pg_tbl，并对该页表中所有页表项进行统计。如果当前页表项所指物理页面存在并且该物理
    // 页面“地址”大于 LOW_MEM，那么就将页表项对应页面纳入统计值。
483                     if (pg_dir[i]>LOW_MEM)
484                             free++,k++;             // 统计页表占用页面。
485                     pg_tbl=(unsigned long *) (0xfffff000 & pg_dir[i]);
486                     for(j=0 ; j<1024 ; j++)
487                             if ((pg_tbl[j]&1) && pg_tbl[j]>LOW_MEM)
    // （若该物理页面地址大于机器最高物理内存地址 HIGH_MEMORY，则说明该页表项内容有问题，
    // 于是显示该页表项内容。否则将页表项对应页面纳入统计值。）
488                                     if (pg_tbl[j]>HIGH_MEMORY)
489                                             printk("page_dir[%d][%d]: %08X\n\r",
490                                                     i,j, pg_tbl[j]);
491                                     else
492                                             k++,free++;  // 统计页表项对应页面。
493                 }
    // 因每个任务线性空间长度是 64MB，所以一个任务占用 16 个目录项。因此这里每统计了 16 个
    // 目录项就把进程的任务结构占用的页表统计进来。若此时 k=0 则表示当前的 16 个页目录所对
    // 应的进程在系统中不存在（没有创建或者已经终止）。 在显示了对应进程号和其占用的物理
    // 内存页统计值 k 后，将 k 清零，以用于统计下一个进程占用的内存页面数。
494                 i++;
495                 if (!(i&15) && k) {                 // k !=0 表示相应进程存在。
496                         k++,free++;     /* one page/process for task_struct */
497                         printk("Process %d: %d pages\n\r",(i>>4)-1,k);
498                         k = 0;
499                 }
500         }
    // 最后显示系统中正在使用的内存页面和主内存区中总的内存页面数。
501         printk("Memory found: %d (%d)\n\r",free-shared,total);
502 }
503
```

13.2 程序 13-2 linux/mm/page.s

```
 1 /*
 2  *  linux/mm/page.s
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * page.s contains the low-level page-exception code.
 9  * the real work is done in mm.c
10  */
   /*
    * page.s 程序包含底层页异常处理代码。实际工作在 memory.c 中完成。
    */
11
12 .globl _page_fault           # 声明为全局变量。将在 traps.c 中用于设置页异常描述符。
13
14 _page_fault:
15         xchgl %eax,(%esp)    # 取出错码到 eax。
16         pushl %ecx
17         pushl %edx
18         push %ds
19         push %es
20         push %fs
21         movl $0x10,%edx      # 置内核数据段选择符。
22         mov %dx,%ds
23         mov %dx,%es
24         mov %dx,%fs
25         movl %cr2,%edx       # 取引起页面异常的线性地址。
26         pushl %edx           # 将该线性地址和出错码压入栈中，作为将调用函数的参数。
27         pushl %eax
28         testl $1,%eax        # 测试页存在标志 P（位 0），如果不是缺页引起的异常则跳转。
29         jne 1f
30         call _do_no_page     # 调用缺页处理函数（mm/memory.c,365 行）。
31         jmp 2f
32 1:      call _do_wp_page     # 调用写保护处理函数（mm/memory.c,247 行）。
33 2:      addl $8,%esp         # 丢弃压入栈的两个参数，弹出栈中寄存器并退出中断。
34         pop %fs
35         pop %es
36         pop %ds
37         popl %edx
38         popl %ecx
39         popl %eax
40         iret
```

13.3 程序 13-3 linux/mm/swap.c

```
 1 /*
 2  *  linux/mm/swap.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * This file should contain most things doing the swapping from/to disk.
 9  * Started 18.12.91
10  */
   /*
    * 本程序应该包括绝大部分执行内存交换的代码（从内存到磁盘或反之）。
    * 从 91 年 12 月 18 日开始编制。
    */
11
12 #include <string.h>        // 字符串头文件。定义了一些有关内存或字符串操作的嵌入函数。
13
14 #include <linux/mm.h>      // 内存管理头文件。定义页面长度，和一些内存管理函数原型。
15 #include <linux/sched.h>   // 调度程序头文件。定义了任务结构 task_struct、任务 0 的数据，
                              // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
16 #include <linux/head.h>    // head 头文件。定义了段描述符的简单结构，和几个选择符常量。
17 #include <linux/kernel.h>  // 内核头文件。含有一些内核常用函数的原型定义。
18
   // 每个字节 8 位，因此 1 页（4096 字节）共有 32768 个比特位。若 1 个比特位对应 1 页内存，
   // 则最多可管理 32768 个页面，对应 128MB 内存容量。
19 #define SWAP_BITS (4096<<3)
20
   // 比特位操作宏。通过给定不同的"op"，可定义对指定比特位进行测试、设置或清除三种操作。
   // 参数 addr 是指定线性地址；nr 是指定地址处开始的比特位偏移位。该宏把给定地址 addr 处
   // 第 nr 个比特位的值放入进位标志，设置或复位该比特位并返回进位标志值（即原比特位值）。
   // 第 25 行上第一个指令随"op"字符的不同而组合形成不同的指令：
   // 当"op"= ""时，就是指令 bt  -  （Bit Test）测试并用原值设置进位位。
   // 当"op"="s"时，就是指令 bts -  （Bit Test and Set）设置比特位值并用原值设置进位位。
   // 当"op"="r"时，就是指令 btr -  （Bit Test and Reset）复位比特位值并原值设置进位位。
   // 输入：%0  -  （返回值），%1 -位偏移(nr)；%2  -基址(addr)；%3  -  加操作寄存器初值(0)。
   // 内嵌汇编代码把基地址（%2）和比特偏移值（%1）所指定的比特位值先保存到进位标志 CF 中，
   // 然后设置（复位）该比特位。指令 adcl 是带进位位加，用于根据进位位 CF 设置操作数（%0）。
   // 如果 CF = 1 则返回寄存器值 = 1，否则返回寄存器值 = 0 。
21 #define bitop(name,op) \
22 static inline int name(char * addr,unsigned int nr) \
23 { \
24 int __res; \
25 __asm__ __volatile__("bt" op " %1,%2; adcl $0,%0" \
26 :"=g" (__res) \
27 :"r" (nr),"m" (*(addr)),"0" (0)); \
28 return __res; \
29 }
30
   // 这里根据不同的 op 字符定义 3 个内嵌函数。
```

```
31 bitop(bit,"")                // 定义内嵌函数 bit(char * addr, unsigned int nr)。
32 bitop(setbit,"s")             // 定义内嵌函数 setbit(char * addr, unsigned int nr)。
33 bitop(clrbit,"r")             // 定义内嵌函数 clrbit(char * addr, unsigned int nr)。
34
35 static char * swap_bitmap = NULL;
36 int SWAP_DEV = 0;                              // 内核初始化时设置的交换设备号。
37
38 /*
39  * We never page the pages in task[0] - kernel memory.
40  * We page all other pages.
41  */
   /*
    * 我们从不交换任务 0（task[0]）的页面 - 即不交换内核页面。
    * 我们只对其他页面进行交换操作。
    */
   // 第 1 个虚拟内存页面。即从任务 0 末端（64MB）处开始的虚拟内存页面。
42 #define FIRST_VM_PAGE (TASK_SIZE>>12)              // = 64MB/4KB = 16384。
43 #define LAST_VM_PAGE (1024*1024)                   // = 4GB/4KB  = 1048576。
44 #define VM_PAGES (LAST_VM_PAGE - FIRST_VM_PAGE)    // = 1032192（从 0 开始计）。
45
   // 申请 1 页交换页面。
   // 扫描整个交换映射位图（除对应位图本身的位 0 以外），返回值为 1 的第一个比特位号，
   // 即目前空闲的交换页面号。若操作成功则返回交换页面号，否则返回 0。
46 static int get_swap_page(void)
47 {
48         int nr;
49
50         if (!swap_bitmap)
51                 return 0;
52         for (nr = 1; nr < 32768 ; nr++)
53                 if (clrbit(swap_bitmap,nr))
54                         return nr;                 // 返回目前空闲的交换页面号。
55         return 0;
56 }
57
   // 释放交换设备中指定的交换页面。
   // 在交换位图中设置指定页面号对应的比特位（置 1）。若原来该比特位就等于 1，则表示
   // 交换设备中原来该页面就没有被占用，或者位图出错。于是显示出错信息并返回。
   // 参数指定交换页面号。
58 void swap_free(int swap_nr)
59 {
60         if (!swap_nr)
61                 return;
62         if (swap_bitmap && swap_nr < SWAP_BITS)
63                 if (!setbit(swap_bitmap,swap_nr))
64                         return;
65         printk("Swap-space bad (swap_free())\n\r");
66         return;
67 }
68
   // 把指定页面交换进内存中。
   // 把指定页表项的对应页面从交换设备中读入到新申请的内存页面中。修改交换位图中对应
   // 比特位（置位），同时修改页表项内容，让它指向该内存页面，并设置相应标志。
```

```
69 void swap_in(unsigned long *table_ptr)
70 {
71         int swap_nr;
72         unsigned long page;
73
   // 首先检查交换位图和参数有效性。如果交换位图不存在，或者指定页表项对应的页面已存在
   // 于内存中，或者交换页面号为 0，则显示警告信息并退出。对于已放到交换设备中去的内存
   // 页面，相应页表项中存放的应是交换页面号*2，即(swap_nr << 1)，参见下面对尝试交换函
   // 数 try_to_swap_out()中第 111 行的说明。
74         if (!swap_bitmap) {
75                 printk("Trying to swap in without swap bit-map");
76                 return;
77         }
78         if (1 & *table_ptr) {
79                 printk("trying to swap in present page\n\r");
80                 return;
81         }
82         swap_nr = *table_ptr >> 1;
83         if (!swap_nr) {
84                 printk("No swap page in swap_in\n\r");
85                 return;
86         }
   // 然后申请一页物理内存并从交换设备中读入页面号为 swap_nr 的页面。在把页面交换进来
   // 后，就把交换位图中对应比特位置位。如果其原本就是置位的，说明此次是再次从交换设
   // 备中读入相同的页面，于是显示一下警告信息。最后让页表项指向该物理页面，并设置页
   // 面已修改、用户可读写和存在标志（Dirty、U/S、R/W、P）。
87         if (!(page = get_free_page()))
88                 oom();
89         read_swap_page(swap_nr, (char *) page);  // 在 include/linux/mm.h 中定义。
90         if (setbit(swap_bitmap,swap_nr))
91                 printk("swapping in multiply from same page\n\r");
92         *table_ptr = page | (PAGE_DIRTY | 7);
93 }
94
   // 尝试把页面交换出去。
   // 若页面没有被修改过则不用保存在交换设备中，因为对应页面还可以再直接从相应映像文件
   // 中读入。于是可以直接释放掉相应物理页面了事。否则就申请一个交换页面号，然后把页面
   // 交换出去。此时交换页面号要保存在对应页表项中，并且仍需要保持页表项存在位 P = 0。
   // 参数是页表项指针。页面交换或释放成功返回 1，否则返回 0。
95 int try_to_swap_out(unsigned long * table_ptr)
96 {
97         unsigned long page;
98         unsigned long swap_nr;
99
   // 首先判断参数的有效性。若需要交换出去的内存页面并不存在（或称无效），则即可退出。
   // 若页表项指定的物理页面地址大于分页管理的内存高端 PAGING_MEMORY（15MB），也退出。
100        page = *table_ptr;
101        if (!(PAGE_PRESENT & page))
102                return 0;
103        if (page - LOW_MEM > PAGING_MEMORY)
104                return 0;
   // 若内存页面已被修改过，但是该页面是被共享的，那么为了提高运行效率，此类页面不宜
   // 被交换出去，于是直接退出，函数返回 0。否则就申请一交换页面号，并把它保存在页表
```

```
    // 项中，然后把页面交换出去并释放对应物理内存页面。
105         if (PAGE_DIRTY & page) {
106                 page &= 0xfffff000;                     // 取物理页面地址。
107                 if (mem_map[MAP_NR(page)] != 1)
108                         return 0;
109                 if (!(swap_nr = get_swap_page()))       // 申请交换页面号。
110                         return 0;
    // 对于要到交换设备中的页面，相应页表项中将存放的是(swap_nr << 1)。 乘 2（左移 1 位）
    // 是为了空出原来页表项的存在位（P）。只有存在位 P=0 并且页表项内容不为 0 的页面才会
    // 在交换设备中。 Intel 手册中明确指出，当一个表项的存在位 P = 0 时（无效页表项），
    // 所有其他位（位 31—1）可供随意使用。下面写交换页函数 write_swap_page(nr, buffer)
    // 被定义为 ll_rw_page(WRITE,SWAP_DEV,(nr),(buffer))。参见 linux/mm.h 文件第 12 行。
111                 *table_ptr = swap_nr<<1;
112                 invalidate();                           // 刷新 CPU 页变换高速缓冲。
113                 write_swap_page(swap_nr, (char *) page);
114                 free_page(page);
115                 return 1;
116         }
    // 否则表明页面没有修改过。那么就不用交换出去，而直接释放即可。
117         *table_ptr = 0;
118         invalidate();
119         free_page(page);
120         return 1;
121 }
122
123 /*
124  * Ok, this has a rather intricate logic - the idea is to make good
125  * and fast machine code. If we didn't worry about that, things would
126  * be easier.
127  */
    /*
     * OK，这个函数中有一个非常复杂的逻辑 - 用于产生逻辑性好并且速度快的
     * 机器码。如果我们不对此操心的话，那么事情可能更容易些。
     */
    // 把内存页面放到交换设备中。
    // 从线性地址 64MB 对应的目录项（FIRST_VM_PAGE>>10）开始，搜索整个 4GB 线性空间，对有
    // 效页目录二级页表的页表项指定的物理内存页面执行交换到交换设备中去的尝试。一旦成功
    // 地换出一个页面，就返回 1。否则返回 0。该函数会在 get_free_page()中被调用。
128 int swap_out(void)
129 {
130         static int dir_entry = FIRST_VM_PAGE>>10;   // 即任务 1 的第 1 个目录项索引。
131         static int page_entry = -1;
132         int counter = VM_PAGES;
133         int pg_table;
134
    // 首先搜索页目录表，查找二级页表存在的页目录项 pg_table。找到则退出循环，否则调整
    // 页目录项数对应剩余二级页表项数 counter，然后继续检测下一页目录项。 若全部搜索完
    // 还没有找到适合的（存在的）页目录项，就重新继续搜索。
135         while (counter>0) {
136                 pg_table = pg_dir[dir_entry];           // 页目录项内容。
137                 if (pg_table & 1)
138                         break;
139                 counter -= 1024;                        // 1 个页表对应 1024 个页帧。
```

```
140                 dir_entry++;                            // 下一目录项。
141                 if (dir_entry >= 1024)
142                         dir_entry = FIRST_VM_PAGE>>10;
143         }
    // 在取得当前目录项的页表指针后，针对该页表中的所有 1024 个页面，逐一调用交换函数
    // try_to_swap_out()尝试交换出去。一旦某个页面成功交换到交换设备中就返回 1。若对所
    // 有目录项的所有页表都已尝试失败，则显示“交换内存用完”的警告，并返回 0。
144         pg_table &= 0xfffff000;                         // 页表指针（地址）。
145         while (counter-- > 0) {
146                 page_entry++;                           // 页表项索引（初始为-1）。
    // 如果已经尝试处理完当前页表所有项还没有能够成功地交换出一个页面，即此时页表项索引
    // 大于等于 1024，则如同前面第 135 - 143 行执行相同的处理来选出一个二级页表存在的页
    // 目录项，并取得相应二级页表指针。
147                 if (page_entry >= 1024) {
148                         page_entry = 0;
149                 repeat:
150                         dir_entry++;
151                         if (dir_entry >= 1024)
152                                 dir_entry = FIRST_VM_PAGE>>10;
153                         pg_table = pg_dir[dir_entry];  // 页目录项内容。
154                         if (!(pg_table&1))
155                                 if ((counter -= 1024) > 0)
156                                         goto repeat;
157                                 else
158                                         break;
159                         pg_table &= 0xfffff000;         // 页表指针。
160                 }
161                 if (try_to_swap_out(page_entry + (unsigned long *) pg_table))
162                         return 1;
163         }
164         printk("Out of swap-memory\n\r");
165         return 0;
166 }
167
168 /*
169  * Get physical address of first (actually last :-) free page, and mark it
170  * used. If no free pages left, return 0.
171  */
    /*
     * 获取首个(实际上是最后 1 个:-)空闲页面，并标记为已使用。如果没有空闲页面，
     * 就返回 0。
     */
    //// 在主内存区中申请 1 页空闲物理页面。
    // 如果已经没有可用物理内存页面，则调用执行交换处理。然后再次申请页面。
    // 输入：%1(ax=0) - 0；%2(LOW_MEM)内存字节位图管理的起始位置；%3(cx= PAGING_PAGES)；
    // %4(edi=mem_map+PAGING_PAGES-1)。
    // 输出：返回%0（ax = 物理页面起始地址）。函数返回新页面的物理地址。
    // 上面%4 寄存器实际指向 mem_map[]内存字节位图的最后一个字节。本函数从位图末端开始向
    // 前扫描所有页面标志（页面总数为 PAGING_PAGES），若有页面空闲（内存位图字节为 0）则
    // 返回页面地址。注意！本函数只是指出在主内存区的一页空闲物理页面，但并没有映射到某
    // 个进程的地址空间中去。后面的 put_page() 函数即用于把指定页面映射到某个进程的地址
    // 空间中。当然对于内核使用本函数并不需要再使用 put_page() 进行映射，因为内核代码和
    // 数据空间（16MB）已经对等地映射到物理地址空间。
```

```
    // 第 65 行定义了一个局部寄存器变量。该变量将被保存在 eax 寄存器中，以便于高效访问和
    // 操作。这种定义变量的方法主要用于内嵌汇编程序中。详细说明参见 gcc 手册“在指定寄存
    // 器中的变量”。
172 unsigned long get_free_page(void)
173 {
174 register unsigned long __res asm("ax");
175
    // 首先在内存映射字节位图中查找值为 0 的字节项，然后把对应物理内存页面清零。如果得到
    // 的页面地址大于实际物理内存容量则重新寻找。如果没有找到空闲页面则去调用执行交换处
    // 理，并重新查找。最后返回空闲物理页面地址。
176 repeat:
177         __asm__("std ; repne ; scasb\n\t"     // 置方向位，al(0)与对应每个页面的(di)内容比较，
178                 "jne 1f\n\t"                  // 如果没有等于 0 的字节，则跳转结束（返回 0）。
179                 "movb $1,1(%%edi)\n\t"         // 1 =>[1+edi]，将对应页面内存映像比特位置 1。
180                 "sall $12,%%ecx\n\t"          // 页面数*4K = 相对页面起始地址。
181                 "addl %2,%%ecx\n\t"           // 再加上低端内存地址，得页面实际物理起始地址。
182                 "movl %%ecx,%%edx\n\t"        // 将页面实际起始地址➔edx 寄存器。
183                 "movl $1024,%%ecx\n\t"        // 寄存器 ecx 置计数值 1024。
184                 "leal 4092(%%edx),%%edi\n\t"  // 将 4092+edx 的位置➔edi（该页面的末端）。
185                 "rep ; stosl\n\t"             // 将 edi 所指内存清零（反方向，即将该页面清零）。
186                 "movl %%edx,%%eax\n"          // 将页面起始地址➔eax（返回值）。
187                 "1:"
188                 :"=a" (__res)
189                 :"0" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
190                 "D" (mem_map+PAGING_PAGES-1)
191                 :"di","cx","dx");
192         if (__res >= HIGH_MEMORY)     // 页面地址大于实际内存容量则重新寻找。
193                 goto repeat;
194         if (!__res && swap_out())     // 若没得到空闲页面则执行交换处理，并重新查找。
195                 goto repeat;
196         return __res;                 // 返回空闲物理页面地址。
197 }
198
    // 内存交换初始化。
199 void init_swapping(void)
200 {
    // blk_size[]指向指定主设备号的块设备块数数组。该块数数组每一项对应一个子设备上所
    // 拥有的数据块总数（1 块大小=1KB）。
201         extern int *blk_size[];             // blk_drv/ll_rw_blk.c，49 行。
202         int swap_size,i,j;
203
    // 如果没有定义交换设备则返回。如果交换设备没有设置块数数组，则显示信息并返回。
204         if (!SWAP_DEV)
205                 return;
206         if (!blk_size[MAJOR(SWAP_DEV)]) {
207                 printk("Unable to get size of swap device\n\r");
208                 return;
209         }
    // 取指定交换设备号的交换区数据块总数 swap_size。若为 0 则返回，若总块数小于 100 块
    // 则显示信息“交换设备区太小”，然后退出。
210         swap_size = blk_size[MAJOR(SWAP_DEV)][MINOR(SWAP_DEV)];
211         if (!swap_size)
212                 return;
```

```
213         if (swap_size < 100) {
214                 printk("Swap device too small (%d blocks)\n\r",swap_size);
215                 return;
216         }
    // 交换数据块总数转换成对应可交换页面总数。该值不能大于 SWAP_BITS 所能表示的页面数。
    // 即交换页面总数不得大于 32768。  然后申请一页物理内存用来存放交换页面位映射数组
    // swap_bitmap，其中每 1 比特代表 1 页交换页面。
217         swap_size >>= 2;
218         if (swap_size > SWAP_BITS)
219                 swap_size = SWAP_BITS;
220         swap_bitmap = (char *) get_free_page();
221         if (!swap_bitmap) {
222                 printk("Unable to start swapping: out of memory :-)\n\r");
223                 return;
224         }
    // read_swap_page(nr, buffer)被定义为 ll_rw_page(READ,SWAP_DEV,(nr),(buffer))。
    // 参见 linux/mm.h 文件第 11 行。这里把交换设备上的页面 0 读到 swap_bitmap 页面中。
    // 该页面是交换区管理页面。其中第 4086 字节开始处含有 10 个字符的交换设备特征字
    // 符串“SWAP-SPACE”。若没有找到该特征字符串，则说明不是一个有效的交换设备。
    // 于是显示信息，释放刚申请的物理页面并退出函数。否则将特征字符串字节清零。
225         read_swap_page(0,swap_bitmap);
226         if (strncmp("SWAP-SPACE",swap_bitmap+4086,10)) {
227                 printk("Unable to find swap-space signature\n\r");
228                 free_page((long) swap_bitmap);
229                 swap_bitmap = NULL;
230                 return;
231         }
232         memset(swap_bitmap+4086,0,10);
    // 然后检查读入的交换位映射图。应该 32768 个比特位全为 0，若位图中有置位的比特位 0，
    // 则表示位图有问题，于是显示出错信息、释放位图占用的页面并退出函数。为了加快检查
    // 速度，这里首先仅挑选查看位图中位 0 和最后一个交换页面对应的比特位，即 swap_size
    // 交换页面对应的比特位，以及随后到 SWAP_BITS（32768）比特位。
233         for (i = 0 ; i < SWAP_BITS ; i++) {
234                 if (i == 1)
235                         i = swap_size;
236                 if (bit(swap_bitmap,i)) {
237                         printk("Bad swap-space bit-map\n\r");
238                         free_page((long) swap_bitmap);
239                         swap_bitmap = NULL;
240                         return;
241                 }
242         }
    // 然后再仔细地检测位 1 到位 swap_size 所有比特位是否为 0。若有不是 0 的比特位存在，
    // 则表示位图有问题，于是释放位图占用的页面并退出函数。否则显示交换设备工作正常
    // 以及交换页面数和交换空间总字节数。
243         j = 0;
244         for (i = 1 ; i < swap_size ; i++)
245                 if (bit(swap_bitmap,i))
246                         j++;
247         if (!j) {
248                 free_page((long) swap_bitmap);
249                 swap_bitmap = NULL;
250                 return;
```

```
251             }
252             printk("Swap device ok: %d pages (%d bytes) swap-space\n\r",j,j*4096);
253 }
254
```

第14章 内核包含程序

14.1 程序 14-1 linux/include/a.out.h

```
 1 #ifndef _A_OUT_H
 2 #define _A_OUT_H
 3
 4 #define __GNU_EXEC_MACROS__
 5
   // 第 6--108 行是该文件第 1 部分。定义目标文件执行结构以及相关操作的宏定义。
   // 目标文件头结构。参见程序后的详细说明。
   // ============================
   // unsigned long a_magic        // 执行文件魔数。使用 N_MAGIC 等宏访问。
   // unsigned a_text              // 代码长度，字节数。
   // unsigned a_data              // 数据长度，字节数。
   // unsigned a_bss               // 文件中的未初始化数据区长度，字节数。
   // unsigned a_syms              // 文件中的符号表长度，字节数。
   // unsigned a_entry             // 执行开始地址。
   // unsigned a_trsize            // 代码重定位信息长度，字节数。
   // unsigned a_drsize            // 数据重定位信息长度，字节数。
   // ----------------------------
 6 struct exec {
 7   unsigned long a_magic;        /* Use macros N_MAGIC, etc for access */
 8   unsigned a_text;              /* length of text, in bytes */
 9   unsigned a_data;              /* length of data, in bytes */
10   unsigned a_bss;               /* length of uninitialized data area for file, in bytes */
11   unsigned a_syms;              /* length of symbol table data in file, in bytes */
12   unsigned a_entry;             /* start address */
13   unsigned a_trsize;            /* length of relocation info for text, in bytes */
14   unsigned a_drsize;            /* length of relocation info for data, in bytes */
15 };
16
   // 用于取上述 exec 结构中的魔数。
17 #ifndef N_MAGIC
18 #define N_MAGIC(exec) ((exec).a_magic)
19 #endif
20
21 #ifndef OMAGIC
22 /* Code indicating object file or impure executable.  */
   /* 指明为目标文件或者不纯的可执行文件的代号 */
   // 历史上最早在 PDP-11 计算机上，魔数（幻数）是八进制数 0407（0x107）。它位于执行程序
   // 头结构的开始处。原本是 PDP-11 的一条跳转指令，表示跳转到随后 7 个字后的代码开始处。
   // 这样加载程序（loader）就可以在把执行文件放入内存后直接跳转到指令开始处运行。 现在
   // 已没有程序使用这种方法，但这个八进制数却作为识别文件类型的标志（魔数）保留了下来。
   // OMAGIC 可以认为是 Old Magic 的意思。
23 #define OMAGIC 0407
24 /* Code indicating pure executable.  */
   /* 指明为纯可执行文件的代号 */       // New Magic，1975 年以后开始使用。涉及虚存机制。
25 #define NMAGIC 0410                 // 0410 == 0x108
26 /* Code indicating demand-paged executable.  */
```

```
   /* 指明为需求分页处理的可执行文件 */ // 其头结构占用文件开始处 1K 空间。
27 #define ZMAGIC 0413                    // 0413 == 0x10b
28 #endif /* not OMAGIC */
29 // 另外还有一个 QMAGIC，是为了节约磁盘容量，把盘上执行文件的头结构与代码紧凑存放。
   // 下面宏用于判断魔数字段的正确性。如果魔数不能被识别，则返回真。
30 #ifndef N_BADMAG
31 #define N_BADMAG(x)                                          \
32  (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC               \
33   && N_MAGIC(x) != ZMAGIC)
34 #endif
35
36 #define _N_BADMAG(x)                                         \
37  (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC               \
38   && N_MAGIC(x) != ZMAGIC)
39
   // 目标文件头结构末端到 1024 字节之间的长度。
40 #define _N_HDROFF(x) (SEGMENT_SIZE - sizeof (struct exec))
41
   // 下面宏用于操作目标文件的内容，包括.o 模块文件和可执行文件。

   // 代码部分起始偏移值。
   // 如果文件是 ZMAGIC 类型的，即是执行文件，那么代码部分是从执行文件的 1024 字节偏移处
   // 开始；否则执行代码部分紧随执行头结构末端（32 字节）开始，即文件是模块文件（OMAGIC
   // 类型）。
42 #ifndef N_TXTOFF
43 #define N_TXTOFF(x) \
44  (N_MAGIC(x) == ZMAGIC ? _N_HDROFF((x)) + sizeof (struct exec) : sizeof (struct exec))
45 #endif
46
   // 数据部分起始偏移值。从代码部分末端开始。
47 #ifndef N_DATOFF
48 #define N_DATOFF(x) (N_TXTOFF(x) + (x).a_text)
49 #endif
50
   // 代码重定位信息偏移值。从数据部分末端开始。
51 #ifndef N_TRELOFF
52 #define N_TRELOFF(x) (N_DATOFF(x) + (x).a_data)
53 #endif
54
   // 数据重定位信息偏移值。从代码重定位信息末端开始。
55 #ifndef N_DRELOFF
56 #define N_DRELOFF(x) (N_TRELOFF(x) + (x).a_trsize)
57 #endif
58
   // 符号表偏移值。从上面数据段重定位表末端开始。
59 #ifndef N_SYMOFF
60 #define N_SYMOFF(x) (N_DRELOFF(x) + (x).a_drsize)
61 #endif
62
   // 字符串信息偏移值。在符号表之后。
63 #ifndef N_STROFF
64 #define N_STROFF(x) (N_SYMOFF(x) + (x).a_syms)
65 #endif
```

```
66
   // 下面对可执行文件被加载到内存（逻辑空间）中的位置情况进行操作。
67 /* Address of text segment in memory after it is loaded.  */
   /* 代码段加载后在内存中的地址 */
68 #ifndef N_TXTADDR
69 #define N_TXTADDR(x) 0                          // 可见，代码段从地址 0 开始执行。
70 #endif
71
72 /* Address of data segment in memory after it is loaded.
73    Note that it is up to you to define SEGMENT_SIZE
74    on machines not listed here.  */
   /* 数据段加载后在内存中的地址。
      注意，对于下面没有列出名称的机器，需要你自己来定义
      对应的 SEGMENT_SIZE */
75 #if defined(vax) || defined(hp300) || defined(pyr)
76 #define SEGMENT_SIZE PAGE_SIZE
77 #endif
78 #ifdef  hp300
79 #define PAGE_SIZE       4096
80 #endif
81 #ifdef  sony
82 #define SEGMENT_SIZE    0x2000
83 #endif  /* Sony.  */
84 #ifdef is68k
85 #define SEGMENT_SIZE 0x20000
86 #endif
87 #if defined(m68k) && defined(PORTAR)
88 #define PAGE_SIZE 0x400
89 #define SEGMENT_SIZE PAGE_SIZE
90 #endif
91
   // 这里，Linux 0.12 内核把内存页定义为 4KB，段大小定义为 1KB。因此没有使用上面的定义。
92 #define PAGE_SIZE 4096
93 #define SEGMENT_SIZE 1024
94
   // 以段为界的大小（进位方式）。
95 #define  _N_SEGMENT_ROUND(x) (((x) + SEGMENT_SIZE - 1) & ~(SEGMENT_SIZE - 1))
96
   // 代码段尾地址。
97 #define _N_TXTENDADDR(x) (N_TXTADDR(x)+(x).a_text)
98
   // 数据段开始地址。
   // 如果文件是 OMAGIC 类型的，那么数据段就直接紧随代码段后面。否则的话数据段地址从代码
   // 段后面段边界开始（1KB 边界对齐）。例如 ZMAGIC 类型的文件。
99 #ifndef N_DATADDR
100 #define N_DATADDR(x) \
101     (N_MAGIC(x)==OMAGIC? (_N_TXTENDADDR(x)) \
102      : (_N_SEGMENT_ROUND (_N_TXTENDADDR(x))))
103 #endif
104
105 /* Address of bss segment in memory after it is loaded.  */
   /* bss 段加载到内存以后的地址 */
   // 未初始化数据段 bbs 位于数据段后面，紧跟数据段。
```

```
106 #ifndef N_BSSADDR
107 #define N_BSSADDR(x) (N_DATADDR(x) + (x).a_data)
108 #endif
109
    // 第 110—185 行是第 2 部分。对目标文件中的符号表项和相关操作宏进行定义和说明。
    // a.out 目标文件中符号表项结构（符号表记录结构）。参见程序后的详细说明。
110 #ifndef N_NLIST_DECLARED
111 struct nlist {
112   union {
113     char *n_name;
114     struct nlist *n_next;
115     long n_strx;
116   } n_un;
117   unsigned char n_type;         // 该字节分成 3 个字段，146--154 行是相应字段的屏蔽码。
118   char n_other;
119   short n_desc;
120   unsigned long n_value;
121 };
122 #endif
123
    // 下面定义 nlist 结构中 n_type 字段值的常量符号。
124 #ifndef N_UNDF
125 #define N_UNDF 0
126 #endif
127 #ifndef N_ABS
128 #define N_ABS 2
129 #endif
130 #ifndef N_TEXT
131 #define N_TEXT 4
132 #endif
133 #ifndef N_DATA
134 #define N_DATA 6
135 #endif
136 #ifndef N_BSS
137 #define N_BSS 8
138 #endif
139 #ifndef N_COMM
140 #define N_COMM 18
141 #endif
142 #ifndef N_FN
143 #define N_FN 15
144 #endif
145
    // 以下 3 个常量定义是 nlist 结构中 n_type 字段的屏蔽码（八进程表示）。
146 #ifndef N_EXT
147 #define N_EXT 1                 // 0x01（0b0000,0001）符号是否是外部的（全局的）。
148 #endif
149 #ifndef N_TYPE
150 #define N_TYPE 036              // 0x1e（0b0001,1110）符号的类型位。
151 #endif
152 #ifndef N_STAB                  // STAB -- 符号表类型（Symbol table types）。
153 #define N_STAB 0340             // 0xe0（0b1110,0000）这几个比特用于符号调试器。
154 #endif
```

```
155
156 /* The following type indicates the definition of a symbol as being
157    an indirect reference to another symbol.  The other symbol
158    appears as an undefined reference, immediately following this symbol.
159
160    Indirection is asymmetrical.  The other symbol's value will be used
161    to satisfy requests for the indirect symbol, but not vice versa.
162    If the other symbol does not have a definition, libraries will
163    be searched to find a definition.  */
    /* 下面的类型指明对一个符号的定义是作为对另一个符号的间接引用。紧接该
     * 符号的其他的符号呈现为未定义的引用。
     *
     * 这种间接引用是不对称的。另一个符号的值将被用于满足间接符号的要求，
     * 但反之则不然。如果另一个符号没有定义，则将搜索库来寻找一个定义 */
164 #define N_INDR 0xa
165
166 /* The following symbols refer to set elements.
167    All the N_SET[ATDB] symbols with the same name form one set.
168    Space is allocated for the set in the text section, and each set
169    element's value is stored into one word of the space.
170    The first word of the space is the length of the set (number of elements).
171
172    The address of the set is made into an N_SETV symbol
173    whose name is the same as the name of the set.
174    This symbol acts like a N_DATA global symbol
175    in that it can satisfy undefined external references.  */
    /* 下面的符号与集合元素有关。所有具有相同名称 N_SET[ATDB]的符号
       形成一个集合。在代码部分中已为集合分配了空间，并且每个集合元素
       的值存放在一个字（word）的空间中。空间的第一个字存有集合的长度（集合元素数目）。

       集合的地址被放入一个 N_SETV 符号中，它的名称与集合同名。
       在满足未定义的外部引用方面，该符号的行为象一个 N_DATA 全局符号。*/
176
177 /* These appear as input to LD, in a .o file.  */
    /* 以下这些符号在 .o 文件中是作为链接程序 LD 的输入。*/
178 #define N_SETA  0x14        /* Absolute set element symbol */  /* 绝对集合元素符号 */
179 #define N_SETT  0x16        /* Text set element symbol */      /* 代码集合元素符号 */
180 #define N_SETD  0x18        /* Data set element symbol */      /* 数据集合元素符号 */
181 #define N_SETB  0x1A        /* Bss set element symbol */       /* Bss 集合元素符号 */
182
183 /* This is output from LD.  */
    /* 下面是 LD 的输出。*/
184 #define N_SETV  0x1C        /* Pointer to set vector in data area.  */
                                /* 指向数据区中集合向量。*/
185
186 #ifndef N_RELOCATION_INFO_DECLARED
187
188 /* This structure describes a single relocation to be performed.
189    The text-relocation section of the file is a vector of these structures,
190    all of which apply to the text section.
191    Likewise, the data-relocation section applies to the data section.  */
    /* 下面结构描述单个重定位操作的执行。
       文件的代码重定位部分是这些结构的一个数组，所有这些适用于代码部分。
```

```
      类似地，数据重定位部分用于数据部分。*/
192
    // a.out 目标文件中代码和数据重定位信息结构。
193 struct relocation_info
194 {
195   /* Address (within segment) to be relocated.  */
      /* 段内需要重定位的地址。*/
196   int r_address;
197   /* The meaning of r_symbolnum depends on r_extern.  */
      /* r_symbolnum 的含义与 r_extern 有关。*/
198   unsigned int r_symbolnum:24;
199   /* Nonzero means value is a pc-relative offset
200      and it should be relocated for changes in its own address
201      as well as for changes in the symbol or section specified.  */
      /* 非零意味着值是一个 pc 相关的偏移值，因而在其自己地址空间
         以及符号或指定的节改变时，需要被重定位 */
202   unsigned int r_pcrel:1;
203   /* Length (as exponent of 2) of the field to be relocated.
204      Thus, a value of 2 indicates 1<<2 bytes.  */
      /* 需要被重定位的字段长度（是 2 的次方）。
         因此，若值是 2 则表示 1<<2 字节数。*/
205   unsigned int r_length:2;
206   /* 1 => relocate with value of symbol.
207           r_symbolnum is the index of the symbol
208           in file's the symbol table.
209      0 => relocate with the address of a segment.
210           r_symbolnum is N_TEXT, N_DATA, N_BSS or N_ABS
211           (the N_EXT bit may be set also, but signifies nothing).  */
      /* 1 => 以符号的值重定位。
              r_symbolnum 是文件符号表中符号的索引。
         0 => 以段的地址进行重定位。
              r_symbolnum 是 N_TEXT、N_DATA、N_BSS 或 N_ABS
              (N_EXT 比特位也可以被设置，但是毫无意义)。*/
212   unsigned int r_extern:1;
213   /* Four bits that aren't used, but when writing an object file
214      it is desirable to clear them.  */
      /* 没有使用的 4 个比特位，但是当进行写一个目标文件时
         最好将它们复位掉。*/
215   unsigned int r_pad:4;
216 };
217 #endif /* no N_RELOCATION_INFO_DECLARED.  */
218
219
220 #endif /* __A_OUT_GNU_H__ */
221
```

14.2 程序 14-2 linux/include/const.h

```
 1 #ifndef _CONST_H
 2 #define _CONST_H
 3
 4 #define BUFFER_END 0x200000          // 定义缓冲使用内存的末端（代码中没有使用该常量）。
 5
   // i 节点数据结构中 i_mode 字段的各标志位。
 6 #define I_TYPE          0170000      // 指明 i 节点类型（类型屏蔽码）。
 7 #define I_DIRECTORY     0040000      // 是目录文件。
 8 #define I_REGULAR       0100000      // 是常规文件，不是目录文件或特殊文件。
 9 #define I_BLOCK_SPECIAL 0060000      // 是块设备特殊文件。
10 #define I_CHAR_SPECIAL  0020000      // 是字符设备特殊文件。
11 #define I_NAMED_PIPE    0010000      // 是命名管道节点。
12 #define I_SET_UID_BIT   0004000      // 在执行时设置有效用户 ID 类型。
13 #define I_SET_GID_BIT   0002000      // 在执行时设置有效组 ID 类型。
14
15 #endif
16
```

14.3 程序 14-3 linux/include/ctype.h

```
 1 #ifndef _CTYPE_H
 2 #define _CTYPE_H
 3
 4 #define _U      0x01    /* upper */          // 该比特位用于大写字符[A-Z]。
 5 #define _L      0x02    /* lower */          // 该比特位用于小写字符[a-z]。
 6 #define _D      0x04    /* digit */          // 该比特位用于数字[0-9]。
 7 #define _C      0x08    /* cntrl */          // 该比特位用于控制字符。
 8 #define _P      0x10    /* punct */          // 该比特位用于标点字符。
 9 #define _S      0x20    /* white space (space/lf/tab) */ // 空白字符，如空格、\t、\n 等。
10 #define _X      0x40    /* hex digit */          // 该比特位用于十六进制数字。
11 #define _SP     0x80    /* hard space (0x20) */  // 该比特位用于空格字符（0x20）。
12
13 extern unsigned char _ctype[];     // 字符特性数组（表），定义各个字符对应上面的属性。
14 extern char _ctmp;                 // 一个临时字符变量（在定义 lib/ctype.c 中）。
15
   // 下面是一些确定字符类型的宏。
16 #define isalnum(c) ((_ctype+1)[c]&(_U|_L|_D))     // 是字符或数字[A-Z]、[a-z]或[0-9]。
17 #define isalpha(c) ((_ctype+1)[c]&(_U|_L))        // 是字符。
18 #define iscntrl(c) ((_ctype+1)[c]&(_C))           // 是控制字符。
19 #define isdigit(c) ((_ctype+1)[c]&(_D))           // 是数字。
20 #define isgraph(c) ((_ctype+1)[c]&(_P|_U|_L|_D))  // 是图形字符。
21 #define islower(c) ((_ctype+1)[c]&(_L))           // 是小写字符。
22 #define isprint(c) ((_ctype+1)[c]&(_P|_U|_L|_D|_SP))  // 是可打印字符。
23 #define ispunct(c) ((_ctype+1)[c]&(_P))           // 是标点符号。
24 #define isspace(c) ((_ctype+1)[c]&(_S))           // 是空白字符如空格,\f,\n,\r,\t,\v。
25 #define isupper(c) ((_ctype+1)[c]&(_U))           // 是大写字符。
26 #define isxdigit(c) ((_ctype+1)[c]&(_D|_X))       // 是十六进制数字。
27
   // 在下面两个定义中，宏参数前使用了前缀（unsigned），因此 c 应该加括号，即表示成 (c)。
   // 因为在程序中 c 可能是一个复杂的表达式。例如，如果参数是 a + b，若不加括号，则在宏定
   // 义中变成了：(unsigned) a + b。这显然不对。加了括号就能正确表示成(unsigned)(a + b)。
28 #define isascii(c) (((unsigned) c)<=0x7f)           // 是 ASCII 字符。
29 #define toascii(c) (((unsigned) c)&0x7f)            // 转换成 ASCII 字符。
30
   // 以下两个宏定义中使用一个临时变量_ctmp 的原因是：在宏定义中，宏的参数只能被使用一次。
   // 但对于多线程来说这是不安全的，因为两个或多个线程可能在同一时刻使用这个公共临时变量。
   // 因此从 Linux 2.2.x 版本开始更改为使用两个函数来取代这两个宏定义。
31 #define tolower(c) (_ctmp=c,isupper(_ctmp)?_ctmp-('A'-'a'):_ctmp)  // 转换成小写字符。
32 #define toupper(c) (_ctmp=c,islower(_ctmp)?_ctmp-('a'-'A'):_ctmp)  // 转换成大写字符。
33
34 #endif
35
```

14.4 程序 14-4　linux/include/errno.h

```
 1 #ifndef _ERRNO_H
 2 #define _ERRNO_H
 3
 4 /*
 5  * ok, as I hadn't got any other source of information about
 6  * possible error numbers, I was forced to use the same numbers
 7  * as minix.
 8  * Hopefully these are posix or something. I wouldn't know (and posix
 9  * isn't telling me - they want $$$ for their f***ing standard).
10  *
11  * We don't use the _SIGN cludge of minix, so kernel returns must
12  * see to the sign by themselves.
13  *
14  * NOTE! Remember to change strerror() if you change this file!
15  */
   /*
    * ok，由于我没有得到任何其他有关出错号的资料，我只能使用与 minix 系统
    * 相同的出错号了。
    * 希望这些是 POSIX 兼容的或者在一定程度上是这样的，我不知道（而且 POSIX
    * 没有告诉我 - 要获得他们的混蛋标准需要出钱）。
    *
    * 我们没有使用 minix 那样的_SIGN 簇，所以内核的返回值必须自己辨别正负号。
    *
    * 注意！如果你改变该文件的话，记着也要修改 strerror()函数。
    */
16
   // 系统调用以及很多库函数返回一个特殊的值以表示操作失败或出错。这个值通常选择-1 或者
   // 其他一些特定的值来表示。但是这个返回值仅说明错误发生了。 如果需要知道出错的类型，
   // 就需要查看表示系统出错号的变量 errno。该变量即在 errno.h 文件中声明。在程序开始执
   // 行时该变量值被初始化为 0。
17 extern int errno;
18
   // 在出错时，系统调用会把出错号放在变量 errno 中（负值），然后返回-1。因此程序若需要知
   // 道具体错误号，就需要查看 errno 的值。
19 #define ERROR           99              // 一般错误。
20 #define EPERM            1              // 操作没有许可。
21 #define ENOENT           2              // 文件或目录不存在。
22 #define ESRCH            3              // 指定的进程不存在。
23 #define EINTR            4              // 中断的系统调用。
24 #define EIO              5              // 输入/输出错。
25 #define ENXIO            6              // 指定设备或地址不存在。
26 #define E2BIG            7              // 参数列表太长。
27 #define ENOEXEC          8              // 执行程序格式错误。
28 #define EBADF            9              // 文件句柄(描述符)错误。
29 #define ECHILD          10              // 子进程不存在。
30 #define EAGAIN          11              // 资源暂时不可用。
31 #define ENOMEM          12              // 内存不足。
32 #define EACCES          13              // 没有许可权限。
33 #define EFAULT          14              // 地址错。
```

```
34 #define ENOTBLK         15              // 不是块设备文件。
35 #define EBUSY           16              // 资源正忙。
36 #define EEXIST          17              // 文件已存在。
37 #define EXDEV           18              // 非法连接。
38 #define ENODEV          19              // 设备不存在。
39 #define ENOTDIR         20              // 不是目录文件。
40 #define EISDIR          21              // 是目录文件。
41 #define EINVAL          22              // 参数无效。
42 #define ENFILE          23              // 系统打开文件数太多。
43 #define EMFILE          24              // 打开文件数太多。
44 #define ENOTTY          25              // 不恰当的 IO 控制操作(没有 tty 终端)。
45 #define ETXTBSY         26              // 不再使用。
46 #define EFBIG           27              // 文件太大。
47 #define ENOSPC          28              // 设备已满（设备已经没有空间）。
48 #define ESPIPE          29              // 无效的文件指针重定位。
49 #define EROFS           30              // 文件系统只读。
50 #define EMLINK          31              // 连接太多。
51 #define EPIPE           32              // 管道错。
52 #define EDOM            33              // 域(domain)出错。
53 #define ERANGE          34              // 结果太大。
54 #define EDEADLK         35              // 避免资源死锁。
55 #define ENAMETOOLONG    36              // 文件名太长。
56 #define ENOLCK          37              // 没有锁定可用。
57 #define ENOSYS          38              // 功能还没有实现。
58 #define ENOTEMPTY       39              // 目录不空。
59
60 /* Should never be seen by user programs */
   /* 用户程序不应该见到下面这两中错误号 */
61 #define ERESTARTSYS     512             // 重新执行系统调用。
62 #define ERESTARTNOINTR  513             // 重新执行系统调用，无中断。
63
64 #endif
65
```

14.5 程序 14-5　linux/include/fcntl.h

```
 1 #ifndef _FCNTL_H
 2 #define _FCNTL_H
 3
 4 #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
 5
 6 /* open/fcntl - NOCTTY, NDELAY isn't implemented yet */
   /* open/fcntl - NOCTTY 和 NDELAY 现在还没有实现 */
 7 #define O_ACCMODE       00003                   // 文件访问模式屏蔽码。
   // 打开文件 open()和文件控制函数 fcntl()使用的文件访问模式。同时只能使用三者之一。
 8 #define O_RDONLY          00                    // 以只读方式打开文件。
 9 #define O_WRONLY          01                    // 以只写方式打开文件。
10 #define O_RDWR            02                    // 以读写方式打开文件。
   // 下面是文件创建和操作标志，用于 open()。可与上面访问模式用'位或'的方式一起使用。
11 #define O_CREAT         00100   /* not fcntl */ // 如果文件不存在就创建。fcntl 函数不用。
12 #define O_EXCL          00200   /* not fcntl */ // 独占使用文件标志。
13 #define O_NOCTTY        00400   /* not fcntl */ // 不分配控制终端。
14 #define O_TRUNC         01000   /* not fcntl */ // 若文件已存在且是写操作，则长度截为 0。
15 #define O_APPEND        02000                   // 以添加方式打开，文件指针置为文件尾。
16 #define O_NONBLOCK      04000   /* not fcntl */ // 非阻塞方式打开和操作文件。
17 #define O_NDELAY        O_NONBLOCK              // 非阻塞方式打开和操作文件。
18
19 /* Defines for fcntl-commands. Note that currently
20  * locking isn't supported, and other things aren't really
21  * tested.
22  */
   /* 下面定义了 fcntl 的命令。注意目前锁定命令还没有支持，而其他
    * 命令实际上还没有测试过。
    */
   // 文件句柄(描述符)操作函数 fcntl()的命令（cmd）。
23 #define F_DUPFD         0       /* dup */         // 拷贝文件句柄为最小数值的句柄。
24 #define F_GETFD         1      /* get f_flags */  // 取句柄标志。仅 1 个标志 FD_CLOEXEC。
25 #define F_SETFD         2       /* set f_flags */ // 设置文件句柄标志。
26 #define F_GETFL         3       /* more flags (cloexec) */ // 取文件状态标志和访问模式。
27 #define F_SETFL         4                                  // 设置文件状态标志和访问模式。
   // 下面是文件锁定命令。fcntl()的第三个参数 lock 是指向 flock 结构的指针。
28 #define F_GETLK         5       /* not implemented */ // 返回阻止锁定的 flock 结构。
29 #define F_SETLK         6       // 设置(F_RDLCK 或 F_WRLCK)或清除(F_UNLCK)锁定。
30 #define F_SETLKW        7       // 等待设置或清除锁定。
31
32 /* for F_[GET|SET]FL */
   /* 用于 F_GETFL 或 F_SETFL */
   // 在执行 exec()簇函数时需要关闭的文件句柄。(执行时关闭 - Close On EXECution)
33 #define FD_CLOEXEC      1       /* actually anything with low bit set goes */
                                   /* 实际上只要低位为 1 即可 */
34
35 /* Ok, these are locking features, and aren't implemented at any
36  * level. POSIX wants them.
37  */
   /* OK，以下是锁定类型，任何函数中都还没有实现。POSIX 标准要求这些类型。
```

```
 */
38 #define F_RDLCK         0       // 共享或读文件锁定。
39 #define F_WRLCK         1       // 独占或写文件锁定。
40 #define F_UNLCK         2       // 文件解锁。
41
42 /* Once again - not implemented, but ... */
   /* 同样 - 也还没有实现，但是... */
   // 文件锁定操作数据结构。描述了受影响文件段的类型(l_type)、开始偏移(l_whence)、
   // 相对偏移(l_start)、锁定长度(l_len)和实施锁定的进程 id。
43 struct flock {
44         short l_type;           // 锁定类型（F_RDLCK，F_WRLCK，F_UNLCK）。
45         short l_whence;         // 开始偏移(SEEK_SET，SEEK_CUR 或 SEEK_END)。
46         off_t l_start;          // 阻塞锁定的开始处。相对偏移（字节数）。
47         off_t l_len;            // 阻塞锁定的大小；如果是 0 则为到文件末尾。
48         pid_t l_pid;            // 加锁的进程 id。
49 };
50
   // 以下是使用上述标志或命令的函数原型。
   // 创建新文件或重写一个已存在文件。
   // 参数 filename 是欲创建文件的文件名，mode 是创建文件的属性（见 include/sys/stat.h）。
51 extern int creat(const char * filename,mode_t mode);
   // 文件句柄操作，会影响文件的打开。
   // 参数 fildes 是文件句柄，cmd 是操作命令，见上面 23--30 行。该函数可有以下几种形式：
   // int fcntl(int fildes, int cmd);
   // int fcntl(int fildes, int cmd, long arg);
   // int fcntl(int fildes, int cmd, struct flock *lock);
52 extern int fcntl(int fildes,int cmd, ...);
   // 打开文件。在文件与文件句柄之间建立联系。
   // 参数 filename 是欲打开文件的文件名，flags 是上面 7-17 行上的标志的组合。
53 extern int open(const char * filename, int flags, ...);
54
55 #endif
56
```

14.6 程序 14-6　linux/include/signal.h

```
 1 #ifndef _SIGNAL_H
 2 #define _SIGNAL_H
 3
 4 #include <sys/types.h>                  // 类型头文件。定义了基本的系统数据类型。
 5
 6 typedef int sig_atomic_t;               // 定义信号原子操作类型。
 7 typedef unsigned int sigset_t;          /* 32 bits */  // 定义信号集类型。
 8
 9 #define _NSIG             32            // 定义信号种类 -- 32 种。
10 #define NSIG              _NSIG          // NSIG = _NSIG
11
   // 以下这些是 Linux 0.12 内核中定义的信号。其中包括了 POSIX.1 要求的所有 20 个信号。
12 #define SIGHUP            1              // Hang Up     -- 挂断控制终端或进程。
13 #define SIGINT            2              // Interrupt   -- 来自键盘的中断。
14 #define SIGQUIT           3              // Quit        -- 来自键盘的退出。
15 #define SIGILL            4              // Illeagle    -- 非法指令。
16 #define SIGTRAP           5              // Trap        -- 跟踪断点。
17 #define SIGABRT           6              // Abort       -- 异常结束。
18 #define SIGIOT            6              // IO Trap     -- 同上。
19 #define SIGUNUSED         7              // Unused      -- 没有使用。
20 #define SIGFPE            8              // FPE         -- 协处理器出错。
21 #define SIGKILL           9              // Kill        -- 强迫进程终止。
22 #define SIGUSR1           10             // User1       -- 用户信号 1，进程可使用。
23 #define SIGSEGV           11             // Segment Violation -- 无效内存引用。
24 #define SIGUSR2           12             // User2       -- 用户信号 2，进程可使用。
25 #define SIGPIPE           13             // Pipe        -- 管道写出错，无读者。
26 #define SIGALRM           14             // Alarm       -- 实时定时器报警。
27 #define SIGTERM           15             // Terminate   -- 进程终止。
28 #define SIGSTKFLT         16             // Stack Fault -- 栈出错（协处理器）。
29 #define SIGCHLD           17             // Child       -- 子进程停止或被终止。
30 #define SIGCONT           18             // Continue    -- 恢复进程继续执行。
31 #define SIGSTOP           19             // Stop        -- 停止进程的执行。
32 #define SIGTSTP           20             // TTY Stop    -- tty 发出停止进程，可忽略。
33 #define SIGTTIN           21             // TTY In      -- 后台进程请求输入。
34 #define SIGTTOU           22             // TTY Out     -- 后台进程请求输出。
35
36 /* Ok, I haven't implemented sigactions, but trying to keep headers POSIX */
   /* OK，我还没有实现 sigactions 的编制，但在头文件中仍希望遵守 POSIX 标准 */
   // 上面原注释已经过时，因为在 0.12 内核中已经实现了 sigaction()。下面是 sigaction 结构
   // sa_flags 标志字段可取的符号常数值。
37 #define SA_NOCLDSTOP      1              // 当子进程处于停止状态，就不对 SIGCHLD 处理。
38 #define SA_INTERRUPT      0x20000000     // 系统调用被信号中断后不重新启动系统调用。
39 #define SA_NOMASK         0x40000000     // 不阻止在指定的信号处理程序中再收到该信号。
40 #define SA_ONESHOT        0x80000000     // 信号句柄一旦被调用过就恢复到默认处理句柄。
41
   // 以下常量用于 sigprocmask(how, )-- 改变阻塞信号集(屏蔽码)。用于改变该函数的行为。
42 #define SIG_BLOCK         0  /* for blocking signals */   // 在阻塞信号集中加上给定信号。
43 #define SIG_UNBLOCK       1  /* for unblocking signals */ // 从阻塞信号集中删除指定信号。
44 #define SIG_SETMASK       2  /* for setting the signal mask */   // 设置阻塞信号集。
```

```
45
   // 以下两个常数符号都表示指向无返回值的函数指针，且都有一个 int 整型参数。这两个指针
   // 值是逻辑上讲实际上不可能出现的函数地址值。可作为下面 signal 函数的第二个参数。用
   // 于告知内核，让内核处理信号或忽略对信号的处理。使用方法参见 kernel/signal.c 程序，
   // 第 94--96 行。
46 #define SIG_DFL         ((void (*)(int))0)      /* default signal handling */
                                                   // 默认信号处理程序（信号句柄）。
47 #define SIG_IGN         ((void (*)(int))1)      /* ignore signal */
                                                   // 忽略信号的处理程序。
48 #define SIG_ERR         ((void (*)(int))-1)     /* error return from signal */
                                                   // 信号处理返回错误。
49
   // 下面定义初始操作设置 sigaction 结构信号屏蔽码的宏。
50 #ifdef notdef
51 #define sigemptyset(mask) ((*(mask) = 0), 1)    // 将 mask 清零。
52 #define sigfillset(mask) ((*(mask) = ~0), 1)    // 将 mask 所有比特位置位。
53 #endif
54
   // 下面是 sigaction 的数据结构。
   // sa_handler 是对应某信号指定要采取的行动。可以用上面的 SIG_DFL，或 SIG_IGN 来忽略该
   // 信号，也可以是指向处理该信号函数的一个指针。
   // sa_mask 给出了对信号的屏蔽码，在信号程序执行时将阻塞对这些信号的处理。
   // sa_flags 指定改变信号处理过程的信号集。它是由 37-40 行的位标志定义的。
   // sa_restorer 是恢复函数指针，由函数库 Libc 提供，用于清理用户态堆栈。参见 signal.c。
   // 另外，引起触发信号处理的信号也将被阻塞，除非使用了 SA_NOMASK 标志。
55 struct sigaction {
56         void (*sa_handler)(int);
57         sigset_t sa_mask;
58         int sa_flags;
59         void (*sa_restorer)(void);
60 };
61
   // 下面 signal 函数用于是为信号_sig 安装一新的信号处理程序（信号句柄），与 sigaction()
   // 类似。该函数含有两个参数：指定需要捕获的信号_sig；具有一个参数且无返回值的函数指针
   // _func。该函数返回值也是具有一个 int 参数（最后一个(int)）且无返回值的函数指针，它是
   // 处理该信号的原处理句柄。
62 void (*signal(int _sig, void (*_func)(int)))(int);
   // 下面两函数用于发送信号。kill() 用于向任何进程或进程组发送信号。raise()用于向当前进
   // 程自身发送信号。其作用等价于 kill(getpid(),sig)。参见 kernel/exit.c，60 行。
63 int raise(int sig);
64 int kill(pid_t pid, int sig);
   // 在进程的任务结构中，除有一个以比特位表示当前进程待处理的 32 位信号字段 signal 以外，
   // 还有一个同样以比特位表示的用于屏蔽进程当前阻塞信号集（屏蔽信号集）的字段 blocked，
   // 也是 32 位，每个比特代表一个对应的阻塞信号。修改进程的屏蔽信号集可以阻塞或解除阻塞
   // 所指定的信号。 以下五个函数就是用于操作进程屏蔽信号集，虽然简单实现起来很简单，但
   // 本版本内核中还未实现。
   // 函数 sigaddset() 和 sigdelset() 用于对信号集中的信号进行增、删修改。 sigaddset()用
   // 于向 mask 指向的信号集中增加指定的信号 signo。sigdelset 则反之。函数 sigemptyset()和
   // sigfillset() 用于初始化进程屏蔽信号集。 每个程序在使用信号集前，都需要使用这两个函
   // 数之一对屏蔽信号集进行初始化。 sigemptyset()用于清空屏蔽的所有信号，也即响应所有的
   // 信号。sigfillset()向信号集中置入所有信号，也即屏蔽所有信号。当然 SIGINT 和 SIGSTOP
   // 是不能被屏蔽的。
   // sigismember()用于测试一个指定信号是否在信号集中（1 - 是，0 - 不是，-1 - 出错）。
```

```
65 int sigaddset(sigset_t *mask, int signo);
66 int sigdelset(sigset_t *mask, int signo);
67 int sigemptyset(sigset_t *mask);
68 int sigfillset(sigset_t *mask);
69 int sigismember(sigset_t *mask, int signo); /* 1 - is, 0 - not, -1 error */
   // 对 set 中的信号进行检测，看是否有挂起的信号。在 set 中返回进程中当前被阻塞的信号集。
70 int sigpending(sigset_t *set);
   // 下面函数用于改变进程目前被阻塞的信号集（信号屏蔽码）。若 oldset 不是 NULL，则通过其
   // 返回进程当前屏蔽信号集。若 set 指针不是 NULL，则根据 how（41-43 行）指示修改进程屏蔽
   // 信号集。
71 int sigprocmask(int how, sigset_t *set, sigset_t *oldset);
   // 下面函数用 sigmask 临时替换进程的信号屏蔽码,然后暂停该进程直到收到一个信号。若捕捉
   // 到某一信号并从该信号处理程序中返回，则该函数也返回，并且信号屏蔽码会恢复到调用调用
   // 前的值。
72 int sigsuspend(sigset_t *sigmask);
   // sigaction() 函数用于改变进程在收到指定信号时所采取的行动，即改变信号的处理句柄能。
   // 参见对 kernel/signal.c 程序的说明。
73 int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);
74
75 #endif /* _SIGNAL_H */
76
```

14.7 程序 14-7 linux/include/stdarg.h

```
 1 #ifndef _STDARG_H
 2 #define _STDARG_H
 3
 4 typedef char *va_list;  // 定义 va_list 是一个字符指针类型。
 5
 6 /* Amount of space required in an argument list for an arg of type TYPE.
 7    TYPE may alternatively be an expression whose type is used.  */
   /* 下面给出了类型为 TYPE 的 arg 参数列表所要求的空间容量。
      TYPE 也可以是使用该类型的一个表达式 */
 8
   // 下面这句定义了取整后的 TYPE 类型的字节长度值。是 int 长度(4)的倍数。
 9 #define __va_rounded_size(TYPE)  \
10   (((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))
11
   // 下面这个宏初始化指针 AP，使其指向传给函数的可变参数表的第一个参数。
   // 在第一次调用 va_arg 或 va_end 之前，必须首先调用 va_start 宏。参数 LASTARG 是函数定义
   // 中最右边参数的标识符，即'...'左边的一个标识符。AP 是可变参数表参数指针，LASTARG 是
   // 最后一个指定参数。&(LASTARG) 用于取其地址（即其指针），并且该指针是字符类型。加上
   // LASTARG 的宽度值后 AP 就是可变参数表中第一个参数的指针。该宏没有返回值。
   // 第 17 行上的函数 __builtin_saveregs() 是在 gcc 的库程序 libgcc2.c 中定义的，用于保存
   // 寄存器。 相关说明参见 gcc 手册“Target Description Macros”章中“Implementing the
   // Varargs Macros”小节。
12 #ifndef __sparc__
13 #define va_start(AP, LASTARG)                                           \
14  (AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
15 #else
16 #define va_start(AP, LASTARG)                                           \
17  (__builtin_saveregs (),                                                \
18   AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
19 #endif
20
   // 下面该宏用于被调用函数完成一次正常返回。va_end 可以修改 AP 使其在重新调用
   // va_start 之前不能被使用。va_end 必须在 va_arg 读完所有的参数后再被调用。
21 void va_end (va_list);          /* Defined in gnulib */  /* 在 gnulib 中定义 */
22 #define va_end(AP)
23
   // 下面宏用于扩展表达式使其与下一个被传递参数具有相同的类型和值。
   // 对于缺省值，va_arg 可以用字符、无符号字符和浮点类型。在第一次使用 va_arg 时，它返
   // 回表中的第一个参数，后续的每次调用都将返回表中的下一个参数。这是通过先访问 AP，然
   // 后增加其值以指向下一项来实现的。va_arg 使用 TYPE 来完成访问和定位下一项，每调用一
   // 次 va_arg，它就修改 AP 以指示表中的下一参数。
24 #define va_arg(AP, TYPE)                                \
25  (AP += __va_rounded_size (TYPE),                       \
26   *((TYPE *) (AP - __va_rounded_size (TYPE))))
27
28 #endif /* _STDARG_H */
29
```

14.8 程序 14-8 linux/include/stddef.h

```
 1 #ifndef _STDDEF_H
 2 #define _STDDEF_H
 3
 4 #ifndef _PTRDIFF_T
 5 #define _PTRDIFF_T
 6 typedef long ptrdiff_t;                // 两个指针相减结果的类型。
 7 #endif
 8
 9 #ifndef _SIZE_T
10 #define _SIZE_T
11 typedef unsigned long size_t;          // sizeof 返回的类型。
12 #endif
13
14 #undef NULL
15 #define NULL ((void *)0)               // 空指针。
16
   // 下面定义了一个计算某成员在类型中偏移位置的宏。使用该宏可以确定一个成员（字段）在
   // 包含它的结构类型中从结构开始处算起的字节偏移量。宏的结果是类型为 size_t 的整数常
   // 数表达式。这里是一个技巧用法。((TYPE *)0)是将一个整数 0 类型投射（type cast）成数
   // 据对象指针类型，然后在该结果上进行运算。
17 #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
18
19 #endif
20
```

14.9 程序 14-9 linux/include/string.h

```
 1 #ifndef _STRING_H_
 2 #define _STRING_H_
 3
 4 #ifndef NULL
 5 #define NULL ((void *) 0)
 6 #endif
 7
 8 #ifndef _SIZE_T
 9 #define _SIZE_T
10 typedef unsigned int size_t;
11 #endif
12
13 extern char * strerror(int errno);
14
15 /*
16  * This string-include defines all string functions as inline
17  * functions. Use gcc. It also assumes ds=es=data space, this should be
18  * normal. Most of the string-functions are rather heavily hand-optimized,
19  * see especially strtok,strstr,str[c]spn. They should work, but are not
20  * very easy to understand. Everything is done entirely within the register
21  * set, making the functions fast and clean. String instructions have been
22  * used through-out, making for "slightly" unclear code :-)
23  *
24  *              (C) 1991 Linus Torvalds
25  */
   /*
    * 这个字符串头文件以内嵌函数的形式定义了所有字符串操作函数。使用 gcc 时，同时
    * 假定了 ds=es=数据空间，这应该是常规的。绝大多数字符串函数都是经手工进行大量
    * 优化的，尤其是函数 strtok、strstr、str[c]spn。它们应该能正常工作，但却不是那
    * 么容易理解。所有的操作基本上都是使用寄存器集来完成的，这使得函数即快又整洁。
    * 所有地方都使用了字符串指令，这又使得代码“稍微”难以理解☺
    *
    *              (C) 1991 Linus Torvalds
    */
26
   //// 将一个字符串(src)拷贝到另一个字符串(dest)，直到遇到 NULL 字符后停止。
   // 参数：dest - 目的字符串指针，src - 源字符串指针。
   // %0 - esi(src)，%1 - edi(dest)。
27 extern inline char * strcpy(char * dest,const char *src)
28 {
29 __asm__("cld\n"                         // 清方向位。
30         "1:\tlodsb\n\t"                  // 加载 DS:[esi]处 1 字节➔al，并更新 esi。
31         "stosb\n\t"                      // 存储字节 al➔ES:[edi]，并更新 edi。
32         "testb %%al,%%al\n\t"            // 刚存储的字节是 0?
33         "jne 1b"                         // 不是则向后跳转到标号 1 处，否则结束。
34         ::"S" (src),"D" (dest):"si","di","ax");
35 return dest;                            // 返回目的字符串指针。
36 }
37
```

```
//// 拷贝源字符串 count 个字节到目的字符串。
// 如果源串长度小于 count 个字节，就附加空字符(NULL)到目的字符串。
// 参数：dest - 目的字符串指针，src - 源字符串指针，count - 拷贝字节数。
// %0 - esi(src)，%1 - edi(dest)，%2 - ecx(count)。
extern inline char * strncpy(char * dest,const char *src,int count)
{
__asm__("cld\n"                         // 清方向位。
        "1:\tdecl %2\n\t"               // 寄存器 ecx--（count--）。
        "js 2f\n\t"                     // 如果 count<0 则向前跳转到标号 2，结束。
        "lodsb\n\t"                     // 取 ds:[esi]处 1 字节➔al，并且 esi++。
        "stosb\n\t"                     // 存储该字节➔es:[edi]，并且 edi++。
        "testb %%al,%%al\n\t"           // 该字节是 0?
        "jne 1b\n\t"                    // 不是，则向前跳转到标号 1 处继续拷贝。
        "rep\n\t"                       // 否则，在目的串中存放剩余个数的空字符。
        "stosb\n"
        "2:"
        ::"S" (src),"D" (dest),"c" (count):"si","di","ax","cx");
return dest;                            // 返回目的字符串指针。
}

//// 将源字符串拷贝到目的字符串的末尾处。
// 参数：dest - 目的字符串指针，src - 源字符串指针。
// %0 - esi(src)，%1 - edi(dest)，%2 - eax(0)，%3 - ecx(-1)。
extern inline char * strcat(char * dest,const char * src)
{
__asm__("cld\n\t"                       // 清方向位。
        "repne\n\t"                     // 比较 al 与 es:[edi]字节，并更新 edi++，
        "scasb\n\t"                     // 直到找到目的串中是 0 的字节，此时 edi 已指向后 1 字节。
        "decl %1\n"                     // 让 es:[edi]指向 0 值字节。
        "1:\tlodsb\n\t"                 // 取源字符串字节 ds:[esi]➔al，并 esi++。
        "stosb\n\t"                     // 将该字节存到 es:[edi]，并 edi++。
        "testb %%al,%%al\n\t"           // 该字节是 0?
        "jne 1b"                        // 不是，则向后跳转到标号 1 处继续拷贝，否则结束。
        ::"S" (src),"D" (dest),"a" (0),"c" (0xffffffff):"si","di","ax","cx");
return dest;                            // 返回目的字符串指针。
}

//// 将源字符串的 count 个字节复制到目的字符串的末尾处，最后添一空字符。
// 参数：dest - 目的字符串，src - 源字符串，count - 欲复制的字节数。
// %0 - esi(src)，%1 - edi(dest)，%2 - eax(0)，%3 - ecx(-1)，%4 - (count)。
extern inline char * strncat(char * dest,const char * src,int count)
{
__asm__("cld\n\t"                       // 清方向位。
        "repne\n\t"                     // 比较 al 与 es:[edi]字节，edi++。
        "scasb\n\t"                     // 直到找到目的串的末端 0 值字节。
        "decl %1\n\t"                   // edi 指向该 0 值字节。
        "movl %4,%3\n"                  // 欲复制字节数➔ecx。
        "1:\tdecl %3\n\t"               // ecx--（从 0 开始计数）。
        "js 2f\n\t"                     // ecx <0 ?，是则向前跳转到标号 2 处。
        "lodsb\n\t"                     // 否则取 ds:[esi]处的字节➔al，esi++。
        "stosb\n\t"                     // 存储到 es:[edi]处，edi++。
        "testb %%al,%%al\n\t"           // 该字节值为 0?
        "jne 1b\n"                      // 不是则向后跳转到标号 1 处，继续复制。
```

```
81          "2:\txorl %2,%2\n\t"            // 将 al 清零。
82          "stosb"                         // 存到 es:[edi]处。
83          ::"S" (src),"D" (dest),"a" (0),"c" (0xffffffff),"g" (count)
84          :"si","di","ax","cx");
85  return dest;                            // 返回目的字符串指针。
86  }
87
    //// 将一个字符串与另一个字符串进行比较。
    // 参数：cs - 字符串 1，ct - 字符串 2。
    // %0 - eax(__res)返回值，%1 - edi(cs)字符串 1 指针，%2 - esi(ct)字符串 2 指针。
    // 返回：如果串 1 > 串 2，则返回 1；串 1 = 串 2，则返回 0；串 1 < 串 2，则返回-1。
    // 第 90 行定义了一个局部寄存器变量。该变量将被保存在 eax 寄存器中，以便于高效访问和操作。
    // 这种定义变量的方法主要用于内嵌汇编程序中。详细说明参见 gcc 手册“指定寄存器中的变量”。
88  extern inline int strcmp(const char * cs,const char * ct)
89  {
90  register int __res __asm__("ax");       // __res 是寄存器变量(eax)。
91  __asm__("cld\n"                         // 清方向位。
92          "1:\tlodsb\n\t"                 // 取字符串 2 的字节 ds:[esi]➔al，并且 esi++。
93          "scasb\n\t"                     // al 与字符串 1 的字节 es:[edi]作比较，并且 edi++。
94          "jne 2f\n\t"                    // 如果不相等，则向前跳转到标号 2。
95          "testb %%al,%%al\n\t"           // 该字节是 0 值字节吗（字符串结尾）？
96          "jne 1b\n\t"                    // 不是，则向后跳转到标号 1，继续比较。
97          "xorl %%eax,%%eax\n\t"          // 是，则返回值 eax 清零，
98          "jmp 3f\n"                      // 向前跳转到标号 3，结束。
99          "2:\tmovl $1,%%eax\n\t"         // eax 中置 1。
100         "jl 3f\n\t"                     // 若前面比较中串 2 字符<串 1 字符，则返回正值结束。
101         "negl %%eax\n"                  // 否则 eax = -eax，返回负值，结束。
102         "3:"
103         :"=a" (__res):"D" (cs),"S" (ct):"si","di");
104 return __res;                          // 返回比较结果。
105 }
106
    //// 字符串 1 与字符串 2 的前 count 个字符进行比较。
    // 参数：cs - 字符串 1，ct - 字符串 2，count - 比较的字符数。
    // %0 - eax(__res)返回值，%1 - edi(cs)串 1 指针，%2 - esi(ct)串 2 指针，%3 - ecx(count)。
    // 返回：如果串 1 > 串 2，则返回 1；串 1 = 串 2，则返回 0；串 1 < 串 2，则返回-1。
107 extern inline int strncmp(const char * cs,const char * ct,int count)
108 {
109 register int __res __asm__("ax");       // __res 是寄存器变量(eax)。
110 __asm__("cld\n"                         // 清方向位。
111         "1:\tdecl %3\n\t"               // count--。
112         "js 2f\n\t"                     // 如果 count<0，则向前跳转到标号 2。
113         "lodsb\n\t"                     // 取串 2 的字符 ds:[esi]➔al，并且 esi++。
114         "scasb\n\t"                     // 比较 al 与串 1 的字符 es:[edi]，并且 edi++。
115         "jne 3f\n\t"                    // 如果不相等，则向前跳转到标号 3。
116         "testb %%al,%%al\n\t"           // 该字符是 NULL 字符吗？
117         "jne 1b\n"                      // 不是，则向后跳转到标号 1，继续比较。
118         "2:\txorl %%eax,%%eax\n\t"      // 是 NULL 字符，则 eax 清零（返回值）。
119         "jmp 4f\n"                      // 向前跳转到标号 4，结束。
120         "3:\tmovl $1,%%eax\n\t"         // eax 中置 1。
121         "jl 4f\n\t"                     // 如果前面比较中串 2 字符<串 1 字符，则返回 1 结束。
122         "negl %%eax\n"                  // 否则 eax = -eax，返回负值，结束。
123         "4:"
```

```
124         :"=a" (__res):"D" (cs),"S" (ct),"c" (count):"si","di","cx");
125 return __res;                         // 返回比较结果。
126 }
127
    //// 在字符串中寻找第一个匹配的字符。
    // 参数：s - 字符串，c - 欲寻找的字符。
    // %0 - eax(__res)，%1 - esi(字符串指针 s)，%2 - eax(字符 c)。
    // 返回：返回字符串中第一次出现匹配字符的指针。若没有找到匹配的字符，则返回空指针。
128 extern inline char * strchr(const char * s,char c)
129 {
130 register char * __res __asm__("ax"); // __res 是寄存器变量(eax)。
131 __asm__("cld\n\t"                     // 清方向位。
132         "movb %%al,%%ah\n"            // 将欲比较字符移到 ah。
133         "1:\tlodsb\n\t"               // 取字符串中字符 ds:[esi]➔al，并且 esi++。
134         "cmpb %%ah,%%al\n\t"          // 字符串中字符 al 与指定字符 ah 相比较。
135         "je 2f\n\t"                   // 若相等，则向前跳转到标号 2 处。
136         "testb %%al,%%al\n\t"         // al 中字符是 NULL 字符吗？（字符串结尾？）
137         "jne 1b\n\t"                  // 若不是，则向后跳转到标号 1，继续比较。
138         "movl $1,%1\n"                // 是，则说明没有找到匹配字符，esi 置 1。
139         "2:\tmovl %1,%0\n\t"          // 将指向匹配字符后一个字节处的指针值放入 eax
140         "decl %0"                     // 将指针调整为指向匹配的字符。
141         :"=a" (__res):"S" (s),"0" (c):"si");
142 return __res;                         // 返回指针。
143 }
144
    //// 寻找字符串中指定字符最后一次出现的地方。（反向搜索字符串）
    // 参数：s - 字符串，c - 欲寻找的字符。
    // %0 - edx(__res)，%1 - edx(0)，%2 - esi(字符串指针 s)，%3 - eax(字符 c)。
    // 返回：返回字符串中最后一次出现匹配字符的指针。若没有找到匹配的字符，则返回空指针。
145 extern inline char * strrchr(const char * s,char c)
146 {
147 register char * __res __asm__("dx"); // __res 是寄存器变量(edx)。
148 __asm__("cld\n\t"                     // 清方向位。
149         "movb %%al,%%ah\n"            // 将欲寻找的字符移到 ah。
150         "1:\tlodsb\n\t"               // 取字符串中字符 ds:[esi]➔al，并且 esi++。
151         "cmpb %%ah,%%al\n\t"          // 字符串中字符 al 与指定字符 ah 作比较。
152         "jne 2f\n\t"                  // 若不相等，则向前跳转到标号 2 处。
153         "movl %%esi,%0\n\t"           // 将字符指针保存到 edx 中。
154         "decl %0\n"                   // 指针后退一位，指向字符串中匹配字符处。
155         "2:\ttestb %%al,%%al\n\t"     // 比较的字符是 0 吗（到字符串尾）？
156         "jne 1b"                      // 不是则向后跳转到标号 1 处，继续比较。
157         :"=d" (__res):"0" (0),"S" (s),"a" (c):"ax","si");
158 return __res;                         // 返回指针。
159 }
160
    //// 在字符串 1 中寻找第 1 个字符序列，该字符序列中的任何字符都包含在字符串 2 中。
    // 参数：cs - 字符串 1 指针，ct - 字符串 2 指针。
    // %0 - esi(__res)，%1 - eax(0)，%2 - ecx(-1)，%3 - esi(串 1 指针 cs)，%4 - (串 2 指针 ct)。
    // 返回字符串 1 中包含字符串 2 中任何字符的首个字符序列的长度值。
161 extern inline int strspn(const char * cs, const char * ct)
162 {
163 register char * __res __asm__("si"); // __res 是寄存器变量(esi)。
164 __asm__("cld\n\t"                     // 清方向位。
```

```
165         "movl %4,%%edi\n\t"             // 首先计算串 2 的长度。串 2 指针放入 edi 中。
166         "repne\n\t"                     // 比较 al(0)与串 2 中的字符（es:[edi]），并 edi++。
167         "scasb\n\t"                     // 如果不相等就继续比较(ecx 逐步递减)。
168         "notl %%ecx\n\t"                // ecx 中每位取反。
169         "decl %%ecx\n\t"                // ecx--，得串 2 的长度值。
170         "movl %%ecx,%%edx\n"            // 将串 2 的长度值暂放入 edx 中。
171         "1:\tlodsb\n\t"                 // 取串 1 字符 ds:[esi]➔al，并且 esi++。
172         "testb %%al,%%al\n\t"           // 该字符等于 0 值吗（串 1 结尾）？
173         "je 2f\n\t"                     // 如果是，则向前跳转到标号 2 处。
174         "movl %4,%%edi\n\t"             // 取串 2 头指针放入 edi 中。
175         "movl %%edx,%%ecx\n\t"          // 再将串 2 的长度值放入 ecx 中。
176         "repne\n\t"                     // 比较 al 与串 2 中字符 es:[edi]，并且 edi++。
177         "scasb\n\t"                     // 如果不相等就继续比较。
178         "je 1b\n"                       // 如果相等，则向后跳转到标号 1 处。
179         "2:\tdecl %0"                   // esi--，指向最后一个包含在串 2 中的字符。
180         :"=S" (__res):"a" (0),"c" (0xffffffff),"0" (cs),"g" (ct)
181         :"ax","cx","dx","di");
182 return __res-cs;                        // 返回字符序列的长度值。
183 }
184
    //// 寻找字符串 1 中不包含字符串 2 中任何字符的首个字符序列。
    // 参数：cs - 字符串 1 指针，ct - 字符串 2 指针。
    // %0 - esi(__res)，%1 - eax(0)，%2 - ecx(-1)，%3 - esi(串 1 指针 cs)，%4 - (串 2 指针 ct)。
    // 返回字符串 1 中不包含字符串 2 中任何字符的首个字符序列的长度值。
185 extern inline int strcspn(const char * cs, const char * ct)
186 {
187 register char * __res __asm__("si"); // __res 是寄存器变量(esi)。
188 __asm__("cld\n\t"                       // 清方向位。
189         "movl %4,%%edi\n\t"             // 首先计算串 2 的长度。串 2 指针放入 edi 中。
190         "repne\n\t"                     // 比较 al(0)与串 2 中的字符（es:[edi]），并 edi++。
191         "scasb\n\t"                     // 如果不相等就继续比较(ecx 逐步递减)。
192         "notl %%ecx\n\t"                // ecx 中每位取反。
193         "decl %%ecx\n\t"                // ecx--，得串 2 的长度值。
194         "movl %%ecx,%%edx\n"            // 将串 2 的长度值暂放入 edx 中。
195         "1:\tlodsb\n\t"                 // 取串 1 字符 ds:[esi]➔al，并且 esi++。
196         "testb %%al,%%al\n\t"           // 该字符等于 0 值吗（串 1 结尾）？
197         "je 2f\n\t"                     // 如果是，则向前跳转到标号 2 处。
198         "movl %4,%%edi\n\t"             // 取串 2 头指针放入 edi 中。
199         "movl %%edx,%%ecx\n\t"          // 再将串 2 的长度值放入 ecx 中。
200         "repne\n\t"                     // 比较 al 与串 2 中字符 es:[edi]，并且 edi++。
201         "scasb\n\t"                     // 如果不相等就继续比较。
202         "jne 1b\n"                      // 如果不相等，则向后跳转到标号 1 处。
203         "2:\tdecl %0"                   // esi--，指向最后一个包含在串 2 中的字符。
204         :"=S" (__res):"a" (0),"c" (0xffffffff),"0" (cs),"g" (ct)
205         :"ax","cx","dx","di");
206 return __res-cs;                        // 返回字符序列的长度值。
207 }
208
    //// 在字符串 1 中寻找首个包含在字符串 2 中的任何字符。
    // 参数：cs - 字符串 1 的指针，ct - 字符串 2 的指针。
    // %0 -esi(__res)，%1 -eax(0)，%2 -ecx(0xffffffff)，%3 -esi(串 1 指针 cs)，%4 -(串 2 指针 ct)。
    // 返回字符串 1 中首个包含字符串 2 中字符的指针。
209 extern inline char * strpbrk(const char * cs,const char * ct)
```

```
{
register char * __res __asm__("si"); // __res 是寄存器变量(esi)。
__asm__("cld\n\t"                     // 清方向位。
        "movl %4,%%edi\n\t"           // 首先计算串 2 的长度。串 2 指针放入 edi 中。
        "repne\n\t"                   // 比较 al(0)与串 2 中的字符（es:[edi]），并 edi++。
        "scasb\n\t"                   // 如果不相等就继续比较(ecx 逐步递减)。
        "notl %%ecx\n\t"              // ecx 中每位取反。
        "decl %%ecx\n\t"              // ecx--，得串 2 的长度值。
        "movl %%ecx,%%edx\n"          // 将串 2 的长度值暂放入 edx 中。
        "1:\tlodsb\n\t"               // 取串 1 字符 ds:[esi]➔al，并且 esi++。
        "testb %%al,%%al\n\t"         // 该字符等于 0 值吗（串 1 结尾）？
        "je 2f\n\t"                   // 如果是，则向前跳转到标号 2 处。
        "movl %4,%%edi\n\t"           // 取串 2 头指针放入 edi 中。
        "movl %%edx,%%ecx\n\t"        // 再将串 2 的长度值放入 ecx 中。
        "repne\n\t"                   // 比较 al 与串 2 中字符 es:[edi]，并且 edi++。
        "scasb\n\t"                   // 如果不相等就继续比较。
        "jne 1b\n\t"                  // 如果不相等，则向后跳转到标号 1 处。
        "decl %0\n\t"                 // esi--，指向一个包含在串 2 中的字符。
        "jmp 3f\n"                    // 向前跳转到标号 3 处。
        "2:\txorl %0,%0\n"            // 没有找到符合条件的，将返回值为 NULL。
        "3:"
        :"=S"(__res):"a"(0),"c"(0xffffffff),"0"(cs),"g"(ct)
        :"ax","cx","dx","di");
return __res;                         // 返回指针值。
}

//// 在字符串 1 中寻找首个匹配整个字符串 2 的字符串。
// 参数：cs - 字符串 1 的指针，ct - 字符串 2 的指针。
// %0 -eax(__res)，%1 -eax(0)，%2 -ecx(0xffffffff)，%3 -esi(串 1 指针 cs)，%4 -(串 2 指针 ct)。
// 返回：返回字符串 1 中首个匹配字符串 2 的字符串指针。
extern inline char * strstr(const char * cs,const char * ct)
{
register char * __res __asm__("ax"); // __res 是寄存器变量(eax)。
__asm__("cld\n\t" \                   // 清方向位。
        "movl %4,%%edi\n\t"           // 首先计算串 2 的长度。串 2 指针放入 edi 中。
        "repne\n\t"                   // 比较 al(0)与串 2 中的字符（es:[edi]），并 edi++。
        "scasb\n\t"                   // 如果不相等就继续比较(ecx 逐步递减)。
        "notl %%ecx\n\t"              // ecx 中每位取反。
        "decl %%ecx\n\t"    /* NOTE! This also sets Z if searchstring='' */
                            /* 注意！如果搜索串为空，将设置 Z 标志 */ // 得串 2 的长度值。
        "movl %%ecx,%%edx\n"          // 将串 2 的长度值暂放入 edx 中。
        "1:\tmovl %4,%%edi\n\t"       // 取串 2 头指针放入 edi 中。
        "movl %%esi,%%eax\n\t"        // 将串 1 的指针复制到 eax 中。
        "movl %%edx,%%ecx\n\t"        // 再将串 2 的长度值放入 ecx 中。
        "repe\n\t"                    // 比较串 1 和串 2 字符(ds:[esi],es:[edi])，esi++,edi++。
        "cmpsb\n\t"                   // 若对应字符相等就一直比较下去。
        "je 2f\n\t"         /* also works for empty string, see above */
                            /* 对空串同样有效，见上面 */ // 若全相等，则转到标号 2。
        "xchgl %%eax,%%esi\n\t"       // 串 1 头指针➔esi，比较结果的串 1 指针➔eax。
        "incl %%esi\n\t"              // 串 1 头指针指向下一个字符。
        "cmpb $0,-1(%%eax)\n\t"       // 串 1 指针(eax-1)所指字节是 0 吗？
        "jne 1b\n\t"                  // 不是则转到标号 1，继续从串 1 的第 2 个字符开始比较。
        "xorl %%eax,%%eax\n\t"        // 清 eax，表示没有找到匹配。
```

```
257         "2:"
258         :"=a" (__res):"0" (0),"c" (0xffffffff),"S" (cs),"g" (ct)
259         :"cx","dx","di","si");
260 return __res;                          // 返回比较结果。
261 }
262
    //// 计算字符串长度。
    // 参数：s - 字符串。
    // %0 - ecx(__res)，%1 - edi(字符串指针 s)，%2 - eax(0)，%3 - ecx(0xffffffff)。
    // 返回：返回字符串的长度。
263 extern inline int strlen(const char * s)
264 {
265 register int __res __asm__("cx");     // __res 是寄存器变量(ecx)。
266 __asm__("cld\n\t"                      // 清方向位。
267         "repne\n\t"                    // al(0)与字符串中字符 es:[edi]比较，
268         "scasb\n\t"                    // 若不相等就一直比较。
269         "notl %0\n\t"                  // ecx 取反。
270         "decl %0"                      // ecx--，得字符串得长度值。
271         :"=c" (__res):"D" (s),"a" (0),"0" (0xffffffff):"di");
272 return __res;                          // 返回字符串长度值。
273 }
274
275 extern char * ___strtok;      // 用于临时存放指向下面被分析字符串 1(s)的指针。
276
    //// 利用字符串 2 中的字符将字符串 1 分割成标记(tokern)序列。
    // 将串 1 看作是包含零个或多个单词(token)的序列，并由分割符字符串 2 中的一个或多个字符
    // 分开。第一次调用 strtok()时，将返回指向字符串 1 中第 1 个 token 首字符的指针，并在返
    // 回 token 时将一 null 字符写到分割符处。后续使用 null 作为字符串 1 的调用，将用这种方
    // 法继续扫描字符串 1，直到没有 token 为止。在不同的调用过程中，分割符串 2 可以不同。
    // 参数：s - 待处理的字符串 1，ct - 包含各个分割符的字符串 2。
    // 汇编输出：%0 - ebx(__res)，%1 - esi(__strtok)；
    // 汇编输入：%2 - ebx(__strtok)，%3 - esi(字符串 1 指针 s)，%4 - (字符串 2 指针 ct)。
    // 返回：返回字符串 s 中第 1 个 token，如果没有找到 token，则返回一个 null 指针。
    // 后续使用字符串 s 指针为 null 的调用，将在原字符串 s 中搜索下一个 token。
277 extern inline char * strtok(char * s,const char * ct)
278 {
279 register char * __res __asm__("si");
280 __asm__("testl %1,%1\n\t"              // 首先测试 esi(字符串 1 指针 s)是否是 NULL。
281         "jne 1f\n\t"                   // 如果不是，则表明是首次调用本函数，跳转标号 1。
282         "testl %0,%0\n\t"              // 若是 NULL，表示此次是后续调用，测 ebx(__strtok)。
283         "je 8f\n\t"                    // 如果 ebx 指针是 NULL，则不能处理，跳转结束。
284         "movl %0,%1\n"                 // 将 ebx 指针复制到 esi。
285         "1:\txorl %0,%0\n\t"           // 清 ebx 指针。
286         "movl $-1,%%ecx\n\t"           // 置 ecx = 0xffffffff。
287         "xorl %%eax,%%eax\n\t"         // 清零 eax。
288         "cld\n\t"                      // 清方向位。
289         "movl %4,%%edi\n\t"            // 下面求字符串 2 的长度。edi 指向字符串 2。
290         "repne\n\t"                    // 将 al(0)与 es:[edi]比较，并且 edi++。
291         "scasb\n\t"                    // 直到找到字符串 2 的结束 null 字符，或计数 ecx==0。
292         "notl %%ecx\n\t"               // 将 ecx 取反，
293         "decl %%ecx\n\t"               // ecx--，得到字符串 2 的长度值。
294         "je 7f\n\t"                    /* empty delimeter-string */
                                           /* 分割符字符串空 */ // 若串 2 长度为 0，则转标号 7。
```

```
295         "movl %%ecx,%%edx\n"            // 将串 2 长度暂存入 edx。
296         "2:\tlodsb\n\t"                 // 取串 1 的字符 ds:[esi]➔al，并且 esi++。
297         "testb %%al,%%al\n\t"           // 该字符为 0 值吗(串 1 结束)？
298         "je 7f\n\t"                     // 如果是，则跳转标号 7。
299         "movl %4,%%edi\n\t"             // edi 再次指向串 2 首。
300         "movl %%edx,%%ecx\n\t"          // 取串 2 的长度值置入计数器 ecx。
301         "repne\n\t"                     // 将 al 中串 1 的字符与串 2 中所有字符比较，
302         "scasb\n\t"                     // 判断该字符是否为分割符。
303         "je 2b\n\t"                     // 若能在串 2 中找到相同字符（分割符），则跳转标号 2。
304         "decl %1\n\t"                   // 若不是分割符，则串 1 指针 esi 指向此时的该字符。
305         "cmpb $0,(%1)\n\t"              // 该字符是 NULL 字符吗？
306         "je 7f\n\t"                     // 若是，则跳转标号 7 处。
307         "movl %1,%0\n"                  // 将该字符的指针 esi 存放在 ebx。
308         "3:\tlodsb\n\t"                 // 取串 1 下一个字符 ds:[esi]➔al，并且 esi++。
309         "testb %%al,%%al\n\t"           // 该字符是 NULL 字符吗？
310         "je 5f\n\t"                     // 若是，表示串 1 结束，跳转到标号 5。
311         "movl %4,%%edi\n\t"             // edi 再次指向串 2 首。
312         "movl %%edx,%%ecx\n\t"          // 串 2 长度值置入计数器 ecx。
313         "repne\n\t"                     // 将 al 中串 1 的字符与串 2 中每个字符比较，
314         "scasb\n\t"                     // 测试 al 字符是否是分割符。
315         "jne 3b\n\t"                    // 若不是分割符则跳转标号 3，检测串 1 中下一个字符。
316         "decl %1\n\t"                   // 若是分割符，则 esi--，指向该分割符字符。
317         "cmpb $0,(%1)\n\t"              // 该分割符是 NULL 字符吗？
318         "je 5f\n\t"                     // 若是，则跳转到标号 5。
319         "movb $0,(%1)\n\t"              // 若不是，则将该分割符用 NULL 字符替换掉。
320         "incl %1\n\t"                   // esi 指向串 1 中下一个字符，也即剩余串首。
321         "jmp 6f\n"                      // 跳转标号 6 处。
322         "5:\txorl %1,%1\n"              // esi 清零。
323         "6:\tcmpb $0,(%0)\n\t"          // ebx 指针指向 NULL 字符吗？
324         "jne 7f\n\t"                    // 若不是，则跳转标号 7。
325         "xorl %0,%0\n"                  // 若是，则让 ebx=NULL。
326         "7:\ttestl %0,%0\n\t"           // ebx 指针为 NULL 吗？
327         "jne 8f\n\t"                    // 若不是则跳转 8，结束汇编代码。
328         "movl %0,%1\n"                  // 将 esi 置为 NULL。
329         "8:"
330         :"=b" (__res),"=S" (___strtok)
331         :"0" (___strtok),"1" (s),"g" (ct)
332         :"ax","cx","dx","di");
333 return __res;                           // 返回指向新 token 的指针。
334 }
335
    //// 内存块复制。从源地址 src 处开始复制 n 个字节到目的地址 dest 处。
    // 参数：dest - 复制的目的地址，src - 复制的源地址，n - 复制字节数。
    // %0 - ecx(n)，%1 - esi(src)，%2 - edi(dest)。
336 extern inline void * memcpy(void * dest,const void * src, int n)
337 {
338 __asm__("cld\n\t"                       // 清方向位。
339         "rep\n\t"                       // 重复执行复制 ecx 个字节，
340         "movsb"                         // 从 ds:[esi]到 es:[edi]，esi++，edi++。
341         ::"c" (n),"S" (src),"D" (dest)
342         :"cx","si","di");
343 return dest;                            // 返回目的地址。
344 }
```

```
345
    //// 内存块移动。同内存块复制，但考虑移动的方向。
    // 参数：dest - 复制的目的地址，src - 复制的源地址，n - 复制字节数。
    // 若 dest<src 则：%0 - ecx(n)，%1 - esi(src)，%2 - edi(dest)。
    // 否则：%0 - ecx(n)，%1 - esi(src+n-1)，%2 - edi(dest+n-1)。
    // 这样操作是为了防止在复制时错误地重叠覆盖。
346 extern inline void * memmove(void * dest,const void * src, int n)
347 {
348 if (dest<src)
349 __asm__("cld\n\t"                       // 清方向位。
350         "rep\n\t"                       // 从 ds:[esi]到 es:[edi]，并且 esi++，edi++，
351         "movsb"                         // 重复执行复制 ecx 字节。
352         ::"c" (n),"S" (src),"D" (dest)
353         :"cx","si","di");
354 else
355 __asm__("std\n\t"                       // 置方向位，从末端开始复制。
356         "rep\n\t"                       // 从 ds:[esi]到 es:[edi]，并且 esi--，edi--，
357         "movsb"                         // 复制 ecx 个字节。
358         ::"c" (n),"S" (src+n-1),"D" (dest+n-1)
359         :"cx","si","di");
360 return dest;
361 }
362
    //// 比较 n 个字节的两块内存（两个字符串），即使遇上 NULL 字节也不停止比较。
    // 参数：cs - 内存块 1 地址，ct - 内存块 2 地址，count - 比较的字节数。
    // %0 - eax(__res)，%1 - eax(0)，%2 - edi(内存块 1)，%3 - esi(内存块 2)，%4 - ecx(count)。
    // 返回：若块 1>块 2 返回 1；块 1<块 2，返回-1；块 1==块 2，则返回 0。
363 extern inline int memcmp(const void * cs,const void * ct,int count)
364 {
365 register int __res __asm__("ax");      // __res 是寄存器变量。
366 __asm__("cld\n\t"                       // 清方向位。
367         "repe\n\t"                      // 如果相等则重复，
368         "cmpsb\n\t"                     // 比较 ds:[esi]与 es:[edi]的内容，并且 esi++，edi++。
369         "je 1f\n\t"                     // 如果都相同，则跳转到标号 1，返回 0(eax)值
370         "movl $1,%%eax\n\t"             // 否则 eax 置 1，
371         "jl 1f\n\t"                     // 若内存块 2 内容的值<内存块 1，则跳转标号 1。
372         "negl %%eax\n"                  // 否则 eax = -eax。
373         "1:"
374         :"=a" (__res):"0" (0),"D" (cs),"S" (ct),"c" (count)
375         :"si","di","cx");
376 return __res;                           // 返回比较结果。
377 }
378
    //// 在 n 字节大小的内存块(字符串)中寻找指定字符。
    // 参数：cs - 指定内存块地址，c - 指定的字符，count - 内存块长度。
    // %0 - edi(__res)，%1 - eax(字符 c)，%2 - edi(内存块地址 cs)，%3 - ecx(字节数 count)。
    // 返回第一个匹配字符的指针，如果没有找到，则返回 NULL 字符。
379 extern inline void * memchr(const void * cs,char c,int count)
380 {
381 register void * __res __asm__("di"); // __res 是寄存器变量。
382 if (!count)                             // 如果内存块长度==0，则返回 NULL，没有找到。
383         return NULL;
384 __asm__("cld\n\t"                       // 清方向位。
```

```
385         "repne\n\t"                 // 如果不相等则重复执行下面语句，
386         "scasb\n\t"                 // al 中字符与 es:[edi]字符作比较，并且 edi++，
387         "je 1f\n\t"                 // 如果相等则向前跳转到标号 1 处。
388         "movl $1,%0\n"              // 否则 edi 中置 1。
389         "1:\tdecl %0"               // 让 edi 指向找到的字符（或是 NULL）。
390         :"=D" (__res):"a" (c),"D" (cs),"c" (count)
391         :"cx");
392 return __res;                       // 返回字符指针。
393 }
394
    //// 用字符填写指定长度内存块。
    // 用字符 c 填写 s 指向的内存区域，共填 count 字节。
    // %0 - eax(字符 c)，%1 - edi(内存地址)，%2 - ecx(字节数 count)。
395 extern inline void * memset(void * s,char c,int count)
396 {
397 __asm__("cld\n\t"                   // 清方向位。
398         "rep\n\t"                   // 重复 ecx 指定的次数，执行
399         "stosb"                     // 将 al 中字符存入 es:[edi]中，并且 edi++。
400         ::"a" (c),"D" (s),"c" (count)
401         :"cx","di");
402 return s;
403 }
404
405 #endif
406
```

14.10 程序 14-10 linux/include/termios.h

```
 1 #ifndef _TERMIOS_H
 2 #define _TERMIOS_H
 3
 4 #include <sys/types.h>
 5
 6 #define TTY_BUF_SIZE 1024            // tty 中的缓冲区长度。
 7
 8 /* 0x54 is just a magic number to make these relatively unique ('T') */
   /* 0x54 只是一个魔数，目的是为了使这些常数唯一('T') */
 9
   // tty 设备的 ioctl 调用命令集。ioctl 将命令编码在低位字中。
   // 下面名称 TC[*]的含义是 tty 控制命令。
   // 取相应终端 termios 结构中的信息(参见 tcgetattr())。
10 #define TCGETS          0x5401
   // 设置相应终端 termios 结构中的信息(参见 tcsetattr(), TCSANOW)。
11 #define TCSETS          0x5402
   // 在设置终端 termios 的信息之前，需要先等待输出队列中所有数据处理完(耗尽)。对于修改参数
   // 会影响输出的情况，就需要使用这种形式(参见 tcsetattr(), TCSADRAIN 选项)。
12 #define TCSETSW         0x5403
   // 在设置 termios 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新(清空)输入队列。
   // 再设置（参见 tcsetattr(), TCSAFLUSH 选项）。
13 #define TCSETSF         0x5404
   // 取相应终端 termio 结构中的信息(参见 tcgetattr())。
14 #define TCGETA          0x5405
   // 设置相应终端 termio 结构中的信息(参见 tcsetattr(), TCSANOW 选项)。
15 #define TCSETA          0x5406
   // 在设置终端 termio 的信息之前，需要先等待输出队列中所有数据处理完(耗尽)。对于修改参数
   // 会影响输出的情况，就需要使用这种形式(参见 tcsetattr(), TCSADRAIN 选项)。
16 #define TCSETAW         0x5407
   // 在设置 termio 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新(清空)输入队列。
   // 再设置（参见 tcsetattr(), TCSAFLUSH 选项）。
17 #define TCSETAF         0x5408
   // 等待输出队列处理完毕(空)，若参数值是 0，则发送一个 break（参见 tcsendbreak(), tcdrain()）。
18 #define TCSBRK          0x5409
   // 开始/停止控制。如果参数值是 0，则挂起输出；如果是 1，则重新开启挂起的输出；如果是 2，
   // 则挂起输入；如果是 3，则重新开启挂起的输入（参见 tcflow()）。
19 #define TCXONC          0x540A
   // 刷新已写输出但还没发送或已收但还没有读数据。如果参数是 0，则刷新(清空)输入队列；如果
   // 是 1，则刷新输出队列；如果是 2，则刷新输入和输出队列（参见 tcflush()）。
20 #define TCFLSH          0x540B
   // 下面名称 TIOC[*]的含义是 tty 输入输出控制命令。
   // 设置终端串行线路专用模式。
21 #define TIOCEXCL        0x540C
   // 复位终端串行线路专用模式。
22 #define TIOCNXCL        0x540D
   // 设置 tty 为控制终端。(TIOCNOTTY - 禁止 tty 为控制终端)。
23 #define TIOCSCTTY       0x540E
   // 读取指定终端设备进程的组 id，参见 tcgetpgrp()。该常数符号名称是"Terminal IO Control
   // Get PGRP "的缩写。读取前台进程组 ID。
```

```
24 #define TIOCGPGRP       0x540F
   // 设置指定终端设备进程的组 id(参见 tcsetpgrp())。
25 #define TIOCSPGRP       0x5410
   // 返回输出队列中还未送出的字符数。
26 #define TIOCOUTQ        0x5411
   // 模拟终端输入。该命令以一个指向字符的指针作为参数，并假装该字符是在终端上键入的。用户
   // 必须在该控制终端上具有超级用户权限或具有读许可权限。
27 #define TIOCSTI         0x5412
   // 读取终端设备窗口大小信息（参见 winsize 结构）。
28 #define TIOCGWINSZ      0x5413
   // 设置终端设备窗口大小信息（参见 winsize 结构）。
29 #define TIOCSWINSZ      0x5414
   // 返回 modem 状态控制引线的当前状态比特位标志集（参见下面 185-196 行）。
30 #define TIOCMGET        0x5415
   // 设置单个 modem 状态控制引线的状态(true 或 false)(Individual control line Set)。
31 #define TIOCMBIS        0x5416
   // 复位单个 modem 状态控制引线的状态(Individual control line clear)。
32 #define TIOCMBIC        0x5417
   // 设置 modem 状态引线的状态。如果某一比特位置位，则 modem 对应的状态引线将置为有效。
33 #define TIOCMSET        0x5418
   // 读取软件载波检测标志(1 - 开启；0 - 关闭)。
   // 对于本地连接的终端或其他设备，软件载波标志是开启的，对于使用 modem 线路的终端或设备
   // 则是关闭的。为了能使用这两个 ioctl 调用，tty 线路应该是以 O_NDELAY 方式打开的，这样
   // open()就不会等待载波。
34 #define TIOCGSOFTCAR    0x5419
   // 设置软件载波检测标志(1 - 开启；0 - 关闭)。
35 #define TIOCSSOFTCAR    0x541A
   // 返回输入队列中还未取走字符的数目。
36 #define FIONREAD        0x541B
37 #define TIOCINQ         FIONREAD
38
   // 窗口大小(Window size)属性结构。在窗口环境中可用于基于屏幕的应用程序。
   // ioctls 中的 TIOCGWINSZ 和 TIOCSWINSZ 可用来读取或设置这些信息。
39 struct winsize {
40         unsigned short ws_row;      // 窗口字符行数。
41         unsigned short ws_col;      // 窗口字符列数。
42         unsigned short ws_xpixel;   // 窗口宽度，象素值。
43         unsigned short ws_ypixel;   // 窗口高度，象素值。
44 };
45
   // AT&T 系统 V 的 termio 结构。
46 #define NCC 8                       // termio 结构中控制字符数组的长度。
47 struct termio {
48         unsigned short c_iflag;     /* input mode flags */      // 输入模式标志。
49         unsigned short c_oflag;     /* output mode flags */     // 输出模式标志。
50         unsigned short c_cflag;     /* control mode flags */    // 控制模式标志。
51         unsigned short c_lflag;     /* local mode flags */      // 本地模式标志。
52         unsigned char c_line;       /* line discipline */       // 线路规程（速率）。
53         unsigned char c_cc[NCC];    /* control characters */    // 控制字符数组。
54 };
55
   // POSIX 的 termios 结构。
56 #define NCCS 17                     // termios 结构中控制字符数组长度。
```

```
57 struct termios {
58         tcflag_t c_iflag;           /* input mode flags */      // 输入模式标志。
59         tcflag_t c_oflag;           /* output mode flags */     // 输出模式标志。
60         tcflag_t c_cflag;           /* control mode flags */    // 控制模式标志。
61         tcflag_t c_lflag;           /* local mode flags */      // 本地模式标志。
62         cc_t c_line;                /* line discipline */       // 线路规程（速率）。
63         cc_t c_cc[NCCS];            /* control characters */    // 控制字符数组。
64 };
65
   // 以下是控制字符数组 c_cc[]中项的索引值。该数组初始值定义在 include/linux/tty.h 中。
   // 程序可以更改这个数组中的值。如果定义了_POSIX_VDISABLE（\0），那么当数组某一项值
   // 等于 _POSIX_VDISABLE 的值时，表示禁止使用数组中相应的特殊字符。
66 /* c_cc characters */    /* c_cc 数组中的字符 */
67 #define VINTR 0          // c_cc[VINTR]    = INTR    (^C), \003, 中断字符。
68 #define VQUIT 1          // c_cc[VQUIT]    = QUIT    (^\), \034, 退出字符。
69 #define VERASE 2         // c_cc[VERASE]   = ERASE   (^H), \177, 擦出字符。
70 #define VKILL 3          // c_cc[VKILL]    = KILL    (^U), \025, 终止字符（删除行）。
71 #define VEOF 4           // c_cc[VEOF]     = EOF     (^D), \004, 文件结束字符。
72 #define VTIME 5          // c_cc[VTIME]    = TIME    (\0), \0,   定时器值(参见后面说明)。
73 #define VMIN 6           // c_cc[VMIN]     = MIN     (\1), \1,   定时器值。
74 #define VSWTC 7          // c_cc[VSWTC]    = SWTC    (\0), \0,   交换字符。
75 #define VSTART 8         // c_cc[VSTART]   = START   (^Q), \021, 开始字符。
76 #define VSTOP 9          // c_cc[VSTOP]    = STOP    (^S), \023, 停止字符。
77 #define VSUSP 10         // c_cc[VSUSP]    = SUSP    (^Z), \032, 挂起字符。
78 #define VEOL 11          // c_cc[VEOL]     = EOL     (\0), \0,   行结束字符。
79 #define VREPRINT 12      // c_cc[VREPRINT] = REPRINT (^R), \022, 重显示字符。
80 #define VDISCARD 13      // c_cc[VDISCARD] = DISCARD (^O), \017, 丢弃字符。
81 #define VWERASE 14       // c_cc[VWERASE]  = WERASE  (^W), \027, 单词擦除字符。
82 #define VLNEXT 15        // c_cc[VLNEXT]   = LNEXT   (^V), \026, 下一行字符。
83 #define VEOL2 16         // c_cc[VEOL2]    = EOL2    (\0), \0,   行结束字符 2。
84
   // termios 结构输入模式字段 c_iflag 各种标志的符号常数。
85 /* c_iflag bits */       /* c_iflag 比特位 */
86 #define IGNBRK  0000001          // 输入时忽略 BREAK 条件。
87 #define BRKINT  0000002          // 在 BREAK 时产生 SIGINT 信号。
88 #define IGNPAR  0000004          // 忽略奇偶校验出错的字符。
89 #define PARMRK  0000010          // 标记奇偶校验错。
90 #define INPCK   0000020          // 允许输入奇偶校验。
91 #define ISTRIP  0000040          // 屏蔽字符第 8 位。
92 #define INLCR   0000100          // 输入时将换行符 NL 映射成回车符 CR。
93 #define IGNCR   0000200          // 忽略回车符 CR。
94 #define ICRNL   0000400          // 在输入时将回车符 CR 映射成换行符 NL。
95 #define IUCLC   0001000          // 在输入时将大写字符转换成小写字符。
96 #define IXON    0002000          // 允许开始/停止（XON/XOFF）输出控制。
97 #define IXANY   0004000          // 允许任何字符重启输出。
98 #define IXOFF   0010000          // 允许开始/停止（XON/XOFF）输入控制。
99 #define IMAXBEL 0020000          // 输入队列满时响铃。
100
   // termios 结构中输出模式字段 c_oflag 各种标志的符号常数。
101 /* c_oflag bits */       /* c_oflag 比特位 */
102 #define OPOST   0000001          // 执行输出处理。
103 #define OLCUC   0000002          // 在输出时将小写字符转换成大写字符。
104 #define ONLCR   0000004          // 在输出时将换行符 NL 映射成回车-换行符 CR-NL。
```

```
105 #define OCRNL    0000010          // 在输出时将回车符 CR 映射成换行符 NL。
106 #define ONOCR    0000020          // 在 0 列不输出回车符 CR。
107 #define ONLRET   0000040          // 换行符 NL 执行回车符的功能。
108 #define OFILL    0000100          // 延迟时使用填充字符而不使用时间延迟。
109 #define OFDEL    0000200          // 填充字符是 ASCII 码 DEL。如果未设置，则使用 ASCII NULL。
110 #define NLDLY    0000400          // 选择换行延迟。
111 #define   NL0    0000000          // 换行延迟类型 0。
112 #define   NL1    0000400          // 换行延迟类型 1。
113 #define CRDLY    0003000          // 选择回车延迟。
114 #define   CR0    0000000          // 回车延迟类型 0。
115 #define   CR1    0001000          // 回车延迟类型 1。
116 #define   CR2    0002000          // 回车延迟类型 2。
117 #define   CR3    0003000          // 回车延迟类型 3。
118 #define TABDLY   0014000          // 选择水平制表延迟。
119 #define   TAB0   0000000          // 水平制表延迟类型 0。
120 #define   TAB1   0004000          // 水平制表延迟类型 1。
121 #define   TAB2   0010000          // 水平制表延迟类型 2。
122 #define   TAB3   0014000          // 水平制表延迟类型 3。
123 #define   XTABS  0014000          // 将制表符 TAB 换成空格，该值表示空格数。
124 #define BSDLY    0020000          // 选择退格延迟。
125 #define   BS0    0000000          // 退格延迟类型 0。
126 #define   BS1    0020000          // 退格延迟类型 1。
127 #define VTDLY    0040000          // 纵向制表延迟。
128 #define   VT0    0000000          // 纵向制表延迟类型 0。
129 #define   VT1    0040000          // 纵向制表延迟类型 1。
130 #define FFDLY    0040000          // 选择换页延迟。
131 #define   FF0    0000000          // 换页延迟类型 0。
132 #define   FF1    0040000          // 换页延迟类型 1。
133
    // termios 结构中控制模式标志字段 c_cflag 标志的符号常数（8 进制数）。
134 /* c_cflag bit meaning */     /* c_cflag 比特位的含义 */
135 #define CBAUD    0000017          // 传输速率位屏蔽码。
136 #define   B0     0000000          /* hang up */  /* 挂断线路 */
137 #define   B50    0000001          // 波特率 50。
138 #define   B75    0000002          // 波特率 75。
139 #define   B110   0000003          // 波特率 110。
140 #define   B134   0000004          // 波特率 134。
141 #define   B150   0000005          // 波特率 150。
142 #define   B200   0000006          // 波特率 200。
143 #define   B300   0000007          // 波特率 300。
144 #define   B600   0000010          // 波特率 600。
145 #define   B1200  0000011          // 波特率 1200。
146 #define   B1800  0000012          // 波特率 1800。
147 #define   B2400  0000013          // 波特率 2400。
148 #define   B4800  0000014          // 波特率 4800。
149 #define   B9600  0000015          // 波特率 9600。
150 #define   B19200 0000016          // 波特率 19200。
151 #define   B38400 0000017          // 波特率 38400。
152 #define EXTA B19200               // 扩展波特率 A。
153 #define EXTB B38400               // 扩展波特率 B。

154 #define CSIZE    0000060          // 字符位宽度屏蔽码。
155 #define   CS5    0000000          // 每字符 5 比特位。
```

```
156 #define   CS6    0000020          // 每字符 6 比特位。
157 #define   CS7    0000040          // 每字符 7 比特位。
158 #define   CS8    0000060          // 每字符 8 比特位。
159 #define CSTOPB   0000100          // 设置两个停止位，而不是 1 个。
160 #define CREAD    0000200          // 允许接收。
161 #define PARENB   0000400          // 开启输出时产生奇偶位、输入时进行奇偶校验。
162 #define PARODD   0001000          // 输入/输入校验是奇校验。
163 #define HUPCL    0002000          // 最后进程关闭后挂断。
164 #define CLOCAL   0004000          // 忽略调制解调器(modem)控制线路。
165 #define CIBAUD   03600000         /* input baud rate (not used) */  /* 输入波特率(未使用) */
166 #define CRTSCTS 020000000000      /* flow control */  /* 流控制 */
167
    // termios 结构中本地模式标志字段 c_lflag 的符号常数。
168 /* c_lflag bits */        /* c_lflag 比特位 */
169 #define ISIG     0000001          // 当收到字符 INTR、QUIT、SUSP 或 DSUSP，产生相应的信号。
170 #define ICANON   0000002          // 开启规范模式（熟模式）。
171 #define XCASE    0000004          // 若设置了 ICANON，则终端是大写字符的。
172 #define ECHO     0000010          // 回显输入字符。
173 #define ECHOE    0000020          // 若设置了 ICANON，则 ERASE/WERASE 将擦除前一字符/单词。
174 #define ECHOK    0000040          // 若设置了 ICANON，则 KILL 字符将擦除当前行。
175 #define ECHONL   0000100          // 如设置了 ICANON，则即使 ECHO 没有开启也回显 NL 字符。
176 #define NOFLSH   0000200          // 当生成 SIGINT 和 SIGQUIT 信号时不刷新输入输出队列，当
                                      // 生成 SIGSUSP 信号时，刷新输入队列。
177 #define TOSTOP   0000400          // 发送 SIGTTOU 信号到后台进程的进程组，该后台进程试图写
                                      // 自己的控制终端。
178 #define ECHOCTL  0001000          // 若设置了 ECHO，则除 TAB、NL、START 和 STOP 以外的 ASCII
                                      // 控制信号将被回显成象^X 式样，X 值是控制符+0x40。
179 #define ECHOPRT  0002000          // 若设置了 ICANON 和 IECHO，则字符在擦除时将显示。
180 #define ECHOKE   0004000          // 若设置了 ICANON，则 KILL 通过擦除行上的所有字符被回显。
181 #define FLUSHO   0010000          // 输出被刷新。通过键入 DISCARD 字符，该标志被翻转。
182 #define PENDIN   0040000          // 当下一个字符是读时，输入队列中的所有字符将被重显。
183 #define IEXTEN   0100000          // 开启实现时定义的输入处理。
184
185 /* modem lines */    /* modem 线路信号符号常数 */
186 #define TIOCM_LE        0x001       // 线路允许(Line Enable)。
187 #define TIOCM_DTR       0x002       // 数据终端就绪(Data Terminal Ready)。
188 #define TIOCM_RTS       0x004       // 请求发送(Request to Send)。
189 #define TIOCM_ST        0x008       // 串行数据发送(Serial Transfer)。[??]
190 #define TIOCM_SR        0x010       // 串行数据接收(Serial Receive)。[??]
191 #define TIOCM_CTS       0x020       // 清除发送(Clear To Send)。
192 #define TIOCM_CAR       0x040       // 载波监测(Carrier Detect)。
193 #define TIOCM_RNG       0x080       // 响铃指示(Ring indicate)。
194 #define TIOCM_DSR       0x100       // 数据设备就绪(Data Set Ready)。
195 #define TIOCM_CD        TIOCM_CAR
196 #define TIOCM_RI        TIOCM_RNG
197
198 /* tcflow() and TCXONC use these */   /* tcflow()和 TCXONC 使用这些符号常数 */
199 #define TCOOFF          0          // 挂起输出（是"Terminal Control Output OFF"的缩写）。
200 #define TCOON           1          // 重启被挂起的输出。
201 #define TCIOFF          2          // 系统传输一个 STOP 字符，使设备停止向系统传输数据。
202 #define TCION           3          // 系统传输一个 START 字符，使设备开始向系统传输数据。
203
204 /* tcflush() and TCFLSH use these */  /* tcflush()和 TCFLSH 使用这些符号常数 */
```

```
205 #define TCIFLUSH        0           // 清接收到的数据但不读。
206 #define TCOFLUSH        1           // 清已写的数据但不传送。
207 #define TCIOFLUSH       2           // 清接收到的数据但不读。清已写的数据但不传送。
208
209 /* tcsetattr uses these */          /* tcsetattr()使用这些符号常数 */
210 #define TCSANOW         0           // 改变立即发生。
211 #define TCSADRAIN       1           // 改变在所有已写的输出被传输之后发生。
212 #define TCSAFLUSH       2           // 改变在所有已写的输出被传输之后并且在所有接收到但
                                        // 还没有读取的数据被丢弃之后发生。
213
    // 以下这些函数在编译环境的函数库 libc.a 中实现，内核中没有。在函数库实现中，这些函数通过
    // 调用系统调用 ioctl()来实现。有关 ioctl()系统调用，请参见 fs/ioctl.c 程序。
    // 返回 termios_p 所指 termios 结构中的接收波特率。
214 extern speed_t cfgetispeed(struct termios *termios_p);
    // 返回 termios_p 所指 termios 结构中的发送波特率。
215 extern speed_t cfgetospeed(struct termios *termios_p);
    // 将 termios_p 所指 termios 结构中的接收波特率设置为 speed。
216 extern int cfsetispeed(struct termios *termios_p, speed_t speed);
    // 将 termios_p 所指 termios 结构中的发送波特率设置为 speed。
217 extern int cfsetospeed(struct termios *termios_p, speed_t speed);
    // 等待 fildes 所指对象已写输出数据被传送出去。
218 extern int tcdrain(int fildes);
    // 挂起/重启 fildes 所指对象数据的接收和发送。
219 extern int tcflow(int fildes, int action);
    // 丢弃 fildes 指定对象所有已写但还没传送以及所有已收到但还没有读取的数据。
220 extern int tcflush(int fildes, int queue_selector);
    // 获取与句柄 fildes 对应对象的参数，并将其保存在 termios_p 所指的地方。
221 extern int tcgetattr(int fildes, struct termios *termios_p);
    // 如果终端使用异步串行数据传输，则在一定时间内连续传输一系列 0 值比特位。
222 extern int tcsendbreak(int fildes, int duration);
    // 使用 termios 结构指针 termios_p 所指的数据，设置与终端相关的参数。
223 extern int tcsetattr(int fildes, int optional_actions,
224         struct termios *termios_p);
225
226 #endif
227
```

14.11 程序 14-11 linux/include/time.h

```
 1 #ifndef _TIME_H
 2 #define _TIME_H
 3
 4 #ifndef _TIME_T
 5 #define _TIME_T
 6 typedef long time_t;                   // 从 GMT 1970 年 1 月 1 日午夜 0 时起开始计的时间（秒）。
 7 #endif
 8
 9 #ifndef _SIZE_T
10 #define _SIZE_T
11 typedef unsigned int size_t;
12 #endif
13
14 #ifndef NULL
15 #define NULL ((void *) 0)
16 #endif
17
18 #define CLOCKS_PER_SEC 100              // 系统时钟滴答频率，100HZ。
19
20 typedef long clock_t;                   // 从进程开始执行计起的系统经过的时钟滴答数。
21
22 struct tm {
23         int tm_sec;                     // 秒数 [0，59]。
24         int tm_min;                     // 分钟数 [ 0，59]。
25         int tm_hour;                    // 小时数 [0，59]。
26         int tm_mday;                    // 1 个月的天数 [0，31]。
27         int tm_mon;                     // 1 年中月份 [0，11]。
28         int tm_year;                    // 从 1900 年开始的年数。
29         int tm_wday;                    // 1 星期中的某天 [0，6]（星期天 =0）。
30         int tm_yday;                    // 1 年中的某天 [0，365]。
31         int tm_isdst;                   // 夏令时标志。正数 - 使用；0 - 没有使用；负数 - 无效。
32 };
33
   // 判断是否为闰年的宏。
34 #define __isleap(year)  \
35   ((year) % 4 == 0 && ((year) % 100 != 0 || (year) % 1000 == 0))
36
   // 以下是有关时间操作的函数原型。
   // 确定处理器使用时间。返回程序所用处理器时间（滴答数）的近似值。
37 clock_t clock(void);
   // 取时间（秒数）。返回从 1970.1.1:0:0:0 开始的秒数（称为日历时间）。
38 time_t time(time_t * tp);
   // 计算时间差。返回时间 time2 与 time1 之间经过的秒数。
39 double difftime(time_t time2, time_t time1);
   // 将 tm 结构表示的时间转换成日历时间。
40 time_t mktime(struct tm * tp);
41
   // 将 tm 结构表示的时间转换成一个字符串。返回指向该串的指针。
42 char * asctime(const struct tm * tp);
```

```
    // 将日历时间转换成一个字符串形式，如“Wed Jun 30 21:49:08:1993\n”。
43  char * ctime(const time_t * tp);
    // 将日历时间转换成 tm 结构表示的 UTC 时间（UTC - 世界时间代码 Universal Time Code）。
44  struct tm * gmtime(const time_t *tp);
    // 将日历时间转换成 tm 结构表示的指定时区(Time Zone)的时间。
45  struct tm *localtime(const time_t * tp);
    // 将 tm 结构表示的时间利用格式字符串 fmt 转换成最大长度为 smax 的字符串并将结果存储在 s 中。
46  size_t strftime(char * s, size_t smax, const char * fmt, const struct tm * tp);
    // 初始化时间转换信息，使用环境变量 TZ，对 zname 变量进行初始化。
    // 在与时区相关的时间转换函数中将自动调用该函数。
47  void tzset(void);
48
49  #endif
50
```

14.12 程序 14-12 linux/include/unistd.h

```
 1 #ifndef _UNISTD_H
 2 #define _UNISTD_H
 3
 4 /* ok, this may be a joke, but I'm working on it */
   /* ok, 这也许是个玩笑，但我正在着手处理 */
   // 下面符号常数指出符合 IEEE 标准 1003.1 实现的版本号，是一个整数值。
 5 #define _POSIX_VERSION 198808L
 6
   // chown()和 fchown()的使用受限于进程的权限。/* 只有超级用户可以执行 chown（我想..）*/
 7 #define _POSIX_CHOWN_RESTRICTED /* only root can do a chown (I think..) */
   // 长于(NAME_MAX)的路径名将产生错误，而不会自动截断。/* 路径名不截断（但是请看内核代码）*/
 8 #define _POSIX_NO_TRUNC         /* no pathname truncation (but see in kernel) */
   // 下面这个符号将定义成字符值，该值将禁止终端对其的处理。/* 禁止象^C 这样的字符 */
   // _POSIX_VDISABLE 用于控制终端某些特殊字符的功能。当一个终端 termios 结构中 c_cc[]
   // 数组某项字符代码值等于_POSIX_VDISABLE 的值时，表示禁止使用相应的特殊字符。
 9 #define _POSIX_VDISABLE '\0'    /* character to disable things like ^C */
   // 系统实现支持作业控制。
10 #define _POSIX_JOB_CONTROL
   // 每个进程都有一保存的 set-user-ID 和一保存的 set-group-ID。 /* 已经实现。 */
11 #define _POSIX_SAVED_IDS        /* Implemented, for whatever good it is */
12
13 #define STDIN_FILENO    0           // 标准输入文件句柄（描述符）号。
14 #define STDOUT_FILENO   1           // 标准输出文件句柄号。
15 #define STDERR_FILENO   2           // 标准出错文件句柄号。
16
17 #ifndef NULL
18 #define NULL    ((void *)0)         // 定义空指针。
19 #endif
20
21 /* access */   /* 文件访问 */
   // 以下定义的符号常数用于 access()函数。
22 #define F_OK    0                   // 检测文件是否存在。
23 #define X_OK    1                   // 检测是否可执行（搜索）。
24 #define W_OK    2                   // 检测是否可写。
25 #define R_OK    4                   // 检测是否可读。
26
27 /* lseek */   /* 文件指针重定位 */
   // 以下符号常数用于 lseek()和 fcntl()函数。
28 #define SEEK_SET        0           // 将文件读写指针设置为偏移值。
29 #define SEEK_CUR        1           // 将文件读写指针设置为当前值加上偏移值。
30 #define SEEK_END        2           // 将文件读写指针设置为文件长度加上偏移值。
31
32 /* _SC stands for System Configuration. We don't use them much */
   /* _SC 表示系统配置。我们很少使用 */
   // 下面的符号常数用于 sysconf()函数。
33 #define _SC_ARG_MAX             1   // 最大变量数。
34 #define _SC_CHILD_MAX           2   // 子进程最大数。
35 #define _SC_CLOCKS_PER_SEC      3   // 每秒滴答数。
36 #define _SC_NGROUPS_MAX         4   // 最大组数。
```

```
37 #define _SC_OPEN_MAX            5    // 最大打开文件数。
38 #define _SC_JOB_CONTROL         6    // 作业控制。
39 #define _SC_SAVED_IDS           7    // 保存的标识符。
40 #define _SC_VERSION             8    // 版本。
41
42 /* more (possibly) configurable things - now pathnames */
   /* 更多的（可能的）可配置参数 - 现在用于路径名 */
   // 下面的符号常数用于 pathconf()函数。
43 #define _PC_LINK_MAX            1    // 连接最大数。
44 #define _PC_MAX_CANON           2    // 最大常规文件数。
45 #define _PC_MAX_INPUT           3    // 最大输入长度。
46 #define _PC_NAME_MAX            4    // 名称最大长度。
47 #define _PC_PATH_MAX            5    // 路径最大长度。
48 #define _PC_PIPE_BUF            6    // 管道缓冲大小。
49 #define _PC_NO_TRUNC            7    // 文件名不截断。
50 #define _PC_VDISABLE            8    //
51 #define _PC_CHOWN_RESTRICTED    9    // 改变宿主受限。
52
53 #include <sys/stat.h>     // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。
54 #include <sys/time.h>
55 #include <sys/times.h>    // 定义了进程中运行时间结构 tms 以及 times()函数原型。
56 #include <sys/utsname.h>  // 系统名称结构头文件。
57 #include <sys/resource.h>
58 #include <utime.h>        // 用户时间头文件。定义了访问和修改时间结构以及 utime()原型。
59
60 #ifdef __LIBRARY__
61
   // 以下是实现的系统调用符号常数，用作系统调用函数表中索引值(参见 include/linux/sys.h) 。
62 #define __NR_setup      0       /* used only by init, to get system going */
63 #define __NR_exit       1       /* __NR_setup 仅用于初始化，以启动系统 */
64 #define __NR_fork       2
65 #define __NR_read       3
66 #define __NR_write      4
67 #define __NR_open       5
68 #define __NR_close      6
69 #define __NR_waitpid    7
70 #define __NR_creat      8
71 #define __NR_link       9
72 #define __NR_unlink     10
73 #define __NR_execve     11
74 #define __NR_chdir      12
75 #define __NR_time       13
76 #define __NR_mknod      14
77 #define __NR_chmod      15
78 #define __NR_chown      16
79 #define __NR_break      17
80 #define __NR_stat       18
81 #define __NR_lseek      19
82 #define __NR_getpid     20
83 #define __NR_mount      21
84 #define __NR_umount     22
85 #define __NR_setuid     23
86 #define __NR_getuid     24
```

```
 87 #define __NR_stime      25
 88 #define __NR_ptrace     26
 89 #define __NR_alarm      27
 90 #define __NR_fstat      28
 91 #define __NR_pause      29
 92 #define __NR_utime      30
 93 #define __NR_stty       31
 94 #define __NR_gtty       32
 95 #define __NR_access     33
 96 #define __NR_nice       34
 97 #define __NR_ftime      35
 98 #define __NR_sync       36
 99 #define __NR_kill       37
100 #define __NR_rename     38
101 #define __NR_mkdir      39
102 #define __NR_rmdir      40
103 #define __NR_dup        41
104 #define __NR_pipe       42
105 #define __NR_times      43
106 #define __NR_prof       44
107 #define __NR_brk        45
108 #define __NR_setgid     46
109 #define __NR_getgid     47
110 #define __NR_signal     48
111 #define __NR_geteuid    49
112 #define __NR_getegid    50
113 #define __NR_acct       51
114 #define __NR_phys       52
115 #define __NR_lock       53
116 #define __NR_ioctl      54
117 #define __NR_fcntl      55
118 #define __NR_mpx        56
119 #define __NR_setpgid    57
120 #define __NR_ulimit     58
121 #define __NR_uname      59
122 #define __NR_umask      60
123 #define __NR_chroot     61
124 #define __NR_ustat      62
125 #define __NR_dup2       63
126 #define __NR_getppid    64
127 #define __NR_getpgrp    65
128 #define __NR_setsid     66
129 #define __NR_sigaction  67
130 #define __NR_sgetmask   68
131 #define __NR_ssetmask   69
132 #define __NR_setreuid   70
133 #define __NR_setregid   71
134 #define __NR_sigsuspend 72
135 #define __NR_sigpending 73
136 #define __NR_sethostname 74
137 #define __NR_setrlimit  75
138 #define __NR_getrlimit  76
139 #define __NR_getrusage  77
```

```
140 #define __NR_gettimeofday 78
141 #define __NR_settimeofday 79
142 #define __NR_getgroups  80
143 #define __NR_setgroups  81
144 #define __NR_select     82
145 #define __NR_symlink    83
146 #define __NR_lstat      84
147 #define __NR_readlink   85
148 #define __NR_uselib     86
149
    // 以下定义系统调用嵌入式汇编宏函数。
    // 不带参数的系统调用宏函数。type name(void)。
    // %0 - eax(__res), %1 - eax(__NR_##name)。其中 name 是系统调用的名称，与 __NR_ 组合形成上面
    // 的系统调用符号常数，从而用来对系统调用表中函数指针寻址。
    // 返回：如果返回值大于等于 0，则返回该值，否则置出错号 errno，并返回-1。
    // 在宏定义中，若在两个标记符号之间有两个连续的井号'##'，则表示在宏替换时会把这两个标记
    // 符号连接在一起。例如下面第 139 行上的__NR_##name，在替换了参数 name（例如是 fork）之后，
    // 最后在程序中出现的将会是符号__NR_fork。参见《The C Programming Language》附录 A.12.3。
150 #define _syscall0(type,name) \
151 type name(void) \
152 { \
153 long __res; \
154 __asm__ volatile ("int $0x80" \          // 调用系统中断 0x80。
155         : "=a" (__res) \                 // 返回值➔eax(__res)。
156         : "0" (__NR_##name)); \          // 输入为系统中断调用号__NR_name。
157 if (__res >= 0) \                        // 如果返回值>=0，则直接返回该值。
158         return (type) __res; \
159 errno = -__res; \                        // 否则置出错号，并返回-1。
160 return -1; \
161 }
162
    // 有 1 个参数的系统调用宏函数。type name(atype a)
    // %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a)。
163 #define _syscall1(type,name,atype,a) \
164 type name(atype a) \
165 { \
166 long __res; \
167 __asm__ volatile ("int $0x80" \
168         : "=a" (__res) \
169         : "0" (__NR_##name),"b" ((long)(a))); \
170 if (__res >= 0) \
171         return (type) __res; \
172 errno = -__res; \
173 return -1; \
174 }
175
    // 有 2 个参数的系统调用宏函数。type name(atype a, btype b)
    // %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a), %3 - ecx(b)。
176 #define _syscall2(type,name,atype,a,btype,b) \
177 type name(atype a,btype b) \
178 { \
179 long __res; \
180 __asm__ volatile ("int $0x80" \
```

```
181         : "=a" (__res) \
182         : "0" (__NR_##name),"b" ((long)(a)),"c" ((long)(b))); \
183 if (__res >= 0) \
184         return (type) __res; \
185 errno = -__res; \
186 return -1; \
187 }
188
    // 有 3 个参数的系统调用宏函数。type name(atype a, btype b, ctype c)
    // %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a), %3 - ecx(b), %4 - edx(c)。
189 #define _syscall3(type,name,atype,a,btype,b,ctype,c) \
190 type name(atype a,btype b,ctype c) \
191 { \
192 long __res; \
193 __asm__ volatile ("int $0x80" \
194         : "=a" (__res) \
195         : "0" (__NR_##name),"b" ((long)(a)),"c" ((long)(b)),"d" ((long)(c))); \
196 if (__res>=0) \
197         return (type) __res; \
198 errno=-__res; \
199 return -1; \
200 }
201
202 #endif /* __LIBRARY__ */
203
204 extern int errno;                       // 出错号，全局变量。
205
    // 对应各系统调用的函数原型定义。(详细说明参见 include/linux/sys.h )
206 int access(const char * filename, mode_t mode);
207 int acct(const char * filename);
208 int alarm(int sec);
209 int brk(void * end_data_segment);
210 void * sbrk(ptrdiff_t increment);
211 int chdir(const char * filename);
212 int chmod(const char * filename, mode_t mode);
213 int chown(const char * filename, uid_t owner, gid_t group);
214 int chroot(const char * filename);
215 int close(int fildes);
216 int creat(const char * filename, mode_t mode);
217 int dup(int fildes);
218 int execve(const char * filename, char ** argv, char ** envp);
219 int execv(const char * pathname, char ** argv);
220 int execvp(const char * file, char ** argv);
221 int execl(const char * pathname, char * arg0, ...);
222 int execlp(const char * file, char * arg0, ...);
223 int execle(const char * pathname, char * arg0, ...);
    // 函数名前的关键字 volatile 用于告诉编译器 gcc 该函数不会返回。这样可让 gcc 产生更好一
    // 些的代码，更重要的是使用这个关键字可以避免产生某些（未初始化变量的）假警告信息。
    // 等同于 gcc 的函数属性说明：void do_exit(int error_code) __attribute__ ((noreturn));
224 volatile void exit(int status);
225 volatile void _exit(int status);
226 int fcntl(int fildes, int cmd, ...);
227 int fork(void);
```

```
228 int getpid(void);
229 int getuid(void);
230 int geteuid(void);
231 int getgid(void);
232 int getegid(void);
233 int ioctl(int fildes, int cmd, ...);
234 int kill(pid_t pid, int signal);
235 int link(const char * filename1, const char * filename2);
236 int lseek(int fildes, off_t offset, int origin);
237 int mknod(const char * filename, mode_t mode, dev_t dev);
238 int mount(const char * specialfile, const char * dir, int rwflag);
239 int nice(int val);
240 int open(const char * filename, int flag, ...);
241 int pause(void);
242 int pipe(int * fildes);
243 int read(int fildes, char * buf, off_t count);
244 int setpgrp(void);
245 int setpgid(pid_t pid,pid_t pgid);
246 int setuid(uid_t uid);
247 int setgid(gid_t gid);
248 void (*signal(int sig, void (*fn)(int)))(int);
249 int stat(const char * filename, struct stat * stat_buf);
250 int fstat(int fildes, struct stat * stat_buf);
251 int stime(time_t * tptr);
252 int sync(void);
253 time_t time(time_t * tloc);
254 time_t times(struct tms * tbuf);
255 int ulimit(int cmd, long limit);
256 mode_t umask(mode_t mask);
257 int umount(const char * specialfile);
258 int uname(struct utsname * name);
259 int unlink(const char * filename);
260 int ustat(dev_t dev, struct ustat * ubuf);
261 int utime(const char * filename, struct utimbuf * times);
262 pid_t waitpid(pid_t pid,int * wait_stat,int options);
263 pid_t wait(int * wait_stat);
264 int write(int fildes, const char * buf, off_t count);
265 int dup2(int oldfd, int newfd);
266 int getppid(void);
267 pid_t getpgrp(void);
268 pid_t setsid(void);
269 int sethostname(char *name, int len);
270 int setrlimit(int resource, struct rlimit *rlp);
271 int getrlimit(int resource, struct rlimit *rlp);
272 int getrusage(int who, struct rusage *rusage);
273 int gettimeofday(struct timeval *tv, struct timezone *tz);
274 int settimeofday(struct timeval *tv, struct timezone *tz);
275 int getgroups(int gidsetlen, gid_t *gidset);
276 int setgroups(int gidsetlen, gid_t *gidset);
277 int select(int width, fd_set * readfds, fd_set * writefds,
278         fd_set * exceptfds, struct timeval * timeout);
279
280 #endif
```

281

14.13 程序 14-13 linux/include/utime.h

```
 1 #ifndef _UTIME_H
 2 #define _UTIME_H
 3
 4 #include <sys/types.h>  /* I know - shouldn't do this, but .. */
 5                         /* 我知道 - 不应该这样做，但是.. */
 6 struct utimbuf {
 7         time_t actime;          // 文件访问时间。从 1970.1.1:0:0:0 开始的秒数。
 8         time_t modtime;         // 文件修改时间。从 1970.1.1:0:0:0 开始的秒数。
 9 };
10
   // 设置文件访问和修改时间函数。
11 extern int utime(const char *filename, struct utimbuf *times);
12
13 #endif
14
```

14.14 程序 14-14 linux/include/asm/io.h

```
   //// 硬件端口字节输出函数。
   // 参数：value - 欲输出字节；port - 端口。
 1 #define outb(value,port) \
 2 __asm__ ("outb %%al,%%dx"::"a" (value),"d" (port))
 3
 4
   //// 硬件端口字节输入函数。
   // 参数：port - 端口。返回读取的字节。
 5 #define inb(port) ({ \
 6 unsigned char _v; \
 7 __asm__ volatile ("inb %%dx,%%al":"=a" (_v):"d" (port)); \
 8 _v; \
 9 })
10
   //// 带延迟的硬件端口字节输出函数。使用两条跳转语句来延迟一会。
   // 参数：value - 欲输出字节；port - 端口。
11 #define outb_p(value,port) \
12 __asm__ ("outb %%al,%%dx\n" \
13                 "\tjmp 1f\n" \                  // 向前跳转到标号 1 处（即下一条语句处）。
14                 "1:\tjmp 1f\n" \                // 向前跳转到标号 1 处。
15                 "1:"::"a" (value),"d" (port))
16
   //// 带延迟的硬件端口字节输入函数。使用两条跳转语句来延迟一会。
   // 参数：port - 端口。返回读取的字节。
17 #define inb_p(port) ({ \
18 unsigned char _v; \
19 __asm__ volatile ("inb %%dx,%%al\n" \
20         "\tjmp 1f\n" \                          // 向前跳转到标号 1 处（即下一条语句处）。
21         "1:\tjmp 1f\n" \                        // 向前跳转到标号 1 处。
22         "1:":"=a" (_v):"d" (port)); \
23 _v; \
24 })
25
```

14.15 程序 14-15 linux/include/asm/memory.h

```
1  /*
2   * NOTE!!! memcpy(dest,src,n) assumes ds=es=normal data segment. This
3   * goes for all kernel functions (ds=es=kernel space, fs=local data,
4   * gs=null), as well as for all well-behaving user programs (ds=es=
5   * user data space). This is NOT a bug, as any user program that changes
6   * es deserves to die if it isn't careful.
7   */
    /*
     * 注意!!!memcpy(dest,src,n)假设段寄存器 ds=es=通常数据段。在内核中使用的
     * 所有函数都基于该假设（ds=es=内核空间，fs=局部数据空间，gs=null），具有良好
     * 行为的应用程序也是这样（ds=es=用户数据空间）。如果任何用户程序随意改动了
     * es 寄存器而出错，则并不是由于系统程序错误造成的。
     */
    //// 内存块复制。从源地址 src 处开始复制 n 个字节到目的地址 dest 处。
    // 参数：dest - 复制的目的地址，src - 复制的源地址，n - 复制字节数。
    // %0 - edi(目的地址 dest)，%1 - esi(源地址 src)，%2 - ecx(字节数 n)，
8   #define memcpy(dest,src,n) ({ \
9   void * _res = dest; \
10  __asm__ ("cld;rep;movsb" \                  // 从 ds:[esi]复制到 es:[edi]，并且 esi++，edi++。
                                                // 共复制 ecx(n)字节。
11          ::"D" ((long)(_res)),"S" ((long)(src)),"c" ((long) (n)) \
12          :"di","si","cx"); \
13  _res; \
14  })
15
```

14.16 程序 14-16 linux/include/asm/segment.h

```
   //// 读取 fs 段中指定地址处的字节。
   // 参数：addr - 指定的内存地址。
   // %0 - (返回的字节_v)；%1 - (内存地址 addr)。
   // 返回：返回内存 fs:[addr]处的字节。
   // 第 3 行定义了一个寄存器变量_v，该变量将被保存在一个寄存器中，以便于高效访问和操作。
 1 extern inline unsigned char get_fs_byte(const char * addr)
 2 {
 3         unsigned register char _v;
 4
 5         __asm__ ("movb %%fs:%1,%0":"=r" (_v):"m" (*addr));
 6         return _v;
 7 }
 8
   //// 读取 fs 段中指定地址处的字。
   // 参数：addr - 指定的内存地址。
   // %0 - (返回的字_v)；%1 - (内存地址 addr)。
   // 返回：返回内存 fs:[addr]处的字。
 9 extern inline unsigned short get_fs_word(const unsigned short *addr)
10 {
11         unsigned short _v;
12
13         __asm__ ("movw %%fs:%1,%0":"=r" (_v):"m" (*addr));
14         return _v;
15 }
16
   //// 读取 fs 段中指定地址处的长字(4 字节)。
   // 参数：addr - 指定的内存地址。
   // %0 - (返回的长字_v)；%1 - (内存地址 addr)。
   // 返回：返回内存 fs:[addr]处的长字。
17 extern inline unsigned long get_fs_long(const unsigned long *addr)
18 {
19         unsigned long _v;
20
21         __asm__ ("movl %%fs:%1,%0":"=r" (_v):"m" (*addr)); \
22         return _v;
23 }
24
   //// 将一字节存放在 fs 段中指定内存地址处。
   // 参数：val - 字节值；addr - 内存地址。
   // %0 - 寄存器(字节值 val)；%1 - (内存地址 addr)。
25 extern inline void put_fs_byte(char val,char *addr)
26 {
27 __asm__ ("movb %0,%%fs:%1"::"r" (val),"m" (*addr));
28 }
29
   //// 将一字存放在 fs 段中指定内存地址处。
   // 参数：val - 字值；addr - 内存地址。
   // %0 - 寄存器(字值 val)；%1 - (内存地址 addr)。
30 extern inline void put_fs_word(short val,short * addr)
```

```
31 {
32 __asm__ ("movw %0,%%fs:%1"::"r" (val),"m" (*addr));
33 }
34
   //// 将一长字存放在 fs 段中指定内存地址处。
   // 参数：val - 长字值；addr - 内存地址。
   // %0 - 寄存器(长字值 val)；%1 - (内存地址 addr)。
35 extern inline void put_fs_long(unsigned long val,unsigned long * addr)
36 {
37 __asm__ ("movl %0,%%fs:%1"::"r" (val),"m" (*addr));
38 }
39
40 /*
41  * Someone who knows GNU asm better than I should double check the followig.
42  * It seems to work, but I don't know if I'm doing something subtly wrong.
43  * --- TYT, 11/24/91
44  * [ nothing wrong here, Linus ]
45  */
   /*
    * 比我更懂 GNU 汇编的人应该仔细检查下面的代码。这些代码能使用，但我不知道是否
    * 含有一些小错误。
    * --- TYT，1991 年 11 月 24 日
    * [ 这些代码没有错误，Linus ]
    */
46
   //// 取 fs 段寄存器值(选择符)。
   // 返回：fs 段寄存器值。
47 extern inline unsigned long get_fs()
48 {
49         unsigned short _v;
50         __asm__("mov %%fs,%%ax":"=a" (_v):);
51         return _v;
52 }
53
   //// 取 ds 段寄存器值。
   // 返回：ds 段寄存器值。
54 extern inline unsigned long get_ds()
55 {
56         unsigned short _v;
57         __asm__("mov %%ds,%%ax":"=a" (_v):);
58         return _v;
59 }
60
   //// 设置 fs 段寄存器。
   // 参数：val - 段值（选择符）。
61 extern inline void set_fs(unsigned long val)
62 {
63         __asm__("mov %0,%%fs"::"a" ((unsigned short) val));
64 }
65
66
```

14.17 程序 14-17 linux/include/asm/system.h

```
    //// 移动到用户模式运行。
    // 该函数利用 iret 指令实现从内核模式移动到初始任务 0 中去执行。
 1  #define move_to_user_mode() \
 2  __asm__ ("movl %%esp,%%eax\n\t" \          // 保存堆栈指针 esp 到 eax 寄存器中。
 3          "pushl $0x17\n\t" \                // 首先将堆栈段选择符(SS)入栈。
 4          "pushl %%eax\n\t" \                // 然后将保存的堆栈指针值(esp)入栈。
 5          "pushfl\n\t" \                     // 将标志寄存器(eflags)内容入栈。
 6          "pushl $0x0f\n\t" \                // 将 Task0 代码段选择符(cs)入栈。
 7          "pushl $1f\n\t" \                  // 将下面标号 1 的偏移地址(eip)入栈。
 8          "iret\n" \                         // 执行中断返回指令，则会跳转到下面标号 1 处。
 9          "1:\tmovl $0x17,%%eax\n\t" \       // 此时开始执行任务 0，
10          "movw %%ax,%%ds\n\t" \             // 初始化段寄存器指向本局部表的数据段。
11          "movw %%ax,%%es\n\t" \
12          "movw %%ax,%%fs\n\t" \
13          "movw %%ax,%%gs" \
14          :::"ax")
15
16  #define sti() __asm__ ("sti"::)            // 开中断嵌入汇编宏函数。
17  #define cli() __asm__ ("cli"::)            // 关中断。
18  #define nop() __asm__ ("nop"::)            // 空操作。
19
20  #define iret() __asm__ ("iret"::)          // 中断返回。
21
    //// 设置门描述符宏。
    // 根据参数中的中断或异常处理过程地址 addr、门描述符类型 type 和特权级信息 dpl，设置位于
    // 地址 gate_addr 处的门描述符。（注意：下面“偏移”值是相对于内核代码或数据段来说的）。
    // 参数：gate_addr -描述符地址；type -描述符类型域值；dpl -描述符特权级；addr -偏移地址。
    // %0 - (由 dpl,type 组合成的类型标志字)；%1 - (描述符低 4 字节地址)；
    // %2 - (描述符高 4 字节地址)；%3 - edx(程序偏移地址 addr)；%4 - eax(高字中含有段选择符 0x8)。
22  #define _set_gate(gate_addr,type,dpl,addr) \
23  __asm__ ("movw %%dx,%%ax\n\t" \            // 将偏移地址低字与选择符组合成描述符低 4 字节(eax)。
24          "movw %0,%%dx\n\t" \               // 将类型标志字与偏移高字组合成描述符高 4 字节(edx)。
25          "movl %%eax,%1\n\t" \              // 分别设置门描述符的低 4 字节和高 4 字节。
26          "movl %%edx,%2" \
27          : \
28          : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
29          "o" (*((char *) (gate_addr))), \
30          "o" (*(4+(char *) (gate_addr))), \
31          "d" ((char *) (addr)),"a" (0x00080000))
32
    //// 设置中断门函数（自动屏蔽随后的中断）。
    // 参数：n - 中断号；addr - 中断程序偏移地址。
    // &idt[n]是中断描述符表中中断号 n 对应项的偏移值；中断描述符的类型是 14，特权级是 0。
33  #define set_intr_gate(n,addr) \
34          _set_gate(&idt[n],14,0,addr)
35
    //// 设置陷阱门函数。
    // 参数：n - 中断号；addr - 中断程序偏移地址。
    // &idt[n]是中断描述符表中中断号 n 对应项的偏移值；中断描述符的类型是 15，特权级是 0。
```

```
36 #define set_trap_gate(n,addr) \
37         _set_gate(&idt[n],15,0,addr)
38
   //// 设置系统陷阱门函数。
   // 上面 set_trap_gate()设置的描述符的特权级为 0，而这里是 3。因此 set_system_gate()设置的
   // 中断处理过程能够被所有程序执行。例如单步调试、溢出出错和边界超出出错处理。
   // 参数：n - 中断号；addr - 中断程序偏移地址。
   // &idt[n]是中断描述符表中中断号 n 对应项的偏移值；中断描述符的类型是 15，特权级是 3。
39 #define set_system_gate(n,addr) \
40         _set_gate(&idt[n],15,3,addr)
41
   //// 设置段描述符函数（内核中没有用到）。
   // 参数：gate_addr -描述符地址；type -描述符中类型域值；dpl -描述符特权层值；
   // base - 段的基地址；limit - 段限长。
   // 请参见段描述符的格式。注意，这里赋值对象弄反了。43 行应该是 *((gate_addr)+1)，而
   // 49 行才是 *(gate_addr)。不过内核代码中没有用到这个宏，所以 Linus 没有察觉 :-)
42 #define _set_seg_desc(gate_addr,type,dpl,base,limit) {\
43         *(gate_addr) = ((base) & 0xff000000) | \              // 描述符低 4 字节。
44                 (((base) & 0x00ff0000)>>16) | \
45                 ((limit) & 0xf0000) | \
46                 ((dpl)<<13) | \
47                 (0x00408000) | \
48                 ((type)<<8); \
49         *((gate_addr)+1) = (((base) & 0x0000ffff)<<16) | \    // 描述符高 4 字节。
50                 ((limit) & 0x0ffff); }
51
   //// 在全局表中设置任务状态段/局部表描述符。状态段和局部表段的长度均被设置成 104 字节。
   // 参数：n - 在全局表中描述符项 n 所对应的地址；addr - 状态段/局部表所在内存的基地址。
   //       type - 描述符中的标志类型字节。
   // %0 - eax(地址 addr)；%1 - (描述符项 n 的地址)；%2 - (描述符项 n 的地址偏移 2 处)；
   // %3 - (描述符项 n 的地址偏移 4 处)；%4 - (描述符项 n 的地址偏移 5 处)；
   // %5 - (描述符项 n 的地址偏移 6 处)；%6 - (描述符项 n 的地址偏移 7 处)；
52 #define _set_tssldt_desc(n,addr,type) \
53 __asm__ ("movw $104,%1\n\t" \                 // 将 TSS（或 LDT）长度放入描述符长度域(第 0-1 字节)。
54         "movw %%ax,%2\n\t" \                  // 将基地址的低字放入描述符第 2-3 字节。
55         "rorl $16,%%eax\n\t" \                // 将基地址高字右循环移入 ax 中（低字则进入高字处）。
56         "movb %%al,%3\n\t" \                  // 将基地址高字中低字节移入描述符第 4 字节。
57         "movb $" type ",%4\n\t" \             // 将标志类型字节移入描述符的第 5 字节。
58         "movb $0x00,%5\n\t" \                 // 描述符的第 6 字节置 0。
59         "movb %%ah,%6\n\t" \                  // 将基地址高字中高字节移入描述符第 7 字节。
60         "rorl $16,%%eax" \                    // 再右循环 16 比特，eax 恢复原值。
61         ::"a" (addr), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)), \
62          "m" (*(n+5)), "m" (*(n+6)), "m" (*(n+7)) \
63         )
64
   //// 在全局表中设置任务状态段描述符。
   // n - 是该描述符的指针；addr - 是描述符项中段的基地址值。任务状态段描述符的类型是 0x89。
65 #define set_tss_desc(n,addr) _set_tssldt_desc(((char *) (n)),addr,"0x89")
   //// 在全局表中设置局部表描述符。
   // n - 是该描述符的指针；addr - 是描述符项中段的基地址值。局部表段描述符的类型是 0x82。
66 #define set_ldt_desc(n,addr) _set_tssldt_desc(((char *) (n)),addr,"0x82")
67
```

14.18 程序 14-18 linux/include/linux/config.h

```
 1 #ifndef _CONFIG_H
 2 #define _CONFIG_H
 3
 4 /*
 5  * Defines for what uname() should return
 6  */
   /*
    * 定义 uname()函数应该返回的值。
    */
 7 #define UTS_SYSNAME "Linux"
 8 #define UTS_NODENAME "(none)"  /* set by sethostname() */
 9 #define UTS_RELEASE ""         /* patchlevel */
10 #define UTS_VERSION "0.12"
11 #define UTS_MACHINE "i386"     /* hardware type */
12
13 /* Don't touch these, unless you really know what your doing. */
   /* 请不要随意修改下面定义值，除非你知道自己正在干什么。 */
14 #define DEF_INITSEG     0x9000          // 引导扇区程序将被移动到的段值。
15 #define DEF_SYSSEG      0x1000          // 引导扇区程序把系统模块加载到内存的段值。
16 #define DEF_SETUPSEG    0x9020          // setup 程序所处内存段位置。
17 #define DEF_SYSSIZE     0x3000          // 内核系统模块默认最大节数（16 字节=1 节）。
18
19 /*
20  * The root-device is no longer hard-coded. You can change the default
21  * root-device by changing the line ROOT_DEV = XXX in boot/bootsect.s
22  */
   /*
    * 根文件系统设备已不再是硬编码的了。通过修改 boot/bootsect.s 文件中行
    * ROOT_DEV = XXX，你可以改变根设备的默认设置值。
    */
23
24 /*
25  * The keyboard is now defined in kernel/chr_dev/keyboard.S
26  */
   /*
    * 现在键盘类型被放在 kernel/chr_dev/keyboard.S 程序中定义。
    */
27
28 /*
29  * Normally, Linux can get the drive parameters from the BIOS at
30  * startup, but if this for some unfathomable reason fails, you'd
31  * be left stranded. For this case, you can define HD_TYPE, which
32  * contains all necessary info on your harddisk.
33  *
34  * The HD_TYPE macro should look like this:
35  *
36  * #define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}
37  *
38  * In case of two harddisks, the info should be sepatated by
```

```
 * commas:
 *
 * #define HD_TYPE { h,s,c,wpcom,lz,ctl },{ h,s,c,wpcom,lz,ctl }
 */
/*
 * 通常，Linux 能够在启动时从 BIOS 中获取驱动器德参数，但是若由于未知原因
 * 而没有得到这些参数时，会使程序束手无策。对于这种情况，你可以定义 HD_TYPE，
 * 其中包括硬盘的所有信息。
 *
 * HD_TYPE 宏应该象下面这样的形式：
 *
 * #define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}
 *
 * 对于有两个硬盘的情况，参数信息需用逗号分开：
 *
 * #define HD_TYPE { h,s,c,wpcom,lz,ctl }, {h,s,c,wpcom,lz,ctl }
 */
/*
 This is an example, two drives, first is type 2, second is type 3:

#define HD_TYPE { 4,17,615,300,615,8 }, { 6,17,615,300,615,0 }

 NOTE: ctl is 0 for all drives with heads<=8, and ctl=8 for drives
 with more than 8 heads.

 If you want the BIOS to tell what kind of drive you have, just
 leave HD_TYPE undefined. This is the normal thing to do.
*/
/*
 * 下面是一个例子，两个硬盘，第 1 个是类型 2，第 2 个是类型 3：
 *
 * #define HD_TYPE { 4,17,615,300,615,8 }, {6,17,615,300,615,0 }
 *
 * 注意：对应所有硬盘，若其磁头数<=8，则 ctl 等于 0，若磁头数多于 8 个，
 * 则 ctl=8。
 *
 * 如果你想让 BIOS 给出硬盘的类型，那么只需不定义 HD_TYPE。这是默认操作。
 */

#endif

```

14.19 程序 14-19 linux/include/linux/fdreg.h

```
 1 /*
 2  * This file contains some defines for the floppy disk controller.
 3  * Various sources. Mostly "IBM Microcomputers: A Programmers
 4  * Handbook", Sanches and Canton.
 5  */
   /*
    * 该文件中含有一些软盘控制器的一些定义。这些信息有多处来源，大多数取自 Sanches 和 Canton
    * 编著的"IBM 微型计算机：程序员手册"一书。
    */
 6 #ifndef _FDREG_H      // 该定义用来排除代码中重复包含此头文件。
 7 #define _FDREG_H
 8
   // 一些软盘类型函数的原型说明。
 9 extern int ticks_to_floppy_on(unsigned int nr);
10 extern void floppy_on(unsigned int nr);
11 extern void floppy_off(unsigned int nr);
12 extern void floppy_select(unsigned int nr);
13 extern void floppy_deselect(unsigned int nr);
14
   // 下面是有关软盘控制器一些端口和符号的定义。
15 /* Fd controller regs. S&C, about page 340 */
   /* 软盘控制器(FDC)寄存器端口。摘自 S&C 书中约 340 页 */
16 #define FD_STATUS       0x3f4           // 主状态寄存器端口。
17 #define FD_DATA         0x3f5           // 数据端口。
18 #define FD_DOR          0x3f2           /* Digital Output Register */
                                           // 数字输出寄存器（也称为数字控制寄存器）。
19 #define FD_DIR          0x3f7           /* Digital Input Register (read) */
                                           // 数字输入寄存器。
20 #define FD_DCR          0x3f7           /* Diskette Control Register (write)*/
                                           // 数据传输率控制寄存器。
21
22 /* Bits of main status register */
   /* 主状态寄存器各比特位的含义 */
23 #define STATUS_BUSYMASK 0x0F            /* drive busy mask */
                                           // 驱动器忙位（每位对应一个驱动器）。
24 #define STATUS_BUSY     0x10            /* FDC busy */
                                           // 软盘控制器忙。
25 #define STATUS_DMA      0x20            /* 0- DMA mode */
                                           // 0 - 为 DMA 数据传输模式，1 - 为非 DMA 模式。
26 #define STATUS_DIR      0x40            /* 0- cpu->fdc */
```

```
                                        // 传输方向：0 - CPU → fdc，1 - 相反。
27 #define STATUS_READY    0x80        /* Data reg ready */
                                        // 数据寄存器就绪位。
28
29 /* Bits of FD_ST0 */
   /*状态字节 0（ST0）各比特位的含义 */
30 #define ST0_DS          0x03        /* drive select mask */
                                        // 驱动器选择号（发生中断时驱动器号）。
31 #define ST0_HA          0x04        /* Head (Address) */
                                        // 磁头号。
32 #define ST0_NR          0x08        /* Not Ready */
                                        // 磁盘驱动器未准备好。
33 #define ST0_ECE         0x10        /* Equipment chech error */
                                        // 设备检测出错（零磁道校准出错）。
34 #define ST0_SE          0x20        /* Seek end */
                                        // 寻道或重新校正操作执行结束。
35 #define ST0_INTR        0xC0        /* Interrupt code mask */
                                        // 中断代码位（中断原因），00 - 命令正常结束；
                                    // 01 - 命令异常结束；10 - 命令无效；11 - FDD 就绪状态改变。
36
37 /* Bits of FD_ST1 */
   /*状态字节 1（ST1）各比特位的含义 */
38 #define ST1_MAM         0x01        /* Missing Address Mark */
                                        // 未找到地址标志(ID AM)。
39 #define ST1_WP          0x02        /* Write Protect */
                                        // 写保护。
40 #define ST1_ND          0x04        /* No Data - unreadable */
                                        // 未找到指定的扇区。
41 #define ST1_OR          0x10        /* OverRun */
                                        // 数据传输超时（DMA 控制器故障）。
42 #define ST1_CRC         0x20        /* CRC error in data or addr */
                                        // CRC 检验出错。
43 #define ST1_EOC         0x80        /* End Of Cylinder */
                                        // 访问超过一个磁道上的最大扇区号。
44
45 /* Bits of FD_ST2 */
   /*状态字节 2（ST2）各比特位的含义 */
46 #define ST2_MAM         0x01        /* Missing Addess Mark (again) */
                                        // 未找到数据地址标志。
47 #define ST2_BC          0x02        /* Bad Cylinder */
                                        // 磁道坏。
48 #define ST2_SNS         0x04        /* Scan Not Satisfied */
                                        // 检索（扫描）条件不满足。
49 #define ST2_SEH         0x08        /* Scan Equal Hit */
```

```
                                        // 检索条件满足。
50 #define ST2_WC          0x10        /* Wrong Cylinder */
                                        // 磁道（柱面）号不符。
51 #define ST2_CRC         0x20        /* CRC error in data field */
                                        // 数据场 CRC 校验错。
52 #define ST2_CM          0x40        /* Control Mark = deleted */
                                        // 读数据遇到删除标志。
53
54 /* Bits of FD_ST3 */
   /*状态字节 3（ST3）各比特位的含义 */
55 #define ST3_HA          0x04        /* Head (Address) */
                                        // 磁头号。
56 #define ST3_TZ          0x10        /* Track Zero signal (1=track 0) */
                                        // 零磁道信号。
57 #define ST3_WP          0x40        /* Write Protect */
                                        // 写保护。
58
59 /* Values for FD_COMMAND */
   /* 软盘命令码 */
60 #define FD_RECALIBRATE  0x07        /* move to track 0 */
                                        // 重新校正(磁头退到零磁道)。
61 #define FD_SEEK         0x0F        /* seek track */
                                        // 磁头寻道。
62 #define FD_READ         0xE6        /* read with MT, MFM, SKip deleted */
                                        // 读数据（MT 多磁道操作，MFM 格式，跳过删除数据）。
63 #define FD_WRITE        0xC5        /* write with MT, MFM */
                                        // 写数据（MT，MFM）。
64 #define FD_SENSEI       0x08        /* Sense Interrupt Status */
                                        // 检测中断状态。
65 #define FD_SPECIFY      0x03        /* specify HUT etc */
                                        // 设定驱动器参数（步进速率、磁头卸载时间等）。
66
67 /* DMA commands */
   /* DMA 命令 */
68 #define DMA_READ        0x46        // DMA 读盘，DMA 方式字（送 DMA 端口 12，11）。
69 #define DMA_WRITE       0x4A        // DMA 写盘，DMA 方式字。
70
71 #endif
72
```

14.20 程序 14-20 linux/include/linux/fs.h

```
 1 /*
 2  * This file has definitions for some important file table
 3  * structures etc.
 4  */
   /*
    * 本文件含有某些重要文件表结构的定义等。
    */
 5
 6 #ifndef _FS_H
 7 #define _FS_H
 8
 9 #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
10
11 /* devices are as follows: (same as minix, so we can use the minix
12  * file system. These are major numbers.)
13  *
14  * 0 - unused (nodev)
15  * 1 - /dev/mem
16  * 2 - /dev/fd
17  * 3 - /dev/hd
18  * 4 - /dev/ttyx
19  * 5 - /dev/tty
20  * 6 - /dev/lp
21  * 7 - unnamed pipes
22  */
   /*
    * 系统所含的设备如下：（与 minix 系统的一样，所以我们可以使用 minix 的
    * 文件系统。以下这些是主设备号。）
    *
    * 0 - 没有用到（nodev）
    * 1 - /dev/mem          内存设备。
    * 2 - /dev/fd           软盘设备。
    * 3 - /dev/hd           硬盘设备。
    * 4 - /dev/ttyx         tty 串行终端设备。
    * 5 - /dev/tty          tty 终端设备。
    * 6 - /dev/lp           打印设备。
    * 7 - unnamed pipes     没有命名的管道。
    */
23
24 #define IS_SEEKABLE(x) ((x)>=1 && (x)<=3)      // 判断设备是否是可以寻找定位的。
25
26 #define READ 0
27 #define WRITE 1
28 #define READA 2         /* read-ahead - don't pause */
29 #define WRITEA 3        /* "write-ahead" - silly, but somewhat useful */
30
31 void buffer_init(long buffer_end);              // 高速缓冲区初始化函数。
32
33 #define MAJOR(a) (((unsigned)(a))>>8)           // 取高字节（主设备号）。
```

```
34 #define MINOR(a) ((a)&0xff)                     // 取低字节（次设备号）。
35
36 #define NAME_LEN 14                             // 名字长度值。
37 #define ROOT_INO 1                              // 根 i 节点。
38
39 #define I_MAP_SLOTS 8                           // i 节点位图槽数。
40 #define Z_MAP_SLOTS 8                           // 逻辑块（区段块）位图槽数。
41 #define SUPER_MAGIC 0x137F                      // 文件系统魔数。
42
43 #define NR_OPEN 20                              // 进程最多打开文件数。
44 #define NR_INODE 32                             // 系统同时最多使用 I 节点个数。
45 #define NR_FILE 64                              // 系统最多文件个数（文件数组项数）。
46 #define NR_SUPER 8                              // 系统所含超级块个数（超级块数组项数）。
47 #define NR_HASH 307                             // 缓冲区 Hash 表数组项数值。
48 #define NR_BUFFERS nr_buffers                   // 系统所含缓冲块个数。初始化后不再改变。
49 #define BLOCK_SIZE 1024                         // 数据块长度（字节值）。
50 #define BLOCK_SIZE_BITS 10                      // 数据块长度所占比特位数。
51 #ifndef NULL
52 #define NULL ((void *) 0)
53 #endif
54
   // 每个逻辑块可存放的 i 节点数。
55 #define INODES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct d_inode)))
   // 每个逻辑块可存放的目录项数。
56 #define DIR_ENTRIES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct dir_entry)))
57
   // 管道头、管道尾、管道大小、管道空？、管道满？、管道头指针递增。
58 #define PIPE_READ_WAIT(inode) ((inode).i_wait)
59 #define PIPE_WRITE_WAIT(inode) ((inode).i_wait2)
60 #define PIPE_HEAD(inode) ((inode).i_zone[0])
61 #define PIPE_TAIL(inode) ((inode).i_zone[1])
62 #define PIPE_SIZE(inode) ((PIPE_HEAD(inode)-PIPE_TAIL(inode))&(PAGE_SIZE-1))
63 #define PIPE_EMPTY(inode) (PIPE_HEAD(inode)==PIPE_TAIL(inode))
64 #define PIPE_FULL(inode) (PIPE_SIZE(inode)==(PAGE_SIZE-1))
65
66 #define NIL_FILP        ((struct file *)0)    // 空文件结构指针。
67 #define SEL_IN          1
68 #define SEL_OUT         2
69 #define SEL_EX          4
70
71 typedef char buffer_block[BLOCK_SIZE];        // 块缓冲区。
72
   // 缓冲块头数据结构。（极为重要！！！）
   // 在程序中常用 bh 来表示 buffer_head 类型的缩写。
73 struct buffer_head {
74         char * b_data;                  /* pointer to data block (1024 bytes) */ //指针。
75         unsigned long b_blocknr;        /* block number */      // 块号。
76         unsigned short b_dev;           /* device (0 = free) */ // 数据源的设备号。
77         unsigned char b_uptodate;       // 更新标志：表示数据是否已更新。
78         unsigned char b_dirt;           /* 0-clean,1-dirty */ //修改标志:0 未修改,1 已修改.
79         unsigned char b_count;          /* users using this block */  // 使用的用户数。
80         unsigned char b_lock;           /* 0 - ok, 1 -locked */ // 缓冲区是否被锁定。
81         struct task_struct * b_wait;    // 指向等待该缓冲区解锁的任务。
```

```
82          struct buffer_head * b_prev;    // hash 队列上前一块（这四个指针用于缓冲区的管理）。
83          struct buffer_head * b_next;    // hash 队列上下一块。
84          struct buffer_head * b_prev_free;   // 空闲表上前一块。
85          struct buffer_head * b_next_free;   // 空闲表上下一块。
86  };
87
    // 磁盘上的索引节点(i 节点)数据结构。
88  struct d_inode {
89          unsigned short i_mode;          // 文件类型和属性(rwx 位)。
90          unsigned short i_uid;           // 用户 id（文件拥有者标识符）。
91          unsigned long i_size;           // 文件大小（字节数）。
92          unsigned long i_time;           // 修改时间（自 1970.1.1:0 算起，秒）。
93          unsigned char i_gid;            // 组 id(文件拥有者所在的组)。
94          unsigned char i_nlinks;         // 链接数（多少个文件目录项指向该 i 节点）。
95          unsigned short i_zone[9];       // 直接(0-6)、间接(7)或双重间接(8)逻辑块号。
                                            // zone 是区的意思，可译成区段，或逻辑块。
96  };
97
    // 这是在内存中的 i 节点结构。前 7 项与 d_inode 完全一样。
98  struct m_inode {
99          unsigned short i_mode;          // 文件类型和属性(rwx 位)。
100         unsigned short i_uid;           // 用户 id（文件拥有者标识符）。
101         unsigned long i_size;           // 文件大小（字节数）。
102         unsigned long i_mtime;          // 修改时间（自 1970.1.1:0 算起，秒）。
103         unsigned char i_gid;            // 组 id(文件拥有者所在的组)。
104         unsigned char i_nlinks;         // 文件目录项链接数。
105         unsigned short i_zone[9];       // 直接(0-6)、间接(7)或双重间接(8)逻辑块号。
106 /* these are in memory also */
107         struct task_struct * i_wait;    // 等待该 i 节点的进程。
108         struct task_struct * i_wait2;    /* for pipes */
109         unsigned long i_atime;          // 最后访问时间。
110         unsigned long i_ctime;          // i 节点自身修改时间。
111         unsigned short i_dev;           // i 节点所在的设备号。
112         unsigned short i_num;           // i 节点号。
113         unsigned short i_count;         // i 节点被使用的次数，0 表示该 i 节点空闲。
114         unsigned char i_lock;           // 锁定标志。
115         unsigned char i_dirt;           // 已修改(脏)标志。
116         unsigned char i_pipe;           // 管道标志。
117         unsigned char i_mount;          // 安装标志。
118         unsigned char i_seek;           // 搜寻标志(lseek 时)。
119         unsigned char i_update;         // 更新标志。
120 };
121
    // 文件结构（用于在文件句柄与 i 节点之间建立关系）
122 struct file {
123         unsigned short f_mode;          // 文件操作模式（RW 位）
124         unsigned short f_flags;         // 文件打开和控制的标志。
125         unsigned short f_count;         // 对应文件引用计数值。
126         struct m_inode * f_inode;       // 指向对应 i 节点。
127         off_t f_pos;                    // 文件位置（读写偏移值）。
128 };
129
    // 内存中磁盘超级块结构。
```

```
130 struct super_block {
131         unsigned short s_ninodes;       // 节点数。
132         unsigned short s_nzones;        // 逻辑块数。
133         unsigned short s_imap_blocks;   // i 节点位图所占用的数据块数。
134         unsigned short s_zmap_blocks;   // 逻辑块位图所占用的数据块数。
135         unsigned short s_firstdatazone; // 第一个数据逻辑块号。
136         unsigned short s_log_zone_size; // log(数据块数/逻辑块)。（以 2 为底）。
137         unsigned long s_max_size;       // 文件最大长度。
138         unsigned short s_magic;         // 文件系统魔数。
139 /* These are only in memory */
140         struct buffer_head * s_imap[8]; // i 节点位图缓冲块指针数组(占用 8 块，可表示 64M)。
141         struct buffer_head * s_zmap[8]; // 逻辑块位图缓冲块指针数组（占用 8 块）。
142         unsigned short s_dev;           // 超级块所在的设备号。
143         struct m_inode * s_isup;        // 被安装的文件系统根目录的 i 节点。(isup-super i)
144         struct m_inode * s_imount;      // 被安装到的 i 节点。
145         unsigned long s_time;           // 修改时间。
146         struct task_struct * s_wait;    // 等待该超级块的进程。
147         unsigned char s_lock;           // 被锁定标志。
148         unsigned char s_rd_only;        // 只读标志。
149         unsigned char s_dirt;           // 已修改(脏)标志。
150 };
151
    // 磁盘上超级块结构。上面 125-132 行完全一样。
152 struct d_super_block {
153         unsigned short s_ninodes;       // 节点数。
154         unsigned short s_nzones;        // 逻辑块数。
155         unsigned short s_imap_blocks;   // i 节点位图所占用的数据块数。
156         unsigned short s_zmap_blocks;   // 逻辑块位图所占用的数据块数。
157         unsigned short s_firstdatazone; // 第一个数据逻辑块。
158         unsigned short s_log_zone_size; // log(数据块数/逻辑块)。（以 2 为底）。
159         unsigned long s_max_size;       // 文件最大长度。
160         unsigned short s_magic;         // 文件系统魔数。
161 };
162
    // 文件目录项结构。
163 struct dir_entry {
164         unsigned short inode;           // i 节点号。
165         char name[NAME_LEN];            // 文件名，长度 NAME_LEN=14。
166 };
167
168 extern struct m_inode inode_table[NR_INODE];     // 定义 i 节点表数组（32 项）。
169 extern struct file file_table[NR_FILE];          // 文件表数组（64 项）。
170 extern struct super_block super_block[NR_SUPER]; // 超级块数组（8 项）。
171 extern struct buffer_head * start_buffer;        // 缓冲区起始内存位置。
172 extern int nr_buffers;                           // 缓冲块数。
173
    //// 磁盘操作函数原型。
    // 检测驱动器中软盘是否改变。
174 extern void check_disk_change(int dev);
    // 检测指定软驱中软盘更换情况。如果软盘更换了则返回 1，否则返回 0。
175 extern int floppy_change(unsigned int nr);
    // 设置启动指定驱动器所需等待的时间（设置等待定时器）。
176 extern int ticks_to_floppy_on(unsigned int dev);
```

```
    // 启动指定驱动器。
177 extern void floppy_on(unsigned int dev);
    // 关闭指定的软盘驱动器。
178 extern void floppy_off(unsigned int dev);

    //// 以下是文件系统操作管理用的函数原型。
    // 将 i 节点指定的文件截为 0。
179 extern void truncate(struct m_inode * inode);
    // 刷新 i 节点信息。
180 extern void sync_inodes(void);
    // 等待指定的 i 节点。
181 extern void wait_on(struct m_inode * inode);
    // 逻辑块(区段，磁盘块)位图操作。取数据块 block 在设备上对应的逻辑块号。
182 extern int bmap(struct m_inode * inode,int block);
    // 创建数据块 block 在设备上对应的逻辑块，并返回在设备上的逻辑块号。
183 extern int create_block(struct m_inode * inode,int block);
    // 获取指定路径名的 i 节点号。
184 extern struct m_inode * namei(const char * pathname);
    // 取指定路径名的 i 节点，不跟随符号链接。
185 extern struct m_inode * lnamei(const char * pathname);
    // 根据路径名为打开文件操作作准备。
186 extern int open_namei(const char * pathname, int flag, int mode,
187         struct m_inode ** res_inode);
    // 释放一个 i 节点(回写入设备)。
188 extern void iput(struct m_inode * inode);
    // 从设备读取指定节点号的一个 i 节点。
189 extern struct m_inode * iget(int dev,int nr);
    // 从 i 节点表(inode_table)中获取一个空闲 i 节点项。
190 extern struct m_inode * get_empty_inode(void);
    // 获取（申请一）管道节点。返回为 i 节点指针（如果是 NULL 则失败）。
191 extern struct m_inode * get_pipe_inode(void);
    // 在哈希表中查找指定的数据块。返回找到块的缓冲头指针。
192 extern struct buffer_head * get_hash_table(int dev, int block);
    // 从设备读取指定块（首先会在 hash 表中查找）。
193 extern struct buffer_head * getblk(int dev, int block);
    // 读/写数据块。
194 extern void ll_rw_block(int rw, struct buffer_head * bh);
    // 读/写数据页面，即每次 4 块数据块。
195 extern void ll_rw_page(int rw, int dev, int nr, char * buffer);
    // 释放指定缓冲块。
196 extern void brelse(struct buffer_head * buf);
    // 读取指定的数据块。
197 extern struct buffer_head * bread(int dev,int block);
    // 读 4 块缓冲区到指定地址的内存中。
198 extern void bread_page(unsigned long addr,int dev,int b[4]);
    // 读取头一个指定的数据块，并标记后续将要读的块。
199 extern struct buffer_head * breada(int dev,int block,...);
    // 向设备 dev 申请一个磁盘块（区段，逻辑块）。返回逻辑块号
200 extern int new_block(int dev);
    // 释放设备数据区中的逻辑块(区段，磁盘块)block。复位指定逻辑块 block 的逻辑块位图比特位。
201 extern void free_block(int dev, int block);
    // 为设备 dev 建立一个新 i 节点，返回 i 节点号。
202 extern struct m_inode * new_inode(int dev);
```

```
    // 释放一个 i 节点（删除文件时）。
203 extern void free_inode(struct m_inode * inode);
    // 刷新指定设备缓冲区。
204 extern int sync_dev(int dev);
    // 读取指定设备的超级块。
205 extern struct super_block * get_super(int dev);
206 extern int ROOT_DEV;
207
    // 安装根文件系统。
208 extern void mount_root(void);
209
210 #endif
211
```

14.21 程序 14-21 linux/include/linux/hdreg.h

```
 1 /*
 2  * This file contains some defines for the AT-hd-controller.
 3  * Various sources. Check out some definitions (see comments with
 4  * a ques).
 5  */
   /*
    * 本文件含有一些 AT 硬盘控制器的定义。来自各种资料。请查证某些
    * 定义（带有问号的注释）。
    */
 6 #ifndef _HDREG_H
 7 #define _HDREG_H
 8
 9 /* Hd controller regs. Ref: IBM AT Bios-listing */
   /* 硬盘控制器寄存器端口。参见：IBM AT Bios 程序 */
10 #define HD_DATA         0x1f0   /* _CTL when writing */
11 #define HD_ERROR        0x1f1   /* see err-bits */
12 #define HD_NSECTOR      0x1f2   /* nr of sectors to read/write */
13 #define HD_SECTOR       0x1f3   /* starting sector */
14 #define HD_LCYL         0x1f4   /* starting cylinder */
15 #define HD_HCYL         0x1f5   /* high byte of starting cyl */
16 #define HD_CURRENT      0x1f6   /* 101dhhhh , d=drive, hhhh=head */
17 #define HD_STATUS       0x1f7   /* see status-bits */
18 #define HD_PRECOMP HD_ERROR     /* same io address, read=error, write=precomp */
19 #define HD_COMMAND HD_STATUS    /* same io address, read=status, write=cmd */
20
21 #define HD_CMD          0x3f6   // 控制寄存器端口。
22
23 /* Bits of HD_STATUS */
   /* 硬盘状态寄存器各位的定义(HD_STATUS) */
24 #define ERR_STAT        0x01    // 命令执行错误。
25 #define INDEX_STAT      0x02    // 收到索引。
26 #define ECC_STAT        0x04    /* Corrected error */  // ECC 校验错。
27 #define DRQ_STAT        0x08    // 请求服务。
28 #define SEEK_STAT       0x10    // 寻道结束。
29 #define WRERR_STAT      0x20    // 驱动器故障。
30 #define READY_STAT      0x40    // 驱动器准备好（就绪）。
31 #define BUSY_STAT       0x80    // 控制器忙碌。
32
33 /* Values for HD_COMMAND */
   /* 硬盘命令值（HD_CMD） */
34 #define WIN_RESTORE             0x10    // 驱动器重新校正（驱动器复位）。
35 #define WIN_READ                0x20    // 读扇区。
36 #define WIN_WRITE               0x30    // 写扇区。
37 #define WIN_VERIFY              0x40    // 扇区检验。
38 #define WIN_FORMAT              0x50    // 格式化磁道。
39 #define WIN_INIT                0x60    // 控制器初始化。
40 #define WIN_SEEK                0x70    // 寻道。
41 #define WIN_DIAGNOSE            0x90    // 控制器诊断。
42 #define WIN_SPECIFY             0x91    // 建立驱动器参数。
```

```
43
44 /* Bits for HD_ERROR */
   /* 错误寄存器各比特位的含义（HD_ERROR） */
   // 执行控制器诊断命令时含义与其他命令时的不同。下面分别列出：
   // ================================================
   //            诊断命令时              其他命令时
   // ------------------------------------------------
   // 0x01       无错误                  数据标志丢失
   // 0x02       控制器出错              磁道 0 错
   // 0x03       扇区缓冲区错
   // 0x04       ECC 部件错              命令放弃
   // 0x05       控制处理器错
   // 0x10                               ID 未找到
   // 0x40                               ECC 错误
   // 0x80                               坏扇区
   //-------------------------------------------------
45 #define MARK_ERR        0x01    /* Bad address mark ? */
46 #define TRK0_ERR        0x02    /* couldn't find track 0 */
47 #define ABRT_ERR        0x04    /* ? */
48 #define ID_ERR          0x10    /* ? */
49 #define ECC_ERR         0x40    /* ? */
50 #define BBD_ERR         0x80    /* ? */
51
   // 硬盘分区表结构。参见下面列表后信息。
52 struct partition {
53         unsigned char boot_ind;         /* 0x80 - active (unused) */
54         unsigned char head;             /* ? */
55         unsigned char sector;           /* ? */
56         unsigned char cyl;              /* ? */
57         unsigned char sys_ind;          /* ? */
58         unsigned char end_head;         /* ? */
59         unsigned char end_sector;       /* ? */
60         unsigned char end_cyl;          /* ? */
61         unsigned int start_sect;        /* starting sector counting from 0 */
62         unsigned int nr_sects;          /* nr of sectors in partition */
63 };
64
65 #endif
66
```

14.22 程序 14-22 linux/include/linux/head.h

```
 1 #ifndef _HEAD_H
 2 #define _HEAD_H
 3
 4 typedef struct desc_struct {        // 定义了段描述符的数据结构。该结构仅说明每个描述
 5         unsigned long a,b;         // 符是由 8 个字节构成，每个描述符表共有 256 项。
 6 } desc_table[256];
 7
 8 extern unsigned long pg_dir[1024];  // 内存页目录数组。每个目录项为 4 字节。从物理地址 0 开始。
 9 extern desc_table idt,gdt;          // 中断描述符表，全局描述符表。
10
11 #define GDT_NUL 0                   // 全局描述符表的第 0 项，不用。
12 #define GDT_CODE 1                  // 第 1 项，是内核代码段描述符项。
13 #define GDT_DATA 2                  // 第 2 项，是内核数据段描述符项。
14 #define GDT_TMP 3                   // 第 3 项，系统段描述符，Linux 没有使用。
15
16 #define LDT_NUL 0                   // 每个局部描述符表的第 0 项，不用。
17 #define LDT_CODE 1                  // 第 1 项，是用户程序代码段描述符项。
18 #define LDT_DATA 2                  // 第 2 项，是用户程序数据段描述符项。
19
20 #endif
21
```

14.23 程序 14-23 linux/include/linux/kernel.h

```
1  /*
2   * 'kernel.h' contains some often-used function prototypes etc
3   */
   /*
    * 'kernel.h'定义了一些常用函数的原型等。
    */
   // 验证给定地址开始的内存块是否超限。若超限则追加内存。( kernel/fork.c, 24 )。
4  void verify_area(void * addr,int count);
   // 显示内核出错信息，然后进入死循环。( kernel/panic.c, 16 )。
   // 函数名前的关键字 volatile 用于告诉编译器 gcc 该函数不会返回。这样可让 gcc 产生更好
   // 一些的代码，更重要的是使用这个关键字可以避免产生某未初始化变量的假警告信息。
5  volatile void panic(const char * str);
   // 进程退出处理。（ kernel/exit.c，262 ）。
6  volatile void do_exit(long error_code);
   // 标准打印（显示）函数。( init/main.c, 151)。
7  int printf(const char * fmt, ...);
   // 内核专用的打印信息函数，功能与 printf()相同。( kernel/printk.c, 21 )。
8  int printk(const char * fmt, ...);
   // 控制台显示函数。（ kernel/chr_drv/console.c，995）。
9  void console_print(const char * str);
   // 往 tty 上写指定长度的字符串。( kernel/chr_drv/tty_io.c, 290 )。
10 int tty_write(unsigned ch,char * buf,int count);
   // 通用内核内存分配函数。( lib/malloc.c, 117)。
11 void * malloc(unsigned int size);
   // 释放指定对象占用的内存。( lib/malloc.c, 182)。
12 void free_s(void * obj, int size);
   // 硬盘处理超时。（kernel/blk_drv/hd.c，318）。
13 extern void hd_times_out(void);
   // 停止蜂鸣。（kernel/chr_drv/console.c，944）。
14 extern void sysbeepstop(void);
   // 黑屏处理。（kernel/chr_drv/console.c，981）。
15 extern void blank_screen(void);
   // 恢复被黑屏的屏幕。（kernel/chr_drv/console.c，988）。
16 extern void unblank_screen(void);
17
18 extern int beepcount;              // 蜂鸣时间嘀嗒计数（kernel/chr_drv/console.c，988）。
19 extern int hd_timeout;             // 硬盘超时滴答值（kernel/blk_drv/blk.h）。
20 extern int blankinterval;          // 设定的屏幕黑屏间隔时间。
21 extern int blankcount;             // 黑屏时间计数（kernel/chr_drv/console.c，138、139）。
22
23 #define free(x) free_s((x), 0)
24
25 /*
26  * This is defined as a macro, but at some point this might become a
27  * real subroutine that sets a flag if it returns true (to do
28  * BSD-style accounting where the process is flagged if it uses root
29  * privs).  The implication of this is that you should do normal
30  * permissions checks first, and check suser() last.
31  */
```

```
    /*
     * 下面函数是以宏的形式定义的，但是在某方面来看它可以成为一个真正的子程序，
     * 如果返回是 true 时它将设置标志（如果使用 root 用户权限的进程设置了标志，则用
     * 于执行 BSD 方式的计帐处理）。这意味着你应该首先执行常规权限检查，最后再
     * 检测 suser()。
     */
32  #define suser() (current->euid == 0)            // 检测是否是超级用户。
33
```

14.24 程序 14-24 linux/include/linux/math_emu.h

```
 1 /*
 2  * linux/include/linux/math_emu.h
 3  *
 4  * (C) 1991 Linus Torvalds
 5  */
 6 #ifndef _LINUX_MATH_EMU_H
 7 #define _LINUX_MATH_EMU_H
 8
 9 #include <linux/sched.h>   // 调度程序头文件。定义了任务结构 task_struct、任务 0 的数据，
                               // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
10
   // CPU 产生异常中断 int 7 时在栈中分布的数据构成的结构，与系统调用时内核栈中数据分布类似。
11 struct info {
12         long ___math_ret;      // math_emulate()调用者（int7）返回地址。
13         long ___orig_eip;      // 临时保存原 EIP 的地方。
14         long ___edi;           // 异常中断 int7 处理过程入栈的寄存器。
15         long ___esi;
16         long ___ebp;
17         long ___sys_call_ret; // 中断 7 返回时将去执行系统调用的返回处理代码。
18         long ___eax;           // 以下部分（18--30 行）与系统调用时栈中结构相同。
19         long ___ebx;
20         long ___ecx;
21         long ___edx;
22         long ___orig_eax;      // 如不是系统调用而是其它中断时，该值为-1。
23         long ___fs;
24         long ___es;
25         long ___ds;
26         long ___eip;           // 26 -- 30 行 由 CPU 自动入栈。
27         long ___cs;
28         long ___eflags;
29         long ___esp;
30         long ___ss;
31 };
32
   // 为便于引用 info 结构中各字段（栈中数据）所定义的一些常量。
33 #define EAX (info->___eax)
34 #define EBX (info->___ebx)
35 #define ECX (info->___ecx)
36 #define EDX (info->___edx)
```

```
37 #define ESI (info->___esi)
38 #define EDI (info->___edi)
39 #define EBP (info->___ebp)
40 #define ESP (info->___esp)
41 #define EIP (info->___eip)
42 #define ORIG_EIP (info->___orig_eip)
43 #define EFLAGS (info->___eflags)
44 #define DS (*(unsigned short *) &(info->___ds))
45 #define ES (*(unsigned short *) &(info->___es))
46 #define FS (*(unsigned short *) &(info->___fs))
47 #define CS (*(unsigned short *) &(info->___cs))
48 #define SS (*(unsigned short *) &(info->___ss))
49
    // 终止数学协处理器仿真操作。在 math_emulation.c 程序中实现(L488 行)。
    // 下面 52-53 行上宏定义的实际作用是把__math_abort 重新定义为一个不会返回的函数
    // （即在前面加上了 volatile）。该宏的前部分：
    // (volatile void (*)(struct info *,unsigned int))
    // 是函数类型定义，用于重新指明 __math_abort 函数的定义。后面是其相应的参数。
    // 关键词 volatile 放在函数名前来修饰函数，是用来通知 gcc 编译器该函数不会返回,
    // 以让 gcc 产生更好一些的代码。详细说明请参见第 3 章 $3.3.2 节内容。
    // 因此下面的宏定义，其主要目的就是利用__math_abort，让它即可用作普通有返回函数，
    // 又可以在使用宏定义 math_abort() 时用作不返回的函数。
50 void __math_abort(struct info *, unsigned int);
51
52 #define math_abort(x,y) \
53 (((volatile void (*)(struct info *,unsigned int)) __math_abort)((x),(y)))
54
55 /*
56   * Gcc forces this stupid alignment problem: I want to use only two longs
57   * for the temporary real 64-bit mantissa, but then gcc aligns out the
58   * structure to 12 bytes which breaks things in math_emulate.c. Shit. I
59   * want some kind of "no-alignt" pragma or something.
60   */
    /*
     * Gcc 会强迫这种愚蠢的对齐问题：我只想使用两个 long 类型数据来表示 64 比特的
     * 临时实数尾数，但是 gcc 却会将该结构以 12 字节来对齐，这将导致 math_emulate.c
     * 中程序出问题。唉，我真需要某种非对齐“no-align”编译指令。
     */
61
    // 临时实数对应的结构。
62 typedef struct {
63  long a,b;               // 共 64 比特尾数。其中 a 为低 32 位，b 为高 32 位（包括 1 位固定位）。
64  short exponent;       // 指数值。
65 } temp_real;
```

```
66
    // 为了解决上面英文注释中所提及的对齐问题而设计的结构，作用同上面 temp_real 结构。
67 typedef struct {
68   short m0,m1,m2,m3;
69   short exponent;
70 } temp_real_unaligned;
71
    // 把 temp_real 类型值 a 赋值给 80387 栈寄存器 b (ST(i))。
72 #define real_to_real(a,b) \
73 ((*(long long *) (b) = *(long long *) (a)),((b)->exponent = (a)->exponent))
74
    // 长实数（双精度）结构。
75 typedef struct {
76   long a,b;                  // a 为长实数的低 32 位；b 为高 32 位。
77 } long_real;
78
79 typedef long short_real;    // 定义短实数类型。
80
    // 临时整数结构。
81 typedef struct {
82   long a,b;                  // a 为低 32 位；b 为高 32 位。
83   short sign;                // 符号标志。
84 } temp_int;
85
    // 80387 协处理器内部的状态字寄存器内容对应的结构。（参见图 11-6）
86 struct swd {
87   int ie:1;                  // 无效操作异常。
88   int de:1;                  // 非规格化异常。
89   int ze:1;                  // 除零异常。
90   int oe:1;                  // 上溢出异常。
91   int ue:1;                  // 下溢出异常。
92   int pe:1;                  // 精度异常。
93   int sf:1;                  // 栈出错标志，表示累加器溢出造成的异常。
94   int ir:1;                  // ir, b: 若上面 6 位任何未屏蔽异常发生，则置位。
95   int c0:1;                  // c0--c3: 条件码比特位。
96   int c1:1;
97   int c2:1;
98   int top:3;                 // 指示 80387 中当前位于栈顶的 80 位寄存器。
99   int c3:1;
100  int b:1;
101 };
102
    // 80387 内部寄存器控制方式常量。
103 #define I387 (current->tss.i387)            // 进程的 80387 状态信息。参见 sched.h 文件。
```

```
104 #define SWD (*(struct swd *) &I387.swd)      // 80387 中状态控制字。
105 #define ROUNDING ((I387.cwd >> 10) & 3)      // 取控制字中舍入控制方式。
106 #define PRECISION ((I387.cwd >> 8) & 3)      // 取控制字中精度控制方式。
107
    // 定义精度有效位常量。
108 #define BITS24 0                    // 精度有效数：24 位。（参见图 11-6）
109 #define BITS53 2                    // 精度有效数：53 位。
110 #define BITS64 3                    // 精度有效数：64 位。
111
    // 定义舍入方式常量。
112 #define ROUND_NEAREST    0          // 舍入方式：舍入到最近或偶数。
113 #define ROUND_DOWN  1               // 舍入方式：趋向负无限。
114 #define ROUND_UP 2                  // 舍入方式：趋向正无限。
115 #define ROUND_0          3          // 舍入方式：趋向截 0。
116
    // 常数定义。
117 #define CONSTZ      (temp_real_unaligned) {0x0000,0x0000,0x0000,0x0000,0x0000}    // 0
118 #define CONST1      (temp_real_unaligned) {0x0000,0x0000,0x0000,0x8000,0x3FFF}    // 1.0
119 #define CONSTPI     (temp_real_unaligned) {0xC235,0x2168,0xDAA2,0xC90F,0x4000}    // Pi
120 #define CONSTLN2 (temp_real_unaligned) {0x79AC,0xD1CF,0x17F7,0xB172,0x3FFE}    // Loge(2)
121 #define CONSTLG2 (temp_real_unaligned) {0xF799,0xFBCF,0x9A84,0x9A20,0x3FFD}    // Log10(2)
122 #define CONSTL2E (temp_real_unaligned) {0xF0BC,0x5C17,0x3B29,0xB8AA,0x3FFF}    // Log2(e)
123 #define CONSTL2T (temp_real_unaligned) {0x8AFE,0xCD1B,0x784B,0xD49A,0x4000}    // Log2(10)
124
    // 设置 80387 各状态
125 #define set_IE() (I387.swd |= 1)
126 #define set_DE() (I387.swd |= 2)
127 #define set_ZE() (I387.swd |= 4)
128 #define set_OE() (I387.swd |= 8)
129 #define set_UE() (I387.swd |= 16)
130 #define set_PE() (I387.swd |= 32)
131
    // 设置 80387 各控制条件
132 #define set_C0() (I387.swd |= 0x0100)
133 #define set_C1() (I387.swd |= 0x0200)
134 #define set_C2() (I387.swd |= 0x0400)
135 #define set_C3() (I387.swd |= 0x4000)
136
137 /* ea.c */
138
    // 计算仿真指令中操作数使用到的有效地址值，即根据指令中寻址模式字节计算有效地址值。
    // 参数：__info - 中断时栈中内容对应结构；__code - 指令代码。
    // 返回：有效地址值。
139 char * ea(struct info * __info, unsigned short __code);
```

```
140
141 /* convert.c */
142
    // 各种数据类型转换函数。在 convert.c 文件中实现。
143 void short_to_temp(const short_real * __a, temp_real * __b);
144 void long_to_temp(const long_real * __a, temp_real * __b);
145 void temp_to_short(const temp_real * __a, short_real * __b);
146 void temp_to_long(const temp_real * __a, long_real * __b);
147 void real_to_int(const temp_real * __a, temp_int * __b);
148 void int_to_real(const temp_int * __a, temp_real * __b);
149
150 /* get_put.c */
151
    // 存取各种类型数的函数。
152 void get_short_real(temp_real *, struct info *, unsigned short);
153 void get_long_real(temp_real *, struct info *, unsigned short);
154 void get_temp_real(temp_real *, struct info *, unsigned short);
155 void get_short_int(temp_real *, struct info *, unsigned short);
156 void get_long_int(temp_real *, struct info *, unsigned short);
157 void get_longlong_int(temp_real *, struct info *, unsigned short);
158 void get_BCD(temp_real *, struct info *, unsigned short);
159 void put_short_real(const temp_real *, struct info *, unsigned short);
160 void put_long_real(const temp_real *, struct info *, unsigned short);
161 void put_temp_real(const temp_real *, struct info *, unsigned short);
162 void put_short_int(const temp_real *, struct info *, unsigned short);
163 void put_long_int(const temp_real *, struct info *, unsigned short);
164 void put_longlong_int(const temp_real *, struct info *, unsigned short);
165 void put_BCD(const temp_real *, struct info *, unsigned short);
166
167 /* add.c */
168
    // 仿真浮点加法指令的函数。
169 void fadd(const temp_real *, const temp_real *, temp_real *);
170
171 /* mul.c */
172
    // 仿真浮点乘法指令。
173 void fmul(const temp_real *, const temp_real *, temp_real *);
174
175 /* div.c */
176
    // 仿真浮点除法指令。
177 void fdiv(const temp_real *, const temp_real *, temp_real *);
178
```

```
179 /* compare.c */
180
    // 比较函数。
181 void fcom(const temp_real *, const temp_real *);      // 仿真浮点指令 FCOM，比较两个数。
182 void fucom(const temp_real *, const temp_real *);     // 仿真浮点指令 FUCOM，无次序比较。
183 void ftst(const temp_real *);                         // 仿真浮点指令 FTST，栈顶累加器与 0 比较。
184
185 #endif
186
```

14.25 程序 14-25 linux/include/linux/mm.h

```
 1 #ifndef _MM_H
 2 #define _MM_H
 3
 4 #define PAGE_SIZE 4096     // 定义 1 页内存页面字节数。注意高速缓冲块长度是 1024 字节。
 5
 6 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原型定义。
 7 #include <signal.h>       // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。
 8
 9 extern int SWAP_DEV;      // 内存页面交换设备号。定义在 mm/memory.c 文件中，第 36 行。
10
   // 从交换设备读入和写出被交换内存页面。ll_rw_page()定义在 blk_drv/ll_rw_block.c 文件中。
   // 参数 nr 是主内存区中页面号；buffer 是读/写缓冲区。
11 #define read_swap_page(nr,buffer) ll_rw_page(READ,SWAP_DEV,(nr),(buffer));
12 #define write_swap_page(nr,buffer) ll_rw_page(WRITE,SWAP_DEV,(nr),(buffer));
13
   // 在主内存区中取空闲物理页面。如果已经没有可用内存了，则返回 0。
14 extern unsigned long get_free_page(void);
   // 把一内容已修改过的物理内存页面映射到线性地址空间指定处。与 put_page()几乎完全一样。
15 extern unsigned long put_dirty_page(unsigned long page,unsigned long address);
   //释放物理地址 addr 开始的 1 页面内存。
16 extern void free_page(unsigned long addr);
17 void swap_free(int page_nr);
18 void swap_in(unsigned long *table_ptr);
19
   //// 显示内存已用完出错信息，并退出。
   // 下面函数名前的关键字 volatile 用于告诉编译器 gcc 该函数不会返回。这样可让 gcc 产生更好
   // 一些的代码，更重要的是使用这个关键字可以避免产生某些（未初始化变量的）假警告信息。
20 extern inline volatile void oom(void)
21 {
   // do_exit()应该使用退出代码，这里用了信号值 SIGSEGV(11) 相同值的出错码含义是“资源
   // 暂时不可用”，正好同义。
22         printk("out of memory\n\r");
23         do_exit(SIGSEGV);
24 }
25
   // 刷新页变换高速缓冲宏函数。
   // 为了提高地址转换的效率，CPU 将最近使用的页表数据存放在芯片中高速缓冲中。在修改过
   // 页表信息之后，就需要刷新该缓冲区。 这里使用重新加载页目录基址寄存器 cr3 的方法来
   // 进行刷新。下面 eax = 0，是页目录的基址。
26 #define invalidate() \
27 __asm__("movl %%eax,%%cr3"::"a" (0))
28
29 /* these are not to be changed without changing head.s etc */
   /* 下面定义若需要改动，则需要与 head.s 等文件中的相关信息一起改变 */
   // Linux 0.12 内核默认支持的最大内存容量是 16MB，可以修改这些定义以适合更多的内存。
30 #define LOW_MEM 0x100000                    // 机器物理内存低端（1MB）。
31 extern unsigned long HIGH_MEMORY;           // 存放实际物理内存最高端地址。
32 #define PAGING_MEMORY (15*1024*1024)        // 分页内存 15MB。主内存区最多 15M。
33 #define PAGING_PAGES (PAGING_MEMORY>>12)    // 分页后的物理内存页面数（3840）。
```

```
34 #define MAP_NR(addr) (((addr)-LOW_MEM)>>12)  // 指定内存地址映射为页面号。
35 #define USED 100                             // 页面被占用标志，参见 449 行。
36
   // 内存映射字节图（1 字节代表 1 页内存）。每个页面对应的字节用于标志页面当前被引用
   // （占用）次数。它最大可以映射 15Mb 的内存空间。在初始化函数 mem_init()中，对于不
   // 能用作主内存区页面的位置均都预先被设置成 USED（100）。
37 extern unsigned char mem_map [ PAGING_PAGES ];
38
   // 下面定义的符号常量对应页目录表项和页表（二级页表）项中的一些标志位。
39 #define PAGE_DIRTY      0x40                 // 位 6，页面脏（已修改）。
40 #define PAGE_ACCESSED   0x20                 // 位 5，页面被访问过。
41 #define PAGE_USER       0x04                 // 位 2，页面属于：1-用户；0-超级用户。
42 #define PAGE_RW         0x02                 // 位 1，读写权：1 - 写；0 - 读。
43 #define PAGE_PRESENT    0x01                 // 位 0，页面存在：1-存在；0-不存在。
44
45 #endif
46
```

14.26 程序 14-26 linux/include/linux/sched.h

```
 1 #ifndef _SCHED_H
 2 #define _SCHED_H
 3
 4 #define HZ 100                          // 定义系统时钟滴答频率(1 百赫兹，每个滴答 10ms)
 5
 6 #define NR_TASKS        64              // 系统中同时最多任务（进程）数。
 7 #define TASK_SIZE       0x04000000      // 每个任务的长度（64MB）。
 8 #define LIBRARY_SIZE    0x00400000      // 动态加载库长度（4MB）。
 9
10 #if (TASK_SIZE & 0x3fffff)
11 #error "TASK_SIZE must be multiple of 4M"       // 任务长度必须是 4MB 的倍数。
12 #endif
13
14 #if (LIBRARY_SIZE & 0x3fffff)
15 #error "LIBRARY_SIZE must be a multiple of 4M" // 库长度也必须是 4MB 的倍数。
16 #endif
17
18 #if (LIBRARY_SIZE >= (TASK_SIZE/2))
19 #error "LIBRARY_SIZE too damn big!"             // 加载库的长度不得大于任务长度的一半。
20 #endif
21
22 #if (((TASK_SIZE>>16)*NR_TASKS) != 0x10000)
23 #error "TASK_SIZE*NR_TASKS must be 4GB"         // 任务长度*任务总个数必须为 4GB。
24 #endif
25
   // 在进程逻辑地址空间中动态库被加载的位置（60MB 处）。
26 #define LIBRARY_OFFSET (TASK_SIZE - LIBRARY_SIZE)
27
   // 下面宏 CT_TO_SECS 和 CT_TO_USECS 用于把系统当前嘀嗒数转换成用秒值加微秒值表示。
28 #define CT_TO_SECS(x)   ((x) / HZ)
29 #define CT_TO_USECS(x)  (((x) % HZ) * 1000000/HZ)
30
31 #define FIRST_TASK task[0]              // 任务 0 比较特殊，所以特意给它单独定义一个符号。
32 #define LAST_TASK task[NR_TASKS-1]      // 任务数组中的最后一项任务。
33
34 #include <linux/head.h>
35 #include <linux/fs.h>
36 #include <linux/mm.h>
37 #include <sys/param.h>
38 #include <sys/time.h>
39 #include <sys/resource.h>
40 #include <signal.h>
41
42 #if (NR_OPEN > 32)
43 #error "Currently the close-on-exec-flags and select masks are in one long, max 32 files/proc"
44 #endif
45
   // 这里定义了进程运行时可能处的状态。
46 #define TASK_RUNNING            0   // 进程正在运行或已准备就绪。
```

```
47 #define TASK_INTERRUPTIBLE      1   // 进程处于可中断等待状态。
48 #define TASK_UNINTERRUPTIBLE    2   // 进程处于不可中断等待状态，主要用于 I/O 操作等待。
49 #define TASK_ZOMBIE             3   // 进程处于僵死状态，已经停止运行，但父进程还没发信号。
50 #define TASK_STOPPED            4   // 进程已停止。
51
52 #ifndef NULL
53 #define NULL ((void *) 0)           // 定义 NULL 为空指针。
54 #endif
55
   // 复制进程的页目录页表。Linus 认为这是内核中最复杂的函数之一。( mm/memory.c, 105 )
56 extern int copy_page_tables(unsigned long from, unsigned long to, long size);
   // 释放页表所指定的内存块及页表本身。( mm/memory.c, 150 )
57 extern int free_page_tables(unsigned long from, unsigned long size);
58
   // 调度程序的初始化函数。( kernel/sched.c, 385 )
59 extern void sched_init(void);
   // 进程调度函数。( kernel/sched.c, 104 )
60 extern void schedule(void);
   // 异常(陷阱)中断处理初始化函数，设置中断调用门并允许中断请求信号。( kernel/traps.c, 181 )
61 extern void trap_init(void);
   // 显示内核出错信息，然后进入死循环。( kernel/panic.c, 16 )。
62 extern void panic(const char * str);
   // 往 tty 上写指定长度的字符串。( kernel/chr_drv/tty_io.c, 290 )。
63 extern int tty_write(unsigned minor,char * buf,int count);
64
65 typedef int (*fn_ptr)();                    // 定义函数指针类型。
66
   // 下面是数学协处理器使用的结构，主要用于保存进程切换时 i387 的执行状态信息。
67 struct i387_struct {
68         long    cwd;            // 控制字(Control word)。
69         long    swd;            // 状态字(Status word)。
70         long    twd;            // 标记字(Tag word)。
71         long    fip;            // 协处理器代码指针。
72         long    fcs;            // 协处理器代码段寄存器。
73         long    foo;            // 内存操作数的偏移位置。
74         long    fos;            // 内存操作数的段值。
75         long    st_space[20];   /* 8*10 bytes for each FP-reg = 80 bytes */
76 };                              /* 8 个 10 字节的协处理器累加器。*/
77
   // 任务状态段数据结构。
78 struct tss_struct {
79         long    back_link;      /* 16 high bits zero */
80         long    esp0;
81         long    ss0;            /* 16 high bits zero */
82         long    esp1;
83         long    ss1;            /* 16 high bits zero */
84         long    esp2;
85         long    ss2;            /* 16 high bits zero */
86         long    cr3;
87         long    eip;
88         long    eflags;
89         long    eax,ecx,edx,ebx;
90         long    esp;
```

```
 91         long    ebp;
 92         long    esi;
 93         long    edi;
 94         long    es;             /* 16 high bits zero */
 95         long    cs;             /* 16 high bits zero */
 96         long    ss;             /* 16 high bits zero */
 97         long    ds;             /* 16 high bits zero */
 98         long    fs;             /* 16 high bits zero */
 99         long    gs;             /* 16 high bits zero */
100         long    ldt;            /* 16 high bits zero */
101         long    trace_bitmap;   /* bits: trace 0, bitmap 16-31 */
102         struct i387_struct i387;
103 };
104
    // 下面是任务（进程）数据结构，或称为进程描述符。
    // long state                   任务的运行状态（-1 不可运行，0 可运行(就绪)，>0 已停止）。
    // long counter                 任务运行时间计数(递减)（滴答数），运行时间片。
    // long priority                优先数。任务开始运行时 counter=priority，越大运行越长。
    // long signal                  信号位图，每个比特位代表一种信号，信号值=位偏移值+1。
    // struct sigaction sigaction[32]   信号执行属性结构，对应信号将要执行的操作和标志信息。
    // long blocked                 进程信号屏蔽码（对应信号位图）。
    // -------------------
    // int exit_code                任务执行停止的退出码，其父进程会取。
    // unsigned long start_code     代码段地址。
    // unsigned long end_code       代码长度（字节数）。
    // unsigned long end_data       代码长度 + 数据长度（字节数）。
    // unsigned long brk            总长度（字节数）。
    // unsigned long start_stack    堆栈段地址。
    // long pid                     进程标识号(进程号)。
    // long pgrp                    进程组号。
    // long session                 会话号。
    // long leader                  会话首领。
    // int groups[NGROUPS]          进程所属组号。一个进程可属于多个组。
    // task_struct *p_pptr          指向父进程的指针。
    // task_struct *p_cptr          指向最新子进程的指针。
    // task_struct *p_ysptr         指向比自己后创建的相邻进程的指针。
    // task_struct *p_osptr         指向比自己早创建的相邻进程的指针。
    // unsigned short uid           用户标识号（用户 id）。
    // unsigned short euid          有效用户 id。
    // unsigned short suid          保存的用户 id。
    // unsigned short gid           组标识号（组 id）。
    // unsigned short egid          有效组 id。
    // unsigned short sgid          保存的组 id。
    // long timeout                 内核定时超时值。
    // long alarm                   报警定时值（滴答数）。
    // long utime                   用户态运行时间（滴答数）。
    // long stime                   系统态运行时间（滴答数）。
    // long cutime                  子进程用户态运行时间。
    // long cstime                  子进程系统态运行时间。
    // long start_time              进程开始运行时刻。
    // struct rlimit rlim[RLIM_NLIMITS]  进程资源使用统计数组。
    // unsigned int flags;          各进程的标志，在下面第 149 行开始定义（还未使用）。
    // unsigned short used_math     标志：是否使用了协处理器。
```

```
    // -------------------------
    // int tty                      进程使用 tty 终端的子设备号。-1 表示没有使用。
    // unsigned short umask         文件创建属性屏蔽位。
    // struct m_inode * pwd         当前工作目录 i 节点结构指针。
    // struct m_inode * root        根目录 i 节点结构指针。
    // struct m_inode * executable  执行文件 i 节点结构指针。
    // struct m_inode * library     被加载库文件 i 节点结构指针。
    // unsigned long close_on_exec  执行时关闭文件句柄位图标志。（参见 include/fcntl.h）
    // struct file * filp[NR_OPEN]  文件结构指针表，最多 32 项。表项号即是文件描述符的值。
    // struct desc_struct ldt[3]    局部描述符表。0-空，1-代码段 cs，2-数据和堆栈段 ds&ss。
    // struct tss_struct tss        进程的任务状态段信息结构。
    // ====================================
105 struct task_struct {
106 /* these are hardcoded - don't touch */
107         long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
108         long counter;
109         long priority;
110         long signal;
111         struct sigaction sigaction[32];
112         long blocked;   /* bitmap of masked signals */
113 /* various fields */
114         int exit_code;
115         unsigned long start_code,end_code,end_data,brk,start_stack;
116         long pid,pgrp,session,leader;
117         int     groups[NGROUPS];
118         /*
119          * pointers to parent process, youngest child, younger sibling,
120          * older sibling, respectively.  (p->father can be replaced with
121          * p->p_pptr->pid)
122          */
123         struct task_struct      *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
124         unsigned short uid,euid,suid;
125         unsigned short gid,egid,sgid;
126         unsigned long timeout,alarm;
127         long utime,stime,cutime,cstime,start_time;
128         struct rlimit rlim[RLIM_NLIMITS];
129         unsigned int flags;     /* per process flags, defined below */
130         unsigned short used_math;
131 /* file system info */
132         int tty;                /* -1 if no tty, so it must be signed */
133         unsigned short umask;
134         struct m_inode * pwd;
135         struct m_inode * root;
136         struct m_inode * executable;
137         struct m_inode * library;
138         unsigned long close_on_exec;
139         struct file * filp[NR_OPEN];
140 /* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
141         struct desc_struct ldt[3];
142 /* tss for this task */
143         struct tss_struct tss;
144 };
145
```

```
146 /*
147  * Per process flags
148  */
    /* 每个进程的标志 */  /* 打印对齐警告信息。还未实现，仅用于 486 */
149 #define PF_ALIGNWARN    0x00000001      /* Print alignment warning msgs */
150                                         /* Not implemented yet, only for 486*/
151
152 /*
153  *  INIT_TASK is used to set up the first task table, touch at
154  * your own risk!. Base=0, limit=0x9ffff (=640kB)
155  */
    /*
     * INIT_TASK 用于设置第 1 个任务表，若想修改，责任自负☺!
     * 基址 Base = 0，段长 limit = 0x9ffff（=640kB）。
     */
    // 对应上面任务结构的第 1 个任务的信息。
156 #define INIT_TASK \
157 /* state etc */ { 0,15,15, \      // state, counter, priority
158 /* signals */   0,{{},},0, \      // signal, sigaction[32], blocked
159 /* ec,brk... */ 0,0,0,0,0,0, \  // exit_code,start_code,end_code,end_data,brk,start_stack
160 /* pid etc.. */ 0,0,0,0, \        // pid, pgrp, session, leader
161 /* suppl grps*/ {NOGROUP,}, \     // groups[]
162 /* proc links*/ &init_task.task,0,0,0, \    // p_pptr, p_cptr, p_ysptr, p_osptr
163 /* uid etc */   0,0,0,0,0,0, \    // uid, euid, suid, gid, egid, sgid
164 /* timeout */   0,0,0,0,0,0,0,0, \  // alarm,utime,stime,cutime,cstime,start_time,used_math
165 /* rlimits */   { {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff},  \
166                   {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff}, \
167                   {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff}}, \
168 /* flags */     0, \              // flags
169 /* math */      0, \              // used_math, tty,umask,pwd,root,executable,close_on_exec
170 /* fs info */   -1,0022,NULL,NULL,NULL,NULL,0, \
171 /* filp */      {NULL,}, \        // filp[20]
172         { \                       // ldt[3]
173                 {0,0}, \
174 /* ldt */       {0x9f,0xc0fa00}, \  // 代码长 640K，基址 0x0，G=1，D=1，DPL=3，P=1 TYPE=0xa
175                 {0x9f,0xc0f200}, \  // 数据长 640K，基址 0x0，G=1，D=1，DPL=3，P=1 TYPE=0x2
176         }, \
177 /*tss*/ {0,PAGE_SIZE+(long)&init_task,0x10,0,0,0,0,(long)&pg_dir,\       // tss
178          0,0,0,0,0,0,0,0, \
179          0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
180          _LDT(0),0x80000000, \
181                 {} \
182         }, \
183 }
184
185 extern struct task_struct *task[NR_TASKS];      // 任务指针数组。
186 extern struct task_struct *last_task_used_math; // 上一个使用过协处理器的进程。
187 extern struct task_struct *current;             // 当前运行进程结构指针变量。
188 extern unsigned long volatile jiffies;          // 从开机开始算起的滴答数（10ms/滴答）。
189 extern unsigned long startup_time;              // 开机时间。从 1970:0:0:0 开始计时的秒数。
190 extern int jiffies_offset;                      // 用于累计需要调整的时间嘀嗒数。
191
192 #define CURRENT_TIME (startup_time+(jiffies+jiffies_offset)/HZ)  // 当前时间（秒数）。
```

```
193
    // 添加定时器函数（定时时间 jiffies 滴答数，定时到时调用函数*fn()）。( kernel/sched.c )
194 extern void add_timer(long jiffies, void (*fn)(void));
    // 不可中断的等待睡眠。( kernel/sched.c )
195 extern void sleep_on(struct task_struct ** p);
    // 可中断的等待睡眠。( kernel/sched.c )
196 extern void interruptible_sleep_on(struct task_struct ** p);
    // 明确唤醒睡眠的进程。( kernel/sched.c )
197 extern void wake_up(struct task_struct ** p);
    // 检查当前进程是否在指定的用户组 grp 中。
198 extern int in_group_p(gid_t grp);
199
200 /*
201  * Entry into gdt where to find first TSS. 0-nul, 1-cs, 2-ds, 3-syscall
202  * 4-TSS0, 5-LDT0, 6-TSS1 etc ...
203  */
    /*
     * 寻找第 1 个 TSS 在全局表中的入口。0-没有用 nul，1-代码段 cs，2-数据段 ds，3-系统段 syscall
     * 4-任务状态段 TSS0，5-局部表 LTD0，6-任务状态段 TSS1，等。
     */
    // 从该英文注释可以猜想到，Linus 当时曾想把系统调用的代码专门放在 GDT 表中第 4 个独立的段中。
    // 但后来并没有那样做，于是就一直把 GDT 表中第 4 个描述符项（上面 syscall 项）闲置在一旁。
    // 下面定义宏：全局表中第 1 个任务状态段(TSS)描述符的选择符索引号。
204 #define FIRST_TSS_ENTRY 4
    // 全局表中第 1 个局部描述符表(LDT)描述符的选择符索引号。
205 #define FIRST_LDT_ENTRY (FIRST_TSS_ENTRY+1)
    // 宏定义，计算在全局表中第 n 个任务的 TSS 段描述符的选择符值（偏移量）。
    // 因每个描述符占 8 字节，因此 FIRST_TSS_ENTRY<<3 表示该描述符在 GDT 表中的起始偏移位置。
    // 因为每个任务使用 1 个 TSS 和 1 个 LDT 描述符，共占用 16 字节，因此需要 n<<4 来表示对应
    // TSS 起始位置。该宏得到的值正好也是该 TSS 的选择符值。
206 #define _TSS(n) ((((unsigned long) n)<<4)+(FIRST_TSS_ENTRY<<3))
    // 宏定义，计算在全局表中第 n 个任务的 LDT 段描述符的选择符值（偏移量）。
207 #define _LDT(n) ((((unsigned long) n)<<4)+(FIRST_LDT_ENTRY<<3))
    // 宏定义，把第 n 个任务的 TSS 段选择符加载到任务寄存器 TR 中。
208 #define ltr(n) __asm__("ltr %%ax"::"a" (_TSS(n)))
    // 宏定义，把第 n 个任务的 LDT 段选择符加载到局部描述符表寄存器 LDTR 中。
209 #define lldt(n) __asm__("lldt %%ax"::"a" (_LDT(n)))
    // 取当前运行任务的任务号（是任务数组中的索引值，与进程号 pid 不同）。
    // 返回：n - 当前任务号。用于( kernel/traps.c )。
210 #define str(n) \
211 __asm__("str %%ax\n\t" \                    // 将任务寄存器中 TSS 段的选择符复制到 ax 中。
212         "subl %2,%%eax\n\t" \               // (eax - FIRST_TSS_ENTRY*8)➔eax
213         "shrl $4,%%eax" \                   // (eax/16)➔eax = 当前任务号。
214         :"=a" (n) \
215         :"a" (0),"i" (FIRST_TSS_ENTRY<<3))
216 /*
217  *      switch_to(n) should switch tasks to task nr n, first
218  * checking that n isn't the current task, in which case it does nothing.
219  * This also clears the TS-flag if the task we switched to has used
220  * tha math co-processor latest.
221  */
    /*
     * switch_to(n)将切换当前任务到任务 nr，即 n。首先检测任务 n 不是当前任务，
```

```
 * 如果是则什么也不做退出。如果我们切换到的任务最近（上次运行）使用过数学
 * 协处理器的话，则还需复位控制寄存器 cr0 中的 TS 标志。
 */
// 跳转到一个任务的 TSS 段选择符组成的地址处会造成 CPU 进行任务切换操作。
// 输入：%0 - 指向__tmp；                %1 - 指向__tmp.b 处，用于存放新 TSS 的选择符；
//       dx - 新任务 n 的 TSS 段选择符；   ecx - 新任务 n 的任务结构指针 task[n]。
// 其中临时数据结构__tmp 用于组建 177 行远跳转（far jump）指令的操作数。该操作数由 4 字节
// 偏移地址和 2 字节的段选择符组成。因此__tmp 中 a 的值是 32 位偏移值，而 b 的低 2 字节是新
// TSS 段的选择符（高 2 字节不用）。跳转到 TSS 段选择符会造成任务切换到该 TSS 对应的进程。
// 对于造成任务切换的长跳转，a 值无用。177 行上的内存间接跳转指令使用 6 字节操作数作为跳
// 转目的地的长指针，其格式为：jmp 16 位段选择符：32 位偏移值。但在内存中操作数的表示顺
// 序与这里正好相反。任务切换回来之后，在判断原任务上次执行是否使用过协处理器时，是通过
// 将原任务指针与保存在 last_task_used_math 变量中的上次使用过协处理器任务指针进行比较而
// 作出的，参见文件 kernel/sched.c 中有关 math_state_restore()函数的说明。
#define switch_to(n) {\
struct {long a,b;} __tmp; \
__asm__("cmpl %%ecx,_current\n\t" \      // 任务 n 是当前任务吗?(current ==task[n]?)
        "je 1f\n\t" \                    // 是，则什么都不做，退出。
        "movw %%dx,%1\n\t" \             // 将新任务 TSS 的 16 位选择符存入__tmp.b 中。
        "xchgl %%ecx,_current\n\t" \     // current = task[n]；ecx = 被切换出的任务。
        "ljmp %0\n\t" \                  // 执行长跳转至*&__tmp，造成任务切换。
                                         // 在任务切换回来后才会继续执行下面的语句。
        "cmpl %%ecx,_last_task_used_math\n\t" \      // 原任务上次使用过协处理器吗？
        "jne 1f\n\t" \                   // 没有则跳转，退出。
        "clts\n" \                       // 原任务上次使用过协处理器，则清 cr0 中的任务
        "1:" \                           // 切换标志 TS。
        ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
        "d" (_TSS(n)),"c" ((long) task[n])); \
}

// 页面地址对准。（在内核代码中没有任何地方引用!!）
#define PAGE_ALIGN(n) (((n)+0xfff)&0xfffff000)

// 设置位于地址 addr 处描述符中的各基地址字段(基地址是 base)。
// %0 - 地址 addr 偏移 2；%1 - 地址 addr 偏移 4；%2 - 地址 addr 偏移 7；edx - 基地址 base。
#define _set_base(addr,base) \
__asm__("movw %%dx,%0\n\t" \             // 基址 base 低 16 位(位 15-0)➔[addr+2]。
        "rorl $16,%%edx\n\t" \           // edx 中基址高 16 位(位 31-16)➔dx。
        "movb %%dl,%1\n\t" \             // 基址高 16 位中的低 8 位(位 23-16)➔[addr+4]。
        "movb %%dh,%2" \                 // 基址高 16 位中的高 8 位(位 31-24)➔[addr+7]。
        ::"m" (*((addr)+2)), \
          "m" (*((addr)+4)), \
          "m" (*((addr)+7)), \
          "d" (base) \
        :"dx")                   // 告诉 gcc 编译器 edx 寄存器中的值已被嵌入汇编程序改变了。

// 设置位于地址 addr 处描述符中的段限长字段(段长是 limit)。
// %0 - 地址 addr；%1 - 地址 addr 偏移 6 处；edx - 段长值 limit。
#define _set_limit(addr,limit) \
__asm__("movw %%dx,%0\n\t" \             // 段长 limit 低 16 位(位 15-0)➔[addr]。
        "rorl $16,%%edx\n\t" \           // edx 中的段长高 4 位(位 19-16)➔dl。
        "movb %1,%%dh\n\t" \             // 取原[addr+6]字节➔dh，其中高 4 位是些标志。
        "andb $0xf0,%%dh\n\t" \          // 清 dh 的低 4 位(将存放段长的位 19-16)。
```

```
255         "orb %%dh,%%dl\n\t" \           // 将原高 4 位标志和段长的高 4 位(位 19-16)合成 1 字节，
256         "movb %%dl,%1" \                // 并放会[addr+6]处。
257         ::"m" (*(addr)), \
258           "m" (*((addr)+6)), \
259           "d" (limit) \
260         :"dx")
261
    // 设置局部描述符表中 ldt 描述符的基地址字段。
262 #define set_base(ldt,base) _set_base( ((char *)&(ldt)) , base )
    // 设置局部描述符表中 ldt 描述符的段长字段。
263 #define set_limit(ldt,limit) _set_limit( ((char *)&(ldt)) , (limit-1)>>12 )
264
    // 从地址 addr 处描述符中取段基地址。功能与_set_base()正好相反。
    // edx - 存放基地址(__base)；%1 - 地址 addr 偏移 2；%2 - 地址 addr 偏移 4；%3 - addr 偏移 7。
265 #define _get_base(addr) ({\
266 unsigned long __base; \
267 __asm__("movb %3,%%dh\n\t" \            // 取[addr+7]处基址高 16 位的高 8 位(位 31-24)➔dh。
268         "movb %2,%%dl\n\t" \            // 取[addr+4]处基址高 16 位的低 8 位(位 23-16)➔dl。
269         "shll $16,%%edx\n\t" \          // 基地址高 16 位移到 edx 中高 16 位处。
270         "movw %1,%%dx" \                // 取[addr+2]处基址低 16 位(位 15-0)➔dx。
271         :"=d" (__base) \                // 从而 edx 中含有 32 位的段基地址。
272         :"m" (*((addr)+2)), \
273          "m" (*((addr)+4)), \
274          "m" (*((addr)+7))); \
275 __base;})
276
    // 取局部描述符表中 ldt 所指段描述符中的基地址。
277 #define get_base(ldt) _get_base( ((char *)&(ldt)) )
278
    // 取段选择符 segment 指定的描述符中的段限长值。
    // 指令 lsl 是 Load Segment Limit 缩写。它从指定段描述符中取出分散的限长比特位拼成完整的
    // 段限长值放入指定寄存器中。所得的段限长是实际字节数减 1，因此这里还需要加 1 后才返回。
    // %0 - 存放段长值(字节数)；%1 - 段选择符 segment。
279 #define get_limit(segment) ({ \
280 unsigned long __limit; \
281 __asm__("lsll %1,%0\n\tincl %0":"=r" (__limit):"r" (segment)); \
282 __limit;})
283
284 #endif
285
```

14.27 程序 14-27 linux/include/linux/sys.h

```
/*
 * Why isn't this a .c file?  Enquiring minds....
 */
/*
 * 为什么这不是一个.c 文件？动动脑筋自己想想....
 */

extern int sys_setup();         // 0 - 系统启动初始化设置函数。     (kernel/blk_drv/hd.c)
extern int sys_exit();          // 1 - 程序退出。                   (kernel/exit.c )
extern int sys_fork();          // 2 - 创建进程。                   (kernel/system_call.s)
extern int sys_read();          // 3 - 读文件。                     (fs/read_write.c)
extern int sys_write();         // 4 - 写文件。                     (fs/read_write.c)
extern int sys_open();          // 5 - 打开文件。                   (fs/open.c)
extern int sys_close();         // 6 - 关闭文件。                   (fs/open.c)
extern int sys_waitpid();       // 7 - 等待进程终止。               (kernel/exit.c)
extern int sys_creat();         // 8 - 创建文件。                   (fs/open.c)
extern int sys_link();          // 9 - 创建一个文件的硬连接。       (fs/namei.c)
extern int sys_unlink();        // 10 - 删除一个文件名(或删除文件)。  (fs/namei.c)
extern int sys_execve();        // 11 - 执行程序。                  (kernel/system_call.s)
extern int sys_chdir();         // 12 - 更改当前目录。              (fs/open.c)
extern int sys_time();          // 13 - 取当前时间。                (kernel/sys.c)
extern int sys_mknod();         // 14 - 建立块/字符特殊文件。       (fs/namei.c)
extern int sys_chmod();         // 15 - 修改文件属性。              (fs/open.c)
extern int sys_chown();         // 16 - 修改文件宿主和所属组。      (fs/open.c)
extern int sys_break();         // 17 -                             (kernel/sys.c)*
extern int sys_stat();          // 18 - 使用路径名取文件状态信息。(fs/stat.c)
extern int sys_lseek();         // 19 - 重新定位读/写文件偏移。     (fs/read_write.c)
extern int sys_getpid();        // 20 - 取进程 id。                (kernel/sched.c)
extern int sys_mount();         // 21 - 安装文件系统。              (fs/super.c)
extern int sys_umount();        // 22 - 卸载文件系统。              (fs/super.c)
extern int sys_setuid();        // 23 - 设置进程用户 id。          (kernel/sys.c)
extern int sys_getuid();        // 24 - 取进程用户 id。            (kernel/sched.c)
extern int sys_stime();         // 25 - 设置系统时间日期。          (kernel/sys.c)*
extern int sys_ptrace();        // 26 - 程序调试。                  (kernel/sys.c)*
extern int sys_alarm();         // 27 - 设置报警。                  (kernel/sched.c)
extern int sys_fstat();         // 28 - 使用文件句柄取文件的状态信息。(fs/stat.c)
extern int sys_pause();         // 29 - 暂停进程运行。              (kernel/sched.c)
extern int sys_utime();         // 30 - 改变文件的访问和修改时间。(fs/open.c)
extern int sys_stty();          // 31 - 修改终端行设置。            (kernel/sys.c)*
extern int sys_gtty();          // 32 - 取终端行设置信息。          (kernel/sys.c)*
extern int sys_access();        // 33 - 检查用户对一个文件的访问权限。(fs/open.c)
extern int sys_nice();          // 34 - 设置进程执行优先权。        (kernel/sched.c)
extern int sys_ftime();         // 35 - 取日期和时间。              (kernel/sys.c)*
extern int sys_sync();          // 36 - 同步高速缓冲与设备中数据。(fs/buffer.c)
extern int sys_kill();          // 37 - 终止一个进程。              (kernel/exit.c)
extern int sys_rename();        // 38 - 更改文件名。                (kernel/sys.c)*
extern int sys_mkdir();         // 39 - 创建目录。                  (fs/namei.c)
extern int sys_rmdir();         // 40 - 删除目录。                  (fs/namei.c)
extern int sys_dup();           // 41 - 复制文件句柄。              (fs/fcntl.c)
```

```
47 extern int sys_pipe();          // 42 - 创建管道。                    (fs/pipe.c)
48 extern int sys_times();         // 43 - 取运行时间。                  (kernel/sys.c)
49 extern int sys_prof();          // 44 - 程序执行时间区域。            (kernel/sys.c)*
50 extern int sys_brk();           // 45 - 修改数据段长度。              (kernel/sys.c)
51 extern int sys_setgid();        // 46 - 设置进程组 id。               (kernel/sys.c)
52 extern int sys_getgid();        // 47 - 取进程组 id。                 (kernel/sched.c)
53 extern int sys_signal();        // 48 - 信号处理。                    (kernel/signal.c)
54 extern int sys_geteuid();       // 49 - 取进程有效用户 id。           (kenrl/sched.c)
55 extern int sys_getegid();       // 50 - 取进程有效组 id。             (kenrl/sched.c)
56 extern int sys_acct();          // 51 - 进程记帐。                    (kernel/sys.c)*
57 extern int sys_phys();          // 52 -                               (kernel/sys.c)*
58 extern int sys_lock();          // 53 -                               (kernel/sys.c)*
59 extern int sys_ioctl();         // 54 - 设备输入输出控制。            (fs/ioctl.c)
60 extern int sys_fcntl();         // 55 - 文件句柄控制操作。            (fs/fcntl.c)
61 extern int sys_mpx();           // 56 -                               (kernel/sys.c)*
62 extern int sys_setpgid();       // 57 - 设置进程组 id。               (kernel/sys.c)
63 extern int sys_ulimit();        // 58 - 统计进程使用资源情况。        (kernel/sys.c)
64 extern int sys_uname();         // 59 - 显示系统信息。                (kernel/sys.c)
65 extern int sys_umask();         // 60 - 取默认文件创建属性码。        (kernel/sys.c)
66 extern int sys_chroot();        // 61 - 改变根目录。                  (fs/open.c)
67 extern int sys_ustat();         // 62 - 取文件系统信息。              (fs/open.c)
68 extern int sys_dup2();          // 63 - 复制文件句柄。                (fs/fcntl.c)
69 extern int sys_getppid();       // 64 - 取父进程 id。                 (kernel/sched.c)
70 extern int sys_getpgrp();       // 65 - 取进程组 id，等于 getpgid(0)。(kernel/sys.c)
71 extern int sys_setsid();        // 66 - 在新会话中运行程序。          (kernel/sys.c)
72 extern int sys_sigaction();     // 67 - 改变信号处理过程。            (kernel/signal.c)
73 extern int sys_sgetmask();      // 68 - 取信号屏蔽码。                (kernel/signal.c)
74 extern int sys_ssetmask();      // 69 - 设置信号屏蔽码。              (kernel/signal.c)
75 extern int sys_setreuid();      // 70 - 设置真实与/或有效用户 id。    (kernel/sys.c)
76 extern int sys_setregid();      // 71 - 设置真实与/或有效组 id。      (kernel/sys.c)
77 extern int sys_sigpending();    // 73 - 检查暂未处理的信号。          (kernel/signal.c)
78 extern int sys_sigsuspend();    // 72 - 使用新屏蔽码挂起进程。        (kernel/signal.c)
79 extern int sys_sethostname();   // 74 - 设置主机名。                  (kernel/sys.c)
80 extern int sys_setrlimit();     // 75 - 设置资源使用限制。            (kernel/sys.c)
81 extern int sys_getrlimit();     // 76 - 取得进程使用资源的限制。      (kernel/sys.c)
82 extern int sys_getrusage();     // 77 -
83 extern int sys_gettimeofday();  // 78 - 获取当日时间。                (kernel/sys.c)
84 extern int sys_settimeofday();  // 79 - 设置当日时间。                (kernel/sys.c)
85 extern int sys_getgroups();     // 80 - 取得进程所有组标识号。        (kernel/sys.c)
86 extern int sys_setgroups();     // 81 - 设置进程组标识号数组。        (kernel/sys.c)
87 extern int sys_select();        // 82 - 等待文件描述符状态改变。      (fs/select.c)
88 extern int sys_symlink();       // 83 - 建立符号链接。                (fs/namei.c，767)
89 extern int sys_lstat();         // 84 - 取符号链接文件状态。          (fs/stat.c，47)
90 extern int sys_readlink();      // 85 - 读取符号链接文件信息。        (fs/stat.c，69)
91 extern int sys_uselib();        // 86 - 选择共享库。                  (fs/exec.c，42)
92
   // 系统调用函数指针表。用于系统调用中断处理程序(int 0x80)，作为跳转表。
93 fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
94 sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
95 sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
96 sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
97 sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
98 sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
```

```
 99 sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
100 sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
101 sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
102 sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
103 sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
104 sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
105 sys_setreuid,sys_setregid, sys_sigsuspend, sys_sigpending, sys_sethostname,
106 sys_setrlimit, sys_getrlimit, sys_getrusage, sys_gettimeofday,
107 sys_settimeofday, sys_getgroups, sys_setgroups, sys_select, sys_symlink,
108 sys_lstat, sys_readlink, sys_uselib };
109
110 /* So we don't have to do any more manual updating.... */
    /* 下面这样定义后，我们就无需手工更新系统调用数目了 */
111 int NR_syscalls = sizeof(sys_call_table)/sizeof(fn_ptr);
112
```

14.28 程序 14-28 linux/include/linux/tty.h

```
 1 /*
 2  * 'tty.h' defines some structures used by tty_io.c and some defines.
 3  *
 4  * NOTE! Don't touch this without checking that nothing in rs_io.s or
 5  * con_io.s breaks. Some constants are hardwired into the system (mainly
 6  * offsets into 'tty_queue'
 7  */
   /*
    * 'tty.h'中定义了 tty_io.c 程序使用的某些结构和其他一些定义。
    *
    * 注意！在修改这里的定义时，一定要检查 rs_io.s 或 con_io.s 程序中不会出现问题。
    * 在系统中有些常量是直接写在程序中的（主要是一些 tty_queue 中的偏移值）。
    */
 8
 9 #ifndef _TTY_H
10 #define _TTY_H
11
12 #define MAX_CONSOLES    8           // 最大虚拟控制台数量。
13 #define NR_SERIALS      2           // 串行终端数量。
14 #define NR_PTYS         4           // 伪终端数量。
15
16 extern int NR_CONSOLES;             // 虚拟控制台数量。
17
18 #include <termios.h>        // 终端输入输出函数头文件。主要定义控制异步通信口的终端接口。
19
20 #define TTY_BUF_SIZE 1024           // tty 缓冲区（缓冲队列）大小。
21
   // tty 字符缓冲队列数据结构。用于 tty_struc 结构中的读、写和辅助（规范）缓冲队列。
22 struct tty_queue {
23         unsigned long data;                 // 队列缓冲区中含有字符行数值（不是当前字符数）。
                                               // 对于串口终端，则存放串行端口地址。
24         unsigned long head;                 // 缓冲区中数据头指针。
25         unsigned long tail;                 // 缓冲区中数据尾指针。
26         struct task_struct * proc_list;     // 等待本队列的进程列表。
27         char buf[TTY_BUF_SIZE];             // 队列的缓冲区。
28 };
29
30 #define IS_A_CONSOLE(min)       (((min) & 0xC0) == 0x00)        // 是一个控制终端。
31 #define IS_A_SERIAL(min)        (((min) & 0xC0) == 0x40)        // 是一个串行终端。
32 #define IS_A_PTY(min)           ((min) & 0x80)                  // 是一个伪终端。
33 #define IS_A_PTY_MASTER(min)    (((min) & 0xC0) == 0x80)        // 是一个主伪终端。
34 #define IS_A_PTY_SLAVE(min)     (((min) & 0xC0) == 0xC0)        // 是一个辅伪终端。
35 #define PTY_OTHER(min)          ((min) ^ 0x40)                  // 其他伪终端。
36
   // 以下定义了 tty 等待队列中缓冲区操作宏函数。（tail 在前，head 在后，参见 tty_io.c 的图）。
   // a 缓冲区指针前移 1 字节，若已超出缓冲区右侧，则指针循环。
37 #define INC(a) ((a) = ((a)+1) & (TTY_BUF_SIZE-1))
   // a 缓冲区指针后退 1 字节，并循环。
38 #define DEC(a) ((a) = ((a)-1) & (TTY_BUF_SIZE-1))
```

```
    // 清空指定队列的缓冲区。
39  #define EMPTY(a) ((a)->head == (a)->tail)
    // 缓冲区还可存放字符的长度（空闲区长度）。
40  #define LEFT(a) (((a)->tail-(a)->head-1)&(TTY_BUF_SIZE-1))
    // 缓冲区中最后一个位置。
41  #define LAST(a) ((a)->buf[(TTY_BUF_SIZE-1)&((a)->head-1)])
    // 缓冲区满（如果为 1 的话）。
42  #define FULL(a) (!LEFT(a))
    // 缓冲区中已存放字符的长度（字符数）。
43  #define CHARS(a) (((a)->head-(a)->tail)&(TTY_BUF_SIZE-1))
    // 从 queue 队列项缓冲区中取一字符(从 tail 处，并且 tail+=1)。
44  #define GETCH(queue,c) \
45  (void)({c=(queue)->buf[(queue)->tail];INC((queue)->tail);})
    // 往 queue 队列项缓冲区中放置一字符（在 head 处，并且 head+=1）。
46  #define PUTCH(c,queue) \
47  (void)({(queue)->buf[(queue)->head]=(c);INC((queue)->head);})
48
    // 判断终端键盘字符类型。
49  #define INTR_CHAR(tty) ((tty)->termios.c_cc[VINTR])     // 中断符。发中断信号 SIGINT。
50  #define QUIT_CHAR(tty) ((tty)->termios.c_cc[VQUIT])     // 退出符。发退出信号 SIGQUIT。
51  #define ERASE_CHAR(tty) ((tty)->termios.c_cc[VERASE])   // 削除符。擦除一个字符。
52  #define KILL_CHAR(tty) ((tty)->termios.c_cc[VKILL])     // 删除行。删除一行字符。
53  #define EOF_CHAR(tty) ((tty)->termios.c_cc[VEOF])       // 文件结束符。
54  #define START_CHAR(tty) ((tty)->termios.c_cc[VSTART])   // 开始符。恢复输出。
55  #define STOP_CHAR(tty) ((tty)->termios.c_cc[VSTOP])     // 停止符。停止输出。
56  #define SUSPEND_CHAR(tty) ((tty)->termios.c_cc[VSUSP])  // 挂起符。发挂起信号 SIGTSTP。
57
    // tty 数据结构。
58  struct tty_struct {
59          struct termios termios;                 // 终端 io 属性和控制字符数据结构。
60          int pgrp;                               // 所属进程组。
61          int session;                            // 会话号。
62          int stopped;                            // 停止标志。
63          void (*write)(struct tty_struct * tty); // tty 写函数指针。
64          struct tty_queue *read_q;               // tty 读队列。
65          struct tty_queue *write_q;              // tty 写队列。
66          struct tty_queue *secondary;            // tty 辅助队列(存放规范模式字符序列),
67          };                                      // 可称为规范(熟)模式队列。
68
69  extern struct tty_struct tty_table[];           // tty 结构数组。
70  extern int fg_console;                          // 前台控制台号。
71
    // 根据终端类型在 tty_table[] 中取对应终端号 nr 的 tty 结构指针。第 73 行后半部分用于
    // 根据子设备号 dev 在 tty_table[]表中选择对应的 tty 结构。如果 dev = 0，表示正在使用
    // 前台终端，因此直接使用终端号 fg_console 作为 tty_table[] 项索引取 tty 结构。如果
    // dev 大于 0，那么就要分两种情况考虑：① dev 是虚拟终端号；② dev 是串行终端号或者
    // 伪终端号。对于虚拟终端其 tty 结构在 tty_table[]中索引项是 dev-1（0 -- 63）。对于
    // 其它类型终端，则它们的 tty 结构索引项就是 dev。例如，如果 dev = 64，表示是一个串
    // 行终端 1，则其 tty 结构就是 ttb_table[dev]。 如果 dev = 1，则对应终端的 tty 结构是
    // tty_table[0]。参见 tty_io.c 程序第 70 -- 73 行。
72  #define TTY_TABLE(nr) \
73  (tty_table + ((nr) ? (((nr) < 64)? (nr)-1:(nr)) : fg_console))
74
```

```
    // 这里给出了终端 termios 结构中可更改的特殊字符数组 c_cc[]的初始值。该 termios 结构
    // 定义在 include/termios.h 中。POSIX.1 定义了 11 个特殊字符，但是 Linux 系统还另外定
    // 义了 SVR4 使用的 6 个特殊字符。如果定义了_POSIX_VDISABLE（\0），那么当某一项值等
    // 于_POSIX_VDISABLE 的值时，表示禁止使用相应的特殊字符。[8 进制值]
 75 /*      intr=^C         quit=^|         erase=del       kill=^U
 76         eof=^D          vtime=\0        vmin=\1         sxtc=\0
 77         start=^Q        stop=^S         susp=^Z         eol=\0
 78         reprint=^R      discard=^U      werase=^W       lnext=^V
 79         eol2=\0
 80 */
    /* 中断 intr=^C      退出 quit=^|      删除 erase=del      终止 kill=^U
     * 文件结束 eof=^D   vtime=\0          vmin=\1           sxtc=\0
     * 开始 start=^Q     停止 stop=^S      挂起 susp=^Z        行结束 eol=\0
     * 重显 reprint=^R   丢弃 discard=^U   werase=^W          lnext=^V
     * 行结束 eol2=\0
     */
 81 #define INIT_C_CC "\003\034\177\025\004\0\1\0\021\023\032\0\022\017\027\026\0"
 82
 83 void rs_init(void);        // 异步串行通信初始化。(kernel/chr_drv/serial.c)
 84 void con_init(void);       // 控制终端初始化。     (kernel/chr_drv/console.c)
 85 void tty_init(void);       // tty 初始化。          (kernel/chr_drv/tty_io.c)
 86
 87 int tty_read(unsigned c, char * buf, int n);   // (kernel/chr_drv/tty_io.c)
 88 int tty_write(unsigned c, char * buf, int n);  // (kernel/chr_drv/tty_io.c)
 89
 90 void con_write(struct tty_struct * tty);       // (kernel/chr_drv/console.c)
 91 void rs_write(struct tty_struct * tty);        // (kernel/chr_drv/serial.c)
 92 void mpty_write(struct tty_struct * tty);      // (kernel/chr_drv/pty.c)
 93 void spty_write(struct tty_struct * tty);      // (kernel/chr_drv/pty.c)
 94
 95 void copy_to_cooked(struct tty_struct * tty);  // (kernel/chr_drv/tty_io.c)
 96
 97 void update_screen(void);                      // (kernel/chr_drv/console.c)
 98
 99 #endif
100
```

14.29 程序 14-29 linux/include/sys/param.h

```
 1 #ifndef _SYS_PARAM_H
 2 #define _SYS_PARAM_H
 3
 4 #define HZ 100                  // 系统时钟频率，每秒中断 100 次。
 5 #define EXEC_PAGESIZE 4096      // 页面大小。
 6
 7 #define NGROUPS          32     /* Max number of groups per user */ /* 每个进程最多组号*/
 8 #define NOGROUP          -1
 9
10 #define MAXHOSTNAMELEN 8        // 主机名最大长度，8 字节。
11
12 #endif
13
```

14.30 程序 14-30 linux/include/sys/resource.h

```
 1 /*
 2  * Resource control/accounting header file for linux
 3  */
   /*
    * Linux 资源控制/审计头文件。
 4
 5 #ifndef _SYS_RESOURCE_H
 6 #define _SYS_RESOURCE_H
 7
   // 以下符号常数和结构用于 getrusage()。参见 kernel/sys.c 文件第 412 行开始。
 8 /*
 9  * Definition of struct rusage taken from BSD 4.3 Reno
10  *
11  * We don't support all of these yet, but we might as well have them....
12  * Otherwise, each time we add new items, programs which depend on this
13  * structure will lose.  This reduces the chances of that happening.
14  */
   /*
    * rusage 结构的定义取自 BSD 4.3 Reno 系统。
    *
    * 我们现在还没有支持该结构中的所有这些字段，但我们可能会支持它们的....
    * 否则的话，每当我们增加新的字段，那些依赖于这个结构的程序就会出问题。
    * 现在把所有字段都包括进来就可以避免这种事情发生。
    */
   // 下面是 getrusage()的参数 who 所使用的符号常数。
15 #define RUSAGE_SELF     0     // 返回当前进程的资源利用信息。
16 #define RUSAGE_CHILDREN -1    // 返回当前进程已终止和等待着的子进程的资源利用信息。
17
   // rusage 是进程的资源利用统计结构，用于 getrusage()返回指定进程对资源利用的统计值。
   // Linux 0.12 内核仅使用了前两个字段，它们都是 timeval 结构（include/sys/time.h）。
   // ru_utime - 进程在用户态运行时间统计值；ru_stime - 进程在内核态运行时间统计值。
18 struct  rusage {
19         struct timeval ru_utime;        /* user time used */
20         struct timeval ru_stime;        /* system time used */
21         long    ru_maxrss;              /* maximum resident set size */
22         long    ru_ixrss;               /* integral shared memory size */
23         long    ru_idrss;               /* integral unshared data size */
24         long    ru_isrss;               /* integral unshared stack size */
25         long    ru_minflt;              /* page reclaims */
26         long    ru_majflt;              /* page faults */
27         long    ru_nswap;               /* swaps */
28         long    ru_inblock;             /* block input operations */
29         long    ru_oublock;             /* block output operations */
30         long    ru_msgsnd;              /* messages sent */
31         long    ru_msgrcv;              /* messages received */
32         long    ru_nsignals;            /* signals received */
33         long    ru_nvcsw;               /* voluntary context switches */
34         long    ru_nivcsw;              /* involuntary " */
35 };
```

```
36
   // 下面是 getrlimit()和 setrlimit()使用的符号常数和结构。
37 /*
38  * Resource limits
39  */
   /*
    * 资源限制。
    */
   // 以下是 Linux 0.12 内核中所定义的资源种类，是 getrlimit()和 setrlimit()中第 1 个参数
   // resource 的取值范围。 其实这些符号常数就是进程任务结构中 rlim[] 数组的项索引值。
   // rlim[]数组的每一项都是一个 rlimit 结构，该结构见下面第 58 行。
40
41 #define RLIMIT_CPU      0           /* CPU time in ms */        /* 使用的 CPU 时间 */
42 #define RLIMIT_FSIZE    1           /* Maximum filesize */      /* 最大文件长度 */
43 #define RLIMIT_DATA     2           /* max data size */         /* 最大数据长度 */
44 #define RLIMIT_STACK    3           /* max stack size */        /* 最大栈长度 */
45 #define RLIMIT_CORE     4           /* max core file size */    /* 最大 core 文件长度 */
46 #define RLIMIT_RSS      5           /* max resident set size */ /* 最大驻留集大小 */
47
   // 如果定义了符号 notdef，则也包括以下符号常数定义。
48 #ifdef notdef
49 #define RLIMIT_MEMLOCK  6           /* max locked-in-memory address space*/ /* 锁定区 */
50 #define RLIMIT_NPROC    7           /* max number of processes */   /* 最大子进程数 */
51 #define RLIMIT_OFILE    8           /* max number of open files */  /* 最大打开文件数 */
52 #endif
53
   // 这个符号常数定义了 Linux 中限制的资源种类。RLIM_NLIMITS=6，因此仅前面 6 项有效。
54 #define RLIM_NLIMITS    6
55
   // 表示资源无限，或不能修改。
56 #define RLIM_INFINITY   0x7fffffff
57
   // 资源界限结构。
58 struct rlimit {
59         int     rlim_cur;           // 当前资源限制，或称软限制（soft limit）。
60         int     rlim_max;           // 硬限制（hard limit）。
61 };
62
63 #endif /* _SYS_RESOURCE_H */
64
```

14.31 程序 14-31 linux/include/sys/stat.h

```
 1 #ifndef _SYS_STAT_H
 2 #define _SYS_STAT_H
 3
 4 #include <sys/types.h>
 5
 6 struct stat {
 7         dev_t   st_dev;       // 含有文件的设备号。
 8         ino_t   st_ino;       // 文件 i 节点号。
 9         umode_t st_mode;      // 文件类型和属性（见下面）。
10         nlink_t st_nlink;     // 指定文件的连接数。
11         uid_t   st_uid;       // 文件的用户(标识)号。
12         gid_t   st_gid;       // 文件的组号。
13         dev_t   st_rdev;      // 设备号(如果文件是特殊的字符文件或块文件)。
14         off_t   st_size;      // 文件大小（字节数）（如果文件是常规文件）。
15         time_t  st_atime;     // 上次（最后）访问时间。
16         time_t  st_mtime;     // 最后修改时间。
17         time_t  st_ctime;     // 最后节点修改时间。
18 };
19
   //
   // 下面是为 st_mode 字段所用的值定义的符号名称。这些值均用八进制表示。参见第 12 章文件
   // 系统中图 12-5（i 节点属性字段内容）。 为便于记忆，这些符号名称均为一些英文单词的首
   // 字母或缩写组合而成。 例如名称 S_IFMT 的每个字母分别代表单词 State、Inode、File、
   // Mask 和 Type；而名称 S_IFREG 则是 State、Inode、File 和 REGular 几个大写字母的组合；
   // 名称 S_IRWXU 是 State、Inode、Read、Write、eXecute 和 User 的组合。其它名称可以此类推。
   // 文件类型：
20 #define S_IFMT  00170000      // 文件类型比特位屏蔽码（8 进制表示）。
21 #define S_IFLNK  0120000      // 符号链接。
22 #define S_IFREG  0100000      // 常规文件。
23 #define S_IFBLK  0060000      // 块特殊（设备）文件，如磁盘 dev/fd0。
24 #define S_IFDIR  0040000      // 目录。
25 #define S_IFCHR  0020000      // 字符设备文件。
26 #define S_IFIFO  0010000      // FIFO 特殊文件。
   // 文件属性位：
   // S_ISUID 用于测试文件的 set-user-ID 标志是否置位。若该标志置位，则当执行该文件时，进程的
   // 有效用户 ID 将被设置为该文件宿主的用户 ID。S_ISGID 则是针对组 ID 进行相同处理。
27 #define S_ISUID  0004000    // 执行时设置用户 ID（set-user-ID）。
28 #define S_ISGID  0002000    // 执行时设置组 ID（set-group-ID）。
29 #define S_ISVTX  0001000    // 对于目录，受限删除标志。
30
31 #define S_ISLNK(m)      (((m) & S_IFMT) == S_IFLNK)   // 测试是否符号链接文件。
32 #define S_ISREG(m)      (((m) & S_IFMT) == S_IFREG)   // 测试是否常规文件。
33 #define S_ISDIR(m)      (((m) & S_IFMT) == S_IFDIR)   // 是否目录文件。
34 #define S_ISCHR(m)      (((m) & S_IFMT) == S_IFCHR)   // 是否字符设备文件。
35 #define S_ISBLK(m)      (((m) & S_IFMT) == S_IFBLK)   // 是否块设备文件。
36 #define S_ISFIFO(m)     (((m) & S_IFMT) == S_IFIFO)   // 是否 FIFO 特殊文件。
37
   // 文件访问权限：
38 #define S_IRWXU 00700       // 宿主可以读、写、执行/搜索（名称最后字母代表 User）。
```

```
39 #define S_IRUSR 00400      // 宿主读许可。
40 #define S_IWUSR 00200      // 宿主写许可。
41 #define S_IXUSR 00100      // 宿主执行/搜索许可。
42
43 #define S_IRWXG 00070      // 组成员可以读、写、执行/搜索（名称最后字母代表 Group）。
44 #define S_IRGRP 00040      // 组成员读许可。
45 #define S_IWGRP 00020      // 组成员写许可。
46 #define S_IXGRP 00010      // 组成员执行/搜索许可。
47
48 #define S_IRWXO 00007      // 其他人读、写、执行/搜索许可（名称最后字母 O 代表 Other）。
49 #define S_IROTH 00004      // 其他人读许可（最后 3 个字母代表 Other）。
50 #define S_IWOTH 00002      // 其他人写许可。
51 #define S_IXOTH 00001      // 其他人执行/搜索许可。
52
53 extern int chmod(const char *_path, mode_t mode);     // 修改文件属性。
54 extern int fstat(int fildes, struct stat *stat_buf);  // 取指定文件句柄的文件状态信息。
55 extern int mkdir(const char *_path, mode_t mode);     // 创建目录。
56 extern int mkfifo(const char *_path, mode_t mode);    // 创建管道文件。
57 extern int stat(const char *filename, struct stat *stat_buf); // 取指定文件名的文件状态信息。
58 extern mode_t umask(mode_t mask);  // 设置属性屏蔽码。
59
60 #endif
61
```

14.32 程序 14-32 linux/include/sys/time.h

```
 1 #ifndef _SYS_TIME_H
 2 #define _SYS_TIME_H
 3
 4 /* gettimofday returns this */                    // gettimeofday()函数返回该时间结构。
 5 struct timeval {
 6         long    tv_sec;         /* seconds */       // 秒。
 7         long    tv_usec;        /* microseconds */  // 微秒。
 8 };
 9
   // 时间区结构。tz 为时区（Time Zone）的缩写，DST（Daylight Saving Time）是夏令时的缩写。
10 struct timezone {
11         int     tz_minuteswest; /* minutes west of Greenwich */   // 格林威治西部分钟时间。
12         int     tz_dsttime;     /* type of dst correction */      // 夏令时区调整时间。
13 };
14
15 #define DST_NONE        0       /* not on dst */                  // 非夏令时。
16 #define DST_USA         1       /* USA style dst */               // USA 形式的夏令时。
17 #define DST_AUST        2       /* Australian style dst */        // 澳洲形式的夏令时。
18 #define DST_WET         3       /* Western European dst */
19 #define DST_MET         4       /* Middle European dst */
20 #define DST_EET         5       /* Eastern European dst */
21 #define DST_CAN         6       /* Canada */
22 #define DST_GB          7       /* Great Britain and Eire */
23 #define DST_RUM         8       /* Rumania */
24 #define DST_TUR         9       /* Turkey */
25 #define DST_AUSTALT     10      /* Australian style with shift in 1986 */
26
   // 文件描述符集的设置宏，用于 select()函数。
27 #define FD_SET(fd,fdsetp)       (*(fdsetp) |= (1 << (fd)))
28 #define FD_CLR(fd,fdsetp)       (*(fdsetp) &= ~(1 << (fd)))
29 #define FD_ISSET(fd,fdsetp)     ((*(fdsetp) >> fd) & 1)
30 #define FD_ZERO(fdsetp)         (*(fdsetp) = 0)
31
32 /*
33  * Operations on timevals.
34  *
35  * NB: timercmp does not work for >= or <=.
36  */
   // timeval 时间结构的操作函数。
37 #define timerisset(tvp)         ((tvp)->tv_sec || (tvp)->tv_usec)
38 #define timercmp(tvp, uvp, cmp) \
39         ((tvp)->tv_sec cmp (uvp)->tv_sec || \
40          (tvp)->tv_sec == (uvp)->tv_sec && (tvp)->tv_usec cmp (uvp)->tv_usec)
41 #define timerclear(tvp)         ((tvp)->tv_sec = (tvp)->tv_usec = 0)
42
43 /*
44  * Names of the interval timers, and structure
45  * defining a timer setting.
46  */
```

```
   /* 内部定时器名称和结构，用于定义定时器设置。 */
47 #define ITIMER_REAL     0               // 以实际时间递减。
48 #define ITIMER_VIRTUAL  1               // 以进程虚拟时间递减。
49 #define ITIMER_PROF     2               // 以进程虚拟时间或者当系统运行时以进程时间递减。
50
   // 内部时间结构。其中 it（Internal Timer）是内部定时器的缩写。
51 struct  itimerval {
52         struct  timeval it_interval;    /* timer interval */
53         struct  timeval it_value;       /* current value */
54 };
55
56 #include <time.h>
57 #include <sys/types.h>
58
59 int gettimeofday(struct timeval * tp, struct timezone * tz);
60 int select(int width, fd_set * readfds, fd_set * writefds,
61         fd_set * exceptfds, struct timeval * timeout);
62
63 #endif /*_SYS_TIME_H*/
64
```

14.33 程序 14-33 linux/include/sys/times.h

```
 1 #ifndef _TIMES_H
 2 #define _TIMES_H
 3
 4 #include <sys/types.h>     // 类型头文件。定义了基本的系统数据类型。
 5
 6 struct tms {
 7         time_t tms_utime;  // 用户使用的 CPU 时间。
 8         time_t tms_stime;  // 系统（内核）CPU 时间。
 9         time_t tms_cutime; // 已终止的子进程使用的用户 CPU 时间。
10         time_t tms_cstime; // 已终止的子进程使用的系统 CPU 时间。
11 };
12
13 extern time_t times(struct tms * tp);
14
15 #endif
16
```

14.34 程序 14-34 linux/include/sys/types.h

```
 1 #ifndef _SYS_TYPES_H
 2 #define _SYS_TYPES_H
 3
 4 #ifndef _SIZE_T
 5 #define _SIZE_T
 6 typedef unsigned int size_t;        // 用于对象的大小（长度）。
 7 #endif
 8
 9 #ifndef _TIME_T
10 #define _TIME_T
11 typedef long time_t;                // 用于时间（以秒计）。
12 #endif
13
14 #ifndef _PTRDIFF_T
15 #define _PTRDIFF_T
16 typedef long ptrdiff_t;
17 #endif
18
19 #ifndef NULL
20 #define NULL ((void *) 0)
21 #endif
22
23 typedef int pid_t;                  // 用于进程号和进程组号。
24 typedef unsigned short uid_t;       // 用于用户号（用户标识号）。
25 typedef unsigned char gid_t;        // 用于组号。
26 typedef unsigned short dev_t;       // 用于设备号。
27 typedef unsigned short ino_t;       // 用于文件序列号。
28 typedef unsigned short mode_t;      // 用于某些文件属性。
29 typedef unsigned short umode_t;     //
30 typedef unsigned char nlink_t;      // 用于连接计数。
31 typedef int daddr_t;
32 typedef long off_t;                 // 用于文件长度（大小）。
33 typedef unsigned char u_char;       // 无符号字符类型。
34 typedef unsigned short ushort;      // 无符号短整数类型。
35
36 typedef unsigned char cc_t;
37 typedef unsigned int speed_t;
38 typedef unsigned long tcflag_t;
39
40 typedef unsigned long fd_set;       // 文件描述符集。每比特代表 1 个描述符。
41
42 typedef struct { int quot,rem; } div_t;     // 用于 DIV 操作。
43 typedef struct { long quot,rem; } ldiv_t;   // 用于长 DIV 操作。
44
   // 文件系统参数结构，用于 ustat()函数。最后两个字段未使用，总是返回 NULL 指针。
45 struct ustat {
46         daddr_t f_tfree;              // 系统总空闲块数。
47         ino_t f_tinode;               // 总空闲 i 节点数。
48         char f_fname[6];              // 文件系统名称。
```

```
49          char f_fpack[6];                // 文件系统压缩名称。
50 };
51
52 #endif
53
```

14.35 程序 14-35 linux/include/sys/utsname.h

```
 1 #ifndef _SYS_UTSNAME_H
 2 #define _SYS_UTSNAME_H
 3
 4 #include <sys/types.h>     // 类型头文件。定义了基本的系统数据类型。
 5 #include <sys/param.h>     // 内核参数文件。
 5
 6 struct utsname {
 7         char sysname[9];   // 当前运行系统的名称。
 8         char nodename[MAXHOSTNAMELEN+1];  // 与实现相关的网络中节点名称（主机名称）。
 9         char release[9];   // 本操作系统实现的当前发行级别。
10         char version[9];   // 本次发行的操作系统版本级别。
11         char machine[9];   // 系统运行的硬件类型名称。
12 };
13
14 extern int uname(struct utsname * utsbuf);
15
16 #endif
17
```

14.36 程序 14-36 linux/include/sys/wait.h

```
 1 #ifndef _SYS_WAIT_H
 2 #define _SYS_WAIT_H
 3
 4 #include <sys/types.h>
 5
 6 #define _LOW(v)         ( (v) & 0377)           // 取低字节（8 进制表示）。
 7 #define _HIGH(v)        ( ((v) >> 8) & 0377)    // 取高字节。
 8
 9 /* options for waitpid, WUNTRACED not supported */
   /* waitpid 的选项，其中 WUNTRACED 未被支持 */
   // [ 注：其实 0.12 内核已经支持 WUNTRACED 选项。上面这条注释应该是以前内核版本遗留下来的。 ]
   // 以下常数符号是函数 waitpid(pid_t pid, long *stat_addr, int options)中 options 使用的选项。
10 #define WNOHANG         1       // 如果没有状态也不要挂起，并立刻返回。
11 #define WUNTRACED       2       // 报告停止执行的子进程状态。
12
   // 以下宏定义用于判断 waitpid()函数返回的状态字（第 20、21 行的参数*stat_loc）的含义。
13 #define WIFEXITED(s)    (!((s)&0xFF)        // 如果子进程正常退出，则为真。
14 #define WIFSTOPPED(s)   (((s)&0xFF)==0x7F) // 如果子进程正停止着，则为 true。
15 #define WEXITSTATUS(s)  (((s)>>8)&0xFF)     // 返回退出状态。
16 #define WTERMSIG(s)     ((s)&0x7F)          // 返回导致进程终止的信号值（信号量）。
17 #define WCOREDUMP(s)    ((s)&0x80)          // 判断进程是否执行了内存映像转储（dumpcore）。
18 #define WSTOPSIG(s)     (((s)>>8)&0xFF)     // 返回导致进程停止的信号值。
19 #define WIFSIGNALED(s)  (((unsigned int)(s)-1 & 0xFFFF) < 0xFF) // 如果由于未捕捉信号而
                                                                   // 导致子进程退出则为真。
20
   // wait()和 waitpit()函数允许进程获取与其子进程之一的状态信息。各种选项允许获取已经终止或
   // 停止的子进程状态信息。如果存在两个或两个以上子进程的状态信息，则报告的顺序是不指定的。
   // wait()将挂起当前进程，直到其子进程之一退出（终止），或者收到要求终止该进程的信号，
   // 或者是需要调用一个信号句柄（信号处理程序）。
   // waitpid()挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止该进程的信号，
   // 或者是需要调用一个信号句柄（信号处理程序）。
   // 如果 pid= -1，options=0，则 waitpid()的作用与 wait()函数一样。否则其行为将随 pid 和 options
   // 参数的不同而不同。（参见 kernel/exit.c,142）
   // 参数 pid 是进程号；*stat_loc 是保存状态信息位置的指针；options 是等待选项，见第 10，11 行。
21 pid_t wait(int *stat_loc);
22 pid_t waitpid(pid_t pid, int *stat_loc, int options);
23
24 #endif
25
```

第15章 内核库函数程序

15.1 程序 15-1 linux/lib/_exit.c

```
 1 /*
 2  *  linux/lib/_exit.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #define __LIBRARY__           // 定义一个符号常量，见下行说明。
 8 #include <unistd.h>           // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                                 // 若定义了__LIBRARY__，则还含系统调用号和内嵌汇编 syscall0()等。
 9
   //// 内核使用的程序(退出)终止函数。
   // 直接调用系统中断 int 0x80，功能号__NR_exit。
   // 参数：exit_code - 退出码。
   // 函数名前的关键字 volatile 用于告诉编译器 gcc 该函数不会返回。这样可让 gcc 产生更好一
   // 些的代码，更重要的是使用这个关键字可以避免产生某些（未初始化变量的）假警告信息。
   // 等同于 gcc 的函数属性说明：void do_exit(int error_code) __attribute__ ((noreturn));
10 volatile void _exit(int exit_code)
11 {
   // %0 - eax(系统调用号__NR_exit)；%1 - ebx(退出码 exit_code)。
12         __asm__("int $0x80"::"a" (__NR_exit),"b" (exit_code));
13 }
14
```

15.2 程序 15-2 linux/lib/close.c

```
 1 /*
 2  *  linux/lib/close.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #define __LIBRARY__
 8 #include <unistd.h>        // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                              // 如定义了__LIBRARY__，则还含系统调用号和内嵌汇编 syscall0()等。
 9
   // 关闭文件函数。
   // 下面该调用宏函数对应：int close(int fd)。直接调用了系统中断 int 0x80，参数是__NR_close。
   // 其中 fd 是文件描述符。
10 _syscall1(int,close,int,fd)
11
```

15.3 程序 15-3 linux/lib/ctype.c

```
1  /*
2   *  linux/lib/ctype.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <ctype.h>          // 字符类型头文件。定义了一些有关字符类型判断和转换的宏。
8
9  char _ctmp;                 // 一个临时字符变量，供 ctype.h 文件中转换字符宏函数使用。
   // 字符特性数组(表)，定义了各个字符对应的属性，这些属性类型(如_C 等)在 ctype.h 中定义。
   // 用于判断字符是控制字符(_C)、大写字符(_U)、小写字符(_L)等所属类型。
10 unsigned char _ctype[] = {0x00,                   /* EOF */
11 _C, _C, _C, _C, _C, _C, _C, _C,                  /* 0-7 */
12 _C, _C|_S, _C|_S, _C|_S, _C|_S, _C|_S, _C, _C,   /* 8-15 */
13 _C, _C, _C, _C, _C, _C, _C, _C,                  /* 16-23 */
14 _C, _C, _C, _C, _C, _C, _C, _C,                  /* 24-31 */
15 _S|_SP, _P, _P, _P, _P, _P, _P, _P,              /* 32-39 */
16 _P, _P, _P, _P, _P, _P, _P, _P,                  /* 40-47 */
17 _D, _D, _D, _D, _D, _D, _D, _D,                  /* 48-55 */
18 _D, _D, _P, _P, _P, _P, _P, _P,                  /* 56-63 */
19 _P, _U|_X, _U|_X, _U|_X, _U|_X, _U|_X, _U|_X, _U,  /* 64-71 */
20 _U, _U, _U, _U, _U, _U, _U, _U,                  /* 72-79 */
21 _U, _U, _U, _U, _U, _U, _U, _U,                  /* 80-87 */
22 _U, _U, _U, _P, _P, _P, _P, _P,                  /* 88-95 */
23 _P, _L|_X, _L|_X, _L|_X, _L|_X, _L|_X, _L|_X, _L,  /* 96-103 */
24 _L, _L, _L, _L, _L, _L, _L, _L,                  /* 104-111 */
25 _L, _L, _L, _L, _L, _L, _L, _L,                  /* 112-119 */
26 _L, _L, _L, _P, _P, _P, _P, _C,                  /* 120-127 */
27 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                 /* 128-143 */
28 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                 /* 144-159 */
29 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                 /* 160-175 */
30 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                 /* 176-191 */
31 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                 /* 192-207 */
32 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                 /* 208-223 */
33 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                 /* 224-239 */
34 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};                /* 240-255 */
35
36
```

15.4 程序 15-4 linux/lib/dup.c

```
 1 /*
 2  *  linux/lib/dup.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #define __LIBRARY__
 8 #include <unistd.h>        // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                              // 如定义了__LIBRARY__，则还含系统调用号和内嵌汇编_syscall0()等。
 9
   //// 复制文件描述符函数。
   // 下面该调用宏函数对应：int dup(int fd)。直接调用了系统中断 int 0x80，参数是__NR_dup。
   // 其中 fd 是文件描述符。
10 _syscall1(int,dup,int,fd)
11
```

15.5 程序 15-5 linux/lib/errno.c

```
1 /*
2  *  linux/lib/errno.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7 int errno;
8
```

15.6 程序 15-6 linux/lib/execve.c

```
 1 /*
 2  *  linux/lib/execve.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #define __LIBRARY__
 8 #include <unistd.h>        // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                              // 如定义了__LIBRARY__，则还含系统调用号和内嵌汇编_syscall0()等。
 9
   //// 加载并执行子进程(其他程序)函数。
   // 下面该调用宏函数对应：int execve(const char * file, char ** argv, char ** envp)。
   // 参数：file - 被执行程序文件名；argv - 命令行参数指针数组；envp - 环境变量指针数组。
   // 直接调用了系统中断 int 0x80，参数是__NR_execve。参见 include/unistd.h 和 fs/exec.c 程序。
10 _syscall3(int,execve,const char *,file,char **,argv,char **,envp)
11
```

15.7 程序 15-7 linux/lib/malloc.c

```
 1 /*
 2  * malloc.c --- a general purpose kernel memory allocator for Linux.
 3  *
 4  * Written by Theodore Ts'o (tytso@mit.edu), 11/29/91
 5  *
 6  * This routine is written to be as fast as possible, so that it
 7  * can be called from the interrupt level.
 8  *
 9  * Limitations: maximum size of memory we can allocate using this routine
10  *      is 4k, the size of a page in Linux.
11  *
12  * The general game plan is that each page (called a bucket) will only hold
13  * objects of a given size.  When all of the object on a page are released,
14  * the page can be returned to the general free pool.  When malloc() is
15  * called, it looks for the smallest bucket size which will fulfill its
16  * request, and allocate a piece of memory from that bucket pool.
17  *
18  * Each bucket has as its control block a bucket descriptor which keeps
19  * track of how many objects are in use on that page, and the free list
20  * for that page.  Like the buckets themselves, bucket descriptors are
21  * stored on pages requested from get_free_page().  However, unlike buckets,
22  * pages devoted to bucket descriptor pages are never released back to the
23  * system.  Fortunately, a system should probably only need 1 or 2 bucket
24  * descriptor pages, since a page can hold 256 bucket descriptors (which
25  * corresponds to 1 megabyte worth of bucket pages.)  If the kernel is using
26  * that much allocated memory, it's probably doing something wrong.  :-)
27  *
28  * Note: malloc() and free() both call get_free_page() and free_page()
29  *      in sections of code where interrupts are turned off, to allow
30  *      malloc() and free() to be safely called from an interrupt routine.
31  *      (We will probably need this functionality when networking code,
32  *      particularily things like NFS, is added to Linux.)  However, this
33  *      presumes that get_free_page() and free_page() are interrupt-level
34  *      safe, which they may not be once paging is added.  If this is the
35  *      case, we will need to modify malloc() to keep a few unused pages
36  *      "pre-allocated" so that it can safely draw upon those pages if
37  *      it is called from an interrupt routine.
38  *
39  *      Another concern is that get_free_page() should not sleep; if it
40  *      does, the code is carefully ordered so as to avoid any race
41  *      conditions.  The catch is that if malloc() is called re-entrantly,
42  *      there is a chance that unecessary pages will be grabbed from the
43  *      system.  Except for the pages for the bucket descriptor page, the
44  *      extra pages will eventually get released back to the system, though,
45  *      so it isn't all that bad.
46  */
47
```

```
/*
 *     malloc.c - Linux 的通用内核内存分配函数。
```

```
 *
 * 由 Theodore Ts'o 编制 (tytso@mit.edu), 11/29/91
 *
 * 该函数被编写成尽可能地快，从而可以从中断层调用此函数。
 *
 * 限制：使用该函数一次所能分配的最大内存是 4k，也即 Linux 中内存页面的大小。
 *
 * 编写该函数所遵循的一般规则是每页(被称为一个存储桶)仅分配所要容纳对象的大小。
 * 当一页上的所有对象都释放后，该页就可以返回通用空闲内存池。当 malloc()被调用
 * 时，它会寻找满足要求的最小的存储桶，并从该存储桶中分配一块内存。
 *
 * 每个存储桶都有一个作为其控制用的存储桶描述符，其中记录了页面上有多少对象正被
 * 使用以及该页上空闲内存的列表。就象存储桶自身一样，存储桶描述符也是存储在使用
 * get_free_page()申请到的页面上的，但是与存储桶不同的是，桶描述符所占用的页面
 * 将不再会释放给系统。幸运的是一个系统大约只需要 1 到 2 页的桶描述符页面，因为一
 * 个页面可以存放 256 个桶描述符(对应 1MB 内存的存储桶页面)。如果系统为桶描述符分
 * 配了许多内存，那么肯定系统什么地方出了问题☺。
 *
 * 注意！ malloc()和 free()两者关闭了中断的代码部分都调用了 get_free_page()和
 *       free_page()函数，以使 malloc()和 free()可以安全地被从中断程序中调用
 *       (当网络代码，尤其是 NFS 等被加入到 Linux 中时就可能需要这种功能)。但前
 *       提是假设 get_free_page()和 free_page()是可以安全地在中断级程序中使用的，
 *       这在一旦加入了分页处理之后就可能不是安全的。如果真是这种情况，那么我们
 *       就需要修改 malloc()来“预先分配”几页不用的内存，如果 malloc()和 free()
 *       被从中断程序中调用时就可以安全地使用这些页面。
 *
 *       另外需要考虑到的是 get_free_page()不应该睡眠；如果会睡眠的话，则为了防止
 *       任何竞争条件，代码需要仔细地安排顺序。 关键在于如果 malloc()是可以重入地
 *       被调用的话，那么就会存在不必要的页面被从系统中取走的机会。除了用于桶描述
 *       符的页面，这些额外的页面最终会释放给系统，所以并不是象想象的那样不好。
 */

#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <linux/mm.h>     // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
#include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

// 存储桶描述符结构。
struct bucket_desc {    /* 16 bytes */
        void                    *page;          // 该桶描述符对应的内存页面指针。
        struct bucket_desc      *next;          // 下一个描述符指针。
        void                    *freeptr;       // 指向本桶中空闲内存位置的指针。
        unsigned short          refcnt;         // 引用计数。
        unsigned short          bucket_size;    // 本描述符对应存储桶的大小。
};

// 存储桶描述符目录结构。
struct _bucket_dir {    /* 8 bytes */
        int                     size;       // 该存储桶的大小(字节数)。
        struct bucket_desc      *chain;     // 该存储桶目录项的桶描述符链表指针。
};

/*
 * The following is the where we store a pointer to the first bucket
```

```
 * descriptor for a given size.
 *
 * If it turns out that the Linux kernel allocates a lot of objects of a
 * specific size, then we may want to add that specific size to this list,
 * since that will allow the memory to be allocated more efficiently.
 * However, since an entire page must be dedicated to each specific size
 * on this list, some amount of temperance must be exercised here.
 *
 * Note that this list *must* be kept in order.
 */
/*
 * 下面是我们存放第一个给定大小存储桶描述符指针的地方。
 *
 * 如果 Linux 内核分配了许多指定大小的对象，那么我们就希望将该指定的大小加到
 * 该列表(链表)中，因为这样可以使内存的分配更有效。但是，因为一页完整内存页面
 * 必须用于列表中指定大小的所有对象，所以需要做总数方面的测试操作。
 */
// 存储桶目录列表(数组)。
struct bucket_dir bucket_dir[] = {
        { 16,   (struct bucket_desc *) 0},      // 16 字节长度的内存块。
        { 32,   (struct bucket_desc *) 0},      // 32 字节长度的内存块。
        { 64,   (struct bucket_desc *) 0},      // 64 字节长度的内存块。
        { 128,  (struct bucket_desc *) 0},      // 128 字节长度的内存块。
        { 256,  (struct bucket_desc *) 0},      // 256 字节长度的内存块。
        { 512,  (struct bucket_desc *) 0},      // 512 字节长度的内存块。
        { 1024, (struct bucket_desc *) 0},      // 1024 字节长度的内存块。
        { 2048, (struct bucket_desc *) 0},      // 2048 字节长度的内存块。
        { 4096, (struct bucket_desc *) 0},      // 4096 字节(1 页)内存。
        { 0,    (struct bucket_desc *) 0}};     /* End of list marker */

/*
 * This contains a linked list of free bucket descriptor blocks
 */
/*
 * 下面是含有空闲桶描述符内存块的链表。
 */
struct bucket_desc *free_bucket_desc = (struct bucket_desc *) 0;

/*
 * This routine initializes a bucket description page.
 */
/*
 * 下面的子程序用于初始化一页桶描述符页面。
 */
//// 初始化桶描述符。
// 建立空闲桶描述符链表，并让 free_bucket_desc 指向第一个空闲桶描述符。
static inline void init_bucket_desc()
{
        struct bucket_desc *bdesc, *first;
        int     i;

// 申请一页内存，用于存放桶描述符。如果失败，则显示初始化桶描述符时内存不够出错信息，死机。
        first = bdesc = (struct bucket_desc *) get_free_page();
```

```
103         if (!bdesc)
104                 panic("Out of memory in init_bucket_desc()");
    // 首先计算一页内存中可存放的桶描述符数量，然后对其建立单向连接指针。
105         for (i = PAGE_SIZE/sizeof(struct bucket_desc); i > 1; i--) {
106                 bdesc->next = bdesc+1;
107                 bdesc++;
108         }
109         /*
110          * This is done last, to avoid race conditions in case
111          * get_free_page() sleeps and this routine gets called again....
112          */
            /*
             * 这是在最后处理的，目的是为了避免在 get_free_page()睡眠时该子程序又被
             * 调用而引起的竞争条件。
             */
    // 将空闲桶描述符指针 free_bucket_desc 加入链表中。
113         bdesc->next = free_bucket_desc;
114         free_bucket_desc = first;
115 }
116
    //// 分配动态内存函数。
    // 参数：len - 请求的内存块长度。
    // 返回：指向被分配内存的指针。如果失败则返回 NULL。
117 void *malloc(unsigned int len)
118 {
119         struct _bucket_dir      *bdir;
120         struct bucket_desc      *bdesc;
121         void                    *retval;
122
123         /*
124          * First we search the bucket_dir to find the right bucket change
125          * for this request.
126          */
            /*
             * 首先我们搜索存储桶目录 bucket_dir 来寻找适合请求的桶大小。
             */
    // 搜索存储桶目录，寻找适合申请内存块大小的桶描述符链表。如果目录项的桶字节数大于请求的字节
    // 数，就找到了对应的桶目录项。
127         for (bdir = bucket_dir; bdir->size; bdir++)
128                 if (bdir->size >= len)
129                         break;
    // 如果搜索完整个目录都没有找到合适大小的目录项，则表明所请求的内存块大小太大，超出了该
    // 程序的分配限制(最长为 1 个页面)。于是显示出错信息，死机。
130         if (!bdir->size) {
131                 printk("malloc called with impossibly large argument (%d)\n",
132                         len);
133                 panic("malloc: bad arg");
134         }
135         /*
136          * Now we search for a bucket descriptor which has free space
137          */
            /*
             * 现在我们来搜索具有空闲空间的桶描述符。
             */
```

```
         */
138      cli(); /* Avoid race conditions */  /* 为了避免出现竞争条件，首先关中断 */
    // 搜索对应桶目录项中描述符链表，查找具有空闲空间的桶描述符。如果桶描述符的空闲内存指针
    // freeptr 不为空，则表示找到了相应的桶描述符。
139      for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next)
140              if (bdesc->freeptr)
141                      break;
142      /*
143       * If we didn't find a bucket with free space, then we'll
144       * allocate a new one.
145       */
         /*
          * 如果没有找到具有空闲空间的桶描述符，那么我们就要新建立一个该目录项的描述符。
          */
146      if (!bdesc) {
147              char            *cp;
148              int             i;
149
    // 若 free_bucket_desc 还为空时，表示第一次调用该程序，或者链表中所有空桶描述符都已用完。
    // 此时就需要申请一个页面并在其上建立并初始化空闲描述符链表。free_bucket_desc 会指向第一
    // 个空闲桶描述符。
150              if (!free_bucket_desc)
151                      init_bucket_desc();
    // 取 free_bucket_desc 指向的空闲桶描述符，并让 free_bucket_desc 指向下一个空闲桶描述符。
152              bdesc = free_bucket_desc;
153              free_bucket_desc = bdesc->next;
    // 初始化该新的桶描述符。令其引用数量等于 0；桶的大小等于对应桶目录的大小；申请一内存页面，
    // 让描述符的页面指针 page 指向该页面；空闲内存指针也指向该页开头，因为此时全为空闲。
154              bdesc->refcnt = 0;
155              bdesc->bucket_size = bdir->size;
156              bdesc->page = bdesc->freeptr = (void *) cp = get_free_page();
    // 如果申请内存页面操作失败，则显示出错信息，死机。
157              if (!cp)
158                      panic("Out of memory in kernel malloc()");
159              /* Set up the chain of free objects */
                 /* 在该页空闲内存中建立空闲对象链表 */
    // 以该桶目录项指定的桶大小为对象长度，对该页内存进行划分，并使每个对象的开始 4 字节设置
    // 成指向下一对象的指针。
160              for (i=PAGE_SIZE/bdir->size; i > 1; i--) {
161                      *((char **) cp) = cp + bdir->size;
162                      cp += bdir->size;
163              }
    // 最后一个对象开始处的指针设置为 0(NULL)。
    // 然后让该桶描述符的下一描述符指针字段指向对应桶目录项指针 chain 所指的描述符，而桶目录的
    // chain 指向该桶描述符，也即将该描述符插入到描述符链链头处。
164              *((char **) cp) = 0;
165              bdesc->next = bdir->chain; /* OK, link it in! */  /* OK，将其链入！*/
166              bdir->chain = bdesc;
167      }
    // 返回指针即等于该描述符对应页面的当前空闲指针。然后调整该空闲空间指针指向下一个空闲对象，
    // 并使描述符中对应页面中对象引用计数增 1。
168      retval = (void *) bdesc->freeptr;
169      bdesc->freeptr = *((void **) retval);
```

```
170         bdesc->refcnt++;
    // 最后开放中断，并返回指向空闲内存对象的指针。
171         sti();  /* OK, we're safe again */   /* OK，现在我们又安全了*/
172         return(retval);
173 }
174
175 /*
176  * Here is the free routine.  If you know the size of the object that you
177  * are freeing, then free_s() will use that information to speed up the
178  * search for the bucket descriptor.
179  *
180  * We will #define a macro so that "free(x)" is becomes "free_s(x, 0)"
181  */
    /*
     * 下面是释放子程序。如果你知道释放对象的大小，则 free_s()将使用该信息加速
     * 搜寻对应桶描述符的速度。
     *
     * 我们将定义一个宏，使得"free(x)"成为"free_s(x, 0)"。
     */
    //// 释放存储桶对象。
    // 参数：obj - 对应对象指针；size - 大小。
182 void free_s(void *obj, int size)
183 {
184         void            *page;
185         struct _bucket_dir      *bdir;
186         struct bucket_desc      *bdesc, *prev;
187
188         /* Calculate what page this object lives in */
            /* 计算该对象所在的页面 */
189         page = (void *)  ((unsigned long) obj & 0xfffff000);
190         /* Now search the buckets looking for that page */
            /* 现在搜索存储桶目录项所链接的桶描述符，寻找该页面 */
    //
191         for (bdir = bucket_dir; bdir->size; bdir++) {
192                 prev = 0;
193                 /* If size is zero then this conditional is always false */
                    /* 如果参数 size 是 0，则下面条件肯定是 false */
194                 if (bdir->size < size)
195                         continue;
    // 搜索对应目录项中链接的所有描述符，查找对应页面。如果某描述符页面指针等于 page 则表示找到
    // 了相应的描述符，跳转到 found。如果描述符不含有对应 page，则让描述符指针 prev 指向该描述符。
196                 for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next) {
197                         if (bdesc->page == page)
198                                 goto found;
199                         prev = bdesc;
200                 }
201         }
    // 若搜索了对应目录项的所有描述符都没有找到指定的页面，则显示出错信息，死机。
202         panic("Bad address passed to kernel free_s()");
203 found:
    // 找到对应的桶描述符后，首先关中断。然后将该对象内存块链入空闲块对象链表中，并使该描述符
    // 的对象引用计数减 1。
204         cli(); /* To avoid race conditions */   /* 为了避免竞争条件 */
```

```
205         *((void **)obj) = bdesc->freeptr;
206         bdesc->freeptr = obj;
207         bdesc->refcnt--;
    // 如果引用计数已等于 0，则我们就可以释放对应的内存页面和该桶描述符。
208         if (bdesc->refcnt == 0) {
209                 /*
210                  * We need to make sure that prev is still accurate.  It
211                  * may not be, if someone rudely interrupted us....
212                  */
                /*
                 * 我们需要确信 prev 仍然是正确的，若某程序粗鲁地中断了我们
                 * 就有可能不是了。
                 */
    // 如果 prev 已经不是搜索到的描述符的前一个描述符，则重新搜索当前描述符的前一个描述符。
213                 if ((prev && (prev->next != bdesc)) ||
214                     (!prev && (bdir->chain != bdesc)))
215                         for (prev = bdir->chain; prev; prev = prev->next)
216                                 if (prev->next == bdesc)
217                                         break;
    // 如果找到该前一个描述符，则从描述符链中删除当前描述符。
218                 if (prev)
219                         prev->next = bdesc->next;
    // 如果 prev==NULL，则说明当前一个描述符是该目录项首个描述符，也即目录项中 chain 应该直接
    // 指向当前描述符 bdesc，否则表示链表有问题，则显示出错信息，死机。因此，为了将当前描述符
    // 从链表中删除，应该让 chain 指向下一个描述符。
220                 else {
221                         if (bdir->chain != bdesc)
222                                 panic("malloc bucket chains corrupted");
223                         bdir->chain = bdesc->next;
224                 }
    // 释放当前描述符所操作的内存页面，并将该描述符插入空闲描述符链表开始处。
225                 free_page((unsigned long) bdesc->page);
226                 bdesc->next = free_bucket_desc;
227                 free_bucket_desc = bdesc;
228         }
    // 开中断，返回。
229         sti();
230         return;
231 }
232
233
```

15.8 程序 15-8 linux/lib/open.c

```
 1 /*
 2  *  linux/lib/open.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #define __LIBRARY__
 8 #include <unistd.h>        // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                              // 如定义了__LIBRARY__，则还含系统调用号和内嵌汇编_syscall0()等。
 9 #include <stdarg.h>        // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了-个
                              // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于
                              // vsprintf、vprintf、vfprintf 函数。
10
   //// 打开文件函数。
   // 打开并有可能创建一个文件。
   // 参数：filename - 文件名；flag - 文件打开标志；...
   // 返回：文件描述符，若出错则置出错码，并返回-1。
   // 第 13 行定义了一个寄存器变量 res，该变量将被保存在一个寄存器中，以便于高效访问和操作。
   // 若想指定存放的寄存器（例如 eax），那么可以把该句写成“register int res asm("ax");”。
11 int open(const char * filename, int flag, ...)
12 {
13         register int res;
14         va_list arg;
15
   // 利用 va_start()宏函数，取得 flag 后面参数的指针，然后调用系统中断 int 0x80，功能 open 进行
   // 文件打开操作。
   // %0 - eax(返回的描述符或出错码)；%1 - eax(系统中断调用功能号__NR_open)；
   // %2 - ebx(文件名 filename)；%3 - ecx(打开文件标志 flag)；%4 - edx(后随参数文件属性 mode)。
16         va_start(arg,flag);
17         __asm__("int $0x80"
18                 :"=a" (res)
19                 :"" (__NR_open),"b" (filename),"c" (flag),
20                  "d" (va_arg(arg,int)));
   // 系统中断调用返回值大于或等于 0，表示是一个文件描述符，则直接返回之。
21         if (res>=0)
22                 return res;
   // 否则说明返回值小于 0，则代表一个出错码。设置该出错码并返回-1。
23         errno = -res;
24         return -1;
25 }
26
```

15.9 程序 15-9 linux/lib/setsid.c

```
 1 /*
 2  *  linux/lib/setsid.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #define __LIBRARY__
 8 #include <unistd.h>        // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                              // 如定义了__LIBRARY__，则还含系统调用号和内嵌汇编_syscall0()等。
 9
   //// 创建一个会话并设置进程组号。
   // 下面系统调用宏对应于函数：pid_t setsid()。
   // 返回：调用进程的会话标识符(session ID)。
10 _syscall0(pid_t,setsid)
11
```

15.10 程序 15-10 linux/lib/string.c

```
 1 /*
 2  *  linux/lib/string.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #ifndef __GNUC__              // 需要 GNU 的 C 编译器编译。
 8 #error I want gcc!
 9 #endif
10
11 #define extern
12 #define inline
13 #define __LIBRARY__
14 #include <string.h>
15
```

15.11 程序 15-11 linux/lib/wait.c

```
 1 /*
 2  *  linux/lib/wait.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #define __LIBRARY__
 8 #include <unistd.h>        // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                              // 如定义了__LIBRARY__，则还含系统调用号和内嵌汇编_syscall0()等。
 9 #include <sys/wait.h>      // 等待调用头文件。定义系统调用 wait()和 waitpid()及相关常数符号。
10
   //// 等待进程终止系统调用函数。
   // 该下面宏结构对应于函数：pid_t waitpid(pid_t pid, int * wait_stat, int options)
   //
   // 参数：pid - 等待被终止进程的进程 id，或者是用于指定特殊情况的其他特定数值；
   //       wait_stat - 用于存放状态信息；options - WNOHANG 或 WUNTRACED 或是 0。
11 _syscall3(pid_t,waitpid,pid_t,pid,int *,wait_stat,int,options)
12
   //// wait()系统调用。直接调用 waitpid()函数。
13 pid_t wait(int * wait_stat)
14 {
15         return waitpid(-1,wait_stat,0);
16 }
17
```

15.12 程序 15-12 linux/lib/write.c

```
 1 /*
 2  *  linux/lib/write.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 #define __LIBRARY__
 8 #include <unistd.h>        // Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。
                              // 如定义了__LIBRARY__，则还含系统调用号和内嵌汇编_syscall0()等。
 9
   //// 写文件系统调用函数。
   // 该宏结构对应于函数：int write(int fd, const char * buf, off_t count)
   // 参数：fd - 文件描述符；buf - 写缓冲区指针；count - 写字节数。
   // 返回：成功时返回写入的字节数(0 表示写入 0 字节)；出错时将返回-1，并且设置了出错号。
10 _syscall3(int,write,int,fd,const char *,buf,off_t,count)
11
```

第16章 内核创建组合程序

16.1 程序 16-1 linux/tools/build.c

```
 1 /*
 2  *  linux/tools/build.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * This file builds a disk-image from three different files:
 9  *
10  * - bootsect: max 510 bytes of 8086 machine code, loads the rest
11  * - setup: max 4 sectors of 8086 machine code, sets up system parm
12  * - system: 80386 code for actual system
13  *
14  * It does some checking that all files are of the correct type, and
15  * just writes the result to stdout, removing headers and padding to
16  * the right amount. It also writes some system data to stderr.
17  */
    /*
     * 该程序从三个不同的程序中创建磁盘映像文件：
     *
     * - bootsect：该文件的 8086 机器码最长为 510 字节，用于加载其他程序。
     * - setup：该文件的 8086 机器码最长为 4 个磁盘扇区，用于设置系统参数。
     * - system：实际系统的 80386 代码。
     *
     * 该程序首先检查所有程序模块的类型是否正确，并将检查结果在终端上显示出来，
     * 然后删除模块头部并扩充大正确的长度。该程序也会将一些系统数据写到 stderr。
     */
18
19 /*
20  * Changes by tytso to allow root device specification
21  *
22  * Added swap-device specification: Linux 20.12.91
23  */
    /*
     * tytso 对该程序作了修改，以允许指定根文件设备。
     *
     * 添加了指定交换设备功能：Linus 20.12.91
     */
24
25 #include <stdio.h>      /* fprintf */              // 使用其中的 fprintf()函数。
26 #include <string.h>                                // 字符串操作函数。
27 #include <stdlib.h>     /* contains exit */        // 含 exit 函数原型说明。
28 #include <sys/types.h>  /* unistd.h needs this */  // 该头文件供 unistd.h 文件使用。
29 #include <sys/stat.h>                              // 含文件状态信息结构定义。
30 #include <linux/fs.h>                              // 文件系统头文件。
31 #include <unistd.h>     /* contains read/write */  // 含 read/write 函数原型说明。
```

```
32 #include <fcntl.h>                                  // 包含文件操作模式符号常数。
33
34 #define MINIX_HEADER 32              // minix 二进制目标文件模块头部长度为 32 字节。
35 #define GCC_HEADER 1024              // GCC 头部信息长度为 1024 字节。
36
37 #define SYS_SIZE 0x3000              // system 文件最长节数(字节数为 SYS_SIZE*16=128KB)。
38
   // 默认地把 Linux 根文件系统所在设备设置为在第 2 个硬盘的第 1 个分区上（即设备号为 0x0306），
   // 是因为 Linus 当时开发 Linux 时，把第 1 个硬盘用作 MINIX 系统盘，而第 2 个硬盘用作为 Linux
   // 的根文件系统盘。
39 #define DEFAULT_MAJOR_ROOT 3         // 默认根设备主设备号 - 3（硬盘）。
40 #define DEFAULT_MINOR_ROOT 6         // 默认根设备次设备号 - 6（第 2 个硬盘的第 1 分区）。
41
42 #define DEFAULT_MAJOR_SWAP 0         // 默认交换设备主设备号。
43 #define DEFAULT_MINOR_SWAP 0         // 默认交换设备次设备号。
44
45 /* max nr of sectors of setup: don't change unless you also change
46  * bootsect etc */
   /* 下面指定 setup 模块占的最大扇区数：不要改变该值，除非也改变 bootsect 等相应文件。
47 #define SETUP_SECTS 4                // setup 最大长度为 4 个扇区（2KB）。
48
49 #define STRINGIFY(x) #x              // 把 x 转换成字符串类型，用于出错显示语句中。
50
   //// 显示出错信息，并终止程序。
51 void die(char * str)
52 {
53         fprintf(stderr,"%s\n",str);
54         exit(1);
55 }
56
   // 显示程序使用方法，并退出。
57 void usage(void)
58 {
59         die("Usage: build bootsect setup system [rootdev] [> image]");
60 }
61
   // 主程序开始。
62 int main(int argc, char ** argv)
63 {
64         int i,c,id;
65         char buf[1024];
66         char major_root, minor_root;
67         char major_swap, minor_swap;
68         struct stat sb;
69
   // 首先检查 build 程序执行时实际命令行参数个数，并根据参数个数作相应设置。如果 build 程序
   // 命令行参数个数不是 4 到 6 个（程序名算作 1 个），则显示程序用法并退出。
70         if ((argc < 4) || (argc > 6))
71                 usage();
   // 若程序命令行上有多于 4 个参数，那么如果根设备名不是软盘（"FLOPPY"），则取该设备文件的
   // 状态信息。若取状态出错则显示信息并退出，否则取该设备名状态结构中的主设备号和次设备号
   // 作为根设备号。如果根设备就是 FLOPPY 设备，则让主设备号和次设备号取 0。表示根设备是当前
   // 启动引导设备。
```

```
72          if (argc > 4) {
73                  if (strcmp(argv[4], "FLOPPY")) {
74                          if (stat(argv[4], &sb)) {
75                                  perror(argv[4]);
76                                  die("Couldn't stat root device.");
77                          }
78                          major_root = MAJOR(sb.st_rdev);   // 取设备名状态结构中设备号。
79                          minor_root = MINOR(sb.st_rdev);
80                  } else {
81                          major_root = 0;
82                          minor_root = 0;
83                  }
    // 若参数只有 4 个，则让主设备号和次设备号等于系统默认的根设备号。
84          } else {
85                  major_root = DEFAULT_MAJOR_ROOT;
86                  minor_root = DEFAULT_MINOR_ROOT;
87          }
    // 若程序命令行上有 6 个参数，那么如果最后一个表示交换设备的参数不是无（"NONE"），则取该
    // 设备文件的状态信息。若取状态出错则显示信息并退出，否则取该设备名状态结构中的主设备号
    // 和次设备号作为交换设备号。如果最后一个参数就是"NONE"，则让交换设备的主设备号和次设备
    // 号取为 0。表示交换设备就是当前启动引导设备。
88          if (argc == 6) {
89                  if (strcmp(argv[5], "NONE")) {
90                          if (stat(argv[5], &sb)) {
91                                  perror(argv[5]);
92                                  die("Couldn't stat root device.");
93                          }
94                          major_swap = MAJOR(sb.st_rdev);   // 取设备名状态结构中设备号。
95                          minor_swap = MINOR(sb.st_rdev);
96                  } else {
97                          major_swap = 0;
98                          minor_swap = 0;
99                  }
    // 若参数没有 6 个而是 5 个，表示命令行上没有带交换设备名。于是就让交换设备主设备号和次设备
    // 号等于系统默认的交换设备号。
100         } else {
101                 major_swap = DEFAULT_MAJOR_SWAP;
102                 minor_swap = DEFAULT_MINOR_SWAP;
103         }

    // 接下来在标准错误终端上显示上面所选择的根设备主、次设备号和交换设备主、次设备号。如果
    // 主设备号不等于 2（软盘）或 3（硬盘），也不为 0（取系统默认设备），则显示出错信息并退出。
    // 终端的标准输出被定向到文件 Image，因此被用于输出保存内核代码数据，生成内核映像文件。
104         fprintf(stderr, "Root device is (%d, %d)\n", major_root, minor_root);
105         fprintf(stderr, "Swap device is (%d, %d)\n", major_swap, minor_swap);
106         if ((major_root != 2) && (major_root != 3) &&
107             (major_root != 0)) {
108                 fprintf(stderr, "Illegal root device (major = %d)\n",
109                         major_root);
110                 die("Bad root device --- major #");
111         }
112         if (major_swap && major_swap != 3) {
113                 fprintf(stderr, "Illegal swap device (major = %d)\n",
```

```
114                         major_swap);
115                 die("Bad root device --- major #");
116         }
    // 下面开始执行读取各个文件内容并进行相应的复制处理。首先初始化 1KB 的复制缓冲区，置全 0。
    // 然后以只读方式打开参数 1 指定的文件（bootsect）。从中读取 32 字节的 MINIX 执行文件头结构
    // 内容（参见列表后说明）到缓冲区 buf 中。
117         for (i=0;i<sizeof buf; i++) buf[i]=0;
118         if ((id=open(argv[1],O_RDONLY,0))<0)
119                 die("Unable to open 'boot'");
120         if (read(id,buf,MINIX_HEADER) != MINIX_HEADER)
121                 die("Unable to read header of 'boot'");
    // 接下来根据 MINIX 头部结构判断 bootsect 是否为一个有效的 MINIX 执行文件。若是，则从文件中
    // 读取 512 字节的引导扇区代码和数据。
    // 0x0301 - MINIX 头部 a_magic 魔数；0x10 - a_flag 可执行；0x04 - a_cpu, Intel 8086 机器码。
122         if (((long *) buf)[0]!=0x04100301)
123                 die("Non-Minix header of 'boot'");
    // 判断头部长度字段 a_hdrlen（字节）是否正确（32 字节）。（后三字节正好没有用，是 0）
124         if (((long *) buf)[1]!=MINIX_HEADER)
125                 die("Non-Minix header of 'boot'");
    // 判断数据段长 a_data 字段(long)内容是否为 0。
126         if (((long *) buf)[3]!=0)
127                 die("Illegal data segment in 'boot'");
    // 判断堆 a_bss 字段(long)内容是否为 0。
128         if (((long *) buf)[4]!=0)
129                 die("Illegal bss in 'boot'");
    // 判断执行点 a_entry 字段(long)内容是否为 0。
130         if (((long *) buf)[5] != 0)
131                 die("Non-Minix header of 'boot'");
    // 判断符号表长字段 a_sym 的内容是否为 0。
132         if (((long *) buf)[7] != 0)
133                 die("Illegal symbol table in 'boot'");
    // 在上述判断都正确的条件下读取文件中随后的实际代码数据，应该返回读取字节数为 512 字节。
    // 因为 bootsect 文件中包含的是 1 个扇区的引导扇区代码和数据，并且最后 2 字节应该是可引导
    // 标志 0xAA55。
134         i=read(id,buf,sizeof buf);
135         fprintf(stderr,"Boot sector %d bytes.\n",i);
136         if (i != 512)
137                 die("Boot block must be exactly 512 bytes");
138         if ((*(unsigned short *)(buf+510)) != 0xAA55)
139                 die("Boot block hasn't got boot flag (0xAA55)");
    // 引导扇区的 506、507 偏移处需存放交换设备号，508、509 偏移处需存放根设备号。
140         buf[506] = (char) minor_swap;
141         buf[507] = (char) major_swap;
142         buf[508] = (char) minor_root;
143         buf[509] = (char) major_root;
    // 然后将该 512 字节的数据写到标准输出 stdout，若写出字节数不对，则显示出错信息并退出。
    // 在 linux/Makefile 中，build 程序标准输出被重定向到内核映像文件名 Image 上，因此引导
    // 扇区代码和数据会被写到 Image 开始的 512 字节处。最后关闭 bootsect 模块文件。
144         i=write(1,buf,512);
145         if (i!=512)
146                 die("Write call failed");
147         close (id);
148
```

```
    // 下面以只读方式打开参数 2 指定的文件（setup）。从中读取 32 字节的 MINIX 执行文件头结构
    // 内容到缓冲区 buf 中。处理方式与上面相同。首先以只读方式打开指定的文件 setup。从中读
    // 取 32 字节的 MINIX 执行文件头结构内容到缓冲区 buf 中。
149         if ((id=open(argv[2],0_RDONLY,0))<0)
150                 die("Unable to open 'setup'");
151         if (read(id,buf,MINIX_HEADER) != MINIX_HEADER)
152                 die("Unable to read header of 'setup'");
    // 接下来根据 MINIX 头部结构判断 setup 是否为一个有效的 MINIX 执行文件。若是，则从文件中
    // 读取 512 字节的引导扇区代码和数据。
    // 0x0301- MINIX 头部 a_magic 魔数；0x10- a_flag 可执行；0x04- a_cpu, Intel 8086 机器码。
153         if (((long *) buf)[0]!=0x04100301)
154                 die("Non-Minix header of 'setup'");
    // 判断头部长度字段 a_hdrlen（字节）是否正确（32 字节）。（后三字节正好没有用，是 0）
155         if (((long *) buf)[1]!=MINIX_HEADER)
156                 die("Non-Minix header of 'setup'");
    // 判断数据段长字段 a_data、堆字段 a_bss、起始执行点字段 a_entry 和符号表字段 a_sym 的内容
    // 是否为 0。必须都为 0。
157         if (((long *) buf)[3]!=0)                       // 数据段长 a_data 字段。
158                 die("Illegal data segment in 'setup'");
159         if (((long *) buf)[4]!=0)                       // 堆 a_bss 字段。
160                 die("Illegal bss in 'setup'");
161         if (((long *) buf)[5] != 0)                     // 执行起始点 a_entry 字段。
162                 die("Non-Minix header of 'setup'");
163         if (((long *) buf)[7] != 0)
164                 die("Illegal symbol table in 'setup'");
    // 在上述判断都正确的条件下读取文件中随后的实际代码数据，并且写到终端标准输出。同时统计
    // 写的长度（i），并在操作结束后关闭 setup 文件。之后判断一下利用 setup 执行写操作的代码
    // 和数据长度值，该值不能大于(SETUP_SECTS * 512)字节，否则就得重新修改 build、bootsect
    // 和 setup 程序中设定的 setup 所占扇区数并重新编译内核。若一切正常就显示 setup 实际长度值。
165         for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
166                 if (write(1,buf,c)!=c)
167                         die("Write call failed");
168         close (id);                                     //关闭 setup 模块文件。
169         if (i > SETUP_SECTS*512)
170                 die("Setup exceeds " STRINGIFY(SETUP_SECTS)
171                         " sectors - rewrite build/boot/setup");
172         fprintf(stderr,"Setup is %d bytes.\n",i);
    // 在将缓冲区 buf 清零之后，判断实际写的 setup 长度与(SETUP_SECTS4*512)的数值差，若 setup
    // 长度小于该长度（4*512 字节），则用 NULL 字符将 setup 填足为 4*512 字节。
173         for (c=0 ; c<sizeof(buf) ; c++)
174                 buf[c] = '\0';
175         while (i<SETUP_SECTS*512) {
176                 c = SETUP_SECTS*512-i;
177                 if (c > sizeof(buf))
178                         c = sizeof(buf);
179                 if (write(1,buf,c) != c)
180                         die("Write call failed");
181                 i += c;
182         }
183
    // 下面开始处理 system 模块文件。该文件使用 gas 编译，因此具有 GNU a.out 目标文件格式。
    // 首先以只读方式打开文件，并读取其中 a.out 格式头部结构信息（1KB 长度）。在判断 system
    // 是一个有效的 a.out 格式文件之后，就把该文件随后的所有数据都写到标准输出（Image 文件）
```

```
    // 中，并关闭该文件。然后显示 system 模块的长度。若 system 代码和数据长度超过 SYS_SIZE 节
    // （即 128KB 字节），则显示出错信息并退出。若无错，则返回 0，表示正常退出。
184         if ((id=open(argv[3],O_RDONLY,0))<0)
185                 die("Unable to open 'system'");
186         if (read(id,buf,GCC_HEADER) != GCC_HEADER)
187                 die("Unable to read header of 'system'");
188         if (((long *) buf)[5] != 0)                 // 执行入口点字段 a_entry 值应为 0。
189                 die("Non-GCC header of 'system'");
190         for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
191                 if (write(1,buf,c)!=c)
192                         die("Write call failed");
193         close(id);
194         fprintf(stderr,"System is %d bytes.\n",i);
195         if (i > SYS_SIZE*16)
196                 die("System is too big");
197         return(0);
198 }
199
```